Theory of Compilation

JLex, CUP tools

CS Department, Haifa University
Nov, 2010
By Bilal Saleh

Outlines

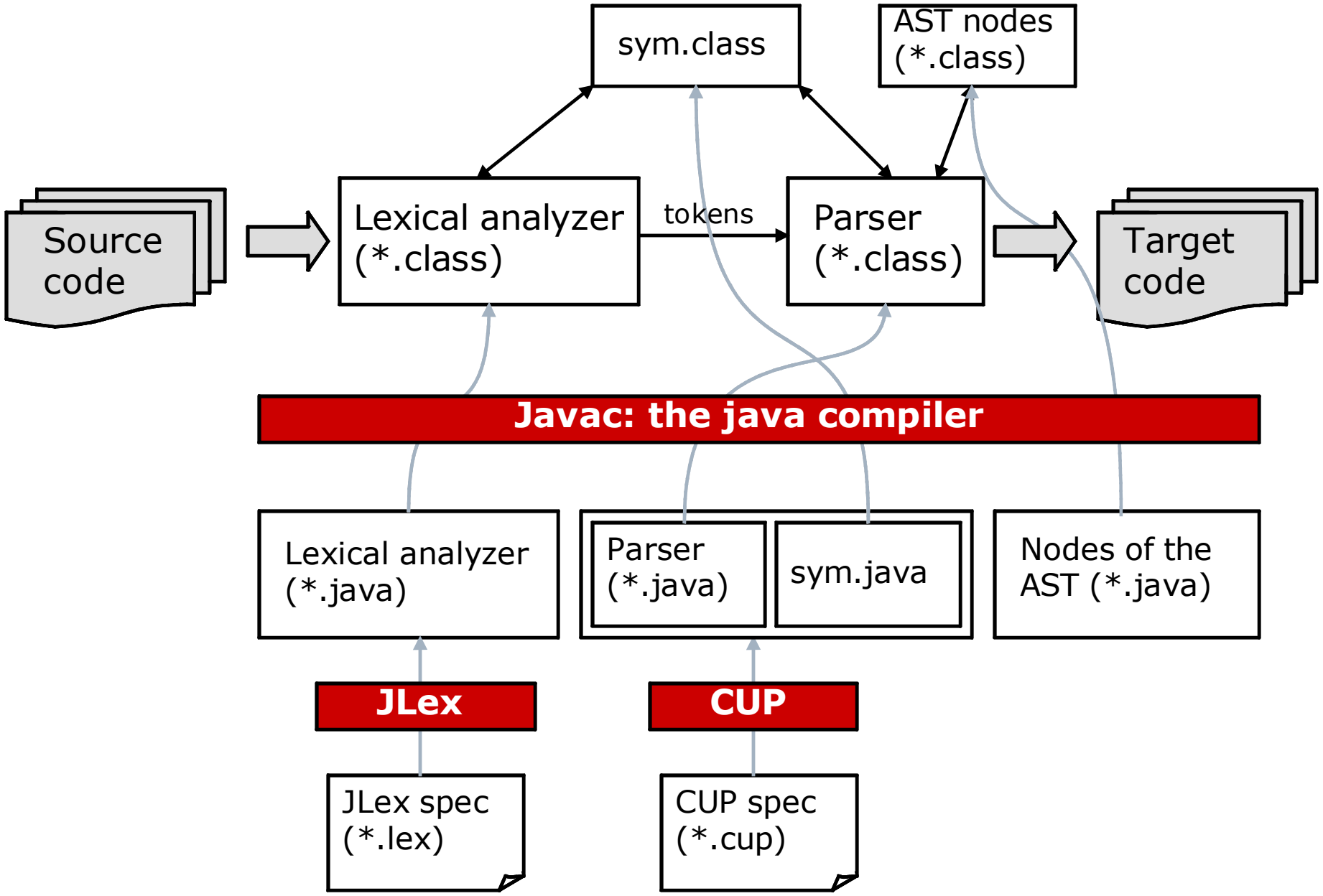
- JLex & CUP tutorials and Links
- JLex & CUP interoperability
- Structure of JLex specification
- JLex specification compilation
- Structure of CUP specification
- CUP specification compilation
- AST Nodes
- Integration in Eclipse workspace

