

EDAN65: Compilers, Lecture 02

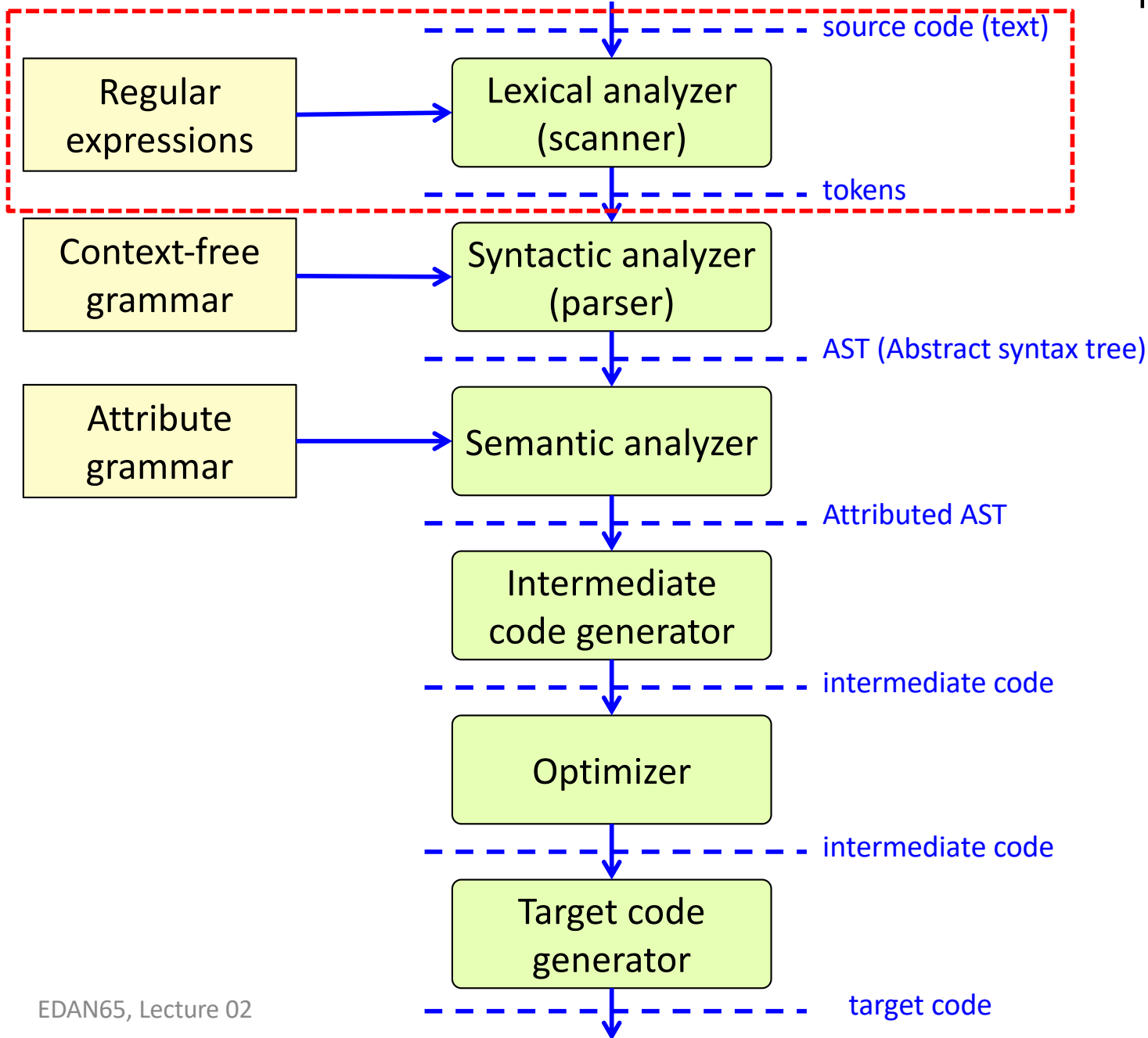
# Regular expressions and scanning

Görel Hedin

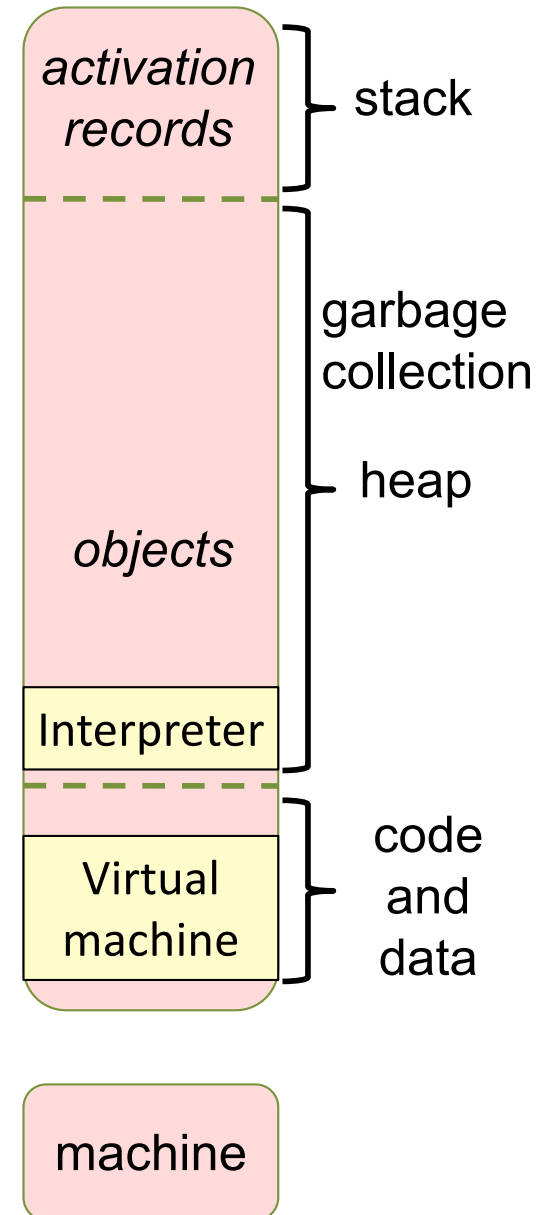
Revised: 2024-09-03

# Course overview

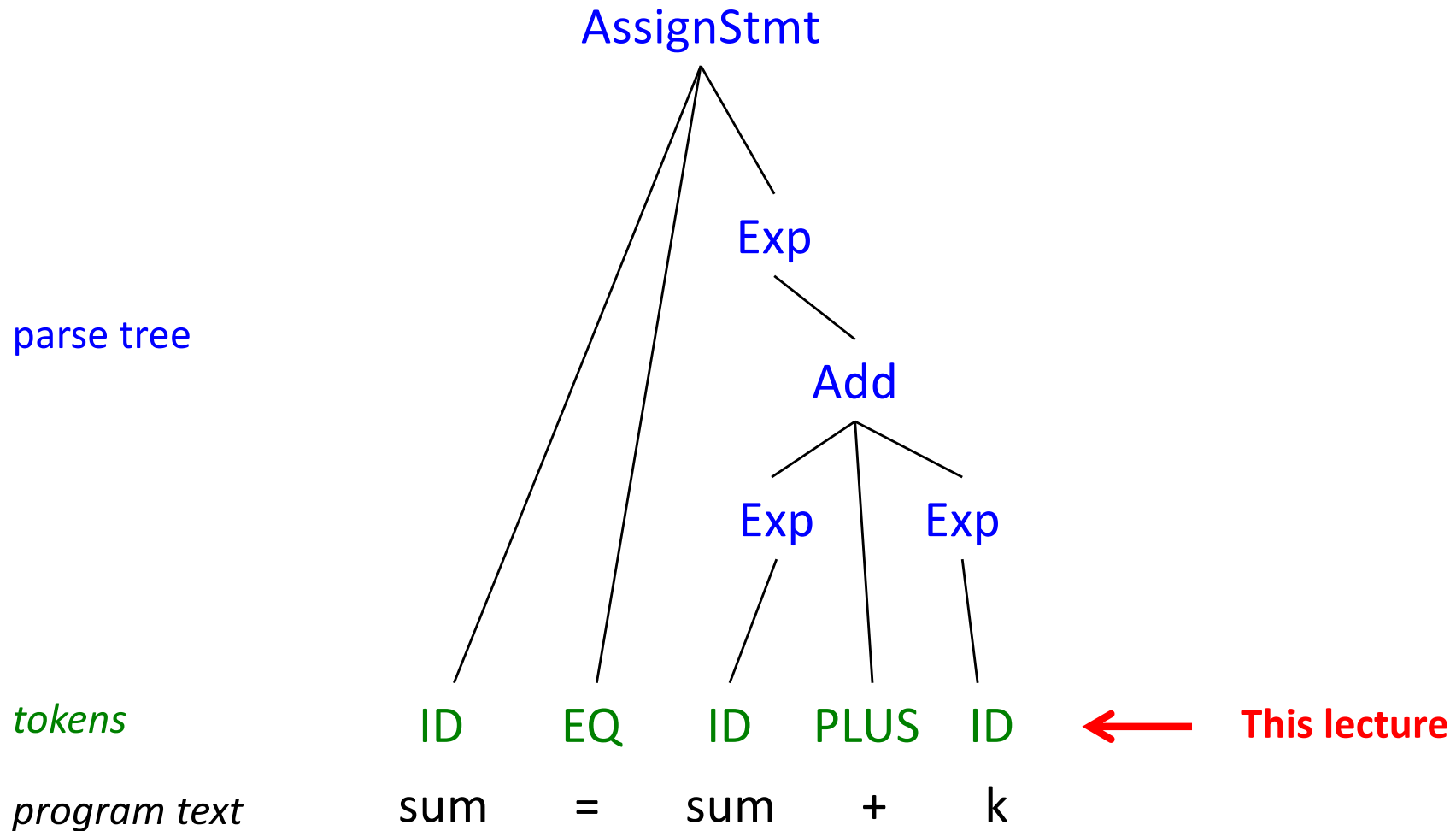
**This lecture**



runtime system



# Analyzing program text



# How split this Java code into tokens?

```
sum = sum + k;  
// possibly print...  
if (sum <= 100)  
    print("The sum is at most 100");
```

# How split this Java code into tokens?

```
sum = sum + k;\n
```

```
// possibly print...\n
```

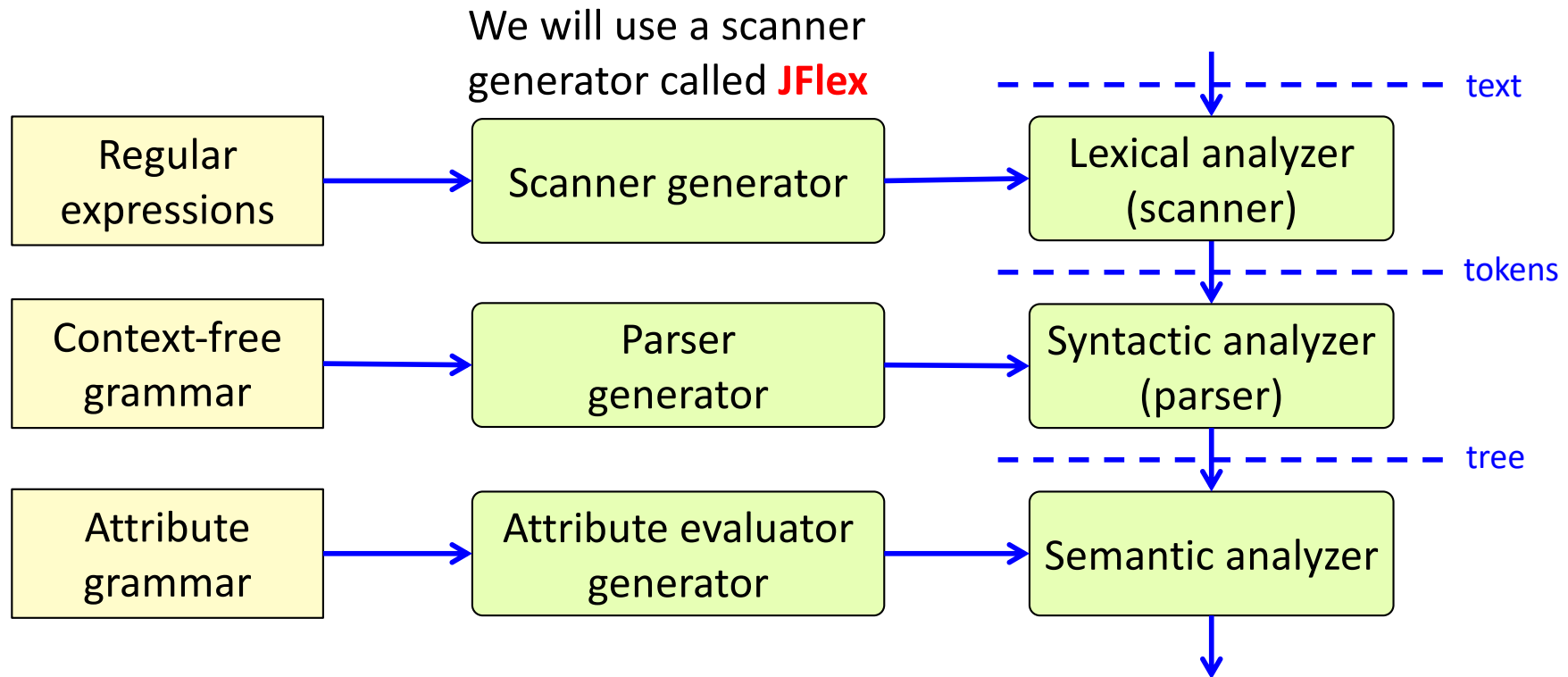
```
if (sum <= 100)\n
```

```
    print("The sum is at most 100");\n
```

 token

 whitespace and comments

# Recall: Generating the compiler:



# Some typical tokens

	Token	Example lexemes
Reserved words (keywords)	IF THEN FOR	if then for
Identifiers	ID	B alpha k10
Literals	INT FLOAT STRING CHAR	1230 99 2016 3.1416 0.2 "Hello" "" "100%" 'A' 'c' '%'
Operators	PLUS INCR NE	+ ++ !=
Separators	SEMI COMMA LPAREN	; , (

# Some typical tokens

	Token	Example lexemes	Regular expression
Reserved words (keywords)	IF THEN FOR	if then for	"if" "then" "for"
Identifiers	ID	B alpha k10	[A-Za-z][A-Za-z0-9]*
Literals	INT FLOAT STRING CHAR	1230 99 2016 3.1416 0.2 "Hello" "" "100%" 'A' 'c' '%'	[0-9]+ [0-9]+ "." [0-9]+ \" [^\"]* \ ' [^\']* \'
Operators	PLUS INCR NE	+ ++ !=	"+" "++" "!="
Separators	SEMI COMMA LPAREN	; , (	" ; " " , " " ( "

JFlex syntax



# Formal languages

# Formal languages

- An *alphabet*,  $\Sigma$ , is a set of symbols (nonempty and finite).
- A *string* is a sequence of symbols (each string is finite)
- A *formal language*,  $L$ , is a set of strings (can be infinite).
  
