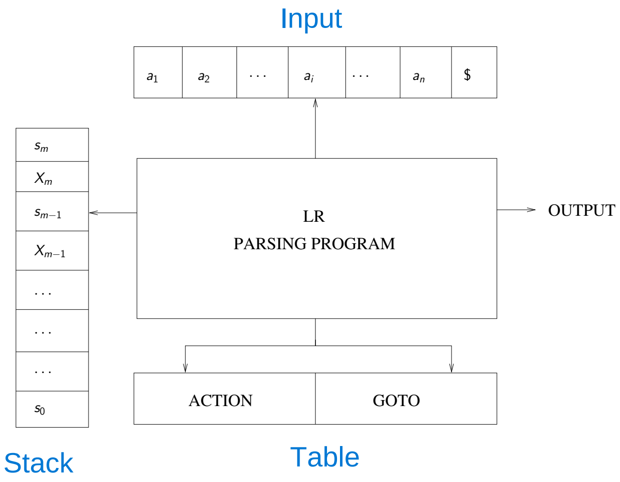
Syntax Analysis

Sudakshina Dutta

IIT Goa

4^{th} March, 2022

LR Parsing Framework



Shift-reduce parsing

This is called LR parsing becoz, We read input from Left to Right and do Right Most derivation from Right to left in Reverse Direction.

- There are context-free grammars for which shift-reduce parsing cannot be used
- We know that shift-reduce parser uses a stack which indicates how much of the input string has been seen and the marker on the input string says how much is remaining
- Shift-reduce parser often reach a configuration in which the parser considering stack and the lookahead cannot decide shift/reduce move or reduce/reduce move
- It is called shift/reduce conflict or reduce/reduce conflict

Shift-reduce conflict

Consider the dangling-else grammar

- *stmt* $\rightarrow$ **if** *expr* **then** *stmt*
- *stmt* $\rightarrow$ **if** *expr* **then** *stmt* **else** *stmt*
- *stmt* $\rightarrow$ *other*

STACK	**INPUT**
$\cdots$ **if** *expr* **then** *stmt*	**else**$\cdots\$$

- The parser does not know whether to shift or to reduce

Shift-reduce Parsing

- It is a bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle

STACK	INPUT	ACTION
$\$$	$id_1 * id_2\$$	shift
$\$id_1$	$*id_2\$$	reduce by $F \to id$
$\$F$	$*id_2\$$	reduce by $T \to F$
$\$T$	$*id_2\$$	shift
$\$T*$	$id_2\$$	shift
$\$T * id_2$	$\$$	reduce by $F \to id$
$\$T * F$	$\$$	reduce by $T \to T * F$
$\$T$	$\$$	reduce by $E \to T$
$\$E$	$\$$	accept

SLR(1) parsing

- The central idea behind "Simple LR" or SLR parsing is the construction from the grammar of $LR(0)$ auomaton
- SLR method for constructing parsing table is a good starting point for studying LR parsing
- The parsing table is called SLR table
- If any conflicting actions are generated by the following rules, we say the grammar is not $SLR(1)$. The algorithm fails to produce a parser in this case.

- We shall consider sets of $LR(0)$ items which is called *canonical LR(0) collection* which provides the basis for constructing a DFA
- For this purpose, we define an augmented grammar with a new start symbol S' and production $S' \to S$

  — Acceptance occurs when and only when the parser is about to reduce $S' \to S$
- We also define two functions, CLOSURE and GOTO

- **Closure of Item Sets**
- Let I be the set of items for a grammar G.
  1. Initially, add every item in I to $CLOSURE(I)$
  2. If $A \rightarrow \alpha.B\beta$ and $B \rightarrow \gamma$ is a production, then add the itemm $B \rightarrow .\gamma$ to $CLOSURE(I)$, if it is not already there. Apply this rule until no more new items can be added to $CLOSURE(I)$
- **The function GOTO**
- $GOTO(I, X)$, where I is a set of items and X is a grammar symbol, is defined to be closure of the set of all items $[A \rightarrow \alpha X.\beta]$ such that $[A \rightarrow \alpha.X\beta]$