Tutorials & links

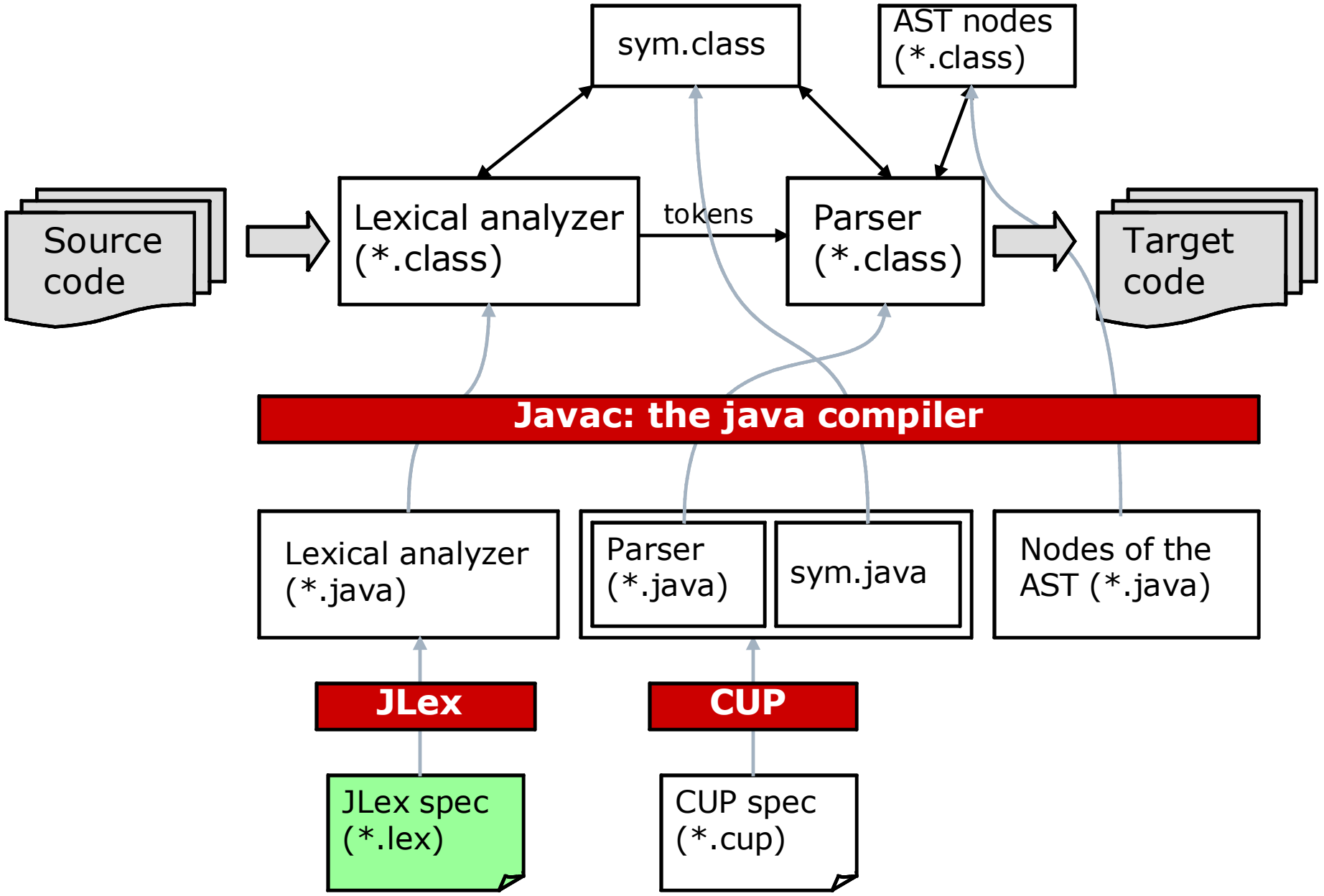
- JLex: lexical analyzer generator in Java. Online tutorial http://home.in.tum.de/~kleing/jflex/manual.html
- CUP: parser generator in Java. Online tutorial http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html
- A ready-to-use JLex_CUP workspace: http://cs.haifa.ac.il/courses/compilers/BILAL/JLex_CUP.zip

JLex & CUP interoperability

Interoperability



JLex specification


sym.class
AST nodes (*.class)
Source code
Lexical analyzer (*.class)
tokens
Parser (*.class)
Target code
Javac: the java compiler
Lexical analyzer (*.java)
Parser (*.java)
sym.java
Nodes of the AST (*.java)
JLex
CUP
JLex spec (*.lex)
CUP spec (*.cup)

