Syntax Analysis

Table of Contents

[Syntax Errors 2](#_Toc138879070)

[Context-Free Grammar 3](#_Toc138879071)

[Parse Tree 4](#_Toc138879072)

[Grammar Ambiguity 4](#_Toc138879073)

[Types of Parsing 7](#_Toc138879074)

[Top-Down Parsing 8](#_Toc138879075)

[Eliminating Left Recursion 8](#_Toc138879076)

[Left Factoring 10](#_Toc138879077)

[Parse Table 13](#_Toc138879078)

[Predictive Parsing 15](#_Toc138879079)

[Bottom-Up Parsing 17](#_Toc138879080)

[LR Parsers 18](#_Toc138879081)

[SLR Transition Diagram 19](#_Toc138879082)

[SLR Parse Table Generation 21](#_Toc138879083)

The role of a **syntax analyzer** is to create a parse tree from the tokens passed to it by the lexical analyzer.

## Syntax Errors

**Syntactical Errors** are errors caused by invalid orders of tokens. For example, arithmetic expressions with unbalanced brackets, two consecutive operands or operators and the lack of a semicolon at the end of a statement are all errors that will be caught by the syntax analyzer.

The goal of the syntax error handler is to report the presence of syntactical errors clearly and accurately, recover from each error quickly and avoid slowing down the entire compilation process.

There are several error recovery strategies that are attempted:

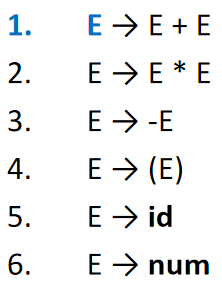
* **Panic Mode** – Delete input symbols one by one until a symbol is found that results in the error being resolved.
* **Phrase Level** – The compiler designer can specify some local corrections to attempt, such as replacing a symbol, inserting a symbol, deleting a symbol, etc.
* **Error Productions** – We can add grammar to the production rules that detect potential errors.
* **Global Corrections** – Ideally, we would like to make as few changes to the input as possible since the more changes we make, the less likely we are to obtain the intended code. There are algorithms that can detect which set of changes will lead to an acceptable code with the least changes.

## Context-Free Grammar

A **context-free grammar** consists of a set of terminals, a set of non-terminals, a start symbol and a set of productions.

* **Terminals** – These are the basic symbols from which the input strings are formed.
* **Non-Terminals** – Syntactic variables that denote sets of strings.
* **Start Symbol** – A special non-terminal from which the generated strings will start.
* **Productions** – The grammar that specifies how the terminals and non-terminals can be combined to form strings.

An example production is given below:



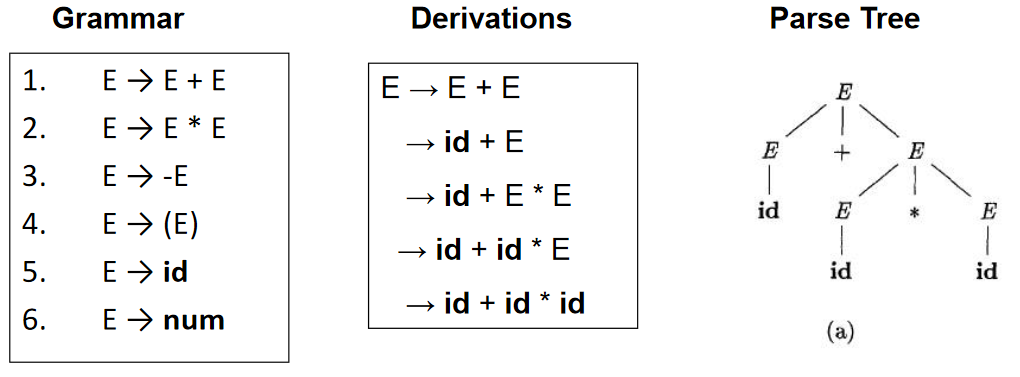
Here, , , , , , and are terminals, is a non-terminal and is the start symbol.

Productions essentially tell us how something on the left-hand side can be replaced using something on the right-hand side. This replacement process is called **derivation**.

By convention, terminals use lower case letters, operators, punctuation, digits and boldface strings while non-terminals use uppercase letters. The start symbol is usually denoted as . Usually, the first production’s left-hand side symbol is the start symbol. Lowercase Greek letters may also be used to represent strings of grammar symbols.

