# Compilation

0368-3133

Tutorial 1:

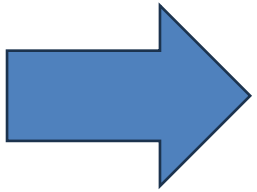
Lexical Analysis

# What is a Compiler?

- Translation of code (text) to executable code (machine code)

**source code**

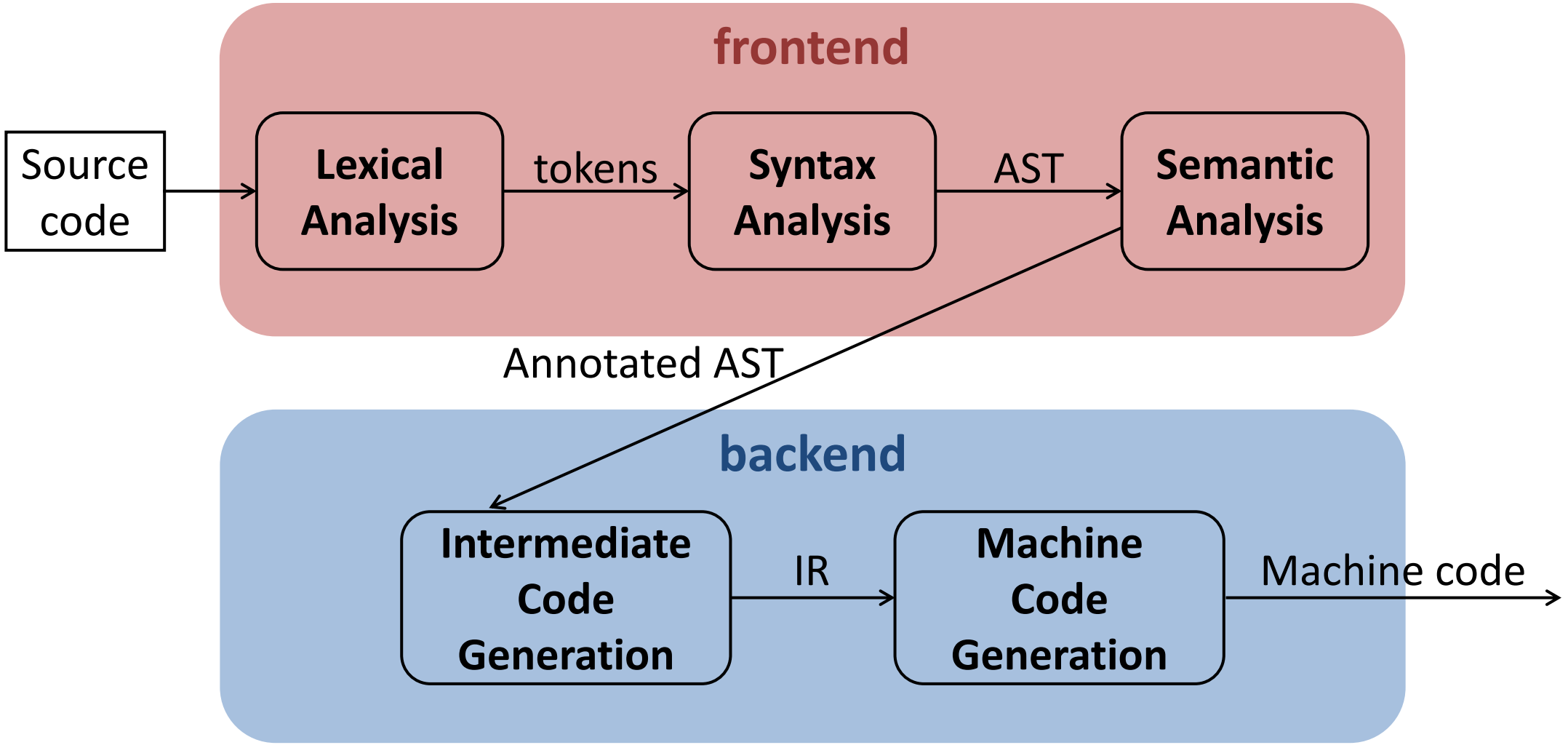
```
int foo(int x, int y) {
    return x + y;
}
```



