Shift reduce parsing …

- Symbols on the left of "**.** or **|**" are kept on a stack
  - Top of the stack is at "**.** or **|**"
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (RHS of production) and pushes a non terminal (LHS of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

Issues in bottom-up parsing

- How do we know which action to take
  - whether to shift or reduce
  - which production to use for reduction?

- Sometimes parser can reduce but it should not:

  $X \rightarrow \epsilon$ can always be used for reduction!

Issues in bottom-up parsing

- Sometimes parser can reduce in different ways!

- Given stack **δ** and input symbol **a**, should the parser
  - Shift **a** onto stack (making it **δa**)
  - Reduce by some production **A → β** assuming that stack has form **αβ** (making it **αA**)
  - Stack can have many combinations of **αβ**

    (But sometimes both seem possible at the same time, leading to **confusion**.)
  - How to keep track of length of **β?**

- The issue in bottom-up parsing is that the parser doesn't always know whether to **shift more input** or **reduce immediately**, and if reduce, then by **which rule**.

- LR parsers solve this using carefully constructed **parse tables** so that only one action is valid in each state.

Handles

- The basic steps of a bottom-up parser are
  - to identify *a substring* within a *rightmost sentential form* which matches the LHS of a rule.
  - when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost-sentential form.
- Such a substring is called a handle

Grammer:

E → E + T | T
T → id

Input: id + id

Right Most Derivation

1. E
2. E + T
3. T + T
4. id + T
5. id + id

Now Reverse (Bottom-Up-Parser)

1. Start with id + id
2. Handle = id (matches T → id) → reduce → T + id
3. Handle = id (again T → id) → reduce → T + T
4. Handle = T (right side of E → T) → reduce → E + T
5. Handle = E + T (matches E → E + T) → reduce → E

Identifying the correct **handle** at each step tells the parser: (1) **what to reduce**, (2) and by **which production.**

Handle (Definition in Simple Words)

- A **handle** of a string (right sentential form) γ is:
  - a **production rule** A → β
  - and an **occurrence of β inside γ`
  - such that **replacing β with A** gives you the **previous string** in the **rightmost derivation** of γ.
- So, a handle is *what you can reduce right now* in bottom-up parsing.

Grammer:

E → E + T | T
T → id

Input: id + id

Right Most Derivation

1. E
2. E + T
3. E + id
4. id + id

Start from `γ = id + id`:

1. `γ = id + id`
   - Handle = `id` (rightmost one)
   - Rule = `T → id`
   - Replace → `id + T`
2. `γ = id + T`
   - Handle = `id` (leftmost one)
   - Rule = `T → id`
   - Replace → `T + T`

Handle

- Formally, if

$$S \Rightarrow_{rm}^{*} \alpha A w \Rightarrow_{rm} \alpha\beta w,$$

- Then
    - β in the position following α,
    - and the corresponding production $A \rightarrow \beta$ is a handle of $\alpha\beta w$.
- The string w consists of only terminal symbols

Handle … Contd.,

- We only want to reduce handle and not any RHS
- Handle pruning: If β is a handle and A → β is a production then replace β by A
- A right most derivation in reverse can be obtained by handle pruning.

Handle: Observation

- Only terminal symbols can appear to the right of a handle in a rightmost sentential form.
- Why?

Handle: Observation

- Is this scenario possible:
    - $\alpha\beta\gamma$ is the content of the stack
    - $A \rightarrow \gamma$ is a handle
    - The stack content reduces to $\alpha\beta A$
    - Now B $\rightarrow$ β is the handle
- In other words, handle is not on top, but buried inside stack

Handles …

- Consider two cases of right most derivation to understand the fact that handle appears on the top of the stack.

$$S \rightarrow \alpha A z \rightarrow \alpha\beta B y z \rightarrow \alpha\beta\gamma y z$$

$$S \rightarrow \alpha B x A z \rightarrow \alpha B x y z \rightarrow \alpha\gamma x y z$$

