中山大學

Principles of Compiler Construction

Lecture 7 Syntax Analysis (III)

Lecturer: C H A N G H U I Y O U

Note that most of these slides were created by:

Prof. Wen-jun LI (School of Software)

语法分析方法的分类

Top-down

Bottom-up

Bottom-up Parsing

优点：可以分析更多的文法

缺点：实现较麻烦

Bottom-up Parsing

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

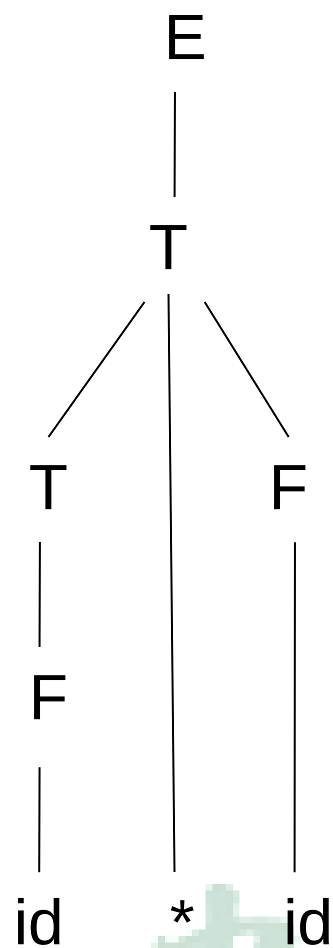
E→T

T→T*F

F→id

T→F

F→id



Reduction（归约）

归约: 用产生式的左边代替产生式的右边

自底向上的语法分析过程就是将只含有终端符号的输入串逐步归约到文法起始符号的过程

$$\mathbf{id} * \mathbf{id},\quad F * \mathbf{id},\quad T * \mathbf{id},\quad T * F,\quad T,\quad E$$

反过来，

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

所以上述归约过程实际上是最右推导的逆过程

Shift-Reduce（移进-归约）

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2\,\$$	shift
$\$\,T *$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

4类action: shift, reduce, accept, error

问题

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2\,\$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2\,\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2\,\$$	reduce by $T \rightarrow F$
$\$\,T$	$* \mathbf{id}_2\,\$$	shift
$\$\,T *$	$\mathbf{id}_2\,\$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$\,T$	$\$$	reduce by $E \rightarrow T$
$\$\,E$	$\$$	accept

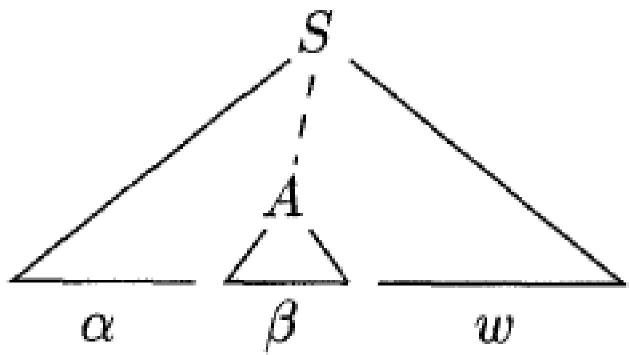
为什么是移进而不是将T归约为E？

Handle（句柄）

Right Sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Handle（句柄）

如果 $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$，则称句型 β 的句柄 $\alpha\beta w$.



注意：在上式中w是终端符号串，A是非终端符号，α和β中可以有终端符号和非终端符号.

直观的理解：一个句型的句柄就是这个句型的分析树中最左那棵只有父子两代的子树的所有叶子的从左到右的排列

LR Parsing

Proposed by

- D. Knuth (Stanford U.). On the Translation of Languages from Left to Right. Information and Control, 8(6), 1965, pp. 607-639
  - Prof. 高德纳: The Art of Computer Programming, T_EX, KMP Algorithm, LR Parsing, Attribute Grammar, etc.

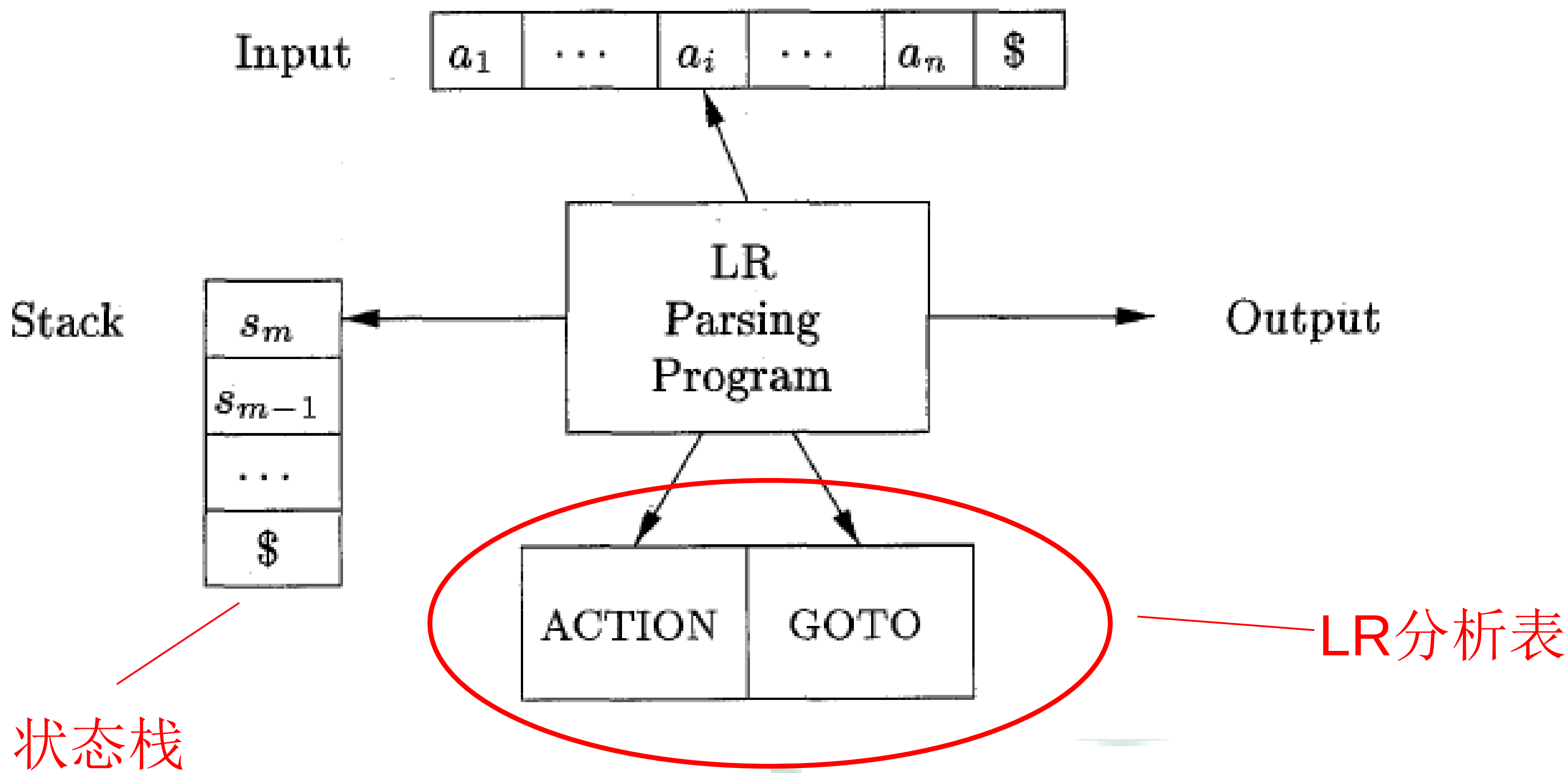
LR(k) parsing:

"L" : left-to-right scanning of the input

"R“: constructing a rightmost derivation in reverse

k: the number of input symbols of lookahead that are used in making parsing decisions, when (k) is omitted, k is assumed to be 1.

LR Parsing


Input
a_1
...
a_i
...
a_n
$
LR Parsing Program
Stack
s_m
s_{m-1}
...
$
Output
ACTION
GOTO
LR分析表
状态栈

Example

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

LR Parsing

不同的LR分析法具有相同的LR分析程序

不同之处在于用不同的方法构造出不同的ACTION表和GOTO表.

Simple LR(SLR) Parsing

最简单的LR分析

如何构造SLR的分析表?

LR(0) Items

An *LR(0)* item (item for short) a production with a dot at some position of the body.

Eg., $A \rightarrow XYZ$ yields the four items:

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow X Y \cdot Z$$

$$A \rightarrow X YZ \cdot$$

The production $A \rightarrow \epsilon$ generates only one item, $A \rightarrow \cdot$

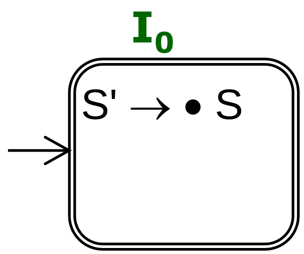
Augmented Grammar

If G is a grammar with start symbol S, then G', the *augmented grammar* for G, is G with a new start symbol S' and production S'→ S.