**machine code**

```
push   %rbp
mov    %rsp,%rbp
mov    %edi,-0x4(%rbp)
mov    %esi,-0x8(%rbp)
mov    -0x4(%rbp),%edx
mov    -0x8(%rbp),%eax
add    %edx,%eax
pop    %rbp
retq
```

# Compilation Steps



# Administration

- Teaching assistant: Noa Schiller
  - noaschiller@mail.tau.ac.il
  - Office hours: Thursday's 15:00 – 16:00 (before class)
    Check Point 341

- Grade:
  - Exam 60%
  - **Project 40%**

# Course Project

- **Goal:** Build a compiler for an OOP Programming Language
  - Simplified version of known programming languages

| Exe. 1<br>Lexical analysis | Exe. 2<br>Syntax analysis | Exe. 3<br>Semantic analysis | Exe. 4<br>Intermediate code | Exe. 5<br>Code generation |
|---|---|---|---|---|
| 3% | 4% | 8% | 10% | 15% |

- Implemented in Java
- Work in groups of 3 students
- For questions, use the forum in the course Moodle

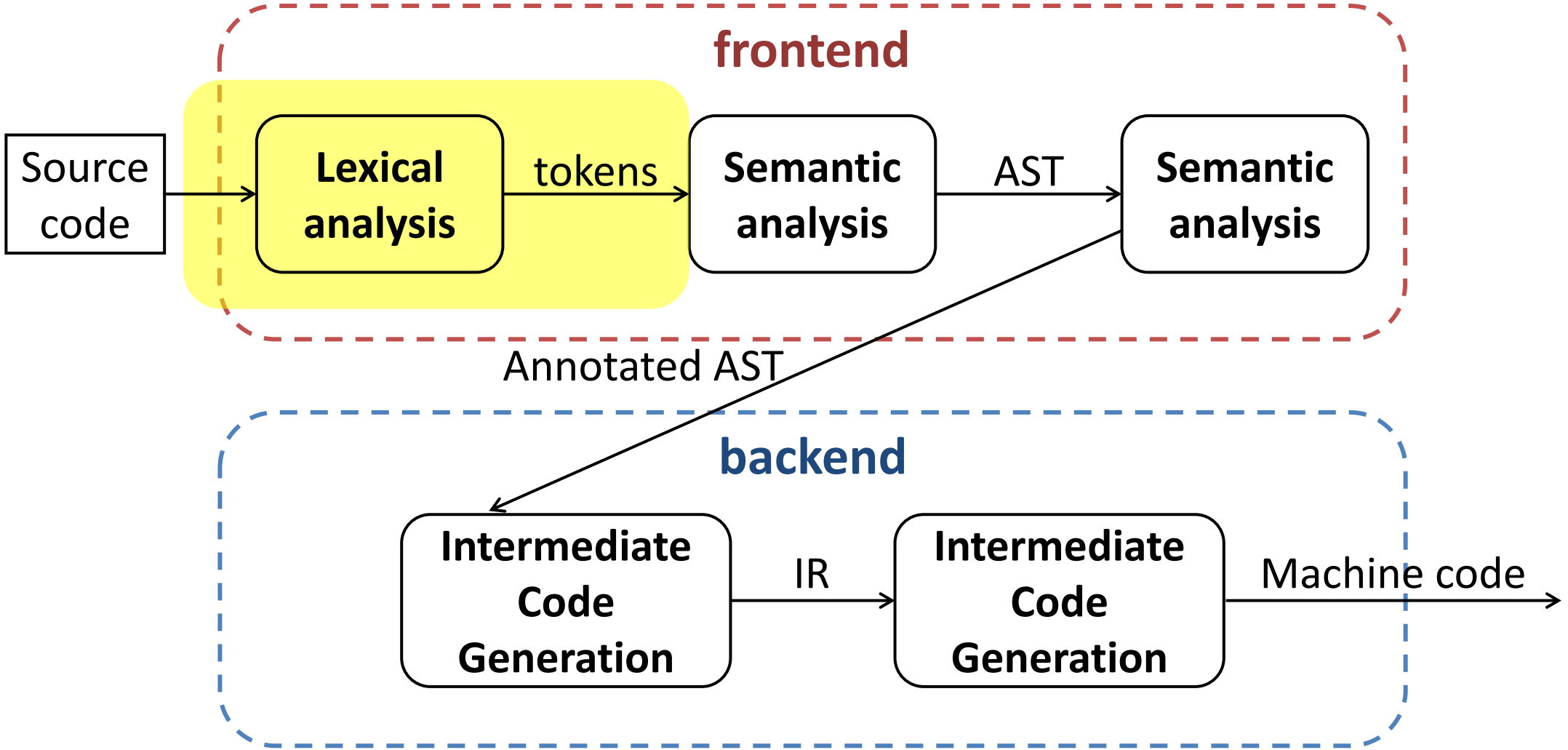
# Submission Guidelines

- With each exercise you get a starter code
  - Treat this as a minimal working example
  - You can change any part of it not only add