Handle always appears on the top

Case I: $S \to \alpha Az \to \alpha\beta Byz \to \alpha\beta\gamma yz$

stack	input	action
$\alpha\beta\gamma$	yz	reduce by $B \to \gamma$
$\alpha\beta B$	yz	shift y
$\alpha\beta By$	z	reduce by $A \to \beta By$
αA	z	

Case II: $S \to \alpha BxAz \to \alpha Bxyz \to \alpha\gamma xyz$

stack	input	action
$\alpha\gamma$	xyz	reduce by $B \to \gamma$
αB	xyz	shift x
αBx	yz	shift y
αBxy	z	reduce $A \to y$
αBxA	z	

Shift Reduce Parsers and Its Conflicts

- The general shift-reduce technique is:
  - – if there is no handle on the stack then shift
  - – If there is a handle then reduce
- Bottom-up parsing is essentially the process of detecting handles and reducing them.
- Different bottom-up parsers differ in the way they detect handles.

- Conflicts
  - What happens when there is a choice
    - –What action to take in case both shift and reduce are valid? shift-reduce conflict
    - –Which rule to use for reduction if reduction is possible by more than one rule? reduce-reduce conflict

Conflicts

Shift-Reduce Conflict: parser is unsure whether it should: (1) **Shift** (read the next input symbol), or (2) **Reduce** (apply a grammar rule).

Left Table Explanation "(Reduce First)"
Wrong Prediction Bcz + before *

Right Table Explanation "(Shift First)":
Valid Bcz * first before +

Consider the grammar E → E+E | E*E | id

and the input id+id*id

stack	input	action
E+E	*id	reduce by E→E+E
E	*id	shift
E*	id	shift
E*id		reduce by E→id
E*E		reduce byE→E*E
E		

stack	input	action
E+E	*id	shift
E+E*	id	shift
E+E*id		reduce by E→id
E+E*E		reduce by E→E*E
E+E		reduce by E→E+E
E		

Conflicts

Reduce-reduce conflict: happens when the parser finds **more than one rule** that can be applied to reduce the same substring.

Left Table Explanation
(Reduce by R → c first, then M → R+R)

Right Table Explanation:
(Reduce by M → R+c directly)

Consider the grammar $M \rightarrow R+R \mid R+c \mid R$
$R \rightarrow c$

and the input c+c

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by R→c
R+R		reduce by M→R+R
M		

Stack	input	action
	c+c	shift
c	+c	reduce by R→c
R	+c	shift
R+	c	shift
R+c		reduce by M→R+c
M		

- **Left Table Path:** parses c+c as M → R+R
- **Right Table Path:** parses c+c as M → R+c
- Both valid → this is why it's called a **Reduce-Reduce Conflict**.

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
  LR(k) grammar.

left to right scanning — right-most derivation — k lookhead

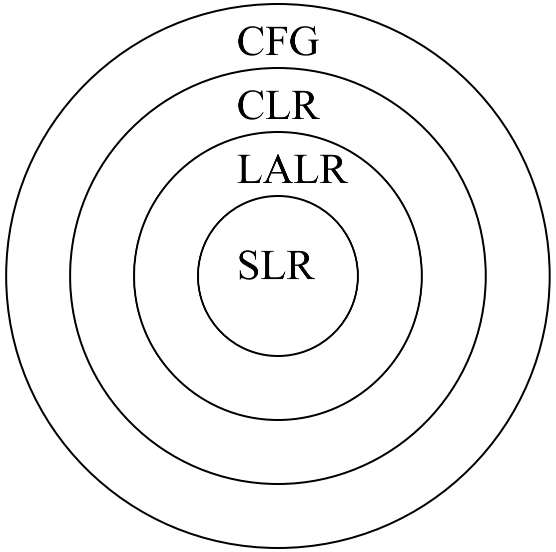
- *An ambiguous grammar can never be a LR grammar.*

