Chapter 6:
Bottom-Up Parsing (Shift-Reduce)

陳奇業 成功大學資訊工程系

Objectives of Bottom-Up Parsing

- attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top). i.e., reduce a string w to the start symbol of a grammar. At each reduction step a particular substring matching the right side of a production (grammar rule) is replaced by the left nonterminal symbol. A rightmost derivation is traced out in reverse.

An Example

Grammar :

S -> aABe

A -> Abc | b

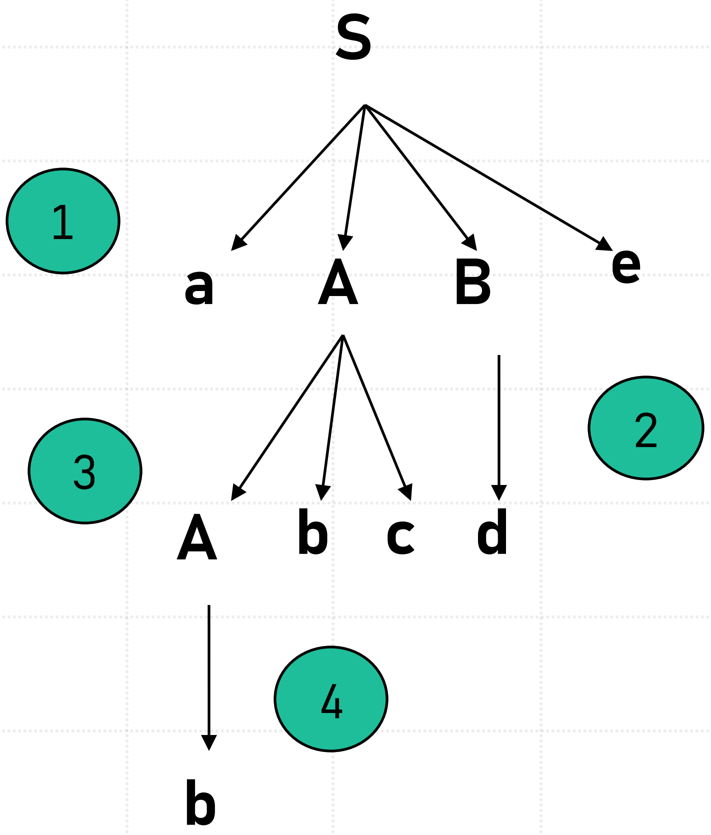
B -> d

w = **abbcde**

$S \overset{rm}{\Rightarrow} aABe \overset{rm}{\Rightarrow} aAde \overset{rm}{\Rightarrow} aAbcde \overset{rm}{\Rightarrow} abbcde$ (rightmost derivation)

LR parsing:

abbcde ==> aAbcde ==> aAde ==> aABe ==> S (rightmost derivation in reverse)



LR parsing:

abbcde ==> aAbcde ==> aAde ==> aABe ==> S

Stack Implementation of Bottom-Up Parsing

- There are four actions a parser can make:

(1) shift (2) reduce (3) accept (4) error.

- There is an important fact that justifies the use of a

stack in shift-reduce parsing: the handle will always eventually appear on top of the stack, never inside.

Initially, (stack) \$ w\$ (input buffer)

Finally, (stack) \$S \$ (input buffer) // S is a start symbol of grammar G

Example 5.1 Consider the following augmented grammar for balanced parentheses:

$$S' \rightarrow S$$
$$S \rightarrow (\ S\ )\ \ S \mid \varepsilon$$

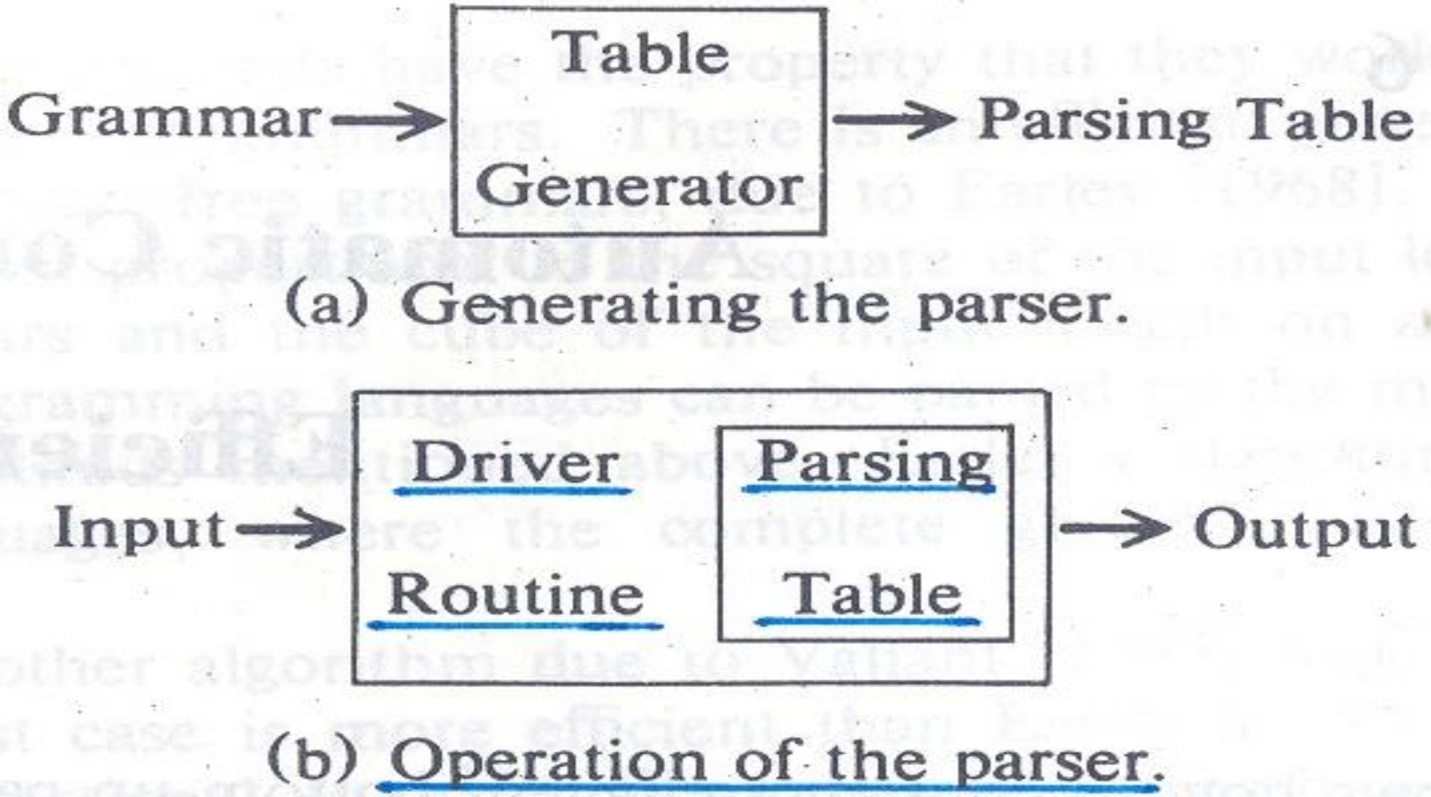
A bottom-up parse of the string `()` using this grammar is given in Table 5.1.