JLex specification

- Consists of 3 sections
  1. User code:
     - Package name
     - Import packages of java
  2. Options & declarations:
     - Specifying directives such as %cup, %byacc
     - The code included in %{…%} will be automatically injected inside the generated lexer
     - Defining macros that might be used in the Lexical rules section
  3. Lexical rules
     - Contains the regular expressions and actions to be performed
- Sections are separated by %%

JLex specification example

```
package miny_pascal;
import java_cup.runtime.Symbol;
import java.io.FileInputStream;
import java.io.InputStream;

%%

%cup
%line
%{
        private int countLines(String str){
                …
        }
%}
DIGIT  =  [0-9]
LETTER =  [a-zA-Z_]
IDE    =  {LETTER}({LETTER}|{DIGIT})*
INT    =  {DIGIT}+

%%

"IF"        { return new Symbol(sym.IF); }
"+"         { return new Symbol(sym.ADD); }
{INT}       { return new Symbol(sym.INTCONST, new Integer(Integer.parseInt(yytext()))); }
{IDE}       { return new Symbol(sym.IDE, yytext()); }
[\n]        { ++yyline; }
[\r\t\f\ ]+                    {  }
```

Some notes

- JLex designates tokens with **longest match**, for example

  input: abc
  rule: [a-z]+

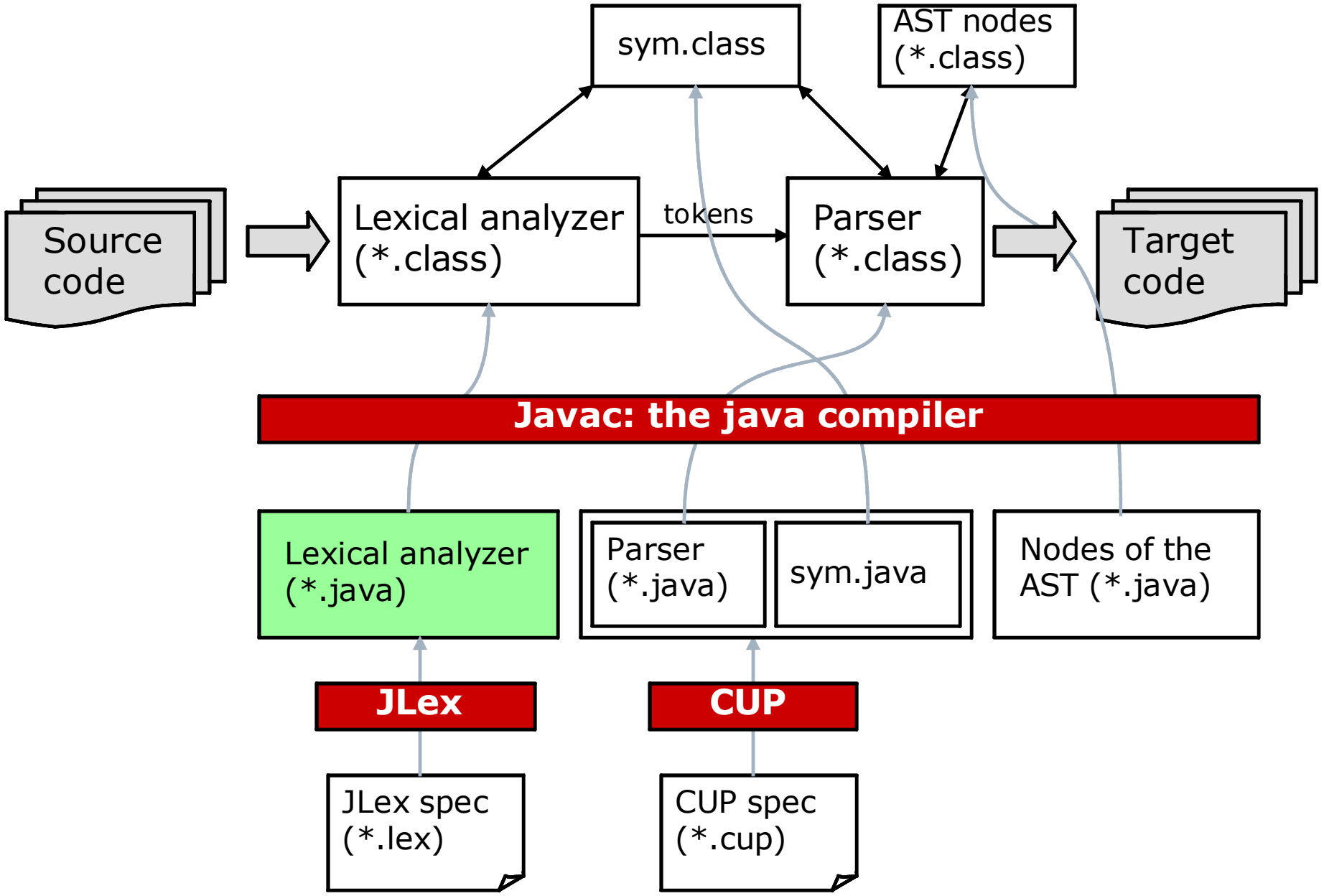
  result will be abc (not a, ab)

