UNIVERSITY OF NICOSIA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΛΕΥΚΩΣΙΑΣ

COMP-421 Compiler Design

Presented by
Dr Ioanna Dionysiou

Lecture Outline

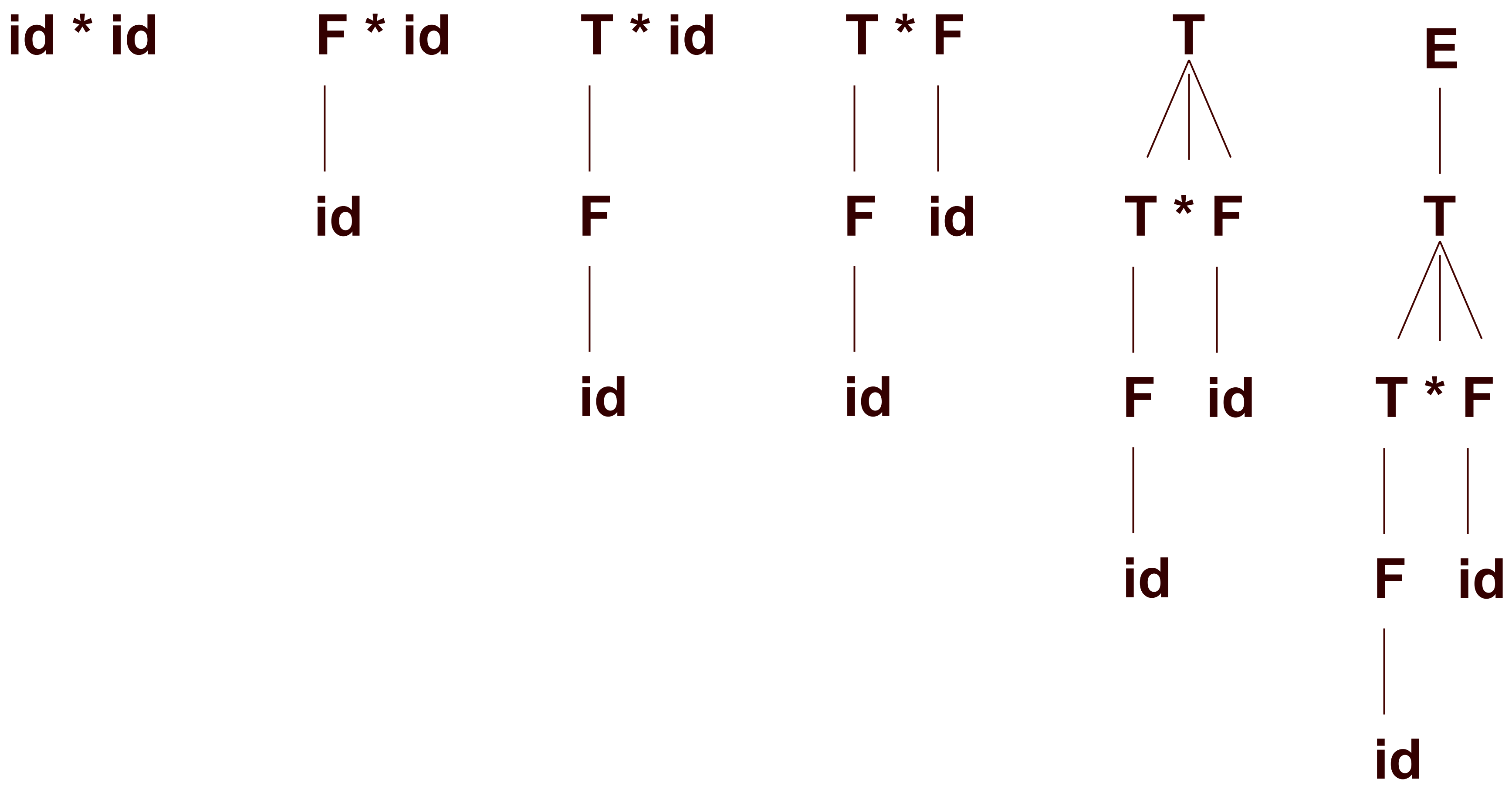
- Bottom-up Parsing
  - Handles, reductions and shift-reduce parsing
  - LR parsers and LR parsing algorithm