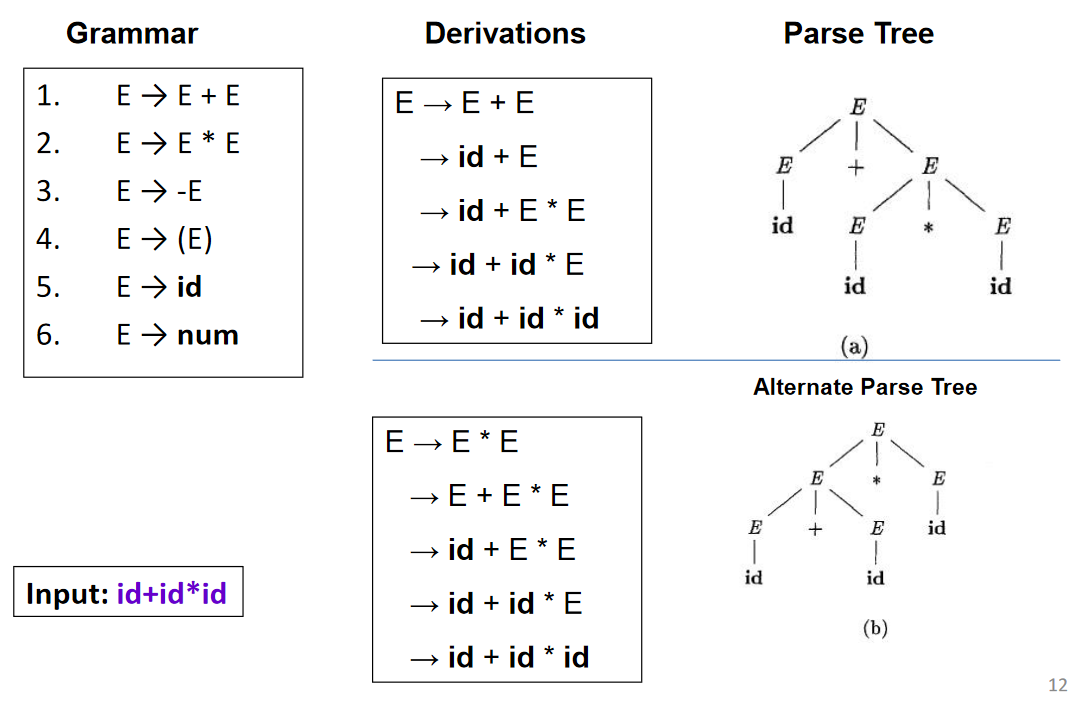
## Parse Tree

The derivation process can be visually represented using a **parse tree**.



## Grammar Ambiguity

There are cases where a grammar can create multiple parse trees for the same input.



Grammars that can result in situations like this are said to be **ambiguous**. Suppose we have the string ‘2+3\*4’. The correct output for this is 14. However, the two parse trees above will result in two different answers. The first one will process the input as ‘2+(3\*4)’ while the second will process it as ‘(2+3)\*4’.

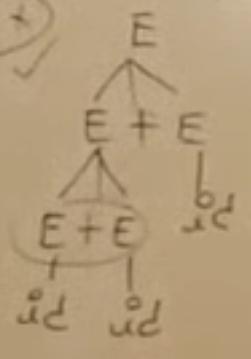
There are no direct methods or algorithms that can **eliminate the ambiguity**. It is a manual process that we need to check using different inputs. There are ways we can go about doing this though.

Consider why we ran into the issue in the first place in the example above. The **precedence** of the operators was not respected by the grammar. We should always deal with multiplication first, but the grammar does not specify a rule for this so we end up being confused about whether to prioritize the multiplication or the addition.

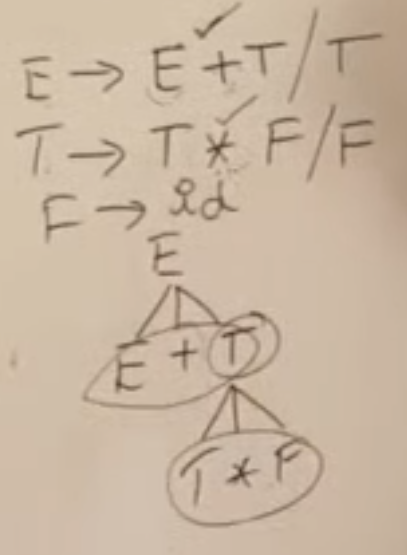
Another possible ambiguity would be for the string ‘2+3+4’. There is an operand (3) that can be associated with either the operator on the left-hand side or the one on the right-hand side. Depending on which we do, we end up with a different parse tree. Even though the results are the same, the grammar is still ambiguous. In such cases, we should choose the **left-hand side**. This will lead to correct results.

The way we can integrate these rules into the grammar so that it becomes unambiguous is as follows:

1. The parse tree derivation should be **left recursive**, i.e., the left-most non-terminal should be expanded more. Left-recursive grammars have the recursive symbol (the same symbol) on the left-hand side.



