Compilers

Third course
Spring term

Alfonso Ortega: alfonso.ortega@uam.es
Enrique Alfonseca: enrique.alfonseca@uam.es

Top-down analysis

Overview

- **Top-down analysis**
  - Blind search, depth-first search and width-first search.
  - Slow backtracking
- **Top-down analysis with LL(1) grammars**
  - Procedures for modifying grammars
    - Elimination of left-recursion
    - Elimination of lambda-rules
  - Greibach Normal Form
    - Definition
    - Procedure for obtaining the GNF of a grammar
    - Examples
  - LL(1) grammars
    - Initial examples
    - Definition of LL(1) grammars
    - Syntactic analysers based on LL(1) grammars

Introduction

- We need to find a derivation from the axiom to the program analysed.
- However, depending on the grammar, the number of possible derivations is extremely large.
- There are general strategies to solve any problem with blind search:
  - Breadth-first search and depth-first search.
  - In **breadth-first search**, it is guaranteed that we shall find a solution.
  - In **depth-first search**, there might be branches with infinite lengths. If that is not the case, sometimes it is possible to find a solution with less effort than in breadth-first search.

Top-down analysis

Top-down with backtracking: concepts

- We can consider "top-down analysis with a **backtracking parser**" as a particular case of depth-first blind search.
- The criterion to decide whether a rule is applicable in a given position is the coincidence with the non-terminal that is in the left-hand side of the rule.
- We shall not continue along an analysis path when we have terminals that do not coincide with the program.
- We can finish the process in two situations:
  - If we have been able to generate all the terminals in the program sequence. In this case, we have been able to construct a syntactic tree for the program, which is correct.
  - If it was not possible to finish with the derivation tree, but we have tried all the rules in each position and there is no other option available to try. In this case, the program will be syntactically incorrect.
- This technique will be illustrated with the following examples:

Top-down analysis

Previous examples

- Syntactic analysis of the word **aaabbb**

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

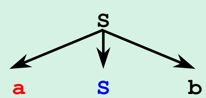
S

aaabbb

Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

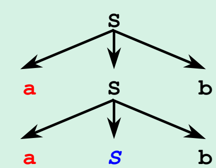


aaabbb

Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```



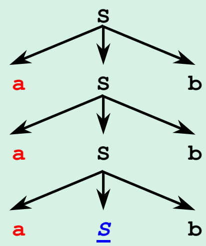
aaabbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

aaa*b*bb

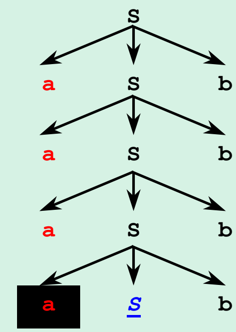


Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

aaa*b*bb

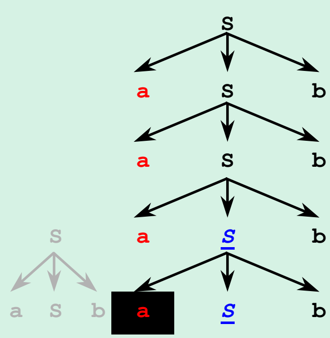


Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

aaa*b*bb

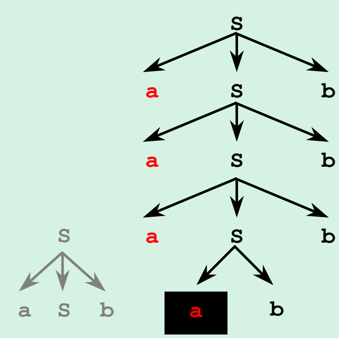


Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

aaa*b*bb

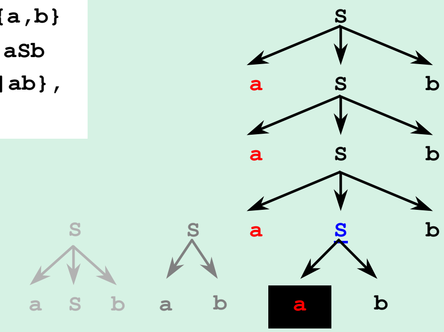


Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

aaab̲bb

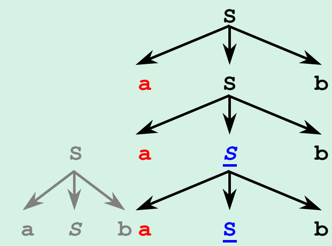


Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

aaab̲bb

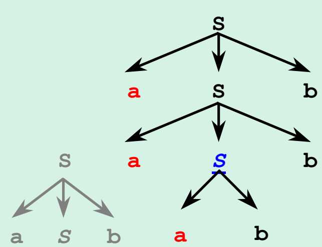


Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

aaab̲bb

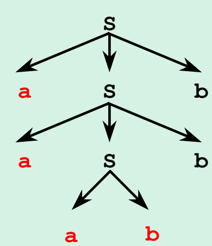


Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
      |ab},
   S>
```

aaabb̲b



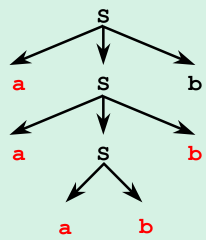
Top-down analysis

Previous examples

- La next palaba

```
G=<{S},{a,b}
  {S → aSb
       |ab},
    S>
```

aaabbb

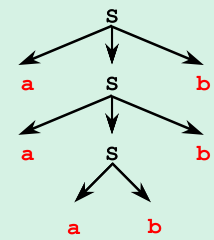


Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
       |ab},
    S>
```

aaabbb



ACCEPTED

Top-down analysis

Previous examples

- Syntactic analysis of the word **abbb**

```
G=<{S},{a,b}
  {S → aSb
       |ab},
    S>
```

S

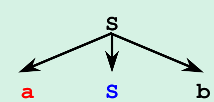
abbb

Top-down analysis

Previous examples

```
G=<{S},{a,b}
  {S → aSb
       |ab},
    S>
```

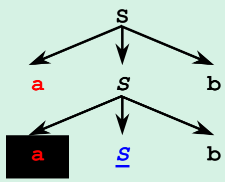
abbb



Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

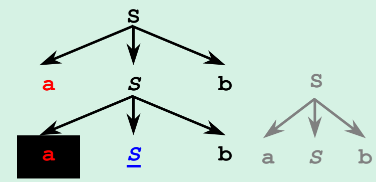


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

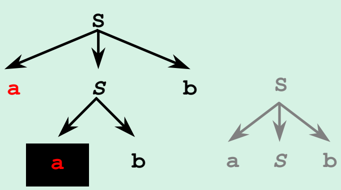


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

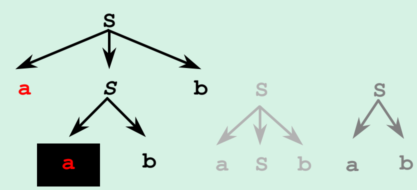


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

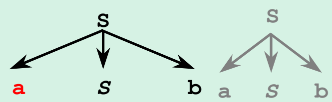


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

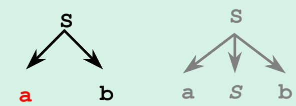


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

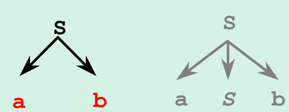


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>

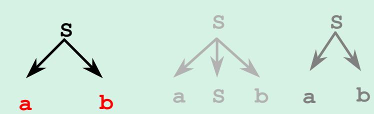


abbb

Top-down analysis

Previous examples

G=<{S},{a,b}
{S → aSb
|ab},
S>



abbb

REJECTED

Top-down analysis

Previous examples

- Analysis of the word `i+--i`

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

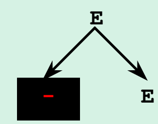
E

i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

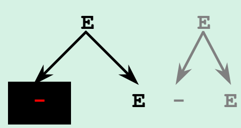


i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

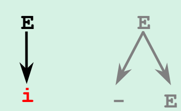


i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

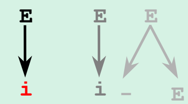


i*+*--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
    E>
```

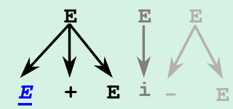


i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
    E>
```

