Bottom Up Parsing

Bottom-Up Parsing

- A **bottom-up parser** creates the parse tree of the given input starting from leaves towards the root
- A bottom-up parser tries to find the **right-most derivation** of the given input in the reverse order.

$S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

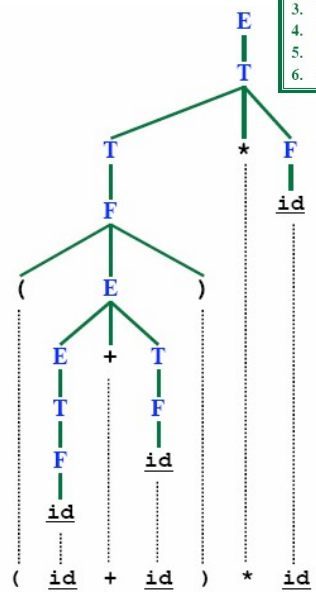
$\leftarrow$ (the bottom-up parser finds the right-most derivation in the reverse order)

Rightmost Derivation

Input string: (id + id) * id

Rules Used:	*Right-Sentential Forms:*
E → T	E
T → T * F	T
F → id	T * F
T → F	T * id
F → (E)	F * id
E → E + T	(E) * id
T → F	(E + T) * id
F → id	(E + F) * id
E → T	(E + id) * id
T → F	(T + id) * id
F → id	(F + id) * id
	(id + id) * id

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id



Reduction

- A reduction step replaces a specific substring (matching the body of a production)

(id + id) * id	(E) * id
(F + id) * id	F * id
(T + id) * id	T * id
(E + id) * id	T * F
(E + F) * id	T
(E + T) * id	E

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

- Reduction is the opposite of derivation
- Bottom up parsing is a process of reducing a string ω to the start symbol S of the grammar

Handle

- Informally, a **handle** is a substring (in the parsing string) that matches the right side of a production rule.
  - But not every substring matches the right side of a production rule is handle
- A **handle** of a right sentential form γ ($\equiv \alpha\beta\omega$) is a production rule $A \rightarrow \beta$ and a position of γ

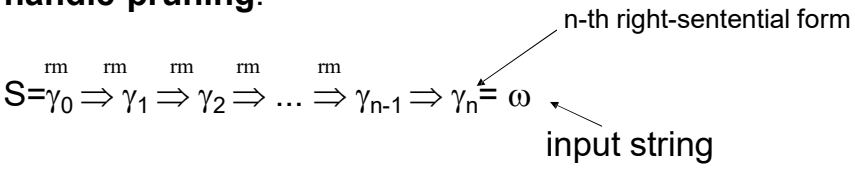
  where the string β may be found and replaced by A to produce

  the previous right-sentential form in a rightmost derivation of γ.

$$S \Rightarrow \alpha A\omega \Rightarrow \alpha\beta\omega$$

Handle Pruning

- A right-most derivation in reverse can be obtained by **handle-pruning**.



- Start from γ_n, find a handle $A_n \rightarrow \beta_n$ in γ_n, and replace β_n in by A_n to get γ_{n-1}.
- Then find a handle $A_{n-1} \rightarrow \beta_{n-1}$ in γ_{n-1}, and replace β_{n-1} in by A_{n-1} to get γ_{n-2}.
- Repeat this, until we reach S.

Shift-Reduce Parsing

- Bottom-up parsing is also known as **shift-reduce parsing** because its two main actions are shift and reduce.
- data structures: input-string and stack
- Operations
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - Accept: Announce successful completion of parsing
  - Error: Discover a syntax error and call error recovery

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Remaining input: abbcde

Rightmost derivation:

S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ a b b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b

Remaining input: bcde

a b

Rightmost derivation:
S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ a b b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b

Remaining input: bcde

T
|
a b

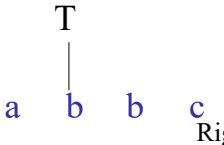
Rightmost derivation:
S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c

Remaining input: de



Rightmost derivation:

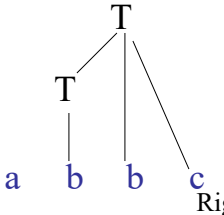
S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c
Reduce T --> T b c

Remaining input: de



Rightmost derivation:

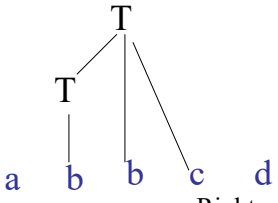
S ⓪ a T R e
⓪ a **T d** e
⓪ a **T b c** d e
⓪ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c
Reduce T --> T b c
Shift d

Remaining input: e



Rightmost derivation:

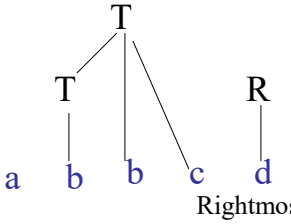
S ⇒ a T R e
⇒ a T **d** e
⇒ a **T b c** de
⇒ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c
Reduce T --> T b c
Shift d
Reduce R --> d

Remaining input: e



Rightmost derivation:

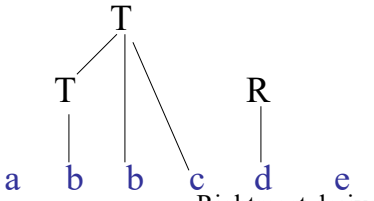
S ⇒ a T R e
⇒ a T **d** e
⇒ a **T b c** de
⇒ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c
Reduce T --> T b c
Shift d
Reduce R --> d
Shift e

Remaining input:



Rightmost derivation:

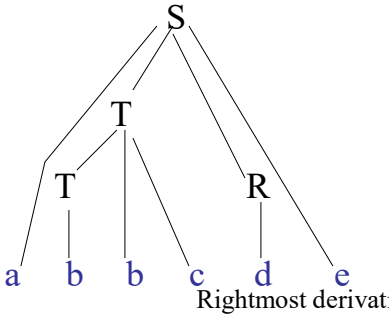
S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ **a b** b c d e

Shift Reduce Parsing

S -> a T R e
T -> T b c | b
R -> d

Shift a, Shift b
Reduce T --> b
Shift b, Shift c
Reduce T --> T b c
Shift d
Reduce R --> d
Shift e
Reduce S --> a T R e

Remaining input:



Rightmost derivation:

S ⓪ a T R e
⓪ a T **d** e
⓪ a **T b c** d e
⓪ **a b** b c d e

Example Shift-Reduce Parsing

Consider the grammar:

Input string: id + id

Stack	Input	Action
$	id_1 + $id_2$$	shift
$$id_1$$	+ $id_2$$	reduce 6
$F	+ $id_2$$	reduce 4
$T	+ $id_2$$	reduce 2
$E	+ $id_2$$	shift
$E +	$id_2$$	shift
$E + $id_2$$		reduce 6
$E + F		reduce 4
$E + T		reduce 1
$E		accept

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsers cannot be used.
- Stack contents and the next input symbol may not decide action:
  - **shift/reduce conflict**: Whether make a shift operation or a reduction.
  - **reduce/reduce conflict**: The parser cannot decide which of several reductions to make.
- If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.

- An ambiguous grammar can never be a LR grammar.

Shift-Reduce Conflict in Ambiguous Grammar

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

STACK	**INPUT**
….**if** *expr* **then** *stmt*	**else**….$

- We cant decide whether to shift or reduce?
- But we can adapt to parse certain ambiguous grammar to using shift-reducing parsers
  - We resolve in favor of SHIFT then we have a solution

Reduce-Reduce Conflict in Ambiguous Grammar

(1)	*stmt*	→	**id** (*parameter_list*)
(2)	*stmt*	→	*expr* := *expr*
(3)	*parameter_list*	→	*parameter_list* , *parameter*
(4)	*parameter_list*	→	*parameter*
(5)	*parameter*	→	**id**
(6)	*expr*	→	**id** (*expr_list*)
(7)	*expr*	→	**id**
(8)	*expr_list*	→	*expr_list* , *expr*
(9)	*expr_list*	→	*expr*

Figure 4.30: Productions involving procedure calls and array references

STACK	INPUT
··· **id** (**id**	, **id**) ···

We have to reduce ***id***
We can use both *(5)* and (7) to reduce ***id***

Canonical LR(1) or LR(1) Parser

- Makes use of one lookahead symbol in the input.
- LR(1) items carry more information.
- LR(1) Item: A → X1...Xi · Xi+1...Xj , t
  - Have states X1...Xi in the stack already.
  - Expect to put Xi+1...Xj onto the stack and then reduce.
  - Only reduce when the token following Xj is t.

Cluster the items:
[A → α · , a/b/c] means the three items: [A → α · , a], [A → α · , b], [A → α · , c].
"Reduce α to A if the next token is a or b or c."

LR(1) Sets of Items

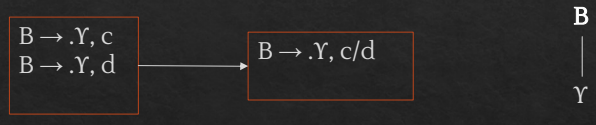
There are more items and item sets in LR(1) than in SLR.
Closure: For each item [A → α · B β, a] in the set of items I, for each production B → γ in the grammar G, and for each terminal b in First(βa), add [B → · γ, b] to I.
Once we have a closed item set, use the LR(1) successor/GOTO function to compute transitions and next items.
Example:
S' → S
S → dca | dAb | Aa
A → c
Initial Item: [S' → · S, $]
Closure:
[S → · dca, $]
[S → · dAb, $]
[S → · Aa, $]
[A → · c, a]

Closure

$A \rightarrow \alpha \cdot B\ \beta, a$

For each item $[A \rightarrow \alpha \cdot B\ \beta, a]$ in the set of items I, for each production $B \rightarrow \gamma$ in the grammar G, and for each terminal b in First(βa), add $[B \rightarrow \cdot\ \gamma, b]$ to I.

