# Enum types

Special classes that define ordered groups of logically related constants

## Simple example

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL
}
public class SeasonTest {
    public static String feeling(Season s) {
        return switch (s) { // new Java 12 switch expressions
        case WINTER -> "Cold!";
        case SPRING -> "Flowers!";
        case SUMMER -> "Vacations!";
        case FALL -> "Rain!";
        };
    }
    public static void main(String[] args) {
        assert feeling(Season.SUMMER).equals("Vacations!");
        assert feeling(Season.SPRING).equals("Flowers!");
    }
}
```

# More details on enum types

## Basic rules

- each constant of the enum type corresponds to a **`public static final`** field (=public constant class field)
- an enum type has no objects other than those defined by its enum constants; it is not allowed to create new objects of an enum type
- it is safe to use == with enum constants

## Example

```
public enum Season {
    WINTER, SPRING, SUMMER, FALL;

    public static boolean niceSeason(Season s) {
        return s == SPRING || s == SUMMER;
    }
}
```

# Other rules and features of enum types

## Rules on inheritance/implementation

- Enum types cannot be extended
- Enum types can implement interfaces
- Each enum type `T` implicitly extends the predefined class `Enum<T>`

## Example

```
enum Season {
  WINTER, SPRING, SUMMER, FALL
}

public class SeasonTest {
  public static void main(String[] args) {
    Season[] seasons = Season.values(); // an array in definition order
    for (Season s : seasons) { // iterates over seasons
      // s.ordinal() returns the position of the element starting from 0
      assert seasons[s.ordinal()] == s;
      assert s.toString().equals(s.name()); // toString() redifinable, not name()
      assert Season.valueOf(s.name()) == s;
    }
  }
}
```

# Enum types and token types

## A practical use of enum types in tokenizers