Bottom-Up Parsing

- Bottom-up parsing
  - Shift-reduce parsing
    - Attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root
      - Reducing a string w to the start symbol of a grammar
        - » Substring that matches the right side of a production is replaced by the symbol on the left of that production
      - Consider string **id + id** and production **E' → id**
        - » Reduced to id + E'
    - Methods
      - LR parsing (used in a number of automatic parser generators)

Bottom-Up Parsing

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$



Reduction and rightmost derivation

Consider sentence abbcde and grammar

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

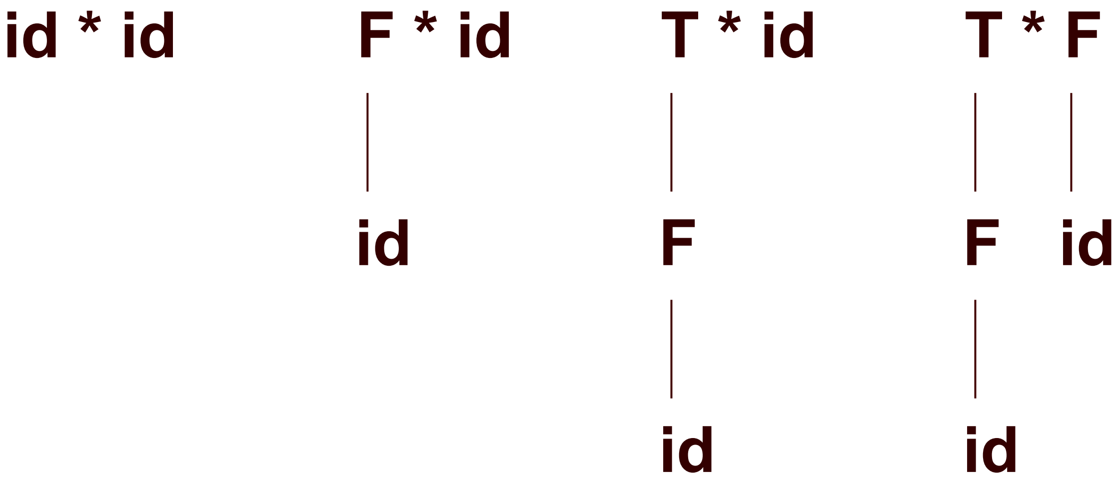
Scan sentence looking for a substring that matches the right side of some production

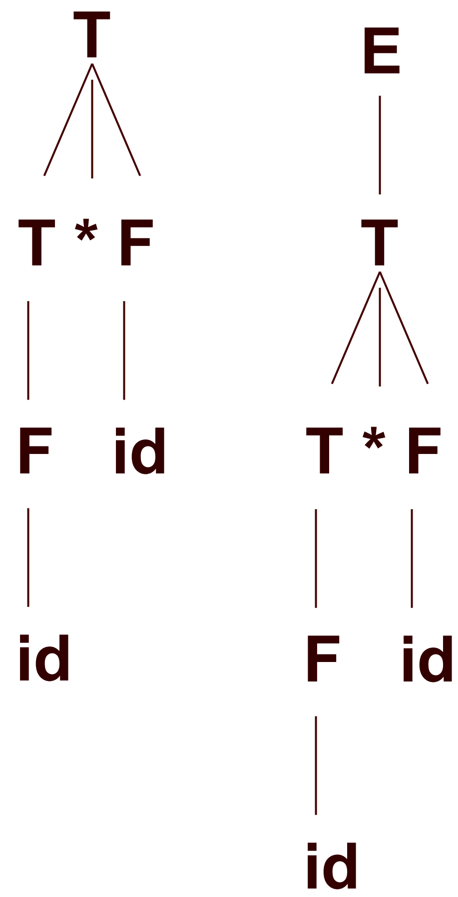
$a\underline{b}bcde$	replace b by A using $A \rightarrow b$
$a\underline{Abc}de$	replace Abc by A using $A \rightarrow Abc$
$aA\underline{d}e$	replace d by B using $B \rightarrow d$
$\underline{aABe}$	replace aABe by S using $S \rightarrow aABe$
S	