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
     - Canonical LR – most general LR parser
     - LALR – intermediate LR parser (look-head LR parser)



- *SLR, CLR and LALR work same, only their parsing tables are different.*

LR Parsers

- The most powerful shift-reduce parsing (yet efficient) is:

LR(k) parsing.

left to right scanning

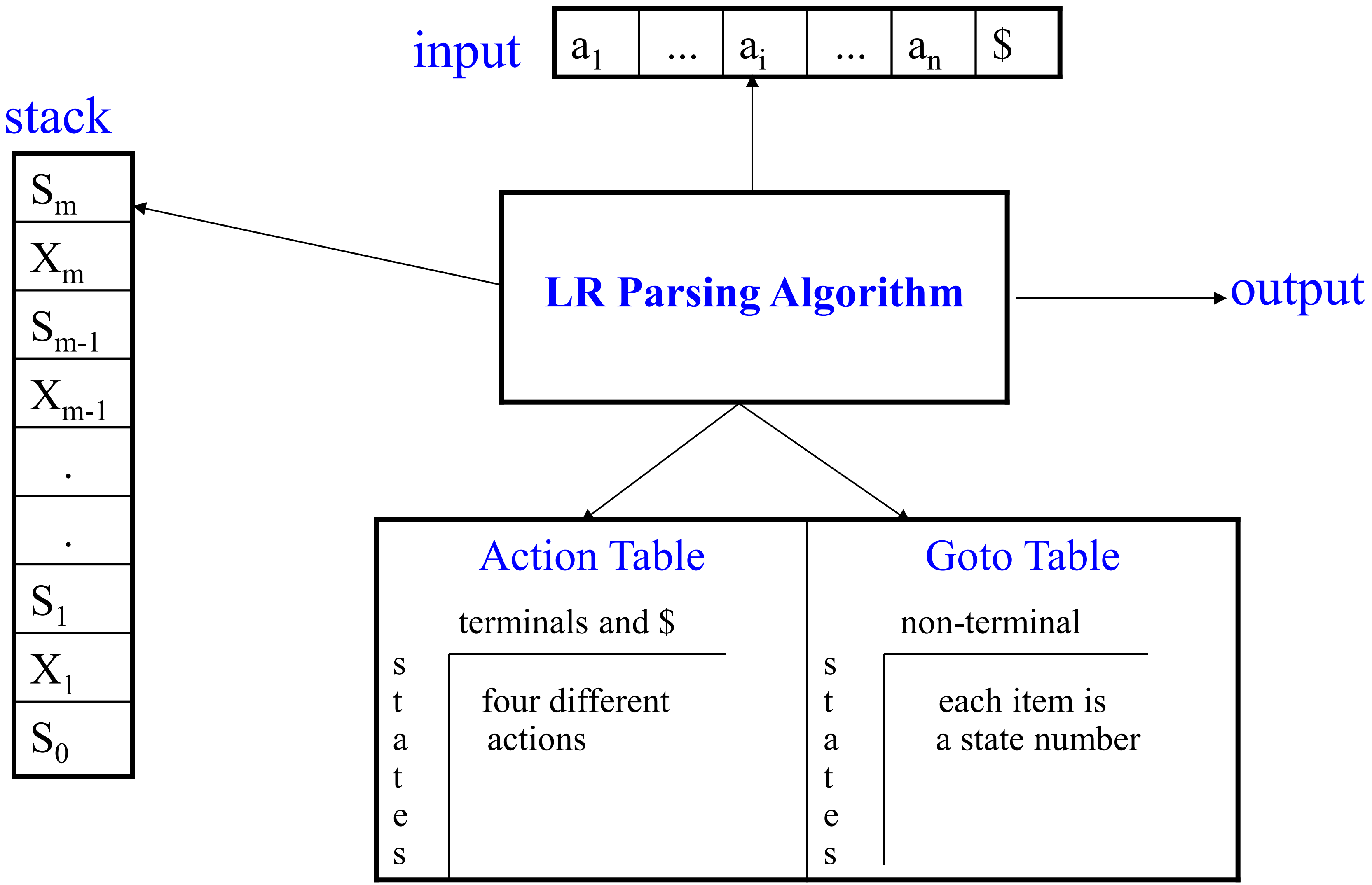
right-most derivation

k lookahead
(k is omitted ➔ it is 1)

