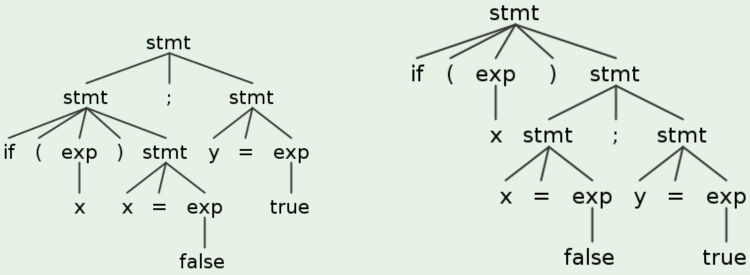
Ambiguous syntax for statements

Ambiguity: not only expressions ...

```
Stmt ::= ID '=' Exp | 'if' '(' Exp ')' Stmt | Stmt ';' Stmt
  | '{' Stmt '}'
Exp ::= ID | BOOL // ID and BOOL defined by regular expressions
```

Two derivation trees for "`if(x)x=false;y=true`"



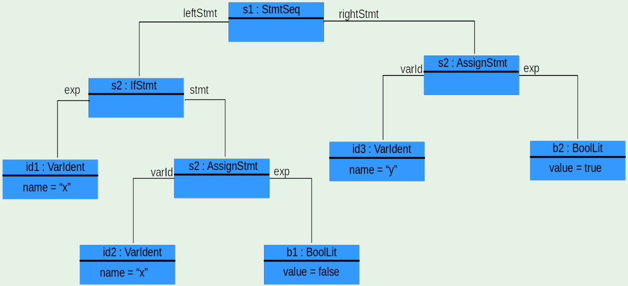
Ambiguous syntax for statements

Ambiguity: not only expressions ...

```
Stmt ::= ID '=' Exp | 'if' '(' Exp ')' Stmt | Stmt ';' Stmt
   | '{' Stmt '}'
Exp ::= ID | BOOL // ID and BOOL defined by regular expressions
```

Abstract syntax tree for `"if(x)x=false;y=true"`

`if` `(...)...` “has higher precedence” (standard case)



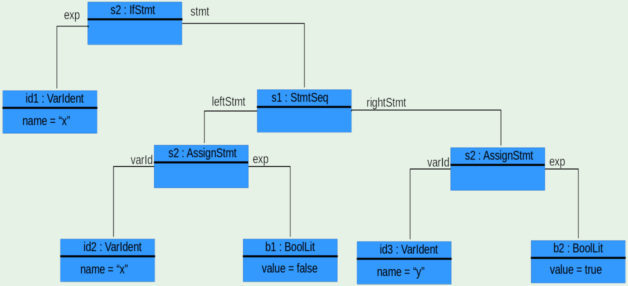
Ambiguous syntax for statements

Ambiguity: not only expressions ...

```
Stmt ::= ID '=' Exp | 'if' '(' Exp ')' Stmt | Stmt ';' Stmt
   | '{' Stmt '}'
Exp ::= ID | BOOL // ID and BOOL defined by regular expressions
```

Abstract syntax tree for `"if(x)x=false;y=true"`

`...;...` “has higher precedence” (non standard case)



Syntax and semantics

Remark

As happens for expressions, rules for syntax disambiguation have an impact on semantics!

Example in JavaScript (or other C-like languages)

```
x=false;
y=false;
if(x)x=false;y=true // indentation would help!
```

After execution `x` contains **`false`** and `y` **`true`**

```
x=false;
y=false;
if(x){x=false;y=true}
```

After execution both `x` and `y` contain **`false`**

How to build a parser from a grammar

Step 1: the grammar must be non-ambiguous

A non-ambiguous grammar

```
// * with higher precedence, both + and * are left-associtive
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Step 2: for each input token the parser has to choose the (unique) production that has to be used to find the correct parse tree

Problem

- tokens are read left-to-right by the parser
- simplest hypothesis: the parser knows only the next token. Technically: parsers with *one lookahead token*
- a parser with *one lookahead token* is not able to choose the right production for the grammar above!

How to build a parser from a grammar

A non-ambiguous grammar

```
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

A parser with *one lookahead token* cannot be built for the non-terminal `Exp` of the grammar above

Counter-example

If the first lookahead token is a number, then both productions for `Exp` could work.
Depending on the second lookahead token *t*:

- production (`Exp`,`Mul`) is used if *t* is either `'*'` or the end of the input stream
- production (`Exp`,`Exp '+' Mul`) is used if *t* is `'+'`

Toward a possible solution

Observations

- To build a parse tree, sooner or later production (`Exp`,`Mul`) must be used
- When productions of `Exp` are used consecutively, strings of the following shape are obtained:

  `Mul`

  or

  `Mul` followed by string `'+'Mul` repeated one or more times

Observations above suggest the following transformation

```
Exp ::= Mul | Mul AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Considerations on the new transformed grammar

Transformed grammar

```
Exp ::= Mul | Mul AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Considerations