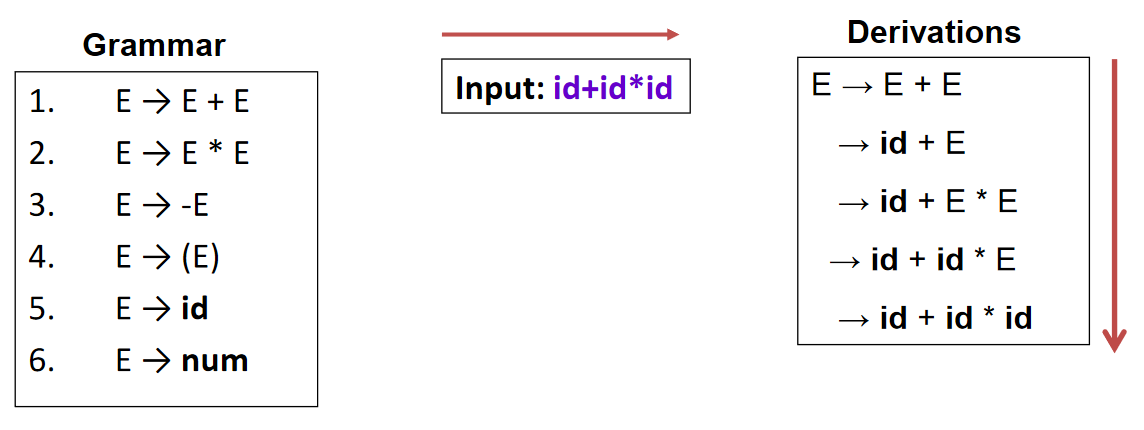
1. Add non-terminals to the grammar in a way such that there is no choice but to respect the priority of the operators. The operators with lower priority should come first and depend on the outputs of the operators with higher priority. In the example below, it is not possible to computer without first computing , thus preserving the priority.



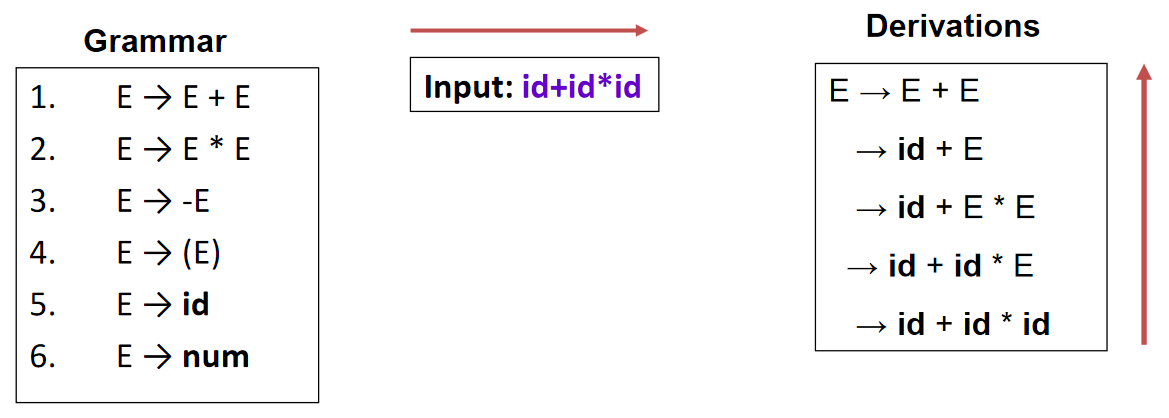
## Types of Parsing

There are mainly two methods of parsing a string, **Top-Down Parsing** (LL Parsing) and **Bottom-Up Parsing** (LR Parsing). There are subcategories to each of these types which we will be looking into later.

In LL Parsing, we scan the input from left to right and use the left-most derivation.



In LR Parsing, we scan the input from left to right and use the right-most derivation.

