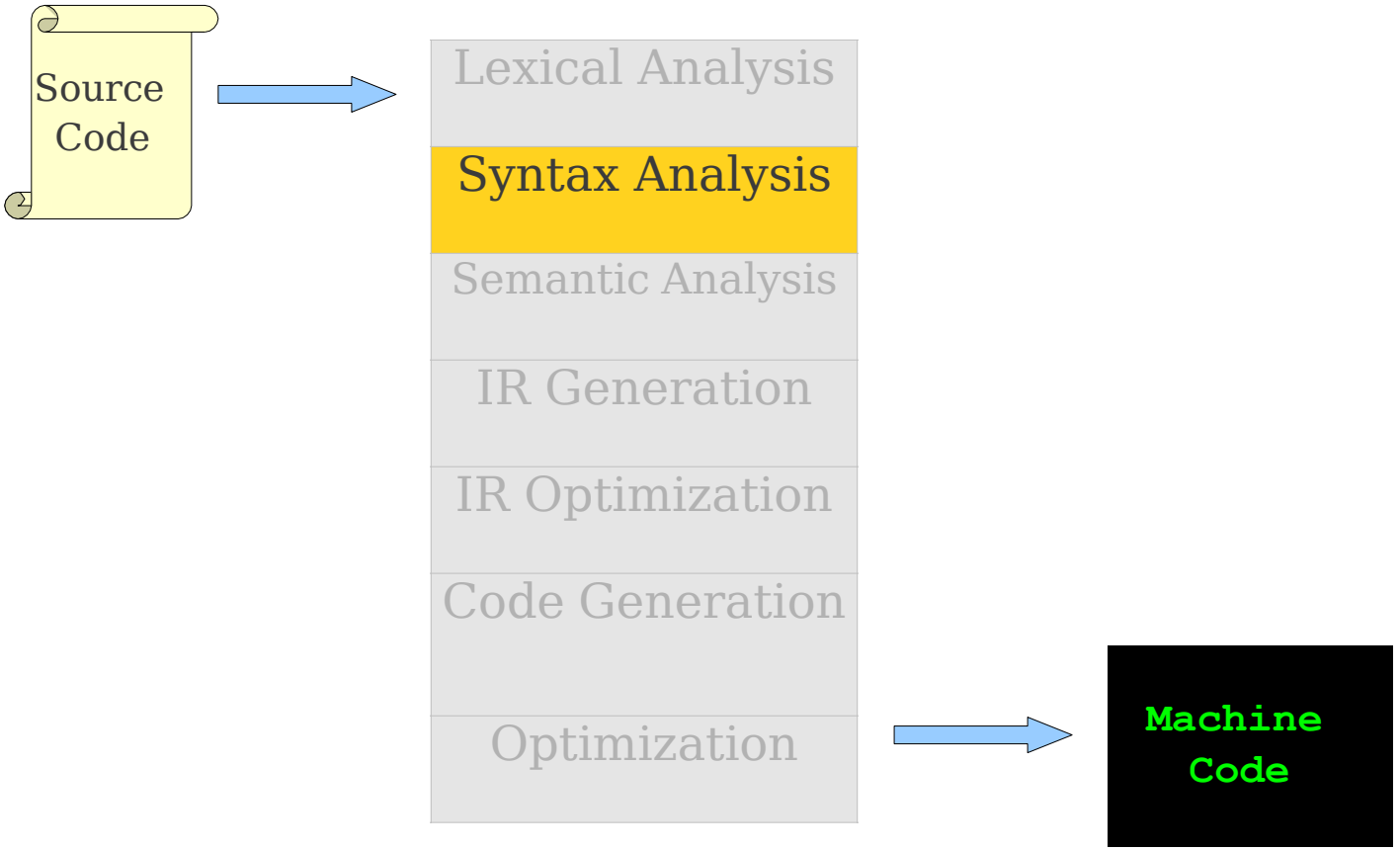
بسم الله الرحمن الرحيم

Syntax Analysis:
Subsection 0: Syntax Graph

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Context Free Grammar-Free Parser!

- Is there any other way to describe a programming language?
- Look at the following Pascal code
- **Idea:** divide the code into small pieces

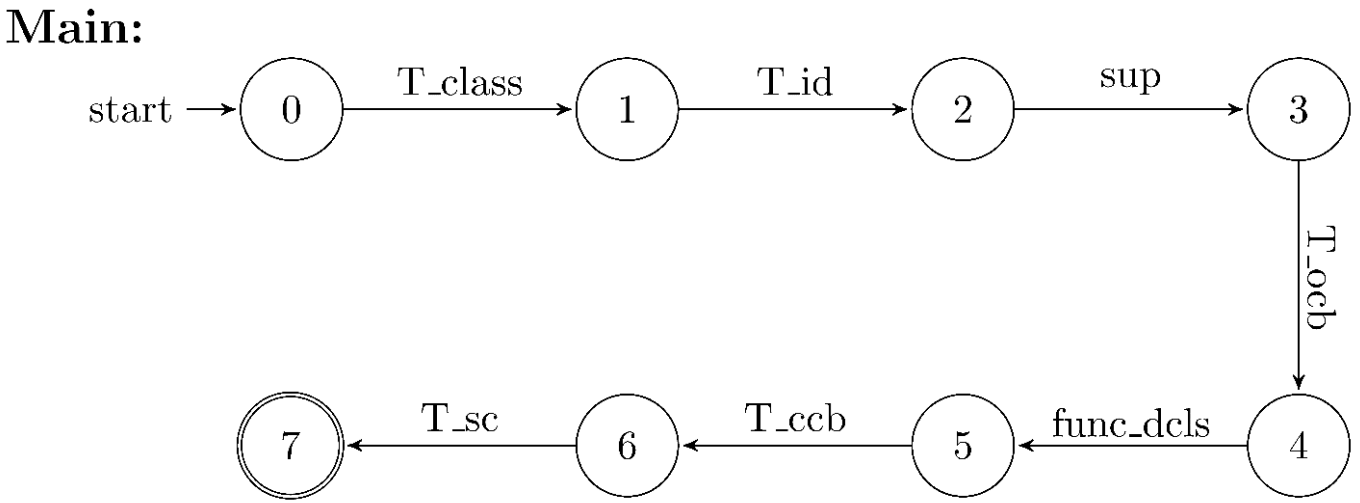
```
Program id;
id : id;
id : id;

begin
     id := e;
     if be then
     .
     .
end.
```

Explain it for a child!

```
class Main inherits IO {
  main() : Object {
    out_string("Hello, world!\n")
  };
};
```