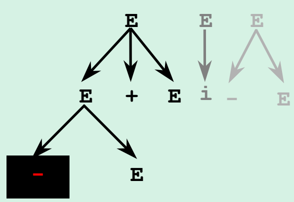


i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
    E>
```

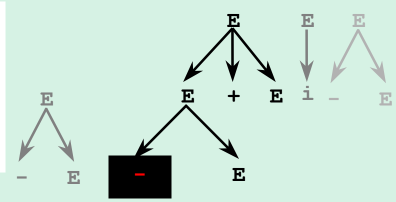


i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
    E>
```

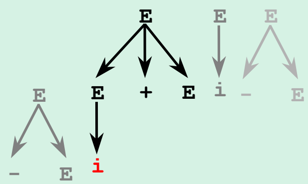


i+--i

Top-down analysis

Previous examples

G=<{E},{-,+,i}
{E→-E
|i
|E+E},
E>

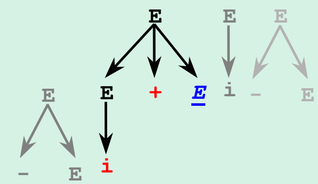


i+--i

Top-down analysis

Previous examples

G=<{E},{-,+,i}
{E→-E
|i
|E+E},
E>

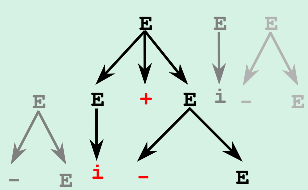


i+--i

Top-down analysis

Previous examples

G=<{E},{-,+,i}
{E→-E
|i
|E+E},
E>

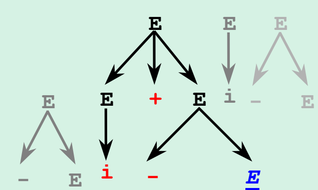


i+--i

Top-down analysis

Previous examples

G=<{E},{-,+,i}
{E→-E
|i
|E+E},
E>



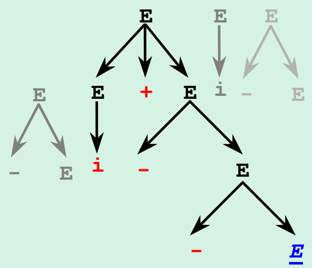
i+--i

Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

i+--i

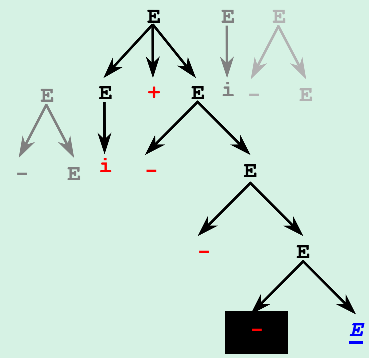


Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

i+--i

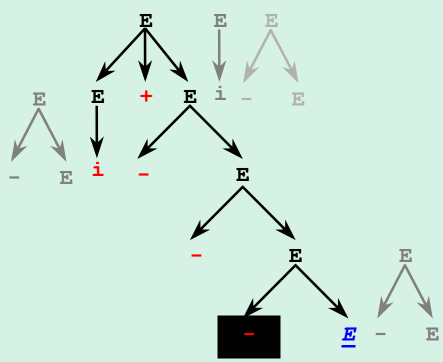


Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

i+--i

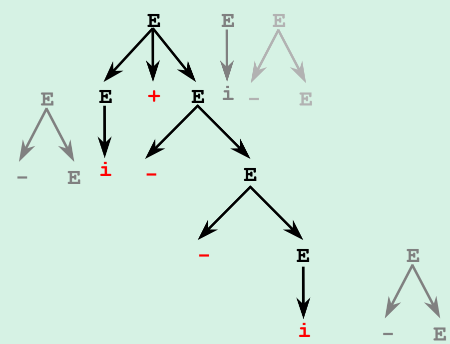


Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

i+--i

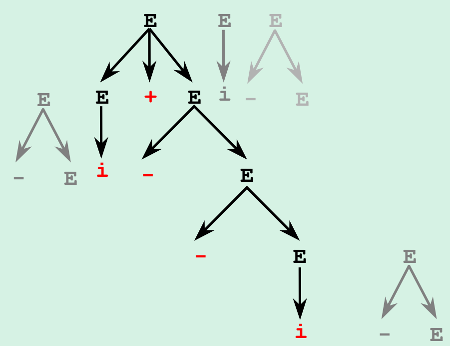


Top-down analysis

Previous examples

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

i+--i



ACCEPTED

Top-down analysis

Previous examples

- Consider the analysis of **i++i**

```
G=<{E},{-,+,i}
  {E→-E
     |i
     |E+E},
   E>
```

E

i++i

Top-down analysis

Conclusions

- If there are left-recursive rules, the analysis of the word may provoke that the algorithm enters an infinite loop.
- In the following slides, we shall study the possibilities for avoiding that fact, and we shall see other properties of grammars which are interesting for building efficient top-down syntactic analysers.

Top-down analysis

Overview

- **Top-down analysis**
  - Blind search, depth-first search and width-first search.
  - Slow backtracking
- **Top-down analysis with LL(1) grammars**
  - Procedures for modifying grammars
    - Elimination of left-recursion
    - Elimination of lambda-rules
  - Greibach Normal Form
    - Definition
    - Procedure for obtaining the GNF of a grammar
    - Examples
  - LL(1) grammars
    - Initial examples
    - Definition of LL(1) grammars
    - Syntactic analysers based on LL(1) grammars

Properties of grammars for top-down analysis

Properties

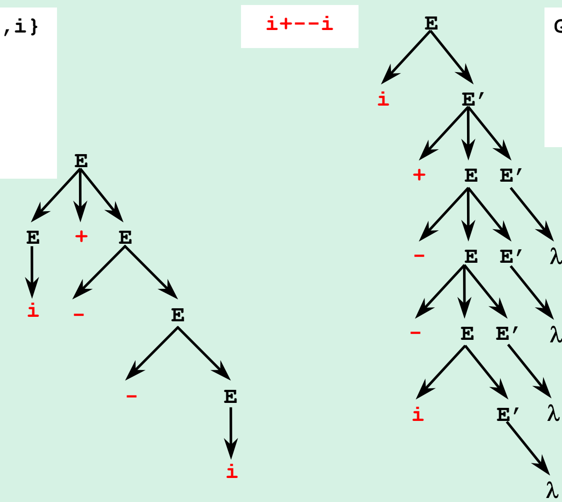
- It is easy to build LL(1) analysers, in which the input is read from left to right, and the derivations in the tree are also made from left to right. The number “1” means that it needs one look-ahead symbol from the input string.
- In order to build an LL(1) analyser, the grammar has to be expressed in LL(1) form.
- Possible steps to transform a grammar into LL(1) form are:
  - Removing all the left-recursive rules.
  - Removing inaccessible symbols.
  - Removing rules that generate the empty word: $A \to \lambda$
  - Expressing the grammar in Greibach Normal Form.
  - Left factorisation (which is a necessary condition for LL(1) grammars)

Greibach Normal Form

Removing left-recursive rules

- Let us analyse the following example:

```
G=<{E},{-,+,i}
  {E→-E
    |i
    |E+E},
  E>
```




```
G=<{E},{-,+,i}
  {E→-EE'|iE'
   E'→+EE'|λ},
  E>
```


Greibach Normal Form

Removing left-recursive rules

- It is difficult to think of a general procedure from the previous example.
- However, the technique can be generalised.