Ideas

Consider the following grammar

- $S' \rightarrow S$
- $S \rightarrow a\ A \mid b\ B$
- $A \rightarrow c\ A \mid d$
- $B \rightarrow c\ B \mid d$



Initial state

I_0

$S' \rightarrow \bullet\, S$
$S \rightarrow \bullet\, \mathbf{a}\, A$
$S \rightarrow \bullet\, \mathbf{b}\, B$

Equivalent closure

Kernel items

I_0

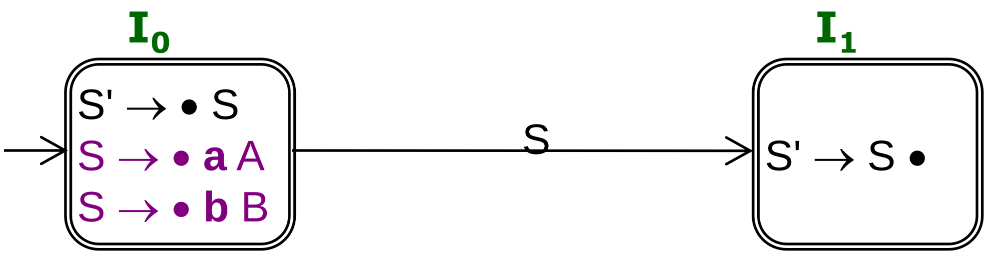
$S' \rightarrow \bullet S$

$S \rightarrow \bullet \mathbf{a} A$

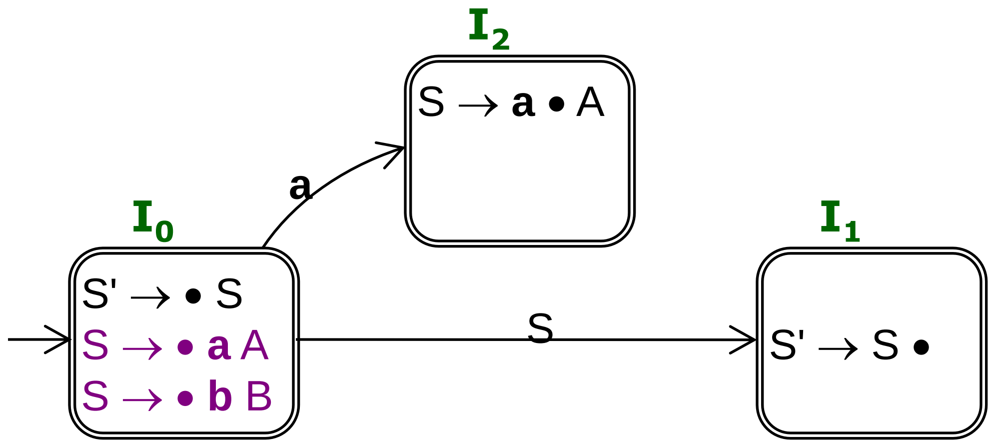
$S \rightarrow \bullet \mathbf{b} B$

Nonkernel items or closure items

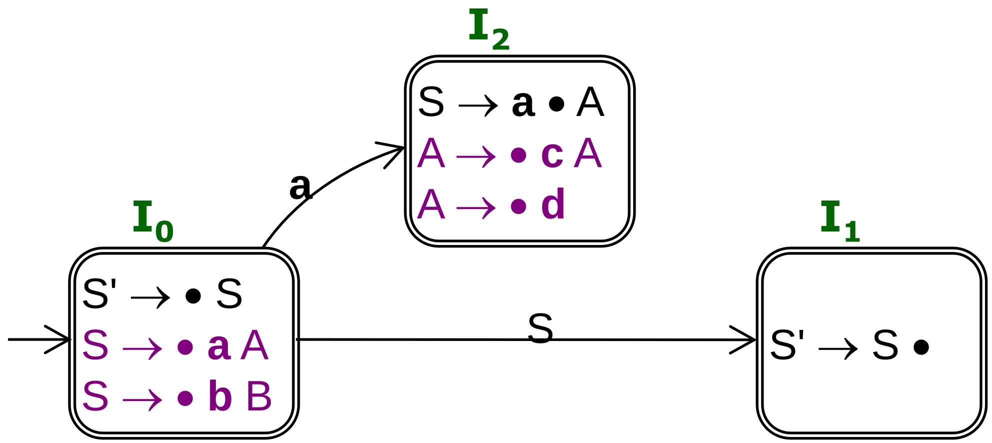
Equivalent closure



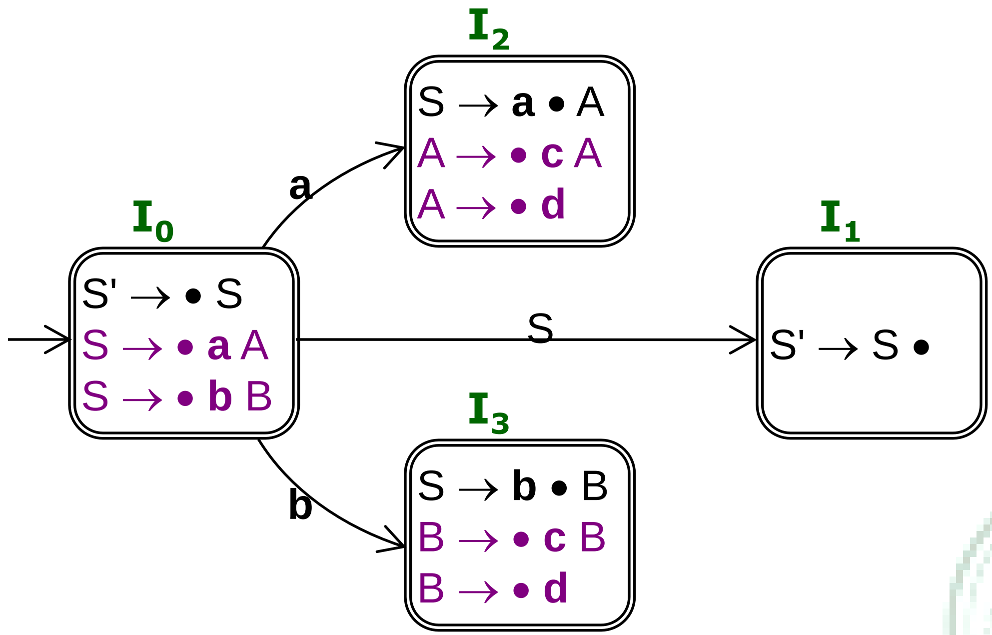
If the remaining string can be reduced to S (expected !)



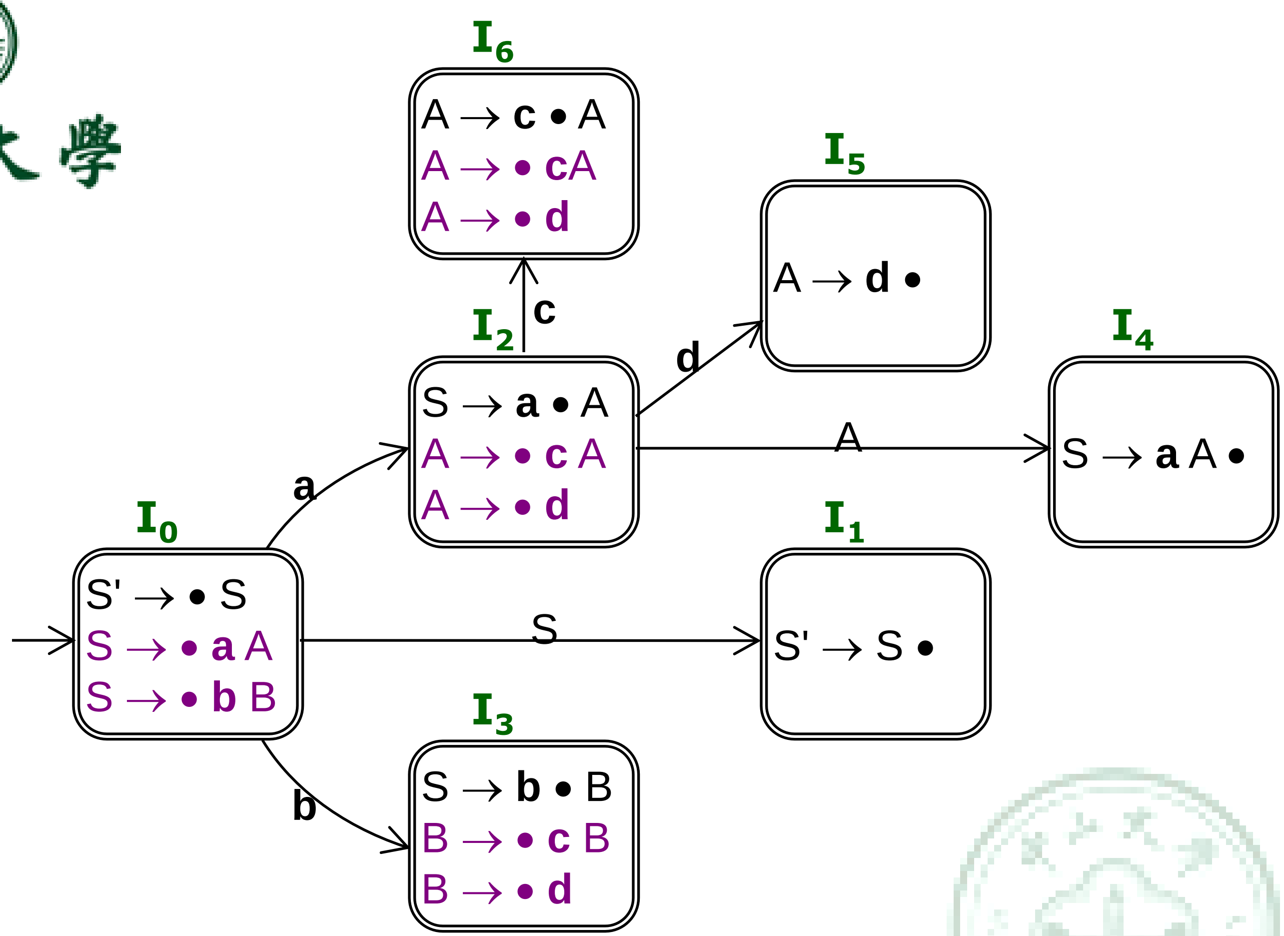
If the first symbol of remaining string is **a** (shift !)