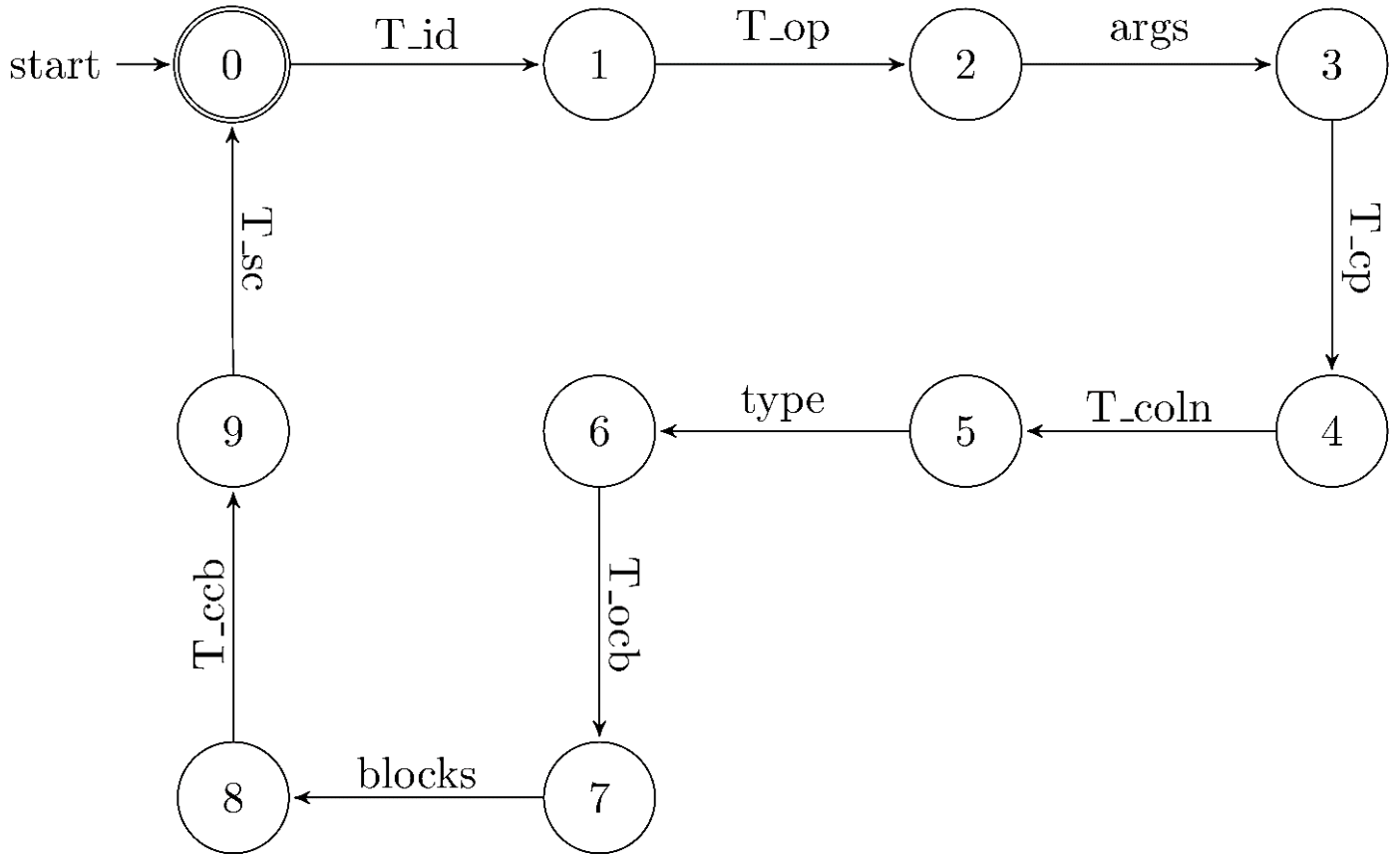
Example

- Graph of Program:



Example

func_dcls:



Syntax Graph

- A Structure that represent and parse the program.
- It contains some **directed graphs.**
- Each **node** represents a state in parsing process.
- Each **directed edge**, represent any possible way we can continue parsing correctly. They are labeled with **tokens** or **graph names.**
- Each node can have **at most one** edge with a graph name.

Syntax Graph

- Edges also can have one **Semantic Routine** at most. These routines are pieces of code which analyze semantics and generates the target code. (So it is part of **Code Generation**)
- Parser use graphs and a stack calls, **Parse Stack (PS)** to parse the code.
- Each graph must have at least one accepting (final) node and exactly one **starting node**.
- The graph must be **deterministic**.

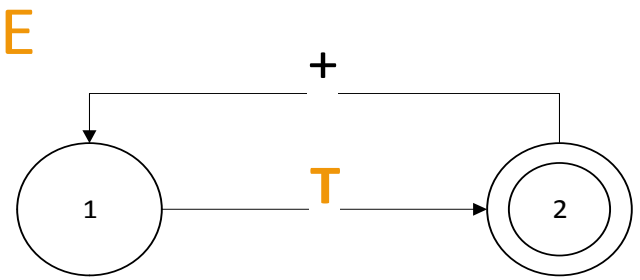
Syntax Graph Rules

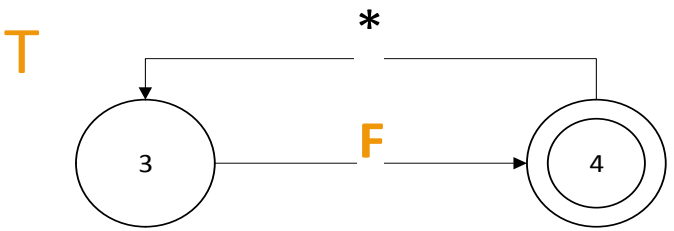
- Parser Starts from node number **0**. (or 1)
- At each non final node parser get next token and:
  1. Take **out** the edge labeled with that token if exists.
  2. Otherwise, if there is exactly one edge with a **graph name**, parser pushes **current node number** into P-Stack, and **jump** to the starting point of that graph.
  3. Otherwise: **error**

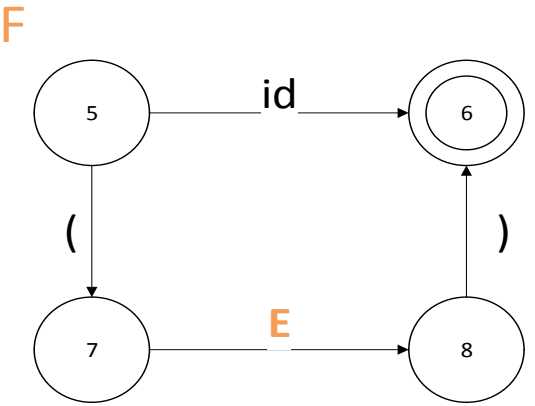
Syntax Graph Rules

- When parser is in a final state of a graph, if no edge matches with the current token, returns from that (to possibly the top of PS).
- Reaching the final state at main graph (first graph) means end of parsing. If current token be $ (special letter for EOF) it means **success!** Otherwise **error!**
- After an edge passed, its semantic routine is called if exists.