```
public enum TokenType {
  // used internally by the tokenizer, should never been accessed by the parser
  SYMBOL, KEYWORD, SKIP,
  // non singleton categories
  IDENT, NUM,
  // end-of-file
  EOF,
  // symbols
  ASSIGN, MINUS, PLUS, TIMES, NOT, AND, EQ, STMT_SEP, PAIR_OP, OPEN_PAR,
       CLOSE_PAR, OPEN_BLOCK, CLOSE_BLOCK,
  // keywords
  PRINT, VAR, BOOL, IF, ELSE, FST, SND,
}
public class MyLangTokenizer implements Tokenizer {
  ...
  public TokenType next() throws TokenizerException {...}
  public TokenType tokenType() {...}
  ...
}
```

# Recap on parsers

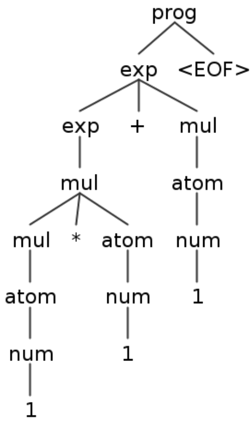
### Syntax analysis of a (programming) language

- recognizing valid sequences of tokens accordingly to the syntactic rules of the language
- building, in case of success, a parse (or derivation) tree, or an Abstract Syntax Tree (AST)
- parse tree = a proof that the sequence of tokens is grammatically correct
- AST = an abstraction of the parse tree where useless details of the concrete syntax are omitted
- trees make explicit the hierarchical structure of the syntax: they show how statements and expressions are built on top of simpler sub-statements and sub-expressions
- AST = input to the other steps of a programming language implementation: typechecking, interpretation/compilation

# Recap on parsers

## Parse tree

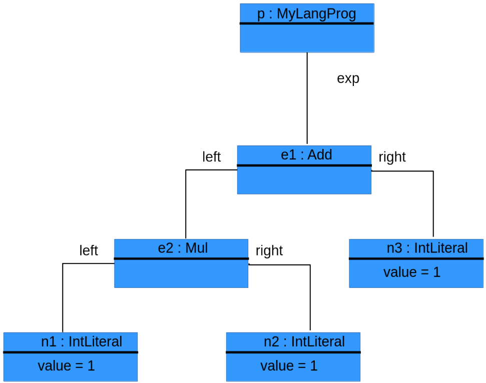
Parse tree for the string `"1*1+1"`



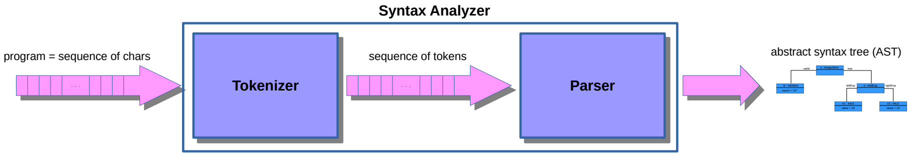
# Recap on parsers

## Abstract syntax tree

Abstract syntax tree for the string `"1*1+1"`



# Recap on parsers




## Parser = syntax analyzer

- input: sequence of tokens of a program, recognized by a tokenizer
- it checks that the sequence of tokens verifies the syntax rules
- the syntax rules are formally defined by a grammar
- output: a parse (or derivation) tree or an Abstract Syntax Tree (AST)
- it can be hand-written or automatically generated by an application (ANTLR, Bison, . . . )

# Recap on parsers

## Example 1 with C/Java/C++/C# syntax

Input string: `"x2 042=;"`

Recognized tokens:

- type `IDENT` with syntactic data: the name `"x2"`
- type `NUM` with semantic data: the value thirty-four
- type `ASSIGN` with no further data
- type `STMT_END` with no further data

## Result of the parser

failure, the sequence is not recognized and error messages are reported

# Recap on parsers

## Example 2 with C/Java/C++/C# syntax

Input string: `"x2=042+012;"`

Recognized tokens:

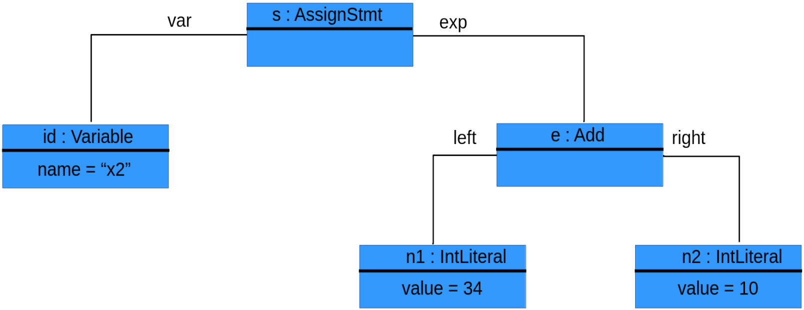
- type `IDENT` with syntactic data: the name `"x2"`
- type `ASSIGN` with no further data
- type `NUM` with semantic data: the value thirty-four
- type `PLUS` with no further data
- type `NUM` with semantic data: the value ten
- type `STMT_END` with no further data

## Result of the parser

success, the sequence is recognized and an AST is generated (see next slide)

# Recap on parsers

## Result of the parser: an AST



# Parsers and grammars

### Problem

How is it possible to implement a parser from a grammar?

- if the grammar has a certain shape, then the parser can be generated automatically
- two main approaches
  - top-down parser: checks if there is a parse tree starting from its root
  - bottom-up parser: checks if there is a parse tree starting from its leaves
- top-down parsers are simpler
  - they consist of several procedures, one for each non-terminal symbol of the grammar
  - the code of the procedures is driven by the productions
  - most of the procedures are mutually recursive

# Parsers and grammars

### Some general assumptions

- the parser reads the tokens from left to right by using a tokenizer
- it needs a fixed number of lookahead tokens to decide how to proceed
- parsers that use one lookahead token are the simplest ones

### Simplification

For simplicity, we only consider grammars for which it is possible to develop top-down parsers that use one lookahead token

# How to build a parser from a grammar

Important assumption: the grammar must be non-ambiguous, otherwise it is not possible to build a unique AST

A non-ambiguous grammar

A grammar where `'*'` has higher precedence than `'+'` and both operators are left associative

