Chapter 5 Bottom-Up Parsing

❑**Study Goals:**
- Master
  LR(0) parsing, SLR(1) parsing
- Understand:
  Right sentential form, Variable prefix, Handle
- Know
  LALR(1) parsing, LR(1) parsing

5.1 Overview of Bottom-Up Parsing

We will talk about:

1 The Main Idea of Bottom-Up Parsing
2 The implementation of Bottom-Up Parsing
3 Characters of Bottom-Up Parse

---

1 The Main Idea of Bottom-Up Parsing

❑Definition

Parsing begins with the input string, by steps of reduction, tries to reduce the input string to the start symbol of the grammar

The construction of the parse tree:

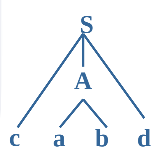
The input string are leaves of the parse tree, parsing works up towards the root, which is the start symbol

---

Example G： S → cAd
A → ab
A → a

Bottom-up parsing of string “cabd”

the process of reduction： cabd |-cAd |-S



using “|-” represents a reduction step

❑**The Key of Bottom-Up Parsing**

two processes of reduction for “cabd”

S→cAd A→ab A→a

(1)cabd|-cAd|-S can reduce to S

(2)cabd|-cAbd can’t reduce to S

- In each reduction step, a particular substring matching the right hand side of a production is replaced by the left hand structure name of the production
- The key of bottom-up parsing is how to determine the substring for reduction

2 Implementation of Bottom-Up Parsing

- Parsing Stack
  - A bottom-up parser uses an explicit stack to perform a parse
  - A schematic for bottom-up parsing is:

Parsing stack	Input	Action
$	InputString$	
…	…	
$StartSymbol	$	accept

- Stack in top-down and bottom-up parsing
  - Top-down parsing stack contains tokens and nonterminals, while bottom-up parsing stack contains tokens, nonterminals and states
  - Top-down parsing stack stores symbols waiting for matched in the parsing, while bottom-up parsing stack stores the symbols having been matched

- Implementation of bottom-up parsing
  - Base on the stack content and use the next token in the input as a lookahead to determine the next action to be performed
  - A bottom-up parser has two possible actions
    1. Shift: shift a terminal from the front of the input to the top of the stack
    2. Reduce: reduce a string α at the top of the stack to a nonterminal A, given the production A->α

So bottom-up parsing is also called shift-reduce parsing

Example G: S->aAcBe A->b A->Ab B->d

Bottom-up parsing for string "abbcde$" is :

	Stack	Input	Action
1	$	abbcde$	shift
2	$a	bbcde$	shift
3	$a<u>b</u>	bcde$	reduce A→b
4	$aA	bcde$	shift
5	$a<u>Ab</u>	cde$	reduce A→Ab
6	$aA	cde$	shift
7	$aAc	de$	shift
8	$aAc<u>d</u>	e$	reduce B→d
9	$aAcB	e$	shift
10	$<u>aAcBe</u>	$	reduce S→aAcBe
11	$S	$	accept

- Explanation
  - A bottom-up parser may need to look deeper into the stack than just the top in LL(1) parsing to determine what action to perform
  - So we will use state to denote the content in the stack

- The key of implementation
  - Determine when to shift and when to reduce?
  - Different determine methods result in different shift-reduce parsers of varying power and complexity

3 Characters of Bottom-Up Parse

1) Right Sentential Form
2) Viable Prefix
3) Handle

1) Right Sentential Form

- A shift-reduce parser traces out a rightmost derivation of the input string in reverse order

For example
S->aAcBe A->b A->Ab B->d
Rightmost derivation of “abbcde” is:
S=>aAcBe=>aAcde=>aAbcde=>abbcde
Shift-reduce parsing process is
abbcde |- aAbcde |- aAcde |- aAcBe |- S

- Each of the intermediate strings in rightmost derivation is called a right sentential form

abbcde |- aAbcde |- aAcde |- aAcBe |- S

- Each right sentential form is split between the parsing stack and the input

	Stack	Input	Action
1	$	abbcde$	shift
2	$a	bbcde$	shift
3	$ab	bcde$	reduce A→b
4	$aA	bcde$	shift
5	$aAb	cde$	reduce A→Ab
6	$aA	cde$	shift

- Right sentential form and shift-reduce parsing

A shift-reduce parser will shift terminals from the input to the stack until it is possible to perform a reduction to obtain the next right sentential form

2) Viable Prefix

- The sequence of symbols on the parsing stack is called a viable prefix of the right sentential form

	Stack	Input	Action
6	$aA	cde$	shift
7	$aAc	de$	shift
8	$aAcd	e$	reduce B→d

“aAcde” is a right sentential form, it is split between the parsing stack and the input in step 6,7 and 8

aA, aAc, aAcd are all viable prefix of aAcde

- Viable prefix and shift-reduce parsing

As long as the content of the parsing stack is a viable prefix of a right sentential form, the shift-reduce parsing is correct

3) Handle

- The string matches the right-hand side of the production that is used in the next reduction
- Together with the position in the right sentential form where it occurs
- And the production used to reduce it

is called the handle of the right sentential form

For example

In the right sentential form "abbcde", the handle is the string consisting of the leftmost "b", together with the production A->b

