

(01JEUHT) Formal Languages and Compilers

Laboratory N°4

Stefano Scanzio

mail: stefano.scanzio@polito.it

Web: <http://www.skenz.it/compilers>



Attributes of Symbols

- A set of attributes can be associated to each symbol; attributes can be:
 - **Synthesized**: calculated from the values of the attributes of the node's children in the parse tree,
 - **Inherited**: calculated from the values of the parents / siblings in the parse tree.
- A set of semantic rules, specifying how attributes are calculated, is associated to each production.
- The scanner passes semantic values to the parser which, while recognizing the grammar, updates the nodes of the parse tree



Synthesized attributes

- A grammar whose attributes are all synthesized is denoted as an S-attribute grammar.
- In this case, it is possible to calculate the values of all attributes using a bottom-up strategy, from the leaves to the root of the parse tree.

$E \rightarrow E_1 \text{ ' + ' } T$

$E.\text{value} ::= E_1.\text{value} + T.\text{value}$

$E \rightarrow T$

$E.\text{value} ::= T.\text{value}$

$T \rightarrow \text{number}$

$T.\text{value} ::= \text{number.value}$



Cup & Semantics: the Symbol class

- In Cup, each symbol in the stack is an object of class Symbol (`cup/java_cup/runtime/Symbol.java`)
- It contains the following information:
 - A number uniquely identifying the symbol
 - ✦ `public int sym;`
 - The state in which the parse is
 - ✦ `public int parse_state;`
 - Two integers that are used to pass the line and column number from the scanner to the parser
 - ✦ `public int left, right;`
 - An object of class Object to handle semantics
 - ✦ `public Object value;`



scanner.jflex

Passing semantic values to the parser

- Symbol and semantic value:

```
[a-zA-Z][a-zA-Z0-9_]* { return new Symbol(sym.ID, new String(yytext())); }
```

- Symbol, line number, column number, and semantic value:

```
%{  
    private Symbol symbol(int type, Object value){  
        return new Symbol(type, yyline, yycolumn, value);  
    }  
}%
```

```
%%
```

```
[a-zA-Z][a-zA-Z0-9_]* { return symbol(sym.ID, new String(yytext())); }
```

- Or equivalently:

```
[a-zA-Z][a-zA-Z0-9_]* {  
    return new Symbol(sym.ID, yyline, yycolumn, new String(yytext())); }
```

Symbol Constructors:

```
public Symbol( int sym_id)  
public Symbol( int sym_id, int left, int right)  
public Symbol( int sym_id, Object o)  
public Symbol( int sym_id, int left, int right, Object o)
```



Cup & Semantic: specifying nodes types

- Cup must know the type of the semantic value of each symbol
- It uses the following definition of terminals and non-terminals:
 - terminal **<Object>** <list_of_terminals> ;
 - non terminal **<Object>** <list_of_not_terminals> ;
- **<Object>** is the class of the object associated to a given symbol
- Example:
 - terminal String ID;
 - ✦ An object of class String will be associated to ID.
 - terminal Integer NUM;
 - non terminal MyObject var;

```
class MyObject {  
    public String var_name;  
    public String var_type;  
}
```



Cup & Semantic: using semantic values

- Given a set of productions:

$$E ::= E \text{ PLUS } T \\ | E \text{ MINUS } T ;$$

- One can refer to the semantic value of each symbol by adding labels to the symbols of interest:

- A label is constituted by the ':' character followed by a **name**

$$E ::= E:n1 \text{ PLUS } T:n2 \\ | E:n1 \text{ MINUS } T:n2 ;$$

- Within each production, the labels can be used normally as objects of the class specified in the definition of terminals and non-terminals:

$$E ::= E:n1 \text{ PLUS } T:n2 \quad \{ : \text{ System.out.print}(n1 + " + " + n2); : \} \\ | E:n1 \text{ MINUS } T:n2 \quad \{ : \text{ System.out.print}(n1 + " - " + n2); : \}$$