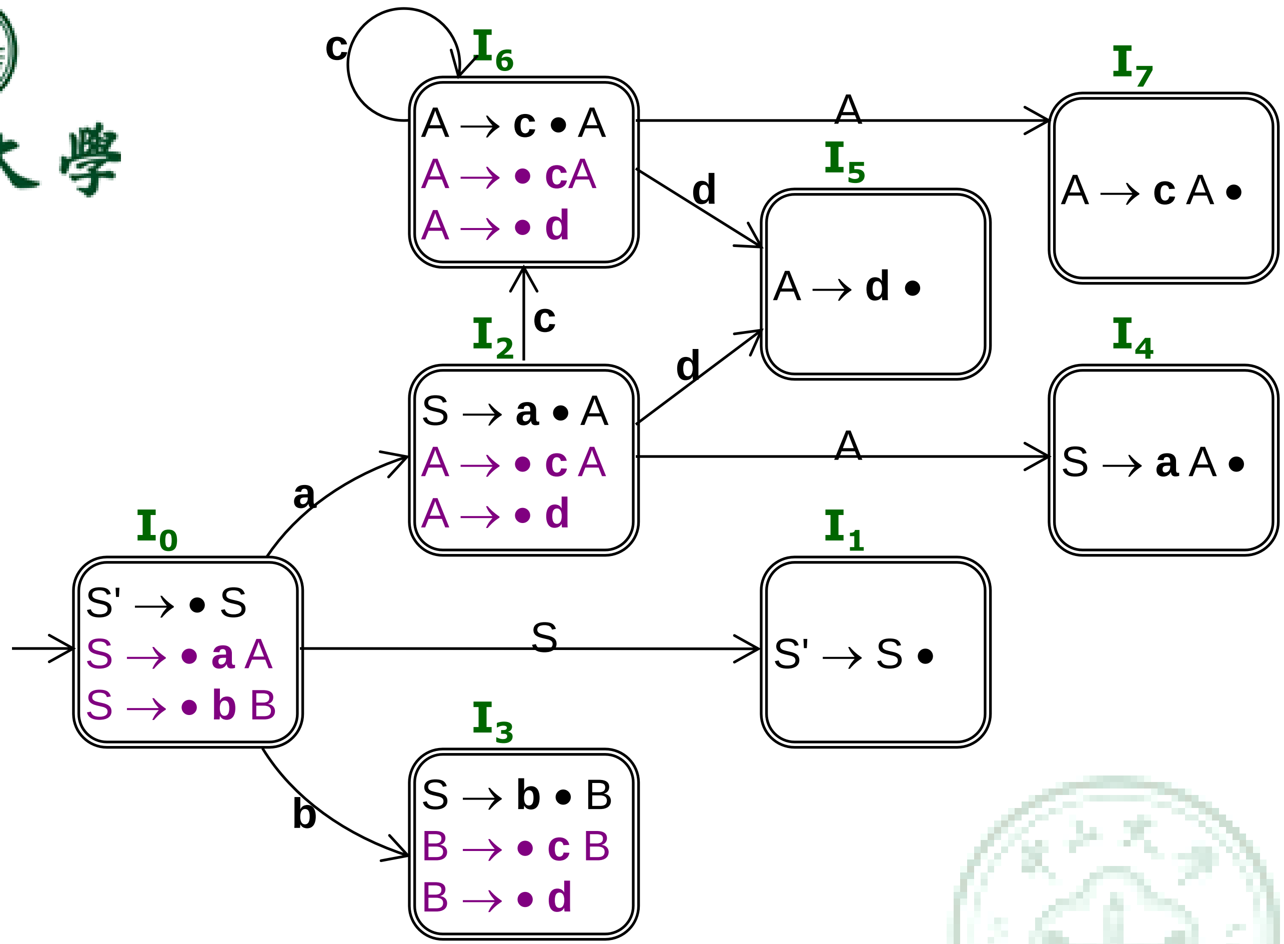
Equivalent closure in state I_2



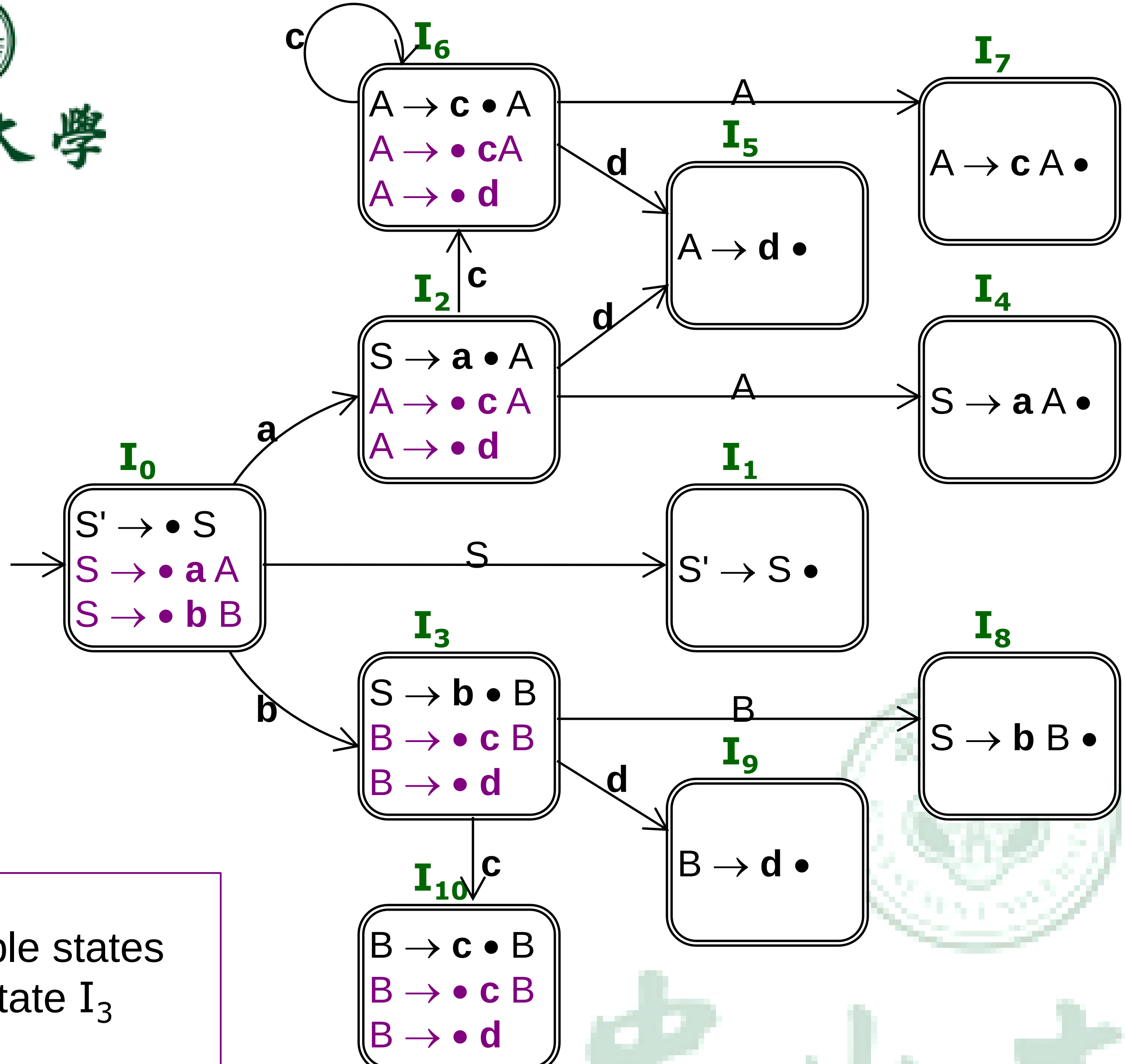
If the first symbol of remaining string is **b** (shift !)



