Context-free Grammar

BNF-notation

<u1> → <u1> + <u2> | <u2>
<u2> → <u2> * <u3> | <u3>
<u3> → (<u1>) | a | b

Derivation

Substitute all occurrences of nonterminal symbols with a right hand side of a BNF-rule for the non-terminal:

<u1> ⇒ <u1> + <u2> ⇒ <u2> + <u2> ⇒

<u3> + <u2> ⇒ <u3> + <u2> * <u3> ⇒

<u3> + <u2> * a ⇒ <u3> + <u3> * a ⇒

<u3> + b * a ⇒ a + b * a

↑ the substitution has terminated

From this the names terminal/non-terminal:

<u1>, <u2> , <u3>	are non-terminal symbols
a , b , (,) , + , *	are terminal symbols

Context-Free Grammar

$G=(V_T,V_N,S,P)$

- V_T : the Terminal Alphabet, the set of terminal symbols
- V_N : the Nonterminal Alphabet, the set of Nonterminal symbols

  $V= V_T \cup V_N$
- S : startsymbol, $S \in V_N$
- P : a finite non-empty set of productions

  A production has the form

  $A \rightarrow \alpha$, where $A \in V_N$, $\alpha \in V^*$

Example:

V_T = {a,b,(,),+,*}
V_N = {E,T,F}
S = E
P= E → E + T
E → T
T → T * F
T → F
F → (E)
F → a
F → b

Derivation (formal definition)

Consider a context-free grammar $G=(V_T,V_N,S,P)$, $V= V_T \cup V_N$

Relations in V^* :

$\Rightarrow$	*derives in one step*
	definition; $\alpha A\beta \Rightarrow \alpha\gamma\beta$ iff $(A\rightarrow\gamma)\in P$, $\alpha, \beta \in V^*$
$\Rightarrow^+$	*derives in one or more steps*
	definition: Let $\alpha_1 , \alpha_2 , ... , \alpha_n \in V^*$ such that $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow ... \Rightarrow \alpha_n$, (n>1), a derivation of α_n from α_1 then $\alpha_1 \Rightarrow^+ \alpha_n$ or α_1 *derives in one ore more steps* α_n
$\Rightarrow^*$	*derives in zero or more steps*
	definition: $\alpha \Rightarrow^* \beta$ iff $\alpha \Rightarrow^+ \beta$ or $\alpha = \beta$

The Language defined by G

$G=(V_T,V_N,S,P)$, $V= V_T \cup V_N$

$S \Rightarrow^* \alpha$, $\alpha \in V^*$, α is a *sentential form*

$S \Rightarrow^* x$, $x \in V_T^*$, x is a *sentence*

$L(G) = \{x \mid S \Rightarrow^+ x \wedge x \in V_T^*\}$, the set of all sentences

Context-Free Grammar

Derivation

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid a \mid b$

Derives in one step $\Rightarrow$

$T + F * a \Rightarrow T + (E)* a$

Derives in one ore more steps $\Rightarrow^+$

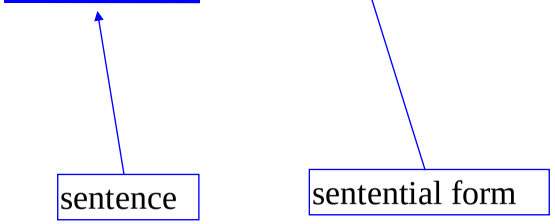
$T + \underline{F} * a \Rightarrow T + (\underline{E})* a \Rightarrow T + (E + \underline{T})* a \Rightarrow$
$T + (E + \underline{F})* a \Rightarrow T + (E + b) * a$

is a derivation of $T + (E + b) * a$ from $T + F * a$ so

$T + F * a \Rightarrow^+ T + (E + b) * a$

Sentential Form, Sentence

$\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow a + \underline{T} \Rightarrow$
$a + \underline{T} * F \Rightarrow a + \underline{F} * F \Rightarrow a + b * \underline{F} \Rightarrow$
$a + b * b$