$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} aAbcde \underset{rm}{\Rightarrow} abbcde$$

Reduction and rightmost derivation

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$





$E \underset{rm}{\Rightarrow} T \underset{rm}{\Rightarrow} T * F \underset{rm}{\Rightarrow} T * id \underset{rm}{\Rightarrow} F * id \underset{rm}{\Rightarrow} id * id$

Handles

- Informally,
  - Handle of a string is 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, we need to choose the appropriate handle

a<u>b</u>bcde
aA<u>b</u>cde
aAAcde   <u>cannot be reduced to S</u>

Handles

- Formally
  - A handle of a right-sentential form γ is a production $A \rightarrow \beta$ 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

$$S \underset{rm}{\Rightarrow} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$$

$A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$

Shift-Reduce Parsing

- If we want to parse by handle pruning (rightmost derivation in reverse) we need to solve 2 problems
  – Locate the substring to be reduced in a right-sentential form
  – Determine what production to use in case there are multiple productions with that substring on the right side
    • LR parsing

Shift-Reduce Parsing

- Use a stack to hold
  - Grammar symbols
- Use an input buffer to hold
  - String w to be parsed
- Use $ to indicate
  - the bottom of stack
  - The right end of the input
- Basic Idea
  - Parser shifts
    - zero or more input symbols onto stack until a handle is on top of the stack
  - Parser reduces
    - the handle to the left side of the appropriate production
  - Parser repeats this cycle
    - until it has detected an error or
    - until stack contains start symbol and input is empty

Shift-Reduce Parsing

- There are 4 possible actions a shift-reduce parser can make
  - Shift
    - The next input symbol is shifted onto the top of the stack
  - Reduce
    - Parser knows the right end of the handle is at the top of the stack
    - Locates the left end of the handle within the stack and decide which nonterminal to replace the handle
  - Accept
    - Parser announces successful completion of parsing
  - Error
    - Syntax error has occurred

Shift-Reduce Parsing

Grammar G

E → E + E
E → E * E
E → (E)
E → **id**

Input w

id + id * id

WARNING! Grammar has 2 rightmost derivations (because grammar is ambiguous), so there are 2 sequences of steps that the shift-reduce parser might take.

Shift-Reduce Parsing

Grammar G

E → E + E
E → E * E
E → (E)
E → **id**

Input w

id + id * id

STACK	INPUT	ACTION
$	**id + id * id** $	shift

Shift-Reduce Parsing

Grammar G

E → E + E
E → E * E
E → (E)
E → **id**

Input w

id + id * id

STACK	INPUT	ACTION
$	**id** + **id** * **id** $	shift
$ **id**	+ **id** * **id** $	reduce by E → **id**

Continue the process….

Shift-Reduce Parsing

Grammar G

E → E + E
E → E * E
E → (E)
E → **id**

Input w

id + id * id

STACK	INPUT	ACTION
$	**id** + **id** * **id** $	shift
$ **id**	+ **id** * **id** $	reduce by E → **id**
$E	+ **id** * **id** $	shift
$E +	**id** * **id** $	shift
$E + **id**	* **id**$	reduce by E → **id**
$E + E	* **id**$	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	$	accept

Conflicts during shift-reduce parsing

- There are context-free grammars for which shift-reduce parsing cannot be used
  - Parser cannot decide whether to shift or reduce
    - shift/reduce conflict
  - Parser cannot decide which of several reductions to make
    - reduce/reduce conflict
  - These grammars are not in LR(k) class of grammars (non-LR grammars)