- 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 Parsing Algorithm



Example

Consider a grammar and its parse table

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

State	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			

action

goto

A Configuration of LR Parsing Algorithm

- A configuration of a LR parsing is:

$$\underbrace{(\ S_o\ X_1\ S_1\ \ldots\ X_m\ S_m,}_{\text{Stack}}\ \ a_i\ \underbrace{a_{i+1}\ \ldots\ a_n\ \$\ )}_{\text{Rest of Input}}$$

- S_m and a_i decides the parser action by consulting the parsing action table. (*Initial Stack* contains just S_o)

- A configuration of a LR parsing represents the right sentential form:

$$X_1\ \ldots\ X_m\ a_i\ a_{i+1}\ \ldots\ a_n\ \$$$

Actions of A LR-Parser

1. **shift s** -- shifts the next input symbol and the state **s** onto the stack

   $(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$\ ) \rightarrow (S_o\ X_1\ S_1 \ldots X_m\ S_m\ a_i\ s,\ a_{i+1} \ldots a_n\ \$\ )$

2. **reduce $A \rightarrow \beta$** (or **rn** where n is a production number)
   - pop $2|\beta|$ (=r) items from the stack (both symbols and states);
   - then push **A** and **s** where **s = goto[s_{m-r},A]**

     $(S_o\ X_1\ S_1 \ldots X_m\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$\ ) \rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$\ )$
   - Output is the reducing production reduce $A \rightarrow \beta$

3. **Accept** – Parsing successfully completed

4. **Error** -- Parser detected an error (an empty entry in the action table)

Reduce Action

- pop 2|β| (=r) items from the stack (both symbols and states); let us assume that $\beta = Y_1 Y_2 \ldots Y_r$
- then push **A** and **s** (Next State) where **s = goto [s_{m-r}, A]**

$$(S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ Y_1\ S_{m-r} \ldots Y_r\ S_m,\ a_i\ a_{i+1} \ldots a_n\ \$\ ) \rightarrow (S_o\ X_1\ S_1 \ldots X_{m-r}\ S_{m-r}\ A\ s,\ a_i \ldots a_n\ \$\ )$$

Before reduction

After reduction

- In fact, $Y_1 Y_2 \ldots Y_r$ is a handle.

$$X_1 \ldots X_{m-r}\ A\ a_i \ldots a_n\ \$ \Rightarrow X_1 \ldots X_m\ Y_1 \ldots Y_r\ a_i\ a_{i+1} \ldots a_n\ \$$$

In short: Pop RHS symbols → Push LHS non-terminal → Update state using goto → Recognize handle → Continue parsing

LR parsing Algorithm

Input pointer (`ip`) points to the first symbol of **`w`**.

```
Initial state:        Stack: S0     Input: w$

while (1) {
   if (action[S,a] = shift S') {
      push(a); push(S'); ip++
   } else if (action[S,a] = reduce A→β) {
      pop (2*|β|)  symbols;
      push(A); push (goto[S'',A])
      (S'' is the state at stack top after popping symbols)
   }  else if (action[S,a] = accept) {
        exit
   } else { error }
```

At each step, look at **current state (S)** and **input symbol (a)** from the input pointer. There are 4 possible actions:

Case 1: Shift

Case 2: Reduce

Case 3: Accept
Parsing succeeds → Exit

Case 4: Error
Report **syntax error** → Exit parsing.