Table 5.1

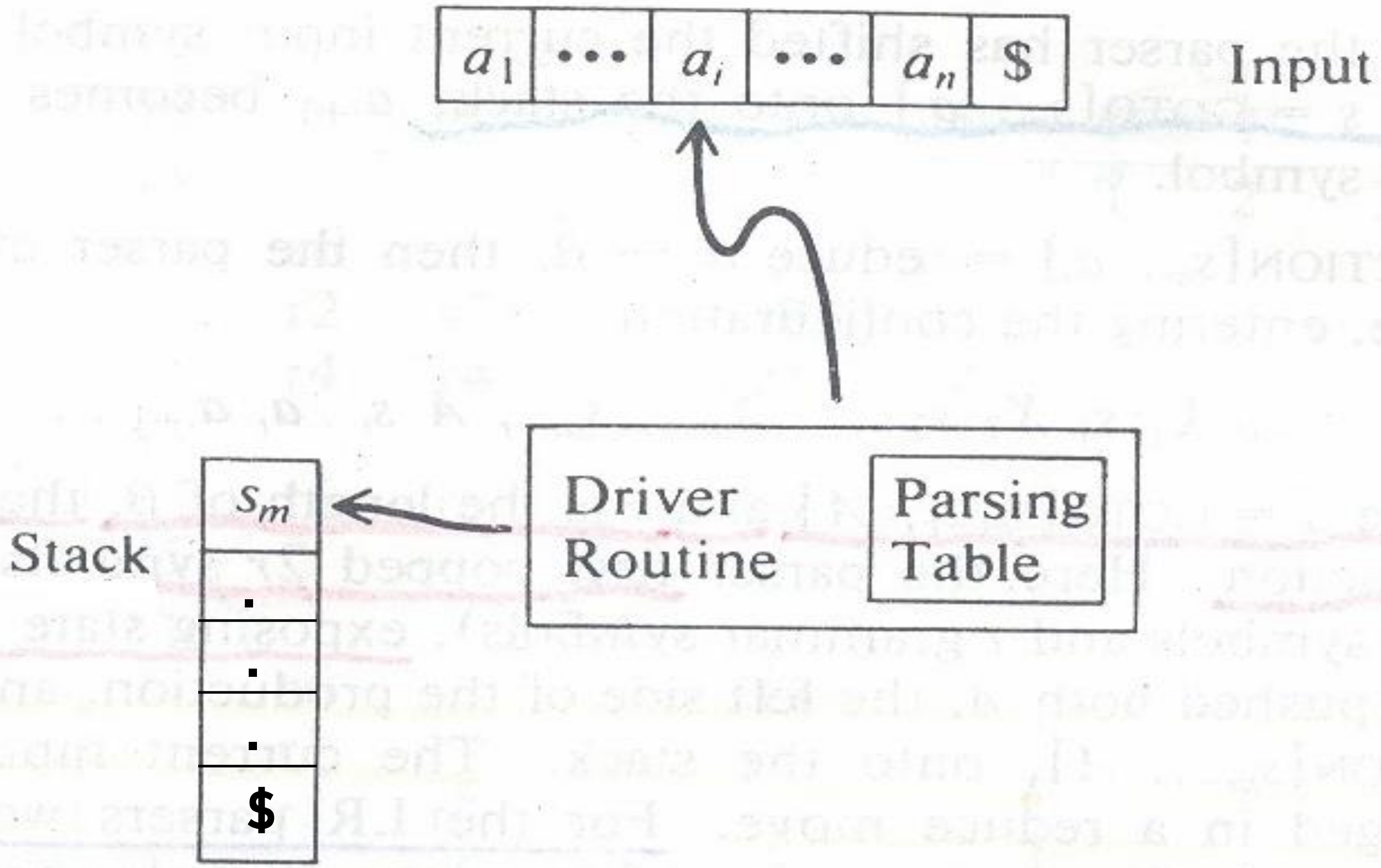
Parsing actions of a bottom-up parser for the grammar of Example 5.1

	Parsing stack	Input	Action
1	$	( ) $	shift
2	$ (	) $	reduce $S \rightarrow \varepsilon$
3	$ (S	) $	shift
4	$ (S)	$	reduce $S \rightarrow \varepsilon$
5	$ (S) S	$	reduce $S \rightarrow (\ S\ )\ S$
6	$ S	$	reduce $S' \rightarrow S$
7	$ S'	$	accept

§



Generating an LR parser.



LR parser

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			

Parsing table.

	Stack	Input
(1)	$ 0	id * id + id $
(2)	$ 0 id 5	* id + id $
(3)	$ 0 *F* 3	* id + id $
(4)	$ 0 *T* 2	* id + id $
(5)	$ 0 *T* 2 * 7	id + id $
(6)	$ 0 *T* 2 * 7 id 5	+ id $
(7)	$ 0 *T* 2 * 7 *F* 10	+ id $
(8)	$ 0 *T* 2	+ id $
(9)	$ 0 *E* 1	+ id $
(10)	$ 0 *E* 1 + 6	id $
(11)	$ 0 *E* 1 + 6 id 5	$
(12)	$ 0 *E* 1 + 6 *F* 3	$
(13)	$ 0 *E* 1 + 6 *T* 9	$
(14)	$ 0 *E* 1	$

Moves of LR parser on **id * id + id**.