Let, First(β) = {c, d}
First(βa) = {c, d}

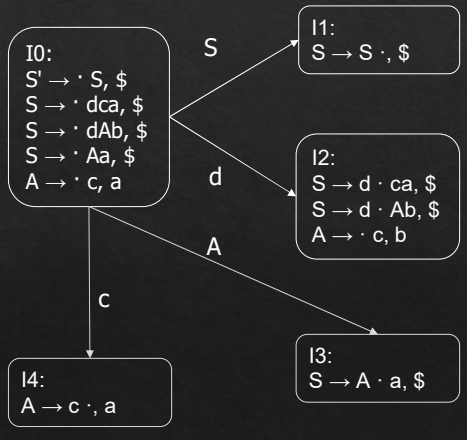


What if, First(β) = {ε, c, d}?
First(βa) = {a, c, d}

B → · ϒ, a
B → · ϒ, c
B → · ϒ, d
Or, B → · ϒ, a/b/c

LR(1) GOTO Function

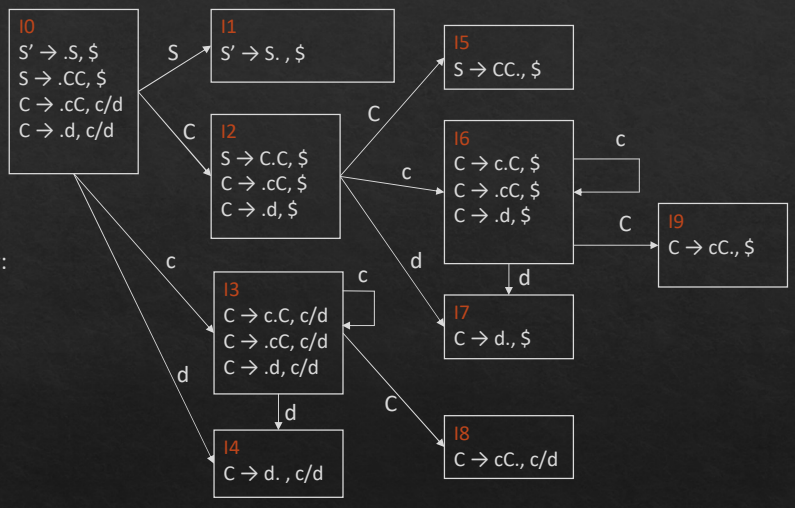
Given an item set I, initialize an empty set J. For each $[A \rightarrow \alpha \cdot X\ \beta, a]$, add $[A \rightarrow \alpha\ X \cdot \beta, a]$ to item set J.
GOTO(I, X) is the closure of set J.



GOTO graph of LR(1) Items

The Grammar:
S → CC
C → cC|d

Augmented Grammar:
0.S' → S
1.S → CC
2.C → cC
3.C → d



Canonical LR(1) Parsing Table

Construct the collection of sets of LR(1) items C' = {I0, I1, I2, ..., In} for the augmented grammar G'.
State i of the parser is constructed from Ii. The parsing actions for state i are determined as follows:
- If [A → α · a β, b] is in Ii and GOTO(Ii, a) = Ij, then set ACTION[i, a] to "SHIFT j". Here, a must be a terminal.
- If [A → α · , b] is in Ii and A ≠ S', then set ACTION[i, b] to "REDUCE A → α". Here, b must be a terminal.
- If [S' → S · , $] is in Ii, then set ACTION[i, $] to "Accept".

The GOTO transitions for state i are constructed for all non-terminals A using the rule: If GOTO(Ii, A) = Ij, then GOTO[i, A] = j.
Any entries that are not defined by the points above are made "error".
The initial state of the parser is constructed from the set of items containing [S' → S · , $].

I0
S' → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c/d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

c c d c d $

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

c c d c d $

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

c c | d c d $

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

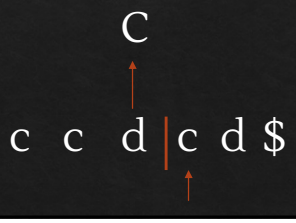
c c d | c d $

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

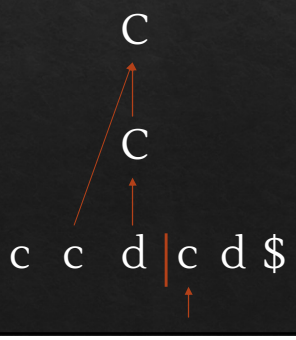


Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

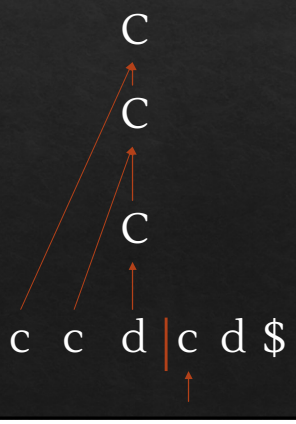


Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

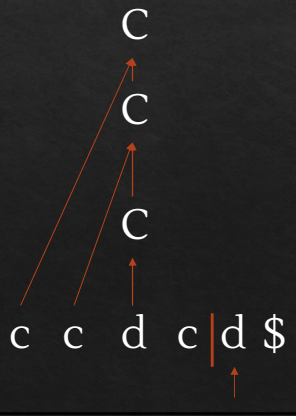


Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

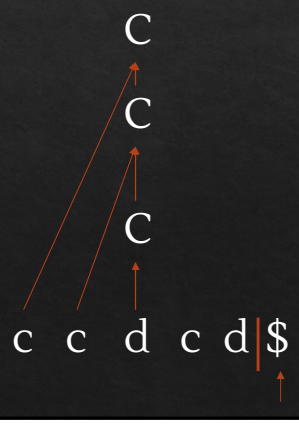


Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept



Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

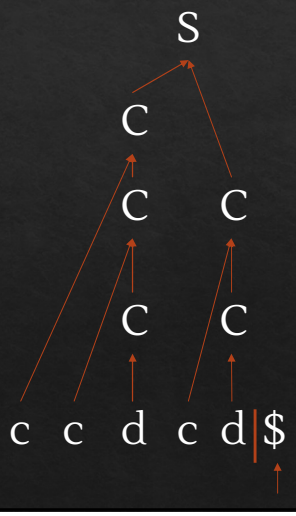
	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

Simulation

Augmented Grammar:
0.S′ → S
1.S → CC
2.C → cC
3.C → d

States	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

	Stack	Symbol	Input	Action
0	0		ccdcd$	Shift 3
1	03	c	cdcd$	Shift 3
2	033	cc	dcd$	Shift 4
3	0334	ccd	cd$	Reduce by C->d
4	0338	ccC	cd$	Reduce by C -> cC
5	038	cC	cd$	Reduce by C -> cC
7	02	C	cd$	Shift 6
7	026	Cc	d$	Shift 7
8	0267	Ccd	$	Reduce by C->d
9	0269	CcC	$	Reduce by C -> cC
10	025	CC	$	Reduce by S -> CC
11	01	S	$	Accept

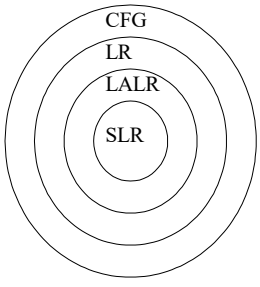


SLR (1) Parsing

Shift-Reduce Parsers

There are two main categories of shift-reduce parsers

1. **Operator-Precedence Parser**
   - simple, but only a small class of grammars.



2. **LR-Parsers**
   - covers wide range of grammars.
     - SLR – simple LR parser
     - LR – most general LR parser
     - LALR – intermediate LR parser (lookhead LR parser)
   - SLR, LR and LALR work same, only their parsing tables are different.

LR Parsers

LR parsing is attractive because:

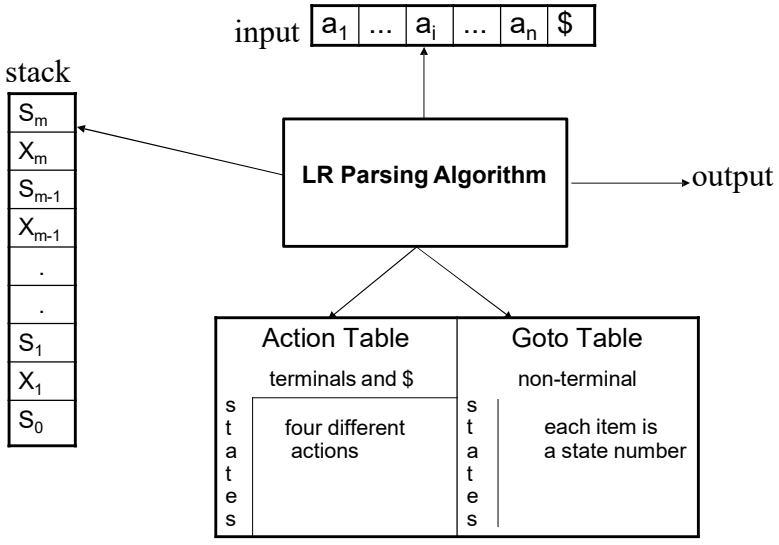
- LR parsing is most general non-backtracking shift-reduce parsing, yet it is still efficient.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

  LL(1)-Grammars $\subset$ LR(1)-Grammars
- An LR-parser can detect a syntactic error as soon as it is possible to do so a left-to-right scan of the input.
- LR parsers can be constructed to recognize virtually all programming language constructs for which CFG grammars can be written

Drawback of LR method:

- Too much work to construct LR parser by hand
  - Fortunately tools (LR parsers generators) are available

LR Parsing Algorithm



Items and LR(0) Automaton

An LR(0) item of a grammar G is a production of G with a dot in some position in the body . Thus a production A-> XYZ yields the following 4 items:

$$A \to \cdot XYZ$$
$$A \to X \cdot YZ$$
$$A \to XY \cdot Z$$
$$A \to XYZ \cdot$$

The production A-> ε generates only one item, A->.