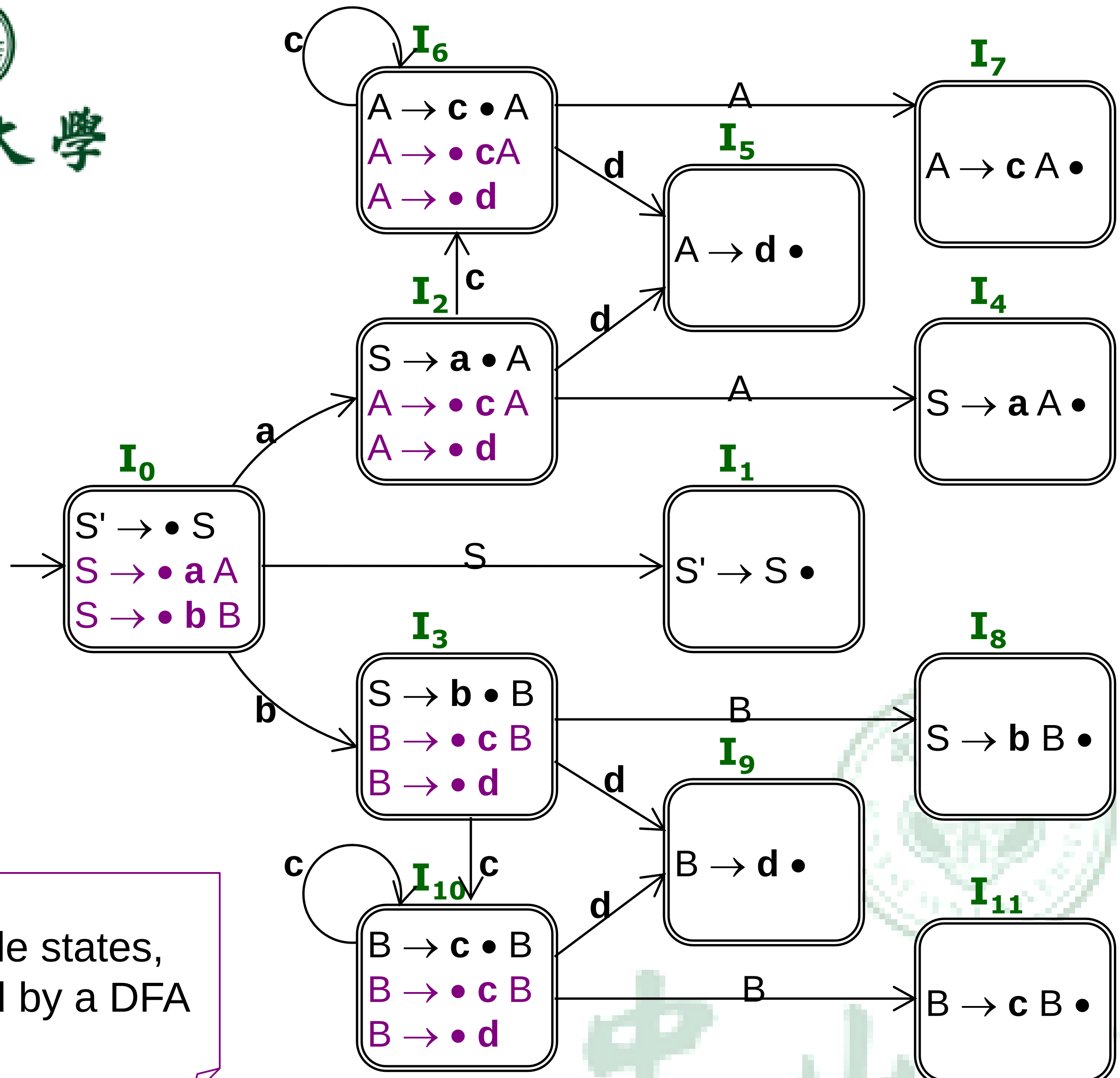
All possible states from state I_2



All possible states from state I_6



All possible states from state I_3

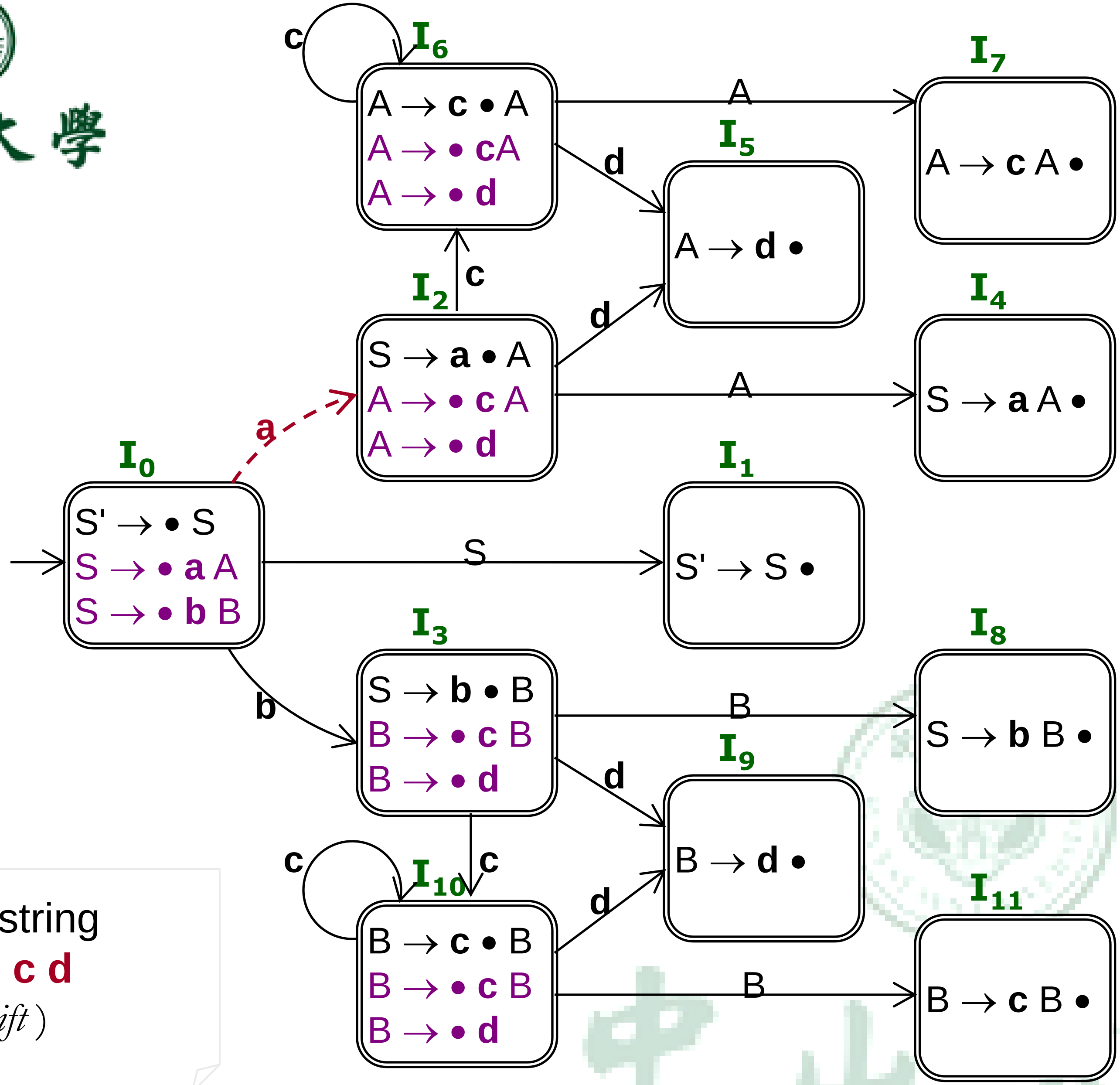


All possible states, recognized by a DFA

Working with the DFA

Consider the following sentence:

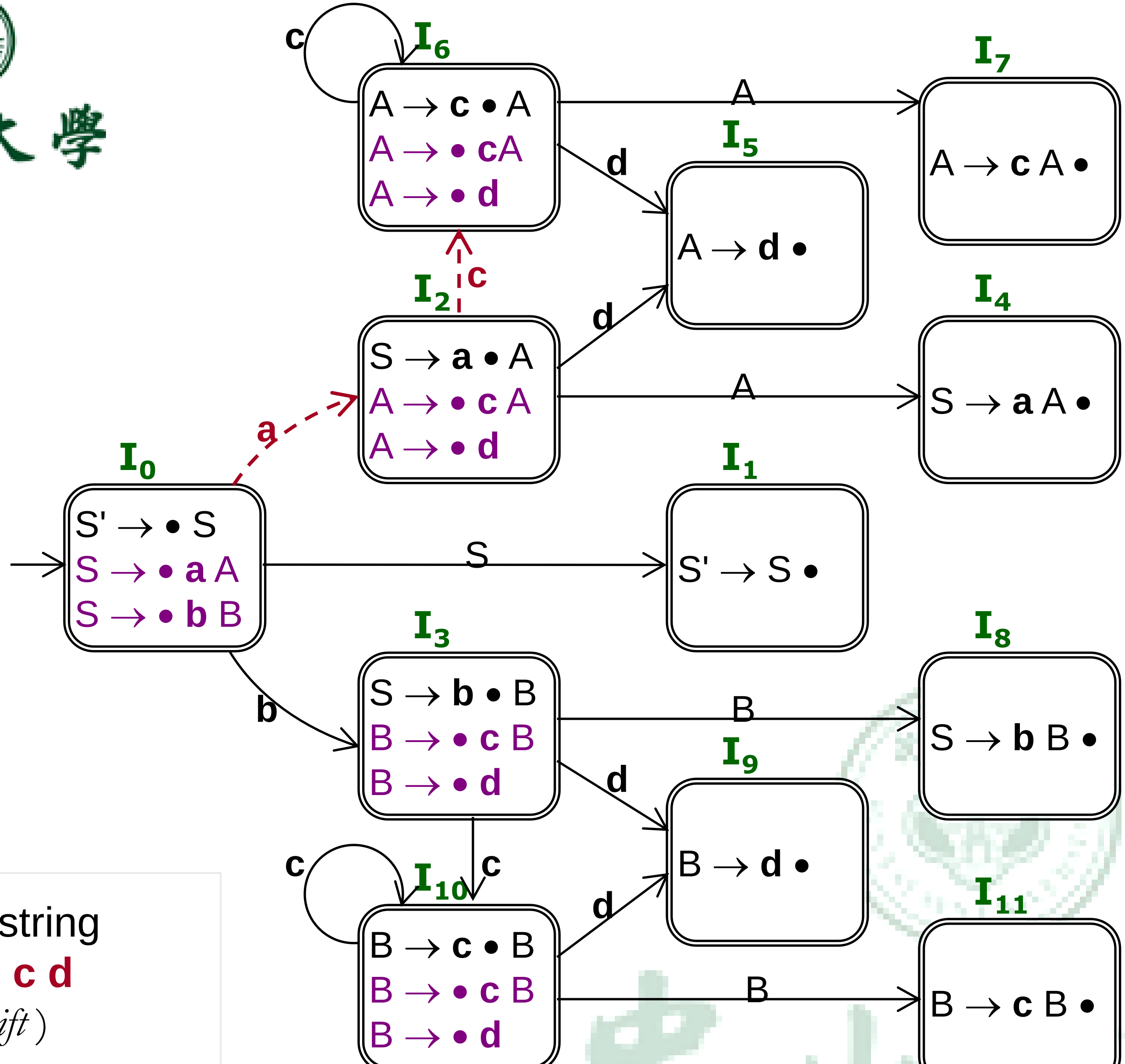
- a c c d
- We have the right-most derivation:
- $S' \Rightarrow S \Rightarrow aA \Rightarrow acA \Rightarrow accA \Rightarrow accd$



Input string

a | c c d

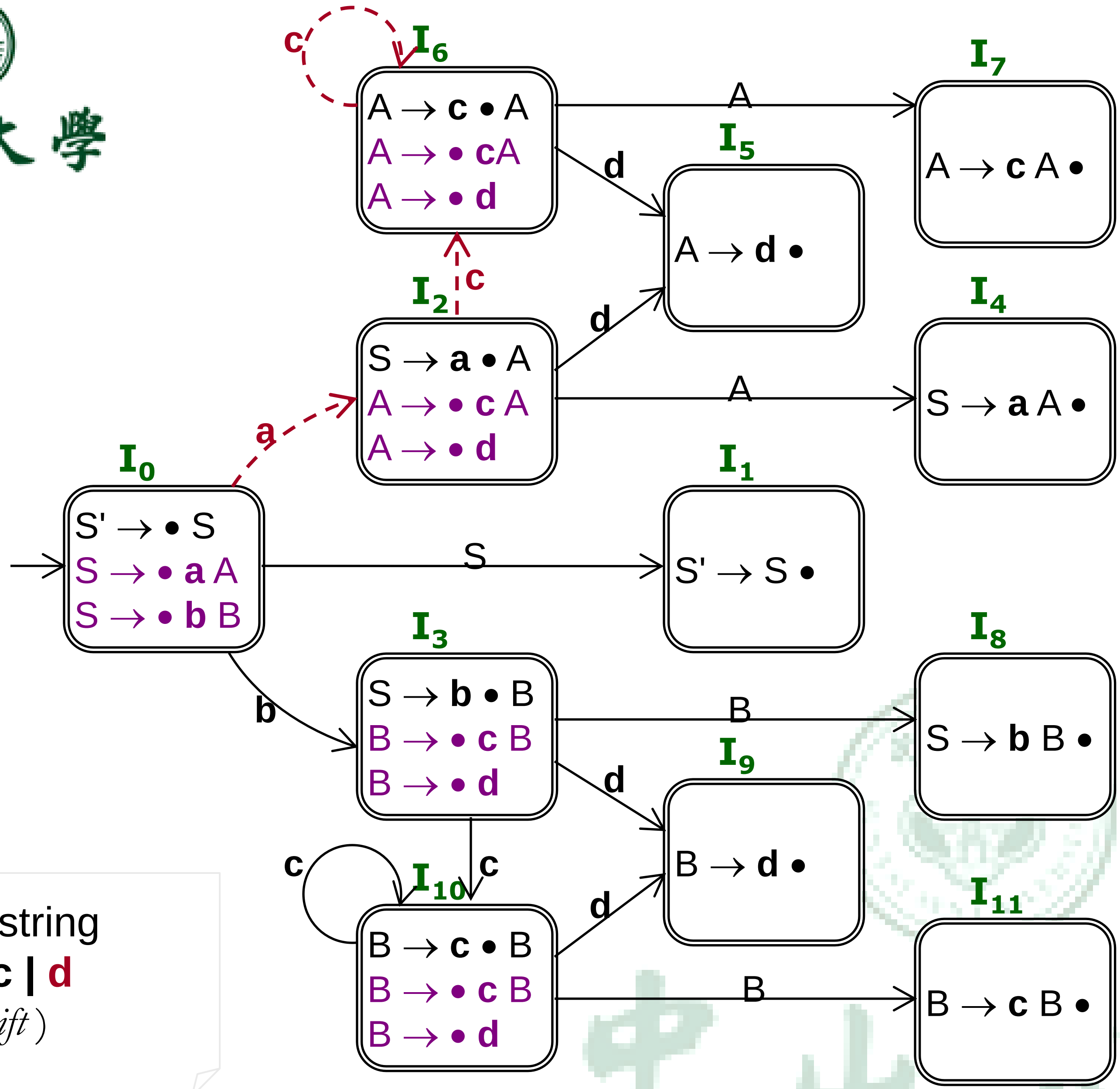
(*shift*)



