Syntax Analysis(Parser)

A Syntax analyzer is formally defined as :

> An Algorithm that Groups the Set of Tokens Sent by the Scanner to Form **Syntax Structures** Such As Expressions, Statements, Blocks,etc.

Simply put, the parser examines if the source code written follows the grammar(production rules) of the language.

The Syntax structure of programming languages and even spoken languages can be expressed in what is called **BNF** notation, which stands for **B**akus **N**aur **F**orm.

For example, in spoken English, we can say the following:

> sentence --> noun-phrase verb-phrase
>
> noun-phrase --> article noun
>
> article --> THE | A | ...
>
> noun --> STUDENT | BOOK | ...
>
> verb-phrase --> verb noun-phrase
>
> verb --> READS | BUYS | ....

Note : The BNF Notation uses different symbols, for example, a sentence is defined as :

> < sentence > ::= < noun-phrase > < verb-phrase >

But this is very cumbersome, so we use the first notation, since its easier to use.

Now, let us derive a sentence :

> sentence --> **noun-phrase** verb-phrase
>
> --> **article** noun verb-phrase
>
> --> THE **noun** verb-phrase
>
> --> THE STUDENT **verb-phrase**
>
> --> THE STUDENT **verb** noun-phrase
>
> --> THE STUDENT READS **noun-phrase**
>
> --> THE STUDENT READS **article** noun
>
> --> THE STUDENT READS A **noun**
>
> --> THE STUDENT READS A BOOK

In the same way, the parser tries to **derive** your source program from the starting symbol of the grammar.

Lets say we have these sentences :

> THE BOOK BUYS A STUDENT
>
> THE BOOK WRITES A DISH
>
> THE DISH TAKES A STROLL

Syntax-wise, all of these sentences are correct. However, their meaning is not correct, and they are not useful. What differentiates 2 sentences that are grammatically correct is their meaning or their **semantics**. You and I can agree that the meaning of a grammatically correct sentence is not correct, but how does the computer do it?

Grammar

> A grammar G=(V_N, V_T, S, P) where:
>
> 1. V_N : A finite set of nonterminals(nonterminals set).
> 2. V_T : A finite set of terminals(terminals set).
> 3. S ∈ V_N : The Starting symbol of the grammar.
> 4. P = A set of **production rules**(productions).<-- Pending <==> Basically the whole grammar.

Note :

1. V_N ∩ V_T = ∅.
2. V_N ∪ V_T = V(the vocabulary of the grammar).

Note : We will use

1. Uppercase Letters A,B,...,Z for non-terminals.
2. Lowercase Letters a,b,...,z for terminals.
3. Greek letters α,β,γ,... for strings formed from V_N OR V_T = V. eg,

   if

   > V_N = {S,A,B}

   and

   > V_T = {0,1}

   then

   > α = 0A11B
   >
   > β = S110B
   >
   > γ = 0010

Productions

1. A Production α --> β(alpha derives beta) is a rewriting rule such that the occurrence of α can be substituted by β in any string.

   Note that α must contain at least one nonterminal from,∈V_N.

   For example, Assume we have the string γασ,

   > γασ --> γβσ

2. A Derivation is a sequence of strings α_0, α_1, α_2, α_3,....,α_n, then :

```
- &alpha;<sub>0</sub> -*-> &alpha;<sub>n</sub>, n &ge; 0.

- &alpha;<sub>0</sub> -<sup>+</sup>-> &alpha;<sub>n</sub>, n &ge; 1.
```

Given a grammar G, then :

> L(G) = Language Generated By the Grammar.

for example, Given the Grammar, G = ({S,B,C},{a,b,c},S,P)

P :

> S --> aSBC

> S --> abC
>
> CB --> BC
>
> bB --> bb
>
> bC --> bc
>
> cC --> CC

L(G)=?

Lets follow through on the derivations

> S --> a**bC** --> abc(all terminals) ∈ L(G) <--- A sentence
>
> S --> a**S**BC --> aa**bC**BC --> aabbcBC --> blocked, so we try another path
>
> S --> a**S**BC --> aab**CB**C --> aa**bB**CC --> aab**bC**C --> aabb**cC** --> aabbcc ∈ L(G) <--- A sentence
>
> S --> a**S**BC -->.....-->aaabbbccc ∈ L(G) <--- A sentence
>
> Therefore, L(G)={a^n,b^n,c^n| n ≥ 1}

As another Example, we have these productions

> E --> E+T <-- we can write the productions 1 and 2 as a single production E --> E+T | T
>
> E --> T
>
> T --> T*F
>
> T --> F
>
> F --> (E) <-- we can write the productions 5 and 6 as a single production F --> (E) | n
>
> F --> n

Lets follow through some derivations

> E --> **T** --> **F** --> n ∈ L(E)
>
> E --> **E**+T --> T+**T** --> T+**F** --> **T**+n --> **F**+n --> n+n ∈ L(E)
>
> E --> **E**+T --> **T**+T --> **F**+T --> n+**T** --> n+**F** --> n + (**E**) --> n+(**T**) --> n+(**T***F) --> n+(**F***F) --> n+(n***F**) --> n+(n*n) ∈ L(E)
>
> Therefore, L(G) = {Any arithmetic expression with * and + operations}, n is an operand here.

Note that, if we add the productions

> E --> E+T | E-T | T
>
> T --> T*F | T/F | T%F

We would have a language to express all arithmetic expressions with (*,\,+,-) operations.

Lets Take another Example(things in double quotes are terminals)

> Program --> block "#"
>
> block --> "{" stmt-List "}"
>
> stmt-List --> statement ";" stmt-List | λ
>
> statement --> if-stmt | while-stmt | read-stmt | write-stmt | assignment-stmt | block
>
> if-stmt --> "if" condition....
>
> while-stmt --> "while" condition.....
>
> ....

> ....
>
> read-stmt --> "read"
>
> write-stmt --> "write"

V_N = {Program, block, stmt-List, statement, if-stmt, while-stmt, read-stmt, write-stmt, assignment-stmt}

V_T = { "{", "}", "#", ";", "if", "while", "read", "write" }

Lets Follow through some derivations :

> Program --> **block** # --> { **stmt-list** } # --> { λ } #
>
> Program --> **block** # --> {**stmt-list**} # --> {statement ; **stmt-list**} # --> {statement ; statement ; **stmt-list**} # --> {statement ; statement ; λ} # --> {**statement** ; statement ;} # --> {**READ-statement** ; statement ;} # -->{READ ; **statement** ;} # --> {READ ; write-statement ;} # --> {READ ; WRITE ;} #

We can write this as

```
{ READ;
  WRITE;
}#
```

The language of this language is defined as

> L(G) = {Set of all programs that can be written in this language}.

This is only a simple example, of a simple language. For something more complex such as C or Pascal, there are hundreds of productions.