LR(0) collection of items

- A collection of sets of LR(0) items is called the canonical LR(0) collection
- This collection provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions
- To construct the canonical LR(0) collection for a grammar , we define an augmented grammar and two functions, CLOSURE and GOTO
- 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

LR(0) collection of items

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:

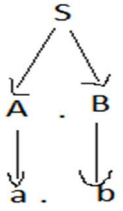
1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

S -> AB

A -> a

B -> b

S -> A.B



LR(0) collection of items

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:

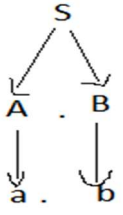
1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

S -> AB

A -> a

B -> b

S -> A.B



INSERT B-> .b in CLOSURE

LR(0) collection of items

CLOSURE EXAMPLE:

Example 4.40: Consider the augmented expression grammar:

$$\begin{array}{rcl} E' & \rightarrow & E \\ E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ E & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

If I is the set of one item $\{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) contains the set of items I_0 in Fig. 4.31.

LR(0) collection of items

CLOSURE EXAMPLE:

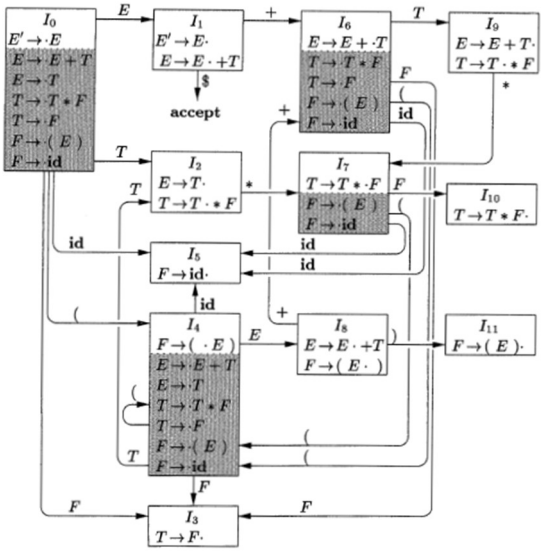
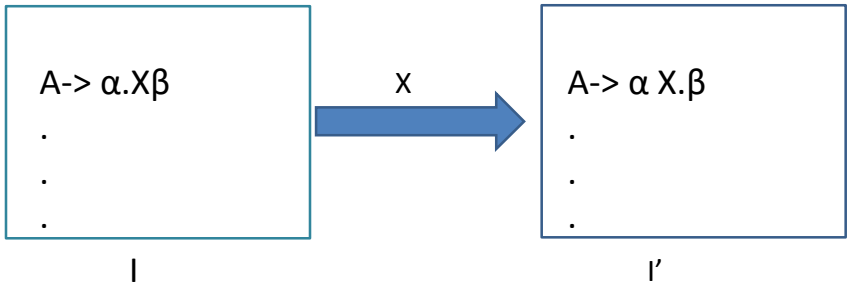


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

LR(0) collection of items

GOTO:



GOTO (I,X) = I'

LR(0) collection of items

GOTO EXAMPLE:

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

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

LR(0) collection of items

Consider the following grammar:

S-> AB

A-> a

B -> b

Let's build an LR(0) Automaton for the grammar

First step is to augment the grammar by introducing a new start symbol

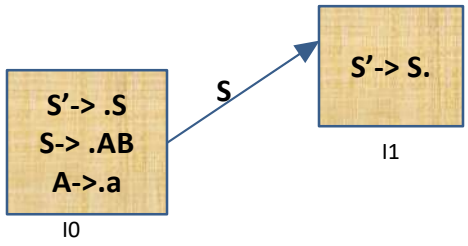
S'->S
S-> AB
A-> a
B -> b

LR(0) collection of items

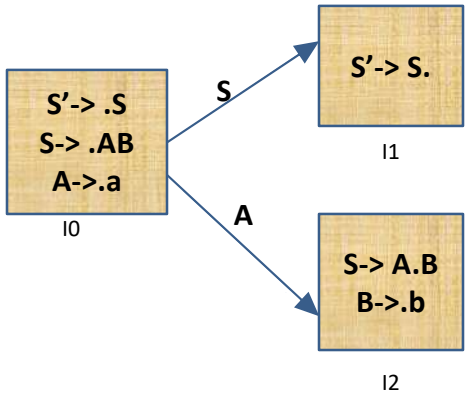
S'-> .S
S-> .AB
A->.a

I0

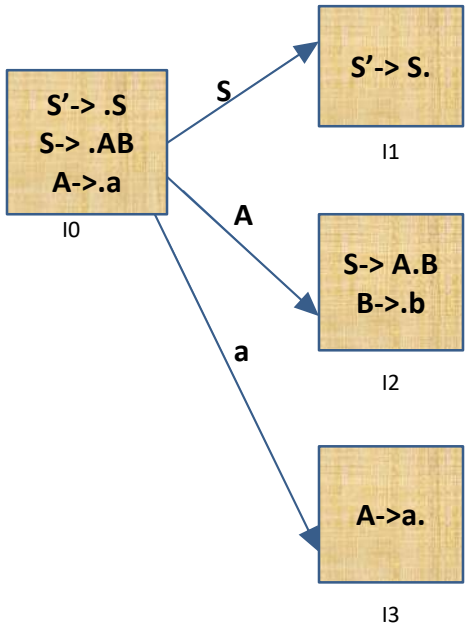
LR(0) collection of items



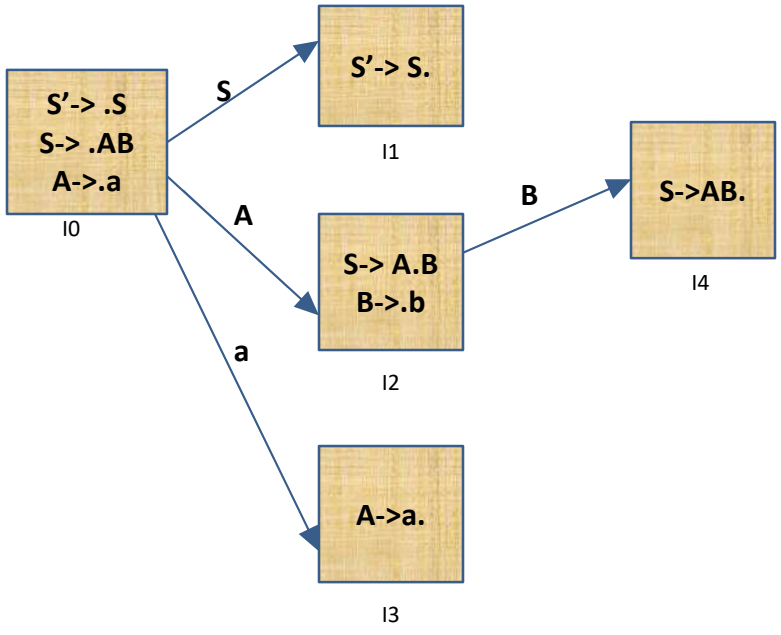
LR(0) collection of items



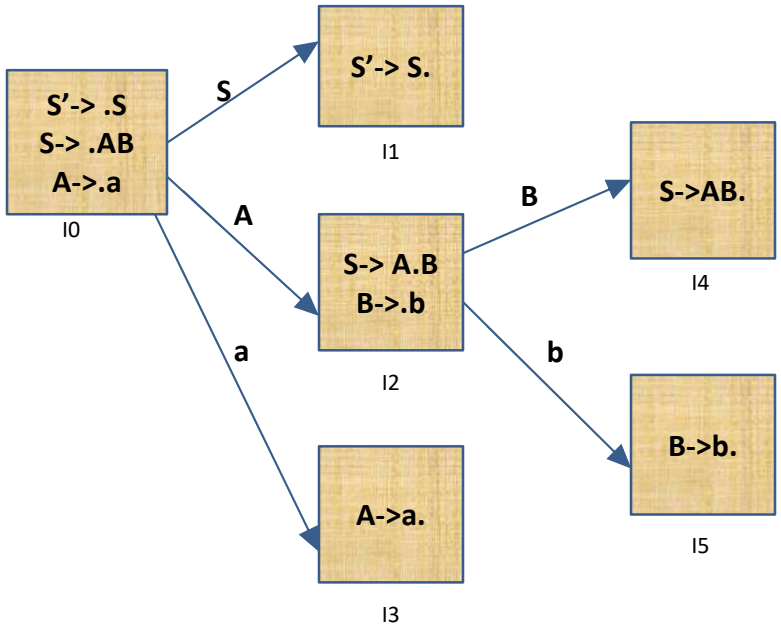
LR(0) collection of items



LR(0) collection of items



LR(0) collection of items



LR Parse table

Now, build SLR(1) Parse table from canonical LR(0) collection of items

Action(i,a)	GOTO(i,A)

Structure of LR Parsing Table

Structure of the LR Parsing Table

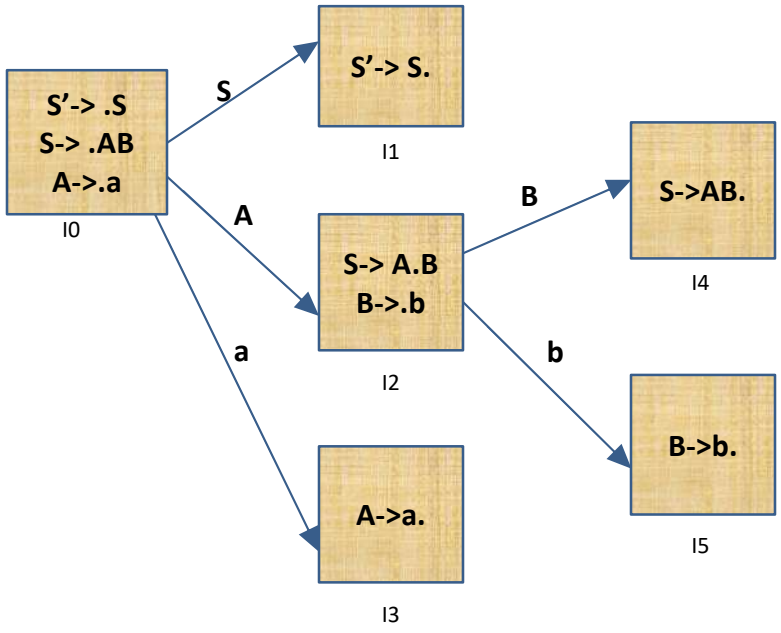
The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:
   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.
   (b) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A.
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