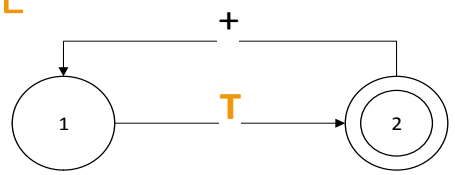
Expression Graph



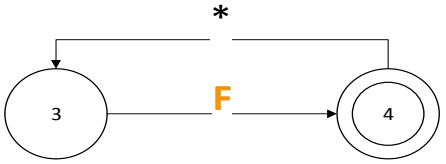




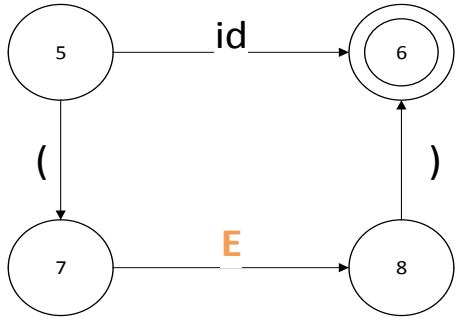
E



T



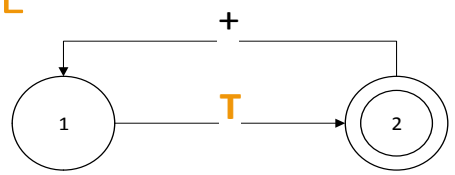
F



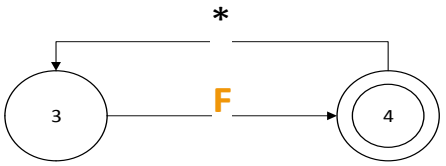
a + b * c EOF

PS

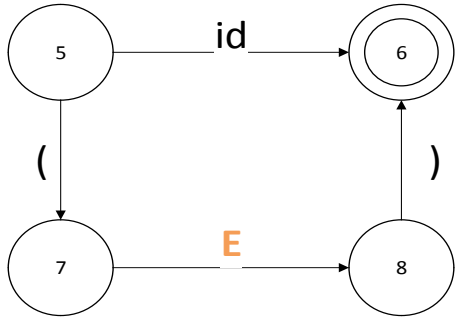
E



T

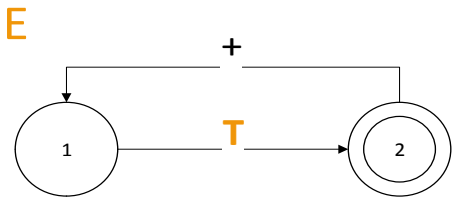


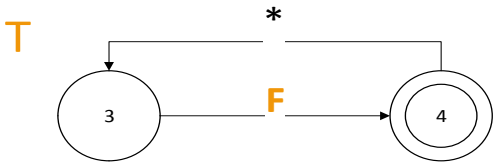
F

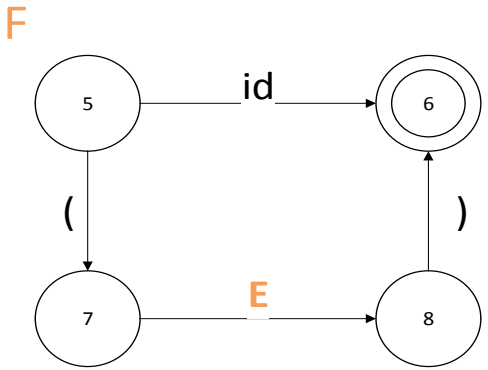


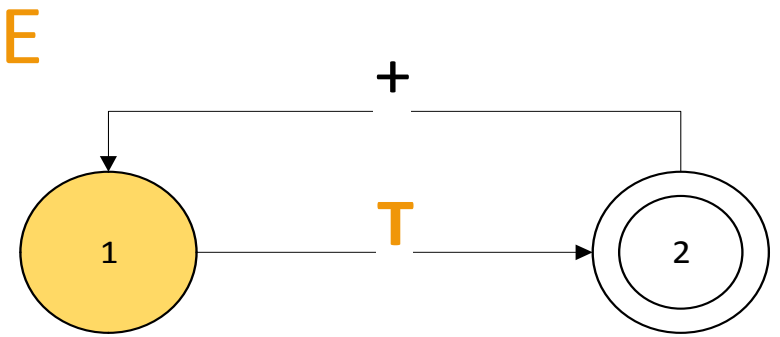
id + id * id $

PS

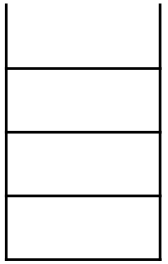




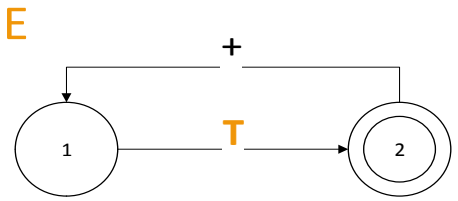


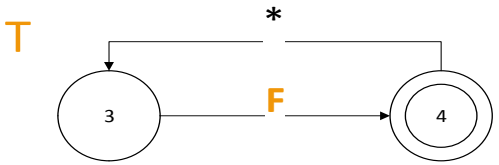


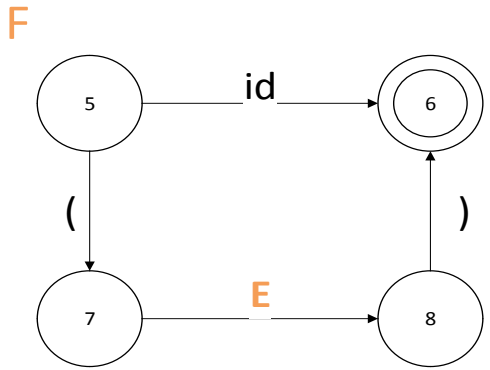
id + id * id $

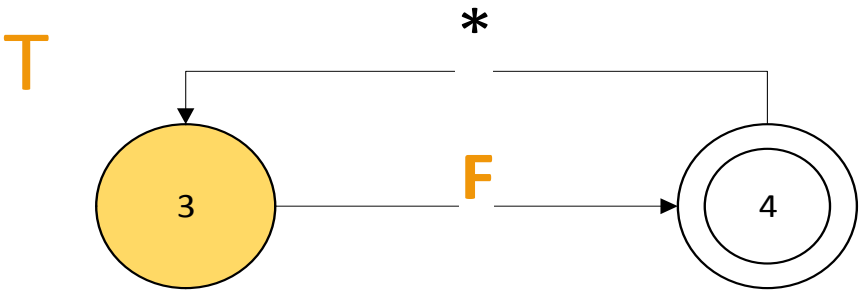


PS

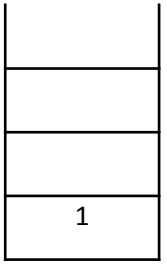




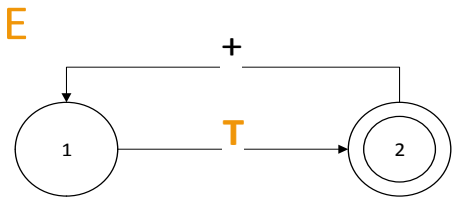


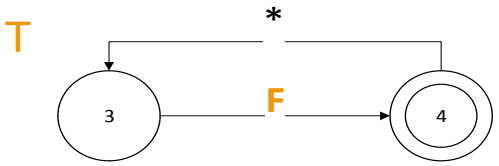


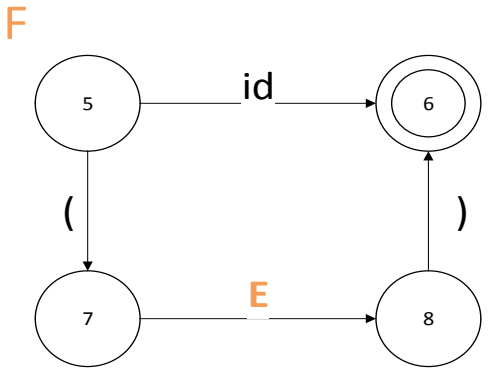
id + id * id $

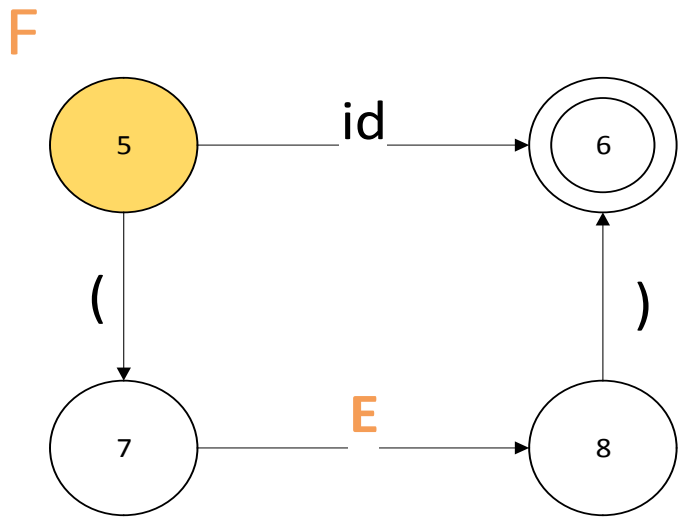


PS

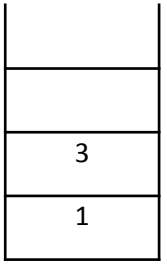






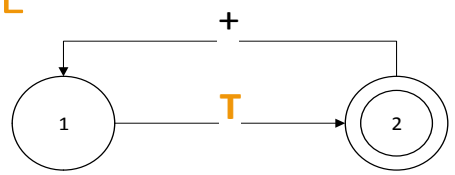