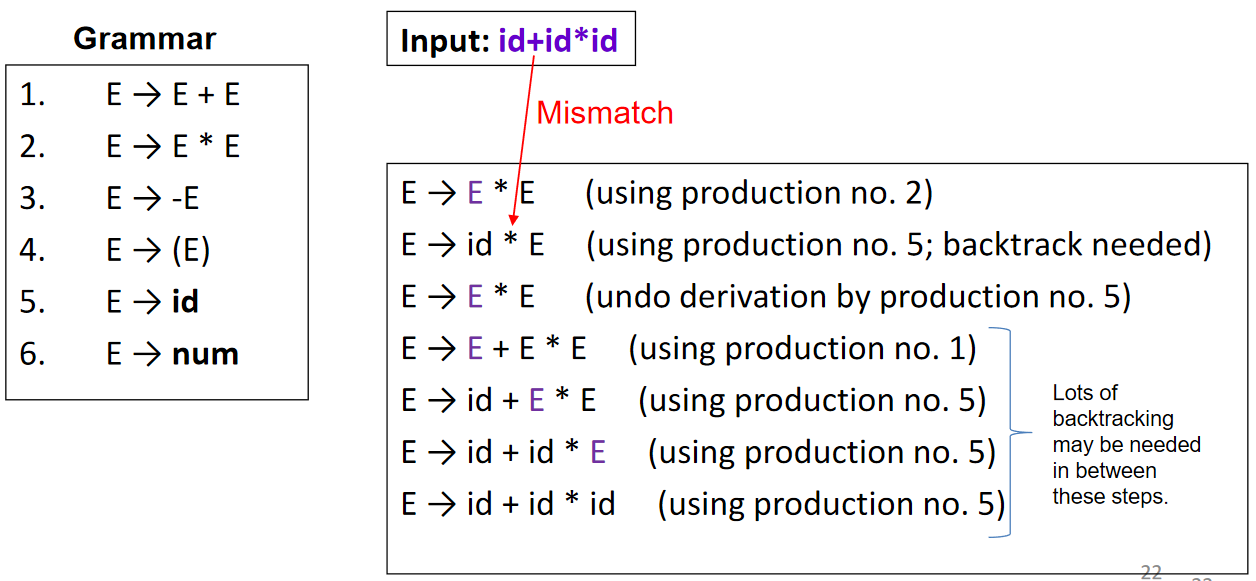


Each of the steps during the derivation process creates a string that is said to be in **sentential form**. If the string is the result of right-most derivation, it is said to be in **right-sentential form**. If the string is the result of left-most derivation, it is said to be in **left-sentential form**. A **sentence** is a sentential form consisting only of terminals.

## Top-Down Parsing

There are two types of **Top-Down Parsing**, Recursive Descent Parsing and Non-recursive Predictive Parsing.

In **recursive descent parsing**, we randomly pick a production rule to derive a non-terminal. If we hit a dead end, we backtrack.

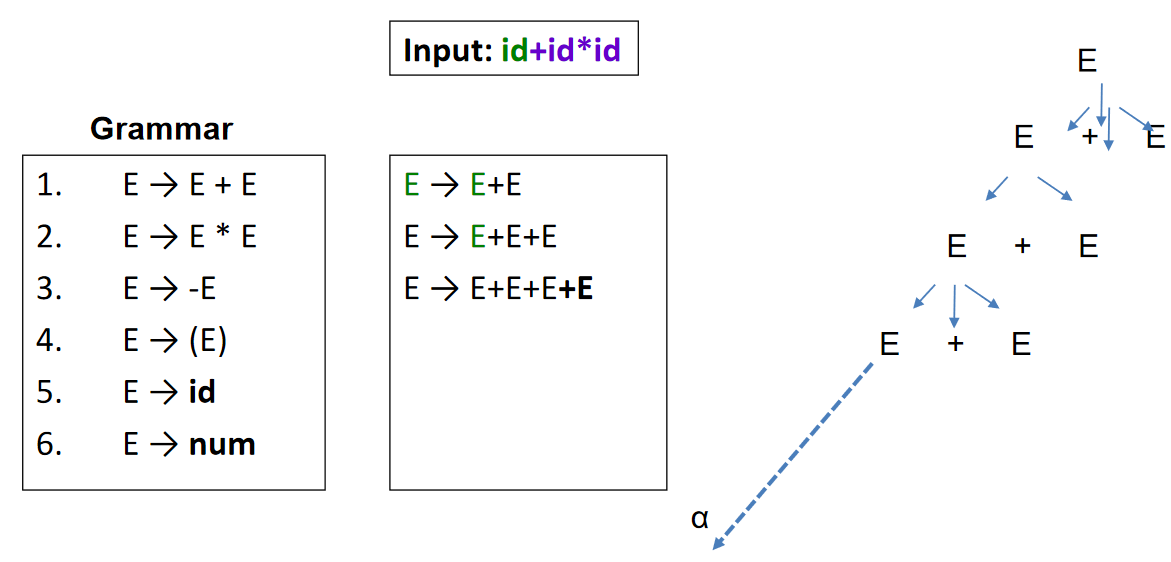


In **non-recursive predictive parsing**, we attempt to predict the correct production rule to use at each stage. This is a complicated process that requires many steps.

The first step to non-recursive predictive parsing is to modify the grammar in preparation of predictive parsing. This involves two steps, elimination of left recursion and performing left factoring.