Cup & Semantic: Actions and RESULT

- An action can be associated to each production, (`{:/* Java Code*/:}`) and is executed every time the corresponding production is reduced
- The action updates the semantic value of each symbol
- For each production, the **RESULT** object, of class **Object**, is defined.
- **RESULT** represents the result of the semantic rules contained in the action, **and is therefore associated to the symbol in the left hand side of the production**



Calculating synthesized attributes

- Given the algebraic expressions grammar, the following rule assigns to the symbol 'E' the sum or the subtraction of the values of the addends/subtrahends:

non terminal **Integer** E;

E ::= E:n1 PLUS T:n2

{: RESULT = new Integer(n1.intValue() + n2.intValue()); :}

| E:n1 MINUS T:n2

{: RESULT = new Integer(n1.intValue() - n2.intValue()); :}

;

- Note:

- RESULT must be assigned an object of class Integer: new Integer()
- Mathematical operators work on numbers, not objects: n1.intValue()



Calculating synthesized attributes (2)

- It is possible to propagate more than one semantic value through RESULT, in the following way:

```
terminal RO, RC;  
terminal String identifier;  
terminal Integer Args;  
non terminal Object[ ] Func;  
non terminal goal;
```

```
goal ::= Func:a {:  
    System.out.println( "Function name: " + a[0] + "Number of parameters: " + a[1] );  
};
```

```
Func ::= identifier:a RO Args:b RC {:  
    RESULT = new Object[2];  
    RESULT[0] = new String(a);  
    RESULT[1] = new Integer(b);  
};
```



Calculating synthesized attributes (3)

- Alternatively, one can write a class that contains all the required information:

action code {:

```
class MyFunc {  
    public String id;  
    public Integer args;  
    MyFunc(String id, Integer args) {  
        this.id = new String(id);  
        this.args = new Integer(args);  
    }  
} :}
```

non terminal MyFunc Func;

```
goal ::= Func:a {:  
    System.out.println( "Function name : " + a.id + "Number of parameters: " + a.args );  
} ;
```

```
Func ::= identifier:a RO Args:b RC {: RESULT = new MyFunc( a, b ); :} ;
```

Parser debugging

- A series of options are available in Cup to visualize the parser's internal structures:
 - `-dump_grammar` : Prints the list of terminals, non-terminals and productions
 - `-dump_states` : Prints the state graph
 - `-dump_table` : Prints the ACTION TABLE and the REDUCE TABLE
 - `-dump` : Prints all information
- The parser can be executed in debug mode (all the actions performed to analyze the input sequence are printed)

Normal mode:

```
Yylex l = new Yylex(new FileReader(file));  
parser p = new parser(l);  
Object result = p.parse();
```

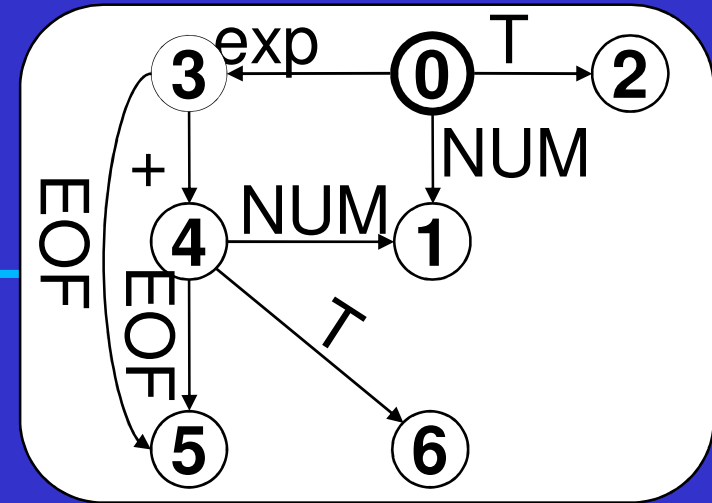
Debug mode:

```
Yylex l = new Yylex(new FileReader(file));  
parser p = new parser(l);  
Object result = p.debug_parse();
```



