Module 03: CS31003: Compilers:

Syntax Analysis or Parsing

Indranil Sengupta
Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

isg@iitkgp.ac.in
ppd@cse.iitkgp.ac.in

September 14, 15, 21 & 22, 2020

Module Objectives

- Understand Parsing Fundamental
- Understand LR Parsing

Module Outline

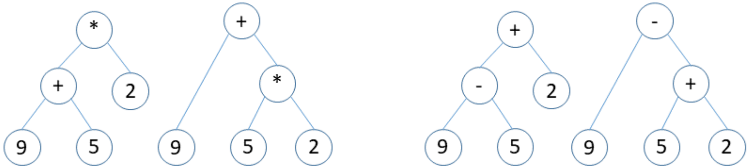
Infix $\rightarrow$ Postfix

Resolving Ambiguity by Infix → Postfix

Let us recap what we did in PDS:

```
9 + 5 * 2 =
    ((9 + 5) * 2) = 28
    (9 + (5 * 2)) = 19

9 - 5 + 2 =
    ((9 - 5) + 2) = 6
    (9 - (5 + 2)) = 2
```

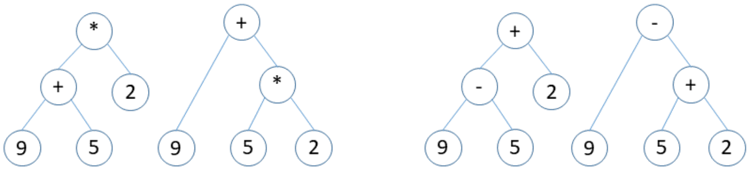


Expression Ambiguity Resolution: Infix → Postfix

$$
\begin{aligned}
9 + 5 * 2 = & (9 + (5 * 2)) = 9\ 5\ 2\ *\ + \\
& ((9 + 5) * 2) = 9\ 5\ +\ 2\ *
\end{aligned}
$$

$$
\begin{aligned}
9 - 5 + 2 = & (9 - (5 + 2)) = 9\ 5\ 2\ +\ - \\
& ((9 - 5) + 2) = 9\ 5\ -\ 2\ +
\end{aligned}
$$

Postfix notation is also called Reverse Polish Notation (RPN)



Associativity and Precedence

Operators

- $*, \ /$ (left)
- $+, \ -$ (left)
- $<, \ \leq, \ >, \ \geq$ (left)
- $! =, \ ==$ (left)
- $=$ (right)

Infix → Postfix: Examples

Infix	Postfix
A + B	A B +
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

```
A + B * C ->
A + (B * C) ->
A (B * C) + ->
A  B  C  *  +
```

Infix → Postfix: Rules

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Operator Precedence Table

Module 03

I Sengupta & P P Das

Objectives & Outline

Infix → Postfix

Grammar

Derivations

Parsing Fundamentals

RD Parsers

Left-Recursion

Ambiguous Grammar

LR Parsers

SR Parsers

	Input						
	$\$$	$+$	$-$	$*$	$/$	$($	$)$
$\$$		$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	
$+$	$\gg$	$\gg$	$\gg$	$\ll$	$\ll$	$\ll$	$\gg$
$-$	$\gg$	$\gg$	$\gg$	$\ll$	$\ll$	$\ll$	$\gg$
$*$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\ll$	$\gg$
$/$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\ll$	$\gg$
$($	$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	$\ll$	$=$
$)$							

Infix → Postfix: Rules

- **Requires operator precedence information**
- **Operands**: Add to postfix expression.
- **Close parenthesis**: Pop stack symbols until an open parenthesis appears.
- **Operators**: Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.
- **End of input**: Pop all remaining stack symbols and add to the expression.

Infix → Postfix: Rules

Module 03

I Sengupta & P P Das

Objectives & Outline

Infix → Postfix

Grammar

Derivations

Parsing Fundamentals

RD Parsers

Left-Recursion

Ambiguous Grammar

LR Parsers

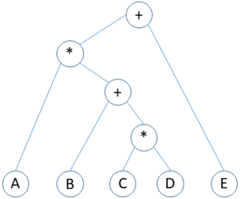
SR Parsers

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +



	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

Evaluating Postfix Expression

- **Create a stack to store operands (or values)**
- **Scan the given expression and do following for every scanned element**
  - If the element is a number, push it into the stack
  - If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
- **When the expression is ended, the number in the stack is the final answer**

Grammar

Grammar

$G =< T, N, S, P >$ is a (context-free) grammar where:

T	:	Set of terminal symbols
N	:	Set of non-terminal symbols
S	:	$S \in N$ is the start symbol
P	:	Set of production rules

Every production rule is of the form: $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$.