- Running environment is the school server (nova)
  - Before submitting you must run a self check script

- Full guidelines including resubmission and grace days in the Moodle

# This Tutorial



# The Lexical Analyzer Algorithm

- Input: source code (text)
- Output: List of valid tokens

1. Set the current position to the beginning of the input
2. Scan loop:
   - If reached end of input, **done**
   - Else, try to match with one of the defined tokens
     - If there is no match, **fail**
     - Otherwise, add token to output, increment the current position and repeat step 2

# Example Tokens

| **Token** | **Examples** |
|---|---|
| Constants | 12, 0x1234, 1.7, 2e+8 |
| Strings | “compilation” |
| Identifiers | var, tmp1 |
| Reserved Keywords | if, else, while, int, char |
| Parentheses | (,),{,} |
| Binary Operators | +,-,*,/ |
| Unary Operators | -,* |

# Lexical Analyzer: Example

```
void f() {
   x = 1;
}
```

**Symbols:**

| void | ws | f | ( | ) | ws | { | ws | x | ws | = | ws | 1 | ; | ws | } |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Lexical Analyzer: Example

```
void f() {
   x = 1;
}
```

**Tokens:** <kind, value>

| VOID | ID | LPAREN | RPAREN | LBRACE | ID | EQ | INT | SEMICOL | RBRACE |
|---|---|---|---|---|---|---|---|---|---|
| | f | | | | x | | 1 | | |

# Lexical Analyzer: Filtering

- The lexical analyzer ignores:
  - Whitespace, tab, newline
  - Comments

- Not needed for later stages

# Defining Tokens with Regular Expressions

- Defining tokes for keywords is easy
- Identifiers, numerical constants, strings, etc., are large sets of values
- Every token can be represented using a regular expression which describes its pattern

- Examples:
  - Identifiers: [a-zA-Z][a-zA-Z0-9]*
  - Hex-decimal constants: 0[xX][0-9a-fA-F]+

# Ambiguity

- Assume we have the following token definition:

  int  “int”

  id  [a-z]+

- “abc” can match “a”, “ab” or “abc”

# Ambiguity

- Assume we have the following token definition:

| | |
|---|---|
| int | “int” |
| id | [a-z]+ |

- “intt” can match “intt” or “int” and then “t”

# Ambiguity

- Assume we have the following token definition:

  int  “int”

  id  [a-z]+

- “int” can match the token int and the token id

# Solving Conflicts

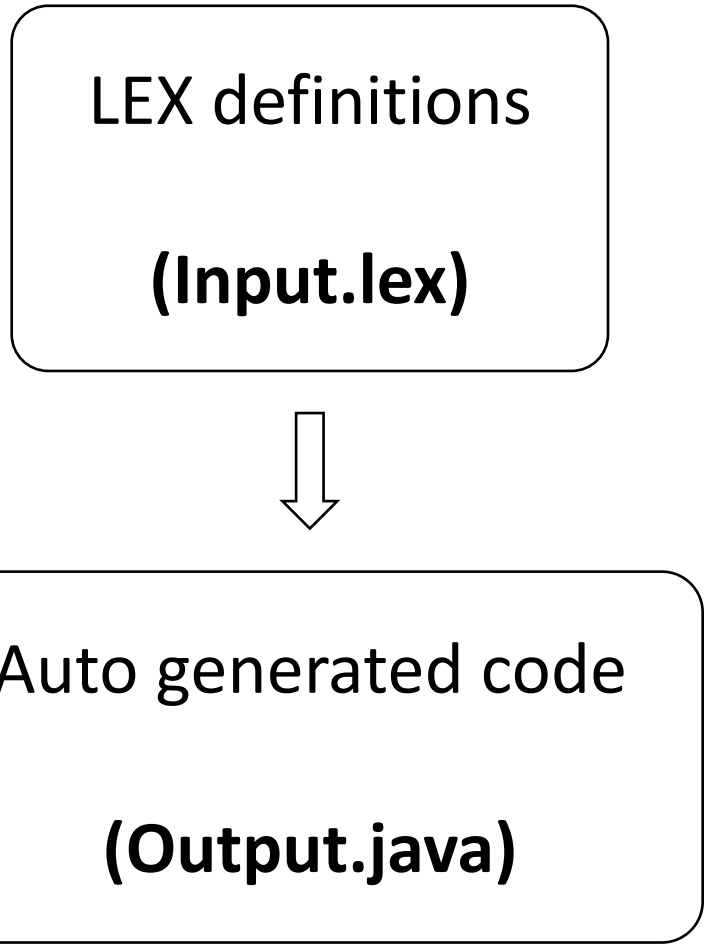
- The analyzer is greedy and always chooses the longest matching symbol
- Break ties for symbols that match more than 1 pattern by order of definitions