id + id * id $

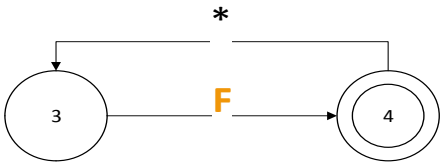


PS

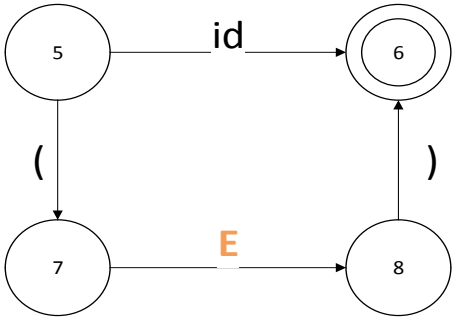
E



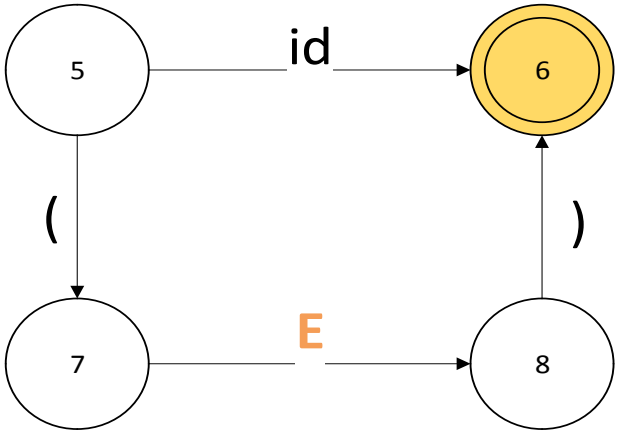
T



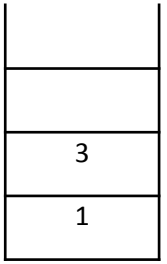
F



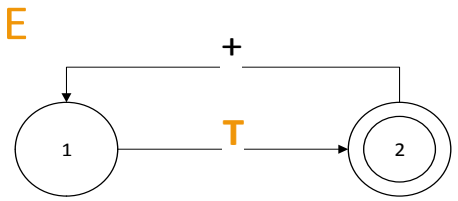
F

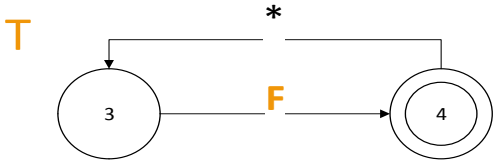


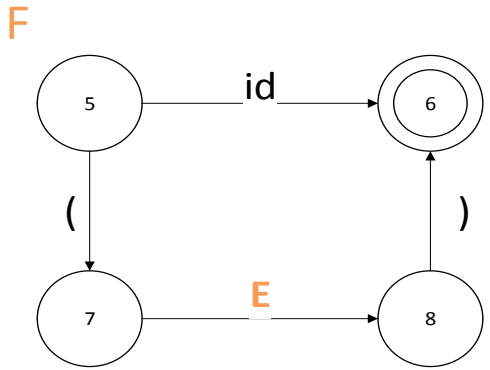
id + id * id $

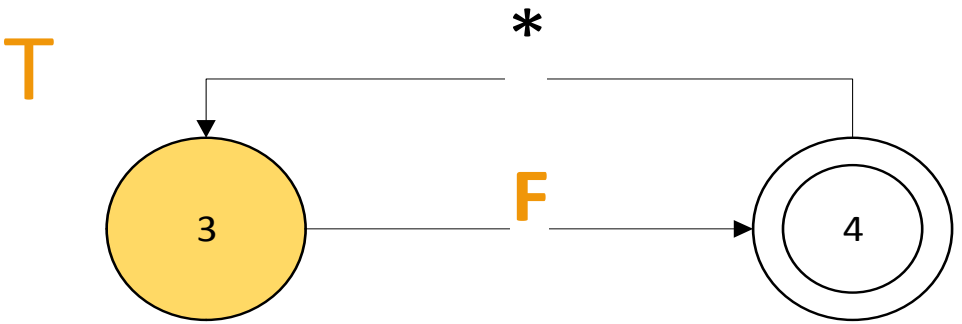


PS

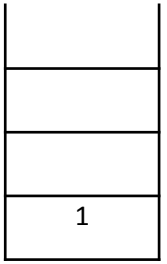






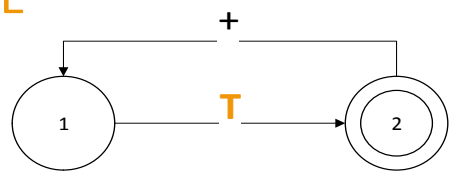


id + id * id $

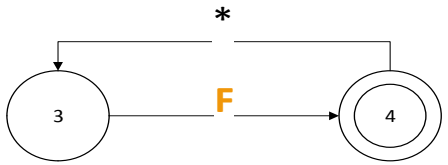


PS

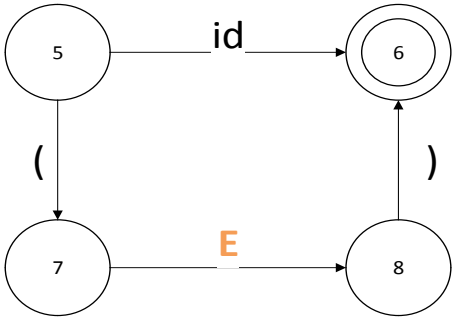
E



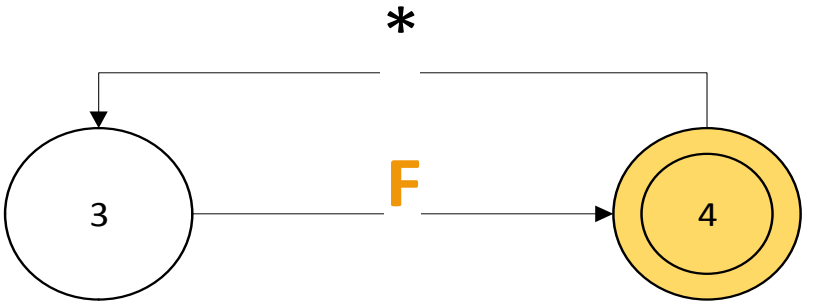
T



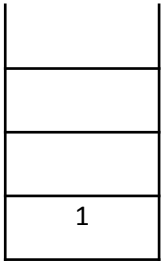
F



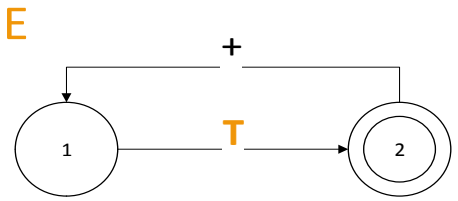
T

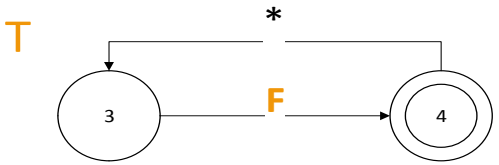


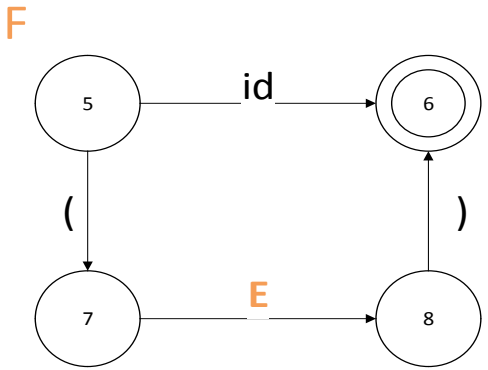
id + id * id $

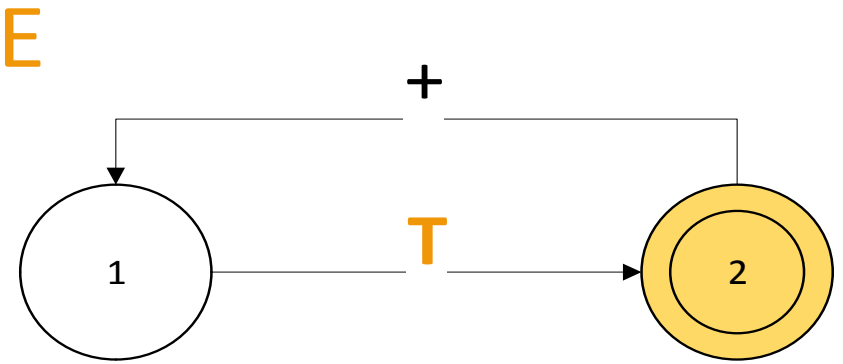


PS



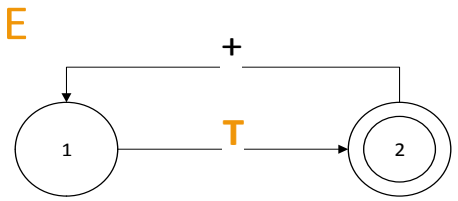


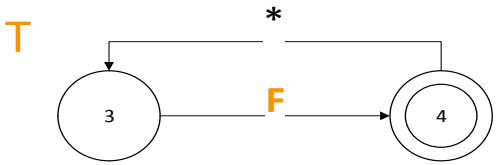


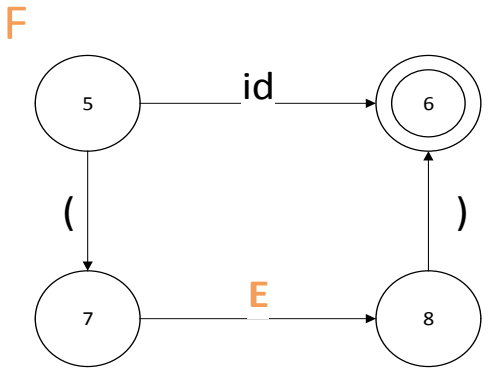


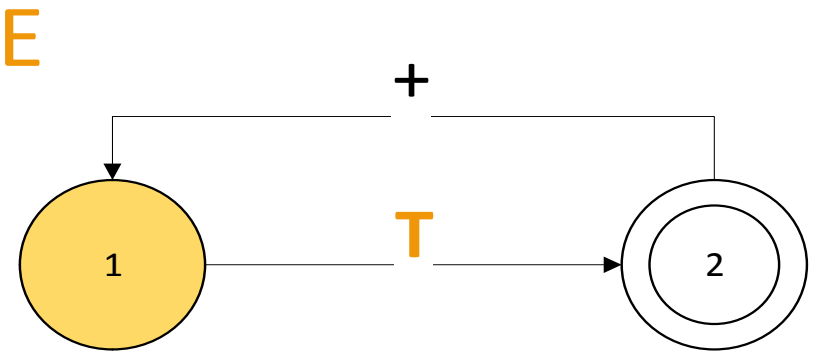