- We would like to have *rules* or *algorithms* for defining a language – deciding if a certain string over the alphabet belongs to the language or not.

# Example: Languages over binary numbers

Suppose we have the alphabet  $\Sigma = \{0, 1\}$

Example languages:

- The set of all possible combinations of zeros and ones:

$L_0 =$

- All binary numbers without unnecessary leading zeros:

$L_1 =$

- All binary numbers with two digits:

$L_2 =$

- ...

# Example: Languages over binary numbers

Suppose we have the alphabet  $\Sigma = \{0, 1\}$

Example languages:

- The set of all possible combinations of zeros and ones:  
 $L_0 = \{\text{"", "0", "1", "00", "01", "10", "11", "000", \dots\}$
- All binary numbers without unnecessary leading zeros:  
 $L_1 = \{\text{"0", "1", "10", "11", "100", "101", "110", "111", "1000", \dots\}$
- All binary numbers with two digits:  
 $L_2 = \{\text{"00", "01", "10", "11"}\}$
- ...

# Example: Languages over UNICODE

Here, the alphabet  $\Sigma$  is the set of UNICODE characters

Example languages:

- All possible Java keywords: {"class", "import", "public", ...}
- All possible lexemes corresponding to Java tokens.
- All possible lexemes corresponding to Java whitespace.
- All binary numbers
- ...

# Example: Languages over Java tokens

Here, the alphabet  $\Sigma$  is the set of Java tokens

Example languages:

- All syntactically correct Java programs
- All that are syntactically incorrect
- All that are compile-time correct
- All that terminate
- ...

# Example: Languages over Java tokens

Here, the alphabet  $\Sigma$  is the set of Java tokens

Example languages:

- All syntactically correct Java programs
- All that are syntactically incorrect
- All that are compile-time correct
- All that terminate (But this language cannot be computed: Termination for Java programs is *undecidable*: it is not possible to construct an algorithm that decides for *any* string, if it is a terminating program or not.)
- ...

# Different kinds of rules

Increasingly powerful:

- Regular expressions (for tokens)
- Context-free grammars (for syntax trees)
- Attribute grammars (context-free grammar + extra rules for further restricting the language)



# Regular expressions (core notation)

RE	read	is called
$a$	$a$	symbol
$M \mid N$	$M$ or $N$	alternative
$M N$	$M$ followed by $N$	concatenation
$\varepsilon$	the empty string	epsilon
$M^*$	zero or more $M$	repetition (Kleene star)
$(M)$		scope

where  $a$  is a symbol in the alphabet (e.g.,  $\{0,1\}$  or UNICODE)  
and  $M$  and  $N$  are regular expressions

Each regular expression defines a language over the alphabet  
(a set of strings that belong to the language).

Priorities:  $M \mid N P^*$  means  $M \mid (N (P^*))$

# Example

$a \mid bc^*$

# Example

$a \mid bc^*$

means

$\{ "a", "b", "bc", "bcc", "bccc", \dots \}$

# Another example

$(a \mid b \mid \varepsilon) c^*$

## Another example

$(a \mid b \mid \varepsilon) c^*$

means

$\{ "a", "b", "", "ac", "bc", "c", "acc", "bcc", "cc", \dots \}$

# REs: core + extended notation

Core RE	read	is called
$a$	$a$	symbol
$M \mid N$	$M$ or $N$	alternative
$M N$	$M$ followed by $N$	concatenation
$\varepsilon$	the empty string	epsilon
$M^*$	zero or more $M$	repetition (Kleene star)
$(M)$		

Extended RE	read	means
$M^+$	at least one ...	$M M^*$
$M^?$	<i>optional ...</i>	$\varepsilon \mid M$
$[aou]$ $[a-zA-Z]$	<i>one of ... (a character class)</i>	$a \mid o \mid u$ $a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z$
$[^0-9]$ (Appel notation: $\sim[0-9]$ )	not ...	one character, but not anyone of those listed
"a+b"	the string ...	$a \backslash + b$

# Exercise

Regular expression	Language
$(ab)^+ c?$	
$[defq]$	
$[g-k]$	
$[a-z]^*$	
$[\wedge b-d]$	
$("hi")^*$	

assuming the alphabet is  $\{a, b, \dots, z\}$

# Solution

Regular expression	Language
$(ab)^+ c?$	$\{ "ab", "abab", \dots, "abc", "ababc", \dots \}$
$[defq]$	$\{ "d", "e", "f", "q" \}$
$[g-k]$	$\{ "g", "h", "i", "j", "k" \}$
$[a-z]^*$	$\{ "", "a", "b", "c", \dots, "z", "aa", "ab", \dots "az", "ba", "bb", \dots "bz", "ca", \dots \}$
$[^b-d]$	$\{ "a", "e", "f", \dots, "z" \}$
$("hi")^*$	$\{ "", "hi", "hihi", "hihihi", \dots \}$

assuming the alphabet is  $\{a, b, \dots, z\}$



# Exercise

Write a regular expression that defines the language of all decimal numbers, like

3.14 0.75 4711 0 ...

But not numbers lacking an integer part. And not numbers with a decimal point but lacking a fractional part. So not numbers like

17. .236 .

Leading and trailing zeros are allowed. So the following are ok:

007 008.00 0.0 1.700

- Use the extended notation.
- Then translate the expression to the core notation
- Then write an expression that disallows unnecessary leading zeros (in the extended notation)

## Core RE

$a$

$M \mid N$

$MN$

$\varepsilon$

$M^*$

$(M)$

## Extended RE

$M^+$

$M^?$

$[aou]$

$[a-zA-Z]$

$[^0-9]$

$"a+b"$

# Solution

a)

$[0-9]^+ (\text{"."}[0-9]^+)?$

b)

$(0 | \dots | 9)(0 | \dots | 9)^* (\epsilon | \text{"."} (0 | \dots | 9) (0 | \dots | 9)^*)$

c)

$(0 | [1-9] [0-9]^*) (\text{"."}[0-9]^+)?$

# Escaped characters

Use backslash to escape metacharacters and non-printing control characters.

Metacharacters
\+
\*
\(
\)
\\
\\
...

Non-printing control characters	
\n	newline
\r	return
\t	tab
\f	formfeed
...	

# Some typical non-tokens

Non-Token	Example lexemes
WHITESPACE	blank tab newline return
ENDOFLINECOMMENT	// comment

Non-tokens are also recognized by the scanner, just like tokens.  
But they are not sent on to the parser.

# Some typical non-tokens

Non-Token	Example lexemes	Regular expression (jflex)
WHITESPACE	blank tab newline return	" "   \t   \n   \r
ENDOFFLINECOMMENT	// comment	"//" [^\n\r]* [\n\r]?