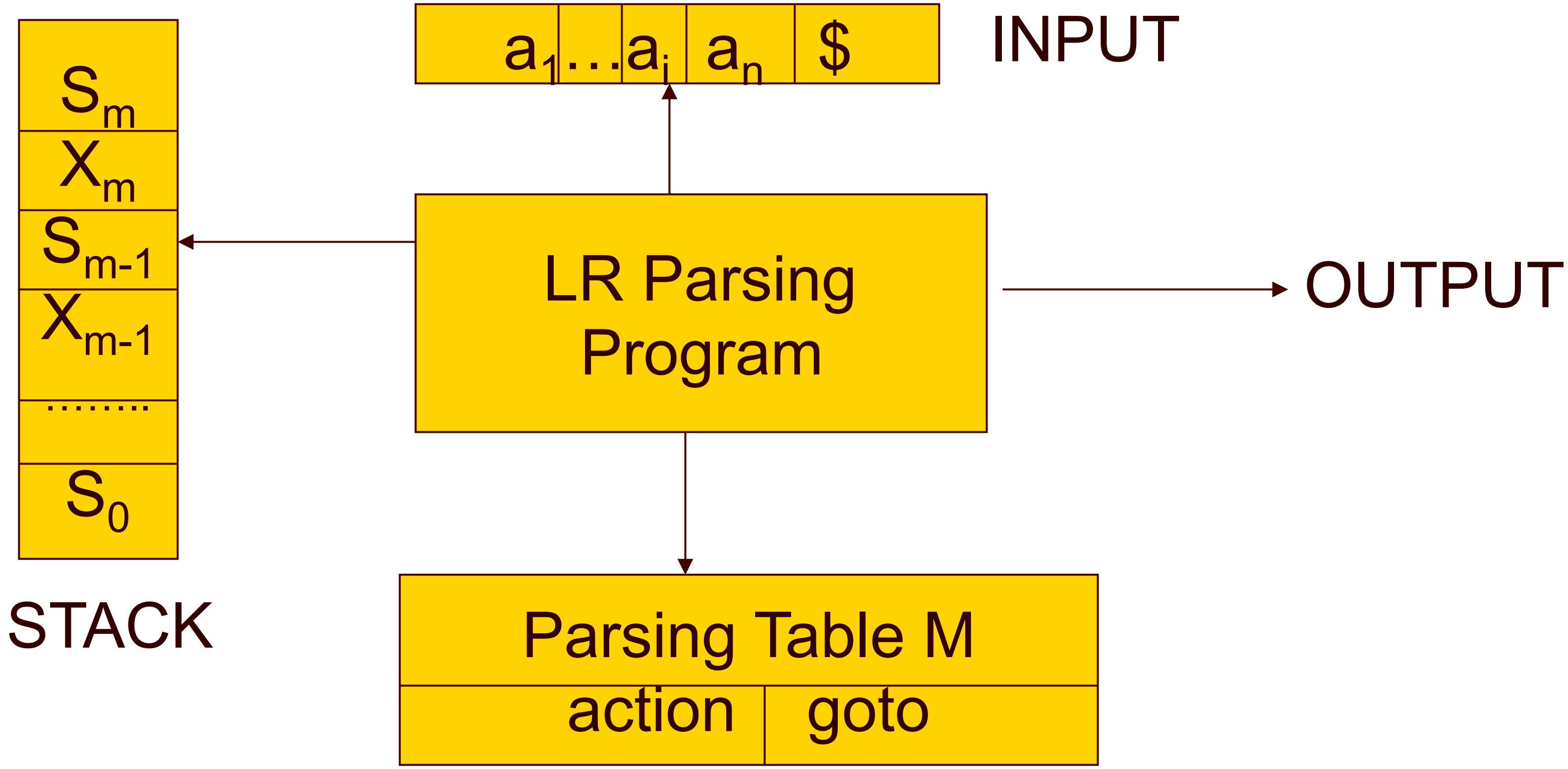
Viable Prefixes

- Viable prefixes Definition
  – Set of prefixes that can appear on the stack of a shift-reduce parser
  – Prefix of a right-sententail form that does not continue past the right end of the rightmost handle of that sentential form
- Will use these when constructing parsing tables for LR parsers

LR Parsers

- LR(k)
  - L
    - left-to-right scanning of inputs
  - R
    - rightmost derivation in reverse
  - K
    - number of input symbols of lookahead that are used to make parsing decisions
      - LR == LR(1)
- Yacc is an LR parser generator
- LR Parsing Algorithm (regardless table technique)
  - 3 techniques to construct an LR parsing table
    - SLR, Canonical LR, LALR

LR Parsing Algorithm



S - state
X - grammar symbol

LR Parsing Algorithm

- The LR Parsing Program determines the next move of the parser by considering
  - a_i, the current input symbol
  - S_m, the current state on top of the stack
  - Consulting action[s_m, a_i] which may contain one of the following values
    - Shift
    - Reduce
    - Accept
    - error