id + id * id $

PS



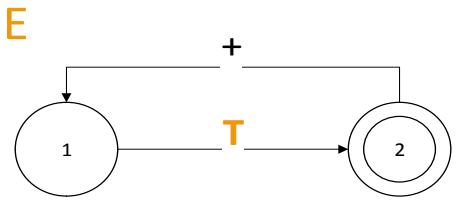


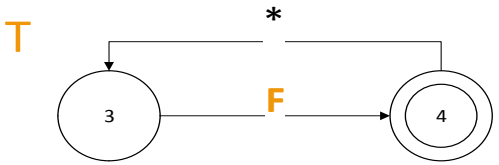


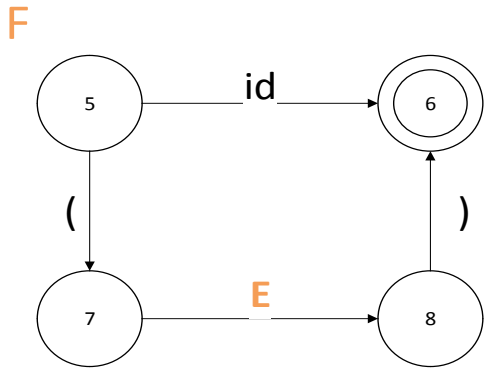


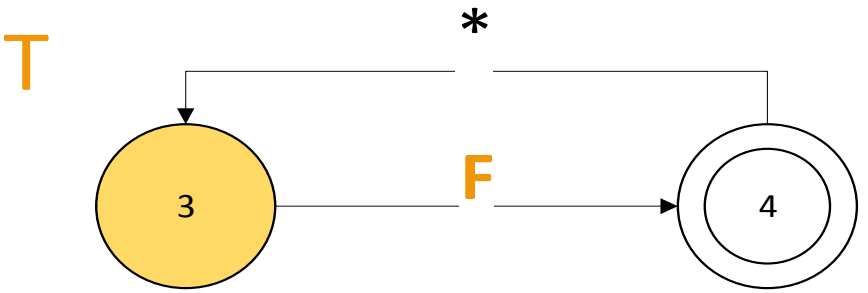
id + id * id $

PS

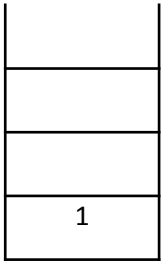






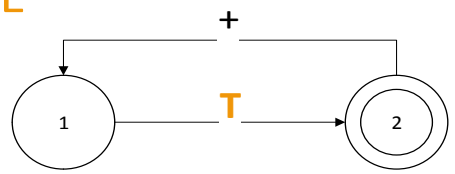


id + id * id $

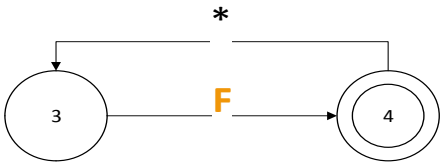


PS

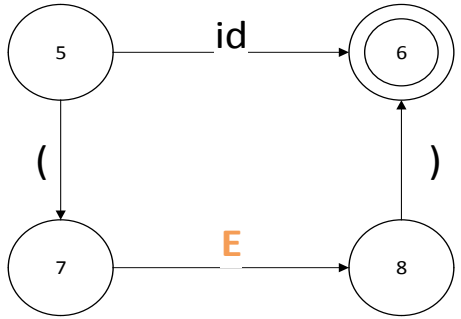
E



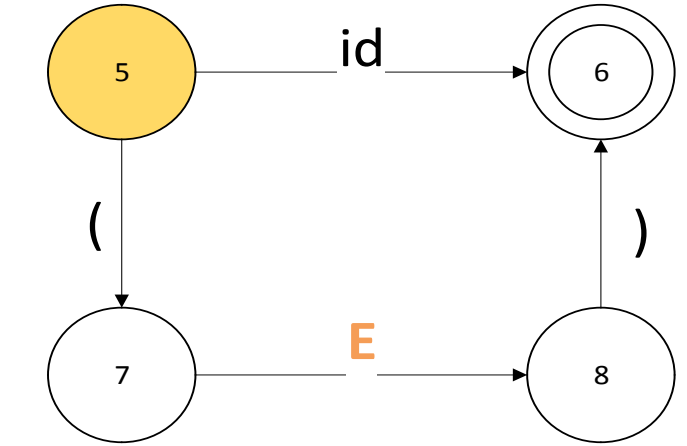
T



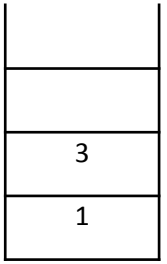
F



F

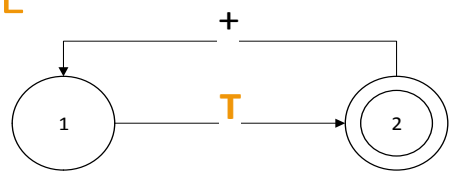


id + id * id $

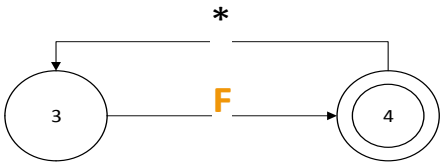


PS

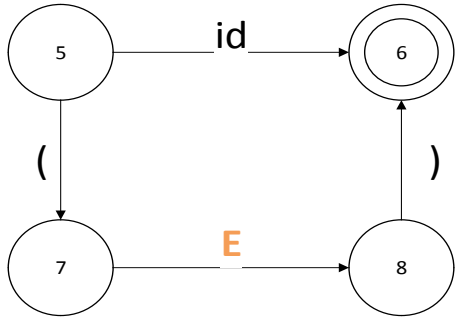
E



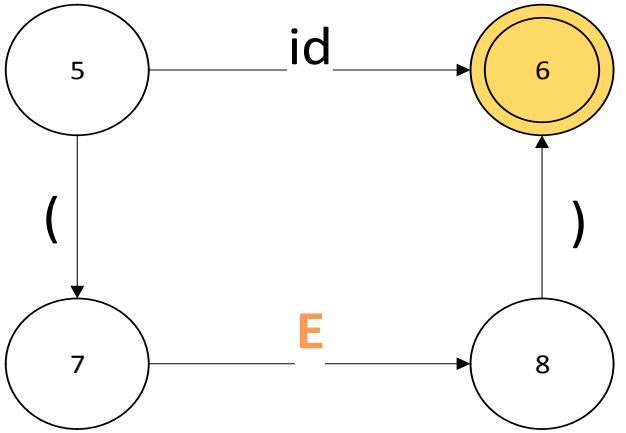
T



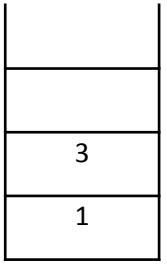
F



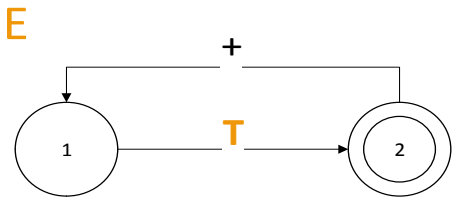
F

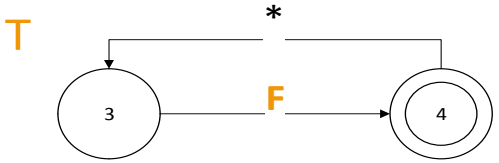


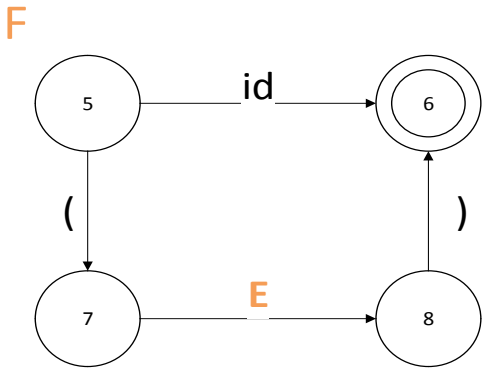
id + id * id $

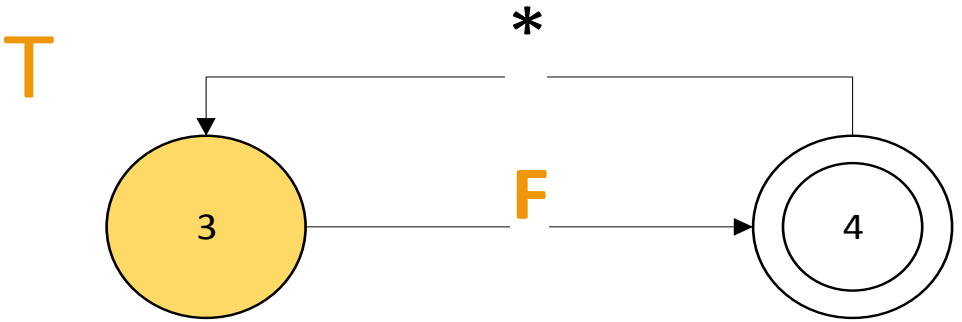


PS

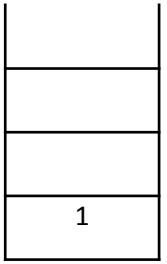




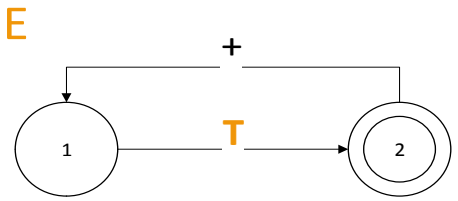