Greibach Normal Form

Removing left-recursive rules: first example

```
G=<{E,T,F},
   {-,*,i}
  {E→E+T|T
   T→T*F|F
   F→i},
   E>
```

$A \to \underline{A\alpha_1} | \dots | \underline{A\alpha_n} | \beta_1 | \dots | \beta_m$

$A \to \beta_1 X | \dots | \beta_m X$

$X \to \underline{\alpha_1} X | \dots | \underline{\alpha_n} X | \lambda$

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→TE'
   E'→+TE'|λ
   T→FT'
   T'→*FT'|λ
   F→i},
   E>
```

Greibach Normal Form

Lemma: removing left-recursive rules

Every context-independent grammar can be transformed into an equivalent grammar without left-recursive rules.

- A left-recursive rule is a rule of the following form:

$$A \to Ax, \; A \in \Sigma_N \wedge x \in (\Sigma_T \cup \Sigma_N)^*$$

- The lemma is shown in a constructive way doing, for each recursive rule, the following treatment:
- Let $< \Sigma_T, \Sigma_N, S, P >$ be a grammar with left-recursive rules:
  - $A \to A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$
  - Where $\{\beta_i\}^n_{i=1}$ represent all the non-left-recursive rules
- We can substitute the previous rules by the following set of rules
  - $A \to \beta_1 X | \ldots | \beta_m X$
  - $X \to \alpha_1 X | \ldots | \alpha_n X | \lambda$

Greibach Normal Form

Removing lambda-rules

We can modify a rule by substituting a non-terminal in its right-hand side by all the right-hand sides of the rules for that non-terminal.

The grammar obtained in this way generates the same language as the original one.

Greibach Normal Form

Removing lambda-rules

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→TE'
   E'→+TE'|λ
   T→FT'
   T'→*FT'|λ
   F→i},
    E>
```

$F \to \underline{i}$ ($\underline{i}$ can be put in place of $\underline{F}$ in) $T \to \underline{F}T'$

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→TE'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```


Greibach Normal Form

Removing lambda-rules

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→TE'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

$T \to \underline{iT'}$ ($\underline{iT'}$ is set in the place of $\underline{T}$) $E \to \underline{T}E'$

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Greibach Normal Form

Removing lambda-rules

We can delete a rule for a non-terminal symbol, provided that we also add new rules by substituting all appearances of the non-terminal by all the right-hand sides of the rules eliminated.

- A case with special interest is the rules that generate the empty word, λ
- A lambda-rule is a rule of the following form:

$$A \to \lambda, , A \in \Sigma_N$$

- It is used to remove a non-terminal symbol from a word.

Greibach Normal Form

Removing lambda-rules

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'
   E'→+TE'|+T
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```


Greibach Normal Form

Removing lambda-rules

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'
   E'→+TE'|+T
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'|iE'|i
   E'→+TE'|+T
   T→iT'|i
   T'→*FT'|*F
   F→i},
    E>
```

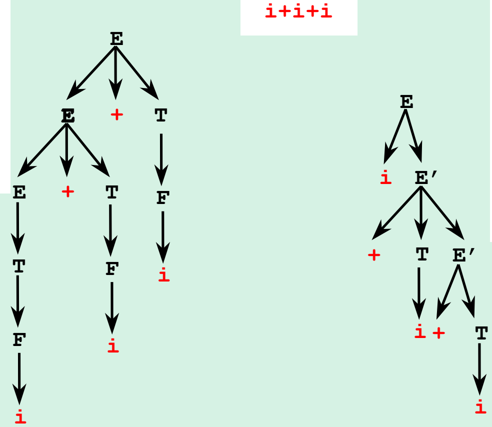

Greibach Normal Form

Removing lambda-rules

- Let us test some derivations:

```
G=<{E,T,F},
   {-,*,i}
  {E→E+T|T
   T→T*F|F
   F→i},
    E>
```

i+i+i



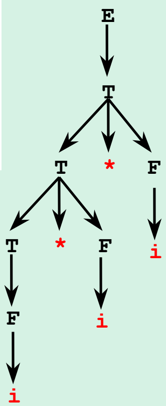
```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'|iE'|i
   E'→+TE'|+T
   T→iT'|i
   T'→*FT'|*F
   F→i},
    E>
```

Greibach Normal Form

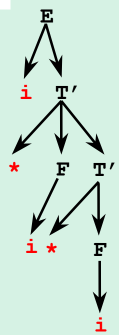
Removing lambda-rules

- Let us test some derivations:

G=<{E,T,F},
{-,*,i}
{E→$\underline{E+T}$|T
T→$\underline{T*F}$|F
F→i},
E>



i*i*i



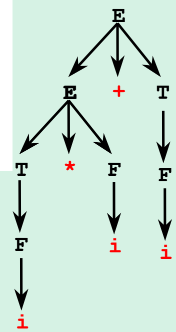
G′=<{E,E′,T,T′,F},
{-,*,i}
{E→iT′E′|iT′|iE′|i
E′→+TE′|+T
T→iT′|i
T′→*FT′|*F
F→i},
E>

Greibach Normal Form

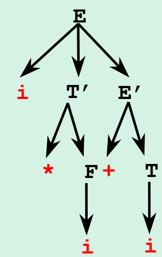
Removing lambda-rules

- Let us test some derivations:

G=<{E,T,F},
{-,*,i}
{E→$\underline{E+T}$|T
T→$\underline{T*F}$|F
F→i},
E>



i*i+i



G′=<{E,E′,T,T′,F},
{-,*,i}
{E→iT′E′|iT′|iE′|i
E′→+TE′|+T
T→iT′|i
T′→*FT′|*F
F→i},
E>

Top-down analysis

Overview

- **Top-down analysis**
  - Blind search, depth-first search and width-first search.
  - Slow backtracking
- **Top-down analysis with LL(1) grammars**
  - Procedures for modifying grammars
    - Elimination of left-recursion
    - Elimination of lambda-rules
  - Greibach Normal Form
    - Definition
    - Procedure for obtaining the GNF of a grammar
    - Examples
  - LL(1) grammars
    - Initial examples
    - Definition of LL(1) grammars
    - Syntactic analysers based on LL(1) grammars

Greibach Normal Form

Definition

- Informally,

A context-independent grammar is in Greibach Normal Form if and only if the right-hand-side of all the rules starts with a terminal symbol, followed, optionally, by non-terminals.

- Formally,

$< \Sigma_T, \Sigma_N, S, P >$, context-independent, is in Greibach Normal Form is ⇔(def)
$\forall\ r\in P\ \ r=A\to ax\ ,,\ a\in\Sigma_T \wedge x\in\Sigma^*_N$

Greibach Normal Form

Theorem

Every context-independent grammar which does not generate the empty word can be expressed in Greibach Normal form.

Greibach Normal Form

Proof of the theorem

- It will be a constructive proof, showing how the grammar can be transformed.
- The transformation is performed in the following way:
  1. If the language contains the empty word, add the following rule (S is the axiom)
     $$S \to \lambda$$
  2. Remove all the left-recursive rules applying the lemma seen before.
  3. The following partial ordering will be established between the non-terminal symbols, deduced from the production rules:
     $$A_i < A_j \Leftrightarrow (\exists \alpha \in \Sigma_T^* \mid A_i \to A_j \alpha \in P) \wedge (\neg \exists \beta \in \Sigma_T^* \mid A_j \to A_i \beta \in P)$$
     If this ordering produces a loop, i.e., if
     $$\exists \alpha, \underline{\beta} \in \Sigma_T^* \mid A_i \to A_j \alpha \in P \wedge A_j \to A_i \beta \in P$$
     we can choose any of the two following options:
     - $A_i < A_j$
     - $A_j < A_i$

Greibach Normal Form

Example of step 3

```
G=<{E,T,F},
   {-,*,i}
   {E→E+T|T, T→T*F|F,  F→i},
   E>
```

- The following partial ordering can be deduced from the rules in the grammar:
  - **E→E+T** does not provide any ordering.
  - From **E→T** we can deduce **E<T**
  - From **T→T*F** we do not get any ordering.
  - From **T→F** we can deduce **T<F**
  - From **F→i** we cannot deduce anything.
- In conclusion,

  E<T<F

Greibach Normal Form

Proof of the theorem

4. We can classify the rules in the following three groups:
   - **Type 1**: Rules of the form
     $$A \to ax,, a \in \Sigma_T, \ x \in \Sigma^*$$
   - **Type 2**: Rules of the form
     $$A \to Bx,, \ A, B \in \Sigma_N, \ x \in \Sigma^* \wedge A < B$$
   - **Type 3**: Rules of the form
     $$A \to Bx,, \ A, B \in \Sigma_N, \ x \in \Sigma^* \wedge B < A$$

Greibach Normal Form

Example of step 4

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→FT', T'→*FT'|λ, F→i},
   E>
```

- Type 1 rules:
  - E'→+TE'
  - T'→*FT'
  - F→i
- Type 2 rules:
  - E→TE'
  - T→FT'
- There are no type 3 rules.
- The lambda-rules will be seen later.