Algorithms for Derivation

> A Leftmost derivation is a derivation in which we replace the **leftmost** nonterminal in each derivation step.

> A Rightmost derivation is a derivation in which we replace the **rightmost** nonterminal in each derivation step.

For example, given the grammar

> V --> S R $
>
> S --> +|-|λ
>
> R --> .dN | dN.N
>
> N --> dN | λ
>
> V_N = {V,R,S,N}
>
> V_T = {+, - , ., d, $}

Lets follow through on the leftmost derivation

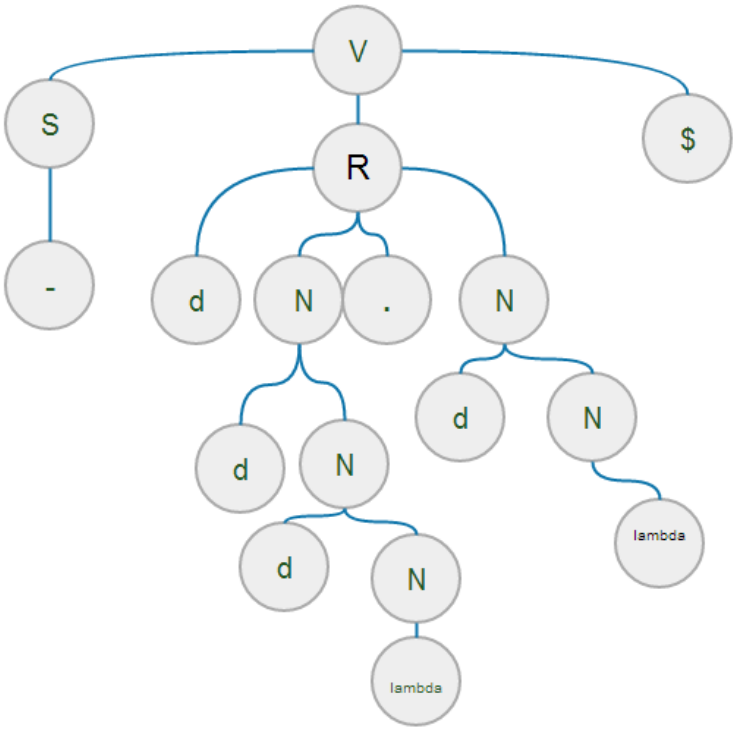
> V --> **S**R$ --> -**R**$ --> -d**N**.N$ --> -dd**N**.N$ --> -dddN.N$ --> -ddd.**N**$ --> -ddd.d**N**$ --> -ddd.d$ <-- A sentence.

Lets follow through on the rightmost derivation

> V --> S**R**$ --> SdN.**N** --> SdN.d**N**$ --> Sd**N**.d$ --> sdd**N**.d$ --> sddd**N**.d$ --> **S**ddd.d$ --> -ddd.d$ <-- A sentence.

Derivation Trees

A Derivation Tree is a Tree that displays the derivation of some sentence in the language. For example, lets look at the tree for the previous example



Note that if we traverse the tree in order, recording **only** the leaves, we obtain the sentence.

Classes of Grammars

According to Chomsky, There are 4 classes of grammars :

1. Unrestricted Grammars : No restrictions whatsoever except the restriction by definition that the left side of the production contains at least one nonterminal from V_S. This grammar is not practical and we cannot work with it.

2. Context-Sensitive Grammars : For each production α --> β, |α| ≤ |β|, ie , the **length of alpha(α)** is less than or equal to the **length of Beta(β)**. This means that in this class of grammar, there are no λ productions in the form A --> λ, since |λ| = 0 and A ≥ 1.

   They Say that Fortran has a context-sensitive grammar.

   It is very difficult to work with this class of grammars.

3. Context-Free Grammar(CFG) : Each production in this grammar class is of the form A --> α , where $A \in V_N$ and $\alpha \in V_*$ that is to say, the left hand side is **only** one nonterminal.

   This is the most important class of grammar. Most programming languages's structures are context-free.

   We will mostly be working with this class of grammar. Most of the examples we have taken are CFG.

4. Regular Grammar (Regular Expressions) : Each production in this grammar class is of the form A -- > aB or A --> a, where $A,B \in V_N$ and $a \in V_T$, **with the exception** of S --> λ

   For example, lets say we have the grammar

   > A --> aA
   >
   > A --> a
   >
   > Therefore, we get
   >
   > $G(L)=a^+$

   However, adding the production

   > A --> λ

   Results in the grammar

> G(L)=a*

Parsing Techniques

There are 2 main parsing techniques used by a compiler.

Top-Down Parsing

In Top-Down Parsing, the parser builds the derivation tree from the root(S : the starting symbol) down to the leaves(sentence).

In Simple words, the parser tries to derive the sentence using leftmost derivation. For example, say we have this grammar :

> V --> SR$
>
> S --> + | - | λ
>
> R --> .dN | dN.N
>
> N --> dN | λ

Lets examine if the sentence

> dd.d$

is derived from this grammar.

> V --> **S**R$ --> +**R**$ --> d**N**.N$ --> dd**N**.N$ --> dd.**N**$ --> dd.d**N**$ --> dd.d$

Therefore, this sentence is derived from the grammar.

However, this approach is very computationally intensive, and more importantly, this requires knowing the source code in advance. The Parser doesnt know which production it should select in each derivation statement. We will learn how to solve these issues later in the course.

Bottom-Up Parsing

In Bottom-Up Parsing, the parser builds the derivation tree from the leaves(sentence) up to the root(S : Starting Symbol). This type of tree, built from the leaves to the root, is called a B-Tree.

In Simple words, the parser starts with the given sentence, does **reduction**(opposite of derivation) steps, until the starting symbol is reached.

Note that the string λ is present everywhere in the string, and we can use it wherever we like.

Lets follow the reduction of the example given above.

> +dd.d$ --> +ddλ.d$ --> +ddN.d$ --> +dN.d$ --> +dN.dλ$ --> +dN.dN$ --> +dN.N$ --> +R$ --> SR$ --> V

Which means that the sentence is in the grammar.

Note that we can run into deadlocks here. say we took this path instead :

> +dd.d$ --> +ddλ.d$ --> +ddN.d$ --> +dN.d$ --> +dN.dλ$ --> +dN.dN$ --> **+dNR$ --> +NR$ --> SNR$** --> Deadlock

This technique also has a major problem : Which substring should we select to reduce in each reduction step?