- JLex uses the **first applicable rule**, for example

  input: FOR
  rule1: “FOR”
  rule2: [a-zA-Z]+

  JLex will choose rule1

Compiling JLex specification


sym.class
AST nodes
(*.class)
Source
code
Lexical analyzer
(*.class)
tokens
Parser
(*.class)
Target
code
Javac: the java compiler
Lexical analyzer
(*.java)
Parser
(*.java)
sym.java
Nodes of the
AST (*.java)
JLex
CUP
JLex spec
(*.lex)
CUP spec
(*.cup)

Compiling JLex specification

1. Download JLex_CUP.zip package from the course website.
2. Extract it somewhere in your hard drive (e.g. `C:\tmp`). Tree view looks like this:

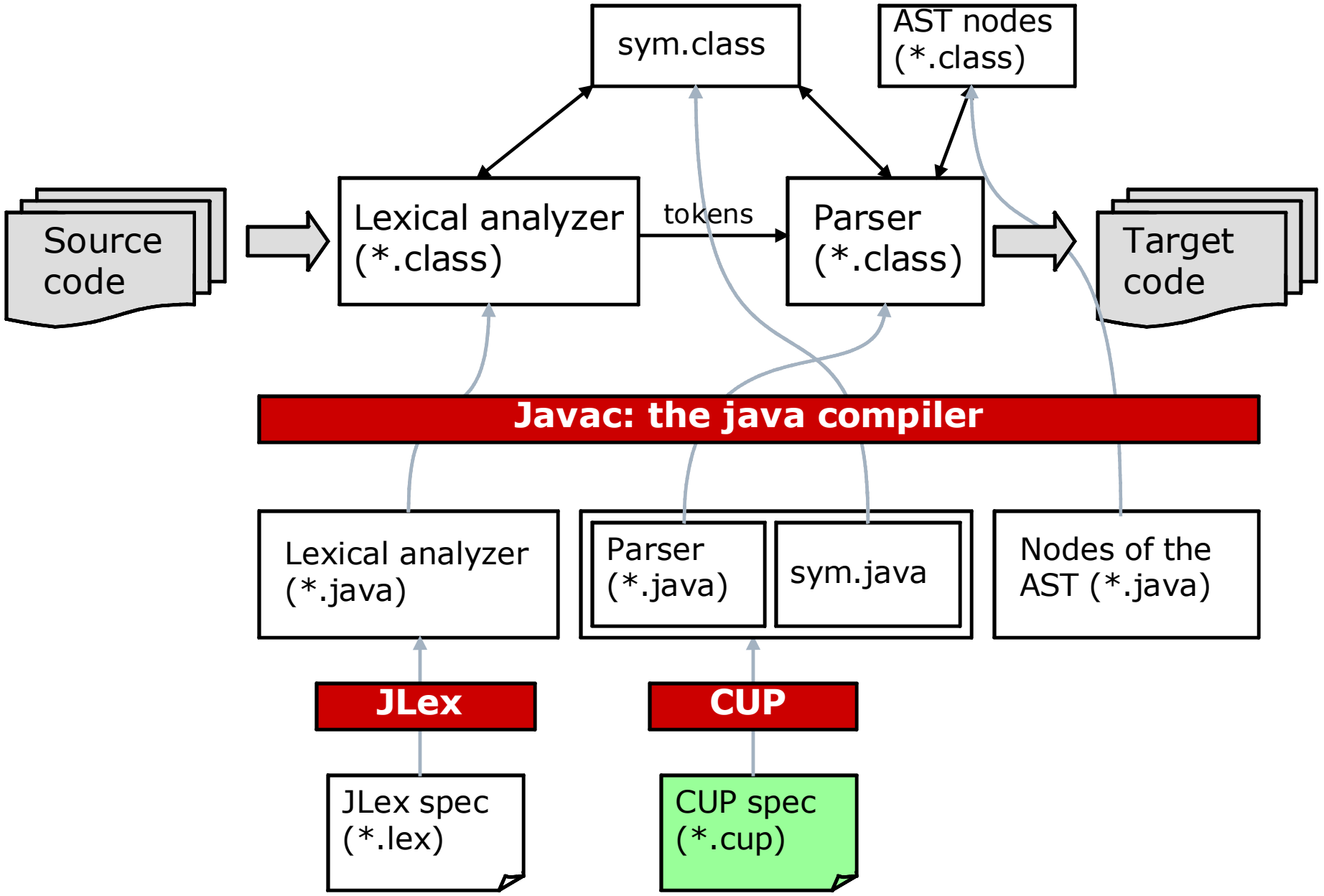
```
C:\
        +--tmp\
                +--JLex\
                        +--Main.java
                +--java_cup\
                        +--Main.java
                        +--runtime\
                +--Yylex.lex
                +--Parser.cup
```