### Eliminating Left Recursion

If we are not careful about choosing our production rules, we can fall into an **infinite recursion**.



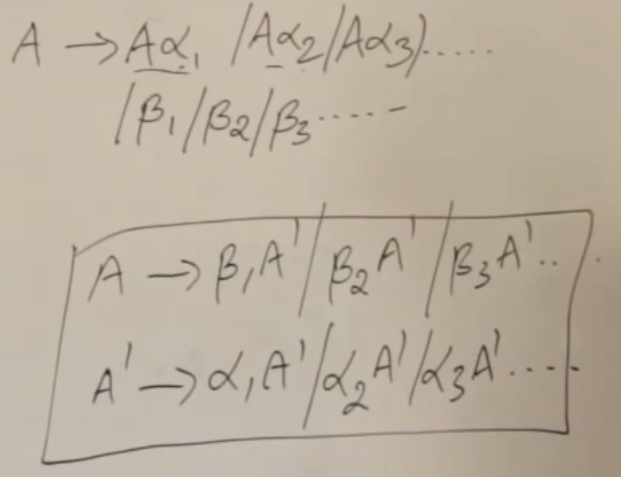
We need to eliminate this possibility without affecting the language represented by the grammar.

Suppose we have the following grammar:

This grammar can trap us in the infinite recursion loop. In reality the grammar is generating sentences of the form . This means that the start of the grammar must always be . Thus, we can re-write the grammar as follows:

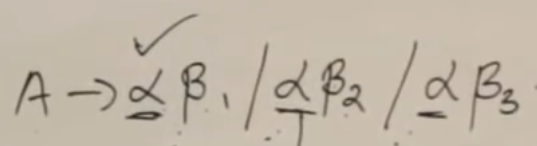
We can use this format to fix other grammars as well.

Even if we have multiple productions that have left recursion in them, it is just an expansion of the same process.

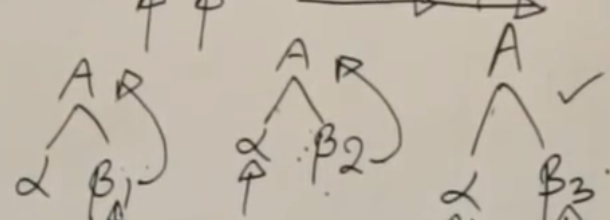


### Left Factoring

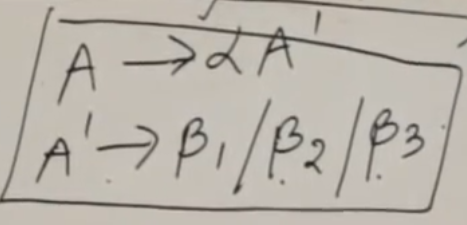
Suppose we have a set of productions where the first symbol of the right-hand side are the same, i.e., they have **common prefixes**.

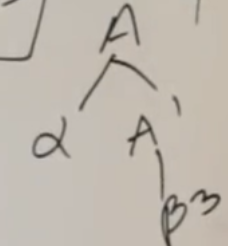


This can cause issues since we will not know which production to use. Suppose we have the string . We might first try , fail, backtrack, try , fail, backtrack and finally get to . The repeated backtracking is happening because of the common prefixes.



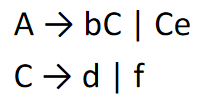
This issue occurs because we are taking the decision too early. We are deciding which production to use based on just the first symbol when the decision is affected by the second symbol as well. To resolve this, we can delay the point at which we have to decide to a later stage so that the backtracking issue does not occur. This is called **left factoring**.





#### First and Follow

Before we can dive into predictive parsing, we need to understand two functions, first and follow. The **first** of a **non-terminal symbol** is the set of symbols that can be the starting point for every string derived from that symbol. Suppose we have the following grammar:

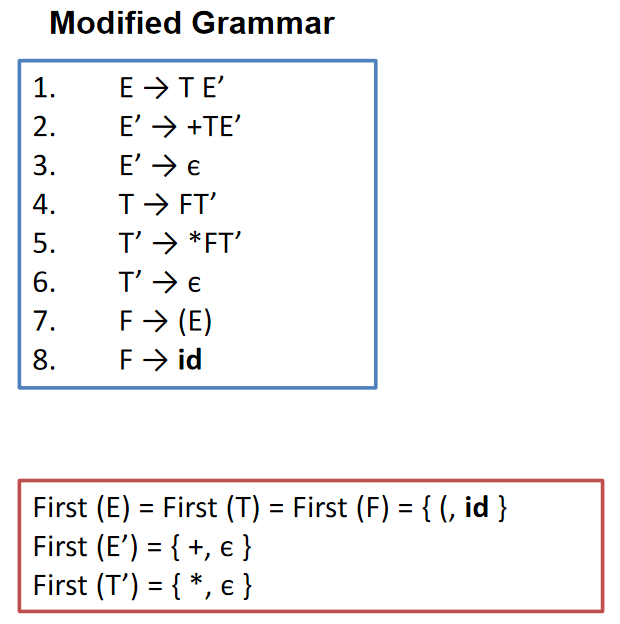


For this grammar, and .