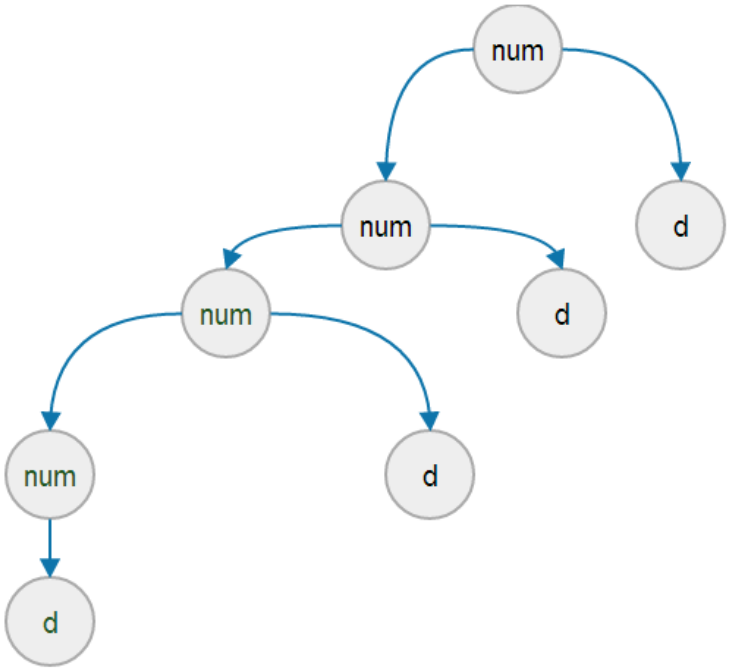
how do we solve this?

Ambiguity

Given the following grammar :

> num --> num d
>
> num --> d

Let us draw the derivation tree for the sentence `dddd`



Question : is there another derivation tree that represents the sentence?

The answer is **no**.

If there is only one derivation tree representing the sentence, this means there is only one way to derive the sentence.

Based on this, we can say that :

> A Grammar G is said to be ambiguous if there is one sentence with more than one derivation tree.
>
> That is, there is more than one way to derive the sentence.
>
> This means that our algorithm is **non-deterministic**.

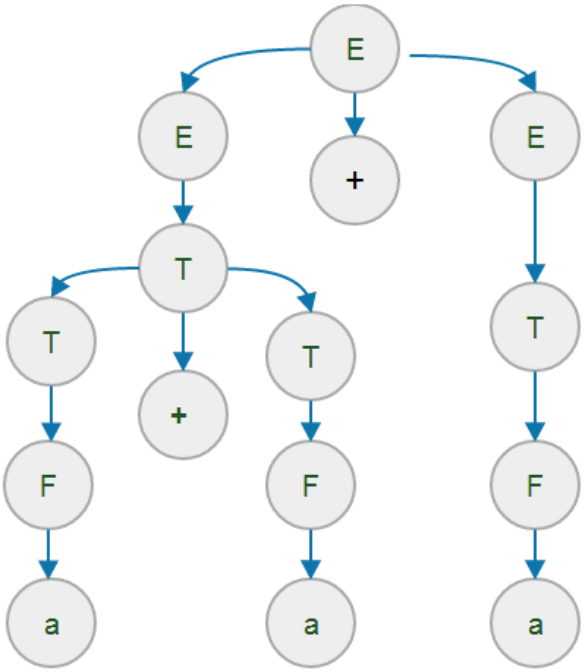
Say we have this grammar

> E --> E + E
>
> E --> E * E
>
> E --> (E) | a

Take the sentence :

> a + a * a

Lets draw the derivation tree



Due to the fact that we have 2 trees that give the same result, we can say that this grammar is ambiguous.

In this case, to enforce the associativity rule, this grammar can be re-written as :

> E --> E + E | T
>
> T --> T*T | F
>
> F--> (E) | a

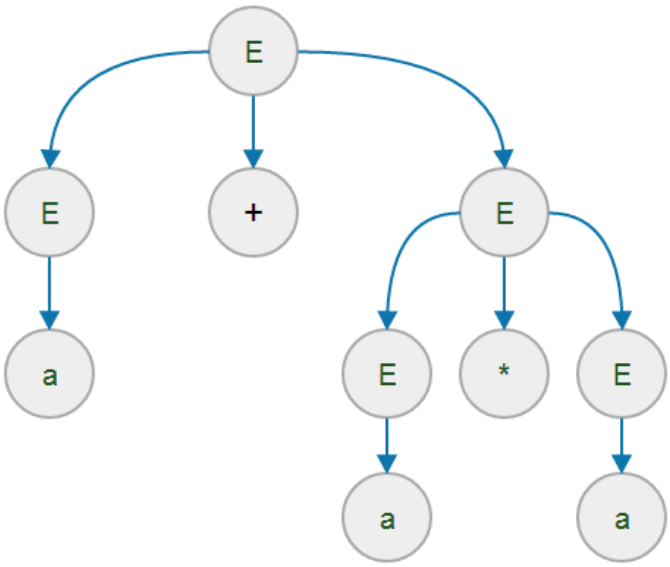
Now, Take the sentence `a + a * a` and find the derivation tree now. left

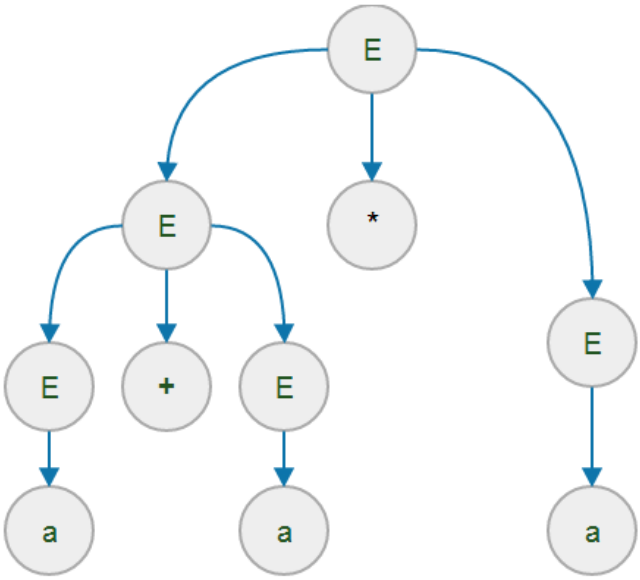
There is only 1 possible derivation tree now. This solves the associativity issue of the grammar before with the `+` and `*` operations.

But lets say we have the sentence :

> a + a + a

Lets try to find the derivation tree and any alternative trees.





We can see here that there is more than 1 derivation tree, and the language is still ambiguous.

We can solve this if we rewrite the grammar with the **left-associative rule**

> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

The resultant grammar is left-associative.