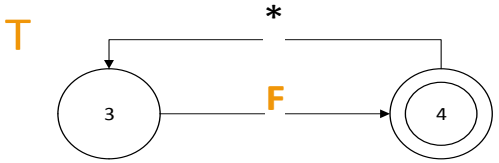


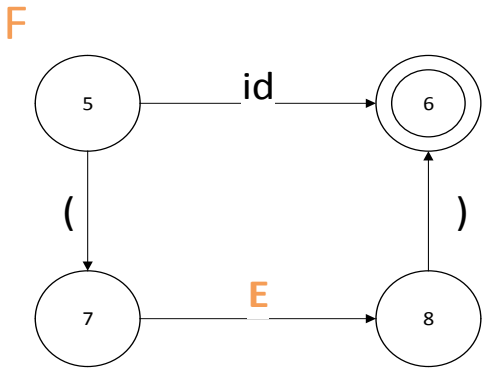
id + id * id

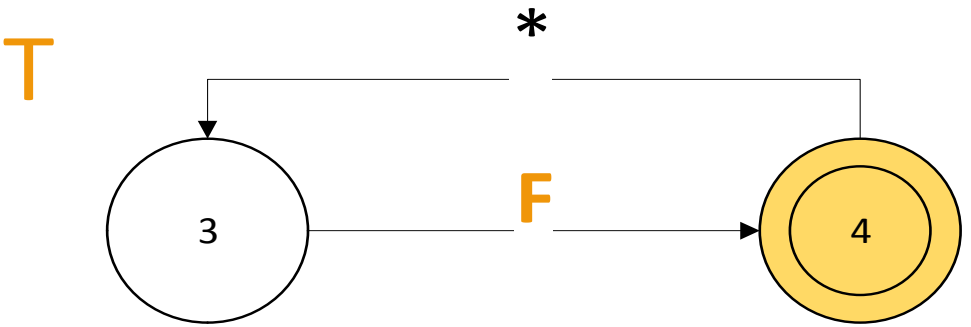


PS

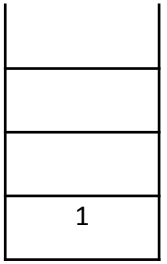




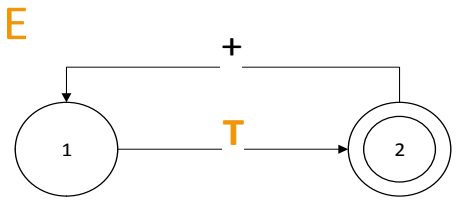


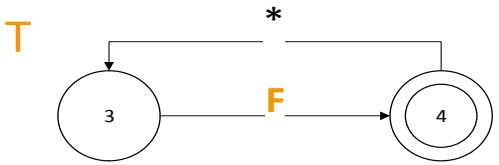


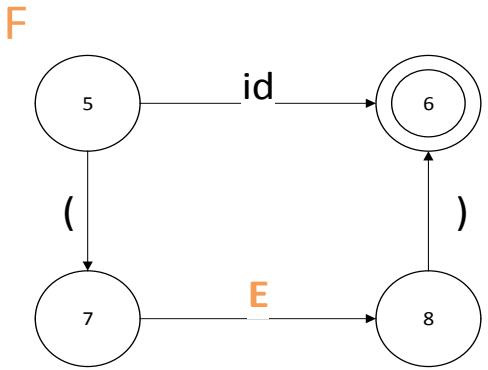
id + id * id $

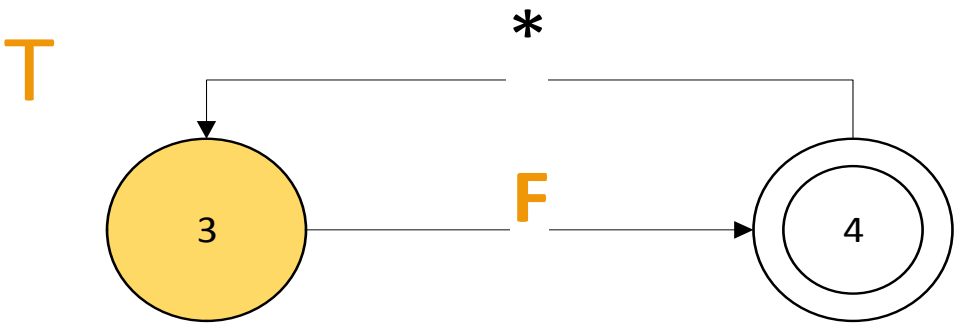


PS

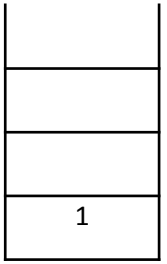




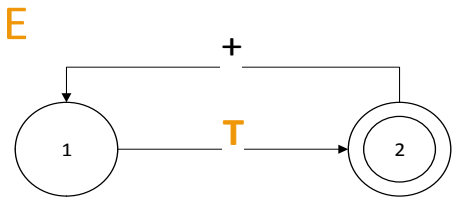


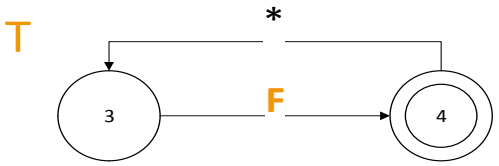


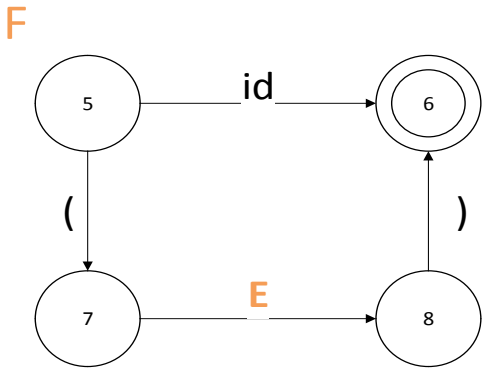
id + id * id $

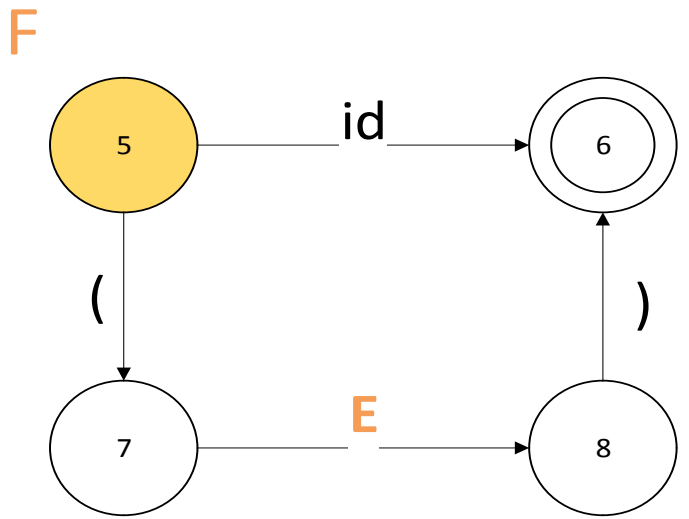


PS

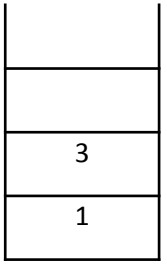






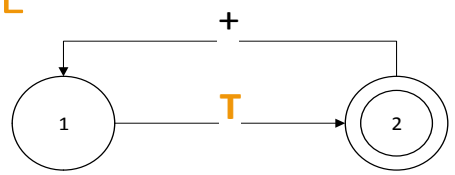


id + id * id $

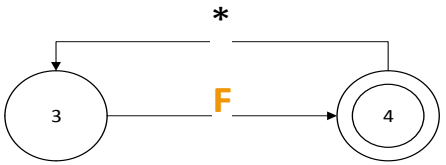


PS

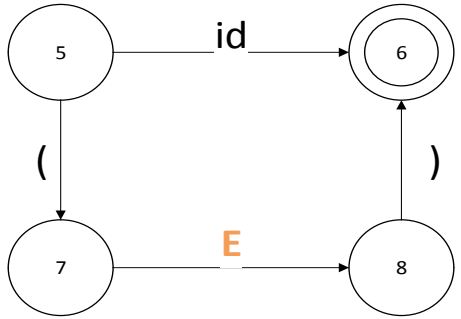
E



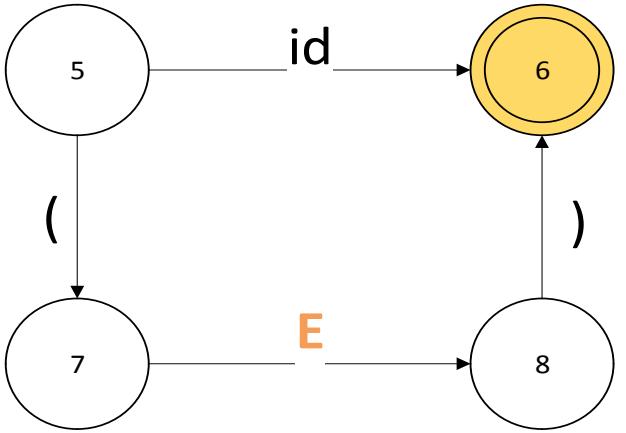
T



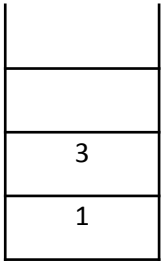
F



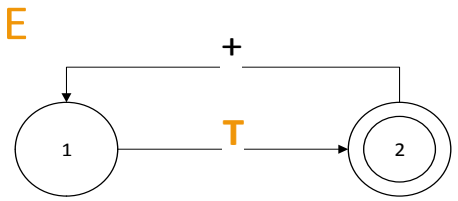
F

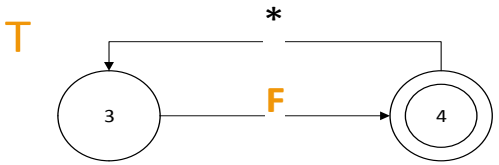


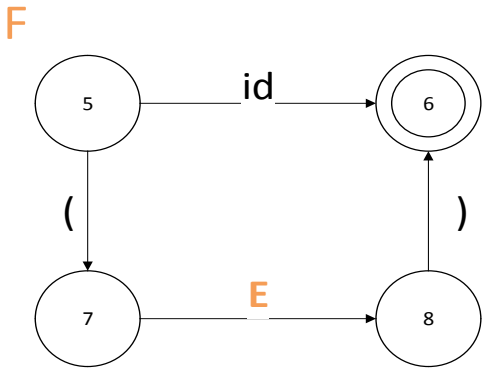
id + id * id $