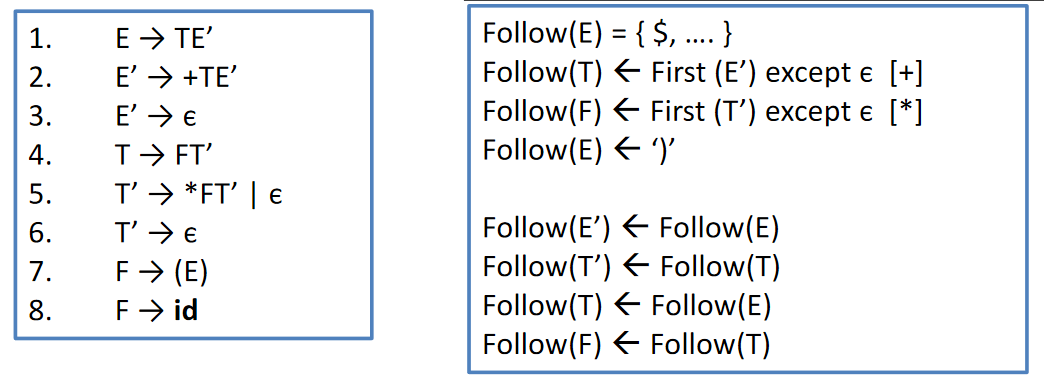
For **terminal symbols**, the first is the symbol itself, i.e., .

One caveat to the First function is when appears. Suppose we have this grammar:

In this case, if , the first symbol actually becomes the first symbol of , so . However, .



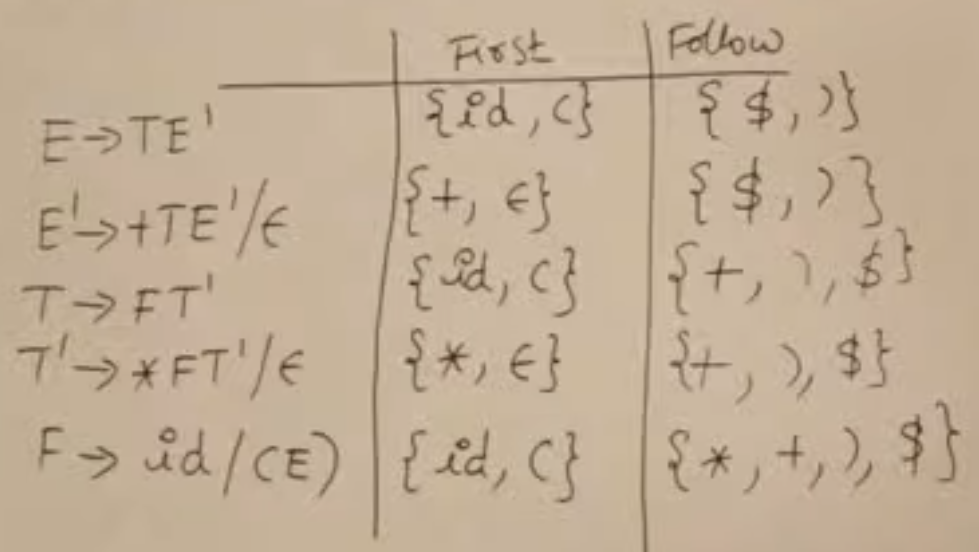
The **follow** of a non-terminal is the set of **terminals** that appear immediately after the symbol. For the grammar above, . However, , the symbol used to indicate the end of the input. This is because nothing follows the starting symbol and does not appear anywhere else in the grammar. The follow of any symbol can never contain .