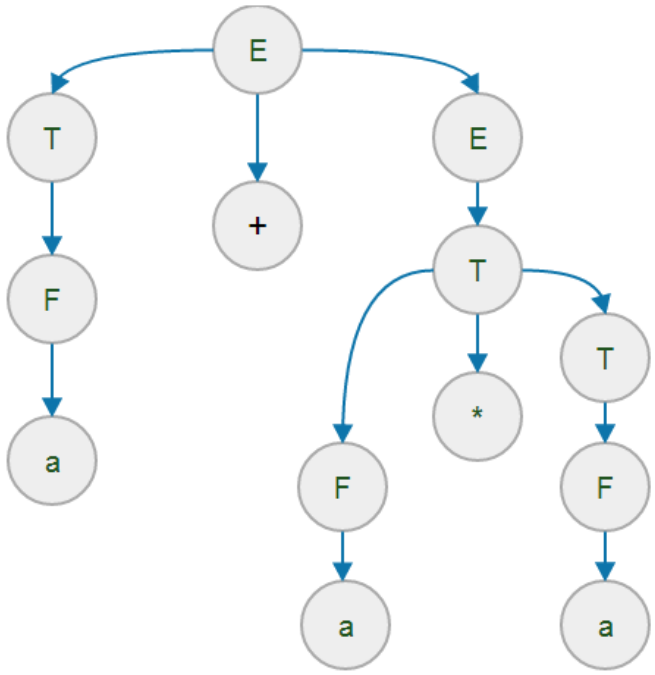
This grammar solves the problems of :

- ambiguity.
- precedence.
- associativity.

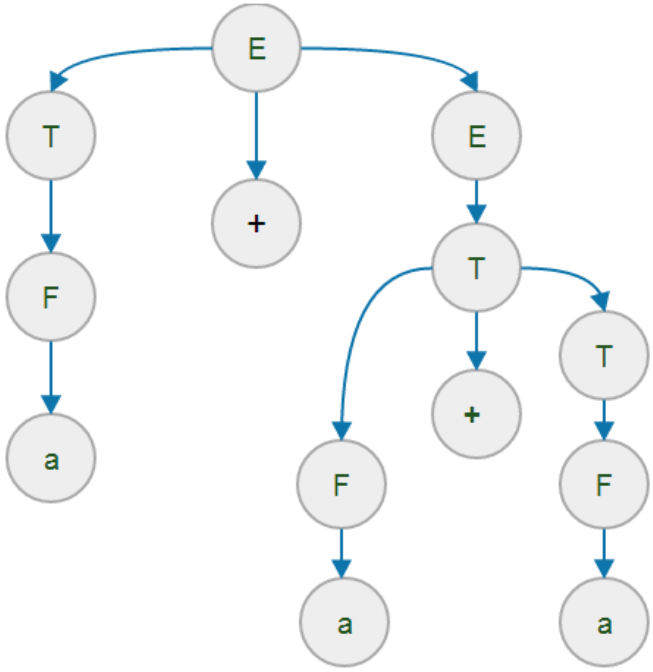
Lets try rewriting it with the **right-associative rule**

> E --> T + E | T
>
> T --> F * T | F
>
> F --> (E) | a

Lets try creating the derivation tree of `a + a * a`



Now lets draw the derivation tree of `a + a + a`



This new grammar is not ambiguious, however, as we can tell from the derivation trees, there are precedence issues now. It's not technically wrong, but it doesnt not follow standard arithmetic rules.

Back to the left-associative grammar now. This grammar is called **left-recursive**. This causes problems when it omes to top-down parsing techniques(we will see why later).

A grammar is said to be left recursive if there is a production of the form:

> A-->Aα

Conversely, a grammar is right-recursive if there is a production of the form:

> A-->αA

And causes no problems in top-down parsing.

our grammar has 2 rules of the form

> A-->Aα

The solution is to transform the grammar to a grammar which is not left-recursive.

This has an algorithm to it.

Given that

> A-->$A\alpha_1$| $A\alpha_2$|$A\alpha_3$|...|$A\alpha_n$
>
> A-->β_1|β_2|β_3|...|β_m

To do this, we must introduce a new non-terminal, say A`.

The grammar now becomes :

> A-->β_1A`|$\beta_2$A`|β_3A`|...|$\beta_m$A`

and

> A`-->$\alpha_1$A`| α_2A`|$\alpha_3$A`|...|α_nA`|λ

For example, say we have

> A-->Ab
>
> A-->a
>
> L(G)=ab*

Then according to the above

> A-->aA`
>
> A`-->bA`|λ

which results in the same grammar.

Lets apply this to the grammar :

> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

This results in :

> E --> T E`
>
> E` --> + T E` | λ
>
> T --> F T`
>
> T` --> * F T` | λ
>
> F --> (E) | a

This grammar is now **perfect**. It solves all our ambiguity issues, and this is a grammar we can use to construct the production rules for our programming language.

Another ambiguity in programming languages is the `if...else` statement.

Lets take a generic if statement in a generic language:

> stmt --> if-stmt | while-stmt | ....
>
> if-stmt --> IF condition stmt
>
> if-stmt --> IF condition stmt ELSE stmt

> condition --> C
>
> stmt --> S

This grammar is ambiguous.

Lets take the nested `if...else` statement :

```
IF C
    IF C
        S
    ELSE
        S
```

This statement results in 2 derivations trees.

Both these trees result in the same traversal, but they have different meanings. The first results in the `ELSE` belonging to the first `IF`, while the second results int he `ELSE` belonging to the second `IF`. We as humans know that the `ELSE` belongs to the second `IF`, since we know that the `ELSE` statement follows the nearest `IF`. but how can the compiler know?

There are a bunch of solutions to this problem:

1. Add a delimiter to the `IF` statement, such as `ENDIF` or `END` or `FI` to the end of the statement, resulting in these productions :

   > if-stmt --> IF condition stmt **ENDIF**
   >
   > if-stmt --> IF condition stmt ELSE stmt **ENDIF**

   Resulting in this statement :

   ```
   IF C
   |   IF C
   |   |   S
   |   |ELSE
   |   |   S
   |   ENDIF
   ENDIF
   ```

   The grammar is now unambigious, since we have to clearly state when an `IF` statement ends. However, this is not a pretty solution, and is extra work for both the programmar and compiler, and results in less readable code.

2. In C and Pascal, the compiler **always** prefers to shift the `ELSE` when it sees it in the source code so it follows the nearest `IF`. We will learn about this in more detail later.

Another thing about this grammar is **left factoring**.

Left Factoring

Consider the productions :

> A --> αβ
>
> A --> αγ

Note how the first part of the productions is the same. This grammar can be transformed by introducing a new non-terminal, So what happens now is:

> A --> αB
>
> B --> βγ

For our grammar, this results in

> if-stmt --> IF condition stmt

> if-stmt --> IF condition stmt ELSE stmt

becoming

> if-stmt --> IF conditon stmt else-part
>
> else-part --> ELSE stmt | λ

Does this solve the ambiguity? No, but it helps in removing choices, since the if-stmt is now one production. If we look at the statement :

```
IF C
    IF C
        S
    ELSE
        S
```

It still has 2 derivation trees

More Ways of Expressing Programming Languages

Extended BNF Notation

So far, we have been using **BNF Notation**(Production rules) to express languages. However, there is another form to Express a language, which is **Extended BNF Notation**

if there is repetition in the grammar, say in the example of the grammar

> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

which can give us a derivation in the form of

> E --> **E** + T --> **E** + T + T --> **E** + T + T + T --> T + T + T + T....+T

or in the same line,

> T --> T *F* --> *T* F *F* --> *T* F *F* F --> T *F* F *F*.... F

We can express this grammar as :

> E --> T { + T }
>
> T --> F { * F }
>
> F --> (E) | a

We know that [x] means that we take x 0 or 1 time only.