- Handle and shift-reduce parsing
- Determining the next handle in a parse is the main task of a shift-reduce parser
- When the next handle is on the top of the stack, action "reduce " is taken
- When the next handle has not formed on the top of the stack, action "shift" is taken

	Stack	Input	Action
1	$	abbcde$	shift
2	$a	bbcde$	shift
3	$ab	bcde$	reduce A→b
4	$aA	bcde$	shift
5	$aAb	cde$	reduce A→Ab
6	$aA	cde$	shift
7	$aAc	de$	shift
8	$aAcd	e$	reduce B→d
9	$aAcB	e$	shift
10	$aAcBe	$	reduce S→aAcBe
11	$S	$	accept

"b" is the handle of "abbcde"
"Ab" is the handle of "aAbcde"
"d" is the handle of "aAcde"
"aAcBe" is the handle of "aAcBe"

- A handle of a string is a substring that matches the right hand side of a production, and whose reduction to the nonterminal on the left hand side of the production represents one step along the reverse of a rightmost derivation.

For example

S->aAcBe A->b A->Ab B->d

S=>aAcBe=>aAcde=>aAbcde=>abbcde

d is the handle of "aAcde"

- That is, if $S \Rightarrow^*_{(rm)} \alpha A w \Rightarrow_{(rm)} \alpha\beta w$, then β in the position following α and $A \rightarrow \beta$ is a handle of $\alpha\beta w$, w to the right of the handle contains only terminal symbols

  there are three conditions for a handle:

1) $\alpha\beta w$ is a right-sentential form
2) $S \Rightarrow^*_{(rm)} \alpha A w$
3) $A \rightarrow \beta$ is a production

- Usually we say "the substring β is a handle of $\alpha\beta w$" for short

Example

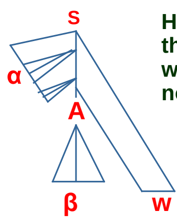
G[E]: E→E+T | T
T→T*F | F
F→(E) | id

The rightmost derivation of "T*F+id" is

E=>E+T=>E+F=>E+id=>T+id=>T*F+id

so T*F is the handle of T*F+id

❑Handle β in the parse tree of a right-sentential form αβw

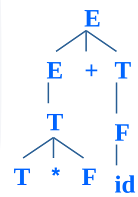


Handle is the leaves of the leftmost subtree which consisting of a node and its children

Example

G[E]:E→E+T|T T→T*F|F F→(E)|id

The parse tree of "T*F+id" is



- "T*F" is the leaves of the leftmost complete subtree consisting of a node and its children
- so "T*F" is the handle of T*F+id

❑Viable Prefix and Handle

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

  Right sentential form aAcde

  (where d is handle)

  Viable prefixes are: a，aA，aAc, aAcd

❑Characters of Bottom-Up Parse taking a view of implementation

- Parser keeps putting viable prefixes in the stack
- Until handle is on the top of the stack, reduction is to take place
- As long as the content of the stack is a viable prefix, paring is correct

❑Characters of Bottom-Up Parse in general

- Bottom-up parse is in general more powerful than top-down parse, it can be used to parse virtually all programming language
- The constructions involved in this parse are also more complex. Indeed, all of the important bottom-up methods are really too complex for hand coding

5.2 Overview of LR Parsing Method

- There are many Bottom-Up parsing methods, we will only talk about LR parsing method
- In LR Parsing we will talk about

  LR(0) parsing

  SLR(1) parsing

  LALR(1) parsing

  LR(1) parsing

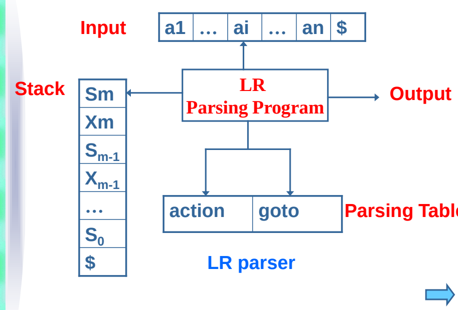
1 LR(K) Parsing

Basing on the string on top of the parsing stack (represented as state) and using the next K(K≥0) tokens in the input as lookahead to determine handle for reduction

- The Meaning of "LR(K)"
  - L indicates that the input is processed from left to right
  - R indicates that a rightmost derivation is produced
  - K for the number of input symbols of lookahead that are used in making parsing decisions

- The consequence of the power of LR(k) parsing
  - LR(0) parsing, where no lookahead is consulted in making parsing decisions
  - SLR(1) parsing(simple LR(1)) is an improvement on LR(0)
  - LALR(1) parsing(lookahead LR(1)) is slightly more powerful than SLR(1) but less complex than general LR(1)
  - LR(1) parsing is the most powerful and most complex

2 Schematic Form of LR parser



3. Parsing Table

Each line represents a state, each column is a grammar symbol

	ACTION						GOTO								
	a	c	e	b	d	$	a	c	e	b	d	$	S	A	B
0	s						2						1		
1						acc									
2		s		s				1		3					
3	r2	r2	r2	r2	r2	r2									

- GOTO: take a state and a grammar symbol, determine the next state
- ACTION: take a state and a grammar symbol, determine the next action to take place

