CS8602:
COMPILER DESIGN

LR Parsers

Session Outcomes

- At the end of this session, participants will be able to
  - Understand the concepts of LR parsers
  - Design Simple LR parser

Outline

- LR Parser
- Augmented grammar
- LR(0) Item construction
- LR parsing table construction
- LR parsing algorithm

LR Parsers

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

  LR(k) parsing.

  left to right scanning | right-most derivation | k lookhead (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.

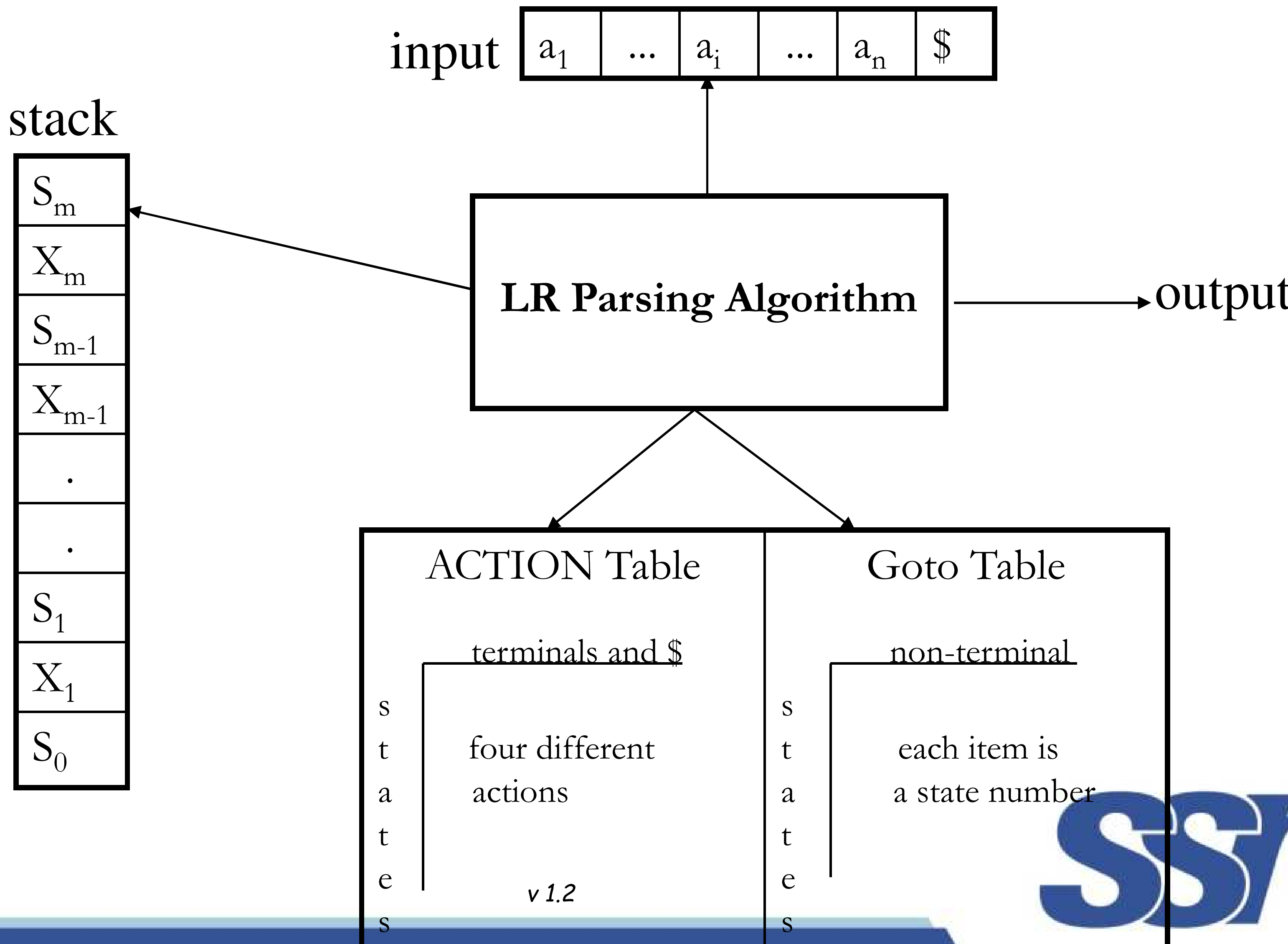
    $$\text{LL(1)-Grammars} \subset \text{LR(1)-Grammars}$$