Building SLR(1) Parse table

Lets see the canonical collection of LR(0) items again

LR(0) collection of items



SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	s3					
1						
2		s5				
3						
4						
5						

SLR (1) parse table

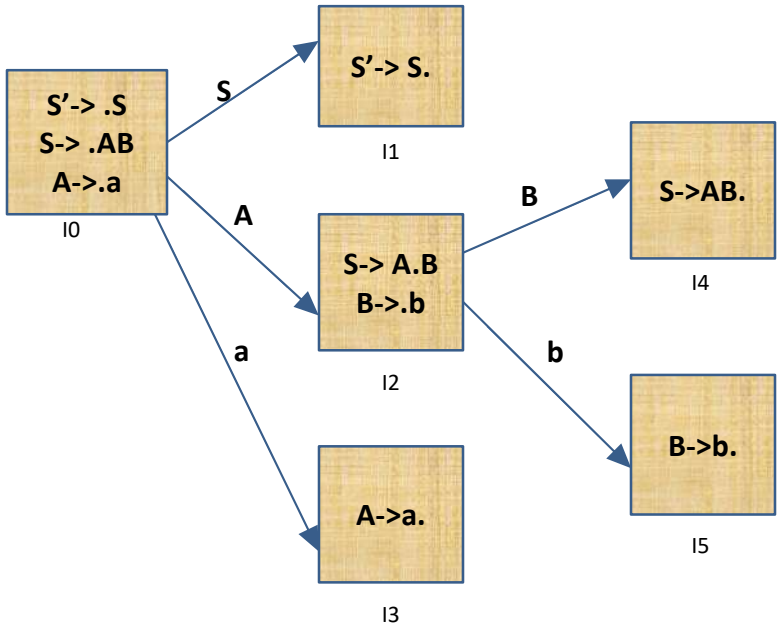
States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	s3			1	2	
1						
2		s5				4
3						
4						
5						

SLR(1) PARSE TABLE

Items in reduced form:

0. S'->S.
1. S-> AB.
2. A->a.
3. B->b.

LR(0) collection of items



SLR(1) PARSE TABLE

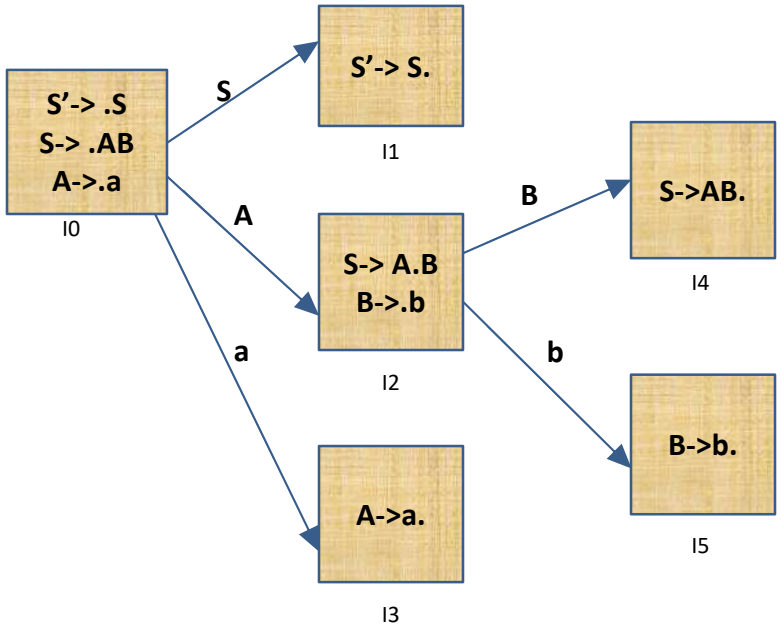
Items in reduced form:

0. S'->S.
Follow (S')={$}
Rule 0 is in state 1, so reduction/ acceptance in state 1

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	s3			1	2	
1			accept			
2		s5				4
3						
4						
5						

LR(0) collection of items



SLR(1) PARSE TABLE

Items in reduced form:

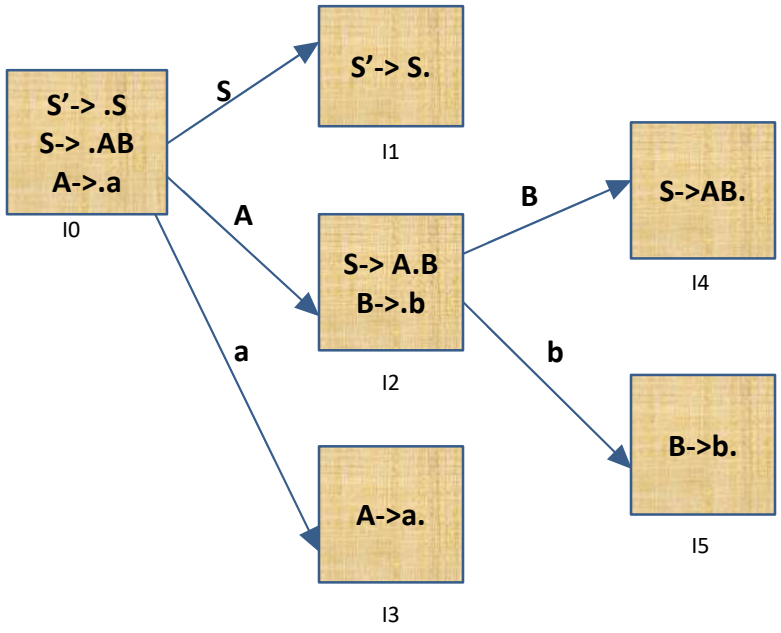
0. S'->S.
Follow (S')={$}
Rule 0 is reduced in state 1, so reduction/acceptance in state 1

1. S-> AB.
Follow (S) = {$}
Rule 1 is reduced in state 4, so reduction by rule 1 in state 4

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3						
4			R1			
5						

LR(0) collection of items



SLR(1) PARSE TABLE

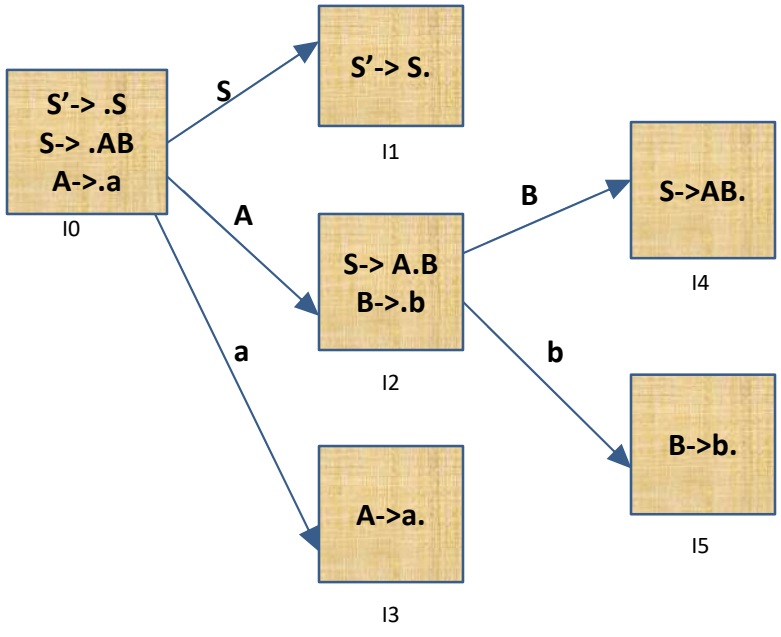
Items in reduced form:

2. A-> a.
Follow (A)={b}
Rule 2 is reduced in state 3, so reduction by rule 2 in state 3

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5						

LR(0) collection of items



SLR(1) PARSE TABLE

Items in reduced form:

3. B->b.
Follow (B)={$}
Rule 3 is reduced in state 5 , so reduction by rule 3 in state 5

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	Shift to 3

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	Shift to 3
0 3	a b $ ⬆	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	Shift to 3
0 3	a b $ ⬆	Reduce using A->a

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Action (3,b) = Reduce using Rule 2
= Reduce using A->a

Pop |a|=1 symbol off the stack, 3 gets popped off
Let state t=0 be the top of stack
Push GOTO (t,A)= GOTO(0,A)= 2 onto stack

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	Shift to 3
0 3	a b $ ⬆	Reduce using A->a
0 2	a b $ ⬆	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5
0 2 5	a b $ ↑	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5
0 2 5	a b $ ↑	Reduce using B->b

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Action (5,$) = Reduce using Rule 3
= Reduce using B->b

Pop |b|=1 symbol off the stack, 3 gets popped off
Let state t=2 be the top of stack
Push GOTO (t,A)= GOTO(2,B)= 4 onto stack

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5
0 2 5	a b $ ↑	Reduce using B->b
0 2 4	a b $ ↑	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ⬆	Shift to 3
0 3	a b $ ⬆	Reduce using A->a
0 2	a b $ ⬆	Shift to 5
0 2 5	a b $ ⬆	Reduce using B->b
0 2 4	a b $ ⬆	Reduce using S->AB