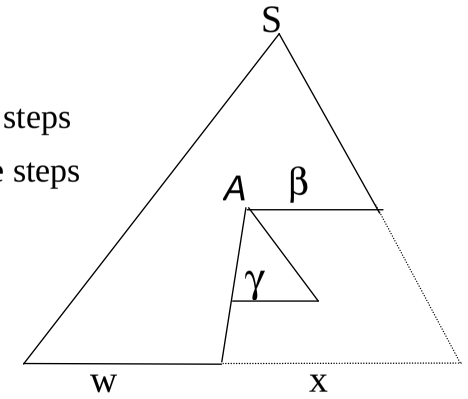
Leftmost & Rightmost Derivations, Parse Trees

Leftmost derivation:

In every derivation step expand leftmost nonterminal:

$S \Rightarrow_l^* w\,A\,\beta \Rightarrow_l w\,\gamma\,\beta \Rightarrow_l^* w\,x\,, \quad (A \to \gamma) \in P, \quad w \in V_T^*, \quad \beta \in V^*$

$\Rightarrow_l$	leftmost derives in one step
$\Rightarrow_l^+$	leftmost derives in one or more steps
$\Rightarrow_l^*$	leftmost derives in zero or more steps
$S \Rightarrow_l^* \alpha$	α is a left sentential form

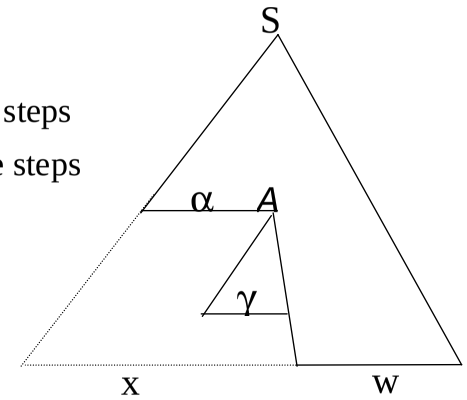


Rightmost derivation:

In every derivation step expand rightmost nonterminal:

$S \Rightarrow_r^* \alpha\,A\,w \Rightarrow_r \alpha\,\gamma\,w \Rightarrow_r^* x\,w\,, \quad (A \to \gamma) \in P, \quad w \in V_T^*, \quad \alpha \in V^*$

$\Rightarrow_r$	rightmost derives in one step
$\Rightarrow_r^+$	rightmost derives in one or more steps
$\Rightarrow_r^*$	rightmost derives in zero or more steps
$S \Rightarrow_r^* \alpha$	α is a right sentential form



A sentence $x \in L(G)$ usually has many derivations

including $S \Rightarrow_l^* x$ (the leftmost derivation), $S \Rightarrow_r^* x$ (the rightmost derivation).