States

● -dump_states



==== Viable Prefix Recognizer ====

```

START lalr_state [0]: {
  [exp ::= (*) T , {EOF PLUS }]
  [exp ::= (*) exp PLUS T , {EOF PLUS }]
  [T ::= (*) NUMBER , {EOF PLUS }]
  [$START ::= (*) exp EOF , {EOF }]
}
transition on exp to state [3]
transition on T to state [2]
transition on NUMBER to state [1]
-----
lalr_state [1]: {
  [T ::= NUMBER (*) , {EOF PLUS }]
}

```

```

-----
lalr_state [2]: {
  [exp ::= T (*) , {EOF PLUS }]
}
-----
lalr_state [3]: {
  [exp ::= exp (*) PLUS T , {EOF PLUS }]
  [$START ::= exp (*) EOF , {EOF }]
}
transition on EOF to state [5]
transition on PLUS to state [4]
-----

```

```

lalr_state [4]: {
  [exp ::= exp PLUS (*) T , {EOF PLUS }]
  [T ::= (*) NUMBER , {EOF PLUS }]
}
transition on T to state [6]
transition on NUMBER to state [1]
-----
lalr_state [5]: {
  [$START ::= exp EOF (*) , {EOF }]
}
-----
lalr_state [6]: {
  [exp ::= exp PLUS T (*) , {EOF PLUS }]
}
-----

```



Action / Reduce Tables

● -dump_tables

----- ACTION_TABLE -----

From state #0

[term 2:SHIFT(to state 1)]

From state #1

[term 0:REDUCE(with prod 3)] [term 3:REDUCE(with prod 3)]

From state #2

[term 0:REDUCE(with prod 2)] [term 3:REDUCE(with prod 2)]

From state #3

[term 0:SHIFT(to state 5)] [term 3:SHIFT(to state 4)]

From state #4

[term 2:SHIFT(to state 1)]

From state #5

[term 0:REDUCE(with prod 0)]

From state #6

[term 0:REDUCE(with prod 1)] [term 3:REDUCE(with prod 1)]

----- REDUCE_TABLE -----

From state #0

[non term 1->state 3] [non term 2->state 2]

From state #1

From state #2

From state #3

From state #4

[non term 2->state 6]

From state #5

From state #6



Grammar

● -dump_grammar

==== Terminals ====

[0]EOF [1]error [2]NUMBER [3]PLUS

==== Non terminals ====

[0]\$START [1]exp [2]T

==== Productions ====

[0] \$START ::= exp EOF

[1] exp ::= exp PLUS T

[2] exp ::= T

[3] T ::= NUMBER

exp \rightarrow exp PLUS T

exp \rightarrow T

T \rightarrow NUMBER



Debugging

● debug_parse()

```
# Initializing parser
FOUND: 3
# Current Symbol is #2
# Shift under term #2 to state #1
FOUND: +
# Current token is #3
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 0, lhs sym 2 -> state 2
# Goto state #2
# Reduce with prod #2 [NT=1, SZ=1]
# Reduce rule: top state 0, lhs sym 1 -> state 3
# Goto state #3
# Shift under term #3 to state #4
FOUND: 5
# Current token is #2
```

Input string: 3+5

```
# Shift under term #2 to state #1
# Current token is #0
# Reduce with prod #3 [NT=2, SZ=1]
# Reduce rule: top state 4, lhs sym 2 -> state 6
# Goto state #6
Found expression
# Reduce with prod #1 [NT=1, SZ=3]
# Reduce rule: top state 0, lhs sym 1 -> state 3
# Goto state #3
-----
# Shift under term #0 to state #5
# Current token is #0
# Reduce with prod #0 [NT=0, SZ=2]
# Reduce rule: top state 0, lhs sym 0 -> state -1
# Goto state #-1
```



Exercise

Salad 2.10;

Wine 12.00;

Pasta 1.50;

Bread 0.40;

%

Stefano : 2 Pasta, 1 Wine;

Giulia : 1 Salad, 1 Bread, 1 Pasta;

/* OUTPUT:

Stefano: 15.0 EURO

Giulia: 4.0 EURO

*/

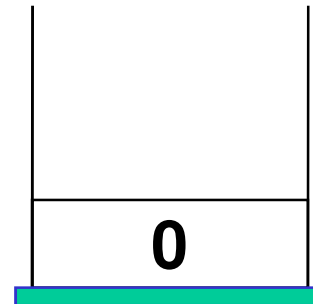
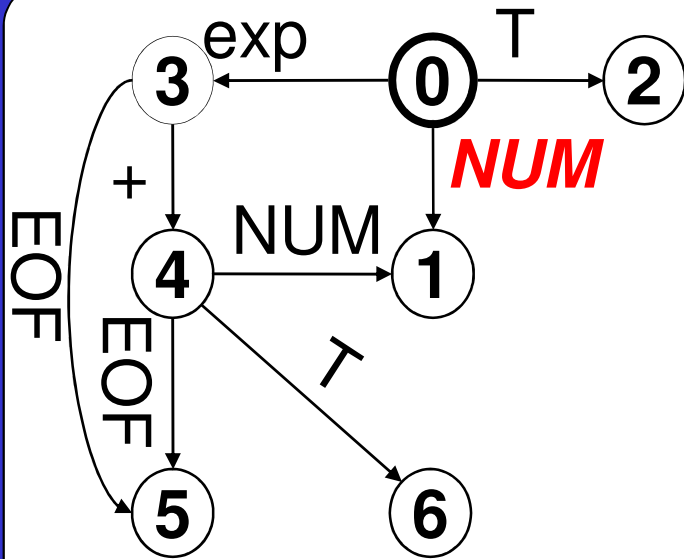


OTHER SLIDES



Initializing parser
Current Symbol is #2
Shift under term #2 to state #1
Current token is #3

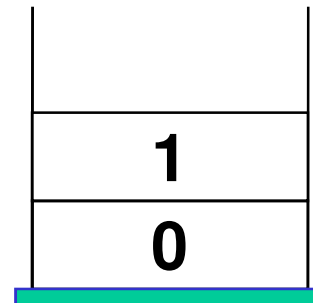
----- ACTION_TABLE -----
From state #0
[term 2:SHIFT(to state 1)]



NUM + NUM eof



shift, go to state 1



NUM + NUM eof



Reduce with prod #3 [NT=2, SZ=1]
 # Reduce rule: top state 0, lhs sym 2 -> state 2
 # Goto state #2

----- ACTION_TABLE -----

From state #1

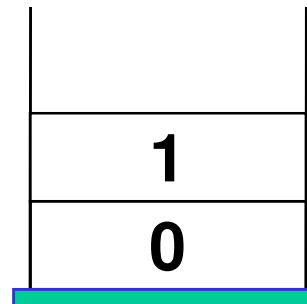
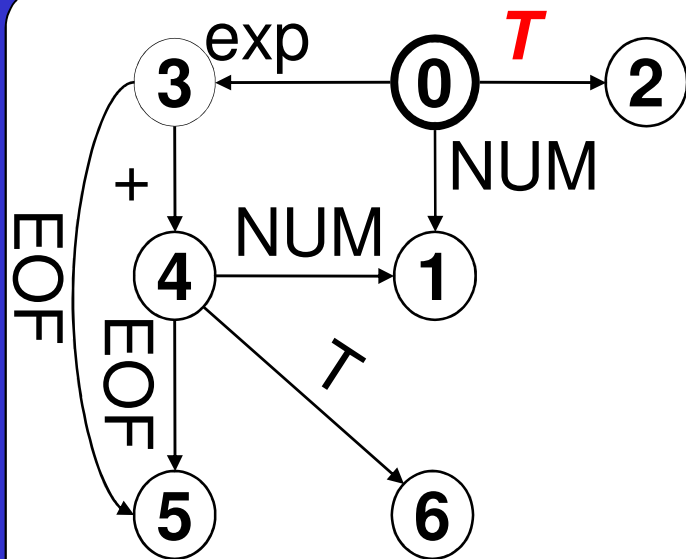
[term 0:REDUCE(with prod 3)]