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Action (4,$) = Reduce using Rule 1
= Reduce using S-> AB

Pop |AB|=2 symbol off the stack, 3 gets popped off
Let state t=0 be the top of stack
Push GOTO (t,A)= GOTO(0,S)= 1 onto stack

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5
0 2 5	a b $ ↑	Reduce using B->b
0 2 4	a b $ ↑	Reduce using S->AB
0 1	a b $ ↑	

SLR (1) parse table

States	ACTION(I,a)			GOTO (I,A)		
	a	b	$	S	A	B
0	S3			1	2	
1			accept			
2		S5				4
3		R2				
4			R1			
5			R3			

SLR PARSING USING PARSE TABLE

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Output of SLR(1) parser on “ab$”

Stack	Input	Action
0	a b $ ↑	Shift to 3
0 3	a b $ ↑	Reduce using A->a
0 2	a b $ ↑	Shift to 5
0 2 5	a b $ ↑	Reduce using B->b
0 2 4	a b $ ↑	Reduce using S->AB
0 1	a b $ ↑	accept

4.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define *FIRST*(α), where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α). For example, in Fig. 4.15, $A \stackrel{*}{\Rightarrow} c\gamma$, so c is in FIRST(A).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions $A \rightarrow \alpha \mid \beta$, where FIRST($\alpha$) and FIRST($\beta$) are disjoint sets. We can then choose between these A-productions by looking at the next input

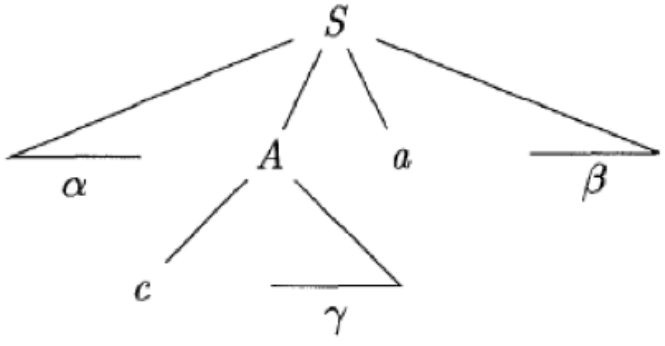


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both. For instance, if a is in FIRST(β) choose the production $A \rightarrow \beta$. This idea will be explored when LL(1) grammars are defined in Section 4.4.3.

Define *FOLLOW*(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β, as in Fig. 4.15. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A); recall that \$ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (\ E\ ) \mid \mathbf{id}
\end{array}
\tag{4.28}
$$

Example 4.30: Consider again the non-left-recursive grammar (4.28). Then:

1. FIRST(F) = FIRST(T) = FIRST(E) = $\{(, \mathbf{id}\}$. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST$(E') = \{+, \epsilon\}$. The reason is that one of the two productions for E' has a body that begins with terminal $+$, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. FIRST$(T') = \{*, \epsilon\}$. The reasoning is analogous to that for FIRST(E').

4. FOLLOW(E) = FOLLOW$(E') = \{), \$\}$. Since E is the start symbol, FOLLOW(E) must contain \$. The production body $(\ E\ )$ explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW$(T') = \{+,), \$\}$. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol $+$. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW$(F) = \{+, *,), \$\}$. The reasoning is analogous to that for T in point (5).

First - Example

- P → i | c | n T S
- Q → P | a S | b S c S T
- R → b | ε
- S → c | R n | ε
- T → R S q

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,b}
- FIRST(R) = {b,ε}
- FIRST(S) = {c,b,n,ε}
- FIRST(T) = {b,c,n,q}

First - Example

- S → a S e | S T S
- T → R S e | Q
- R → r S r | ε
- Q → S T | ε

- FIRST(S) = {a}
- FIRST(R) = {r, ε}
- FIRST(T) = {r, a, ε}
- FIRST(Q) = {a, ε}

Example

- S → a S e | B
- B → b B C f | C
- C → c C g | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) = $\{f,g\} \cup \text{FOLLOW}(B) = \{c,d,e,f,g,\$\}$
- FOLLOW(B) = $\{c,d,f\} \cup \text{FOLLOW}(S) = \{c,d,e,f,\$\}$
- FOLLOW(S) = {$, e }

Example

- S → (A) | ε
- A → T E
- E → & T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {&, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) = {$}
- FOLLOW(A) = {) }
- FOLLOW(E) = FOLLOW(A) = {) }
- FOLLOW(T) = FIRST(E) ∪ FOLLOW(A) ∪ FOLLOW(E) = {&,)}

4.6.2 Items and the LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents $\$T$ and next input symbol $*$ in Fig. 4.28, how does the parser know that T on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce T to E?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items." An *LR(0) item* (*item* for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production $A \rightarrow XYZ$ yields the four items

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

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

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input. Item

$A \to X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ. Item $A \to XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an *LR(0) automaton.*[3] In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (4.1), shown in Fig. 4.31, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO. 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 $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

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).

[3] Technically, the automaton misses being deterministic according to the definition of Section 3.6.4, because we do not have a dead state, corresponding to the empty set of items. As a result, there are some state-input pairs for which no next state exists.

Example 4.40: Consider the augmented expression grammar:

$$
\begin{array}{lcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
E & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

If I is the set of one item $\{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) contains the set of items I_0 in Fig. 4.31.

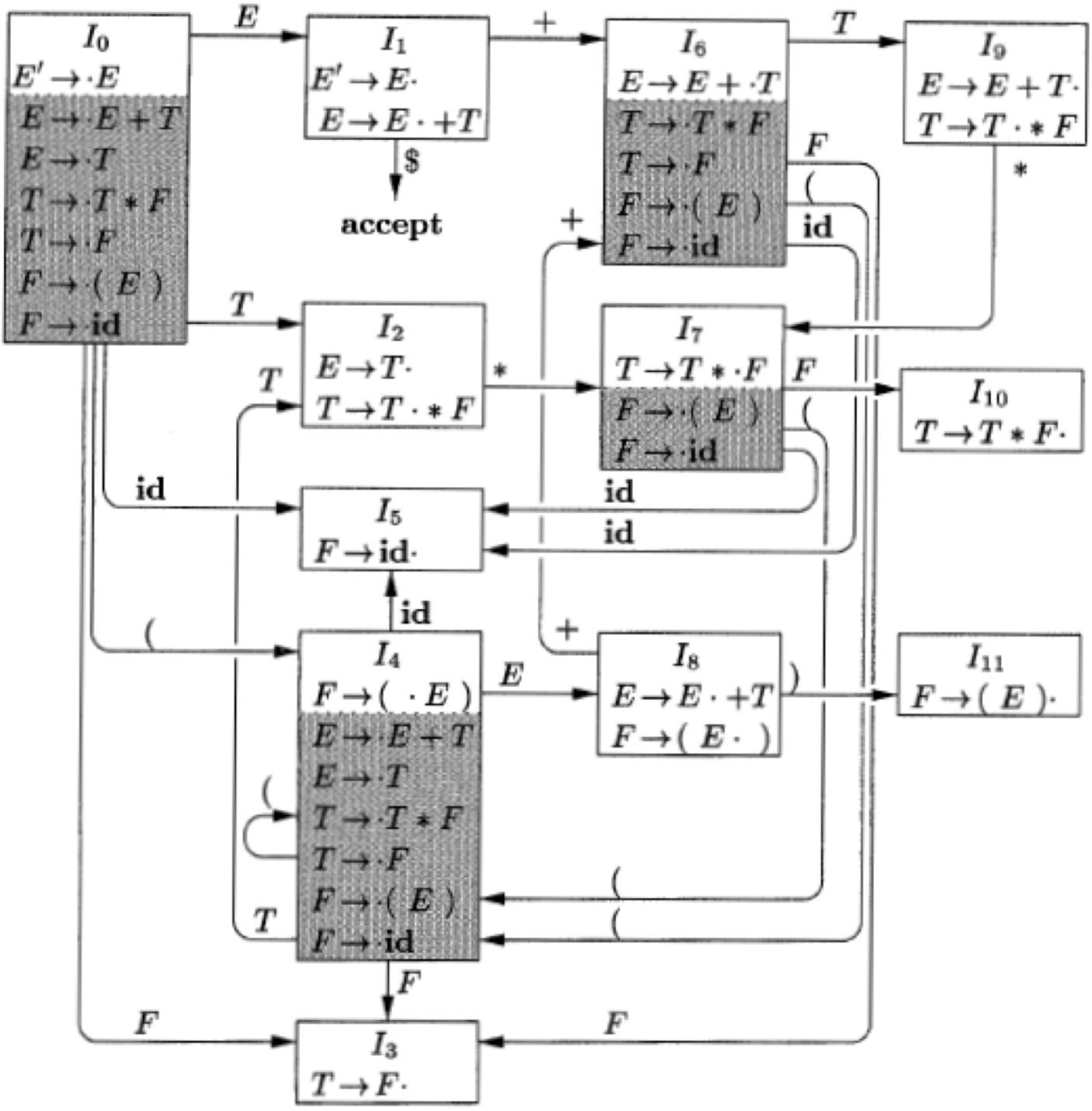


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

To see how the closure is computed, $E' \to \cdot E$ is put in CLOSURE(I) by rule (1). Since there is an E immediately to the right of a dot, we add the E-productions with dots at the left ends: $E \to \cdot E + T$ and $E \to \cdot T$. Now there is a T immediately to the right of a dot in the latter item, so we add $T \to \cdot T * F$ and $T \to \cdot F$. Next, the F to the right of a dot forces us to add $F \to \cdot(E)$ and $F \to \cdot \mathbf{id}$, but no other items need to be added. □

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. Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X.