We moved . post X

Those waiting for X

- **Example**
- Consider the augmented expression grammar

$$E' \to E$$
$$E \to E + T | T$$
$$T \to T * F | F$$
$$F \to (E) | id$$

If I is the set of one item $\{[E' \to .E]\}$, then $CLOSURE(I)$ contains the set of items as given below

$$E' \to .E$$
$$E \to .E + T$$
$$E \to .T$$
$$T \to .T * F$$
$$T \to .F$$
$$F \to .(E)$$
$$F \to .id$$

and $GOTO(I, E)$ is the set $\{E' \to E., \ E \to E. + T\}$

- **LR Parsing Algorithm**
- 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 program uses a stack to store a string of the form $s_0 X_1 s_1 X_1 s_2 X_2 \cdots X_m s_m$. Each X_i is a grammar symbol and each s_i is a symbol called state
- The parsing table consists of two parts, a parsing *action* function and a *goto* function

- **The behavior of the program driving the parser**
- It determines s_m, the state currently on the top of stack, and a_i, the current input symbol. It consults $action[s_m, a_i]$
- Parsing table entry can be one of the following
  1. *shift* s, where s is a state
  2. *reduce* by a grammar production $A \rightarrow \beta$
  3. accept, and
  4. error   Error at places where Table is empty.

- **Constructing an SLR parsing table**
  Input : An augmented grammar G'
  Output : The SLR parsing table functions *action* and *goto* for G'
- Method : Construct $C = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of $LR(0)$ items for G'
- State i is constructed from I_i. The parsing actions for state i are determined as follows:
  1. If $[A \rightarrow \alpha.a\beta]$ is in I_i and $goto(I_i, a) = I_j$, then set $action[i, a]$ to "shift j". Here a must be terminal.
  2. If $[A \rightarrow \alpha.]$ is in I_i, then set $action[i, a]$ to "reduce $A \rightarrow \alpha$" for all a in $FOLLOW(A)$; here A may not be S'.
  3. If $[S' \rightarrow S.]$ is in I_i, then set action $[i, \$]$ to "accept"

  If any conflicting action result from the above rules, the grammar does not remain $SLR(1)$

▶ LR parsing algorithm

```
set ip to point the first symbol w$
repeat forvever begin
        let s be the state on top of the stack and
            a the symbol pointed to by ip;
        if action[s, a] = shift s' then begin
            push a then s' on top of the stack;
            advance ip to the next input symbol
        end
        else if action[s, a] = reduce A → β then begin
            pop 2 ∗ |β| symbols off the stack;
            let s' be the state now on the top of the stack;
            push A then goto[s', A] on top of the stack;
            output the production A → β
        end
        else if action[s, a] = accept then
            return
        else error()
end
```

- Consider the following (augmented) grammar for arithmetic expression $+$ and $*$
  1. $E' \rightarrow E$
  2. $E \rightarrow E + T$
  3. $E \rightarrow T$
  4. $T \rightarrow T * F$
  5. $T \rightarrow F$
  6. $F \rightarrow (E)$
  7. $F \rightarrow id$

- Augment Grammar
- Form collection of LR(0) Items
- Write states & transitions (DFA)
- Construct Table
- Run the algo

Canonical collection of $LR(0)$ items

We will try all possible actions by looking at all elements following . in each state.

If it leads to new state, Create & name it.