To save space, compress ACTION and GOTO on columns of terminals

	ACTION						GOTO								
	a	c	e	b	d	$	a	c	e	b	d	$	S	A	B
0	s						2						1		
1						acc									
2		s		s				1		3					
3	r2	r2	r2	r2	r2	r2									

	ACTION						GOTO		
	a	c	e	b	d	$	S	A	B
0	s2						1		
1						acc			
2		s1		s3					
3	r2	r2	r2	r2	r2	r2			

ACTION Table

The table entry (action[S_m,a_i]) for state S_m and input a_i has four values:

1) Shift(s_k)

Put symbol a_i and state K into the stack

2) Reduction(r_k)

Reduce by number k production (A->γ), the action includes:

- Pop the string γand all of its corresponding states from the stack. Suppose currently the top of stack is state Si
- Push A onto stack
- Push the state Sj =GOTO[Si,A] onto stack

3) Accept

Indicate that parsing is complete successfully

4) Error

Indicate that parsing has discovered an error

Example of LR Parsing (SLR(1))

G[S]: (1)S->aAcBe (2)A->b (3)A->Ab (4)B->d

LR Parsing for string "abbede$"

Parsing table is:

	ACTION						GOTO		
	a	c	e	b	d	$	S	A	B
0	s2						1		
1						acc			
2				s4				3	
3		s5		s6					
4		r2		r2					
5					s8				7
6		r3		r3					
7			s9						
8			r4						
9						r1			

LR parsing process of "abbcde$"

(1)S->aAcBe (2)A->b (3)A->Ab (4)B->d

	Stack	Input	Action	Goto
1	$0	abbcde$	s2	
2	$0a2	bbcde$	s4	
3	$0a2b4	bcde$	r2	3
4	$0a2A3	bcde$	s6	
5	$0a2A3b6	cde$	r3	3
6	$0a2A3	cde$	s5	
7	$0a2A3c5	de$	s8	
8	$0a2A3c5d8	e$	r4	7
9	$0a2A3c5 B 7	e$	s9	
10	$0a2A3c5B7e9	$	r1	1
11	$0 S 1	$	acc	

Summarization of LR parsing method

- Parsing Program is the same for all LR parsers, only parsing table changes from one parser to another
- How can we construct a parsing table for different grammars and different parsers? This is the key of LR parser

5.3 Finite Automata of LR(0) Items and LR(0) Parsing

- LR(0) Parsing
- The LR parser using LR(0) parsing table is LR(0) parser; The grammar for which an LR(0) parser can be constructed is said to be LR(0) grammar
- LR(0) parser uses only the content of stack to determine handle, it doesn't need input token as lookahead
- Almost all "real" grammars are not LR(0), but LR(0) method is a good starting point for studying LR parsing

5.3.1 LR(0) Items

LR(0) Item

- A LR(0) item of a grammar G is a production of G with a distinguished position in its right-hand side
- For example, production U→XYZ has four items

  [0] U→ • XYZ　　[1] U→X • YZ

  [2] U→XY • Z　　[3] U→XYZ •

  production A→εhas only one item A→•
- These are called LR(0) items because they contain no explicit reference to lookahead

Why do we need to construct items?

- Handle is the right hand side of a production
- The rightmost position of the handle string is on the top of the stack when reduction takes place
- Thus, it seems plausible that parser determines its actions based on positions in right hand sides of productions
- When these positions reach the right-hand end of a production, then this production is a candidate for a reduction, and it is possible that the handle is at the top of the stack

The Meaning of Items

An item records an intermediate step in the recognition of the right-hand side of a production

- A→ • α means that we may be about to recognize an A by using production A→ α
- A→β • γ means that βhas already been seen (β must appear at the top of stack) and that it may be possible to derive the next input token from γ
- A→α • means that αnow resides on the top of the stack and may be the handle, if A->αis to be used for the next reduction

Categories of Items

- **Initial Item**

  Item of the form A→ • α, means the initial of recognizing α
- **Complete Item**

  Item of the form A→α • , means the completeness of recognizing α

5.3.2 Finite Automata of Items

The LR(0) items can be used as the states of a finite automaton that maintains information about the parsing stack and the progress of a shift-reduce parse

1 Construct NFA of Items

2 Construct DFA of Sets of Items Directly

3 LR(0) Parsing with DFA

1 Construct NFA of Items

1) Transitions of NFA

Consider item i: A→α•Xβ and item j: A→αX•β, there is a transition on symbol X from i to j

$$A\to\alpha\bullet X\beta \xrightarrow{X} A\to\alpha X\bullet\beta$$

- If X is a token

This transition corresponds to a shift of X from the input to the top of the stack during a parse

- If $X \in V_N$
  - This transition corresponds to the pushing of X onto the stack
  - But X will never appear as an input symbol, this can only occur during a reduction by a production X→r
  - So for each production choice X→r of X, we must add an ε-transition to X->• r

$$A\to\alpha\bullet X\beta \xrightarrow{\varepsilon} X\to\bullet\, r$$

2) Start State of NFA

- Augment the grammar by a single production S'->S, where S' is a new nonterminal, it becomes the start symbol of the Augmented grammar

  Since start symbol S may appear in the right hand side of productions, the purpose of augmenting is to indicate when the parser should stop parsing and announce acceptance of the input