Greibach Normal Form

Proof of the theorem

5. All the rules of type 3 will be deleted.
   - **Type 3:** Rules of the form

     $A \to Bx, \; A, B \in \Sigma_N, \; x \in \Sigma^* \wedge B < A$
   - To do this,
     1. We replace B with all the right-hand sides for B.
     2. This process is repeated for all the type-3 rules.
     3. If, during this process, there appear new left-recursive rules, we transform them using the procedure seen before.
     4. If there appear inaccessible symbols, we eliminate them.

Greibach Normal Form

Example of step 5

```
G'=<{A,B,C}, {a,b}
  {A→BC,  B→CA|a, C→AB|b},
   A>
```

- Type 3 rules:
  - C→AB
  - We substitute A with its right-hand sides (BC): C→ BCB

```
G'=<{A,B,C}, {a,b}
  {A→BC,  B→CA|a, C→BCB|b},
   A>
```


Greibach Normal Form

Example of step 5

- Now, there is a new type-3 rule:
  - C→BCB
  - We substitute B with its left-hand sides (CA|a): C→CACB|aCB

```
G'=<{A,B,C}, {a,b}
  {A→BC,  B→CA|a, C→ CACB|aCB|b},
   A>
```

Greibach Normal Form

Proof of the theorem

6. At this point, there should not be more type-3 rules. The next step will be the removal of 2-type rules, starting with the non-terminals which are at the end of the partial ordering.
   - **Type 2:** Rules of the following form:

     $A \to Bx,,\ A,B \in \Sigma_N,\ x \in \Sigma^* \wedge A<B$

   - To do this,
     1. B will be replaced by all the right-hand sides of the rules for B.
     2. This is repeated until we do not have any more type-2 or type-3 rules.
     3. If there appear new left-recursive rules, they will also be removed.
     4. Inaccessible symbols will also be removed.

Greibach Normal Form

Example of step 6

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→FT', T'→*FT'|λ, F→i},
   E>
```

- There are two type-2 rules, with the ordering **E<T<F**
  - **E→TE'**
  - **T→FT'**
- Firstly, we substitute **F** in **T→FT'** with its right-hand sides (**i**), and obtain:

  T→iT'

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```


Greibach Normal Form

Example of step 6

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```

- Next, we substitute **T** in **E→TE'** with its right-hand sides, (**iT'**). We obtain the rule

  E→iT'E'

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→iT'E',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```


Greibach Normal Form

Proof of the theorem

7. Now, all the rules belong to **type 1:** they all have the following form:

   $A \to ax,,\ a \in \Sigma_T,\ x \in \Sigma^*$

   - The only different with respect to Greibach Normal Form may be due to rules having more than one terminal symbol in the right-hand side.
   - This can be solved with a trivial substitution, adding a new non-terminal symbol, as in the following example.

     $A \to abC$

     We can replace that rule with:

     $A \to aBC,\ B \to b$

   - Where B is a new non-terminal symbol

Greibach Normal Form

Example of step 7

```
G'=<{A,X}, {a,b}
  {A→baX|aX, X→baX|λ},
   A>
```

- There are two rules in which the terminal symbol a has to be replaced by the new non-terminal symbol **A'**:
  - We add the new rule **A'→a**
  - We substitute **A→baX** by **A→bA'X**
  - We substitute **X→baX** by **X→bA'X**

```
G'=<{A,X,A'}, {a,b}
  {A→bA'X|aX, X→bA'X|λ, A'→a},
   A>
```


Greibach Normal Form

Proof of the theorem

8. The last thing to do is the treatment of lambda-rules.
   - In Greibach Normal Form, these rules are forbidden. The only exception is when the language contains the empty word, in which there has to be, necessarily, a lambda rule for the axiom of the grammar.
   - They should be removed using the procedure previously studied.

NOTE: For the purpose of building LL(1) compilers, the grammar needs not be exactly in Greibach Normal form:

- Some of the lambda rules will not be wrong for an LL(1) grammar.
- Otherwise, they will be removed using the procedure already seen.
- When these rules are removed, sometimes it is difficult to comply with all the conditions for LL(1) grammars.
  - In this case, it may be necessary to alter the grammar manually to obtain an equivalent one which can be restated as an LL(1) grammar.

Greibach Normal Form

Example 1

1. As the language does not contain the empty word, there is nothing to do.
2. Remove all the left-recursive rules:

```
G=<{E,T,F},
  {-,*,i}
  {E→E+T|T
   T→T*F|F
   F→i},
    E>
```

$A \to A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$

$A \to \beta_1 X | \ldots | \beta_m X$

$X \to \alpha_1 X | \ldots | \alpha_n X | \lambda$

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→TE'
   E'→+TE'|λ
   T→FT'
   T'→*FT'|λ
   F→i},
    E>
```


Greibach Normal Form

Example 1

3. We establish the partial ordering for the non-terminal symbols. In general, we can establish the ordering using the non-terminal symbols in the original grammar, and the ones added in step 2 may be added to the ordering if necessary.

```
G=<{E,T,F},
   {-,*,i}
   {E→E+T|T, T→T*F|F, F→i},
   E>
```

- The following ordering will be deduced from the previous rules:
  - From **E→E+T** we do not deduce anything
  - From **E→T** we deduce **E<T**
  - From **T→T*F** we deduce nothing
  - From **T→F** we deduce **T<F**
  - From **F→i** we can't deduce anything.
- In conclusion,

E<T<F

Greibach Normal Form

Example 1

4. The rules will be classified in three groups.
5. The grammar does not contain type-3 rules, so there is nothing to be done

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→FT', T'→*FT'|λ, F→i},
   E>
```

- Type-1 rules
  - E'→+TE'
  - T'→*FT'
  - F→i
- Type-2 rules
  - E→TE'
  - T→FT'
- No type-3 rules.
- Lambda rules will be treated at the end.

Greibach Normal Form

Example 1

6. Type-2 rules removal:

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→FT', T'→*FT'|λ, F→i},
   E>
```

- There are two type-2 rules, with the ordering **E<T<F**
  - E→TE'
  - T→FT'
- We first solve **F** in T→FT' and replace it by its right-hand sides (i), and obtain

T→iT'

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```


Greibach Normal Form

Example 1

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→TE',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```

- Next, we delete **T** in E→TE' and replace it with its right-hand sides (iT'), to obtain

E→iT'E'

```
G'=<{E,E',T,T',F}, {-,*,i}
  {E→iT'E',  E'→+TE'|λ, T→iT', T'→*FT'|λ, F→i},
   E>
```


Greibach Normal Form

Example 1

7. As there are no terminal symbols incorrectly set in the right-hand sides of the rules, there is nothing to do at this step.

Greibach Normal Form

Example 1

8. In the case that we want to obtain the grammar in Greibach Normal form, the last step would be to remove all the lambda-rules.