  - 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

- covers wide range of grammars.
- SLR – simple LR parser
- CLR – most general LR parser
- LALR – intermediate LR parser (look-head LR parser)
- SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

LR Parsing Algorithm



v 1.2

Simple LR Parser

- Less Powerful
- Simple to Implement
- **Construction of SLR parsing table involves**
  - Construction of DFA which recognizes viable prefixes of the grammar.
  - Mapping the moves of the DFA into the form of ACTION and GOTO table.
- **Construction of DFA involves**
  - Collection of sets of LR(0) items (or)
  - Canonical LR(0) collection
- **To Construct Canonical LR(0) collection**
  - Find CLOSURE(I) and GOTO(I,X)

Steps

- Construct an augmented grammar G' for any grammar G
- Construct LR(0) collection of items
- Construct DFA
- Find FOLLOW for the non terminals
- Construct the parsing table

Keywords

- Viable Prefixes
- Augmented Grammar
- Item
- CLOSURE(I)
- GOTO(I,X)

Viable Prefixes

- Prefix of a right sentential form that appears on the stack of a Shift-Reduce parser
- It is always possible to add terminal symbols to the end of a viable prefix to obtain a right sentential form

Augmented Grammar

G' is G with a new production rule S'$\rightarrow$S where S' is the new starting symbol.

Grammar:

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \textbf{id}$

Augmented Grammar:

$E' \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \textbf{id}$

ITEM or LR(0) Items of a Grammar

- An *LR*(0) *item* of a grammar *G* is a production of *G* with a • at some position of the right-hand side
- Thus, a production
  $A \to X\ Y\ Z$
  has four items:
  $[A \to \bullet\ X\ Y\ Z]$
  $[A \to X \bullet\ Y\ Z]$
  $[A \to X\ Y \bullet\ Z]$
  $[A \to X\ Y\ Z \bullet]$
- Note that production $A \to \varepsilon$ has one item $[A \to \bullet]$

Significance of •

- Tells us how much of the production is seen in the grammar while in the process of parsing.
- $[A \to X \bullet Y Z]$
  - Seen a string derivable from X in the input
  - Expect to see the string derivable from $Y Z$ next in the input.

The Closure Operation

- If ***I*** is a set of LR(0) items for a grammar G, then ***closure(I)*** is the set of LR(0) items constructed from I by the two rules:

1. Initially, every LR(0) item in I is added to closure(I).
2. If $A \rightarrow \alpha.B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow .\gamma$ will be in the closure(I). We will apply this rule until no more new LR(0) items can be added to closure(I).

The Closure Operation (Example)

$closure(\{[E' \to \bullet E]\}) =$

$\{ [E' \to \bullet E] \}$

Add $[E \to \bullet \gamma]$

$\{ [E' \to \bullet E]$
$[E \to \bullet E + T]$
$[E \to \bullet T] \}$

Add $[T \to \bullet \gamma]$

$\{ [E' \to \bullet E]$
$[E \to \bullet E + T]$
$[E \to \bullet T]$
$[T \to \bullet T * F]$
$[T \to \bullet F] \}$

Add $[F \to \bullet \gamma]$

$\{ [E' \to \bullet E]$
$[E \to \bullet E + T]$
$[E \to \bullet T]$
$[T \to \bullet T * F]$
$[T \to \bullet F]$
$[F \to \bullet \mathbf{(}\, E\, \mathbf{)}]$
$[F \to \bullet \mathbf{id}] \}$

Grammar:
$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to \mathbf{(}\, E\, \mathbf{)}$
$F \to \mathbf{id}$

GOTO Operation

- If I is a set of LR(0) items and X is a grammar symbol (terminal or non-terminal), then GOTO(I, X) is defined as follows:
  - If $A \rightarrow \alpha.X\beta$ in I
    then every item in **CLOSURE({$A \rightarrow \alpha X.\beta$})** will be in GOTO(I,X).

Example

$I = \{ \quad E' \rightarrow .E, \quad E \rightarrow .E+T, \quad E \rightarrow .T,$
$T \rightarrow .T*F, \quad T \rightarrow .F,$
$F \rightarrow .(E), \quad F \rightarrow .id \ \}$

$GOTO(I, E) = \{ E' \rightarrow E., E \rightarrow E.+T \}$
$GOTO(I, T) = \{ E \rightarrow T., T \rightarrow T.*F \}$
$GOTO(I, F) = \{T \rightarrow F. \}$
$GOTO(I, () = \{ F \rightarrow (.E), E \rightarrow .E+T, E \rightarrow .T, T \rightarrow .T*F,$
$T \rightarrow .F, F \rightarrow .(E), F \rightarrow .id \ \}$
$GOTO(I, id) = \{ F \rightarrow id. \}$

Construction of The Canonical LR(0) Collection

- To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.
- ***Algorithm***:

  C is { closure({S'→.S}) }

  repeat the followings until no more set of LR(0) items can be added to ***C***.

      for each I in ***C*** and each grammar symbol X

          if goto(I,X) is not empty and not in ***C***

              add goto(I,X) to ***C***

- goto function is a DFA on the sets in C.