[term 3:REDUCE(with prod 3)]

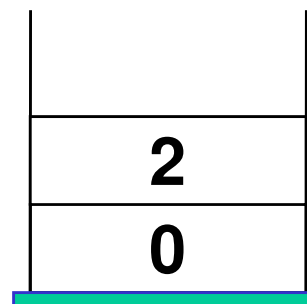
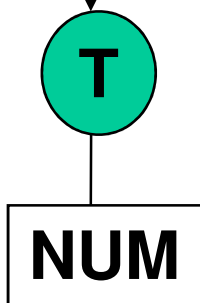
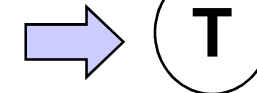
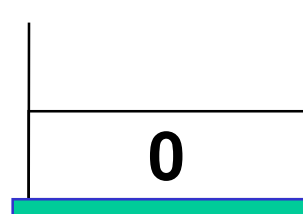
----- REDUCE_TABLE -----

From state #0

[non term 1->state 3] [non term 2->state 2]



reduce (T → NUM)



Reduce with prod #2 [NT=1, SZ=1]
 # Reduce rule: top state 0, lhs sym 1 -> state 3
 # Goto state #3

----- ACTION_TABLE -----

From state #2

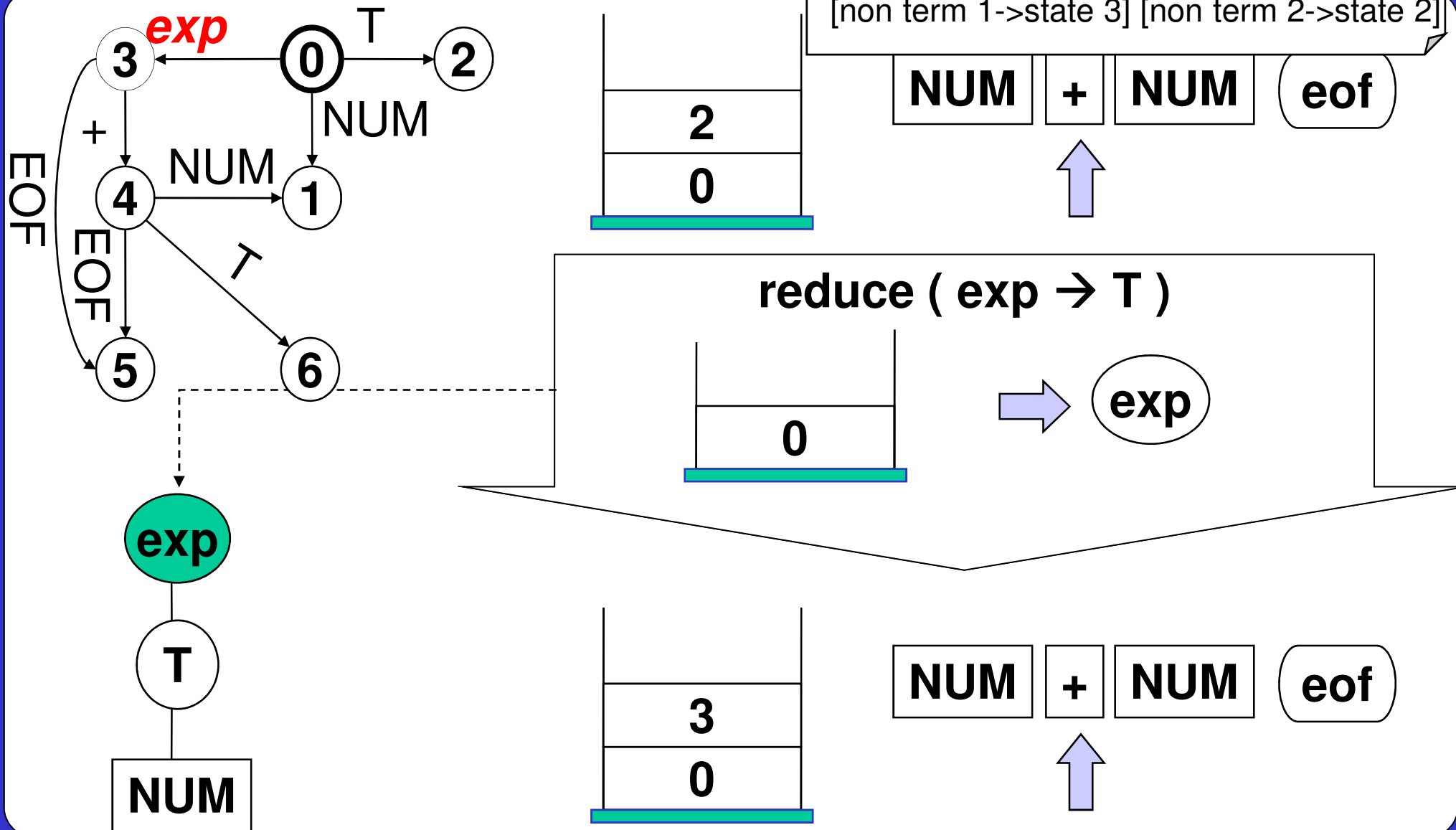
[term 0:REDUCE(with prod 2)]