REMEMBER: this step is not strictly necessary for LL(1) parsers

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'
   E'→+TE'|+T
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```


Greibach Normal Form

Example 1

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'
   E'→+TE'|+T
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'|iT'|iE'|i
   E'→+TE'|+T
   T→iT'|i
   T'→*FT'|*F
   F→i},
    E>
```

When there are no more lambda-rules, we have the grammar, finally, in GNF.

Top-down analysis

Overview

- **Top-down analysis**
  - Blind search, depth-first search and width-first search.
  - Slow backtracking
- **Top-down analysis with LL(1) grammars**
  - Procedures for modifying grammars
    - Elimination of left-recursion
    - Elimination of lambda-rules
  - Greibach Normal Form
    - Definition
    - Procedure for obtaining the GNF of a grammar
    - Examples
  - LL(1) grammars
    - Initial examples
    - Definition of LL(1) grammars
    - Syntactic analysers based on LL(1) grammars

Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

- Some lambda rules are not incorrect in LL(1) grammars.
- Therefore, we are only going to remove those that are not right.

- The following procedure will be followed:
  - When a non-terminal has a lambda rule, e.g.

$$X \to \lambda$$

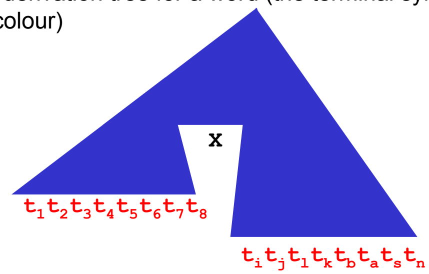
  - The following situation may arise: that the same non-terminal has other rules with a different right-hand side. Because we are in step 8, we know that all rules are now type-1 rules, starting with a terminal symbol.

$$X \to a\alpha,, \quad a \in \Sigma_T \wedge \alpha \in \Sigma^*$$

Adjusting a grammar in Greibach Normal Form to LL(1)

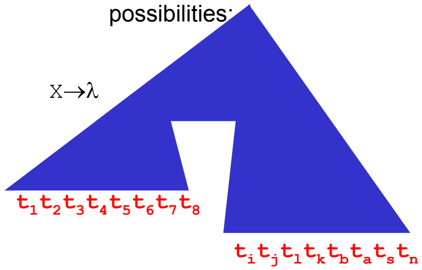
Changes in step 8

- Imagine a syntactic derivation tree for a word (the terminal symbols are represented in red colour)

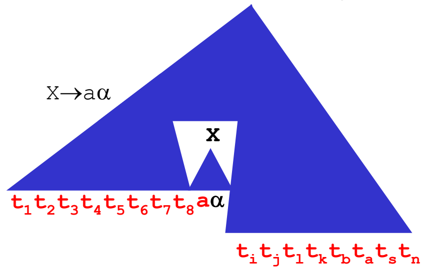


- If X is the only non-terminal that remains, there would be the following two possibilities:

$X \rightarrow \lambda$



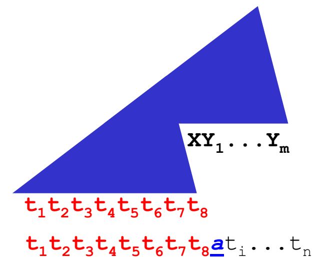
$X \rightarrow a\alpha$



Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

- Imagine that we are doing the top-down parsing of that word, and we are just before the symbol **a**



- Intuitively, the efficiency of LL(1), which is better than simple "top-down parsing with slow backtracking", is due to the indexation of the right-hand sides, for each non-terminal, using the next terminal to be analysed.
- This will be possible only if each non-terminal has just one right-hand side starting with each terminal.
- If we do not have lambda rules ($X \rightarrow \lambda$), the top-down analysis can be done completely without any branching in the analysis.

Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

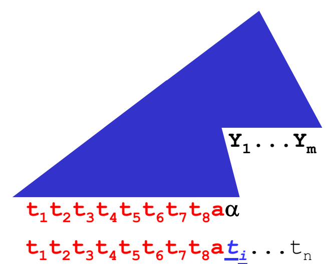
- If we had the following rules for a non-terminal X,

$$X \rightarrow t_1\beta_1$$
$$\dots$$
$$X \rightarrow t_{i-1}\beta_{i-1}$$
$$X \rightarrow \underline{a\alpha}$$
$$X \rightarrow t_{i+1}\beta_{i+1}$$
$$\dots$$
$$X \rightarrow t_k\beta_k$$
$$,,\{t_1,...,t_{i-1},a,t_{i+1},...,t_k\} \subseteq \Sigma_T$$
$$\wedge t_p \neq t_q \neq a \;\; \forall \in p,q \in \{1,...,i-1,i+1,...k\}$$
$$\wedge \{\beta_1,...,\beta_{i-1},a,\beta_{i+1},...,\beta_k\} \subseteq \Sigma_N^*$$

- There would be just one possibility to choose if the next symbol in the input string were "a".

Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

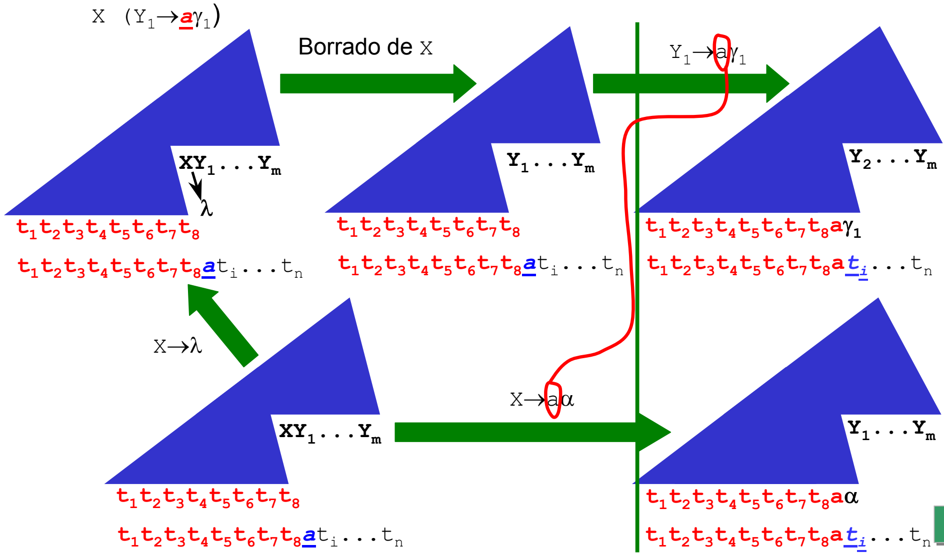


- Lambda-rules will be problematic whenever they produce several possibilities of choosing the next rule to apply.

Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

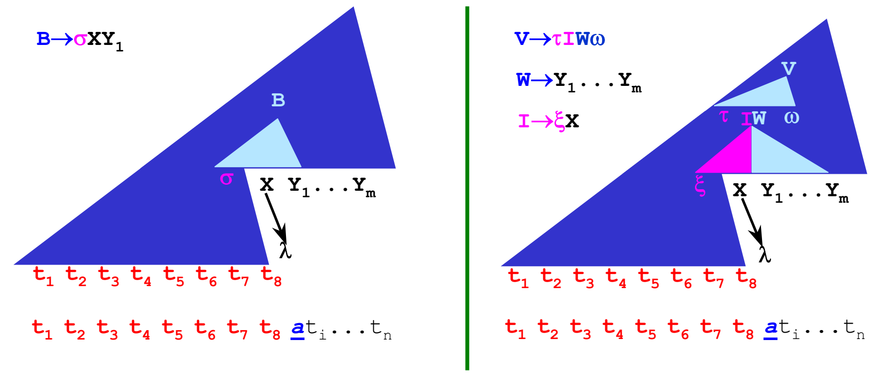
- Let us imagine that we have the following three rules: $X \rightarrow a\alpha$, $X \rightarrow \lambda$ y $Y_1 \rightarrow a\gamma_1$
- A terminal "a" can be derived directly from X ($X \rightarrow a\alpha$), and from Y_1, which follows X ($Y_1 \rightarrow a\gamma_1$)



Adjusting a grammar in Greibach Normal Form to LL(1)

Changes in step 8

- In order to identify these situations, we need to know:
  - Which are the first terminals derived by X (in our example, a)
  - Which is the first terminal that can be derived by what is after X, i.e. next(X)
- The following figures describe the two possible cases:



Adjusting a grammar in Greibach Normal Form to LL(1)

Step 8 for example 1

8. If we want to generate a top-down syntactic analyser with the LL(1) technique, we have to study, in step 8, which lambda-rules produce ambiguities.

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Who can follow **E'**? next(E')

We study all the right-hand sides that contain **E'**

- Let's start with the lambda-rule **E'→λ**.

Adjusting a grammar in Greibach Normal Form to LL(1)

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Who can follow **E'**? next(E')

We study all the right-hand sides that contain **E'**

In the two rules, **E'** appears as the last symbol in the right-hand side.

E' can be followed by anything that follows the left-hand side of those rules:

- **E'**
- **E**

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Therefore:
- next(E') is included in next(E') – obvious
- next(E) is included in next(E'). As E is the axiom, and it does not appear in any other right-hand side. next(E) is the end-of-program symbol, $.

So next(E') is {$}.

On the other hand, first(E') = {+}

- We can conclude that the first lambda rule is correct for LL(1), and can be left like that.

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Who can follow **T'**? next(T')

We study all the right-hand sides that contain **T'**

- Let us continue with the second lambda rule.

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Who can follow **T'**? next(T')

We study all the right-hand sides that contain **T'**

There is one rule in which **T'** is followed by **E'**.
In two other rules, it appears as the last symbol of the right-hand side.

Therefore, next(T') will be:
- first(E')
- next(T)
- next(T') -- obvious

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

We can focus now on **next(T)**

Let us see all the rules in which **T** appears in the right-hand side.

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

We can focus now on **next(T)**

Let us see all the rules in which **T** appears in the right-hand side.

It only appears followed by ***E'***.
Therefore, next(T) = first(***E'***)

In conclusion, next(T') will be:
- first(E')
- next(T) = first(E')
- next(T') -- obvious

- Therefore, next(T') = {+}

Step 8 for example 1

8. (cont.)