Canonical LR(0) Collection - Example

I_0: E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_1: E' → E.
E → E.+T

I_2: E → T.
T → T.*F

I_3: T → F.

I_4: F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id

I_5: F → id.

I_6: E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7: T → T*.F
F → .(E)
F → .id

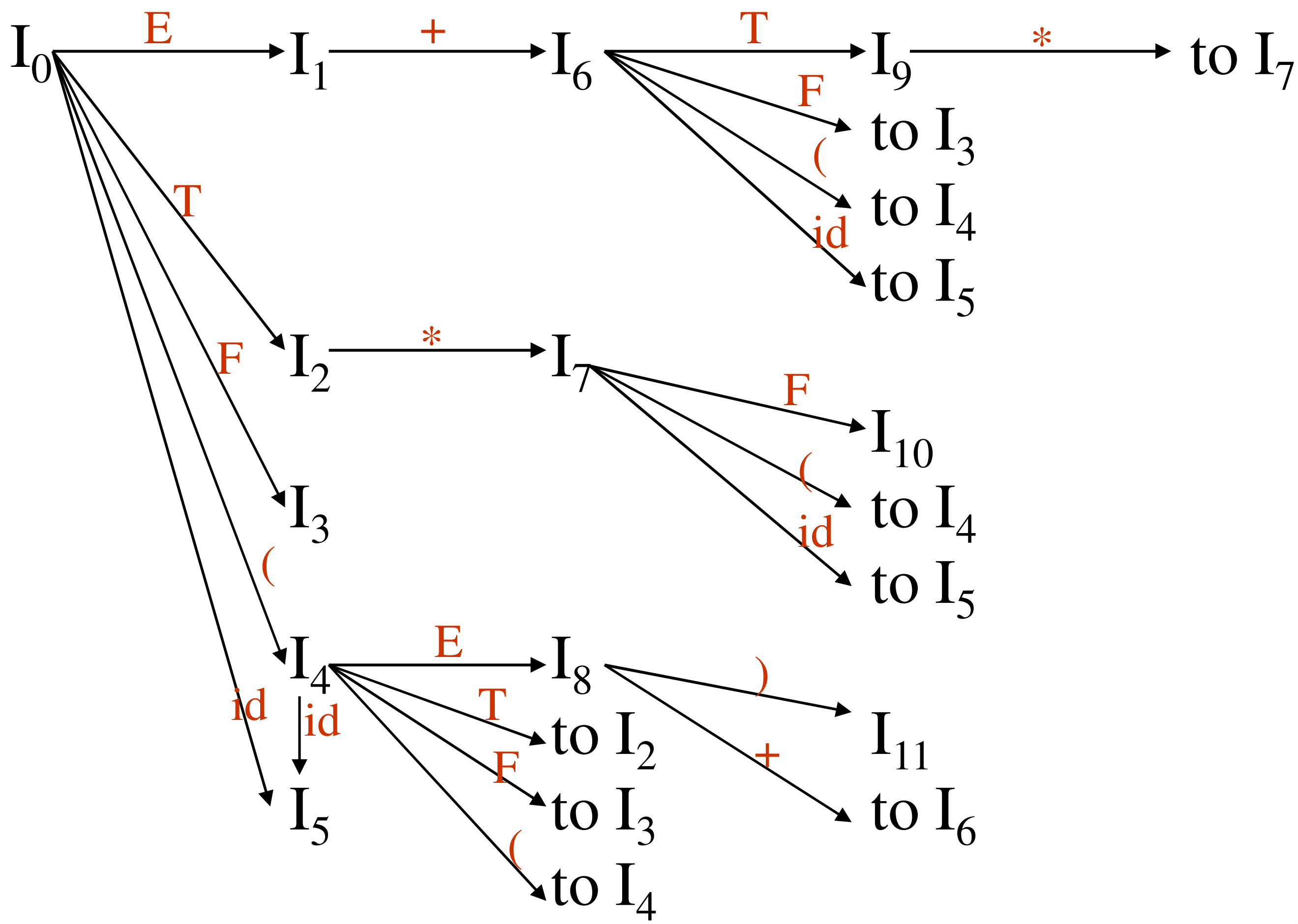
I_8: F → (E.)
E → E.+T

I_9: E → E+T.
T → T.*F

I_{10}: T → T*F.