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

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

We computed GOTO($I, +$) by examining I for items with $+$ immediately to the right of the dot. $E' \to E\cdot$ is not such an item, but $E \to E\cdot + T$ is. We moved the dot over the $+$ to get $E \to E + \cdot T$ and then took the closure of this singleton set. □

4.6.3 The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 4.35. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state*. Each state summarizes the information contained in the stack below it.

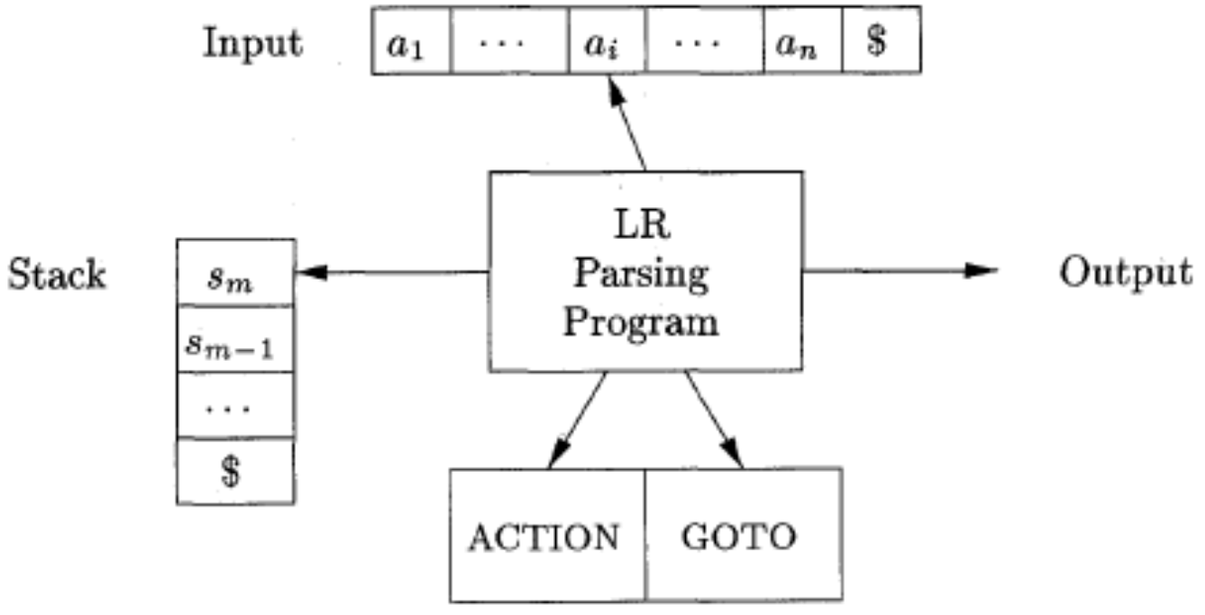


Figure 4.35: Model of an LR parser

Structure of the LR Parsing Table

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state i and a terminal a (or \$, the input endmarker). The value of ACTION$[i, a]$ can have one of four forms:

   (a) Shift j, where j is a state. The action taken by the parser effectively shifts input a to the stack, but uses state j to represent a.

   (b) Reduce $A \to \beta$. The action of the parser effectively reduces β on the top of the stack to head A.

   (c) Accept. The parser accepts the input and finishes parsing.

   (d) Error. The parser discovers an error in its input and takes some corrective action. We shall have more to say about how such error-recovery routines work in Sections 4.8.3 and 4.9.4.

2. We extend the GOTO function, defined on sets of items, to states: if GOTO$[I_i, A] = I_j$, then GOTO also maps a state i and a nonterminal A to state j.

Algorithm 4.44: LR-parsing algorithm.

INPUT: An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G.

OUTPUT: If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.36. □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

Figure 4.36: LR-parsing program

Example 4.45: Figure 4.37 shows the ACTION and GOTO functions of an LR-parsing table for the expression grammar (4.1), repeated here with the productions numbered:

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

The codes for the actions are:

1. si means shift and stack state i,
2. rj means reduce by the production numbered j,
3. acc means accept,
4. blank means error.

Note that the value of GOTO[s, a] for terminal a is found in the ACTION field connected with the shift action on input a for state s. The GOTO field gives GOTO[s, A] for nonterminals A. Although we have not yet explained how the entries for Fig. 4.37 were selected, we shall deal with this issue shortly.

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			

Figure 4.37: Parsing table for expression grammar

4.6.4 Constructing SLR-Parsing Tables

On input **id** $*$ **id** $+$ **id**, the sequence of stack and input contents is shown in Fig. 4.38. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.37 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and **id** has been removed from the input.

Then, $*$ becomes the current input symbol, and the action of state 5 on input $*$ is to reduce by $F \rightarrow$ **id**. One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on F is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. $\square$

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** \$	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** \$	reduce by $F \to$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** \$	reduce by $T \to F$
(4)	0 2	T	$*$ **id** $+$ **id** \$	shift
(5)	0 2 7	$T*$	**id** $+$ **id** \$	shift
(6)	0 2 7 5	$T*$ **id**	$+$ **id** \$	reduce by $F \to$ **id**
(7)	0 2 7 10	$T*F$	$+$ **id** \$	reduce by $T \to T*F$
(8)	0 2	T	$+$ **id** \$	reduce by $E \to T$
(9)	0 1	E	$+$ **id** \$	shift
(10)	0 1 6	$E+$	**id** \$	shift
(11)	0 1 6 5	$E+$ **id**	\$	reduce by $F \to$ **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

Figure 4.38: Moves of an LR parser on **id** $*$ **id** $+$ **id**

Algorithm 4.46: Constructing an SLR-parsing table.

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.

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 $[S' \to \cdot S]$.

Example 4.48: Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1). Consider the grammar with productions

$$\begin{array}{lcl} S & \rightarrow & L = R \mid R \\ L & \rightarrow & *R \mid \mathbf{id} \\ R & \rightarrow & L \end{array} \tag{4.49}$$

Think of L and R as standing for *l*-value and *r*-value, respectively, and $*$ as an operator indicating "contents of."[5] The canonical collection of sets of LR(0) items for grammar (4.49) is shown in Fig. 4.39.

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow L \cdot = R$
$R \rightarrow L \cdot$

I_3: $S \rightarrow R \cdot$

I_4: $L \rightarrow * \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id} \cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow * R \cdot$

I_8: $R \rightarrow L \cdot$

I_9: $S \rightarrow L = R \cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Consider the set of items I_2. The first item in this set makes ACTION$[2, =]$ be "shift 6." Since FOLLOW(R) contains $=$ (to see why, consider the derivation $S \Rightarrow L = R \Rightarrow *R = R$), the second item sets ACTION$[2, =]$ to "reduce $R \rightarrow L$." Since there is both a shift and a reduce entry in ACTION$[2, =]$, state 2 has a shift/reduce conflict on input symbol $=$.

Grammar (4.49) is not ambiguous. This shift/reduce conflict arises from the fact that the SLR parser construction method is not powerful enough to remember enough left context to decide what action the parser should take on input $=$, having seen a string reducible to L. The canonical and LALR methods,

Example 1: Consider the following augmented grammar:

1. $E' \rightarrow E\$$
2. $E \rightarrow id$
3. $E \rightarrow (E)$
4. $E \rightarrow (id)E$

i) Draw the LR(0) automation for this grammar.

ii) Construct SLR parsing table

iii) Is the grammar LR(0)? why/why not?

	FIRST	FOLLOW
E'	first $(E') = \{id, (\}$	~~first~~ Follow $(E') = \{\$\}$
E	first $(E) = \{id, (\}$	follow $(E) = \{\$,)\}$