3. Modify your `Yylex.lex` specification as you desire.
4. Compile `Yylex.lex`:

```
C:\tmp>java JLex.Main Yylex.lex
```

5. If compiled successfully, it will output file `Yylex.lex.java` under `C:\tmp`
6. Rename `Yylex.lex.java` to `Yylex.java`

CUP specification


sym.class
AST nodes
(*.class)
Source
code
Lexical analyzer
(*.class)
tokens
Parser
(*.class)
Target
code
Javac: the java compiler
Lexical analyzer
(*.java)
Parser
(*.java)
sym.java
Nodes of the
AST (*.java)
JLex
CUP
JLex spec
(*.lex)
CUP spec
(*.cup)

CUP specification

- Package & import declarations
- User code components (linking with the lexer)
- Symbols (terminal & non terminal) lists
- Precedence declaration
- Grammar (context-free)

Package & import declarations

```
package miny_pascal;

import java_cup.runtime.*;
import java.io.FileInputStream;
import java.io.InputStream;
```

User code components

```
/* Preliminaries to set up and use the scanner.  */
parser code
{:
        public Node root = null;
        public static parser getParser(String pPath) throws Exception {
                InputStream is = null;
                is = new FileInputStream(pPath);
                return new parser(new Yylex(is));
        }
        public Node getTree() throws Exception {
                if (root == null) {
                        this.parse();
                }
                return root;
        }
        public static void main(String args[]) throws Exception {
                new parser(new Yylex(System.in)).parse();
        }
:}
```

Terminals & non terminals

```
/* Terminals (tokens returned by the scanner). */
terminal PROGRAM, BEGIN, END, DECLARE, PROCEDURE, FUNCTION, …
terminal BOOLEAN, ARRAY, OF, ASSIGN, LC, RC, IF, THEN, ELSE, …
terminal READ, WRITE, TRUE, FALSE, ADD, MIN, MUL, DIV, GOTO;
terminal MOD, LES, LEQ, EQU, NEQ, GRE, GEQ, AND, OR;
terminal NOT, CASE, FOR, FIN, IDENTICAL, FROM, BY, TO, NEW;
terminal UMIN, COLON, SEMI, LPAR, RPAR, LPAR_SQ, RPAR_SQ, DOT, COMMA, PTR;

/* Terminals with attached values */
terminal Integer INTCONST;
terminal String IDE;
terminal Double REALCONST;
terminal String STRING;

/* Non terminals */
non terminal Node var, assign, program, stat_seq, loop_stat, case_stat, …
non terminal Node expr, atom, block, stat, nonlable_stat, cond_stat, case, …
non terminal Node var_decl, type, simple_type, array_type, record_type, …
non terminal Node record_list, dim, dim_list, proc_decl, formal_list, …
non terminal Node inout_stat, new_stat;
```