Handles

- A **substring** that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation. However, in many cases the leftmost substring 'β' that matches the right side of some production A -> β is not a handle, because a reduction by the production yields a string that cannot be reduced to the start symbol.

Handles (Continued)

- A handle of a right sentential form γ is a production A -> β 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. i.e.,

  $S \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha \beta w$, then A -> β in the position following α is a handle of $\alpha\beta w$. The string w to the right of the handle contains only terminal symbols.

Handle = leftmost complete subtree.

Handle Pruning

- A rightmost derivation in reverse can be obtained by "handle pruning".
- Two Problems:

1. To locate the substring to be reduced in right-sentential form.
2. To determine the production with the same substring on the right-hand side to be chosen.

Assignment #4

- Write a LL parser in ? and a LR parser in Yacc separately for the TINY language defined in Fig. 3.6. The parsers will parse any input legal TINY program and generate a parse tree for it. Use the program in Fig. 3.8 to test your parsers and turn in the tested results with your parser codes.

Viable Prefixes

- The set of prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser are called viable prefixes.

* use table generators, i.e., take grammar and produce parsing table

Example 5.2 Consider the following augmented grammar for rudimentary arithmetic expressions (no parentheses and one operation):

$$E' \rightarrow E$$
$$E \rightarrow E \mathbf{+} \mathbf{n} \mid \mathbf{n}$$

A bottom-up parse of the string $\mathbf{n} \mathbf{+} \mathbf{n}$ using this grammar is given in Table 5.2.

Table 5.2
Parsing actions of a bottom-up parser for the grammar of Example 5.2

	Parsing stack	Input	Action
1	$	**n + n** $	shift
2	$ **n**	**+ n** $	reduce $E \rightarrow \mathbf{n}$
3	$ E	**+ n** $	shift
4	$ E **+**	**n** $	shift
5	$ E **+ n**	$	reduce $E \rightarrow E \mathbf{+} \mathbf{n}$
6	$ E	$	reduce $E' \rightarrow E$
7	$ E'	$	accept

§

E' => E => E + n => n + n

E, E+, E+n are all **viable prefixes** of the right-sentential form **E+n**.

Conflicts for shift-reduce parsing

- Parser can reach a configuration in which the parser knowing the stack contents and input symbol cannot decide whether to shift or to reduce (shift-reduce conflicts) , or which of several reductions to make (reduce-reduce conflicts).

Shift/Reduce Conflict

- A situation whether a shift or a reduce could give a parse.

- e.g. stmt -> IF cond THEN stmt

  | IF cond THEN stmt ELSE stmt

  | other

STACK	INPUT
$... IF cond THEN stmt	ELSE ....$

Reduce/Reduce Conflict

- A situation that either two or more rules can be used in a reduction.
- e.g. stmt -> ID **(**parameter_list**)** | expr = expr

  parameter_list -> parameter_list , parameter

  | parameter

  parameter -> ID

  expr -> ID **(**expr_list**)** | ID

  expr_list -> expr_list , expr | expr

Suppose A (I,J) => Id (Id, Id)

STACK	INPUT
... ID **(** ID	, ID)

modify the production

==> stmt -> PROCID (parameter_list)

| expr = expr ;

the lexical analyzer has more job to recognize the ID is PROCID.

* Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize info. far down in the stack to guide the parse.

In Chapter 2

Problems:

1. Y = X + 1

    CFG1: id = function + id

    CFG2: id = id + id

    Ans: Make things as easy as possible for the parser.

    It should be left to scanner to determine if X is a variable or a function.

2. When to quit? X <> Y

    Ans: Go for longest possible fit

LR Parsers

- Advantages:

  (1) LR parsers can be constructed to recognize all programming language construct for which context-free grammars can be written.