(i) LR(0) automation:-

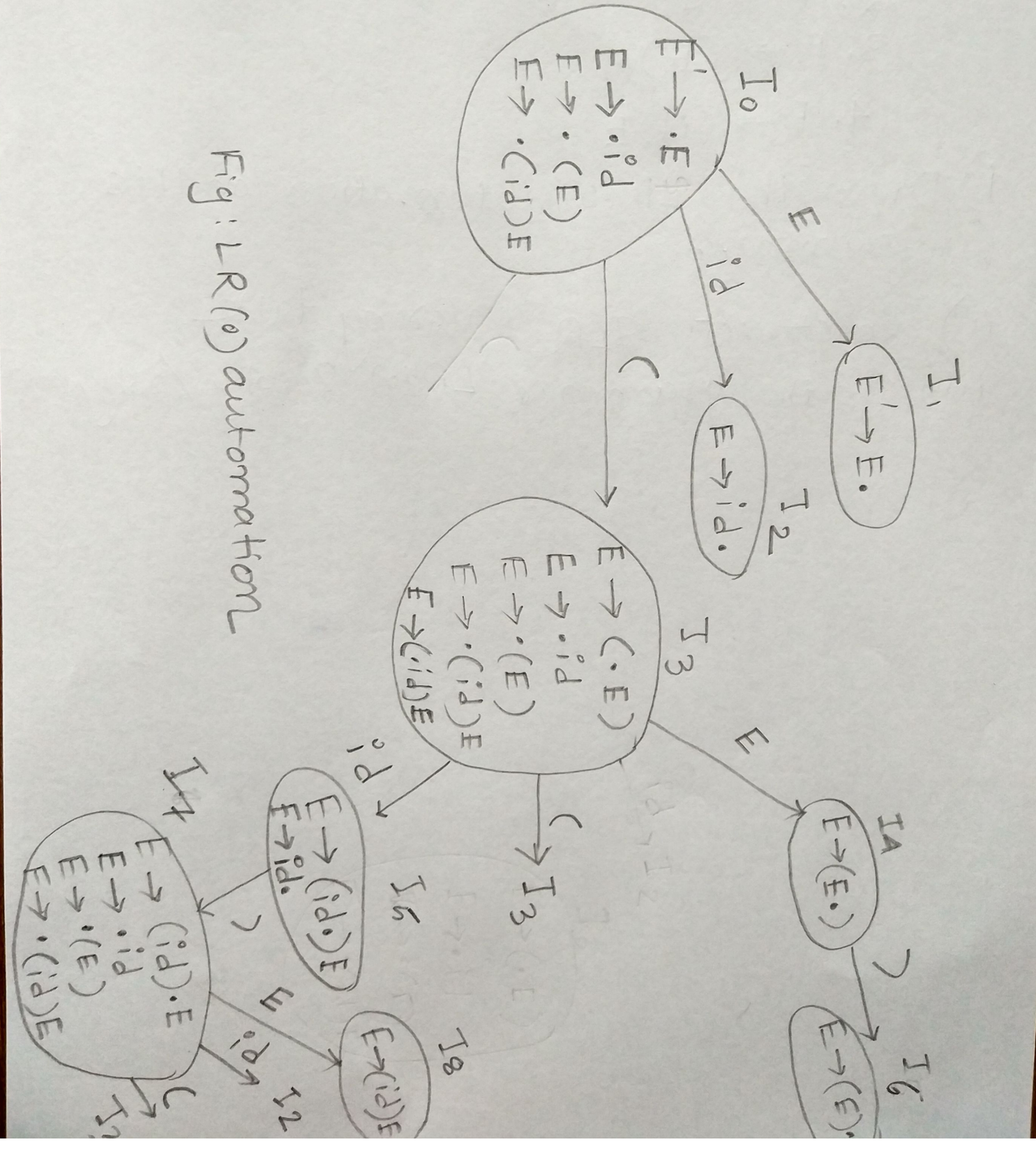


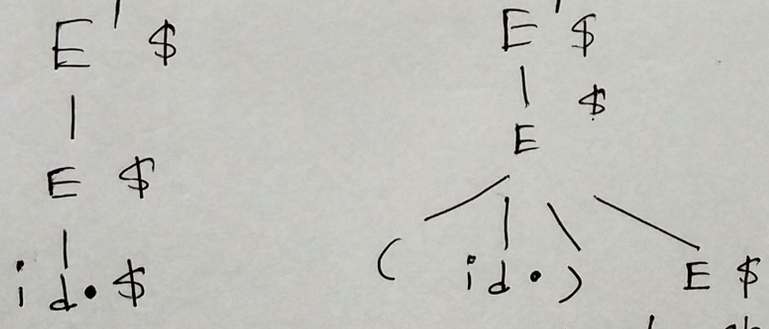
Fig: LR(0) automation

(ii)

	ACTION (i,a)				GOTO(i, A)
State	id	(	)	$	E
0	S2	S3			1
1				acc	
2			R2	R2	
3	S5	S3			4
4			S6		
5			S7 R2	R2	
6			R3	R3	
7	S2	S3			8
8			R4	R4	

(iii)

The grammar is not LR(0), because it has a shift-reduce conflict in state I_5



The appropriate action would be to shift only reduce id to E when lookahead is $.

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
\tag{4.49}
$$

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

I_2: $S \rightarrow L\cdot = R$
$R \rightarrow L\cdot$

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

Figure 4.39: Canonical LR(0) collection for grammar (4.49)

Example 4.51: Let us reconsider Example 4.48, where in state 2 we had item $R \to L\cdot$, which could correspond to $A \to \alpha$ above, and a could be the = sign, which is in FOLLOW(R). Thus, the SLR parser calls for reduction by $R \to L$ in state 2 with = as the next input (the shift action is also called for, because of item $S \to L\cdot{=}R$ in state 2). However, there is no right-sentential form of the grammar in Example 4.48 that begins $R = \cdots$. Thus state 2, which is the state corresponding to viable prefix L only, should not really call for reduction of that L to R. □

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. By splitting states when necessary, we can arrange to have each state of an LR parser indicate exactly which input symbols can follow a handle α for which there is a possible reduction to A.

The extra information is incorporated into the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \cdot \beta, a]$, where $A \to \alpha\beta$ is a production and a is a terminal or the right endmarker \$. We call such an object an *LR(1) item*. The 1 refers to the length of the second component, called the *lookahead* of the item.[6] The lookahead has no effect in an item of the form $[A \to \alpha\cdot\beta, a]$, where β is not ϵ, but an item of the form $[A \to \alpha\cdot, a]$ calls for a reduction by $A \to \alpha$ only if the next input symbol is a. Thus, we are compelled to reduce by $A \to \alpha$ only on those input symbols a for which $[A \to \alpha\cdot, a]$ is an LR(1) item in the state on top of the stack. The set of such a's will always be a subset of FOLLOW(A), but it could be a proper subset, as in Example 4.51.

Example 4.54: Consider the following augmented grammar.

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & C\ C \\ C & \rightarrow & c\ C \mid d \end{array} \tag{4.55}$$

We begin by computing the closure of $\{[S' \rightarrow \cdot S, \$]\}$. To close, we match the item $[S' \rightarrow \cdot S, \$]$ with the item $[A \rightarrow \alpha \cdot B\beta, a]$ in the procedure CLOSURE. That is, $A = S'$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$, and $a = \$$. Function CLOSURE tells us to add $[B \rightarrow \cdot\gamma, b]$ for each production $B \rightarrow \gamma$ and terminal b in FIRST(βa). In terms of the present grammar, $B \rightarrow \gamma$ must be $S \rightarrow CC$, and since β is ϵ and a is \$, b may only be \$. Thus we add $[S \rightarrow \cdot CC, \$]$.

We continue to compute the closure by adding all items $[C \rightarrow \cdot\gamma, b]$ for b in FIRST$(C\$)$. That is, matching $[S \rightarrow \cdot CC, \$]$ against $[A \rightarrow \alpha \cdot B\beta, a]$, we have $A = S$, $\alpha = \epsilon$, $B = C$, $\beta = C$, and $a = \$$. Since C does not derive the empty string, FIRST$(C\$)$ = FIRST(C). Since FIRST(C) contains terminals c and d, we add items $[C \rightarrow \cdot cC, c]$, $[C \rightarrow \cdot cC, d]$, $[C \rightarrow \cdot d, c]$ and $[C \rightarrow \cdot d, d]$. None of the new items has a nonterminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial set of items is

$$\begin{array}{ll} I_0: & S \rightarrow \cdot S, \$ \\ & S \rightarrow \cdot CC,\ \$ \\ & C \rightarrow \cdot cC,\ c/d \\ & C \rightarrow \cdot d,\ c/d \end{array}$$

The brackets have been omitted for notational convenience, and we use the notation $[C \to \cdot cC,\ c/d]$ as a shorthand for the two items $[C \to \cdot cC,\ c]$ and $[C \to \cdot cC,\ d]$.

Now we compute GOTO(I_0, X) for the various values of X. For $X = S$ we must close the item $[S' \to S\cdot,\ \$]$. No additional closure is possible, since the dot is at the right end. Thus we have the next set of items

$$I_1: \quad S' \to S\cdot,\ \$$$

For $X = C$ we close $[S \to C\cdot C,\ \$]$. We add the C-productions with second component \$ and then can add no more, yielding

$$\begin{aligned} I_2: \quad & S \to C\cdot C,\ \$ \\ & C \to \cdot cC,\ \$ \\ & C \to \cdot d,\ \$ \end{aligned}$$

Next, let $X = c$. We must close $\{[C \to c\cdot C,\ c/d]\}$. We add the C-productions with second component c/d, yielding

$$\begin{aligned} I_3: \quad & C \to c\cdot C,\ c/d \\ & C \to \cdot cC,\ c/d \\ & C \to \cdot d,\ c/d \end{aligned}$$

Finally, let $X = d$, and we wind up with the set of items

$$I_4: \quad C \to d\cdot,\ c/d$$

We have finished considering GOTO on I_0. We get no new sets from I_1, but I_2 has goto's on C, c, and d. For GOTO$(I_2,\ C)$ we get

$$I_5: \quad S \to CC\cdot, \$$$

no closure being needed. To compute GOTO$(I_2,\ c)$ we take the closure of $\{[C \to c\cdot C,\ \$]\}$, to obtain

$$\begin{aligned} I_6: \quad & C \to c\cdot C,\ \$ \\ & C \to \cdot cC,\ \$ \\ & C \to \cdot d,\ \$ \end{aligned}$$

Note that I_6 differs from I_3 only in second components. We shall see that it is common for several sets of LR(1) items for a grammar to have the same first components and differ in their second components. When we construct the collection of sets of LR(0) items for the same grammar, each set of LR(0) items will coincide with the set of first components of one or more sets of LR(1) items. We shall have more to say about this phenomenon when we discuss LALR parsing.

Continuing with the GOTO function for I_2, GOTO(I_2, d) is seen to be

$$I_7: \quad C \to d\cdot, \ \$$$

Turning now to I_3, the GOTO's of I_3 on c and d are I_3 and I_4, respectively, and GOTO(I_3, C) is

$$I_8: \quad C \to cC\cdot, \ c/d$$

I_4 and I_5 have no GOTO's, since all items have their dots at the right end. The GOTO's of I_6 on c and d are I_6 and I_7, respectively, and GOTO(I_6, C) is

$$I_9: \quad C \to cC\cdot, \ \$$$

The remaining sets of items yield no GOTO's, so we are done. Figure 4.41 shows the ten sets of items with their goto's. □

To appreciate the new definition of the CLOSURE operation, in particular, why b must be in FIRST(βa), consider an item of the form $[A \to \alpha \cdot B\beta, a]$ in the set of items valid for some viable prefix γ. Then there is a rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$, where $\gamma = \delta\alpha$. Suppose βax derives terminal string by. Then for each production of the form $B \to \eta$ for some η, we have derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$. Thus, $[B \to \cdot\eta, b]$ is valid for γ. Note that b can be the first terminal derived from β, or it is possible that β derives ϵ in the derivation $\beta ax \underset{rm}{\overset{*}{\Rightarrow}} by$, and b can therefore be a. To summarize both possibilities we say that b can be any terminal in FIRST(βax), where FIRST is the function from Section 4.4. Note that x cannot contain the first terminal of by, so FIRST(βax) = FIRST(βa). We now give the LR(1) sets of items construction.