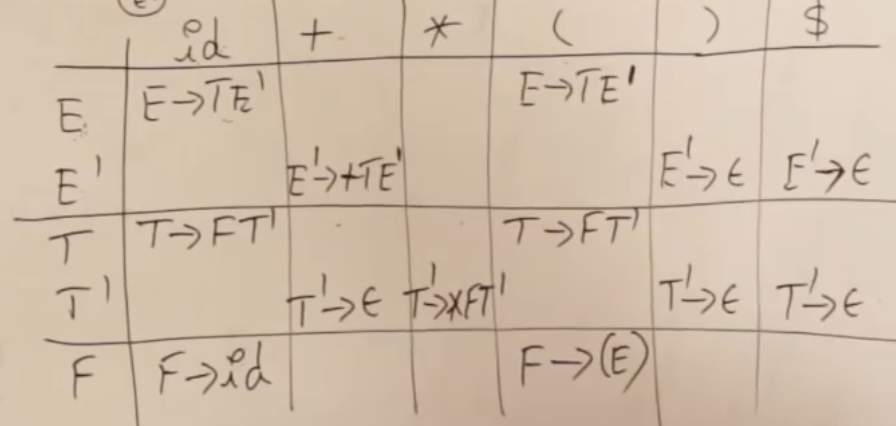
In the example above, notice that . This is because the string can be repeated multiple times. If the string isn’t followed by anything, then , but this might not be the case.

### Parse Table

The reason we need to know the first and follow of each symbol is to generate a **parse table**, which will be used during predictive parsing. Suppose we have the following grammar.



We can use these values to determine which productions to use when we find a particular symbol. This can be determined using . The mapping is the parse table.



There are several things we need to consider here. The table has the **terminals** as **columns** and **non-terminals** as **rows**. We are also considered to be a terminal but not .

For each terminal and non-terminal pair, we write down the **production rule** that has the terminal as the first for the non-terminal. For example, , to the cell for and has the production .

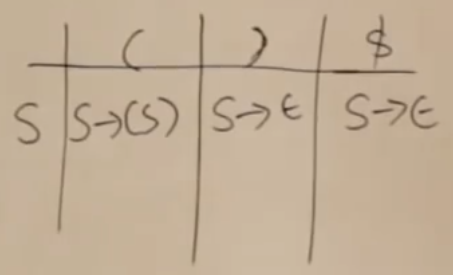
The only time we look at the Follow column is if the First column contains an . If it does, then the terminals in the Follow column should have the production for the corresponding non-terminal. For example, and , so the cells for and and and both contain the production . This makes sense since using during parsing may allow us to get the and .

### Predictive Parsing

Finally, we can look into how the actual parsing process works.

Consider that we have the following grammar:

For this grammar, the parse table is:



We now want to parse the string .

To do this, we use a **stack**. Initially, the value in the stack is .

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

For each symbol we encounter in the input, we remove a terminal from the stack and replace it with its derivation. The derived symbols are added in the reverse order.

Input:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

If the top of the stack matches the input, we remove it and move to the next input symbol.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Input:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Input:

An production will just remove the non-terminal.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Input:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

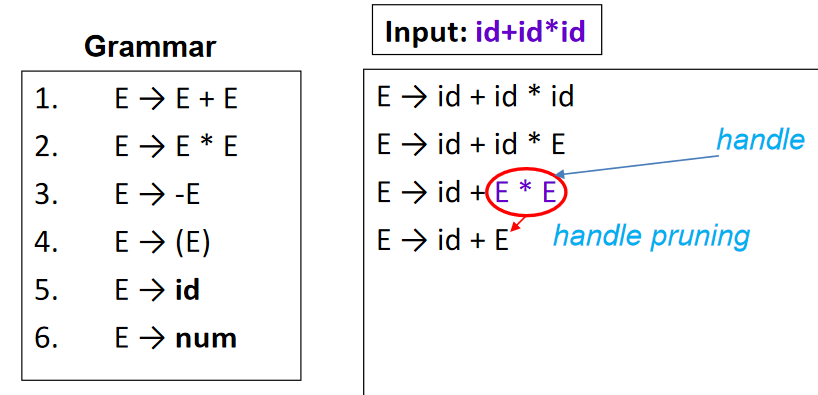
There are however, various scenarios under which we cannot use this parsing method. If a grammar causes **multiple entries** in the parsing table, we cannot use it.

## Bottom-Up Parsing

There are four types of **Bottom-Up Parsing**: Operator Precedence Parsing, Simple LR (SLR), Canonical LR (CLR) and Look Ahead LR (LALR).

During parsing, the prefix of the right-sentential form may appear in the stack. This prefix is called the **viable prefix**.

A **handle** is a substring of the right-sentential form which matches a production rule. Handles always appear at the top of the stack. When we replace the handle with the non-terminal on the left-hand side of the production, the process is called **handle pruning**.

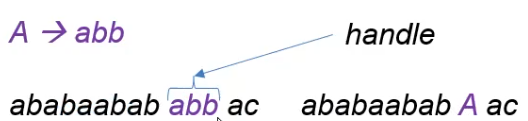


### LR Parsers

All of the **LR parsers** we will be studying work in the same way as the predictive parser we saw earlier. The only change will be in how the **parsing table** is generated.