- the grammar is equivalent to the previous one
- when building a parse tree we know that
  - an `Exp` node must always have a left child *c* labeled by `Mul`
  - after the parse tree with root *c* is built, the correct production is (`Exp`,`Mul AddSeq`) if the lookahead token is `'+'`, otherwise it is (`Exp`,`Mul`)
  - an `AddSeq` node must always have a left child c_1 labeled by `'+'`, followed by a child c_2 labeled by `Mul`
  - after the parse tree with root c_2 is built, the correct production is (`AddSeq`,`'+' Mul AddSeq`) if the lookahead token is `'+'`, otherwise it is (`AddSeq`,`'+' Mul`)

Full solution

A similar transformation can be used for non-terminal `Mul`

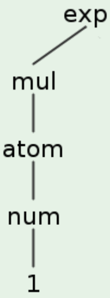
Final transformed grammar

```
Exp ::= Mul | Mul AddSeq
AddSeq ::= '+' Mul | '+' Mul AddSeq
Mul ::= Atom | Atom MulSeq
MulSeq ::= '*' Atom | '*' Atom MulSeq
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Example

Building the parse tree for `"1+1*1"`

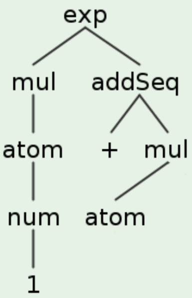
Lookahead token: number `1`



Example

Building the parse tree for `"1+1*1"`

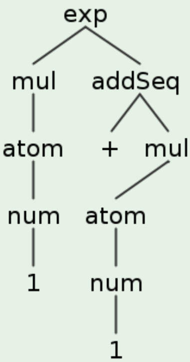
Lookahead token: addition operator



Example

Building the parse tree for `"1+1*1"`

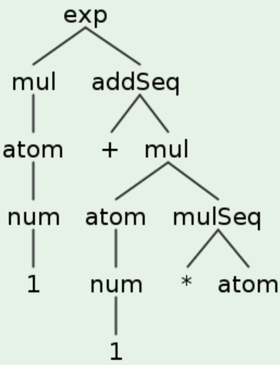
Lookahead token: number `1`



Example

Building the parse tree for "1+1*1"

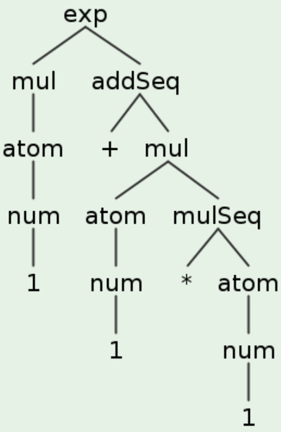
Lookahead token: multiplication operator



Example

Building the parse tree for `"1+1*1"`

Lookahead token: number `1`



EBNF grammars

- The BNF notation is extended with the usual post-fix operators of regular expressions: `*`, `+`, `?`
- The previous grammar can be transformed in a simpler grammar by using the EBNF notation

A simpler solution with an EBNF grammar

```
Exp ::= Mul ('+' Mul)*
Mul ::= Atom ('*' Atom)*
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Remarks

- note the difference between `'('`, `')'`, `'*'` and `(`, `)`, `*`
- `Mul ('+'Mul)*` is equivalent to `Mul | Mul AddSeq` if
  `AddSeq ::= '+'Mul | '+'Mul AddSeq`
- `Atom ('*'Atom)*` is equivalent to `Atom | Atom MulSeq` if
  `MulSeq ::= '*'Atom | '*'Atom MulSeq`

From a grammar to a recursive top-down parser

- For simplicity we consider only the problem of language recognition
- *Top-down* means that the parse tree is built from the root
- Tree generation will be considered in the Java labs

Assumptions on the tokenizer

The tokenizer defines the following procedures:

- `nextToken()`: the next lookahead token is read
- `tokenType()`: the type of the current lookahead token is returned
- `checkTokenType(type)`: the type of the current lookahead is checked, an exception is thrown if the check fails, oherwise the next token is read

Guidelines

- The code of the parser is directly driven by the grammar, including the recursive structure
- The parser consists of a main procedure together with a specific procedure for each non-terminal symbol of the grammar

From an EBNF grammar to a recursive parser

Pseudo-code driven by the previous EBNF grammar

```
parse(){ // main procedure
   nextToken() // reads the first lookahead token
   parseExp()
   checkTokenType(EOS) // checks end-of-stream if no other tokens are allowed
}
parseExp(){ // recognizes string generated from Exp
   parseMul()
   while(tokenType()==ADD){
      nextToken()
      parseMul()
   }
}
parseMul(){ // recognizes string generated from Mul
   parseAtom()
   while(tokenType()==MUL){
      nextToken()
      parseAtom()
   }
}
parseAtom(){ // recognizes string generated from Atom
  if(tokenType()==OPEN_PAR){
     nextToken()
     parseExp()
     checkTokenType(CLOSE_PAR)
  }
  else
     checkTokenType(NUM)
  nextToken()
}
```

EBNF grammar for right-associative operators

For right-associative operators the transformation is simpler

Non-ambiguous grammar

```
// * with higher precedence, both + and * are right-associtive
Exp ::= Mul | Mul '+' Exp
Mul  ::= Atom | Atom '*' Mul
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Equivalent EBNF grammar

```
Exp ::= Mul ('+' Exp)?
Mul  ::= Atom ('*' Mul)?
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

Pseudo code driven by the EBNF grammar: left as exercise