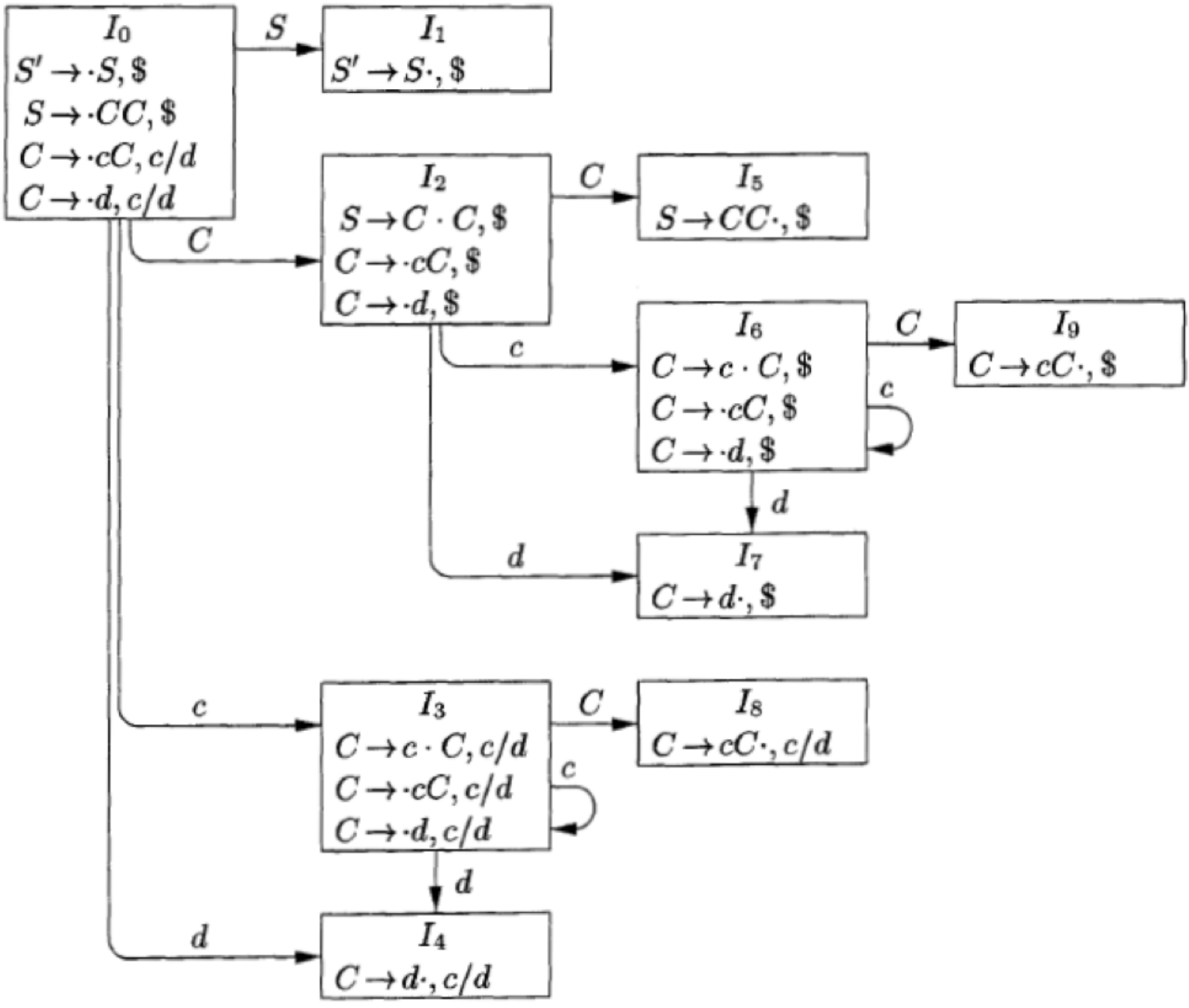


Figure 4.41: The GOTO graph for grammar (4.55)

Algorithm 4.56: Construction of canonical-LR parsing tables.

INPUT: An augmented grammar G'.

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

METHOD:

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

2. State i of the parser is constructed from I_i. The parsing action for state i is determined as follows.

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

   (b) If $[A \to \alpha \cdot, a]$ is in I_i, $A \neq S'$, then set ACTION$[i, a]$ to "reduce $A \to \alpha$."

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

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

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO$(I_i, A) = I_j$, then 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 $[S' \to \cdot S, \$]$.

STATE	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure 4.42: Canonical parsing table for grammar (4.55)

Example 1)
Given the following grammar.

1. $S \rightarrow A$
2. $S \rightarrow xb$
3. $A \rightarrow aAb$
4. $A \rightarrow B$
5. $B \rightarrow b$

Compute the LR(1) items & the corresponding DFA, also construct the parsing table.

Augment the grammar & find First Set:

0. $S' \rightarrow S$
1. $S \rightarrow A \mid xb$
2. $A \rightarrow aAb \mid B$
3. $B \rightarrow b$

$First(S') = \{x, a, b\}$
$First(S) = \{x, a, b\}$
$First(A) = \{a, b\}$
$First(B) = \{b\}$

0. $S' \rightarrow S$
1. $S \rightarrow A$
2. $S \rightarrow xb$
3. $A \rightarrow aAb$
4. $A \rightarrow B$
5. $B \rightarrow b$

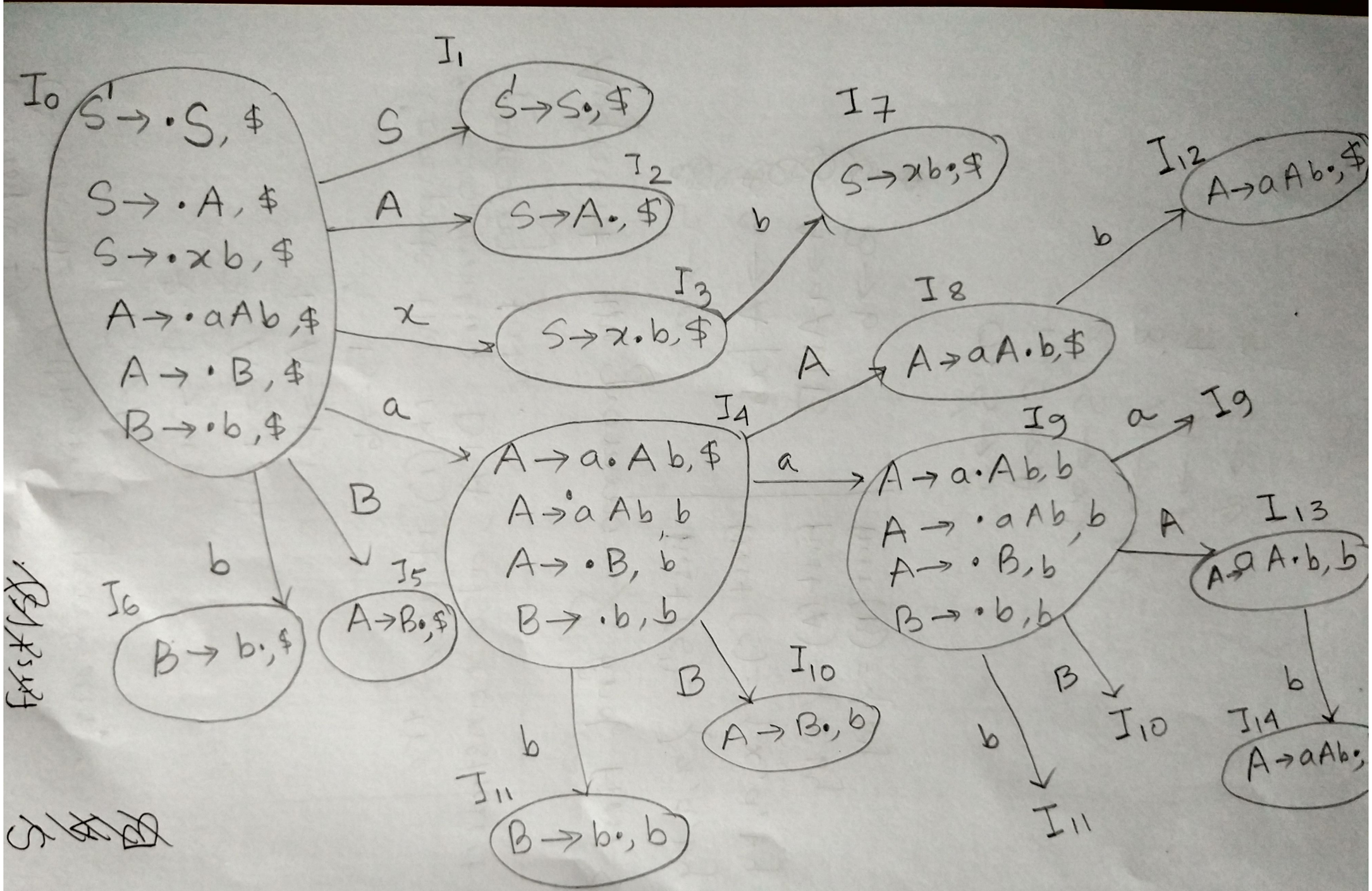


Fig: LR(1) automation

	ACTION (i, a)				GOTO (i, A)		
State	x	a	b	$	S	A	B
0	S3	S4	S6		1	2	5
1				accept			
2				R1			
3			S7				
4		S9	S11			8	10
5				R4			
6				R5			
7				R2			
8			S12				
9		S9	S11				
10			R4			13	10
11			R5				
12				R3			
13			S14				
14			R3				