Input string

a c | c d

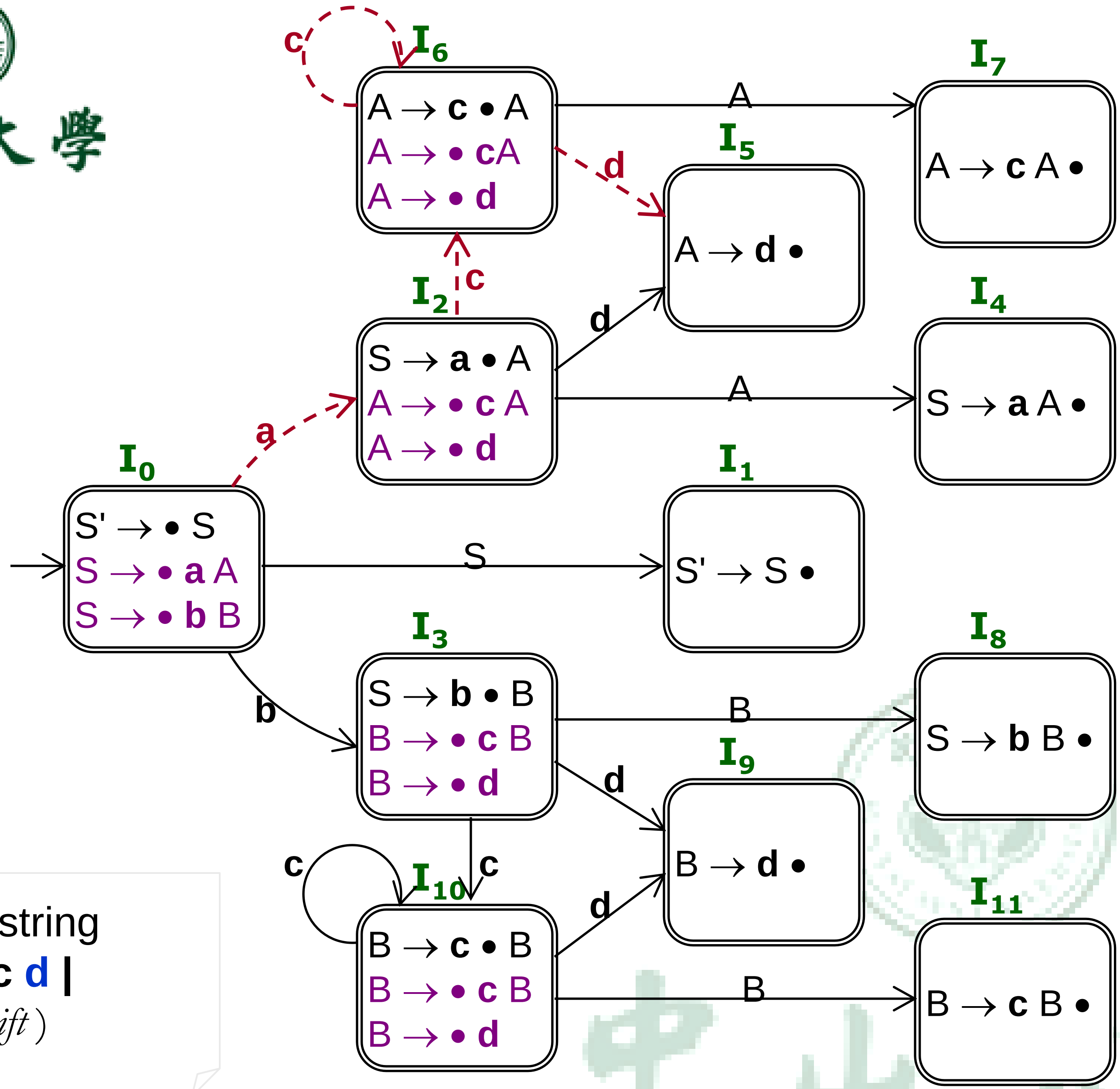
(*shift*)



Input string

a c c | d

(*shift*)

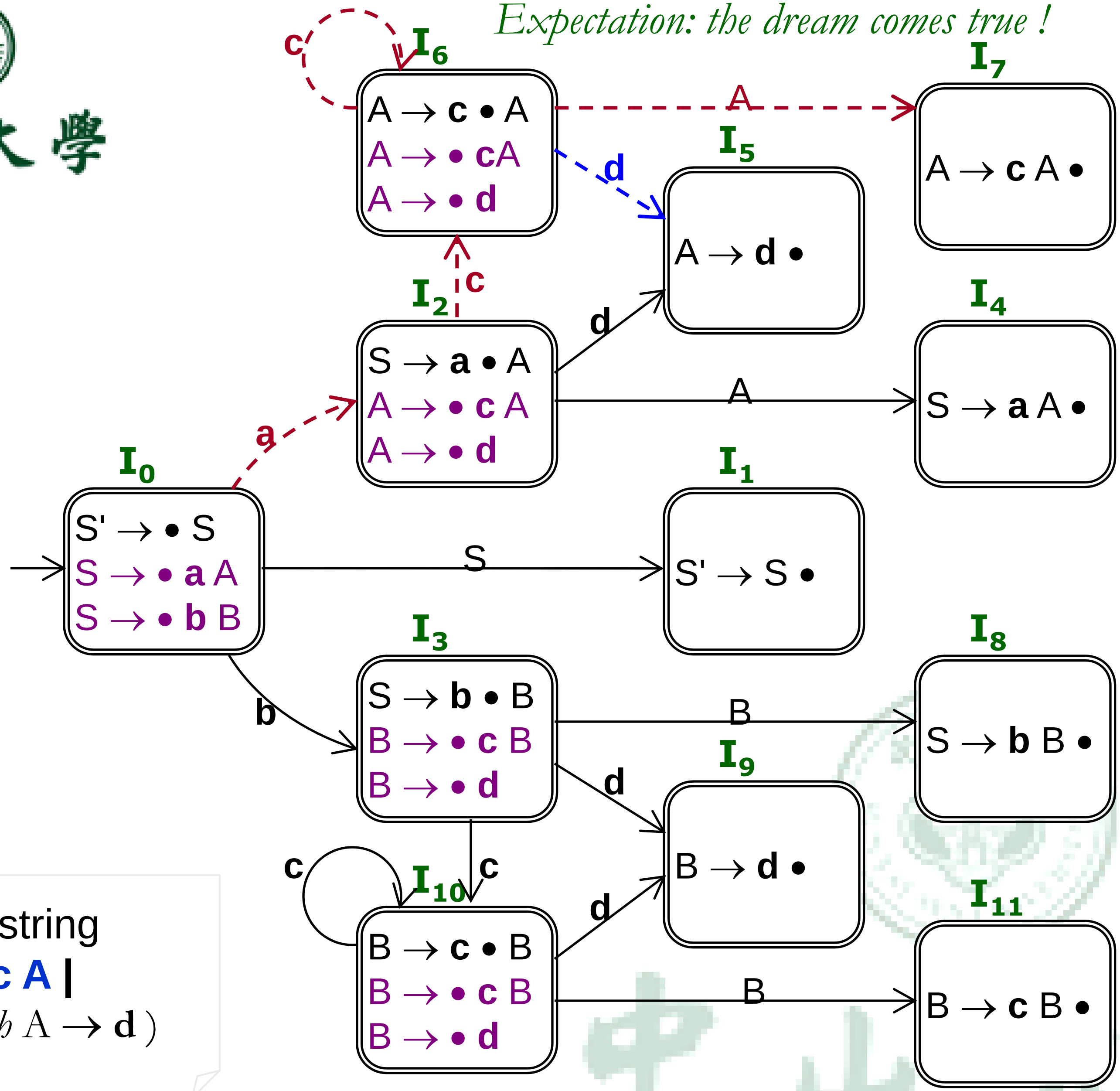


Input string

a c c d |

(*shift*)

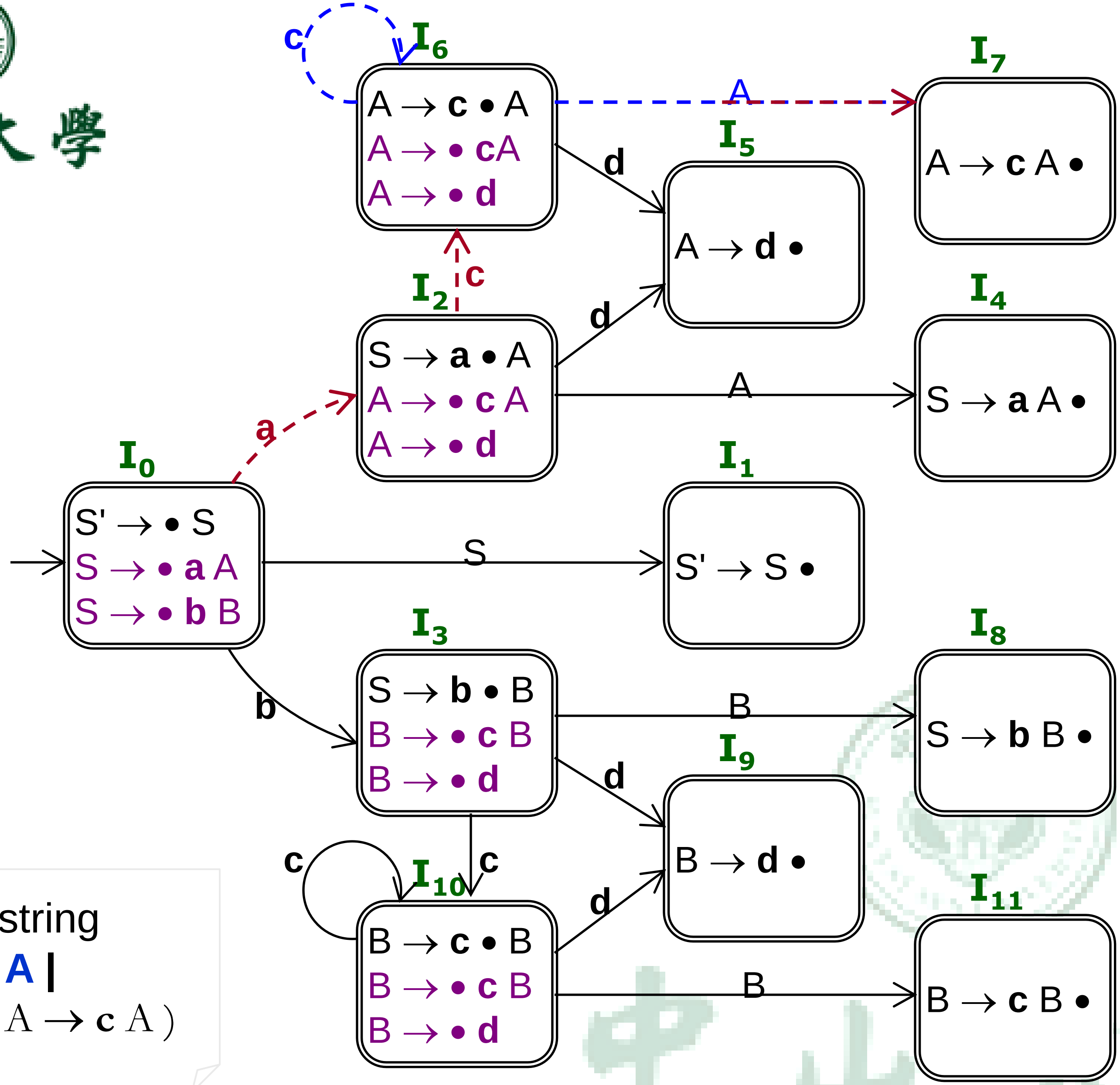
Expectation: the dream comes true !