However, { x } means we take x zero or any number of times. This is equivalent to (x)*

We can also express this grammar as:

> E --> T (+ T)*
>
> T --> F (* F)*
>
> F --> (E) | a

Syntax Diagrams

Another way to express languages are **Syntax Diagrams**. These are used only with Extended-BNF notation.

in these diagrams, A square shape represents a nonterminal and an oval shape represents a terminal. Lets take a look at the examples below :

Parsing Techniques (Continued)

Recall : The parser is an algorithm which accepts or rejects a sentence in the programming language.

Recall : There are 2 kinds of parsers :

1. Top-Down Parsers : In This parsing technique, The parser starts with `S` using leftmost derivation to derive the sentence. The Major problem with this parsing technique is that the parser doesn't know which production it should select in each derivation step.
2. Bottom-Up Parsers : The parser in this parsing technique starts from the sentence, doing reduction steps, until it reaches the starting symbol `S` of the grammar. The Major problem with this technique is that the parser doesn't know which substring the parser should select in each reduction step.

In Top-Down parsing, we have 2 available algorithms for parsing :

1. Recursive Descent Parsing.
2. LL(1) Predictive Parsing.

In Bottom-Up parsing, we have 2 available algorithms for parsing :

1. LR Parsers.
2. Operator Precedence Parsers --> Uses matrix manipulation.

Before we continue, we need to define a few functions

The FIRST() Function

Given a string $\alpha \in V^*$, then

> $FIRST(\alpha) = \{ a \mid \alpha \text{ --*--> } aw, a \in V_T, w \in V^* \}$

in addition, if α --> λ, then we add λ to FIRST(α), that is

> $\lambda \in FIRST(\alpha)$

That is to say, FIRST(α) = Set of all terminals that may begin strings derived from α.

For example

> α --*--> cBx
>
> α --*--> ayD
>
> α --*--> ab
>
> α -----> ddd

Then

> FIRST(α) = {c,a,d}

Assume as well that

> α --*--> λ

then

> FIRST(α) = {c,a,d,λ}

That is to say, **λ appears in the FIRST() function**.

The FOLLOW() Function

We define the FOLLOW() function for **only** nonterminals. That is to say

> FOLLOW(A), $A \in V_N$

Given

> S --*--> uAβ , $u \in V_T$, $A \in V_N$, $\beta \in V^*$

then

> FOLLOW(A)=FIRST(β)

That is to say, FOLLOW(A) = The set of all terminals that may appear after A in the derivation.

> S --*--> aaXdd
>
> S --*--> Xa
>
> S --*--> BXc

Then

> FOLLOW(X) = {d,a,c}

Rules To Compute FIRST() and FOLLOW() Sets

1. FIRST(λ) = {λ}.
2. FIRST(a) = { a }.
3. FIRST(aα)= {a}.
4. FIRST(XY) = FIRST(FIRST(X).FIRST(Y)) **OR** FIRST(X.FIRST(Y)) **OR** FIRST(FIRST(X).Y).
5. Given the production A --> αXβ, Then :
   a. FIRST(β) ⊂ FOLLOW(X) if β ≠ λ.
   b. FOLLOW(A) ⊂ FOLLOW(X) if β = λ.

Note that the FIRST() and FOLLOW() sets are made of **terminals only**

By these rules, say we have

> A -- > αXβ , $X \in V_N$

and say we want FOLLOW(X)

Then

> FIRST(β) ⊂ FOLLOW(X)

We say it is a subset because we can have other productions involving X.

Assuming that β = λ, Things are different.

Say that we have a production that leads to this derivation is

> S --*--> uAγ

and following through this results in this derivation :

> S --*--> uAγ --> uαXγ

Therefore,

> FOLLOW(A) ⊂ FOLLOW(X)

This is because whatever follows A can follow X if there is nothing between them.

Notes :

1. λ **may** appear in FIRST() but it doesn't appear in FOLLOW(). We will see this when we define augmented grammars.
2. Generally, we start computing the FIRST() from bottom to top, But FOLLOW() from top to bottom.
3. When we compute FOLLOW(X), we search for X in the right side of any production.

Augmented Grammars

Given the grammar G=(V_N,V_T,S,P), then the augmented grammar G`=($V_N$`,V_T`,S`,P`) can be obtained from G as follows:

1. V_N` = $V_N$ ∪ {S`}.
2. V_T` = V_T ∪ { $ }.
3. S` = new starting point.
4. P` = P ∪ {S`-->S$}

For example :

> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

Becomes :

> G --> E$
>
> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

This is because we want to create a FOLLOW() set for S.

Lets take another example :

> S` --> S$
>
> S --> AB
>
> A --> a | λ
>
> B --> b | λ

Lets compute the FIRST() sets for this grammar :

> FIRST(A) = {a,λ}
>
> FIRST(B) = {b,λ}
>
> FIRST(S) = FIRST(AB) = FIRST(FIRST(A).FIRST(B))
>
> = FIRST({a,λ}.{b,λ})
>
> = FIRST({a,λ,b,λ})
>
> = {a,b,λ}
>
> FIRST(S`) = FIRST(S$) = FIRST(FIRST(S).FIRST($))
>
> = FIRST({a,b,λ}.$)= FIRST(a$,b$,$)
>
> = {a,b,$}

Now Lets compute the FOLLOW() sets for this grammar :

> FOLLOW(S) = {$}
>
> FOLLOW(A) = {b,$}
>
> FOLLOW(B) = {$}

Lets Take another, slightly more complex example :

> S` --> S$
>
> S --> aAcb
>
> S --> Abc
>
> A --> b | c | λ

Lets take the FIRST() for this grammar :

> FIRST(A) = {b,c,λ}
>
> FIRST(S) = FIRST(aAcb)∪FIRST(Abc) = {a} ∪ {b,c}
>
> = {a,b,c}
>
> FIRST(S`) = FIRST(S$) = FIRST(FIRST(S).FIRST($))
>
> = FIRST({a,b,c}.{$})
>
> = {a,b,c}

Now lets take the FOLLOW() :

> FOLLOW(S) = {$}
>
> FOLLOW(A = {c,b}

Say we have the grammar

> G --> E$
>
> E --> E + T | T
>
> T --> T * F | F
>
> F --> (E) | a

Lets calculate FIRST() :

> FIRST(F) = {(,a}
>
> FIRST(T) = FIRST(T *F)∪FIRST(F) = FIRST(T* F)∪{(,a}
>
> = {(,a} (Because every T will eventually become an F)
>
> FIRST(E) = FIRST(E + T) ∪ FIRST(T) = {(,a} ∪ {(,a}
>
> = {(,a}
>
> FIRST(G) = FIRST(E$) = {(,a}

Now lets Calculate FOLLOW() :

> FOLLOW(E) = {$,+,)}
>
> FOLLOW(T) = FOLLOW(E) ∪ {*} = {$,+,*,)}

> FOLLOW(F) = FOLLOW(T) = {$,+,*,)}

But what makes all this so important?