Leftmost Derivations, Parse Trees

$\underline{E} \Rightarrow$

$\underline{E} + T \Rightarrow$

$\underline{T} + T \Rightarrow$

$\underline{F} + T \Rightarrow$

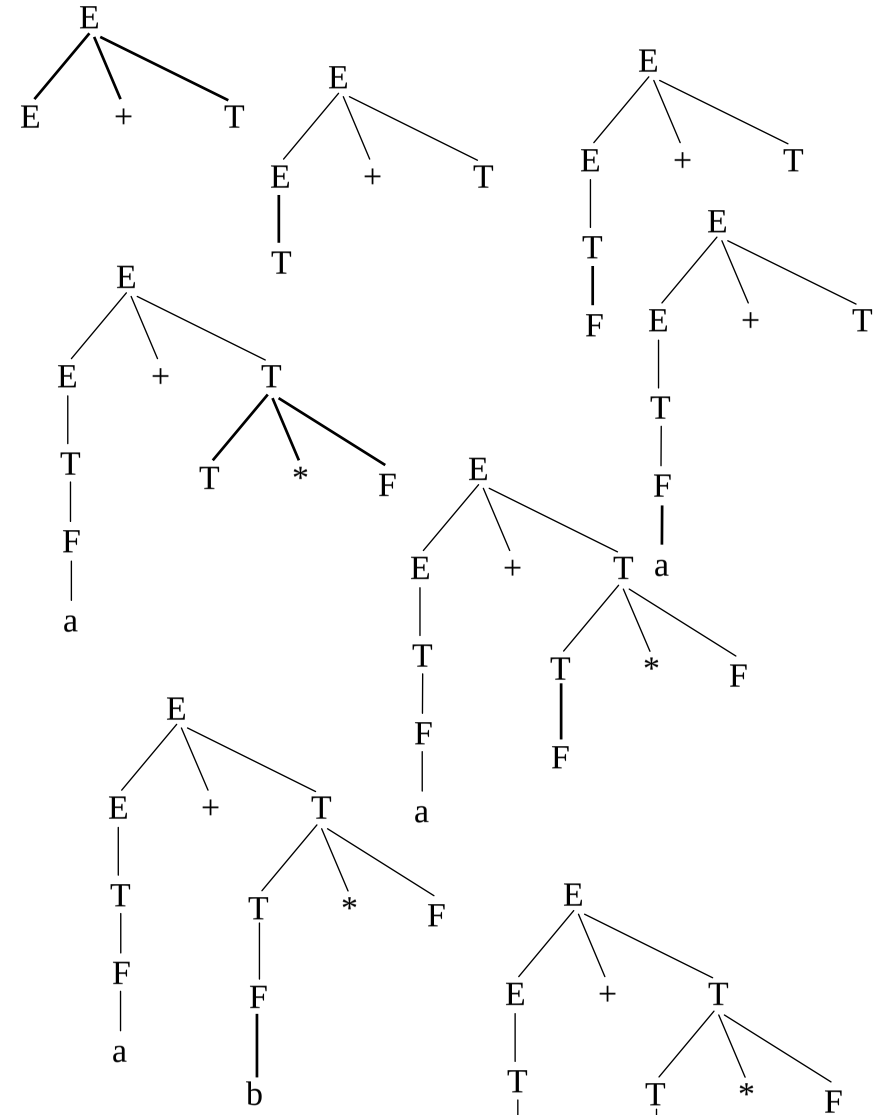
$\underline{a} + T \Rightarrow$

$a + \underline{T} * F \Rightarrow$

$a + \underline{F} * F \Rightarrow$

$a + b * \underline{F} \Rightarrow$

$a + b * a \Rightarrow$



Grammar:

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid a \mid b$

Rightmost Derivations, Parse Trees

$\underline{E} \Rightarrow$

$E + \underline{T} \Rightarrow$

$E + T * \underline{F} \Rightarrow$

$E + \underline{T} * a \Rightarrow$

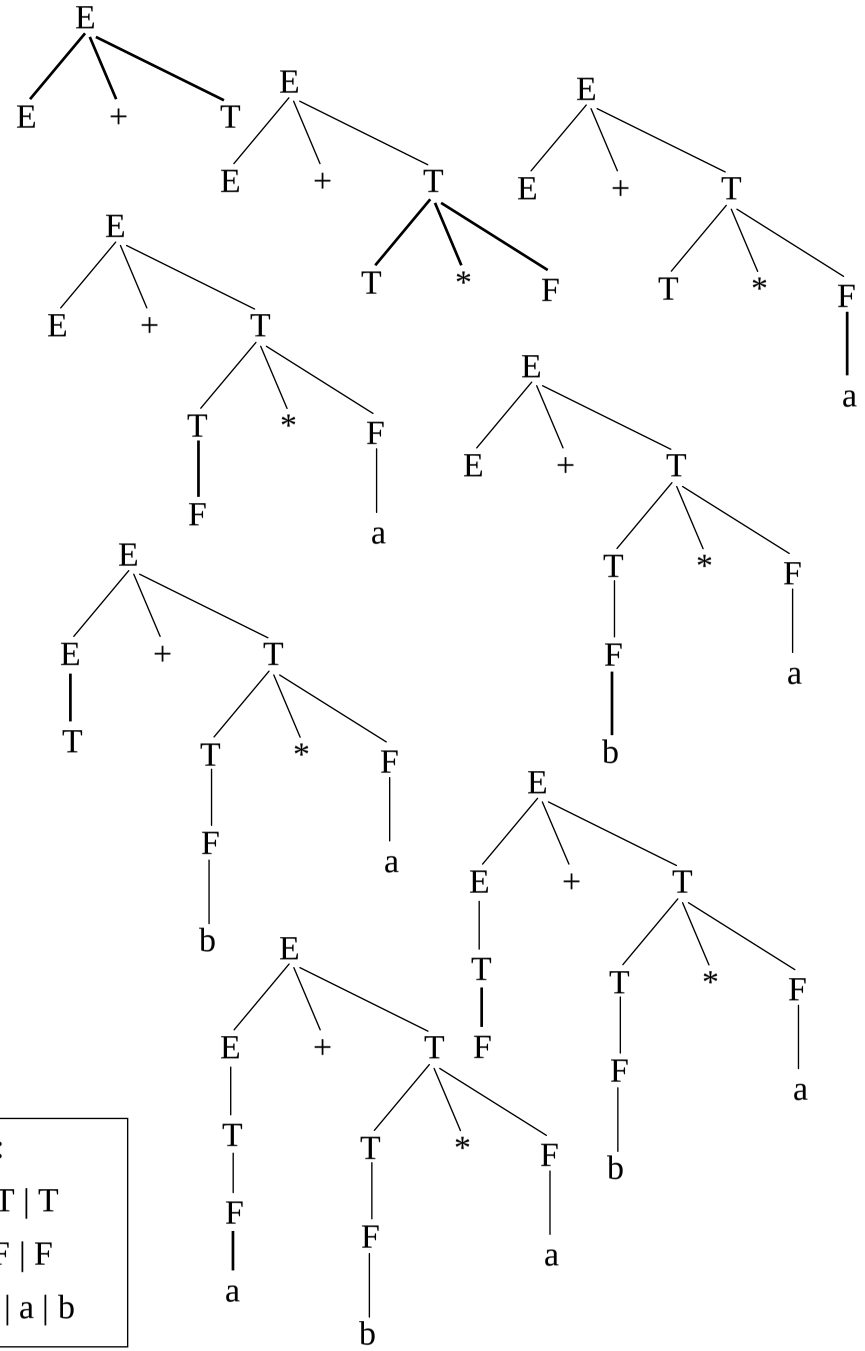
$E + \underline{F} * a \Rightarrow$

$\underline{E} + b * a \Rightarrow$

$\underline{T} + b * a \Rightarrow$

$\underline{F} + b * a \Rightarrow$

$a + b * a \Rightarrow$



Grammar:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a \mid b$

One Language, Several Grammars

A language may be defined by several different grammars:

Example: the Expression Language

G1: $E \rightarrow E + T \mid T$

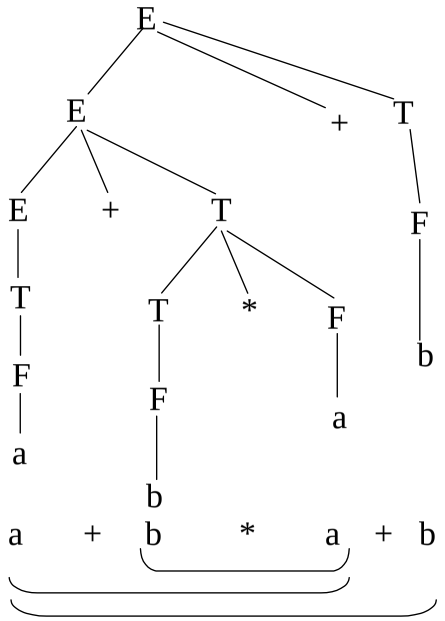
$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid a \mid b$

Operators have different priority,

$pr(*) > pr(+)$.

Operators are left associative .

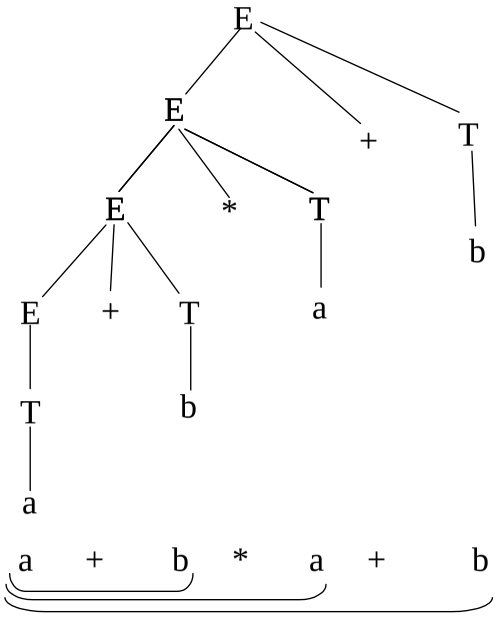


G2: $E \rightarrow E + T \mid E * T \mid T$

$T \rightarrow (E) \mid a \mid b$

Operators have same priority.

Operators are left associativ.