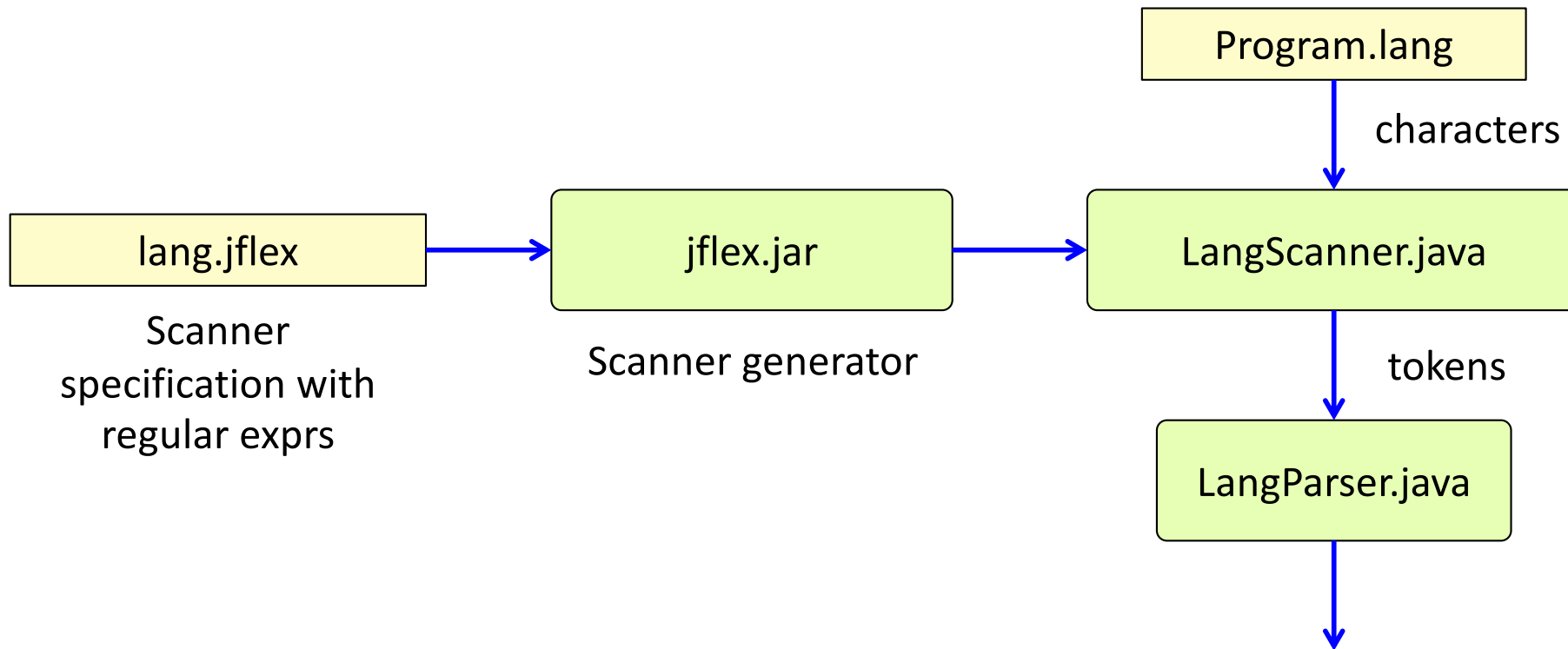
JFlex syntax

Non-tokens are also recognized by the scanner, just like tokens.  
But they are not sent on to the parser.

(The newline/return ending an end-of-line comment is optional in order to allow a file to end with an end-of-line comment, without an extra newline/return.)

# JFlex: A scanner generator

Generating a scanner for a language *lang*



# A JFlex specification

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Our own class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"           { return new Token("IF"); }
"="           { return new Token("ASSIGN"); }
"<"          { return new Token("LT"); }
"<="        { return new Token("LE"); }
[a-zA-Z]+    { return new Token("ID", yytext()); }
...
```

## Rules and lexical actions

Each rule has the form:

*regular-expression* { *lexical action* }

The lexical action consists of arbitrary Java code.

It is run when a regular expression is matched.

The method `yytext()` returns the lexeme (the token value).

# A JFlex specification

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Our own class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"           { return new Token("IF"); }
"="           { return new Token("ASSIGN"); }
"<"          { return new Token("LT"); }
"<="        { return new Token("LE"); }
[a-zA-Z]+     { return new Token("ID", yytext()); }
...
```

## Rules and lexical actions

Each rule has the form:

*regular-expression* { *lexical action* }

The lexical action consists of arbitrary Java code.

It is run when a regular expression is matched.

The method `yytext()` returns the lexeme (the token value).

**What rules are used when scanning "a < b"?**



# Ambiguities?

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"                   { return new Token("IF"); }
"="                   { return new Token("ASSIGN"); }
"<"                   { return new Token("LT"); }
"<="                 { return new Token("LE"); }
[a-zA-Z]+             { return new Token("ID", yytext()); }
...
```

# Ambiguities?

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"                   { return new Token("IF"); }
"="                   { return new Token("ASSIGN"); }
"<"                   { return new Token("LT"); }
"<="                 { return new Token("LE"); }
[a-zA-Z]+             { return new Token("ID", yytext()); }
...
```

## Are the token definitions ambiguous?

Which rules match "<="?

Which rules match "if"?

Which rules match "iff"?

Which rules match "xyz"?

# Extra rules for resolving ambiguities

## **Longest match**

If one rule can be used to match a token, but there is another rule that will match a longer token, the latter rule will be chosen. This way, the scanner will match the longest token possible.

## **Rule priority**

If two rules can be used to match the same sequence of characters, the first one takes priority.

# Implementation of scanners

Observation:

*Regular expressions* are equivalent to *finite automata* (finite-state machines).  
(They can recognize the same class of formal languages: the *regular languages*.)

Overall approach:

- Translate each token regular expression to a finite automaton.  
Label the final state with the token.
- Merge all the automata.
- The resulting automaton will in general be *nondeterministic*
- Translate the nondeterministic automaton to a *deterministic* automaton.
- Implement the deterministic automaton,  
either using switch statements or a table.

A scanner generator automates this process.

# Finite automaton

Regular expression:  
[ab] c\* d?

 state

 transition

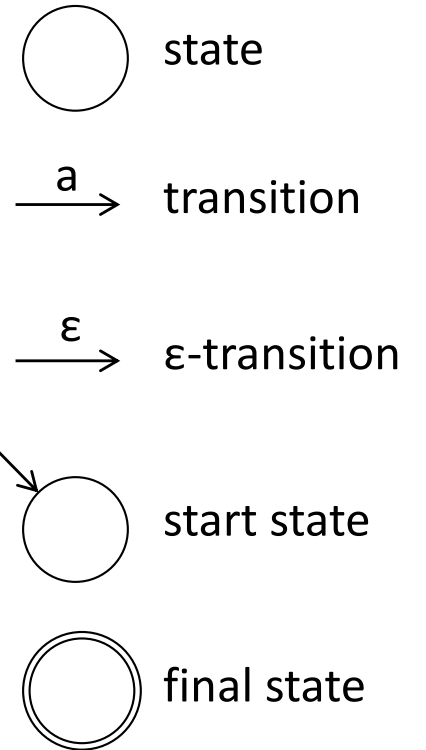
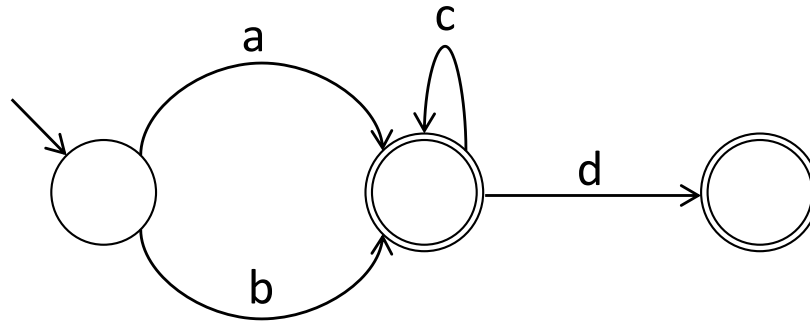
  $\epsilon$ -transition

 start state

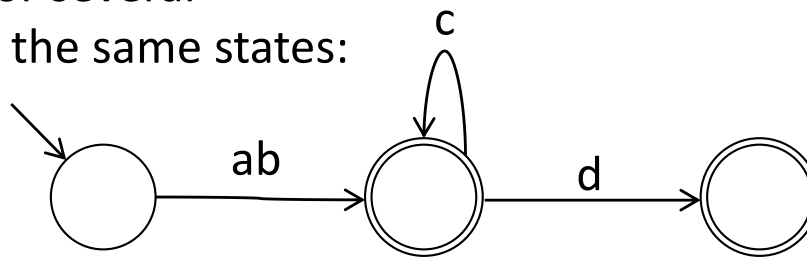
 final state

# Finite automaton

Regular expression:  
[ab] c\* d?