Else point to existing state

$$
\begin{aligned}
I_0 :\ & E' \to .E \\
& E \to .E + T \\
& E \to .T \\
& T \to .T * F \\
& T \to .F \\
& F \to .(E) \\
& F \to .id
\end{aligned}
$$

$$
\begin{aligned}
I_1 :\ & E' \to E. \\
& E \to E. + T
\end{aligned}
$$

$$
\begin{aligned}
I_2 :\ & E \to T. \\
& T \to T. * F
\end{aligned}
$$

$$I_3 : T \to F.$$

$$
\begin{aligned}
I_4 :\ & F \to (.E) \\
& E \to .E + T \\
& E \to .T \\
& T \to .T * F \\
& T \to .F \\
& F \to .(E) \\
& F \to .id
\end{aligned}
$$

$$I_5 : F \to id.$$

$$
\begin{aligned}
I_6 :\ & E \to E + .T \\
& T \to .T * F \\
& T \to .F \\
& F \to .(E) \\
& F \to .id
\end{aligned}
$$

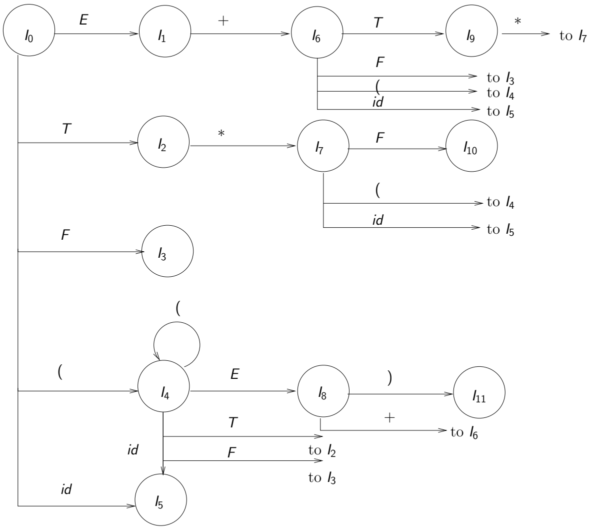
$$
\begin{aligned}
I_7 :\ & T \to T * .F \\
& F \to .(E) \\
& F \to .id
\end{aligned}
$$

$$
\begin{aligned}
I_8 :\ & F \to (E.) \\
& E \to E. + T
\end{aligned}
$$

$$
\begin{aligned}
I_9 :\ & E \to E + T. \\
& T \to T. * F
\end{aligned}
$$

$$I_{10} : T \to T * F.$$

$$I_{11} : F \to (E).$$



Note that only s entries can be deduced directly from this DFA.
For r entries, We need to check the States explicitly & see if anything is ending with . If yes, for every a of Follow of the LHS & We put r_NoInProductionRulesOrder in Table [stateNo, a]

- $FOLLOW(E) = \{+, \$,)\}$
- $FOLLOW(T) = \{+, *, \$,)\}$
- $FOLLOW(F) = \{+, *, \$,)\}$

Parsing Table

Actions (Terminals as input) GOTO (Variables as input)

	$+$	$*$	$($	$)$	id	$\$$	E	T	F
0			s_4		s_5		1	2	3
1	s_6					$accept$			
2	r_3	s_7		r_3		r_3			
3	r_5	r_5		r_5		r_5			
4			s_4		s_5		8	2	3
5	r_7	r_7		r_7		r_7			
6			s_4		s_5			9	3
7			s_4		s_5				10
8	s_6			s_{11}					
9	r_2	s_7		r_2		r_2			
10	r_4	r_4		r_4		r_4			
11	r_6	r_6		r_6		r_6			

Input changes only when the Action is shift.

Check table for 0,id

Check table for 5, *

STACK	INPUT	ACTION
(1)0	$id * id + id\$$	shift
(2)0id5	$*id + id\$$	reduce by $F \to id$
(3)0F3	$*id + id\$$	reduce by $T \to F$
(4)0T2	$*id + id\$$	shift
(5)0T2 $*$ 7	$id + id\$$	shift
(6)0T2 $*$ 7id5	$+id\$$	reduce by $F \to id$
(7)0T2 $*$ 7F10	$+id\$$	reduce by $T \to T * F$
(8)0T2	$+id\$$	reduce by $E \to T$
(9)0E1	$+id\$$	shift
(10)0E1 + 6	$id\$$	shift
(11)0E1 + 6id5	$\$$	reduce by $F \to id$
(12)0E1 + 6F3	$\$$	reduce by $T \to F$
(13)0E1 + 6T9	$\$$	reduce by $E \to E + T$
(14)0E1	$\$$	accept