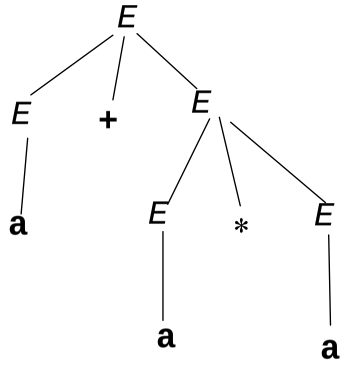
Grammars Defining the Same Language

E → *E* **+** *T* | *T* *T* → *T* ***** *F* | *F* *F* → **(** *E* **)** | **a** | **b**	+ has lower priority than * + and * are left associative
E → *E* **+** *T* | *E* ***** *T* | *T* *T* → **(** *E* **)** | **a** | **b**	+ and * have same priority + and * are left associative
E → *T* **+** *E* | *T* *T* → *F* ***** *T* | *F* *F* → **(** *E* **)** | **a** | **b**	+ has lower priority than * + and * are right associative
E → *E* **+** *T* | *T* *T* → *F* ***** *T* | *F* *F* → **(** *E* **)** | **a** | **b**	+ has lower priority than * + is left associative, * is right associative
E → *T Em* *Em* → **+** *T Em* | ε *T* → *F Tm* Tm → ∗ F Tm | ε *F* → **(** *E* **)** | **a** | **b**	Transformed grammar for top-down syntax-analyses (left recursion removed)
E → *E* **+** *E* | *E* ***** *E* | **(** *E* **)** | **a** | **b**	priority ? associativity ?

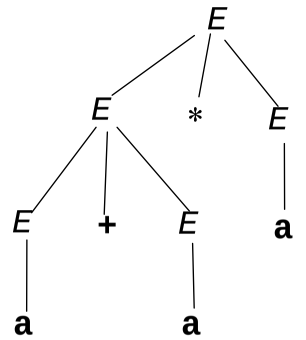
Ambiguous Grammars

E → *E* **+** *E* \|
E ***** *E* \|
(*E* **)** \|
a \| **b**

E ⇒R *E* **+** *E* ⇒R *E* **+** *E* ∗ *E* ⇒R
E **+** *E* ∗ **a** ⇒R *E* **+** **a** ∗ **a** ⇒R
a **+** **a** ∗ **a**



E ⇒R *E* ∗ *E* ⇒R *E* ∗ **a** ⇒R
E **+** *E* ∗ **a** ⇒R *E* **+** **a** ∗ **a** ⇒R
a **+** **a** ∗ **a**



Two different rightmost-derivations of the same expression, and hence two different parse-trees !!

Notice, it is the grammar that is ambiguous, not the language.

If (in a language reference) a language is defined by an ambiguous grammar, *disambiguating rules* must be added, e.g.:

E → *E* **+** *E* \|
E ***** *E* \|
(*E* **)** \|
a \| **b**

Disambiguating rules:

The operators + and ∗ are left associative.

The operator ∗ has greater priority than +

Ambiguous Grammars (dangling else)

```
stat → if cond then stat |
       if cond then stat else stat
       id := expr |
       ...

cond → ...
```

Unambiguous statements:

if C_1 then if C_2 then S_2

if C_1 then S_1 else if C_2 then S_2

if C_1 then if C_2 then S_1 else S_2 else S_3

Ambiguous statements:

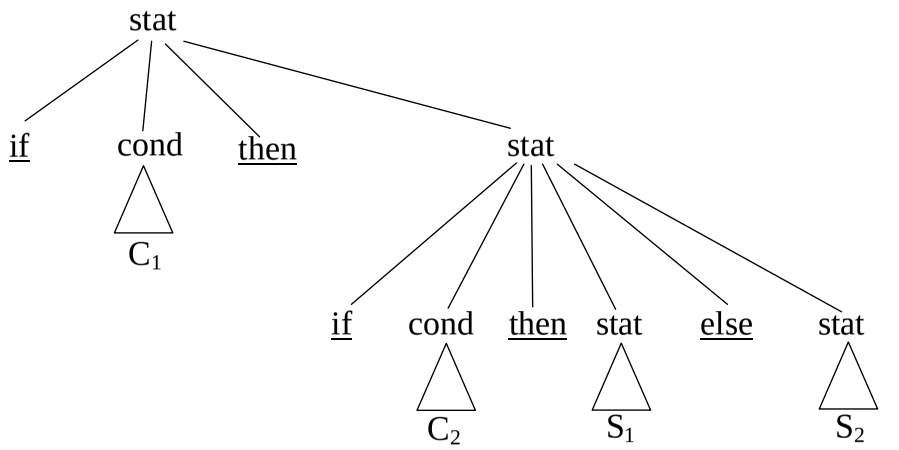
if C_1 then if C_2 then S_1 else S_2 ?

if C_1 then if C_2 then if C_3 then S_1 else S_2 else S_3 ?