or, with shorthand for several transitions between the same states:



# Construct an automaton for each token regexp

"if"

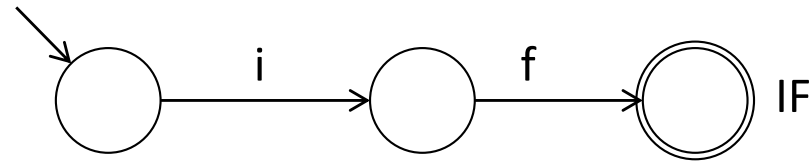
[0-9]+

" " | \n | \t

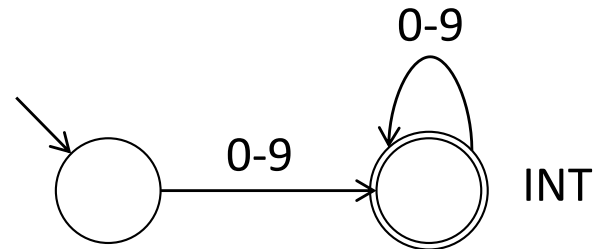
[a-zA-Z]+

# Construct an automaton for each token regexp

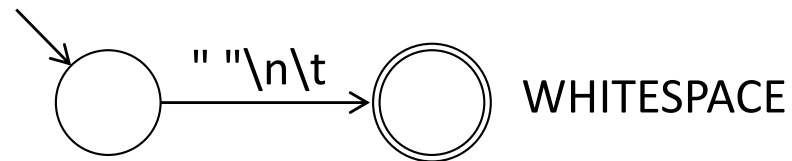
"if"



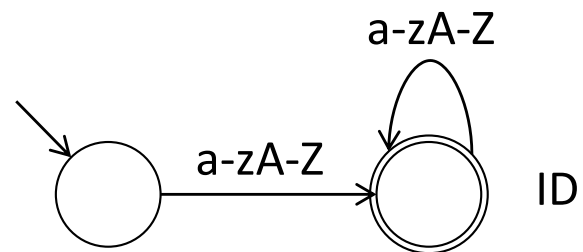
[0-9]<sup>+</sup>



" " | \n | \t

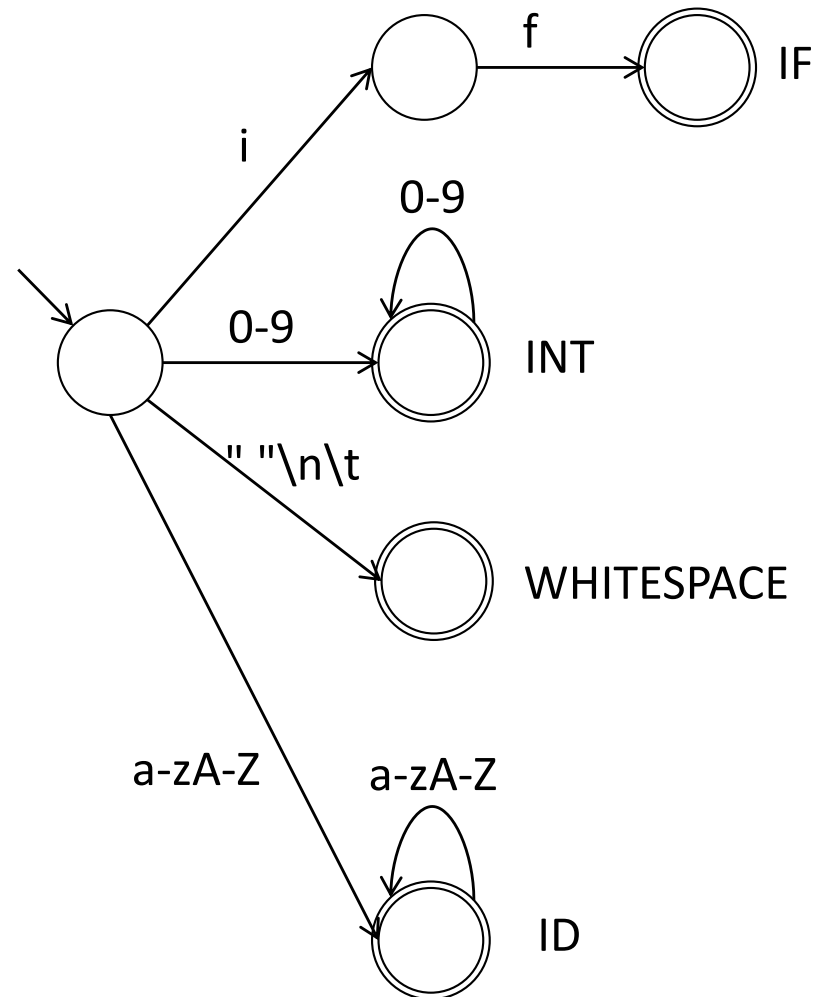


[a-zA-Z]<sup>+</sup>





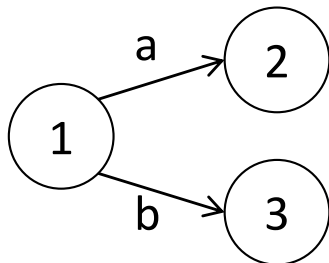
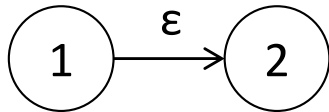
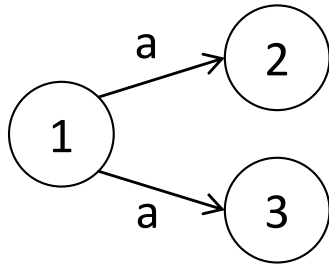
# Merge the start states of the automata



**Is the new automaton deterministic?**

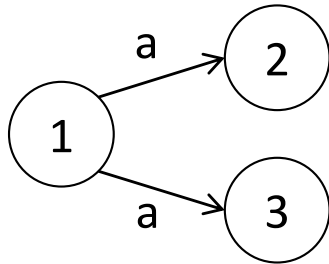
# Deterministic finite automata

Deterministic finite automaton: each transition is uniquely determined by the next symbol.

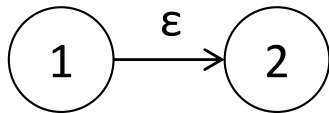


# Deterministic finite automata

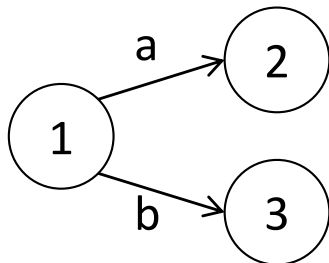
Deterministic finite automaton: each transition is uniquely determined by the next symbol.



**Nondeterministic:** if we read "a" when in state 1, we don't know if we should go to state 2 or 3.



**Nondeterministic:** when we are in state 1, we don't know if we should stay there, or go to state 2 without reading any input. (Epsilon denotes the empty string.)