Symbol convention:

$a, b, c, \cdots$	Lower case letters at the beginning of alphabet	$\in T$
$x, y, z, \cdots$	Lower case letters at the end of alphabet	$\in T^+$
$A, B, C, \cdots$	Upper case letters at the beginning of alphabet	$\in N$
$X, Y, Z, \cdots$	Upper case letters at the end of alphabet	$\in (N \cup T)$
$\alpha, \beta, \gamma, \cdots$	Greek letters	$\in (N \cup T)^*$

Example Grammar: Derivations

$G = < \{\textbf{id}, +, *, (,)\}, \{E, T, F\}, E, P >$ where P is:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Left-most Derivation of **id** + **id** * **id** $:

$$
\begin{array}{lllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & \underline{T} + T\ \$ & \Rightarrow & \underline{F} + T\ \$ \\
 & \Rightarrow & \underline{\textbf{id}} + T\ \$ & \Rightarrow & \textbf{id} + \underline{T * F}\ \$ & \Rightarrow & \textbf{id} + \underline{F} * F\ \$ \\
 & \Rightarrow & \textbf{id} + \underline{\textbf{id}} * F\ \$ & \Rightarrow & \textbf{id} + \textbf{id} * \underline{\textbf{id}}\ \$ & &
\end{array}
$$

Right-most Derivation of **id** + **id** * **id** $:

$$
\begin{array}{lllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & E + \underline{T * F}\ \$ & \Rightarrow & E + T * \underline{\textbf{id}}\ \$ \\
 & \Rightarrow & E + \underline{F} * \textbf{id}\ \$ & \Rightarrow & E + \underline{\textbf{id}} * \textbf{id}\ \$ & \Rightarrow & \underline{T} + \textbf{id} * \textbf{id}\ \$ \\
 & \Rightarrow & \underline{F} + \textbf{id} * \textbf{id}\ \$ & \Rightarrow & \underline{\textbf{id}} + \textbf{id} * \textbf{id}\ \$ & &
\end{array}
$$

Example Grammar: Derivations

$G = <\{\textbf{id}, +, *, (,)\}, \{E, T, F\}, E, P>$ where P is:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Left-most Derivation of **id** * **id** + **id** \$:

$$
\begin{array}{llllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & \underline{T} + T\ \$ & \Rightarrow & \underline{T * F} + T\ \$ \\
 & \Rightarrow & \underline{F} * F + T\ \$ & \Rightarrow & \textbf{id} * \underline{F} + T\ \$ & \Rightarrow & \textbf{id} * \underline{\textbf{id}} + T\ \$ \\
 & \Rightarrow & \textbf{id} * \textbf{id} + \underline{F}\ \$ & \Rightarrow & \textbf{id} * \textbf{id} + \underline{\textbf{id}}\ \$ & &
\end{array}
$$

Right-most Derivation of **id** * **id** + **id** \$:

$$
\begin{array}{llllllll}
E\ \$ & \Rightarrow & \underline{E + T}\ \$ & \Rightarrow & E + \underline{F}\ \$ & \Rightarrow & E + \underline{\textbf{id}}\ \$ \\
 & \Rightarrow & \underline{T} + \textbf{id}\ \$ & \Rightarrow & \underline{T * F} + \textbf{id}\ \$ & \Rightarrow & T * \underline{\textbf{id}} + \textbf{id}\ \$ \\
 & \Rightarrow & \underline{F} * \textbf{id} + \textbf{id}\ \$ & \Rightarrow & \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$ & &
\end{array}
$$

Parsing Fundamentals

Derivation	Parsing	Parser	Remarks
Left-most	Top-Down	Predictive: Recursive Descent, LL(1)	No Ambiguity No Left-recursion Tool: Antlr
Right-most	Bottom-Up	Shift-Reduce: SLR, LALR(1), LR(1)	Ambiguity okay Left-recursion okay Tool: YACC, Bison

RD Parsers

Recursive Descent Parser

$$
\begin{array}{lcl}
S & \rightarrow & c\ A\ d \\
A & \rightarrow & ab \mid a
\end{array}
$$

```
int main() {
    l = getchar();
    S(); // S is a start symbol

    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'b') { match('b');
    }
}
match(char t) { // Match function
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad$ ($S \Rightarrow cAd \Rightarrow cad$), cabd$ ($S \Rightarrow cAd \Rightarrow cabd$), caad$

Recursive Descent Parser

$$
\begin{array}{lcl}
S & \rightarrow & c\ A\ d \\
A & \rightarrow & aAb \mid a
\end{array}
$$