Well, All of the parsing techniques we are going to learn will heavily rely on FIRST() and FOLLOW().

Top-Down Parsing (Continued)

Recursive Descent Parsing

Recursive Descent Parsing is very simple. It works like this :

1. Divide the grammar into primitive/simple components

    i. For the token "a" :

    ```
    ```

    ```
    If(token == "a"){

     get-next();

    }

    else{

     report-error();

    }
    ```

    ii. For X --> $A_1A_2...A_n$: `

    ```
        code(X){

        code(A1);

        code(A2);
        ...

        ...

        ...

        code(An);
        }

        ```
        That is, if a production is made of multiple sub productions in order,
        they must be called in order.

        If FIRST(A_n) of the production is non-empty, and
        A_n ≠ λ, ie, the production is not
        an empty one, then


        ```
        If(token ∈ FIRST(A1)){
            code(A1);
        }
         else If(token ∈ FIRST(A2)){
            code(A2);
        }
        ...

        ...
    ```

```
...
else If(token ∈ FIRST(An)){
    code(An);
}
else{
 report-error();
}
```

Furthermore, the code for x = A*, where we can have
zero or more consecutive repetitions of a production, we say

```
while(token ∈ FIRST(A*)){

 call(A*);

 }
```

Notes :

1. Every nonterminal has a code(a function).
2. S` in augmented grammar is represented by the function "main".
3. We only start with calling "get-token" in function "main".

For Example, lets say we have

> G --> E$
>
> E --> T(+ T)*
>
> T --> F(* F)*
>
> F --> (E) | a

Then we can say :

```
 main(){//represents G

 get-token;

 call E();

 if(token!="$")
 {
    Error;
 }
 else{
    SUCCESS;
 }



}

 function E(){//Represents E -- T (+ T)*

    call T();

    while(token == "+"){

        get-token();

        call T()
    }
```

```
}

function T(){//T--> F (* F)*

   call F();
   while(token == "*"){

       get-token();

       call F();
   }
}

function F(){

   if(token == "(")
   {
       get-token();

       call E();

       if(token == ")")
       {
           get-token();
       }
       else
       {
           ERROR;
       }
   }
   else if(token=="a")
       {
       get-token();
       }
   else
   {
       ERROR;
   }
}
```

Note that `ERROR` is a function we should write.

Lets take another example now.

Given the grammar :

> Program --> body .
>
> body --> Begin stmt (; stmt)* End
>
> stmt --> Read | Write | body | λ

where we will represent λ as `l` from now on in the example;

and

> V_N = { Program, body, stmt, block}
>
> V_T = { ., Begin, ;, End, Read, Write}

examples of programs of this language would be

```
Begin
    Read;
    Write;
    Read;
    Write;
End.
```

or

```
Begin
    Read;
End.
```

or

```
Begin
    Read;
    Begin
        Read;
        Write;
    End.
    Write;
End.
```

or

```
Begin;
    ;
    ;
    ;
    ;
End.
```

Lets write the code for this programming language.

```
main(){

    get-token();

    call body();

    if(token != "."){
        ERROR;
    }
    else{
        SUCCESS;
    }
}

function body(){

    call Begin();
    if(token == "Begin"){
        get-token();
        call stmt();
        while(token !=";"){
            get-token();
            call stmt();
        }
        if(token == "End"){
            get-token();
        }
        else{
            ERROR;
        }

    }
    else{
        ERROR;
    }
}

function stmt(){

    if(token == "Read"){
        get-token();
```

```
    }
    else if(token == "Write"){
        get-token();
    }
    else if(token == "Begin"){
        call body();
    }
    else if(token != ";" || token != "End" ){
        ERROR();
    }
}
```

LL(1) Parsing

This Parsing method is a **table-driven** parsing method. The LL(1) parsing table selects which production to choose for the next derivation step.

Formal Definition of LL(1)

The Formal definition of LL(1) grammars is given by :

> Given the Productions :
>
> A --> α_1
>
> A --> α_2
>
> A --> α_3
>
> A --> α_n

or alternatively

> A --> α_1 | α_2 | α_3 | ... | α_n then the grammar is LL(1) if :

1. FIRST(α_i)∩FIRST(α_j) = ∅ for all i,j
2. if one of α_i is λ, then FIRST(α_j) ∩ FOLLOW(A) = ∅ for all j.

For example, Given the grammar :

> S --> S$
>
> S --> aABC
>
> A --> a | bbD
>
> B --> a | λ
>
> C --> b | λ
>
> D --> C | λ

let us see if it is LL(1)

> FIRST(a)∩FIRST(bbD) = ∅
>
> FIRST(a)∩FOLLOW(B) = ∅
>
> FIRST(b)∩FOLLOW(C) = ∅
>
> FIRST(c)∩FOLLOW(D) = ∅

Then this grammar is LL(1).

Given another Grammar :

> S` --> S$

> S --> aAa | λ
>
> A --> abS | λ

> FIRST(aAa)∩FOLLOW(S) = {a}∩{$,a} = {a} ≠ ∅

This grammar is **not** LL(1).

Lets assume that we have a grammar that is LL(1). How do we build the LL(1) parsing table?

LL(1) Parsing Table Building Algorithm

1. For each production A --> α in the grammar G,
    i. Add to the table entry T[A,a] the production A --> α, where A ∈ FIRST(α)
2. If λ ∈ FIRST(α), Add to the table entry T[A,b] the production A --> α ∀ b ∈ FOLLOW(A).
3. All Remaining Entries are Error Entries.

For example, given the grammar :

> V --> SR $[1]
>
> S --> +[2] | -[3] | λ[4]
>
> R --> dN.N[5] | .dN[6]
>
> N --> dN[7]| λ[8]

note that the superscript denotes the production number.

Then the table will look like this

> FIRST(SR $) = {+,-,d,.}
>
> FIRST(+) = {+}
>
> FOLLOW(S) = {d,.}
>
> FIRST(R) = {d, .}
>
> FIRST(d) = { d }
>
> FOLLOW(N) = {d, ., $}

$V_N \backslash V_T$	+	-	d	.	$
V	1	1	1	1	
S	2	3	4	4	
R			5	5	
N			7	8	8

There should be no conflict(multiple entries) in the LL(1) table.