**Deterministic:** when we are in state 1, the next symbol determines if we go to state 2 or 3.

# DFA versus NFA

## **Deterministic Finite Automaton (DFA)**

A finite automaton is deterministic if

- all outgoing edges from any given state have disjoint character sets
- there are no epsilon edges

Can be implemented efficiently

## **Non-deterministic Finite Automaton (NFA)**

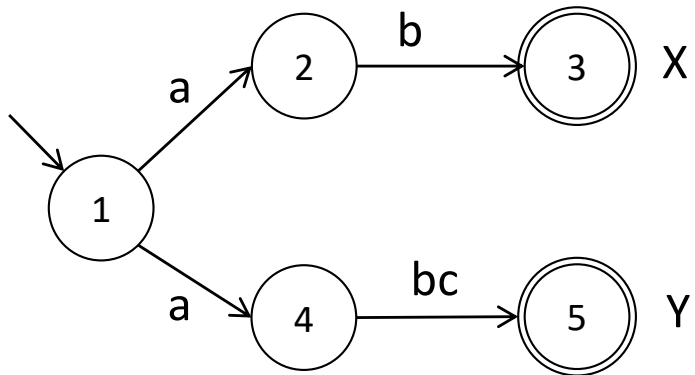
An NFA may have

- two outgoing edges with overlapping character sets
- epsilon edges

Every NFA can be translated to an equivalent DFA.