- “abc” matches id(“abc”)
- “intt” matches id(“intt”)
- “int” matches int

# JFlex

- **J**ava **F**ast **Lex**ical Analyzer
- Inspired by the original **flex** project (written in C)
- Accepts an input file with tokens definitions
- Generates Java code with a scanning API

- This scanning API reads the input and returns:
  - The type of the matched token
  - Or an error if no rule matches



# LEX Format

**user code**

**declarations**

**rules**

```
%{
// java code here
…
%}

<VAR> = <REGEX>
…

%%

<STATE> {
{REGEX} {
// java code here
}
…
}
```

** The first part of the file is omitted. It contains options and code inserted before the generated scanner class declaration, which you don't need to modify.*

# LEX Format

**user code**

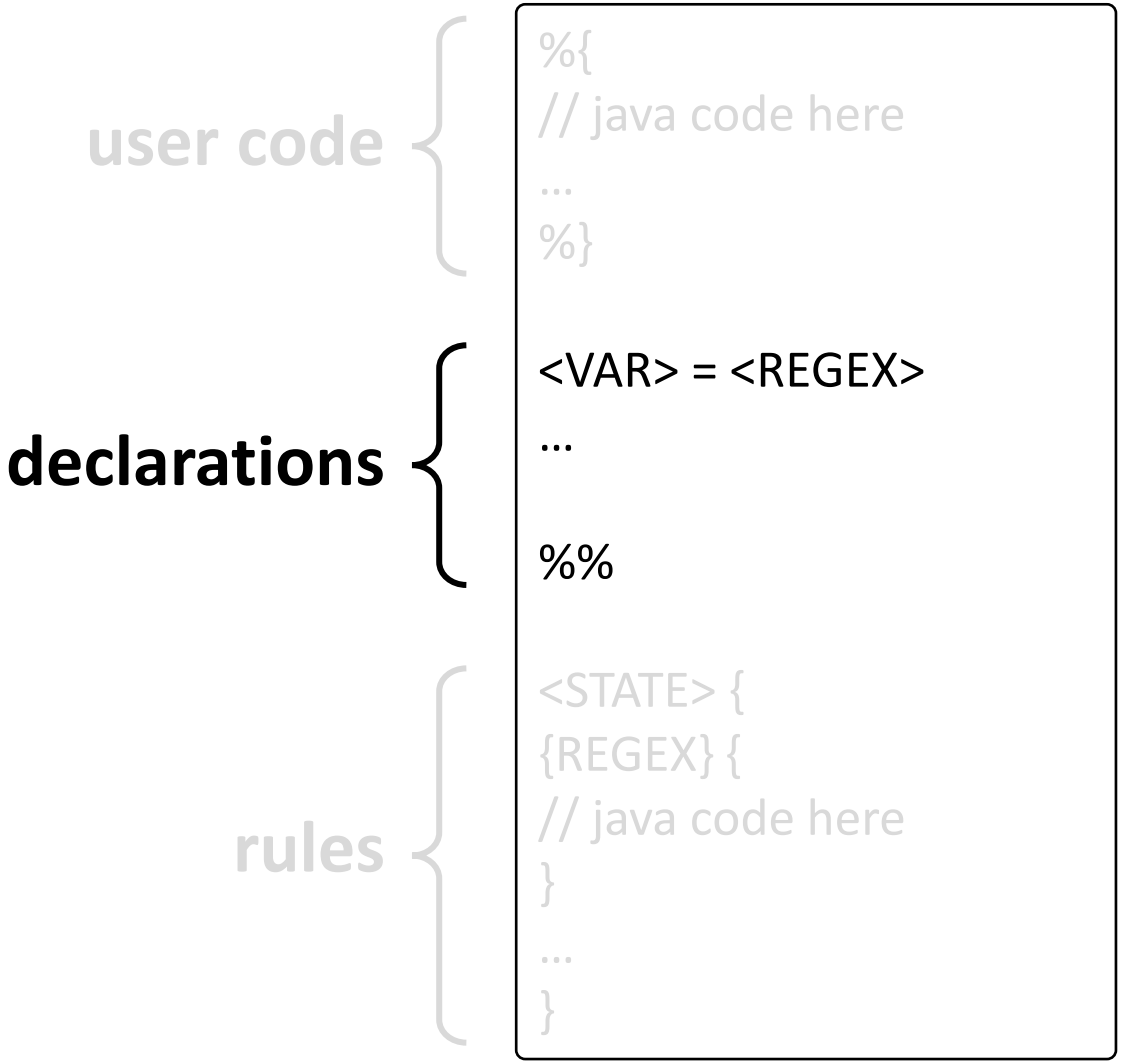
**declarations**

**rules**

```
%{
// java code here
…
%}

<VAR> = <REGEX>
…

%%

<STATE> {
{REGEX} {
// java code here
}
…
}
```

- Regular Java code
- Pasted to the generated file into the scanner class

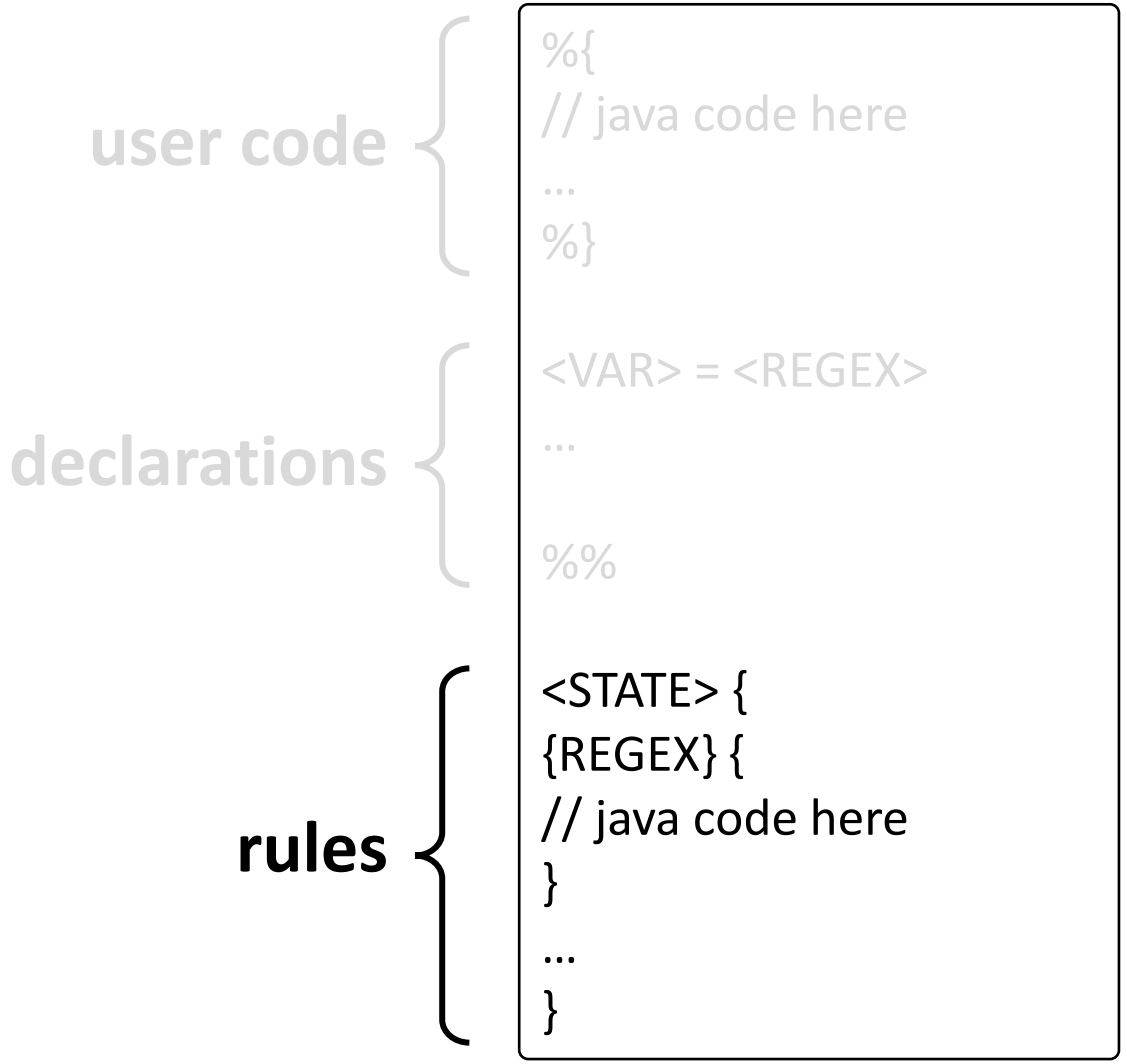
# LEX Format




- Regular Java code
- Pasted to the generated file

- Macro definitions
- Lexical states declarations

# LEX Format




- Regular Java code
- Pasted to the generated file

- Macro definitions
- Lexical states declarations

- If the following holds:
  - Current lexical state is <STATE>
  - <REGEX> is matched
- Then execute the action code