L(G) of this grammar = all floating point numbers.

The parser works like this

Stack	Remaining Input	Action
V	-dd.d$	Production 1
SR$	-dd.d$	Production 3
-R$	-dd.d$	Pop & advance input
R$	dd.d$	Production 5
dN.N$	dd.d$	Pop & advance input

Stack	Remaining Input	Action
N.N$	d.d$	Production 7
dN.N$	d.d$	Pop & advance input
N.N$	.d$	Production 8
.N$	.d$	Pop & advance input
N$	d$	Production 7
dN$	d$	Pop & advance
N$	$	Production 8
$	$	Pop and Advance
λ	λ	Accept

If at any point the parser reaches a place where the input and the stack have 2 different terminal symbols, it throws a syntax error.

Lets Take another example. Let the Grammar be :

> program --> block $ [1]
>
> block --> { declarations stmnts } [2]
>
> decls --> D ; decls [3] | λ [4]
>
> stmnts --> statement ; stmts [5] | λ [6]
>
> statement --> if [7] | while [8] | ass [9] | scan [10] | print[11] | block [12] | λ[13]

V_T = {$,{,},D,;,if,while,ass,scan,print}

$V_N \backslash V_T$	if	while	ass	scan	print	{	}	D	;	$
Program						1				
block						2				
decls	4	4	4	4	4	4	4	3	4	
stmts	5	5	5	5	5	5	6		5	
statement	7	8	9	10	11	2			13	

No conflict.

Another example is the If....else statement with a delimiter. the grammar looks like this :

> S` --> S$
>
> S --> iCSE
>
> E --> S | λ
>
> S --> a
>
> C --> c

$V_N \backslash V_T$	i	a	e	c	$
S `	1	1			
S	2	5			
E			3,4		4
C				6	

There is a conflict. To solve this, we can add a delimiter.

> S` --> S$
>
> S --> iCSEd
>
> E --> eS | λ
>
> S --> a
>
> C --> c

$V_N \backslash V_T$	i	a	e	c	d	$
S `	1	1				
S	2	5				
E			3		4	
C				6		

The grammar is now unambiguous. Alternatively, we can just strike out the transition 4 from the LL(1) table, which is a λ transition. Lets follow through the derivation tree. the resultant table is

$V_N \backslash V_T$	i	a	e	c	$
S `	1	1			
S	2	5			
E			3		4
C				6	

and this works because the natural behaviour of the else-part is to follow the nearest if statement.

Something Important to note is that **if a grammar is LL(1), then it is unambiguous. However, the opposite is not necessarily true.**

Another thing to note is that in Top-Down parsing, we should avoid a grammar that is not LL(1).

Bottom-Up Parsing (Continued)

Recall that in Bottom-Up parsing, the parser starts from the given sentence, applying reductions until it reaches the starting symbol of the grammar or a deadlock. The major problem with Bottom-Up parsing is which substring we should select in each reduction step.

The answer to the above question is : In each reduction step, we select what is called **the handle**.

The Handle is obtained by a **rightmost** derivation **in reverse**

For example, Given the grammar :

> V --> S R $
>
> S --> +|-|λ
>
> R --> .dN | dN.N
>
> N --> dN | λ

and the sentence

> -dd.d$

First, we derive the sentence rightmost.

> V $\xrightarrow{rm}$ SR$ $\xrightarrow{rm}$ SdN.N$ $\xrightarrow{rm}$ SdN.dN$ $\xrightarrow{rm}$ SdN.dd$ $\xrightarrow{rm}$ SddN.d$ $\xrightarrow{rm}$ Sdd.d$ $\xrightarrow{rm}$ -dd.d$

So our handles would be :

> V <-- **SR$** <-- S**dN.N**$ <-- SdN.**dN**$ <-- SdN.d**λ**$ <-- Sd**dN**.d$ <-- Sdd**λ**.d$ <-- **-**dd.d$

But Compilers don't work like this. We already derived the sentence, why would we go back and do it again?

We cannot build a Bottom-Up parser for every Context-Free Grammar. However, we are fortunate enough that there exist subsets of the Context-Free Grammar for which we can build a deterministic Bottom-Up parser ie, the parser can determine/decide precisely where the handle is in each reduction step.

some of these subsets are

- LR Parsers :
  - SLR(Simple-LR).
  - LALR(Look-Ahead LR).
  - LR.
- Operator Precedence.

We will only be talking about the SLR parser, just to get an idea of how Bottom-Up parsing works.

SLR Parsing

SLR parsing, and LR parsing in general, is a tabular parsing method.

All LL(1) grammars are a subset of SLR grammars.

All LR parses contains :

1. A parsing table.
2. A stack.
3. The input string.

As a reminder, the LL(1) parser contains :

1. A parsing table.
2. A stack.
3. The input string.

However, the way we build it is different.

Building the SLR Parsing Table

> An LR(0) item of a grammar G is a production in G with a dot(.) at some position in the right side.

For example, the production

> A --> aBY

This production generates the following LR(0) items :

> A --> .aBY
>
> A --> a.BY
>
> A --> aB.Y
>
> A --> aBY. --> complete item

Note that for A --> λ, this generates only A --> λ. .

Generally speaking, if the right side of the production is L, then there are L+1 resultant LR(0) items.

The LR(0) item

> A --> aB.Y

Means that the parser has scanned on the input a string derived from aB and expects to see a string derived from Y .

We need to define the following 2 functions.

The CLOSURE function

```
function CLOSURE(I)//I is a set of LR(0)items
{
    Repeat
        for(every LR(0) item A-->α.Bβ in I,
        and for every production B-->γ in G,
        Add the LR(0)item B-->.γ to I)
    Until no more items to be added;
}
```

Lets apply this to our grammar :

> (1)E --> E+T
>
> (2)E --> T
>
> (3)T --> T*F
>
> (4)T --> F
>
> (5)F --> (E)
>
> (6)F --> a

This grammar is not LL(1) because

> FIRST(T*F)∩FIRST(F) = {(,a} ≠ ∅

We will need to build the LR(0) sets of items.

Lets pre-compute all the LR(0) items for this grammar:

> E` --> .E
>
> E` --> E. -->complete item
>
> E --> .E+T
>
> E --> E.+T
>
> E --> E+.T
>
> E --> E+T. --> complete item
>
> E --> .T
>
> E --> T. --> complete item
>
> T --> .T*F
>
> T --> T.*F
>
> T --> T*.F
>
> T --> T*F. --> complete item
>
> T --> .F
>
> T --> F. --> complete item
>
> F --> .(E)
>
> F --> (.E)
>
> F --> (E.)