[term 3:REDUCE(with prod 2)]

----- REDUCE_TABLE -----

From state #0

[non term 1->state 3] [non term 2->state 2]



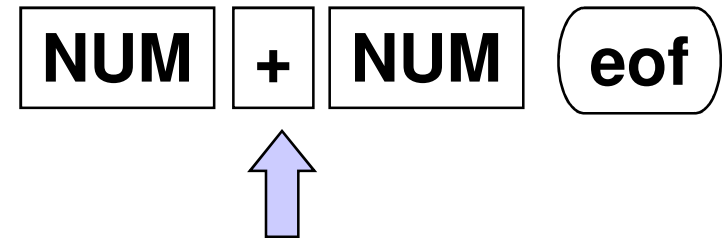
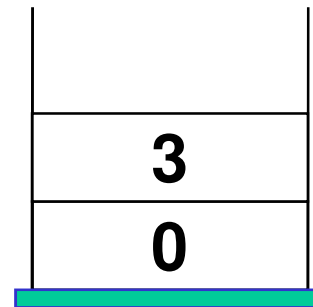
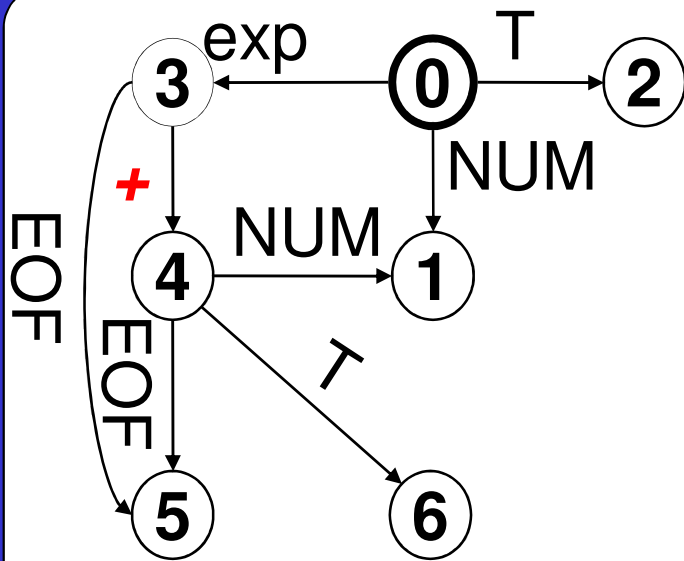
Shift under term #3 to state #4
Current token is #2