- S'-> •S becomes the start state of the NFA

3) Accepting State of NFA

- NFA is used to keep track of the state of a parse, not to recognize strings outright
- Thus, the parser itself will decide when to accept, and the NFA has no accepting states at all

Example G E→aA | bB

A→cA | d

B→cB | d

Augmented grammar G':

S'→E

E→aA | bB

A→cA|d

B→cB | d

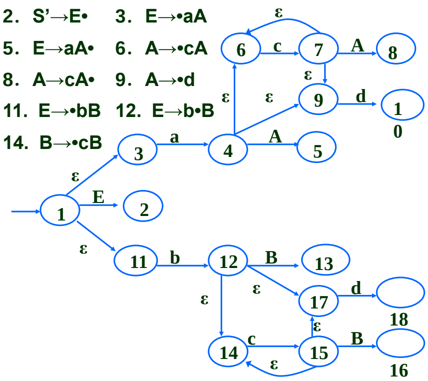
All items of G':

1. S'→•E	2. S'→E•	3. E→•aA
4. E→a•A	5. E→aA•	6. A→•cA
7. A→c•A	8. A→cA•	9. A→•d
10. A→d•	11. E→•bB	12. E→b•B
13. E→bB•	14. B→•cB	15. B→c•B
16. B→cB•	17. B→•d	18. B→d•

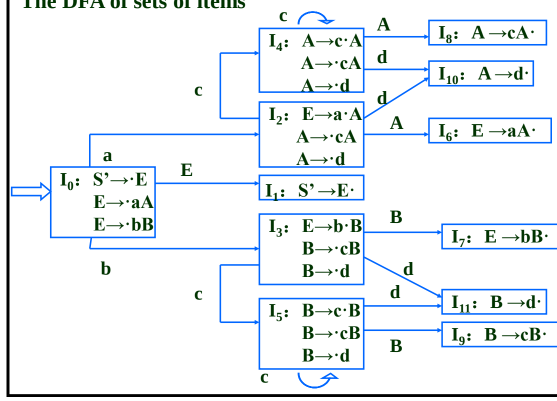
All items of G' :

1. S'→•E
2. S'→E•
3. E→•aA
4. E→a•A
5. E→aA•
6. A→•cA
7. A→c•A
8. A→cA•
9. A→•d
10. A→d•
11. E→•bB
12. E→b•B
13. E→bB•
14. B→•cB
15. B→c•B
16. B→cB•
17. B→•d
18. B→d•

The NFA of items



The DFA of sets of items



- This method starts out as a NFA of items, from this NFA construct the DFA of sets of items using the subset construction
- It is too complex, not utility
- It is much easier to construct the DFA of sets of items directly

2 Construct DFA of Sets of Items Directly

The Construction of DFA

- Each state of DFA is a set of items
- How to construct a state?----closure operation
- How to construct transition form one state to another?----goto operation

1) The Closure Operation

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

a) Initially, each item of I is added to closure(I)

b) If A->α•Bβis in closure(I) and B $\in V_N$, then for each production B→r, add the item B→•r to closure(I), if it is not there.

c) Repeat b) until no more new items are added

For each item where ' • ' is at the right end or followed by a terminal, closure of this item is the item itself

Example G: S'->E E->aA | bB
A->cA|d B->cB | d

if I={ S'→•E } then

closure(I)={ S'->•E, E->•aA, E->•bB }

Make Clear

Item A->α•Bβin closure(I) indicates that, at some point in the parsing process, the next seeing substring is derivable from Bβ. If B→r, the next seeing substring is derivable from r at this point. For this reason B→•r is in closure(I)

- Distinction of Items in the state of DFA:
  - Kernel items

    Which include the S'-> • S and all items whose dots are not at the left end
  - Closure items

    Which have their dots at the left end, they are added to the state during the closure operation

2) The Goto Operation

I is a set of items, $X \in V_N \cup V_T$

goto(I,X)= closure(J)

where J is the set of all items [A→αX•β] such that [A→α• Xβ] is in I

Example

S'→E E→aA | bB A→cA|d B→cB | d

I = { S'→•E, E→•aA, E→•bB }

goto(I, E)=closure({ S'→E•})={ S'→E•}

goto(I,a)= closure({E'→a•A})

={E'→a•A,A→•cA,A→•d }

goto(I, b)= closure({ E→b•B })

={ E→b•B,B→•cB,B→•d }

3) The Construction of DFA

a) IS_0=closure({S'→•S}) is the start state of DFA, and it is unlabeled

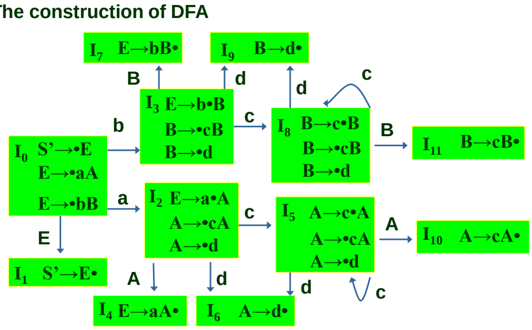
b) Get an unlabeled state IS_i from DFA
   - Label IS_i
   - For each item U->x•Ry($R \in V_N \cup V_T$,x and y are strings) of IS_i, compute goto(IS_i,R)=IS_j
   - Add IS_j to DFA as unlabeled if it is not there
   - Add a transition form IS_i to IS_j on R

c) Repeat b) until there is no unlabeled state in DFA

Example: augmented grammar G':

