Definition of derivation

One-step derivation $\to$

One-step derivation for a grammar $G = (T, N, P)$

- it has shape $\alpha_1 B \alpha_2 \to \alpha_1 \gamma \alpha_2$
- $\alpha_1, \alpha_2 \in (T \cup N)^*$
- $(B, \gamma) \in P$ that is, (B, γ) is a production of G

Multi-step derivation $\to^+$

Transitive closure of $\to$:

- base case: if $\gamma_1 \to \gamma_2$, then $\gamma_1 \to^+ \gamma_2$
- inductive case: if $\gamma_1 \to \gamma_2$ and $\gamma_2 \to^+ \gamma_3$, then $\gamma_1 \to^+ \gamma_3$

Definition of language

Definition of language generated by a grammar

Language L_B generated from $G = (T, N, P)$ for non-terminal $B \in N$

- all strings of terminals that can be derived in one or more steps from B
- formally: $L_B = \{u \in T^* \mid B \to^+ u\}$

Derivation tree (or parse tree)

Trees and multi-step derivations

- CF grammars are used to define languages and implement parsers
- Parsers generate abstract syntax trees, but derivations are not trees!

Different multi-step derivations for the same string

- a derivation step is determined by
  1. the used production
  2. the non-terminal symbol which is replaced
- derived strings do not depend from choice 2

Derivation tree: intuitions and motivations

- non-terminal symbols are replaced simultaneously
- the structure of the analyzed sequence of tokens is made explicit

Examples of derivation trees (in ANTLR)

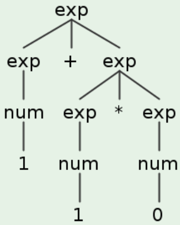
ANTLR Grammar (https://www.antlr.org/index.html)

```
grammar SimpleExp;

exp :  num | exp '*' exp | exp '+' exp | '(' exp ')' ;
num : '0' | '1' ;
```

Remarks: ANTLR (ANother Tool for Language Recognition) is a powerful parser generator which uses a slightly different syntax for BNF grammars

A derivation tree for "1+1*0"



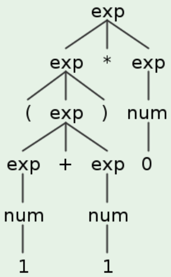
Examples of derivation trees (in ANTLR)

ANTLR Grammar

```
grammar SimpleExp;

exp :  num | exp '*' exp | exp '+' exp | '(' exp ')' ;
num : '0' | '1' ;
```

A derivation tree for "`(1+1)*0`"

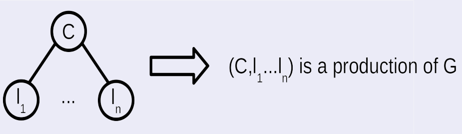


Derivation tree

Definition of derivation tree in G=(T,N,P)

A derivation tree in G for $u \in T^*$ starting from $B \in N$ is a tree with nodes labeled in $T \cup N$ such that:

- if a node is labeled by $C \in T$ then it is a leaf
- if a node is labeled by $C \in N$ and its n children by $l_1, \ldots, l_n$ from left to right, then $(C, l_1 \ldots l_n) \in P$; that is, $(C, l_1 \ldots l_n)$ is a production of G



- the root is labeled by B
- u is obtained by left-to-right concatenation of all terminal labels (necessarily belonging to leaf nodes)

Remark: a node labeled by a non-terminal symbol $C \in N$ can be a leaf if $(C, \epsilon) \in P$

Derivation tree and generated languages

Equivalent definitions of generated language

Language L_B generated from $G = (T, N, P)$ for non-terminal $B \in N$

1. all strings of terminals that can be derived in one or more steps from B
2. all strings of terminals for which there exists a derivation tree starting from B