LR Parsing

- A configuration of an LR parser is a pair

(stack contents, unexpended input)

$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\$)$

$X_1X_2\ldots X_m a_i\ a_{i+1}\ \ldots\ a_n$

LR Parsing - shift

- If $action[s_m, a_i] = shift\ s$ then

(stack contents, unexpended input)

$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m\ ,\ a_i\ a_{i+1}\ \ldots\ a_n\$)$

$\Downarrow$

$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m\ a_i\ s,\ a_{i+1}\ \ldots\ a_n\$)$

LR Parsing - reduce

- If action[s_m, a_i] = reduce $A \rightarrow \beta$ then

(stack contents, unexpended input)

($s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m$, $a_i\ a_{i+1}\ \ldots\ a_n\$$)

$$\Downarrow$$

($s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_{m-r}\ s_{m-r}$ $A\ s$, $a_i\ a_{i+1}\ \ldots\ a_n\$$)

s = goto[s_{m-r},A] and r is the length of β (the right side of the production)

Parser pops 2r symbols off the stack

(r states + r symbols) to reach state s_{m-r}

Parser pushes both A and s onto the stack

LR Parsing Algorithm

Input : A string w and a LR parsing table with functions action and goto for grammar G

Output: If w is in L(G), a bottom-up parse of w; otherwise an error

Method: Initially the parser is in a configuration in which it has:

s_0 on the stack, with s_0 is the initial state

w$ in the input buffer

The algorithm that utilizes the LR parsing table to produce a parse for an input is shown on the next slide

LR Parsing Algorithm

```
set ip to point to the first symbol of w$
repeat
 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
          push a then s' on top of the stack
          advance ip to the next input symbol
    else if action[s,a] = reduce A →β then
          pop 2 * |β| symbols off the stack
          let s' be the new state now on the top of the stack
          push A, then goto[s',A] onto the stack
          output the production A →β
     else if action[s,a] = accept then
          return
     else
          error()
END
```

LR Parsing Algorithm

- [ALSU07], page 251-253
  - Slightly different presentation of the algorithm
  - Idea still the same!

LR Parsing Example

GRAMMAR G

(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**

INPUT string w

id * id + id

LR Parsing Example

- Before we examine the workings of the algorithm, try to derive the rightmost derivation for the input string id*id+id

(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**

id * id + id

LR Parsing Example

Rightmost derivation

$$E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+F \underset{rm}{\Rightarrow} E+id \underset{rm}{\Rightarrow} T+id \underset{rm}{\Rightarrow} T*F+id \underset{rm}{\Rightarrow} T*id+id \underset{rm}{\Rightarrow} F*id+id \underset{rm}{\Rightarrow} id*id+id$$

Parsing Table for Grammar G

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			

[ALSU07], page 252

S_i means shift and stack state i

R_j means reduce by production numbered j

Acc means accept

Blank means error

Moves Made 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			

(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

Moves Made 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			

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

STACK	INPUT	ACTION
0	**id** * **id** + **id** $	shift
0 **id** 5	* **id** + **id** $	reduce by F→ **id**

Moves Made 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			

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → **id**

STACK	INPUT	ACTION
0	**id** * **id** + **id** $	shift
0 **id** 5	* **id** + **id** $	reduce by F →**id**
0 F 3	* **id** + **id** $	reduce by T →F

Moves Made LR Parser

STACK	INPUT	ACTION
0	**id * id + id** $	shift
0 **id** 5	*** id + id** $	reduce by F→ **id**
0 F 3	*** id + id** $	reduce by T→F
0 T 2	*** id + id** $	shift
0 T 2 * 7	**id + id**$	shift
0 T 2 * 7 **id** 5	+ **id**$	reduce by F→ **id**
0 T 2 * 7 F 10	+ **id**$	reduce by T→T*F
0 T 2	+ **id**$	reduce by E→ T
0 E 1	+ **id**$	shift
0 E 1 + 6	**id** $	shift
0 E 1 + 6 **id** 5	**$**	reduce by F→ **id**
0 E 1 + 6 F 3	**$**	reduce by T→F
0 E 1 + 6 T 9	**$**	reduce by E→E+T
0 E 1	**$**	accept