Input string

a c c A |

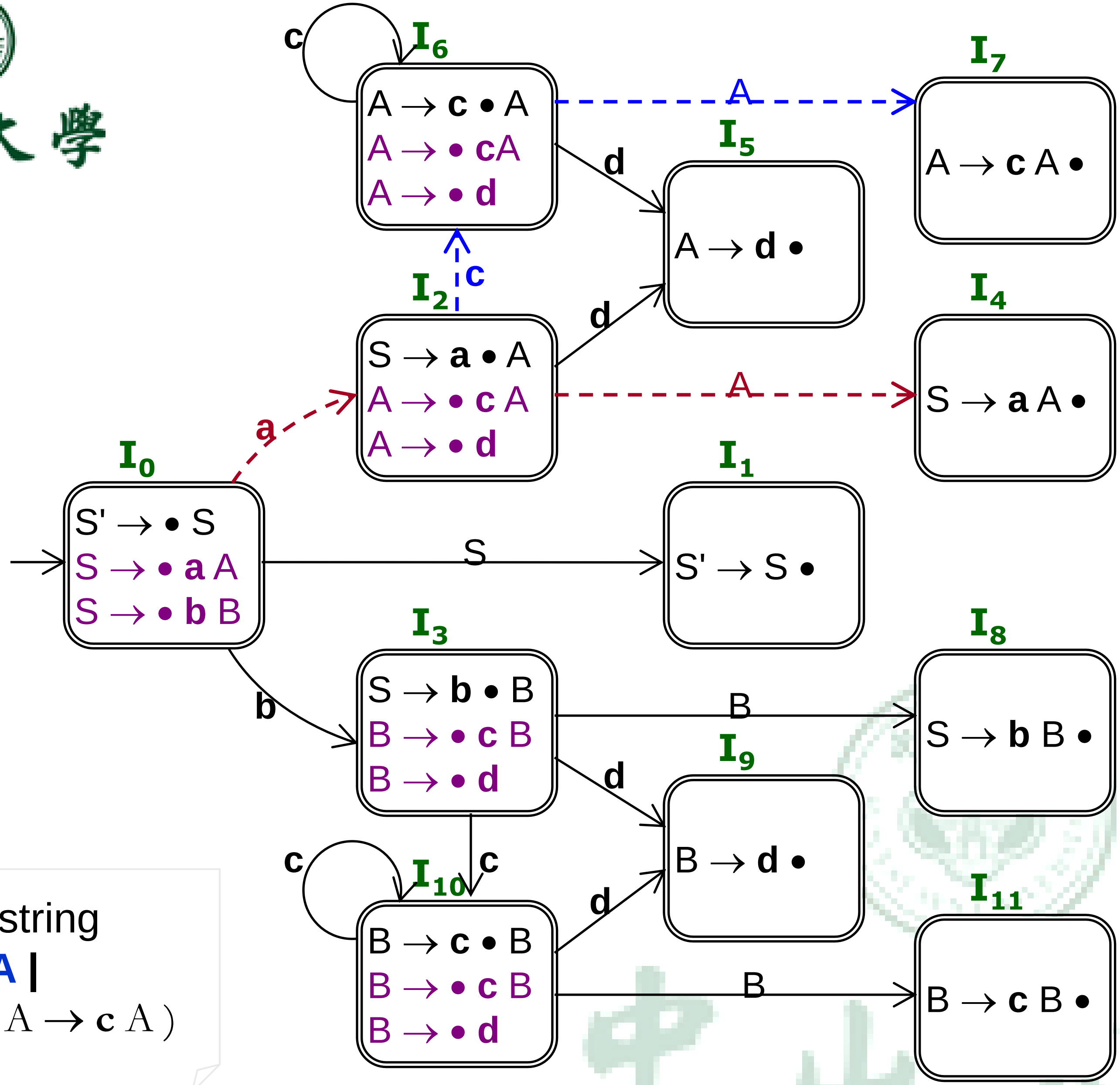
(*reduce with* $A \to \mathbf{d}$)



Input string

a **c** **A** **|**

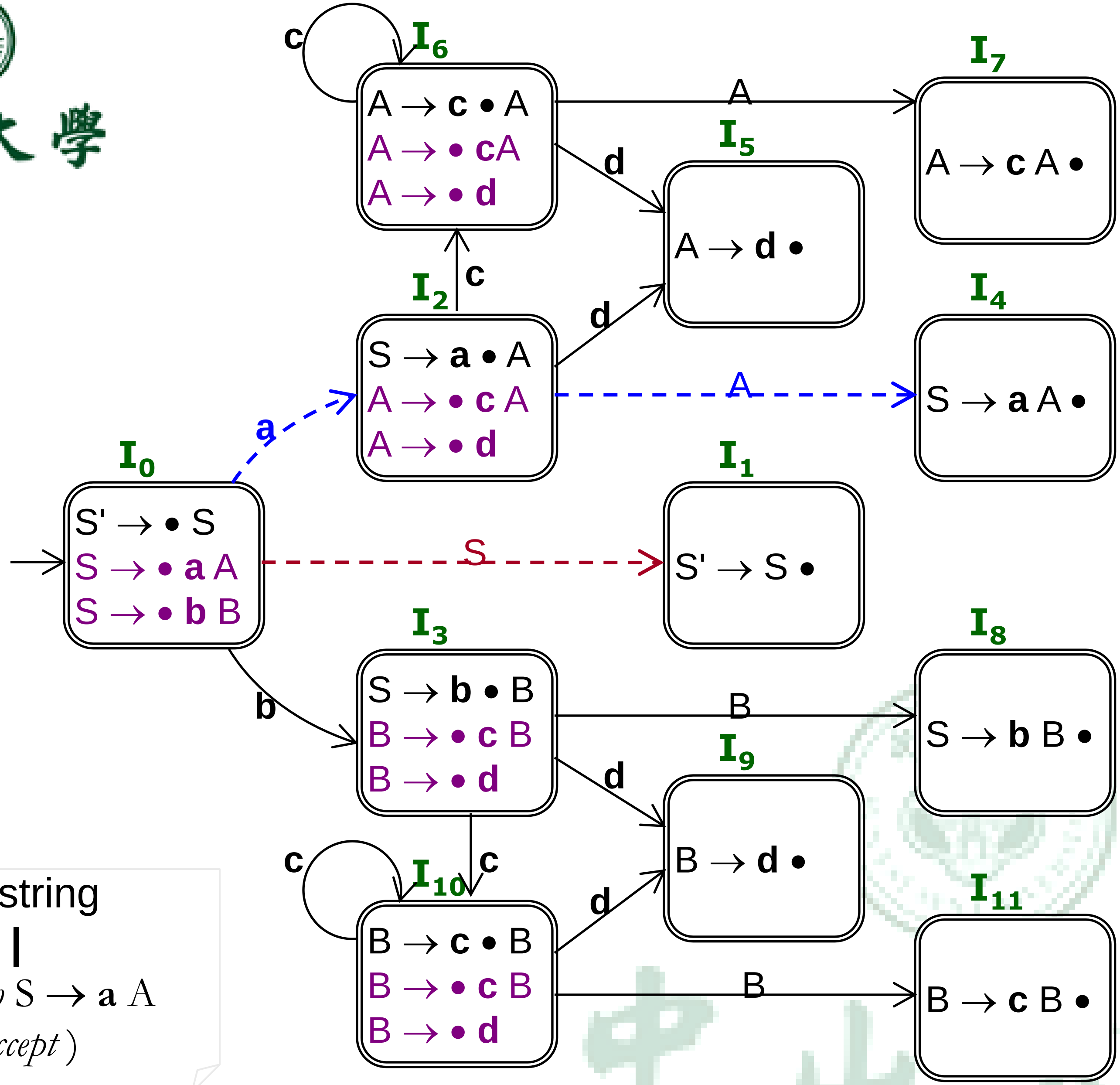
(*reduce with* A → **c** A)



Input string

a A |

(*reduce with* A → **c** A)



Input string

S |

(*reduce with* S → **a** A *and accept*)

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Example

对于扩充文法

$E' \rightarrow E$
$E \rightarrow E+T|T$
$T \rightarrow T*F|F$
$F \rightarrow (E)|id$

求CLOSURE({[$E' \rightarrow \cdot E$]})

The Function GOTO

The second useful function is GOTO(I, X) where I is a set of items and X is a grammar symbol. GOTO(I, X) is defined to be the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I.