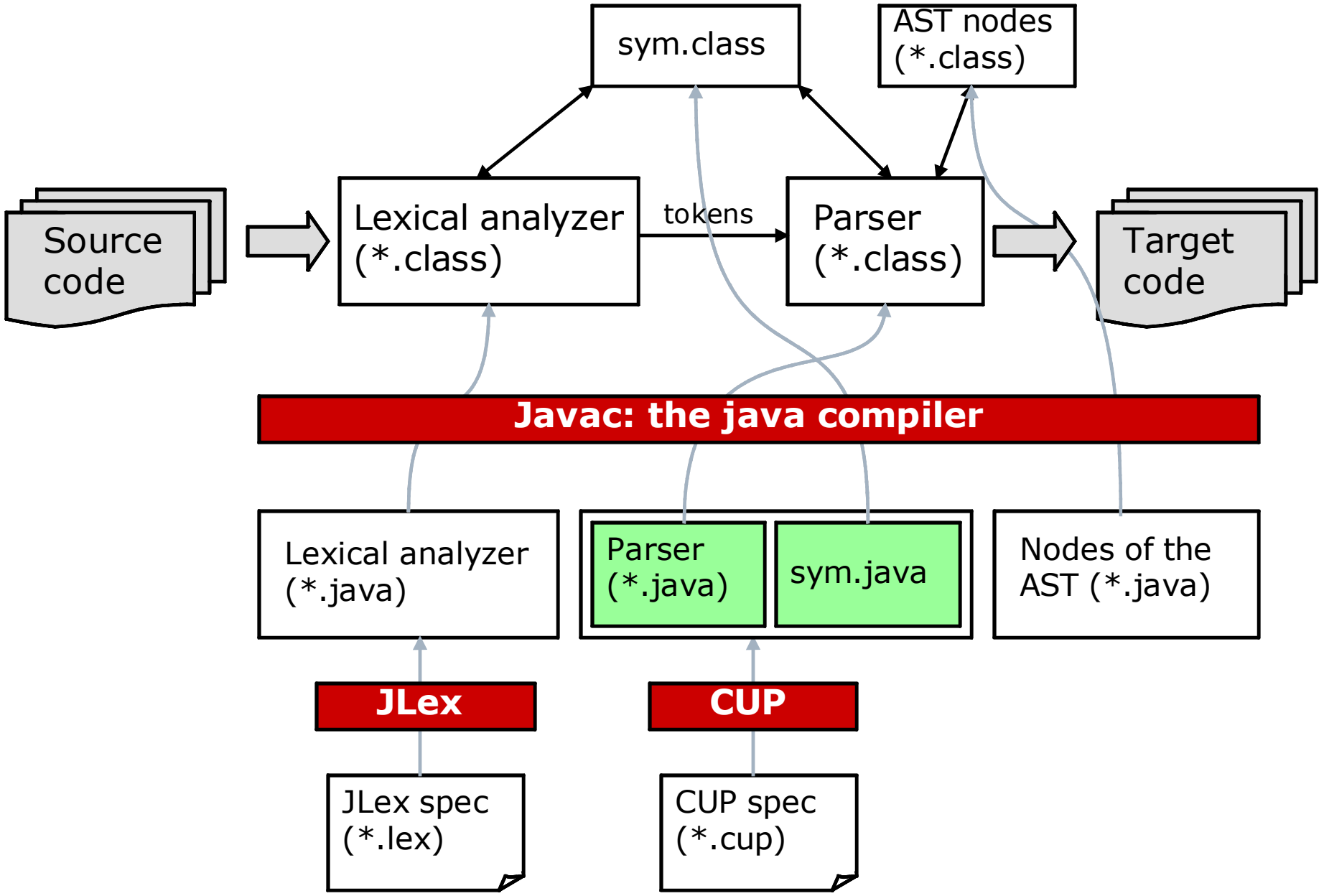
Precedence declaration

```
/* Precedence List */
precedence nonassoc LES, LEQ, EQU, NEQ, GRE, GEQ;
precedence left ADD, MIN, OR;
precedence left MUL, DIV, AND, MOD;
precedence left UMIN;
precedence right NOT;
precedence right DOT;
precedence right PTR;;
```

Grammar (context-free)

```
/* Grammar */
start with program;

program ::= PROGRAM IDE:n block:b    {: RESULT = new Program(b,n);
                                        parser.root=RESULT; :} ;

block ::= LC stat_seq:s RC                  {: RESULT = new Block(s); :}
        | decl_list:d LC stat_seq:s RC    {: RESULT = new Block(d,s); :} ;

decl_list ::= decl:d                {: RESULT = new DeclarationList(d); :}
            | decl:d decl_list:dl  {: RESULT = new DeclarationList(dl,d); :} ;

decl ::= var_decl:vd                        {: RESULT = vd; :}
               | proc_decl:pd                        {: RESULT = pd; :}
               | func_decl:fd                        {: RESULT = fd; :} ;

…


assign ::= var:v ASSIGN expr:e      {: RESULT = new Assign(e,v); :} ;

cond_stat ::= IF expr:e THEN stat_seq:ss FI   {: RESULT = new
                                        ConditionalStatement(e,ss); :}
```