----- ACTION_TABLE -----

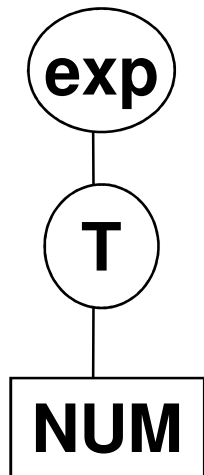
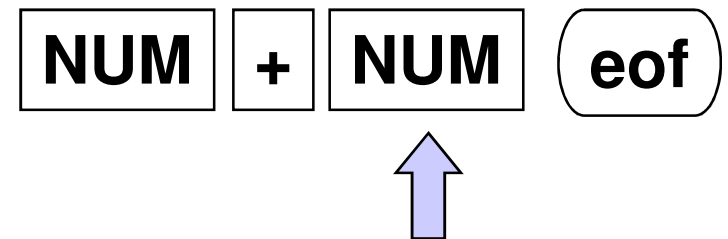
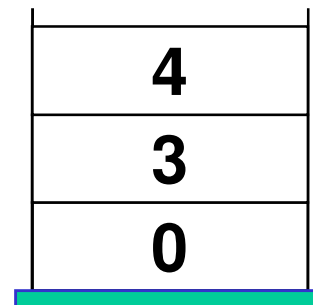
From state #3

[term 0:SHIFT(to state 5)]

[term 3:SHIFT(to state 4)]

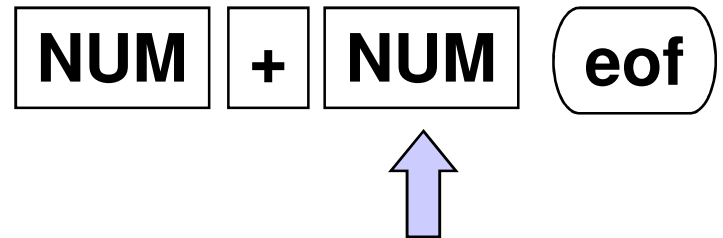
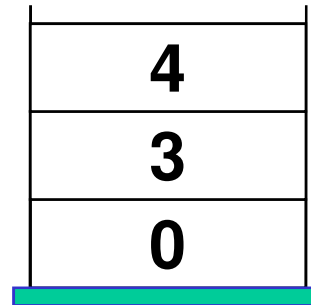
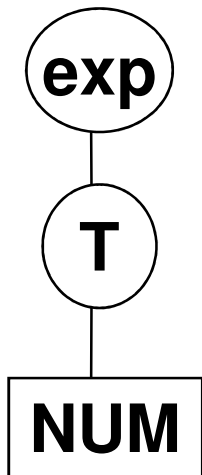
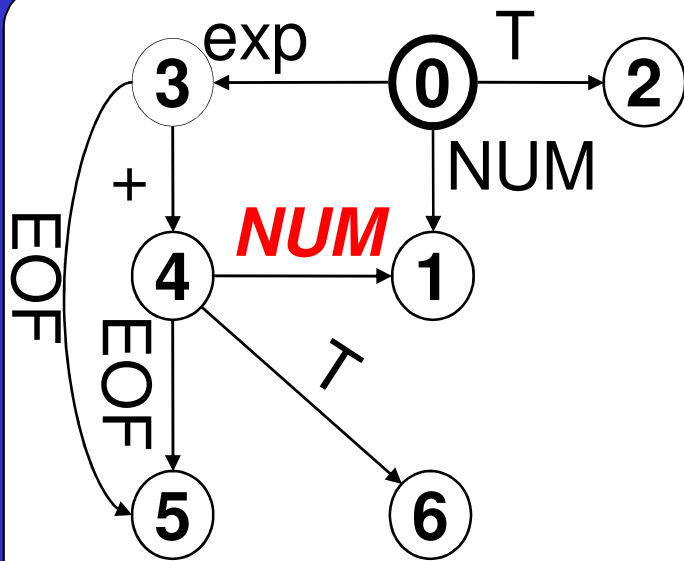