Example 4.41: If I is the set of two items $\{[E' \to E\cdot],\ [E \to E \cdot + T]\}$, then GOTO$(I, +)$ contains the items

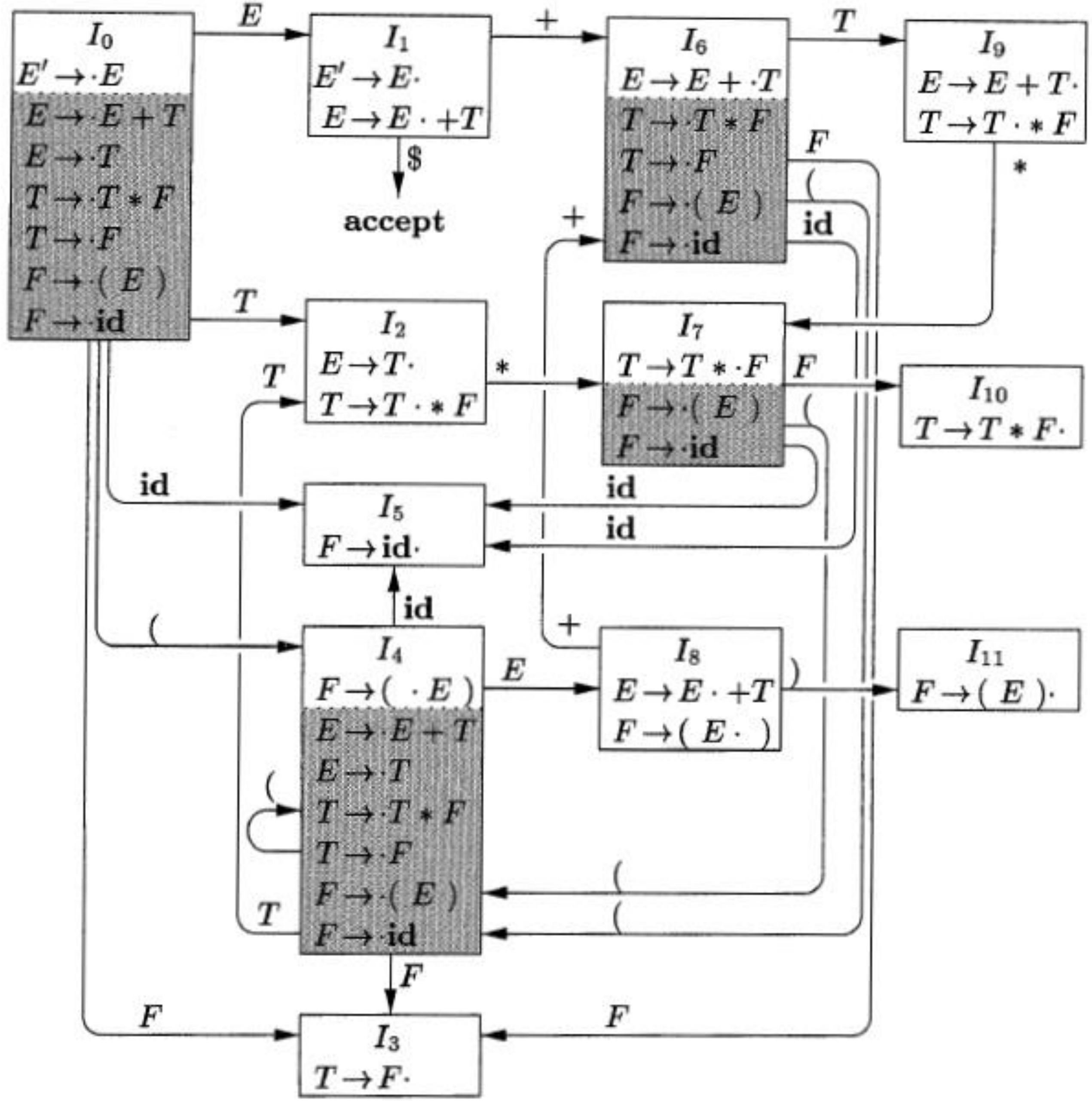
$$
\begin{array}{l}
E \to E + \cdot T \\
T \to \cdot T * F \\
T \to \cdot F \\
F \to \cdot (E) \\
F \to \cdot \mathbf{id}
\end{array}
$$

构造$LR(0)$项的规范集族

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

中山大學

LR(0) 自动机


I0
E' → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E · +T
accept
I2
E → T·
T → T · * F
I3
T → F·
I4
F → (· E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I7
T → T * ·F
F → ·(E)
F → ·id
I8
E → E · +T
F → (E ·)
I9
E → E + T·
T → T · * F
I10
T → T * F·
I11
F → (E)·

构造*SLR*分析表

INPUT: An augmented grammar G'.

OUTPUT: The SLR-parsing table functions ACTION and GOTO for G'.

METHOD:

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(0) items for G'.

2. State i is constructed from I_i. The parsing actions for state i are determined as follows:

   (a) If $[A \to \alpha \cdot a\beta]$ is in I_i and $\text{GOTO}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to "shift j." Here a must be a terminal.

   (b) If $[A \to \alpha \cdot]$ is in I_i, then set $\text{ACTION}[i, a]$ to "reduce $A \to \alpha$" for all a in $\text{FOLLOW}(A)$; here A may not be S'.

   (c) If $[S' \to S \cdot]$ is in I_i, then set $\text{ACTION}[i, \$]$ to "accept."

If any conflicting actions result from the above rules, we say the grammar is not SLR(1). The algorithm fails to produce a parser in this case.

构造SLR分析表

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If $\text{GOTO}(I_i, A) = I_j$, then $\text{GOTO}[i, A] = j$.

4. All entries not defined by rules (2) and (3) are made "error."

5. The initial state of the parser is the one constructed from the set of items containing $[S' \to \cdot S]$.

$\square$

More···

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$\mathbf{id} * \mathbf{id} + \mathbf{id}\,\$$	shift
(2)	0 5	$\mathbf{id}$	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(3)	0 3	F	$* \mathbf{id} + \mathbf{id}\,\$$	reduce by $T \to F$
(4)	0 2	T	$* \mathbf{id} + \mathbf{id}\,\$$	shift
(5)	0 2 7	$T *$	$\mathbf{id} + \mathbf{id}\,\$$	shift
(6)	0 2 7 5	$T * \mathbf{id}$	$+ \mathbf{id}\,\$$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	$T * F$	$+ \mathbf{id}\,\$$	reduce by $T \to T * F$
(8)	0 2	T	$+ \mathbf{id}\,\$$	reduce by $E \to T$
(9)	0 1	E	$+ \mathbf{id}\,\$$	shift
(10)	0 1 6	$E +$	$\mathbf{id}\,\$$	shift
(11)	0 1 6 5	$E + \mathbf{id}$	$\$$	reduce by $F \to \mathbf{id}$
(12)	0 1 6 3	$E + F$	$\$$	reduce by $T \to F$
(13)	0 1 6 9	$E + T$	$\$$	reduce by $E \to E + T$
(14)	0 1	E	$\$$	accept

活前缀(Viable Prefixes)

Viable Prefixes

A viable prefix must be a prefix of a right-sentential form.

$S \Rightarrow^*_{rm} \alpha\, \omega$, where α is the content of the stack and ω contains no nonterminals.

Not all prefixes of a right-sentential form are viable prefixes.

$E \Rightarrow^*_{rm} F * n \Rightarrow_{rm} (E) * n$,

where (, (E, (E) are viable prefixes,

but (E) * is not, since the parser will perform reduction once the handle appears.

Viable Prefixes

A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the handle of that sentential form.

SLR parsing is based on the fact that LR(0) automata recognize viable prefixes.

See you next time!