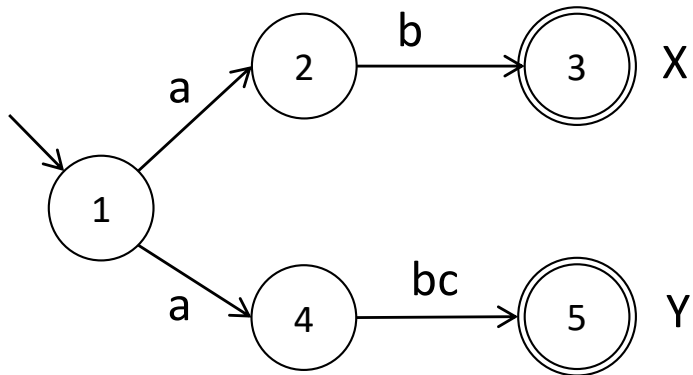
# Example 1

NFA

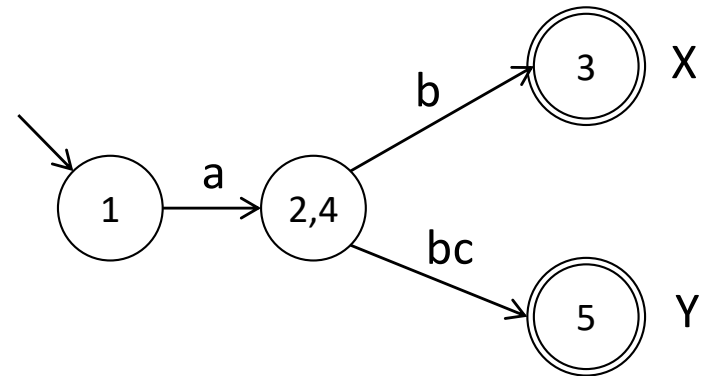


# Example 1

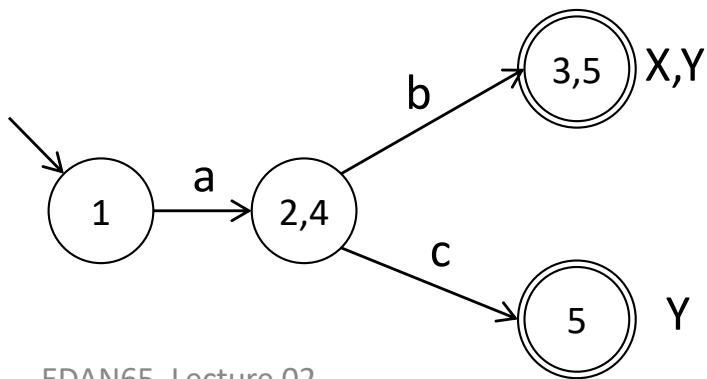
**NFA**



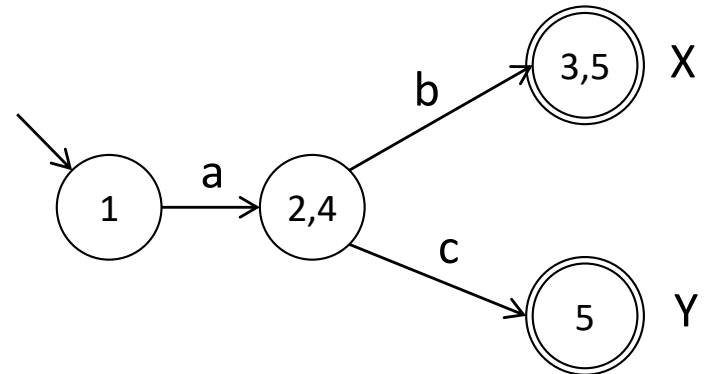
**Still NFA**



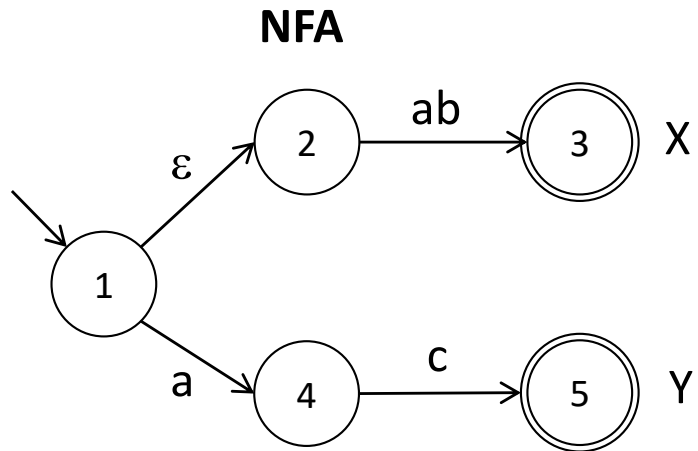
**DFA, but ambiguous final token**



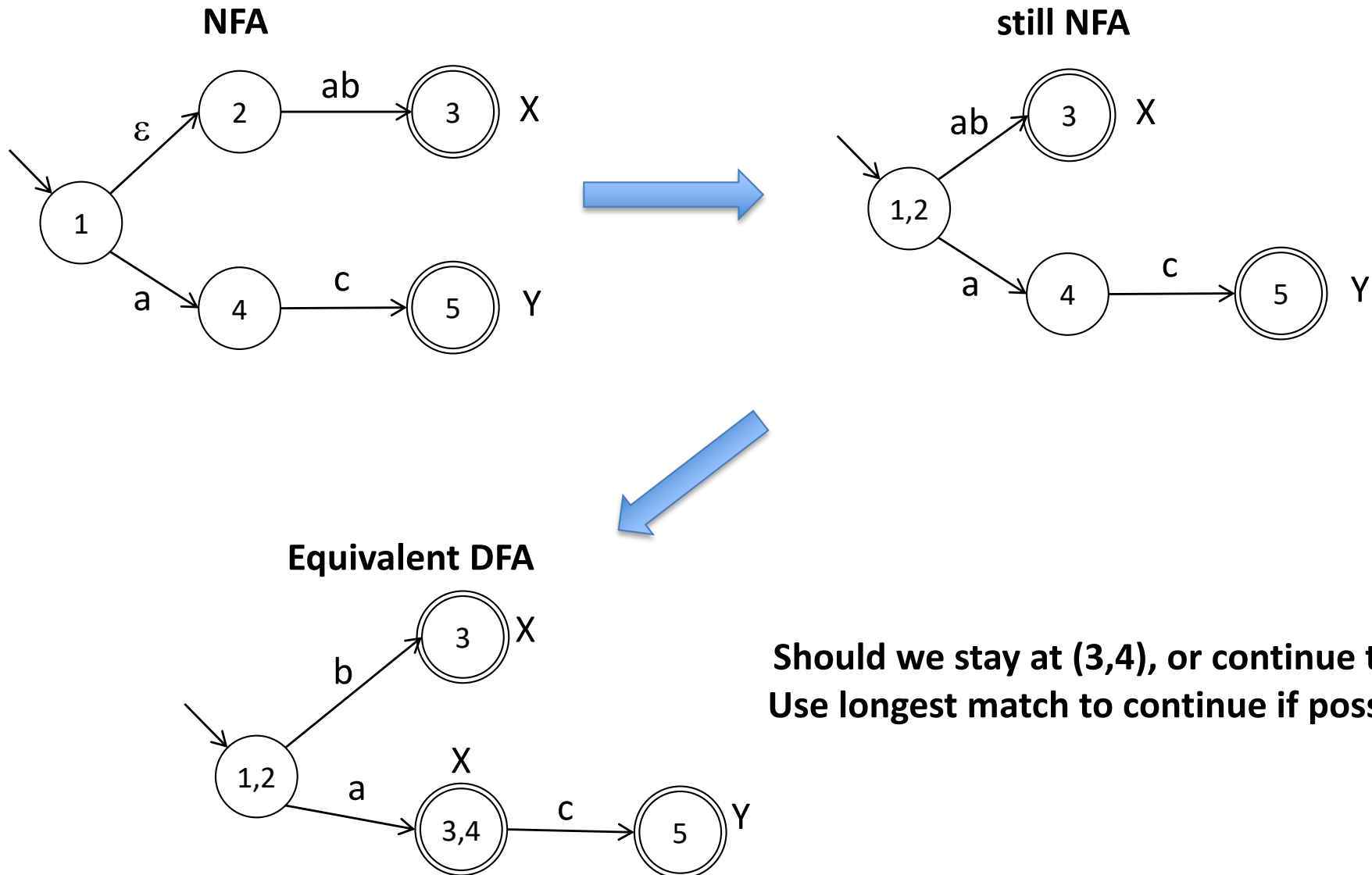
**Use rule priority to pick X  
(assuming rule X is before Y)**



# Example 2



# Example 2





# Translating an NFA to a DFA

## Simulate the NFA

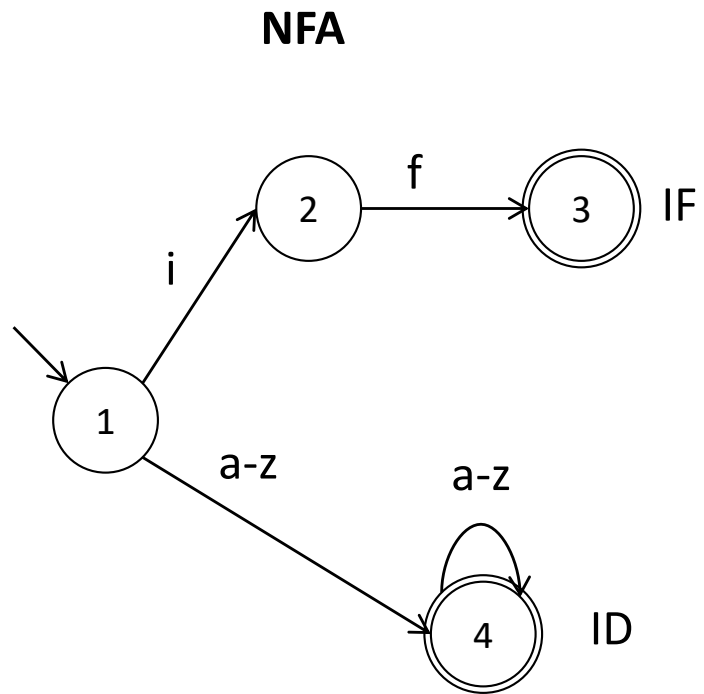
- keep track of a *set* of current NFA-states
- follow  $\epsilon$  edges to extend the current set (take the *closure*)

## Construct the corresponding DFA

- Each such *set* of NFA states corresponds to *one* DFA state
- If any of the NFA states is final, the DFA state is also final, and is marked with the corresponding token.
- If there is more than one token to choose from, select the token that is defined first (rule priority).

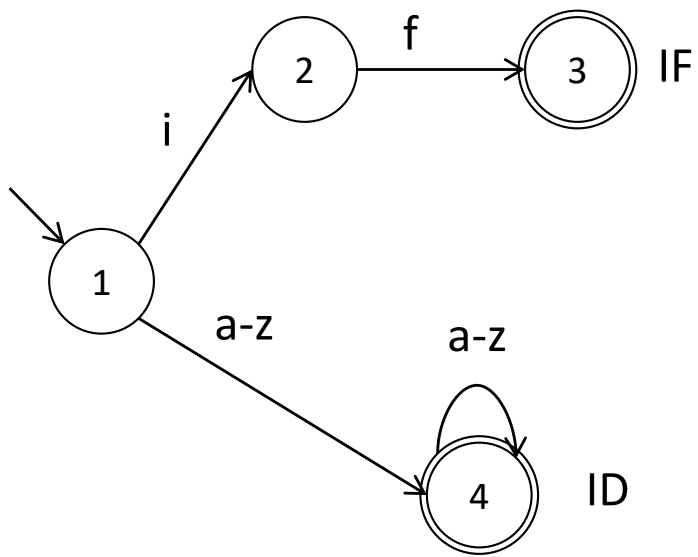
(Minimize the DFA for efficiency)