Remark: definitions 1 and 2 are equivalent

Ambiguous grammars

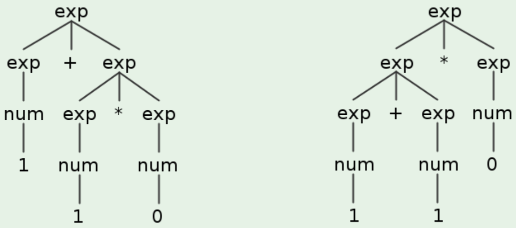
ANTLR Grammar

```
grammar SimpleExp;

exp :  num | exp '*' exp | exp '+' exp | '(' exp ')' ;
num : '0' | '1' ;
```

Remark: this grammar is ambiguous

Two different derivation trees for "1+1*0"



Ambiguous grammars

Definition

Grammar $G = (T, N, P)$ is ambiguous for $B \in N$ if there exist two different derivation trees for the same string starting from B

Reasons for ambiguity

Infix binary operators are intrinsically ambiguous

- Syntactic associativity of a single operator: does `1+1+1` mean `(1+1)+1` or `1+(1+1)`? Is addition left- or right-associative?
- Precedence of operators: does `1+1*1` mean `(1+1)*1` or `1+(1*1)`? Which has higher precedence between addition and multiplication?
- Syntactic associativity of operators with the same precedence: if addition and multiplication have the same precedence, does `1+1*1` mean `(1+1)*1` or `1+(1*1)`? Are addition and multiplication left- or right-associative?

A possible solution to ambiguity

We could change the syntax, but we are more familiar with infix operators . . .

Use prefix notation

```
Exp ::= Num | '+' Exp Exp | '*' Exp Exp
Num ::= '0' | '1'
```

- there is a unique derivation tree for `"+1*1 0"`
- note the difference between `"+1*1 0"` and `"*+1 1 0"`
- parentheses are no longer needed

Use postfix notation

```
Exp ::= Num | Exp Exp '+'  | Exp Exp '*'
Num ::= '0' | '1'
```

- there is a unique derivation tree for `"1 1 0*+"`
- note the difference between `"1 1 0*+"` and `"1 1+0*"`
- parentheses are no longer needed

A possible solution to ambiguity

Use functional notation

Similar to the prefix notation, but more verbose!

```
Exp ::= Num | 'add' '('Exp','Exp')' | 'mul' '('Exp','Exp')'
Num ::= '0' | '1'
```

- there is a unique derivation tree for `"add(1,mul(1,0))"`
- note the difference between `"add(1,mul(1,0))"` and `"mul(add(1,1),0)"`

Are there other (possibly better) solutions?

Observation

Although ambiguous, the infix notation is a more intuitive and practical solution!

Elimination of ambiguity in CF grammars for infix notation

- define syntactic associativity rules for binary operators
  - addition is left-associative: `"1+1+1"` means `"(1+1)+1"`
  - addition is right-associative: `"1+1+1"` means `"1+(1+1)"`
- define precedence rules for operators, use parentheses to override them
  - multiplication has higher precedence over addition: `"1+1*1"` means `"1+(1*1)"`
  - addition has higher precedence over multiplication: `"1*1+1"` means `"1*(1+1)"`
  - addition and multiplication have the same precedence and are left-associative: `"1*1+1*1"` means `"((1*1)+1)*1"`
  - addition and multiplication have the same precedence and are right-associative: `"1*1+1*1"` means `"1*(1+(1*1))"`

More on precedence and associativity rules

Operators with the same precedence

- binary operators can have the same precedence; in this case they must share the same associativity rule
  - addition and multiplication have the same precedence and are left-associative: `"1*1+1*1"` means `"((1*1)+1)*1"`
  - addition and multiplication have the same precedence and are right-associative: `"1*1+1*1"` means `"1*(1+(1*1))"`

Remark on associativity rules

Associativity rules resolve ambiguities between binary operators with the same precedence, that is, also when operators are the same

Operators with different arities (= number of operands)

Mixing together operators of different arities (typically 1, 2 and 3) makes elimination of ambiguity more complex!