Derivation tree (or parse tree)

Observation 1

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

Observation 2

- a derivation step is determined by
  1. the used production
  2. the specific non-terminal symbol which is replaced
- choice 2 does not influence the final string of terminals obtained from the derivation

Intuition

A derivation tree is a generalization of a multi-step derivation such that

- the derived string contains only terminal symbols
- non-terminal symbols are replaced “in parallel”
- the structure of the analyzed sequence of lexemes is made explicit

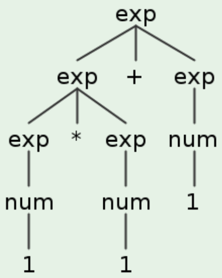
Examples of derivation trees (in ANTLR)

ANTLR Grammar

```
grammar SimpleExp;

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

Derivation tree for "1*1+1"



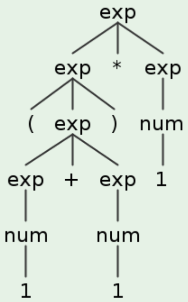
Examples of derivation trees (in ANTLR)

ANTLR Grammar

```
grammar SimpleExp;

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

Derivation tree for "(1+1)*1"

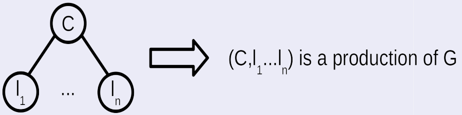


Derivation tree

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

Derivation tree for $u \in T^*$ starting from $B \in N$:

- if a node is labeled by C and its n children by $l_1, \ldots, l_n$, 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 of leaf nodes)

Derivation tree and generated languages

Equivalent definition of generated language

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

- all strings u of terminals such that there exists a derivation tree for u starting from B

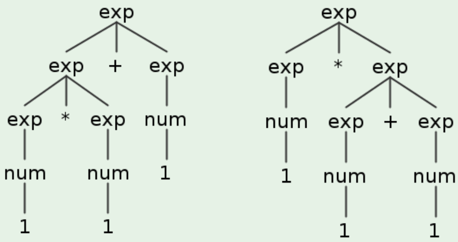
Ambiguous grammars

ANTLR Grammar

```
grammar SimpleExp;

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

Two derivation trees for "1*1+1"



Ambiguous grammars

Definition

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

Reasons for ambiguous grammars

Infix binary operators are intrinsically ambiguous

- Does `1+1+1` mean `(1+1)+1` or `1+(1+1)`? Is addition left- or right-associative?
- Does `1+1*1` mean `(1+1)*1` or `1+(1*1)`? Which has higher precedence between addition and multiplication?

A possible solution to ambiguity

Change the syntax!

Use prefix notation

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

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

Are there other (possibly better) solutions?

Observation

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

Elimination of ambiguity of infix notation

- define 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 between operators, use parentheses to override precedence rules
  - 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)"`

Are there other (possibly better) solutions?

Operators with the same precedence

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

Remark on associativity rules

Associativity rules resolve ambiguities between binary operators with the same precedence

Operators with different arities

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