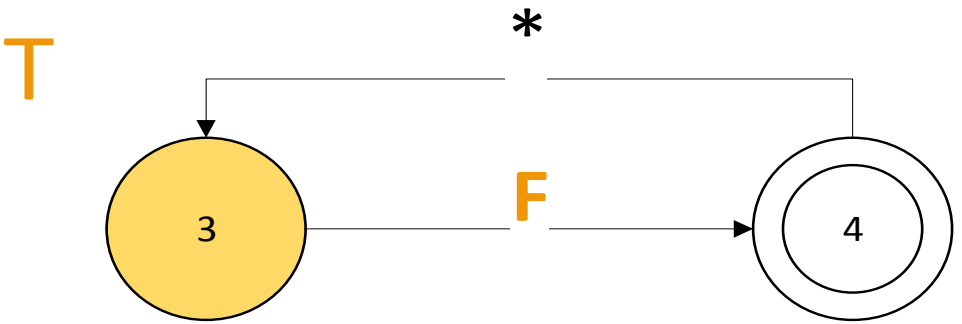


PS

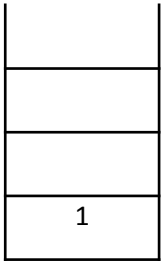




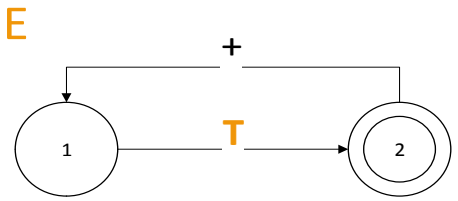


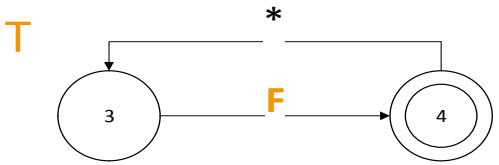


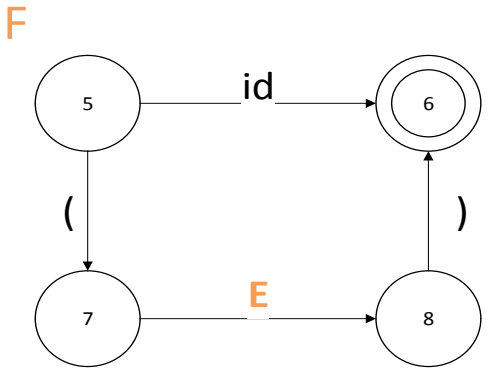
id + id * id $

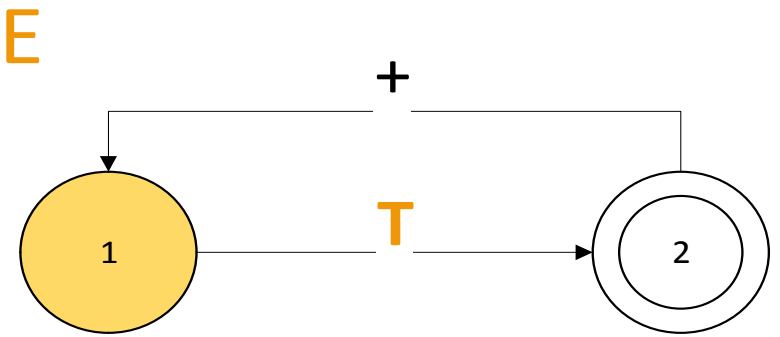


PS



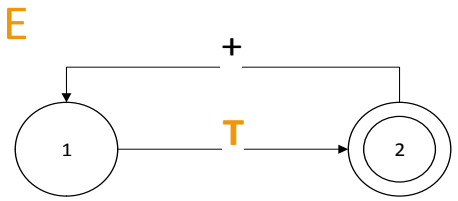


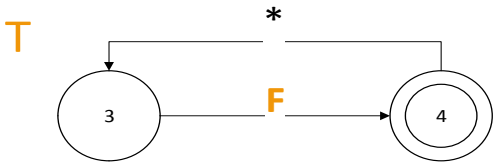


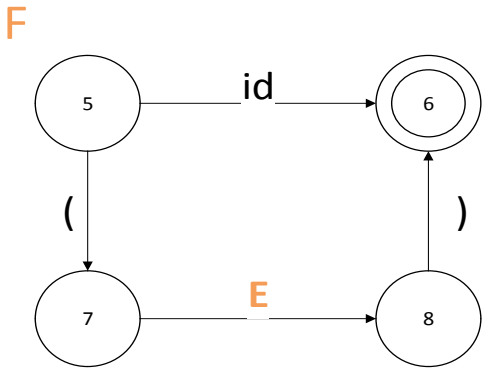


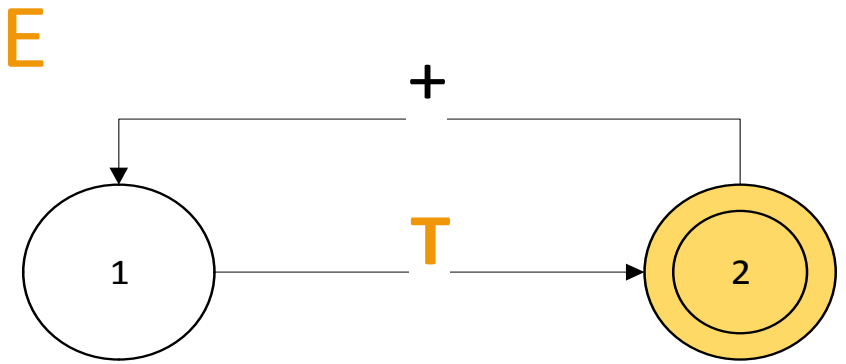
id + id * id $

PS







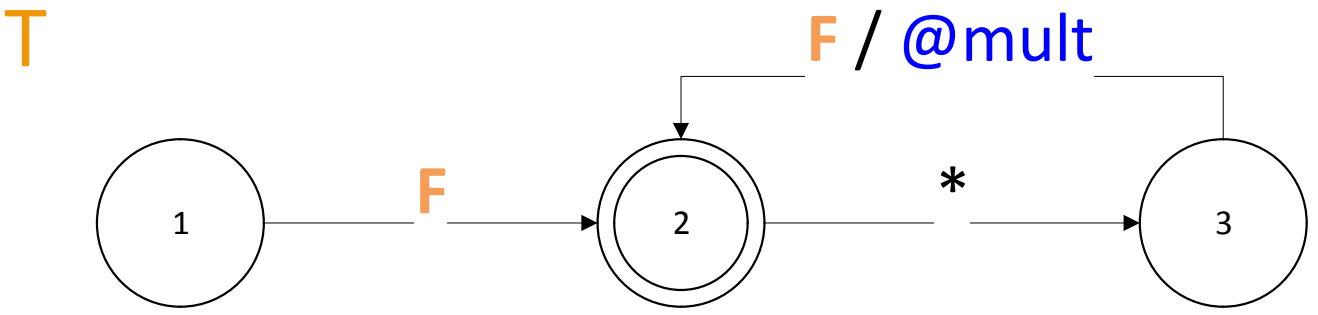


id + id * id $

PS

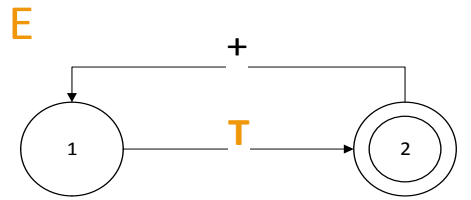
Semantic Routines

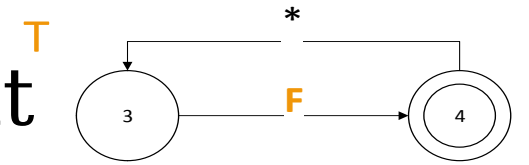
- When operands of an action prepared, we call **semantic routines** to generate the code (we'll talk about in details in IR Generation part)
- Lets imagine we have just add and change the graph E. On the edge labeled with @add two operands are ready.

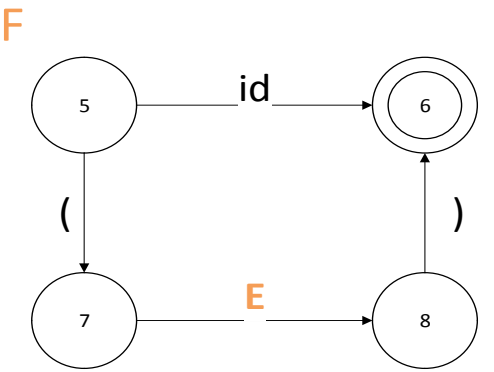


Theoretical Aspect

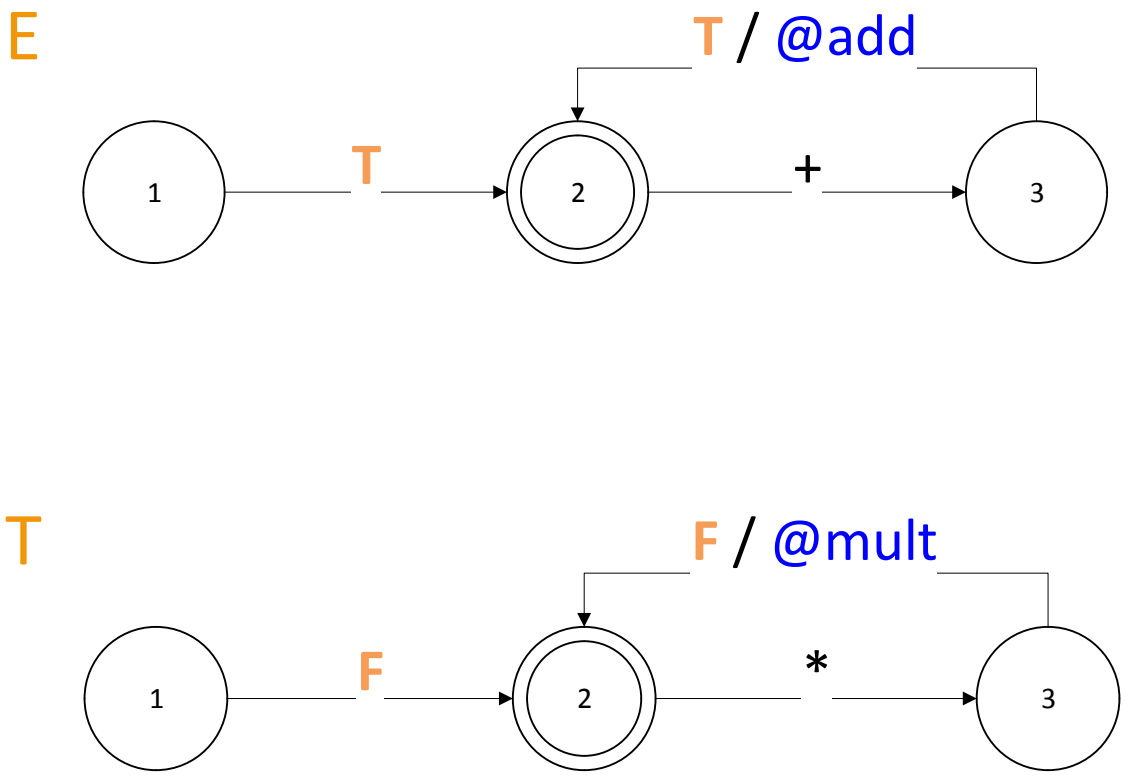
- It is modeled with DPDA.
- It can be managed by epsilon moves.