# LEX Generated Class Members

| Name | Type | Description |
|---|---|---|
| yyline | int | Contains the current line of input (starting with 0) |
| yycolumn | int | contains the current column of the current line (starting with 0) |
| yytext() | String | Returns the last matched symbol |

# Example 1

We want 2 kind of tokens:

- a
- $b^*$
- Everything else is rejected…

# Example 1: LEX Definitions

```
%{
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
public int getLine() { return yyline + 1; }
public int getTokenStartPosition() { return yycolumn + 1; }
%}
A = a
B_STAR = b*
%% // separator…
<YYINITIAL> {
{A}                 { return symbol(TokenNames.A); }
{B_STAR}            { return symbol(TokenNames.B_STAR); }
<<EOF>>             { return symbol(TokenNames.EOF);}
}
```

# Example 1: LEX Definitions

```
%{
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
public int getLine() { return yyline + 1; }
public int getTokenStartPosition() { return yycolumn + 1; }
%}
A = a
B_STAR = b*
%% // separator…
<YYINITIAL> {
{A}                 { return symbol(TokenNames.A); }
{B_STAR}            { return symbol(TokenNames.B_STAR); }
<<EOF>>             { return symbol(TokenNames.EOF);}
}
```

# Example 1: LEX Definitions

```
%{
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
public int getLine() { return yyline + 1; }
public int getTokenStartPosition() { return yycolumn + 1; }