if table has s5, Write shift here and add id,5 to stack

F3 is popped. Table(0,T) = 2

Table(3,*) = r5
So write the 5th production rule here including S' -> S as 1. Then pop 2*Len(RHS) elements from Stack. Now push LHS to stack and then push Table (Lhs_Previous_element ,lhs) on to stack & write in next row.

- **Example of conflicts (reduce/reduce conflict)**
- Consider the following grammar

$$S \rightarrow A|a$$
$$A \rightarrow a$$

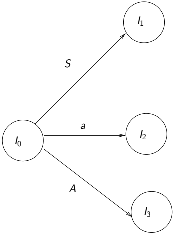
- The canonical collection of $LR(0)$ is given below. Consider the string $aa\$$.

$$\begin{aligned} I_0 : & S' \rightarrow .S \\ & S \rightarrow .A|.a \\ & A \rightarrow .a \end{aligned}$$

$$I_1 : S' \rightarrow S.$$

$$\begin{aligned} I_2 : & S \rightarrow a. \\ & A \rightarrow a. \end{aligned}$$

$$I_3 : S \rightarrow A.$$



	action		goto	
	a	\$	S	A
0	s_2		1	3
1		Accept		
2		r_2/r_3		
3		r_1		

- **Example of conflicts (shift/reduce conflict)**
- Consider the following grammar

$$E \rightarrow E + E | E * E | id$$

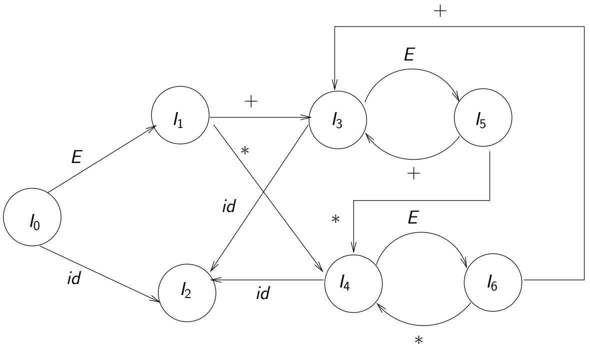
- The canonical collection of $LR(0)$ is given below. Consider the string $id + id + id$

$$\begin{array}{lll}
I_0 : E' \rightarrow .E & I_3 : E \rightarrow E + .E & I_6 : E \rightarrow E * E. \\
\quad E \rightarrow .E + E & \quad E \rightarrow .E + E & \quad E \rightarrow E. + E \\
\quad E \rightarrow .E * E & \quad E \rightarrow .E * E & \quad E \rightarrow E. * E \\
\quad E \rightarrow .id & \quad E \rightarrow .id & \\
 & & \\
I_1 : E' \rightarrow E. & I_4 : E \rightarrow E * .E & \\
\quad E \rightarrow E. + E & \quad E \rightarrow .E + E & \\
\quad E \rightarrow E. * E & \quad E \rightarrow .E * E & \\
 & \quad E \rightarrow .id & \\
 & & \\
I_2 : E \rightarrow .id & I_5 : E \rightarrow E + E. & \\
 & \quad E \rightarrow E. + E & \\
 & \quad E \rightarrow E. * E &
\end{array}$$

For state I1,
1st rule gives r
2,3 rules give s

- States 1, 5, 6 are the sources of conflict

We can check a G for SLR(1) with out performing any Left factoring/ Left recursion elimination.
But for LL(1) we need to do them.


+
E
I_1
+
I_3
I_5
E
*
+
I_0
id
*
E
id
I_2
id
I_4
I_6
*