Compiling CUP specification


sym.class
AST nodes (*.class)
Source code
Lexical analyzer (*.class)
tokens
Parser (*.class)
Target code
Javac: the java compiler
Lexical analyzer (*.java)
Parser (*.java)
sym.java
Nodes of the AST (*.java)
JLex
CUP
JLex spec (*.lex)
CUP spec (*.cup)

Compiling CUP specification

1. Download JLex_CUP.zip package from the course website.
2. Extract it somewhere in your hard drive (e.g. `C:\tmp`). Tree view looks like this:

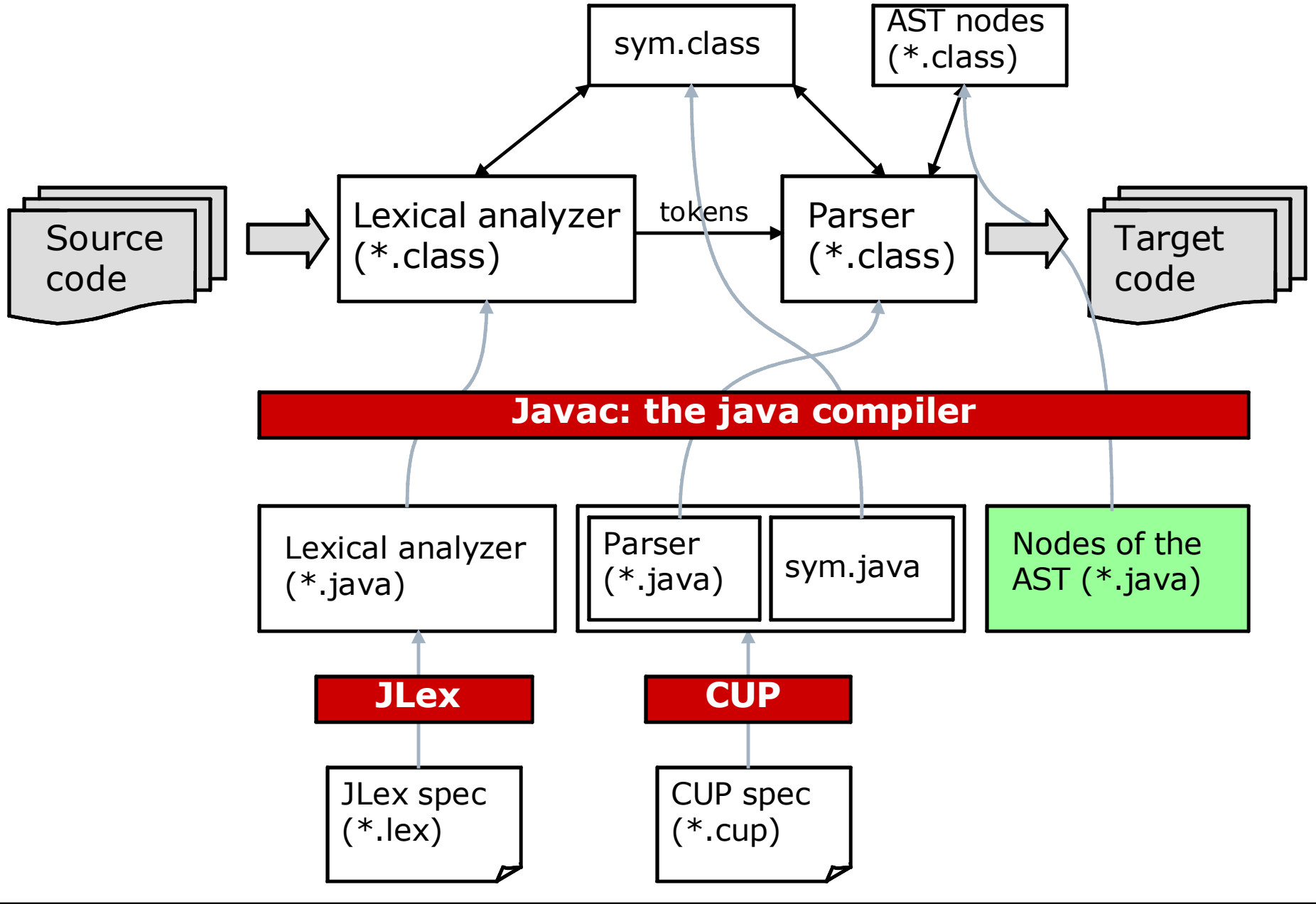
```
C:\
        +--tmp\
                +--JLex\
                        +--Main.java
                +--java_cup\
                        +--Main.java
                        +--runtime\
                +--Yylex.lex
                +--Parser.cup
```