  (2) The LR parsing method is more general and efficient than other shift-reduce technique.

  (3) The class of grammars that can be parsed by LR parser is the proper superset of the class of grammars that can be parsed by predictive parsers.

  (4) LR parsers can detect errors in syntax as soon as possible

LR Parsers (Continued)

- Drawbacks:

  Too much work to do

Parsing Action

- Four components:

  1. an input

  2. a stack

  3. a parsing table

  4. the parsing algorithm

Compilation for Yacc file

- % yacc [-dv] grammar.y ==> produce file y.tab.c

-d: cause a file y.tab.h to be produced, which consists of #define statements which associate token codes with token name.

-v: cause a file y.output be produced, which contains a description of the parsing table and report on ambiguities and error in the grammar.

yyparse() ==> return 0 when successfully complete

Construction of a Simple LR (SLR) Parser

- The construction of a DFA from the grammar to which **viable prefixes** of the right-sentential form of the grammar can be recognized.

Example 5.2 Consider the following augmented grammar for rudimentary arithmetic expressions (no parentheses and one operation):

$$E' \rightarrow E$$
$$E \rightarrow E + \mathbf{n} \mid \mathbf{n}$$

A bottom-up parse of the string $\mathbf{n} + \mathbf{n}$ using this grammar is given in Table 5.2.

Parsing actions of a bottom-up parser for the grammar of Example 5.2

	Parsing stack	Input	Action
1	$	$\mathbf{n} + \mathbf{n}$ $	shift
2	$ $\mathbf{n}$	$+ \mathbf{n}$ $	reduce $E \rightarrow \mathbf{n}$
3	$ E	$+ \mathbf{n}$ $	shift
4	$ E +	$\mathbf{n}$ $	shift
5	$ $E + \mathbf{n}$	$	reduce $E \rightarrow E + \mathbf{n}$
6	$ E	$	reduce $E' \rightarrow E$
7	$ E'	$	accept

§

E, E+, E+n are all viable prefixes of the right-sentential form **E+n**.

- <u>Definition</u> An **LR(0) item** of a grammar G is a production of G with a dot (•) at some position of the right side. e.g. A -> XYZ has 4 items

  <u>A ->•XYZ</u> <u>A -> X•YZ</u> <u>A -> XY•Z</u> <u>A -> XYZ•</u>.

- A -> ε has one item <u>A -> •</u>

- Items can be denoted by pairs of integers in computer.

- <u>Items can be viewed as the states of an NFA recognizing viable prefixes.</u>

Closure Operation

- <u>Definition</u> Closure (I) /* I is a set of items for a grammar G. */
  1. Every item in I is in Closure(I).
  2. If <u>$A \rightarrow \alpha \bullet B \beta$</u> is in closure (I) and <u>$B \rightarrow \gamma$</u> is a production, then add the item <u>$B \rightarrow \bullet \gamma$</u> to I, if it is not already there, apply this rule until no more new items can be added to closure (I).

- Closure (I) for I is exactly the ε-closure of a set of NFA states.

An Example

E' -> E

E -> E + T | T

T -> T * F | F

F -> (E) | id

Let I = { E' -> • E} Compute closure (I).

Compute Closure (I)

I = { E' -> • E}

// E' -> E   E -> E + T | T   T -> T * F | F   F -> (E) | id

{ E' -> • E

E -> • E + T

E -> • T

T -> • T * F

T -> • F

F -> •(E)

F -> • id

}

Goto Operation

- Definition Goto (I, X) /* I is a set of items for a grammar G. */
  - The **closure** of the set of all items

    $A \to \alpha X \bullet \beta$ such that $A \to \alpha \bullet X \beta$ is in I.

- Valid Items: an item $A \to \beta 1 \bullet \beta 2$ is valid for a viable prefix $\alpha \beta 1$ if there is a derivation

$$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha \beta 1 \beta 2 w.$$

Steps for Constructing a Simple LR (SLR) Parsing Table

1. Augment the grammar G to become G'.

2. Construct C, the canonical collection of sets of items for G'. (Group items together into sets (The sets-of-items construction), which give rise to the states of an LR parser.)