S'→E E→aA | bB A→cA|d B→cB | d

The construction of DFA



3 LR(0) Parsing with DFA

- LR(0) parsing algorithm depends on keeping track of the current state in the DFA of sets of items
- We do this by pushing the new state number onto the parsing stack after each push of a symbol
- Let *s* be the current state(at the top of the parsing stack).Then actions are defined as follows:

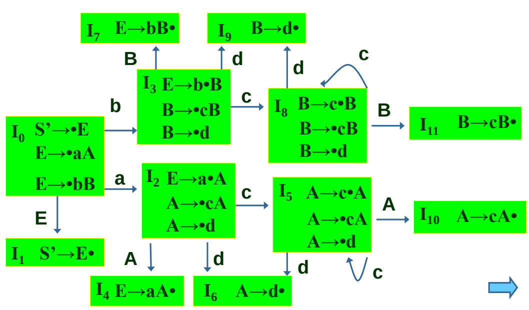
1. If state s contains any item of the form A->α· Xβ,where X $\in V_T$
   - Then the action is to shift the current input token onto the stack.
   - If this token is X, then the new state to be pushed on the stack is the state containing the item A->α X · β
   - If this token is not X, an error is declared
2. If state s contains S’->S· , where S is the start symbol
   - Then the action is to accept, provide the input token is ‘$’
   - And error if the input is not ‘$’

3. If state s contains any complete items (A->γ·)
   - Then the action is to reduce by the rule A->γ
   - The new state is computed as follows.
   - Remove the string γand all of its corresponding states from the stack.
   - Correspondingly, back up in the DFA to the state from which the construction of γbegan, this state must contain an item of the form B->α · Aβ
   - Push A onto the stack, and push the state containing the item B ->α A · β

Example: augmented grammar G’:

S’→E　　E→aA | bB　　A→cA|d　　B→cB | d

LR(0) parsing of string “bccd$”



	Stack	Input	Action
1	$0	bccd$	shift
2	$0b3	ccd$	shift
3	$0b3c8	cd$	shift
4	$0b3c8c8	d$	shift
5	$0b3c8c8d9	$	reduce B->d
6	$0b3c8c8 B11	$	reduce B->cB
7	$0b3c8 B11	$	reduce B->cB
8	$0b3B7	$	reduce E->bB
9	$0 E1	$	accept

5.3.3 Constructing LR(0) Parsing Table

- LR(0) Parsing Table

	ACTION	GOTO								
		a	c	e	b	d	$	S	A	B
0	Shift	2						1		
1	R1									
2	Shift		1		3					
3	R2									

- ACTION: LR(0) parsing does not consult input token, so LR(0) parsing states are either “shift” states or “reduce” states
- GOTO: There must be a column for every symbol (V_N,V_T and $)

- Construction of LR(0) Parsing Table

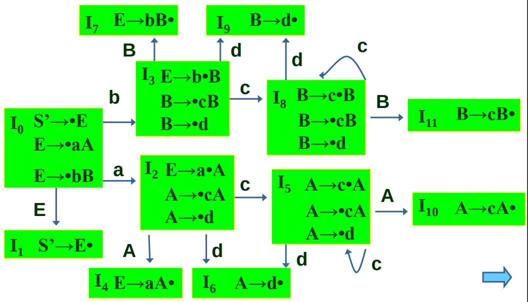
Given a grammar G, we augment G to produce G’

1. Construct DFA of sets of LR(0) items
2. The ACTION section for state K is determined as follows:
   a) If A→α•β∈K, then ACTION[K]=Shift
   b) If A→α•∈K, and the number of A→αis j, then set ACTION[K]=R_j
3. The GOTO section for state K is constructed for all symbols using the rule: If goto(K,X)=J, $X \in V_N \cup V_T \cup \{\$\}$, then set GOTO[K,X]=J

Augmented grammar G’:

(0) S’→E (1) E→aA (2) E→bB (3) A→cA
(4) A→d (5) B→cB (6) B→d

The DFA of sets of items



LR(0) parsing table

	ACTION	GOTO								
		a	c	e	b	d	$	E	A	B
0	Shift	2			3			1		
1	R0									
2	Shift		5			6			4	
3	Shift		8			9				7
4	R1									
5	Shift		5			6			10	
6	R4									
7	R2									
8	Shift		8			9				11
9	R6									
10	R3									
11	R5									

5.3.4 The LR(0) Parsing Algorithm

Let *S* be the current state (at the top of the parsing stack), actions are defined as follows:

1) If ACTION[S]=Shift, and *a* is the current input symbol then push symbol *a* and state j=GOTO[S,a] onto stack. If GOTO[S,a] is empty, an error occurs

2) ACTION[S]=Rj

- If the rule numbered j is S’->S, where S is the start symbol, then the action is to acceptance, provided the input is “$”, and error if the input is not “$”
- Otherwise the action is to reduce by the rule numbered j (A->β)
  - Remove the string β and all of its corresponding states from the stack, suppose currently the top of stack is state K
  - Push A onto stack
  - Push the state i= GOTO[K,A] onto stack