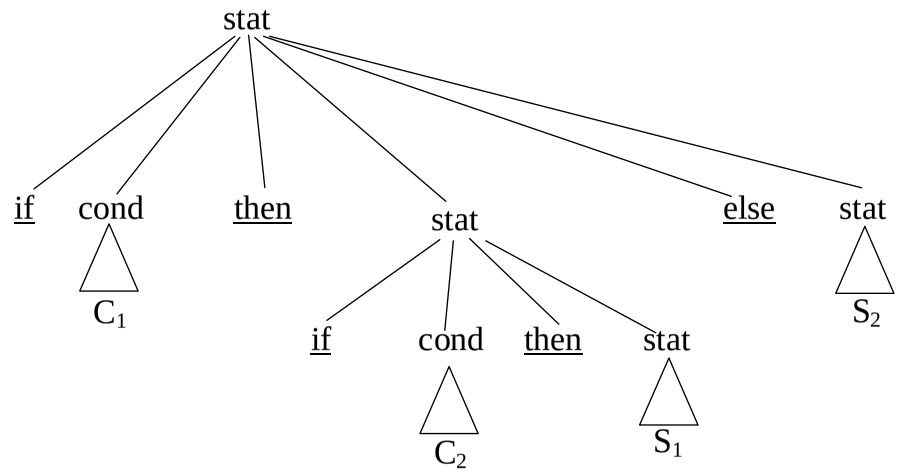
Disambiguating rule:

An **else** should always be paired with the previous unmatched **then**

Ambiguous Grammars (dangling else)



if C1 then **if** **C2** **then** **S1** **else** **S2**



if C1 then **if** **C2** **then** **S1** else S2

Grammars Defining the Same Language

Operator Priority/ Binding Strength

G1: *u0* → **if** *u0* **then** *u0* **else** *u0* | *u1* *u1* → *u1* **&** *u2* | *u2* *u2* → *u3* **=** *u3* | *u3* *u3* → *u3* **+** *u4* | *u4* *u4* → *u4* ***** *u5* | *u5* *u5* → (*u0*) | **Id** | **Const**	lowest priority/ binding strength highest priority/ binding strength

a + **(** **if** a = b **&** c = d **then** 2 **else** a + b + c ∗ d **)** + e

G2: *u0* → *u0* **&** *u1* | *u1* *u1* → *u2* **=** *u2* | *u2* *u2* → *u2* **+** *u3* | *u3* *u3* → *u3* ***** *u4* | *u4* *u4* → **if** *u0* **then** *u0* **else** *u4* | *u5* *u5* → (*u0*) | **Id** | **Const**	lowest priority/ binding strength highest priority/ binding strength

a + **(** **if** a = b **&** c = d **then** 2 **else** a + b + c ∗ d **)** + e

TopDown Parsing

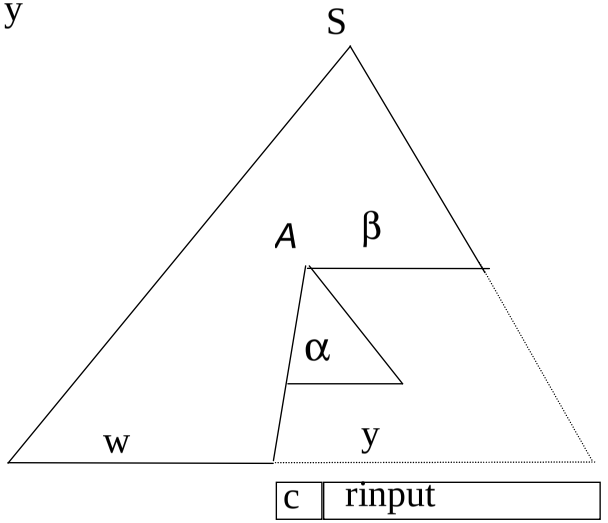
Consider a context-free grammar $G=(V_T,V_N,S,P)$, $V= V_T \cup V_N$

Let $x \in V^*$, find leftmost derivation $S \Rightarrow_L^* x$.