3. Modify your `Parser.cup` specification as you desire.
4. Compile `Parser.cup`:

```
C:\tmp>java java_cup.Main -expect 1 Parser.cup
```

5. If compiled successfully, it will output file `parser.java & sym.java` under `C:\tmp`

AST Nodes

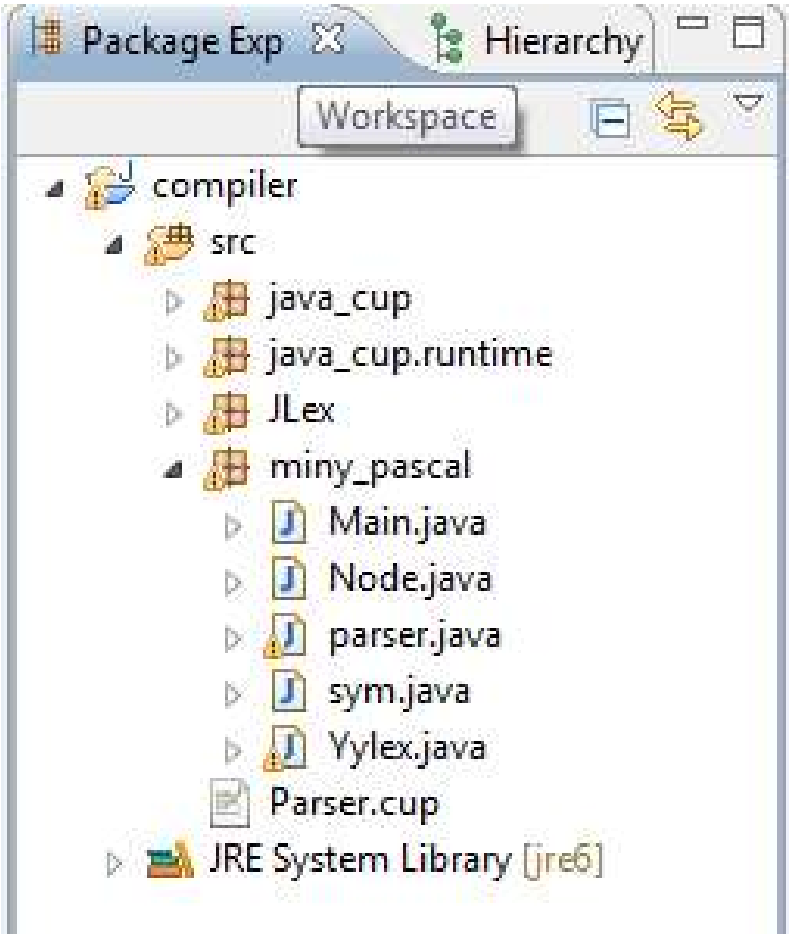

sym.class
AST nodes
(*.class)
Source
code
Lexical analyzer
(*.class)
tokens
Parser
(*.class)
Target
code
Javac: the java compiler
Lexical analyzer
(*.java)
Parser
(*.java)
sym.java
Nodes of the
AST (*.java)
JLex
CUP
JLex spec
(*.lex)
CUP spec
(*.cup)

Main.java | Node.java | Parser.cup | parser.java | Yylex.lex | sym.java

```
package miny_pascal;

import java.io.PrintWriter;

 * @author Bilal Saleh
public abstract class Node implements Cloneable {

    public Node() {

    }

    public abstract void print(PrintWriter pw);

    public abstract void code(PrintWriter pw);

    public abstract void codeL(PrintWriter pw);

    public abstract void codeR(PrintWriter pw);

}

class Expr extends Node {

    private int operator;
    private boolean unary = false;
    private Node atom;
    private Node left = null;
    private Node right = null;

    // Unary operations
    public Expr(int pOperator, Node pAtom) {
        operator = pOperator;
        atom = pAtom;
        unary = true;
```

Integration in Eclipse Project

Eclipse workspace



- In HW2 & HW3 you have to implement the Code-Generation part inside Node.java

- If you need to modify the parser/lexer, follow the instructions from previous slides and then replace Yylex.java, sym.java, parser.java by your own classes.