3. Construct SLR(1) parsing table from C.

Let $C = \{I_0, I_1, I_2, \ldots, I_n\}$, the parsing action for state *i* is determined as follows:

1. If $[A \to \alpha \bullet a\, \beta]$ is in I_i and $Goto(I_i, a) = I_j$, then set action[i, a] to 'shift j'.

(Here 'a' is a *terminal*.)

2. If $[A \to \alpha \bullet]$ is in I_i, then set action [i, a] to 'reduce $A \to \alpha$' for all a in Follow(A).
3. If $[S' \to S\bullet]$ is in I_i, then set action[i, \$] to 'accept'.

The goto transition for state i is constructed

using the rule:

If $Goto(I_i, A) = I_j$, then $Goto[i, A] = j$. Here 'A' is a *non-terminal* symbol.

* In addition, all entries not defined by the former rules are made 'error'; the initial state of the parser is the one constructed from the set of items containing $[S' \rightarrow \bullet S]$.

Note:

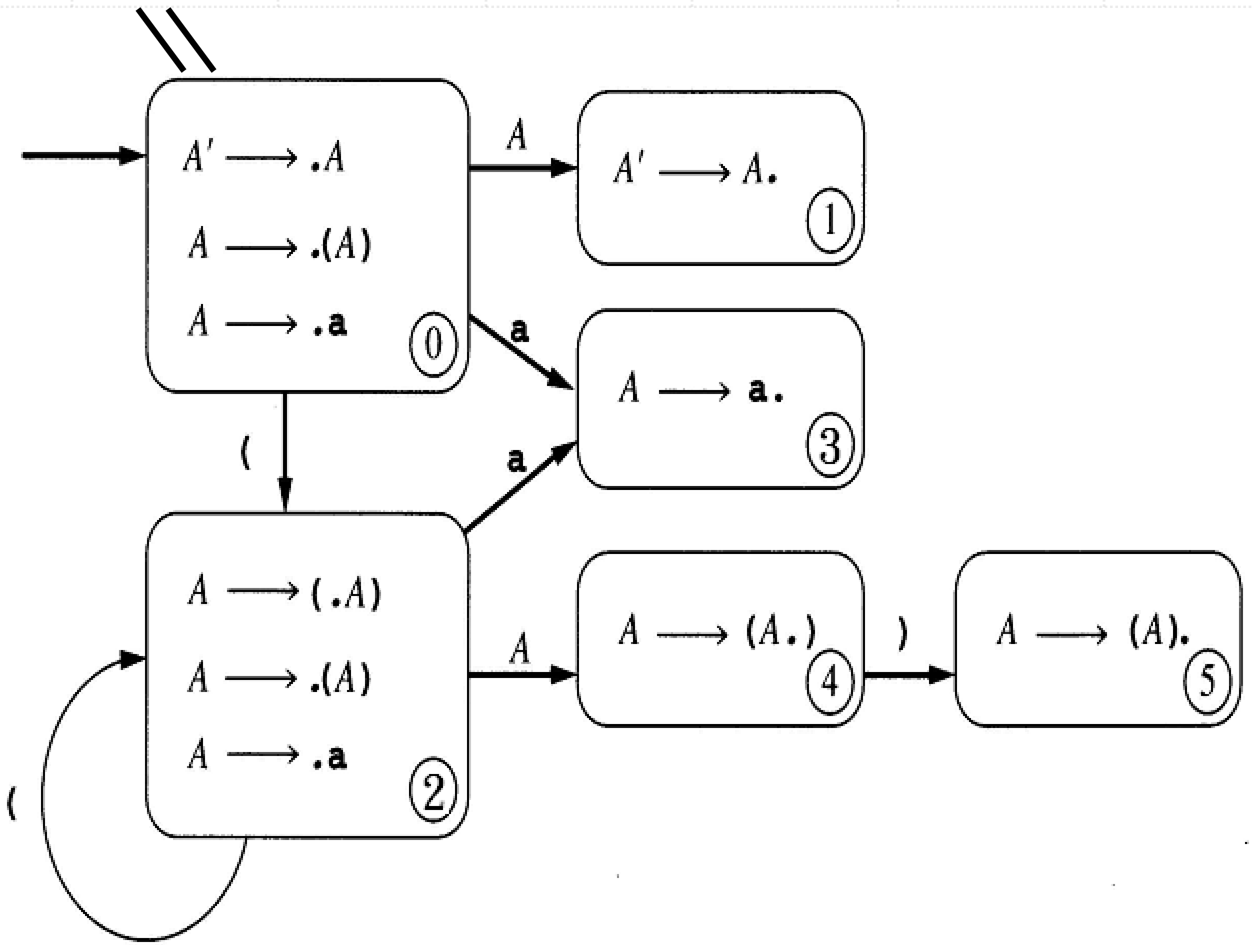
SLR(1) parser construction method is not powerful enough to remember enough left context to decide what action the parser should take.