```
G'=<{E,E',T,T',F},
   {-,*,i}
  {E→iT'E'
   E'→+TE'|λ
   T→iT'
   T'→*FT'|λ
   F→i},
    E>
```

Finally, we need to check whether the lambda rule produces any ambiguity during the analysis.

For instance,
- The first terminal derived by **T'** is *.
- The terminals in next(T') are just +.

- Because the terminals are different (* y +) we can conclude that the lambda rule is appropriate for an LL(1) grammar.

Example 2

- Obtain, if possible, the LL(1) equivalent grammar from the following one:

```
G2=<{A,B},
   {a,b}
  {A→Ba | a
   B→Ab | b},
    A>
```


Example 2

1. The language does not contain the empty word, so there is nothing to do.
2. There are no left-recursive rules, so there is again nothing to do here.
3. We shall establish the partial ordering between the non-terminal symbols.

```
G2=<{A,B},
   {a,b}
  {A→Ba | a
   B→Ab | b},
    A>
```

- From the previous rules, we can deduce two different partial orderings:
  - From **A→Ba** we can deduce **A<B**
  - From **B→Ab** we can deduce **B<A**
- In summary, there are two options:
  - Option 1: **B<A**
  - Option 2: **A<B**
- We shall study the two possibilities separately, to see how the choice taken affects the result.

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

4. The rules will be classified in three groups

```
G2=<{A,B},
  {a,b}
  {A→Ba | a
   B→Ab | b},
   A>
```

- Type 1 rules
  - A→a
  - B→b
- Type 2 rules
  - B→Ab
- Type 3 rules
  - A→Ba

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

5. The type-3 rule will be eliminated by substituting **B**

```
G2=<{A,B},
  {a,b}
  {A→Ba | a
   B→Ab | b},
   A>
```

- Type-3 rules:
  - A→Ba
- The B will be replaced by its right-hand sides (Ab and b), and we obtain:
  - A→Aba | ba

```
G2=<{A,B},
  {a,b}
  {A→Aba | ba | a
   B→Ab | b},
   A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

- In all the steps, we need to check whether there is any inaccessible symbol.

```
G2=<{A,B},
  {a,b}
  {A→Aba | ba | a
   B→Ab | b},
   A>
```

- It can be easily seen that, in this grammar, B cannot be generated from the axiom.
- It is inaccessible, so we can eliminate it and all the rules where it is at the left-hand side:
  - B→Ab | b

```
G2=<{A},
  {a,b}
  {A→Aba | ba | a},
   A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

- We also need to eliminate all the left-recursive rules that have turned up:

```
G2=<{A},
  {a,b}
  {A→Aba | ba | a},
   A>
```

- The rule A→Aba is left-recursive, so we apply the lemma which said:
  - If we have a grammar $< \Sigma_T, \Sigma_N, S, P >$, the left-recursive rules
    - $A \to A\alpha_1 | \ldots | A\alpha_n | \beta_1 | \ldots | \beta_m$
  - will be substituted by
    - $A \to \beta_1 X | \ldots | \beta_m X$
    - $X \to \alpha_1 X | \ldots | \alpha_n X | \lambda$
- In our example,
  - A→baX | aX
  - X→baX | λ

```
G2=<{A,X},
  {a,b}
  {A→baX | aX
   X→baX | λ},
   A>
```

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

6. There no type-2 rules to eliminate
7. We eliminate all the terminal symbols that are not in the first position of the rules:

```
G2=<{A,X},
   {a,b}
   {A→baX | aX
    X→baX | λ},
     A>
```

- We define a new non-terminal **Z** to derive the 'a' (**Z→a**)
- And we substitute the 'a' in all the right-hand sides by 'Z': (**A→bZX** y **X→bZX**)

```
G2=<{A,X},{a,b}
   {A→bZX | aX
    X→bZX | λ
    Z→a},
     A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

8. Study of the λ-rules

```
G2=<{A,X},{a,b}
   {A→bZX | aX
    X→bZX | λ
    Z→a},
     A>
```

There is just one lambda-rule, for X.

Let us study the set next(X), to see whether it is different from first(X).

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 1 **B<A**

8. (cont.)

```
G2=<{A,X},{a,b}
   {A→bZX | aX
    X→bZX | λ
    Z→a},
     A>
```

Concerning next(X), it appears in three rules. From that, we can conclude that next(X) contains:

- next(A)
- next(X) - obvious

Because A is the axiom, and it does not appear in any right-hand side, we can conclude that next(A) will be the end-of-program symbol {$}, and hence next(X) will also be {$}.

On the other hand, first(X) = {b}

- This lambda-rule is not wrong for an LL(1) grammar.

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

4. Let us see now

```
G2=<{A,B},
   {a,b}
   {A→Ba | a
    B→Ab | b},
     A>
```

- Type-1 rules:
  - **A→a**
  - **B→b**
- Type-2 rules:
  - **A→Ba**
- Type-2 rules:
  - **B→Ab**

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

5. We start by eliminating the type-3 rules from the grammar:

```
G2=<{A,B},
  {a,b}
  {A→Ba | a
   B→Ab | b},
   A>
```

- Type-3 rules:
  - `B→Ab`
- We replace `A` with its right-hand sides (`Ba` and `a`) to get:
  - `B→ Bab | ab | b`

```
G2=<{A,B}, {a,b}
  {A→Ba | a
   B→Bab | ab | b},
   A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

- Now, we need to eliminate left-recursive rules.