I_{11}: F → (E).

DFA of GOTO Function



Constructing SLR Parsing Table

1. Construct the canonical collection of sets of LR(0) items for G'. $C \leftarrow \{I_0,\ldots,I_n\}$
2. Create the parsing ACTION table as follows
   - If a is a terminal, $A \rightarrow \alpha.a\beta$ in I_i and $GOTO(I_i,a)=I_j$ then ACTION[i,a] is ***shift j.***
   - If $A \rightarrow \alpha.$ is in I_i , then ACTION[i,a] is ***reduce*** $\boldsymbol{A \rightarrow \alpha}$ for all a in FOLLOW(A) where $A \neq S'$.
   - If $S' \rightarrow S.$ is in I_i , then ACTION[i,$] is ***accept***.
   - If any conflicting actions generated by these rules, the grammar is not SLR(1).
3. Create the parsing GOTO table
   - for all non-terminals A, if $GOTO(I_i,A)=I_j$ then GOTO[i,A]=j
4. All entries not defined by (2) and (3) are errors.
5. Initial state of the parser contains $S' \rightarrow .S$

SLR Parsing Table

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

LR Parsing Algorithm

```
Set ip to point to the first symbol of w$;
Repeat forever begin
        let S be the state on the top of the stack and a be the
        symbol pointed to by ip;
        if ACTION [S, a]=shift S' then
                push a then S' on the top of the stack
                advance ip to the next input symbol
        else if ACTION [S, a]=reduce A→ β then
                pop 2*|β| symbols on the top of the stack
                let s' be the state now on the top of the stack
                Push A then GOTO[S',A] on the top of the stack
                Output the production A→ β
        else if ACTION [S, a]= accept then
                return
        else
                error()
end
```

Example LR Parsing

Grammar:

1. $E \rightarrow E$ **+** T
2. $E \rightarrow T$
3. $T \rightarrow T$ ***** F
4. $T \rightarrow F$
5. $F \rightarrow$ **(** E **)**
6. $F \rightarrow$ **id**

Stack	Input	Action
\$ 0	**id*id+id\$**	shift 5
\$ 0 **id** 5	***id+id\$**	reduce 6 goto 3
\$ 0 F 3	***id+id\$**	reduce 4 goto 2
\$ 0 T 2	***id+id\$**	shift 7
\$ 0 T 2 * 7	**id+id\$**	shift 5
\$ 0 T 2 * 7 **id** 5	**+id\$**	reduce 6 goto 10
\$ 0 T 2 * 7 F 10	**+id\$**	reduce 3 goto 2
\$ 0 T 2	**+id\$**	reduce 2 goto 1
\$ 0 E 1	**+id\$**	shift 6
\$ 0 E 1 + 6	**id\$**	shift 5
\$ 0 E 1 + 6 **id** 5	**\$**	reduce 6 goto 3
\$ 0 E 1 + 6 F 3	**\$**	reduce 4 goto 9
\$ 0 E 1 + 6 T 9	**\$**	reduce 1 goto 1
\$ 0 E 1	**\$**	accept

SLR(1) Grammar

- An LR parser using SLR(1) parsing tables for a grammar G is called as the SLR(1) parser for G.
- If a grammar G has an SLR(1) parsing table, it is called SLR(1) grammar (or SLR grammar in short).
- Every SLR grammar is unambiguous, but every unambiguous grammar is not a SLR grammar.

shift/reduce and reduce/reduce conflicts

- If a state does not know whether it will make a shift operation or reduction for a terminal, we say that there is a **shift/reduce conflict**.
- If a state does not know whether it will make a reduction operation using the production rule i or j for a terminal, we say that there is a **reduce/reduce conflict**.
- If the SLR parsing table of a grammar G has a conflict, we say that that grammar is not SLR grammar.

Summary

- LR Parsers
- Augmented grammar
- LR(0) Item construction
- LR parsing table construction
- LR parsing algorithm
- Conflicts

Check your understanding?

1. Consider the following grammar

   S → L = R

   S → R

   L → *R

   L → id

   R →L

Construct SLR parsing table for the above grammar. Parse the input id = id