(0) S’→E (1) E→aA (2) E→bB (3) A→cA
(4) A→d (5) B→cB (6) B→d

LR(0) parsing of string “bccd$”

	Stack	Input	Action	Goto
1	$0	bccd$	Shift	3
2	$0b 3	ccd$	Shift	8
3	$0b3c 8	cd$	Shift	8
4	$0b3c8c 8	d$	Shift	9
5	$0b3c8c8d 9	$	R_6	11
6	$0b3c8c8B11	$	R_5	11
7	$0b3c8 B11	$	R_5	7
8	$0b3 B 7	$	R_2	1
9	$0 E 1	$	R0	

5.4 SLR(1) Parsing

1. Conflict in the set of items

- **Shift-Reduce Conflict**

  If a set contains shift item A→α•aβand complete item B→r•, an ambiguity arises as to whether shift ‘a’ or reduce ‘r’ to B

- **Reduce-Reduce Conflict**

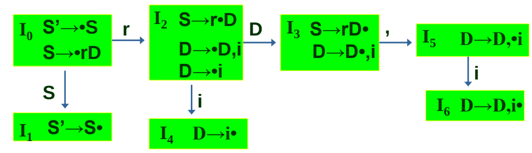
  If a set contains complete item A→β• and B→r•, an ambiguity arises as to which production to use for the reduction

A grammar is LR(0) if and only if non of the set of items has shift-reduce conflict or reduce-reduce conflict.

Example G'

(0) S'→S (1) S→rD (2) D→D,i (3) D→i

The DFA of sets of LR(0) items



In state I_3, S→rD• is a complete item, D→D•,i is a shift item, there exists shift-reduce conflict, so G' is not LR(0) grammar

2. Eliminating Conflicts in SLR(1)

- The Main Idea of SLR(1) (Simple LR(1))
  - SLR(1) parsing is a simple, effective extension of LR(0)
  - It uses the DFA of sets of LR(0) items.
  - Increasing the power of LR(0) parsing by using the next token in the input string to direct its actions
  - The simple use of lookahead is powerful enough to parse almost all practical language

- Two ways of using the lookahead token :

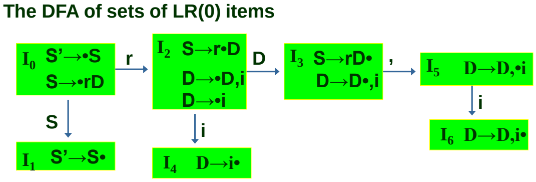
a) It consults the input token before a shift to make sure that an appropriate DFA transition exists

b) It uses the Follow set of a nonterminal to decide if a reduction should be preformed. for item A→r•, reduction only takes place when the next token a∈FOLLOW(A)

Example: augmented grammar G'

(0) S'→S (1) S→rD (2) D→D,i (3) D→i

The DFA of sets of LR(0) items



	FOLLOW
S'	$
S	$
D	$,

For state I_3

- If the next token is '$', then reduce
- If the next token is ',' , then shift

Conflict can be solved

- Eliminating Conflicts in SLR(1)
  - Example

I={X→α•bβ,A→r•,B→δ•}, where b∈V_T,

if FOLLOW(A) ∩FOLLOW(B)=ϕ and not includes b, the action of I is based on the next input token 'a'

- If a=b, then shift
- If a∈FOLLOW(A), then reduce with A→r
- If a∈FOLLOW(B), then reduce with B→δ
- Otherwise, an error occurs

➢ In general

If state I has m shift items:

$A_1\to\alpha_1\bullet a_1\beta_1, A_2\to\alpha_2\bullet a_2\beta_2, \dots, A_m\to\alpha_m\bullet a_m\beta_m$

and n reduction items:

$B_1\to r_1\bullet, B_2\to r_2\bullet, \dots, B_n\to r_n\bullet$,

$\{a_1, a_2, \dots, a_m\} \cap FOLLOW(B_1) \cap FOLLOW(B_2) \cap\dots\cap FOLLOW(B_n)=\varphi$

then the action of I is based on the next token ‘a’

- If $a\in\{a_1, a_2, \dots, a_m\}$, then shift
- If $a\in FOLLOW(B_i)$, i=1,2,…,n, then reduce with $B_i\to r_i$;
- Otherwise, an error occurs

A grammar is SLR(1) if the application of lookahead as above results in no ambiguity

3 Construction of SLR(1) Parse Table

SLR(1) Parsing Table

	ACTION						GOTO		
	a	c	e	b	d	$	S	A	B
0	S2						1		
1						acc			
2		S1		S3					
3		r2			r2				

Construction of SLR(1) Parsing Table

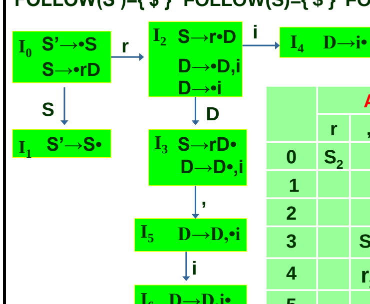
Given a grammar G, we augment G to produce G’