- Instead of graph name, we'll have e-move to first node of that graph.
- From final states we have moves to all other states.
- You can manage it with stack symbols

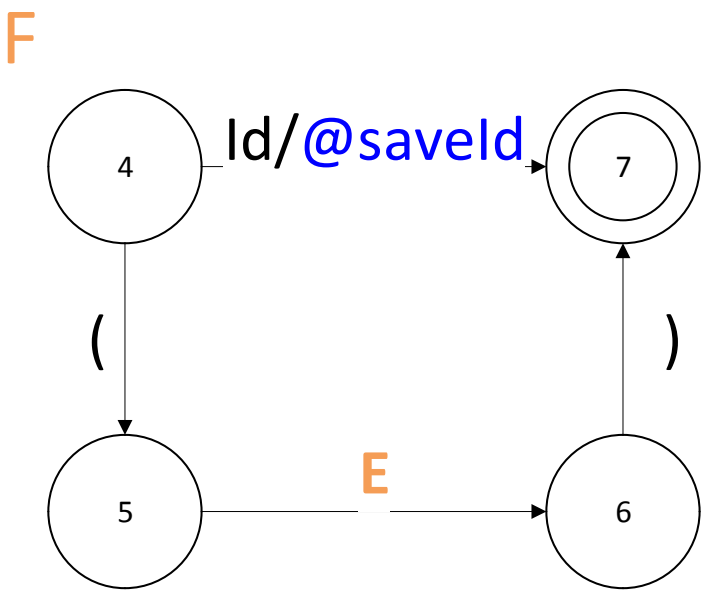






A few Changes

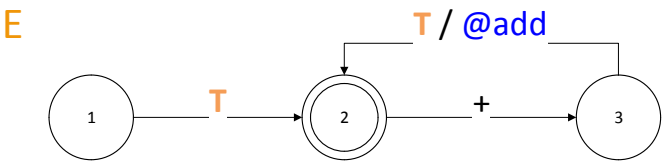


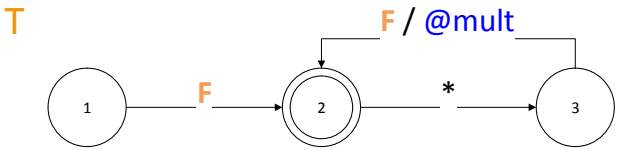


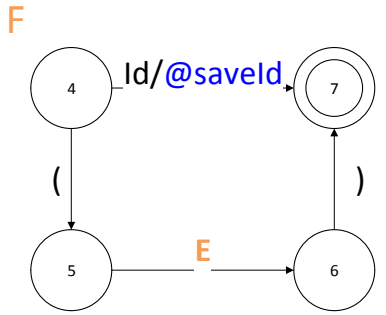
More on Semantics

- Code is generated with a semantic stack.
- With id we saveId on the SS.
- With * we have @mult.
- With + we have @add.
- We store result (or temporary location on the SS)

```
a + b * c EOF
```







Actions

- `@saveId:`
  - `SS.push(id)`
- `@mult:`
  - `t1 = SS.pop()`
  - `t2 = SS.pop()`
  - `t3 = getMem()`
  - `print(“mult, t1, t2, t3”)`
  - `SS.push(t3)`
- `@add: same as above.`

More on Semantics

- We print the code.
- getMem(), allocates a new memory place.
- By push, we insert some data entry on the stack. This data is strongly connected to symbol table's entry.
- We Check them more on semantic analysis part.
- We use intermediate code (more on that later).