%}
```

**A = a**

**B_STAR = b***

```
%% // separator…
<YYINITIAL> {
{A}                 { return symbol(TokenNames.A); }
{B_STAR}            { return symbol(TokenNames.B_STAR); }
<<EOF>>             { return symbol(TokenNames.EOF);}
}
```

# Example 1: LEX Definitions

```
%{
private Symbol symbol(int type) { return new Symbol(type, yyline, yycolumn); }
public int getLine() { return yyline + 1; }
public int getTokenStartPosition() { return yycolumn + 1; }
%}
A = a
B_STAR = b*
%% // separator…
<YYINITIAL> {
{A}                 { return symbol(TokenNames.A); }
{B_STAR}            { return symbol(TokenNames.B_STAR); }
<<EOF>>             { return symbol(TokenNames.EOF);}
}
```

# Example 1: Token Definitions

**TokenNames.java**

```
public interface TokenNames {
        /* terminals */
        public static final int EOF = 0;
        public static final int A = 1;
        public static final int B_STAR = 2;
}
```

# Example 1: Main

Main.java

```
Lexer l = new Lexer(fileReader);      // auto-generated scanner
Symbol s = l.next_token();
while (s.sym != TokenNames.EOF) {
        System.out.print("[");
        System.out.print(l.getLine() + ", " + l.getTokenStartPosition);
        System.out.print("]:");
        System.out.print(s.sym + "\n");
        s = l.next_token();
}
```

# Example 1: Output

What will be the output for the following input?