```
G2=<{A,B}, {a,b}
  {A→Ba | a
   B→Bab | ab | b},
   A>
```

- The rule `B→Bab` is left-recursive, so we apply the lemma.
- In the example, we get:
  - `B→abX | bX`
  - `X→abX | λ`

```
G2=<{A,B,X}, {a,b}
  {A→Ba | a
   B→abX | bX
   X→abX | λ},
   A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

6. Remove type-2 rules:

```
G2=<{A,B,X}, {a,b}
  {A→Ba | a
   B→abX | bX
   X→abX | λ},
   A>
```

- There is one type-2 rule:
  - `A→Ba`
- We substitute `B` with all its right-hand sidres (`abX` y `bX`) to get:

```
G2=<{A,B,X}, {a,b}
  {A→abXa | bXa | a
   B→abX | bX
   X→abX | λ},
   A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

- As before, we need to check whether there is any inaccessible symbol:

```
G2=<{A,B,X}, {a,b}
  {A→abXa | bXa | a
   B→abX | bX
   X→abX | λ},
   A>
```

- Now, it is not possible to arrive to B from the axiom, so we can delete it and all the rules that have it in the left-hand side:
  - `B→abX | bX`

```
G2=<{A,X}, {a,b}
  {A→abXa | bXa | a
   X→abX | λ},
   A>
```

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

7. Removal of all the terminals that are not in the first position.

```
G2=<{A,X}, {a,b}
  {A→abXa | bXa | a
   X→abX | λ},
    A>
```

- First, we can eliminate the symbol *b*: we define a new non-terminal **B** to derive it (**B→b**), and we put the non-terminal in the fight-hand sides (**A→aBXa** y **X→aBX**)

```
G2=<{A,B,X}, {a,b}
  {A→aBXa | bXa | a
   X→aBX | λ
   B→b},
    A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

7. (cont.)

```
G2=<{A,B,X}, {a,b}
  {A→aBXa | bXa | a
   X→aBX | λ
   B→b},
    A>
```

- We also have a misplaced terminal *a*: we define a new non-terminal **Z** to derive it, (**Z→a**), and we substitute it in the problematic right-hand sides (**A→aBXZ | bXZ**)

```
G2=<{A,B,X,Z}, {a,b}
  {A→aBXZ | bXZ | a
   X→aBX | λ
   B→b
   Z→a},
    A>
```


Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

8. Study of the λ-rules.

```
G2=<{A,B,X,Z}, {a,b}
  {A→aBXZ | bXZ | a
   X→aBX | λ
   B→b
   Z→a},
    A>
```

We need to check whether next(X) and first(X) have any terminal symbol in common.

- There is just one lambda-rule.

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

8. (cont.)

```
G2=<{A,B,X,Z}, {a,b}
  {A→aBXZ | bXZ | a
   X→aBX | λ
   B→b
   Z→a},
    A>
```

Let us calculate next(X):

- **X** appears followed by *Z*.
- **X** also appears at the end of a rule.

Therefore, next(X) will contain:

- first(Z) = a
- next(X) - obvious

Adjusting a grammar in Greibach Normal Form to LL(1)

Example 2, option 2 **A<B**

8. (cont.)

```
G2=<{A,B,X,Z}, {a,b}
  {A→aBXZ | bXZ | a
   X→aBX | λ
   B→b
   Z→a},
    A>
```

Let us calculate next(X):

- **X** appears followed by ***Z***.
- **X** also appears at the end of a rule.

Therefore, next(X) will contain:

- first(Z) = a
- next(X) – obvious

But it is the case that:

- **first(X)** is the symbol a
- **next(X)=first(Z)=** a

- So we can conclude that this lambda-rule produces ambiguities during the parsing.

LL(1)

Concept

- **LL(1)** languages are

Those whose grammars appear in **Greibach Normal Form**, and there are not two rules for the same non-terminal, starting with the same terminal symbol in their right-hand side.

LL(1)

Converting from GNF to LL(1)

- A grammar in Greibach Normal Form may not be LL(1)
- For instance,

```
G2 =<{U,V,W,X,Y,Z,T},
    {a,b,c,d,e}
  {···
   U→aV | aW
   V→bX | cY
   W→dZ | eT
   ···},
  U>
```

- Two of the right-hand sides for **U** start with the same terminal symbol, ***a***.

LL(1)

Converting from GNF to LL(1)

- Sometimes, this cam be solved with a “common factor”.
- Steps
  - Obtain the longest common prefix of the two right-hand sides (in the example the symbol ***a***).
  - Leave a single right-hand side which starts with that common prefix, and which ends with a new non-terminal, e.g. **K**

```
G2 =<{U,V,W,X,Y,Z,T},
    {a,b,c,d,e}
  {···
   U→aK
   ···
   V→bX | cY
   W→dZ | eT
   ···},  U>
```

LL(1)

Converting from GNF to LL(1)

- The new non-terminal will generate the remaining of the right-hand sides, K→V|W.

```
G2 =<{U,V,W,X,Y,Z,T},
    {a,b,c,d,e}
  {···
   U→aK
   K→V  | W
   V→bX | cY
   W→dZ | eT
   ···},  U>
```


LL(1)

Converting from GNF to LL(1)

- We have to take care to leave the rules again in GNF...
  - We can derive the initial non-terminals in the rules for K, so they start with a on-terminal.
  - In this case, we can apply the rules for V and W:

```
G2 =<{U,V,W,X,Y,Z,T},
    {a,b,c,d,e}
  {···
   U→aK
   K→ bX | cY | dZ | eT
   V→bX | cY
   W→dZ | eT
   ···},  U>
```


LL(1)

Converting from GNF to LL(1)

- ...and we have to remove the inaccessible symbols (V y W)

```
G2 =<{U,V,W,X,Y,Z,T},
    {a,b,c,d,e}
  {···
   U→aK
   K→ bX | cY | dZ | eT
   ···
  },
 U>
```


Formalisation of LL(1) grammars

Concept

- In the remaining part of this lesson, we are going to describe formally LL(1) analysers, which we have already introduced informally with examples.

Formalisation of LL(1) grammars

Example

- Consider the following grammar, and the first and next sets for each non-terminal:

```
G=< {E,E',T,T',F},
    {+,*,(,),id}
    {   E   → TE'
        E'  → +TE'  | λ
        T   → FT'
        T'  → *FT'  | λ
        F   → (E)   | id      },
    E>

first(E)=first(T)=first(F)={(,id}
first(E')={+,λ}
first(T')={*,λ}
next(E)=next(E')={),$}
next(T)=next(T')={+,),$}
next(F)={+,*,),$}
```


Constructing LL(1) analysers

Algorithm

- The following algorithm calculates the analysis table $T \in M_{|\Sigma_N| \times |\Sigma_T|+1}$
- In this matrix, there will be a row for each non-terminal, and a column for each terminal, including the end-of-program symbol $. :

1. $\forall\ A \rightarrow \alpha \in P$ repeat:
   1. $\forall\ a \in first(\alpha) \cap \Sigma_T \quad A \rightarrow \alpha\ \in T[A,a]$.
   2. If $\lambda \in first(\alpha)$
   - then, $\forall\ b \in next(A) \quad A \rightarrow \alpha\ \in T[A,b]$ (note that b can also be $\$$)

Constructing LL(1) analysers

Examples

- In the grammar for the previous example

Σ_N	$\Sigma_T \cup \{\$\}$					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→λ	E'→λ
T	E→FT'			E→FT'		
T'		T'→λ	T'→*FT'		T'→λ	T'→λ
F	F→id			F→(E)		

Constructing LL(1) analysers

Examples

- Given the following grammar,

```
G=< {P,P',E},
    {i,t,a,e,b}
    {   P   → iEtPP' | a
        P'  → eP     | λ
        E   → b
    P>
```

Constructing LL(1) analysers

Examples

- We can obtain the next table:

Σ_N	$\Sigma_T \cup \{\$\}$					
	a	b	e	i	t	$
P	P→a			P→iEtPP′		
P′			P′→λ P′→eP			P′→λ
E		E→b				

LL(1), first and next sets

Definition

- We can define **LL(1) grammars** as those that comply with the following condition:

> The analysis table constructed with the previous procedure is deterministic, i.e., all the rows contain at most one rule.