$A \rightarrow (A) \quad A \rightarrow a$ ➔ $A' \rightarrow A \quad A \rightarrow (A) \quad A \rightarrow a$

Closure ({A' → · A})

Figure 5.5

The DFA of sets of LR(0) items for Example 5.9



Problem 1:

* Every SLR(1) grammar is unambiguous, but there are <u>many</u> unambiguous grammars that are not SLR(1).

e.g. <u>S -> L = R</u> <u>S -> R</u> <u>L -> * R</u> <u>L -> Id</u> <u>R -> L</u> is <u>not ambiguous</u> but the SLR parsing table has multiply-defined entry

Closure({S'→·S}) = I_0

I_0 {S' -> •S, S -> •L = R S -> •R L -> •*R L ->•Id
R -> •L}

Goto(I_0,S) = I_1 {S' -> S•}

Goto(I_0,L) = I_2 {S -> L• = R R -> L•}

Goto(I_0,R) = I_3 {S -> R•}

Goto(I_0,*) = I_4 {L -> *•R R -> •L L -> •*R L -> •Id}

Goto(I_0,Id) = I_5 {L -> Id•}

Goto(I_2,=) = I_6 {S -> L = •R R -> •L L -> •*R L -> •Id}

Goto(I_4,R) = I_7 {L -> *R•}

Goto(I_4,L) = I_8 {R -> L•}

Goto(I_6,R) = I_9 {S -> L = R•}

Check I_2

==> action [I_2, =] be 'shifts to I_6' but

action [I_2, =] be 'reduces R -> L';

that is, a shift/reduce conflict occurs.

Problem 2:
Semantic Action

* The reduction by A -> α on input symbol a where a is in Follow(A) is incorrect sometimes. Shown on the above example, in I_2 the reduction to become 'R =' is definitely incorrect.

LR parsing

- It is possible to carry more information in the state that will allow us to rule out some of these invalid reduction.
- Define an item to include a terminal symbol as a second component.

Definition of LR(1) item

[A -> $\alpha \bullet \beta$, a], where **A -> αβ** is a production and **a** is a terminal or right endmarker **$**. **a** is subset or proper subset of Follow(A).

1 : refer to the length of the second component, called lookahead of the item.

LR(1) item [A -> $\alpha \bullet \beta$, a] is valid for a viable prefix γ

if there is a derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta \alpha \beta w$, where

1. $\gamma = \delta \alpha$, and
2. either a is the first symbol of w, or w is ε and a is $

```
function closure (I)  //I denotes a set of LR(1) items
 {
   do {
          for (each item [A -> α • B β, a] in I, each
          production B -> γ in G' and each terminal
          b in First(βa) s.t. [B -> • γ, b] is not in I)
              add [B -> • γ, b] to I;
         }
   while (no more items can be added to I);
   return I;
}
```

```
function goto(I, X)
{
   Let J be the set of items [A -> α X • β, a] such that [A -> α • X β, a]
   is in I;
   return closure (J);
};
```

```
void  sets_of_items (G')  //G' is the extended grammar of G.
{
   C = {closure({S' -> • S, $})};
   do
      for each set-of-items I in C and each grammar
      symbol X such that goto(I, X) is not empty and
      not in C  do
           add goto(I, X) to C;
   while (no more set-of-items can be added to C);
}
```