The general step:

$$S \Rightarrow_L^* \underbrace{w\ A\ \beta}\ \Rightarrow_l w\ \alpha\ \beta \Rightarrow_L^* w\ y = x,$$

select $(A \rightarrow \alpha) \in \{ A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots , A \rightarrow \alpha_n, \}$
such that $\alpha\ \beta \Rightarrow_l^* y$



A grammar is LL(1)
if it is possible to determine the production A→α
from A and the *lookahead symbol* c

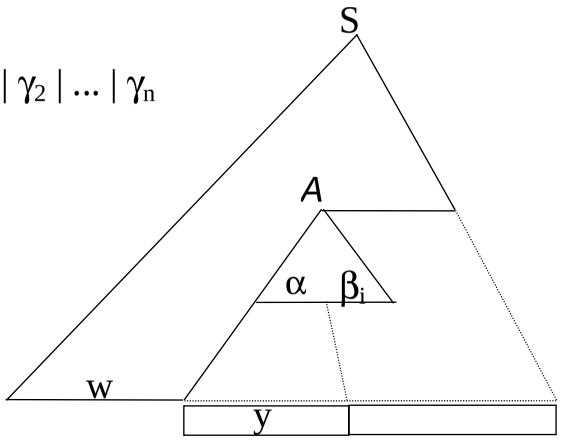
LL(1) ~**L**eft to Right Parse, **L**eftmost-derivation, **1**-symbol lookahead.

Necessary conditions for LL(1):

- No left recursive symbols (e.g. *u2* → *u2* **+** *u3* | *u3*)
- No alternatives starting with same symbol (e.g. *u2* → *u3* **+** *u2* | *u3*)

TopDown Parsing, Left Factoring

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_n$



Select $A \rightarrow \alpha\beta_i \in \{A \rightarrow \alpha\beta_1 , A \rightarrow \alpha\beta_2 , ... , A \rightarrow \alpha\beta_m\}$, impossible !!

Defer decision until the difference appears:

$A \rightarrow \alpha \underbrace{(\beta_1 \mid \beta_2 \mid ... \mid \beta_m)}_{Atail} \mid \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_n$

$\Downarrow$

$A \rightarrow \alpha \ \ Atail \mid \gamma_1 \mid \gamma_2 \mid ... \mid \gamma_n$

$Atail \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_m$

Example:

$S \rightarrow$ **if** *C* **then** *S* **else** *S* **fi** | **if** *C* **then** *S* **fi**