1. Construct DFA of sets of LR(0) items
2. The ACTION section for state K is determined as follows:
   a) If $A\to\alpha\bullet a\beta\in K$, $a\in V_T$, and goto(K,a)=J, then set ACTION[K,a]=‘S_J’
   b) If $A\to\alpha\bullet\in K$, and the number of $A\to\alpha$ is j, then set ACTION[K,a]=‘R_j’ for each $a\in Follow(A)$
   c) If $S'\to S\bullet\in K$, then set ACTION[K,$]=‘acc’

3. The GOTO section for state K is constructed for all nonterminals using the rule: If $A\to\alpha\bullet B\beta\in K$, $B\in V_N$, and goto(K,B)=J, then set GOTO[K,B]=‘J’
4. Empty entries not defined by rule 2 and 3 represent errors

Example G’: (0) S’→S (1) S→rD (2) D→D, i (3) D→i

FOLLOW(S’)={ $ } FOLLOW(S)={ $ } FOLLOW(D)={ $, }



	ACTION				GOTO	
	r	,	i	$	S	D
0	S_2				1	
1				acc		
2			S_4			3
3		S_5		r_1		
4		r_3		r_3		
5			S_6			
6		r_2		r_2		

4 The SLR(1) Parsing Algorithm

Let *S* be the current state (at the top of the parsing stack), *a* be the current input symbol. Then actions are defined as follows:

1) If ACTION[S,a]=Sj, a∈V_T, then push symbol a and state j onto stack

2) If ACTION[S,a]=Rj, a ∈V_T or $ then the action is to reduce by the rule numbered j (A->β)

- Remove the string β and all of its corresponding states from the stack, suppose currently the top of stack is state K
- Push A onto stack
- Push the state j = GOTO[K,A] onto stack

3) If ACTION[S,a]=acc, parsing is completed successfully

4) If ACTION[S,a] is empty, an error occurs

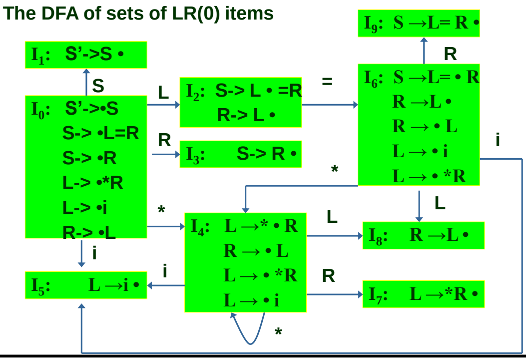
5.5 General LR(1) and LALR(1) Parsing

- There are a few situations in which SLR(1) parsing is not quite powerful enough
- This will lead us to study the more powerful general LR(1) and LALR(1) parsing

Example G':

(0) S' →S (1) S → L=R (2) S →R (3) L →*R (4) L →i (5) R →L

The DFA of sets of LR(0) items



S' →S

S →L=R |R

L →*R | i

R →L

	Follow
S'	$
S	$
L	=, $
R	$, =

I_6: S →L= • R
R →L •
R → • L
L → • i
L → • *R

Follow(R)={$,=} ∩{ *, i}=φ, so conflict in I_6 can be solved in SLR(1)

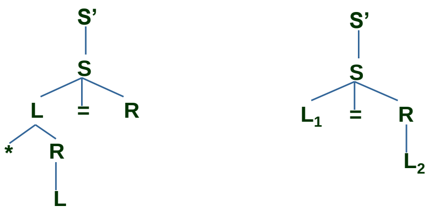
I_2: S-> L • =R
R-> L •

Follow(R)={$,=} ∩{=} ≠ φ, SLR(1) can't eliminate this conflict, G' is not SLR(1)

- SLR(1) parsing uses Follow sets of nonterminals as lookahead. This eliminate some invalidate reduction in LR(0), some conflicts in a state may be removed
- Follow set of A includes all terminals that may follow A in all sentential form, but it is not the case that A may be followed by any terminal in Follow(A) in any sentential form including A , so SLR(1) can't eliminate all conflicts

Example:S' →S S →L=R | R L →*R | i R →L

follow(R)={ $, = }



In "*L=R",L can only be reduced to R in condition that L is followed by '=' ,not '$' in Follow(R)={$,=}

In "L_1=L_2",L_2 can only be reduced to R in condition that L_2 is followed by '$' ,not '=' in Follow(R)={$,=}

5.5.1 Main Idea of General LR(1)

- Main Idea of General LR(1)
- In LR(1) parsing, lookaheads are built for distinct sentential forms
- For example

  All sentential forms that include A are:
  …αAa…, …βAb…, …γAc…, so
  FOLLOW(A)={a,b,c,…}
- For "…αA", reduction to A occurs only when the lookahead is a;
- For "…βA", reduction to A occurs only when the lookahead is b;
- For "…γA", reduction to A occurs only when the lookahead is c;

- Difference with SLR(1)
  - SLR(1) method applies lookahead after the construction of the DFA of LR(0) items.
  - LR(1) method uses a new DFA that has the lookahead build into its construction from the start.

- LR(1) Parsing

1 LR(1) item

2 The construction of the DFA of sets of LR(1) items

3 LR(1) grammar

---

1 LR(1) item

A LR(1) item is a pair consisting of a LR(0) item and a lookahead token

[A-> α•β,a] where *A-> α•β* is a LR(0) item and *a* is a lookahead token

---

2 The Construction of automation of LR(1) items

These are similar to LR(0) transactions except that they keep track of lookaheads.