shift, go to state 4

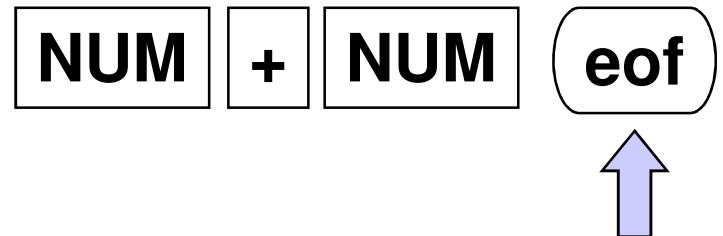
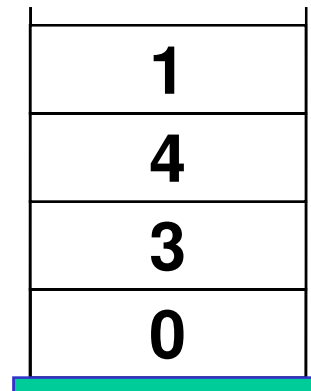


Shift under term #2 to state #1
Current token is #0

----- ACTION_TABLE -----
From state #4
[term 2:SHIFT(to state 1)]



shift, go to state 1



Reduce with prod #3 [NT=2, SZ=1]
 # Reduce rule: top state 4, lhs sym 2 -> state 6
 # Goto state #6

----- ACTION_TABLE -----

From state #1

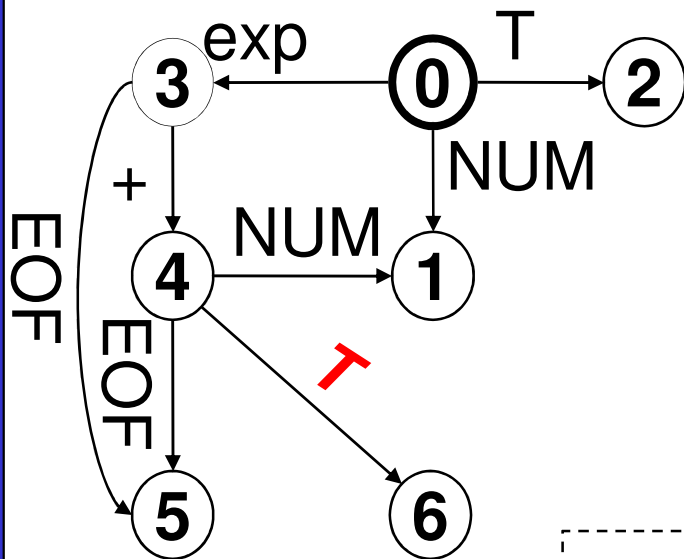
[term 0:REDUCE(with prod 3)]

[term 3:REDUCE(with prod 3)]

----- REDUCE_TABLE -----

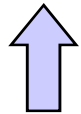
From state #4

[non term 2->state 6]



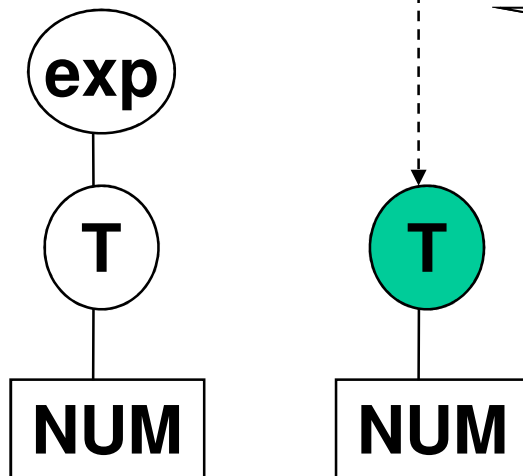
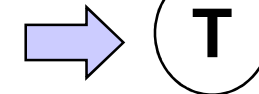
1
4
3
0

NUM + NUM eof



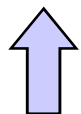
reduce (T → NUM)

4
3
0



6
4
3
0

NUM + NUM eof



Reduce with prod #1 [NT=1, SZ=3]
 # Reduce rule: top state 0, lhs sym 1 -> state 3
 # Goto state #3

----- ACTION_TABLE -----

From state #6

[term 0:REDUCE(with prod 1)]

[term 3:REDUCE(with prod 1)]

----- REDUCE_TABLE -----

From state #0

[non term 1->state 3] [non term 2->state 2]