```
int main() {
    l = getchar();
    S(); // S is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == '$')
        printf("Parsing Successful");
    else printf("Error");
}
S() { // Definition of S, as per the given production
    match('c');
    A();
    match('d');
}
A() { // Definition of A as per the given production
    match('a');
    if (l == 'a') {
        A();
        match('b');
    }
}
match(char t) { // Match function
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: cad$ ($S \Rightarrow cAd \Rightarrow cad$), cabd$, caabd$ ($S \Rightarrow cAd \Rightarrow caAbd \Rightarrow caabd$)

Recursive Descent Parser

$$E \rightarrow a\ E'$$
$$E' \rightarrow +\ a\ E' \mid \epsilon$$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.
    // Here l is lookahead. If l = $, it represents the end of the string
    if (l == '$') printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E, as per the given production
    match('a');
    E'();
}
E'() { // Definition of E' as per the given production
    if (l == '+') {
        match('+');
        match('a');
        E'();
    }
    else return (); // epsilon production
}
match(char t) { // Match function
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a$ ($E \Rightarrow aE' \Rightarrow a$), a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a$), a+a+a$ ($E \Rightarrow aE' \Rightarrow a + aE' \Rightarrow a + a + aE' \Rightarrow a + a + a$)

Recursive Descent Parser

$$E \rightarrow E + E \mid a$$

```
int main() {
    l = getchar();
    E(); // E is a start symbol.

    // Here l is lookahead. if l = $, it represents the end of the string
    if (l == ‘$’)
        printf("Parsing Successful");
    else printf("Error");
}
E() { // Definition of E as per the given production
    if (l == ‘a’) { // Terminate ?
        match(‘a’);
    }

    E();            // Call ?
    match(‘+’);
    E();
}
match(char t) { // Match function
    if (l == t) { l = getchar();
    }
    else printf("Error");
}
```

Check with: a+a$, a+a+a$

Curse or Boon 1: Left-Recursion

A grammar is left-recursive iff there exists a non-terminal A that can derive to a sentential form with itself as the leftmost symbol. Symbolically,

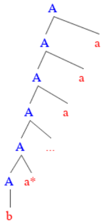
$$A \quad \Rightarrow^{+} \quad A\alpha$$

We cannot have a recursive descent or predictive parser (with left-recursion in the grammar) because we do not know how long should we recur without consuming an input

Curse or Boon 1: Left-Recursion

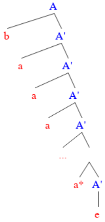
Note that, $\begin{array}{lll} A & \rightarrow & A\alpha \\ A & \rightarrow & \beta \end{array}$ leads to:

$$\begin{array}{lllllllll} A\ \$ & \Rightarrow & A\alpha\ \$ & \Rightarrow & A\alpha\alpha\ \$ & \Rightarrow & A\alpha\alpha\alpha\ \$ & \cdots \\ & \Rightarrow & A\alpha^*\ \$ & \Rightarrow & \beta\alpha^*\ \$ & & & \end{array}$$



Removing left-recursion $\begin{array}{lllll} A & \rightarrow & \beta A' & & \\ A' & \rightarrow & \alpha A' & \mid & \epsilon \end{array}$ leads to:

$$\begin{array}{lllllllll} A\ \$ & \Rightarrow & \beta A'\ \$ & \Rightarrow & \beta\alpha A'\ \$ & \Rightarrow & \beta\alpha\alpha A'\ \$ & \cdots \\ & \Rightarrow & \beta\alpha^* A'\ \$ & \Rightarrow & \beta\alpha^*\ \$ & & & \end{array}$$

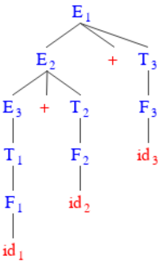


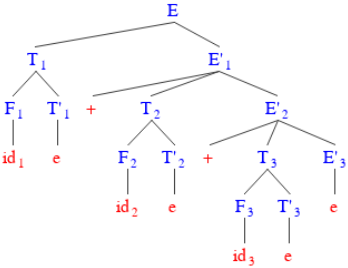
Left-Recursive Example

Grammar G_1 before Left-Recursion Removal			
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**

Grammar G_2 after Left-Recursion Removal					
1:	E	$\rightarrow$	$T\ E'$		
2:	E'	$\rightarrow$	$+\ T\ E'$	$\mid$	ϵ
3:	T	$\rightarrow$	$F\ T'$		
4:	T'	$\rightarrow$	$*\ F\ T'$	$\mid$	ϵ
5:	F	$\rightarrow$	(E)		
6:	F	$\rightarrow$	**id**		

- These are syntactically equivalent. But what happens semantically?
- Can left recursion be effectively removed?
- What happens to Associativity?