$\Downarrow$

$S \rightarrow$ **if** *C* **then** *S* **(** **else** *S* **fi** | **fi** **)**

$\Downarrow$

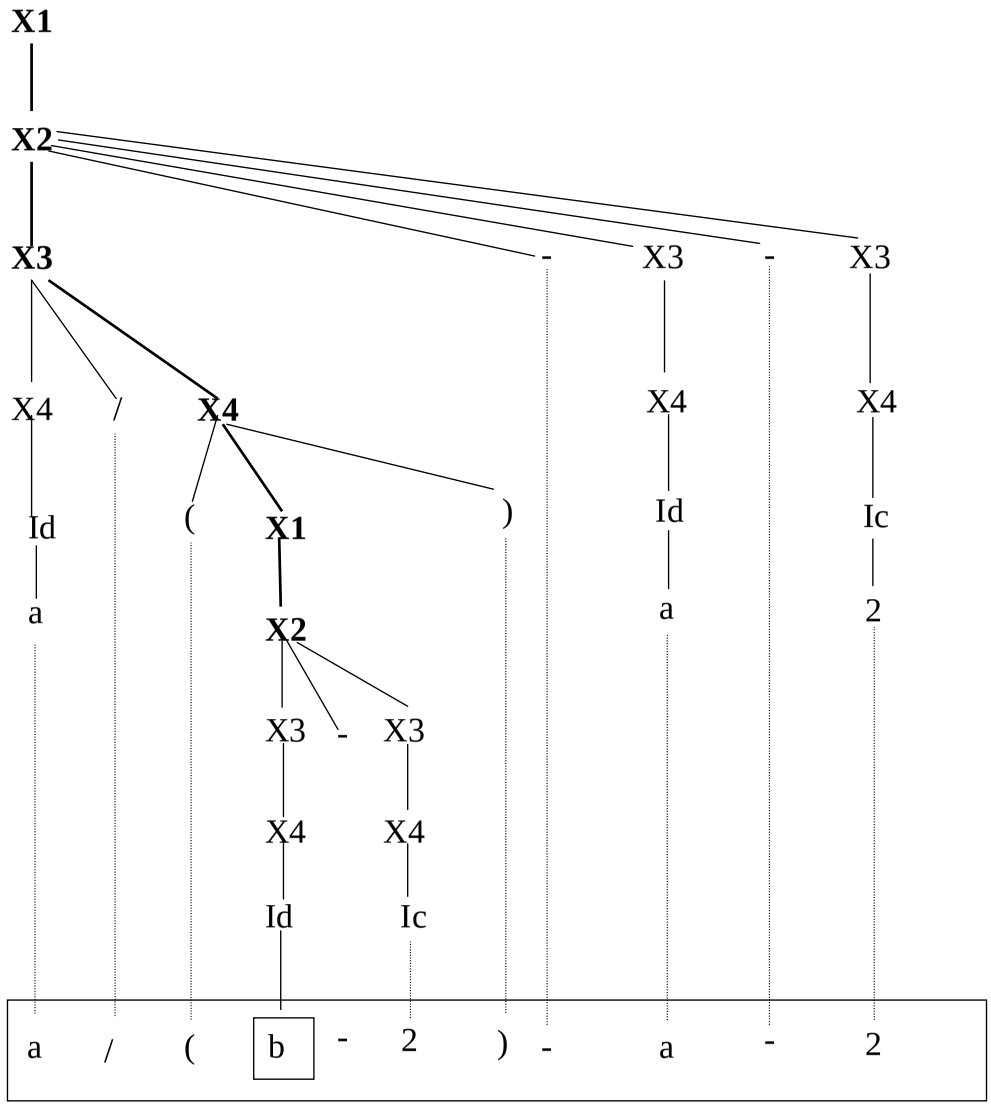
$S \rightarrow$ **if** *C* **then** *S Iftail*

Iftail $\rightarrow$ **else** *S* **fi** | **fi**

Recursive Descent TopDown Parsing

Using EBNF:

```
X1 = X2 [RelOprt X2 ]
RelOprt = ">" | ">="
X2 = X3 { "-" X3 }
X3 = X4 { "/" X4 }
X4 = "(" X1 ")" | Ic | Id
```



Recursive Descent Parser - top down (1)

```
Command   = "quit" | "show" X0 | "eval" X0
```

```
nextCommand()
{  Input new command from user;
   Find firstToken;
   switch(firstToken)
   {  case EOF_T:    break; //empty command line
      case QUIT_T:   //quit if nextToken is EOF
      case SHOW_T:{ findNextToken(); ep= x0P();
                    nextToken should be EOF_T
                    Print ep to screen;} break;
      case EVAL_T:  { findNextToken(); ep= x0P();
                    nextToken should be EOF_T;
                    print value of ep;} break;
      default:      syntaxerror;
   }
}
```


Recursive Descent Parser (2)

```
X0     = X1 { "-" X1 }
X1     = X2 { "/" X2 }
X2     = "(" X0 ")" | Const | Id
```

```
X0P()
{  ep= x1P();
   while(currentToken == MINUS_T)
     {  findNextToken(); rop= x1P();
        ep= Sub(ep,rop);
     }
   return ep;
}

x1P()
{  ep= x2P();
   while(currentToken == DIV_T)
     {  findNextToken(); rop= x2P();
        ep= new Div(ep, rop);
     }
   return ep;
}

x2P()
{  switch(currentToken)
   {  case LeftP_T: findNextToken; ep= x0P();
                    nextToken should be a right parenthesis;
                    return ep;
      case ICONST_T:
        return the value of the constant;
      case ID_T:
        {  Id= Sc.getText; findNextToken;
           return Value of 'Id'; }
      default: syntaxerror();
}
```