- ababbbb

A = a
B_STAR = b*

# Example 1: Output

What will be the output for the following input?

- ababbbb

```
[1,1]:1                    // A
[1,2]:2              // B_STAR
[1,3]:1                    //A
[1,4]:2              //B_STAR
```

***Format:***

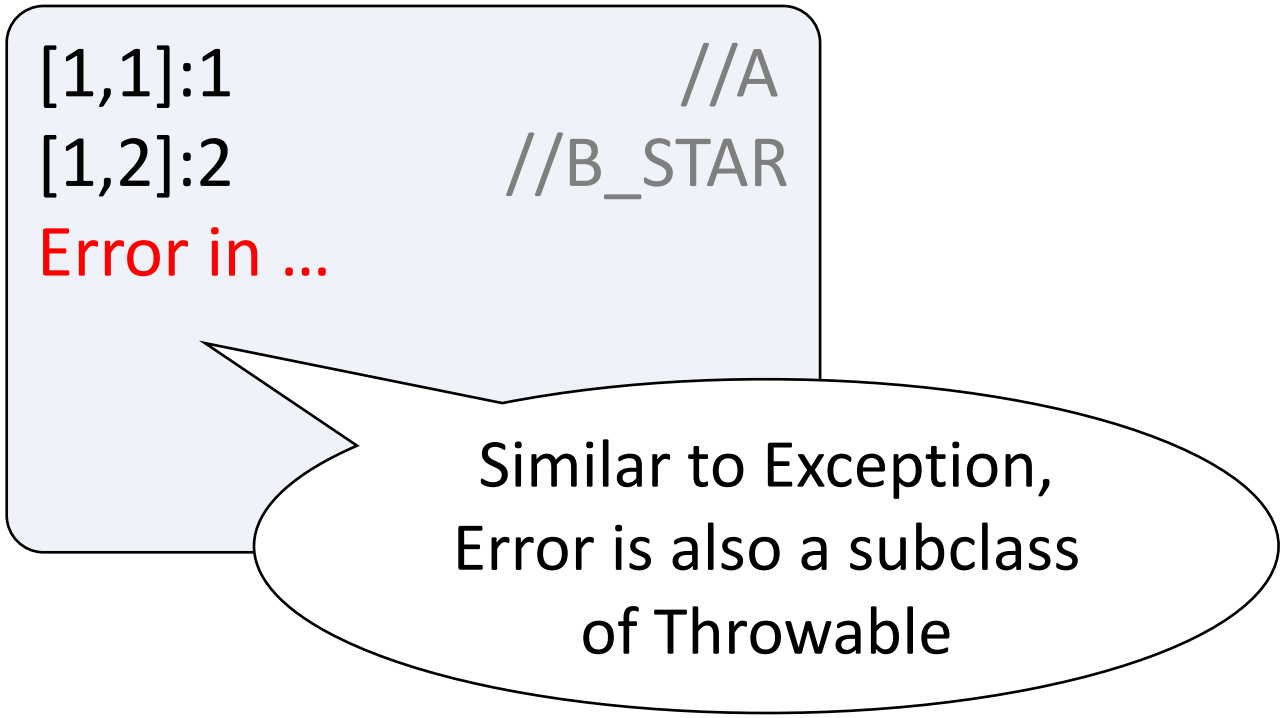
**[line,column]:<token_type>**

A = a
B_STAR = b*

# Example 1: Output

What will be the output for the following input?

- ab abbbb



# Example 1: The EOF Token

- Why do we need the EOF token?

# Example 2: Counting Words

- How can we use JFlex to count words for a given input file?
- Assume only letters

# Example 2: LEX Definitions

```
%{
...
public int word_count = 0;
%}
WORD = [a-zA-Z]+
ANY = [^a-zA-Z]+
%%
<YYINITIAL> {
{WORD}    { word_count++; }
{ANY}     { }
<<EOF>>   { return symbol(TokenNames.EOF); }
}
```

# Example 2

- Other definitions instead of **ANY**?
  - \n|.
  - . excludes newline
  - Another option: [^]

# Example 3: Calculator

How can we use JFlex to detect calculator tokens?