Curse or Boon 2: Ambiguous Grammar

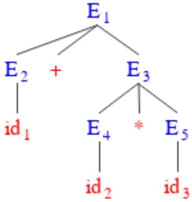
$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \textbf{id}
\end{array}
$$

- Ambiguity simplifies. But, ...
  - Associativity is lost
  - Precedence is lost
- Can *Operator Precedence (infix $\rightarrow$ postfix)* give us a clue?

Ambiguous Derivation of id + id * id

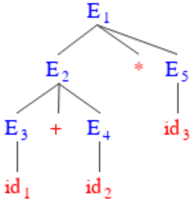
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{E * E}\ \$ \\
&\Rightarrow E + E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

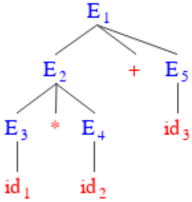
$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{E + E} * \mathbf{id}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}} * \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} + \mathbf{id} * \mathbf{id}\ \$
\end{aligned}
$$



Ambiguous Derivation of **id * id + id**

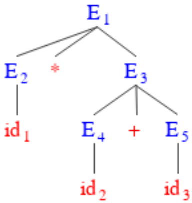
Correct derivation: * has precedence over +

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + E}\ \$ \\
&\Rightarrow E + \underline{\textbf{id}}\ \$ \\
&\Rightarrow \underline{E * E} + \textbf{id}\ \$ \\
&\Rightarrow E * \underline{\textbf{id}} + \textbf{id}\ \$ \\
&\Rightarrow \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$
\end{aligned}
$$



Wrong derivation: + has precedence over *

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E * E}\ \$ \\
&\Rightarrow E * \underline{E + E}\ \$ \\
&\Rightarrow E * E + \underline{\textbf{id}}\ \$ \\
&\Rightarrow E * \underline{\textbf{id}} + \textbf{id}\ \$ \\
&\Rightarrow \underline{\textbf{id}} * \textbf{id} + \textbf{id}\ \$
\end{aligned}
$$



Remove: Ambiguity and Left-Recursion

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & \mathbf{id}
\end{array}
$$

Removing ambiguity:

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \mathbf{id}
\end{array}
$$

Removing left-recursion:

$$
\begin{array}{rlll}
1: & E & \rightarrow & T\ E' \\
2|3: & E' & \rightarrow & +\ T\ E' \mid \epsilon \\
4: & T & \rightarrow & F\ T' \\
5|6: & T' & \rightarrow & *\ F\ T' \mid \epsilon \\
7: & F & \rightarrow & (E) \\
8: & F & \rightarrow & \mathbf{id}
\end{array}
$$

LR Parsers

Shift-Reduce Parser: Example: Grammar

Sample grammar G_1:

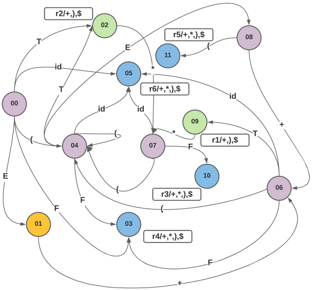
$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & \textbf{id}
\end{array}
$$

Shift-Reduce Parser: Example: Parse Table

State	Action						GO TO		
	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			

Shift-Reduce Parser: Example: Parsing **id** * **id** + **id**

Step	Stack	Symbols	Input	Act.
(1)	0		**id** * **id** + **id** $	s5
(2)	0 5	**id**	* **id** + **id** $	r6
(3)	0 3	F	* **id** + **id** $	r4
(4)	0 2	T	* **id** + **id** $	s7
(5)	0 2 7	T *	**id** + **id** $	s5
(6)	0 2 7 5	T * **id**	+ **id** $	r6
(7)	0 2 7 10	T * F	+ **id** $	r3
(8)	0 2	T	+ **id** $	r2
(9)	0 1	E	+ **id** $	s6
(10)	0 1 6	E +	**id** $	s5
(11)	0 1 6 5	E + **id**	$	r6
(12)	0 1 6 3	E + F	$	r4
(13)	0 1 6 9	E + T	$	r1
(14)	0 1	E	$	acc



1: $E \rightarrow E + T$
2: $E \rightarrow T$
3: $T \rightarrow T * F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow \mathbf{id}$

$$
\begin{aligned}
E\ \$ &\Rightarrow \underline{E + T}\ \$ \\
&\Rightarrow E + \underline{F}\ \$ \\
&\Rightarrow E + \underline{\mathbf{id}}\ \$ \\
&\Rightarrow \underline{T} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{T * F} + \mathbf{id}\ \$ \\
&\Rightarrow T * \underline{\mathbf{id}} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{F} * \mathbf{id} + \mathbf{id}\ \$ \\
&\Rightarrow \underline{\mathbf{id}} * \mathbf{id} + \mathbf{id}\ \$
\end{aligned}
$$

State	Action						GO TO		
	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			