An Example: {S -> CC   C -> cC | d}   (1)

1. Augment the grammar: S' -> S S -> CC C -> cC | d
2. Compute First (C$) = First(C) = {c, d}

I_0: { S' -> • S, $
S -> • CC, $
C -> • cC, c/d
C -> • d, c/d}

GOTO (I_0, S) = I_1

I_1: {S' -> S •, $}

GOTO (I_0, C) = I_2

I_2: { S -> C•C, $
C -> •cC, $
C -> •d, $}

(2)

$GOTO\ (I_0, c) = I_3$

I_3:{ C -> c•C, c/d

C -> •cC, c/d

C -> •d, c/d }

$GOTO\ (I_0, d) = I_4$

I_4:{ C -> d•, c/d }

$GOTO\ (I_2, C) = I_5$

I_5: { S -> CC•, $ }

$GOTO\ (I_2, c) = I_6$

I_6: { C -> c•C, $

C -> •cC, $

C -> •d, $ }

$GOTO\ (I_2, d) = I_7$

I_7: { C -> d•, $ }

$GOTO\ (I_3, C) = I_8$

I_8: { C -> cC•, c/d }

(3)

GOTO $(I_6, C) = I_9$

I_9: { C -> cC•, $ }

We can develop a state transition diagram based on the above states to recognize viable prefixes.

SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.

LALR(1) (Lookahead-LR(1)) parsing table

- often used in practice because the parsing tables obtained are considerable smaller.

Construction method:

1. Construct a collection of sets of items (the LR(1) sets).
2. Shrink the collection by merging those sets with common cores (i.e., set of first component) to become <u>the same size of LR(0) set</u>. (note: in general, the core is a set of LR(0) items)
3. GOTO (J, X) = K , where J is the union of one or more sets of LR(1) items, i.e., J = $I_1 \cup I_2 \cup \ldots \cup I_m$ and K = GOTO (I_1, X) $\cup$ GOTO (I_2, X) $\cup \ldots \cup$ GOTO (I_m, X).

Let us use an example to explain the merging.

See the above-stated sets of LR(1) items.

e.g. I_4 and I_7 => I_{47} ;

I_3 and I_6 => I_{36} ;

I_8 and I_9 => I_{89}

e.g. I_4: C -> d•, c/d

I_7: C -> d•, $

I_{47} : C -> d•, c/d/$

The revised parser (LALR parser) behaves essentially like the original parser, although it might do wrong action (reduce) in circumstance where the original would declare error. However, the error will eventually be caught; in fact, it will be caught before any more input symbols are shifted.

Problem caused by merging:

- reduce/reduce conflict due to merging

e.g. state A { [A -> c • , d] [B -> c • , e]}

state B { [A -> c • , e] [B -> c • , d]}

state AB { [A -> c • , d/e] [B -> c • , d/e]}

How about shift/reduce conflict due to merging?

- it is impossible. if it exists then we must have one state like this (the core is the same):

$\{ [A \rightarrow \alpha \bullet , a] \ \ [B \rightarrow \beta \bullet a \gamma , c] \}$ however, this is a conflict.

That is, the original grammar is not a LR(1).

Disambiguating Rules for Yacc (*required only when there exists a conflict)

1. In a shift/reduce conflict the default is to shift.

2. In a reduce/reduce conflict the default is to reduce by the earlier grammar rule in the input sequence.

3. Precedence and associativity (left, right, nonassoc) are recorded for each token that have them.

4. Precedence and associativity of a production rule is that (if any) of its final (rightmost) token unless a

"%prec " overrides. Then it is the token given following %prec.

5. In a shift/reduce conflict where both the grammar rule and the input (lookahead) have precedence, resolve in favor of the rule of higher precedence. In a tie, use associativity. That is, left assoc. => reduce; right assoc. => shift; nonassoc => error.

6. Otherwise use 1 and 2.

(Please See Page 238 of the Textbook)

Assignment #5a

1. Compute the LR(1) parsing table for the following grammar:

   S -> E

   E -> E + F

   F -> i

   F -> (E)

2. Ex. 5.12, 5.13, 5.17, 5.18 of the textbook.