- Numbers, parentheses, operators
- 1+1, (9), 1+(0000, …

# Example 3: LEX Definitions

```
%{
…
private Symbol symbol(int type, Object value) { return new Symbol(type, yyline, yycolumn, value);}
%}
PLUS = “+”
L_PAREN = “(”
R_PAREN = “)”
NUMBER = [0-9]+
%%
<YYINITIAL> {
{PLUS}              { return symbol(TokenNames.PLUS); }
{L_PAREN}           { return symbol(TokenNames.L_PAREN); }
{R_PAREN}           { return symbol(TokenNames.R_PAREN); }
{NUMBER}            { return symbol(TokenNames.NUMBER, new Integer(yytext())); }
<<EOF>>             { return symbol(TokenNames.EOF); }
}
```

# Example 3: Token Definitions

**TokenNames.java**

```
public interface TokenNames {
    /* terminals */
    public static final int EOF = 0;
    public static final int PLUS = 1;
    public static final int L_PAREN = 2;
    public static final int R_PAREN = 3;
    public static final int NUMBER = 4;
}
```

# Example 1: Main

Main.java

```
Lexer l = new Lexer(fileReader);      // auto-generated scanner
Symbol s = l.next_token();
while (s.sym != TokenNames.EOF) {
        System.out.print("[");
        System.out.print(l.getLine() + ", " + l.getTokenStartPosition);
        System.out.print("]:");
        System.out.print(s.sym + " ");
        System.out.print(s.value + "\n");
        s = l.next_token();
}
```

# Example 3: Output

What will be the output for:

- 1(+2345

# Example 3: Output

What will be the output for:

- 1(+2345

```
[1,1]:4 1       //NUMBER
[1,2]:2 null    //L_PAREN
[1,3]:1 null        //PLUS
[1,4]:4 2345    //NUMBER
```

# Example 4: Definition Order

```
%{
private Symbol symbol(int type) {  return new Symbol(type, yyline, yycolumn); }
public int getLine() { return yyline + 1; }
public int getTokenStartPosition() { return yycolumn + 1; }
%}
T1 = a
T2 = ab*
%%  // separator…
<YYINITIAL> {
{T1}                { return symbol(TokenNames.T1); }
{T2}                { return symbol(TokenNames.T2); }
<<EOF>>             { return symbol(TokenNames.EOF); }
}
```

# Example 4: Definition Order

What will be the output for:

- aabbbba

```
[1,1]:1          //a
[1,2]:2          //ab*
[1,7]:1          //a
```

```
T1 = a
T2 = ab*

<YYINITIAL> {
{T1} { return symbol(TokenNames.T1); }
{T2} { return symbol(TokenNames.T2); }
<<EOF>> { return symbol(TokenNames.EOF); }
}
```

# Example 4: Definition Order

What if the order is swapped?

- aabbbba

```
[1,1]:2          //ab*
[1,2]:2          //ab*
[1,7]:2          //ab*
```

```
T1 = a
T2 = ab*

<YYINITIAL> {
{T2} { return symbol(TokenNames.T2); }
{T1} { return symbol(TokenNames.T1);
<<EOF>> { return symbol(TokenNames.EOF); }
}
```

# Example 5: Lexical States

- Between * allow only numbers
  - *123* is legal
  - ** is legal
  - *** is illegal

# Example 5: Lexical States

```
…
DIGIT = [0-9]
%state ASTERISK
%%
<YYINITIAL> {
“*”             { yybegin(ASTERISK); }
<<EOF>>         { return symbol(TokenNames.EOF); }
}
<ASTERISK> {
“*”             { yybegin(YYINITIAL); }
{DIGIT}         { }
<<EOF>>         { throw new Error(); }
}
```

# Additional Resources

- Jflex manual
  https://www.jflex.de/manual.html

- Regular Expressions Definitions for C
  http://www.lysator.liu.se/c/ANSI-C-grammar-l.html

- Helps understand and build regular expression
  https://regexr.com/
  - Note: includes syntax that is not recognized by JFlex

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

What will the lexer print for the input:

- **abcaacacaaabbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**abcaacacaaabbbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caacacaaabbbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|cacaaabbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaabbbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|bbaaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|b|baaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|b|b|aaabcaaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|b|b|aaab|caaca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|b|b|aaab|caa|ca**

# Exam Question

Consider the following lex-like definition:

- a*b { print “1” }
- ca { print “2” }
- a*ca* { print “3” }

**ab|caa|ca|caaa|b|b|aaab|caa|ca**

- Answer: **132311132**