❖ The transactions between LR(1) items

1) Given an LR(1) item [A-> α•Xγ,a], where X $\in V_N \cup V_T$, there is a transition on X to the item [A-> αX • γ,a]

2) Given an LR(1) item [A-> α•Bγ,a], where B $\in V_N$, there are ε-transition to item [B-> •β,b] for every production B-> β and for every token b in first(γa)

❖ Explanation

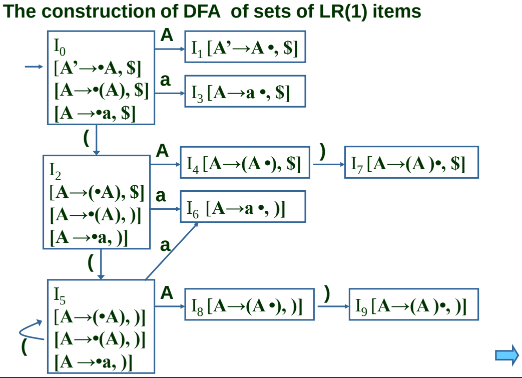
- ε-transition keep track of the context in which the structure B needs to be recognized.
- [A-> α•Bγ,a] means at this point in the parse we may want to recognize a B, but only if this B is followed by a string derivable from the string γa, and such strings must begin with a token in FIRST(γa)
- If γis ε, then there is anε-transition from [A-> α•B,a] to [B-> •β,a]
- Replacing Follow(B) as lookahead for B→ β in SLR(1), FIRST(γa) is a subset of Follow(B)

❖ Start state

- ➢ Argumenting the grammar with a new start symbol S' and a new production S'->S
- ➢ closure({[S'→•S,$]}) is the start state of DFA

Example: augmented grammar G': A'→A A→(A) | a

The construction of DFA of sets of LR(1) items



3 LR(1) grammar

A grammar is LR(1) grammar if and only if, for any state s, the following two conditions are satisfied:

1) For any item [A-> α•X β,a] in s with X $\in V_T$, there is no item in s of the form [B->γ•,X] (otherwise there is a shift-reduce conflict).

2) There are no two items in s of the form [A-> α•,a] and [B-> β •,a] (otherwise there is a reduce-reduce conflict).

❑ Characters of LR(1) parsing

- ➢ Lookaheads are precise in LR(1) parsing, it overcomes the problem with SLR(1) , eliminates all invalidate reductions
- ➢ But at a cost of substantially increased complexity. General LR(1) parsing is usually considered too complex to use in the construction of parsers in most situations

5.5.2 Main Idea of LALR(1)

❑ LALR(1) (“lookahead” LR parsing)

LALR(1) retains some of the benefit of general LR(1) parsing over SLR(1) parsing, while preserving the smaller size of the DFA of LR(0) items

❑ LALR(1) parsing

- The core of a state of the DFA of LR(1) items is the set of LR(0) items
- LALR(1) parsing identifies all the states that have the same core and combines their lookaheads
- In doing so, we end up with a DFA which size is identical to the DFA of LR(0) items
- If there is no conflict in any state of DFA after combining, then it is the DFA of LALR(1)

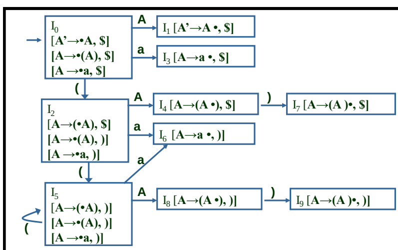
I_2 and I_5 , I_3 and I_6 , I_4 and I_8, I_7 and I_9 are states with the same core

I_2 [A→(•A), $] [A→•(A),)] [A →•a,)]	I_5 [A→(•A),)] [A→•(A),)] [A →•a,)]	⇒	$I_{2,5}$ [A→(•A), $/)] [A→•(A),)] [A →•a,)]
I_3 [A→a •, $]	I_6 [A→a •,)]	⇒	$I_{3,6}$ [A→a •, $/)]
I_4 [A→(A •), $]	I_8 [A→(A •),)]	⇒	$I_{4,8}$ [A→(A •), $/)]
I_7 [A→(A)•, $]	I_9 [A→(A)•,)]	⇒	$I_{7,9}$ [A→(A)•, $/)]

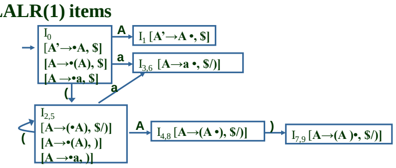
There is no conflict after combining, so it is the DFA of LALR(1)

❑ Two principles for LALR(1) parsing construction

- The core of a state of the DFA of LR(1) items is a state of the DFA of LR(0) items.
- Given two state s1 and s2 of the DFA of LR(1) items that have the same core, suppose there is a transition on the symbol X from s1 to a state t1. Then there is also a transition on X from s2 to t2, and the states t1 and t2 have the same core.



DFA of sets of LALR(1) items



❑ Explanation

- In the case of complete items, the lookahead sets of LALR(1) states are often smaller than the corresponding Follow sets.
- It is possible for the LALR(1) construction to create paring conflicts that do not exit in general LR(1) parsing, but this rarely happens in practice.