```
Prog ::= Exp EOF // the program should end with the EOF token
Exp ::= Mul | Exp '+' Mul
Mul ::= Atom | Mul '*' Atom
Atom ::= Num | '(' Exp ')'
Num ::= '0' | '1'
```

# How to build a parser from a grammar

Main guidelines (simplified version with no AST generation)

- define a parsing method for each non-terminal symbol
  Example: `parseExp()` parses all strings defined by `Exp`
- the implementation of each parsing method is driven by the productions of the corresponding non-terminal symbol
- terminals in productions must be correctly consumed by the tokenizer
  Example: `'+'` corresponds to the call `consume(PLUS)` which checks that the next lookahead token has type `PLUS` and asks the tokenizer to read the next lookahead token
- multiple productions correspond to branches that have to be selected
  Example: `parseExp()` should run the following code:
  - either call `parseMul()`
  - or call `parseExp()`, `consume(PLUS)` and `parseMul()`

# How to build a parser from a grammar

## Problems

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

`parseExp()` should run the following code:

- either call `parseMul()`
- or call `parseExp()`, `consume(PLUS)` and `parseMul()`

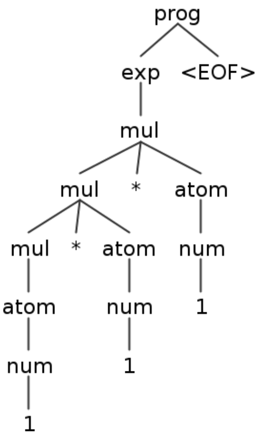
Problems:

- does one lookahead token allows selection of one of the two branches? no, for this grammar it is not possible to develop a parser using a fixed number of lookahead tokens
- the 2nd branch leads to non-terminating recursion

# How to build a parser from a grammar

## A problematic grammar

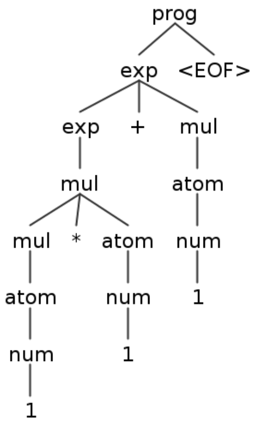
Example 1: parse tree for the string `"1*1*1"` (generated by ANTLR)



# How to build a parser from a grammar

## A problematic grammar

Example 2: parse tree for the string `"1*1+1"` (generated by ANTLR)



# How to build a parser from a grammar

## Solution

Merge the two productions into a single one by using the extended BNF notation

`Exp ::= Mul | Exp '+'Mul` is changed into `Exp ::= Mul ('+'Mul)*`

## Explanation

- production $(Exp, Mul)$ will eventually be used
- production $(Exp, Exp + Mul)$ can be used $n$ times, with $n \geq 0$, before production $(Exp, Mul)$ is used
- $n$ is the number of tokens of type PLUS read with the tokenizer

Examples:

- $n = 0$: $Exp \to Mul$
- $n = 1$: $Exp \to Exp + Mul \to Mul + Mul$
- $n = 2$: $Exp \to Exp + Mul \to Exp + Mul + Mul \to Mul + Mul + Mul$
- ...

# How to build a parser from a grammar

## Full solution

```
Prog ::= Exp EOF
Exp  ::= Mul ('+' Mul)*        // only one production
Mul  ::= Atom ('*' Atom)*      // only one production
Atom ::= Num | '(' Exp ')'     // one lookahead token needed here
Num ::= '0' | '1'
```

## Recap on the EBNF notation

- the BNF notation is extended with the usual post-fix operators of regular expressions: `*`, `+`, `?`
- parentheses can be used to force the precedence rules between the grammar operators
- Remark: `'('`, `')'`, `'+'` and `'*'` are terminal symbols, while `(` and `)` are EBNF parentheses and `+` and `*` are EBNF operators

# How to build a parser from a grammar

## How to deal with EBNF operators

`?` corresponds to an **`if`** statement

Example: `('+'Mul)?` can be translated into

```
if(lookahead has type PLUS){
  consume(PLUS);
  parseMul();
}
```

# How to build a parser from a grammar

## How to deal with EBNF operators

$\star$ corresponds to a **while** statement

Example: `('+'Mul)*` can be translated into

```
while(lookahead has type PLUS){
   consume(PLUS);
   parseMul();
}
```

# How to build a parser from a grammar

### How to deal with EBNF operators

+ corresponds to a **do**−**while** statement

Example: `('+'Mul)+` can be translated into

```
 do{
   consume(PLUS);
   parseMul();
}while(lookahead has type PLUS);
```

# How to build a parser from a grammar

## Java code

Parsing methods for `Prog` and `Exp`

```
public Prog parseProg() throws ParserException {
  nextToken(); // one lookahead token
  var prog = new MyLangProg(parseExp());
  match(EOF); // last token must have type EOF
  return prog;
}
private Exp parseExp() throws ParserException {
  var exp = parseMul();
  while (tokenizer.tokenType() == PLUS) {
    nextToken();
    exp = new Add(exp, parseMul());
  }
  return exp;
}
```

# How to build a parser from a grammar

### Some auxiliary methods used by the parser

Methods of the tokenizer:

- `next()`: the next lookahead token is read and its type returned
- `tokenType()`: the type of the current lookahead token is returned

Methods of the parser:

- `nextToken()`: calls `next()` on the tokenizer, throws an exception of type `ParserException` in case of error
- `match(type)`: checks that the next lookahead token has type `type`, throws an exception of type `ParserException` if not
- `consume(type)`: defined by `match(type); nextToken();`

# How to build a parser from a grammar

### Remarks

except for the main method `parseProg()`, all other parsing methods need to be synchronized with the tokenizer

- before calling `parseExp()`, `parseMul()`, `parseAtom()`, the current lookahead token must be the first token of the sequence to be parsed
- before exiting from `parseExp()`, `parseMul()`, `parseAtom()`, the current lookahead token must be the token that follows the parsed sequence

# How to build a parser from a grammar

## Right-associative operators

### Non-ambiguous grammar

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

### Equivalent EBNF grammar

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

Exercise: write the corresponding Java parser