LR Parsers make **shift-reduce** decisions. They maintain states, which are generated from the grammar, which are used to track where we are in the parsing process. **Shift** refers to shifting to a new state, while **reduce** refers to applying a reduction operation using a handle, i.e., handle pruning.

To identify if a reduction operation should be done, we check if the next input matches .



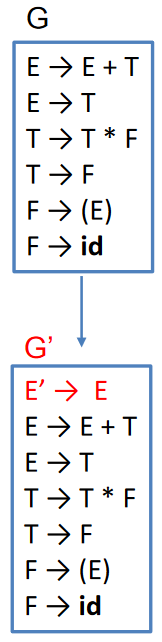
Consider that we have the production rule . It is possible to place a dot () at different positions of this rule as shown below.

Thus, this single production gives us four different states.

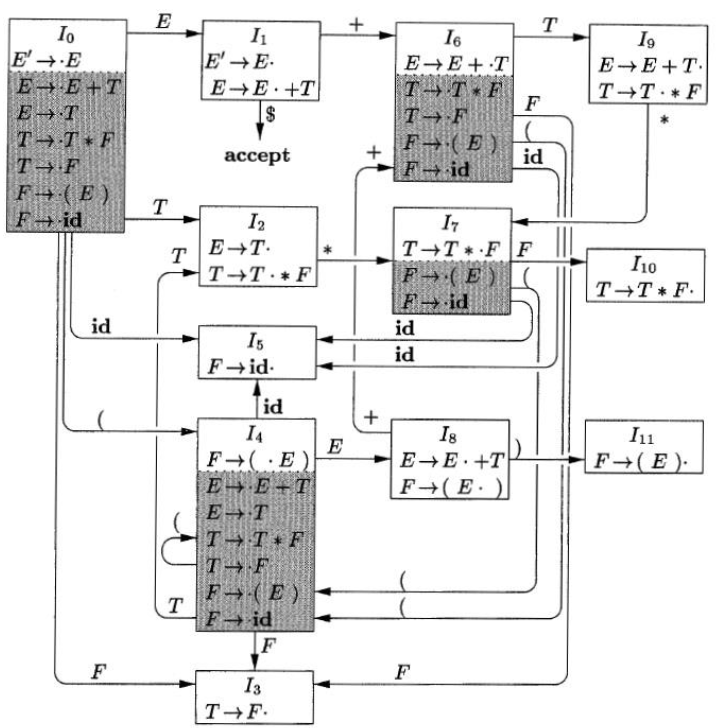
The dot signifies how much of the right-hand side production we have ‘seen’. The dot allows us to identify when we have seen the entire right-hand side of the production. A reduction should only be performed if the entire right-hand side has been seen.

### SLR Transition Diagram

The first step to understanding how the states change is to augment the production rules. The rules below are a problem, since there are two start states. We need to modify this so that there is only one start state.



These rules are then used to generate a **transition diagram**.



Each of the states in the diagram is called an **LR(0) item**. Each state contains an initial production, called the **kernel item**. In addition, we also add the productions for the non-terminals which are on the right of the dot. For example, the state starts with , so we add the production of , which in turn allows us to add the production of and so on. The productions we add are called the **closure items** of the original production, given by .

Each of the points at which there is a dot, we consider the symbol on the right of the dot and create a new state, one where the dot is after the symbol. For example, we see that there are two productions with . These two productions are used to create the state . This same process is repeated for all of the production rules.

Notice that if the symbol on the right of the dot in the new state is a **non-terminal**, we need to add the closure of the production. This is not the case for terminals.

### SLR Parse Table Generation

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State | ACTION | | | | | | GOTO | | |
| Id | + | \* | ( | ) | $ | E | T | F |
| 0 |  |  |  |  |  |  | 1 | 2 | 3 |
| 1 |  |  |  |  |  | acc |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  | 8 | 2 | 3 |
| 5 |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  | 9 | 3 |
| 7 |  |  |  |  |  |  |  |  | 10 |
| 8 |  |  |  |  |  |  |  |  |  |
| 9 |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |  |

The **SLR Parse Table** is generated using the transition diagram. For the **action** column, the cell value is the new state to which we should go given the current state and the terminal which is found on the right of the dot. For the **GOTO** column, we do the same for the non-terminals.

For the ACTION column, represents a new state and represents the accept state. We also have values. Here, refers to the th production rule. If a state has a production where the dot is at the end of the production already, we mark all of the terminals in using that production. For example, in state 2, we have the closure item , which comes from the second production, . Thus, we place in all of the non-terminals which are in .

During this processing, if we encounter multiple values in a single cell, the grammar is **ambiguous** and it is not possible to parse strings for this grammar. During processing a string, if we somehow end up in an empty cell, the string is not correct for this grammar.