> F --> (E). --> complete item
>
> F --> .a
>
> F --> a. --> complete item

we start with I_0

> I_0: E` --> .E
>
> CLOSURE(I_0)
>
> E` --> λ.Eλ
>
> A ---> γ.Bβ

we add all productions starting with E and add the . at the start therefore :

> I_0:{E`-->.E, E-->.E+T, E-->.T}

Lets look at E-->.E+T

> E-->λ.E+T
>
> A ---> γ.Bβ

Therefore, we add all productions starting with T to I_0

> I_0:{E`-->.E, E-->.E+T, E-->.T, T--> .T*F, T-->.F}

we iterate again, and the resultant set is:

> I_0:{E`-->.E, E-->.E+T, E-->.T, T--> .T*F, T-->.F, F-->.(E), F-->.a}

The GOTO function

```
function GOTO(I,X)//I=set of items,X=Grammar symbol
{

   CLOSURE(all items A-->αX.β Where A-->α.Xβ in I)

}
```

Lets apply this to the grammar above. First we separate each grammar symbol production to its own set This results in 4 Item groups:

> I_1 : E`-->.E, E-->.E+T
>
> I_2 : E-->.T, T--> .T*F
>
> I_3 : T-->.F
>
> I_4 : F-->.(E)
>
> I_5 :F-->.a

and take the CLOSURE for all these sets. The resultant is :

> I_0: E`-->.E, E-->.E+T, E-->.T, T--> .T*F, T-->.F, F-->.(E), F-->.a <--(CLOSURE(E`-->.E))
>
> I_1 : E`-->E., E-->E.+T <-- GOTO(I_0,E)
>
> I_2 : E-->T., T--> T.*F <-- GOTO(I_0,T)
>
> I_3 : T-->F. <-- GOTO(I_0,F)
>
> I_4 : F-->(.E), E --> .E+T,E-->.T,E-->.T*F,T-->.F,F-->.(E),F-->.a <-- GOTO(I_0,()

> I_5 : F-->a. <-- GOTO(I_0,a)
>
> I_6 : E-->E+.T,E-->.T*F,T-->.F,F-->.(E),F-->.a <-- GOTO(I_1,+)
>
> I_7 : E-->T.*F*,*F-->.(E),F-->.a* <-- *GOTO(I_2,*)
>
> I_8 : F-->(E.),E-->E.+T <-- GOTO(I_4,E) // GOTO with T and F wouldn't result in a new item set.
>
> I_9 : E --> E+T. , T --> T.*F <-- GOTO(I_6,T)
>
> I_{10} : T --> T*F. <-- GOTO(I_7,F)
>
> I_{11} : F --> (E). <-- GOTO(I_1,))

Constructing the SLR table

Input : LR(0) sets of items

Output : SLR(1) parsing table

1. For every item A-->α.aβ in I_i, a ∈ V_t, and GOTO(I_i,a)=I_j, then ACTION[i,a] = S_j(shift and push j on the stack).
2. For item A-->α.(complete item) in I_i, ACTION[i,b] = Reduce by a-->A ∀ b ∈ FOLLOW(A).
3. For S` --> S. in I_i, ACTION[i,$] = Accept.
4. For all A ∈ V_N, if GOTO(I_i,A) = I_j then set GOTO[i,A] = j.
5. All remaining entries are error entries.

lets apply this to the example above and generate the table

V	Action	a	+	*	(	)	$	GOTO	E	T	F
0		S5			S4				1	2	3
1			S6				A				
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				

No conflict --> SLR(1) grammar

Parsing The SLR Table

> E --> E+T
>
> E --> T
>
> T --> T*F
>
> T --> F
>
> F --> (E)
>
> F --> a

Lets examine the sentence

> a + a $

Stack	Remaining	Action
0	a + a $	S5
0 a 5	+ a $	R6
0 F 3	+ a $	R4
0 T 2	+ a $	R2
0 E 1	+ a $	S6
0 E 1 + 6	a $	S5
0 E 1 + 6 a 5	$	R6
0 E 1 + 6 F 3	$	R4
0 E 1 + 6 T 9	$	R1
0 E 1	$	Accept

But what if the grammar is not SLR?

LR Parsing Techniques

The main difference between LR and SLR is the CLOSURE function

```
function CLOSURE(I)//I is a set of LR(1)items
{
    Repeat
        for(every LR(1) item [A-->α.Bβ, a] in I,
        and for every production B-->γ in G,
        Add the LR(1)item [B-->.γ , b] where b belongs to FIRST(β a)to I)
    Until no more items to be added;
}
```

Where An LR(1) item is an LR(0) item with a **Lookahead Symbol**. For example

[A --> α.β , a] where a is the lookahead. The lookahead symbol "a" has no effect whatsoever on an item [A --> α.β,a], β ≠ λ (not complete item). However, if the item is complete, such that [A --> α.,a], this means we reduce by the production A --> α on token "a".

For example, Give the grammar :

> S --> S`
>
> S --> CC (1)
>
> C --> cC (2)
>
> C --> d (3)

I_0:

- S` --> .S, $ I_1
- S --> .CC, FIRST(λ$) = FIRST($) = $ I_2
- C --> .cC, FIRST(C$) = c,d I_3
- C --> .d, c,d I_4

I_1:

- S` --> .S, $ Accept

I_2:

- S --> C.C , \$ I_5 (if the lookahead is different it goes to a new state)
- C --> .cC , \$ I_6
- C --> .d , \$ I_7

I_3:

- C --> c.C , c,d I_8
- C --> .cC c,d I_3
- C --> .d I_4

I_4:

- C --> d., c,d Complete

I_5 :

- S --> CC. , \$ Complete

I_6 :

- C --> c.C , \$ I_9
- C --> .cC , \$ I_6
- C --> .d , \$ I_7

I_7 :

- C --> d. , \$ Complete

I_8 :

- C --> cC. , c,d Complete

I_9 :

- C --> cC. , \$ Complete

V	Action	c	d	\$	GOTO	S	C
0		S3	S4			1	2
1				A			
2		S6	S7				5
3		S3	S4				8
4		R3	R3				
5				R1			
6		S6	S7				9
7				R3			
8		R2	R2				
9				R2			

No conflict, the grammar is an LR grammar.

If we look at the above example, we can see that some sets of items have the same core items(LR(0) items), but the lookahead is different. For example (I_7 , I_4), (I_3 , I_6), (I_8 , I_9). Lets say we merge the states. What happens is the we add the lookaheads and change the state references for equivalent states . The table becomes this:

V	Action	c	d	\$	GOTO	S	C
0		S3	S4			1	2

V	Action	c	d	$	GOTO	S	C
1				A			
2		S3	S4				5
3		S3	S4				8
4		R3	R3	R3			
5				R1			
8		R2	R2	R2			

This is now a simplified table. if the parsing table after merging has no conflicts(like in the above example), then the grammar is an LALR(1) Grammar.