Examples

- The grammar in the first example is LL(1)
- The grammar in the second example is not LL(1)

Selective top-down analysers

Concept

- LL(1) analysers are also called "with no backtracking", because they are deterministic and it will never be necessary to backtrack during the analysis of a program.
- They are also called "recursive-descent" parsers, because of the kind of analysis programs that are produced from LL(1) grammars.
- The reason of the efficiency of these parsers is because the right-hand sides of the rules for each non-terminal symbol can be considered to be indexed by the next terminal.

Selective top-down analysers

Automatically building a parser for an LL(1) grammar

- According to the procedure described, in an LL(1) grammar we are going to have two kinds of rules:
  - **Rules for non-terminals that have a λ-rule:**

$$U \rightarrow xX_1X_2...X_n \mid yY_1Y_2...Y_m \mid ... \mid zZ_1...Z_p \mid \lambda$$

  - **Rules for non-terminals that do not have a λ-rule:**

$$U \rightarrow xX_1X_2...X_n \mid yY_1Y_2...Y_m \mid ... \mid zZ_1...Z_p$$

Selective top-down analysers

Automatically building a parser for an LL(1) grammar: λ rules

- The next C function will be generated:

```
int U(char * string, int i)
{
 if (i < 0) return i;
/* this propagates errors */
 switch (string[i]) {
   case x;
     i++;
     i=X1(string, i);
     i=X2(string, i);
         ...
     i=Xn(string, i);
     break;
   case y:
     i++;
     i=Y1(string, i);
     i=Y2(string, i);
         ...
     i=Ym(string, i);
     break;

   ...
```

```
   case z:
     i++;
     i=Z1(string, i);
     i=Z2(string, i);
         ...
     i=Zp(string, i);
     break;
 }
 return i;
}
```


Selective top-down analysers

Automatically building a parser for an LL(1) grammar: rules without λ

- The next C function will be generated:

```
int U(char * string, int i)
{
 if (i < 0) return i;
/* This propagates errors */
 switch (string[i]) {
   case x;
     i++;
     i=X1(string, i);
     i=X2(string, i);
         ...
     i=Xn(string, i);
     break;
   case y:
     i++;
     i=Y1(string, i);
     i=Y2(string, i);
         ...
     i=Ym(string, i);
     break;

   ...
```

```
   case z:
     i++;
     i=Z1(string, i);
     i=Z2(string, i);
         ...
     i=Zp(string, i);
     break;
   /* End-of-string goes to the
default case */
   default:
     return -n;
   /* The error will be
different for each rule */
  }
 return i;
}
```


Selective top-down analysers

Complete example

- Let us start with the following context-independent grammar

```
G3 =<{E,T,F},{i,+,-,*,/,(,)}
   {E → T+E | T-E | T
    T → F*T | F/T | F
    F → i | (E)
   },
 E>
```


Selective top-down analysers

Complete example

- The following is the grammar in Greibach Normal Form

```
G3 =<{E,T,M,S,P,D,C},
     {i,+,-,*,/,(,)}
  {E→ iPTME | (ECPTME | iDTME | (ECDTME | iME | (ECME |
      iPTSE | (ECPTSE | iDTSE | (ECDTSE | iSE | (ECSE |
      iPT   | (ECPT   | iDT   | (ECDT   | i   | (EC
   T→ iPT   | (ECPT   | iDT   | (ECDT   | i   | (EC
   M→ +
   S→ -
   P→ *
   D→ /
   C→ )},  E>
```

Complete example

- The following is a possible LL(1) grammar, obtained from the GNF grammar by taking "common factor" in the right-hand sides of the rules.

$$
\begin{aligned}
G_3 = <\{&E,T,M,S,P,D,C\},\\
&\{i,+,-,*,/,(,)\}\\
\{&E \rightarrow iV \mid (ECV\\
&V \rightarrow *TX \mid /TX \mid +E \mid -E \mid \lambda\\
&X \rightarrow +E \mid -E \mid \lambda\\
&T \rightarrow iU \mid (ECU\\
&U \rightarrow *T \mid /T \mid \lambda\\
&C \rightarrow)\},\quad E>
\end{aligned}
$$

Complete example

- The following would be the LL(1) analyser (continued)

```
int E(char * string, int i)
{
 if (i < 0) return i;
/* This propagates previous
errors */
 switch (string[i]) {
   case 'i':
     i++;
     i=V(string, i);
     break;
   case '(':
     i++;
     i=E(string, i);
     i=C(string, i);
     i=V(string, i);
     break;
   default: return -1;/*no λ*/
  }
 return i;
}
```

```
int V(char * string, int i)
{
 if (i < 0) return i;
/* This propagates previous
errors */
 switch (string[i]) {
   case '*':
   case '/':
     i++;
     i=T(string, i);
     i=X(string, i);
     break;
   case '+':
   case '-':
     i++;
     i=E(string, i);
     break; /* λ */
  }
 return i;
}
```


Complete example

- The following would be the LL(1) analyser (continued)

```
int X(char * string, int i)
{
 if (i < 0) return i;
/* This propagates previous
errors */
 switch (string[i]) {
   case '+':
   case '-':
     i++;
     i=E(string, i);
     break; /* λ */
 }
 return i;
}
```

```
int T(char * string, int i)
{
 if (i < 0) return i;
/* This propagates previous
errors */
 switch (string[i]) {
   case 'i':
     i++;
     i=U(string, i);
     break;
   case '(':
     i++;
     i=E(string, i);
     i=C(string, i);
     i=U(string, i);
     break;
   default: return -2;/*no λ*/
  }
 return i;
}
```


Complete example

- The following would be the LL(1) analyser (continued)

```
int U(char * string, int i)
{
 if (i < 0) return i;
/* This propagates previous
errors */
 switch (string[i]) {
   case '*':
   case '/':
     i++;
     i=T(string, i);
     break; /* λ */
 }
 return i;
}
```

```
int C(char * string, int i)
{
 if (i < 0) return i;
/* Propagate previous errors
*/
 switch (string[i]) {
   case ')':
     i++;
     break;
   default: return -4;/*no λ*/
 }
 return i;
}
```

Selective top-down analysers

Complete example

- A string x would be analysed with the following call:

```
axiom( x, 0);
```

- If the value returned equals the length of the string, it was correct. If it is a negative number, it was incorrect.
- The following execution example illustrates the LL(1) analysis:

```
E("i + i * i", 0)
 0[1]: E( "I + I * I", 0)
   1[1]: V( "I + I * I", 1)
     2[1]: E( "I + I * I", 2)
       3[1]: V( "I + I * I", 3)
         4[1]: T( "I + I * I", 4)
           5[1]: U( "I + I * I", 5)
           5[1]: returned 5
         4[1]: returned 5
         4[1]: X( "I + I * I", 5)
         4[1]: returned 5
       3[1]: returned 5
     2[1]: returned 5
   1[1]: returned 5
 0[1]: returned 5
5
```


Selective top-down analysers

Complete example

- The following string will not be recognised

```
E("i + i *", 0)
 0[1]: E( "I + I *", 0)
   1[1]: V( "I + I *", 1)
     2[1]: E( "I + I *", 2)
       3[1]: V( "I + I *", 3)
         4[1]: T( "I + I *", 4)
         4[1]: returned -2
         4[1]: X( "I + I *", -2)
         4[1]: returned -2
       3[1]: returned -2
     2[1]: returned -2
   1[1]: returned -2
 0[1]: returned -2
-2
```


Syntactic analysis

Bibliography

[Alf] "*Teoría de Autómatas y lenguajes formales*" M. Alfonseca y otros

[Hop] *"Introducción a la teoría de autómatas, lenguajes y computación*" Hopcroft, J.; Motwani, R.; Ullman, J.

[Aho] "*Compiladores. Principios, técnicas y herramientas*" A. V. Aho; R. Sefthi; J. D. Ullman