# Example

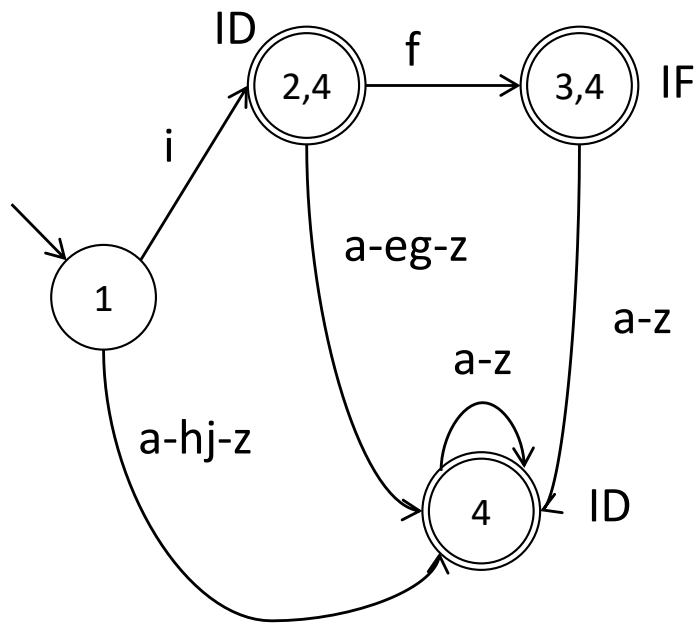


# Example

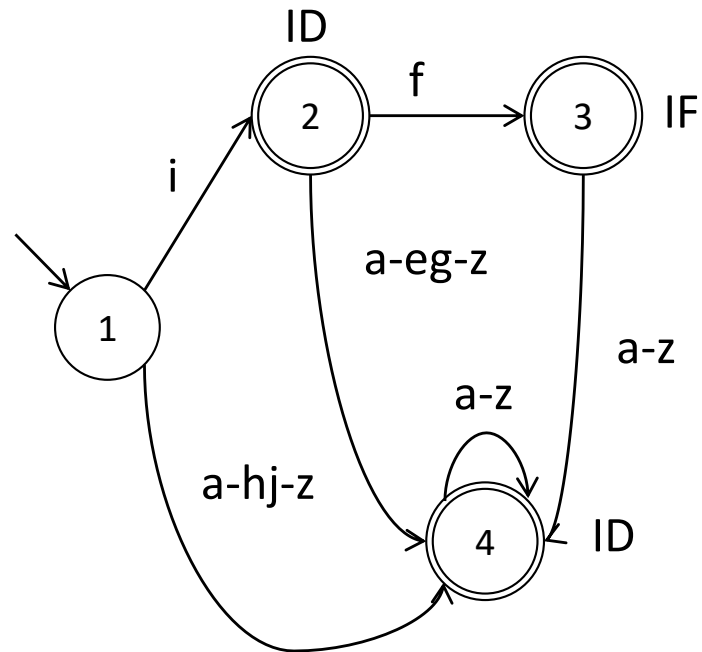
NFA



DFA



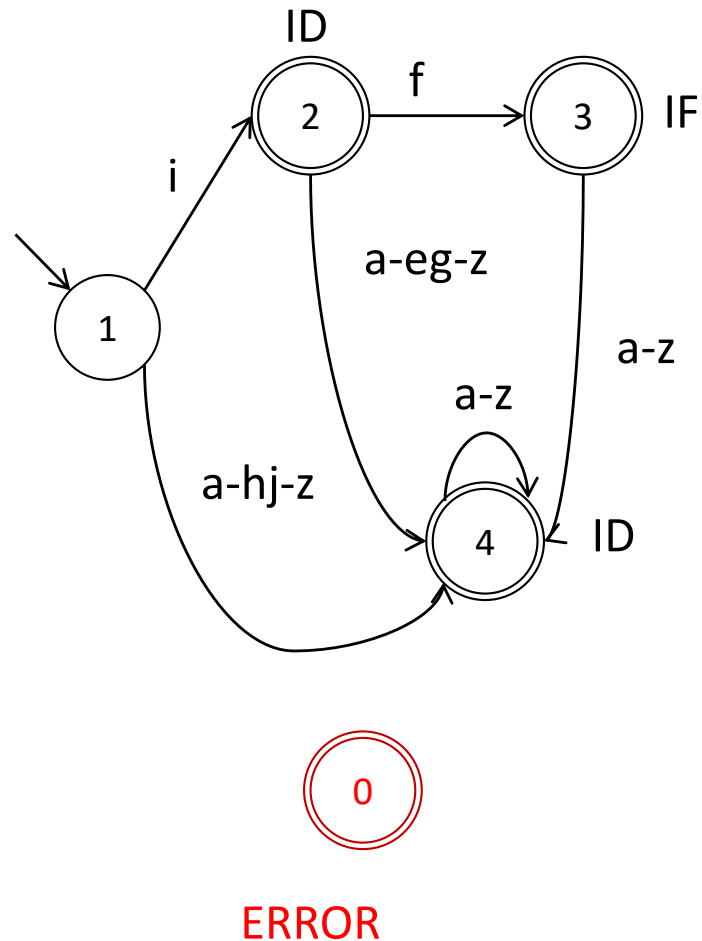
# Error handling



Conceptually (we typically don't draw this explicitly – too much clutter):

- Add a "dead state" (state 0), corresponding to erroneous input.
- Add transitions to the "dead state" for all erroneous input.
- Generate an "ERROR token" when the dead state is reached.

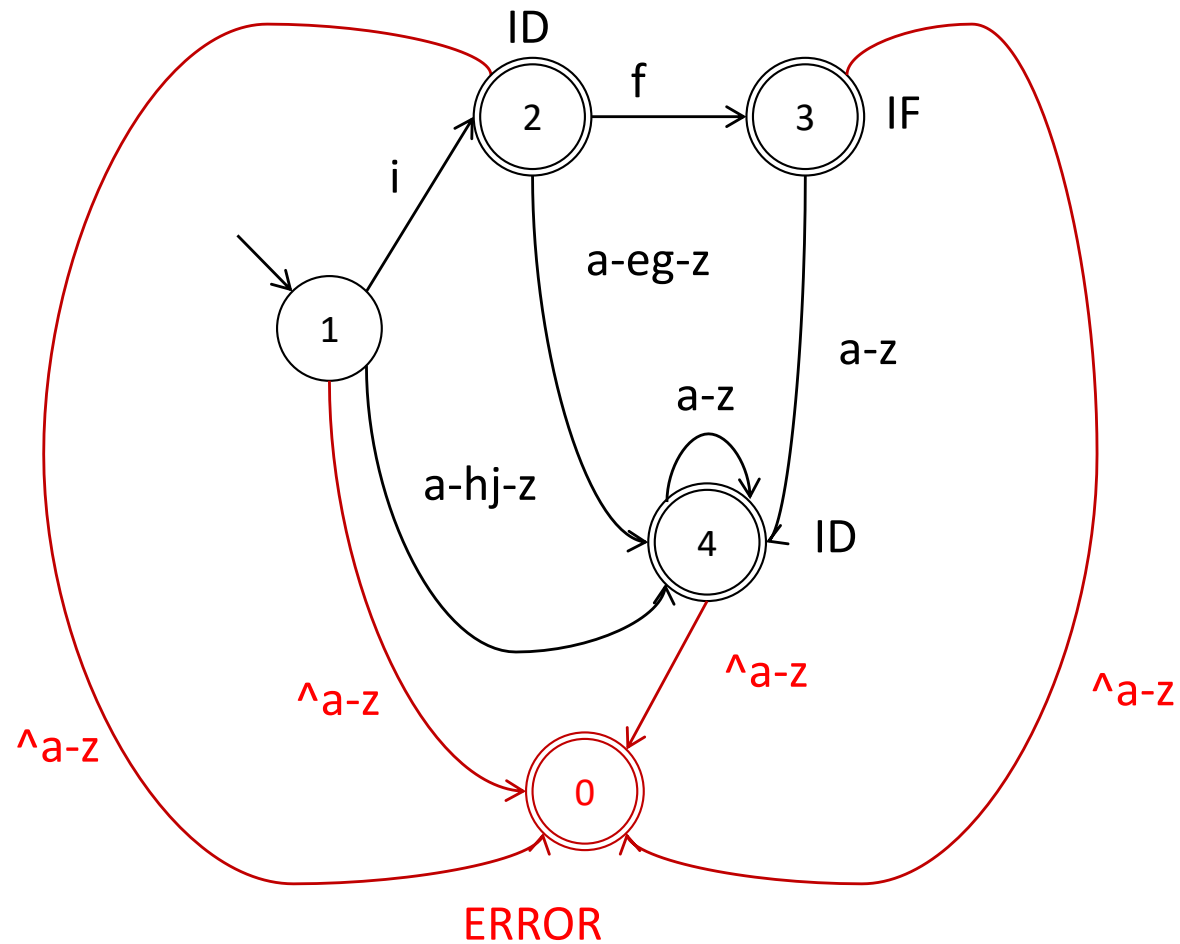
# Error handling



Conceptually (we typically don't draw this explicitly – too much clutter):

- Add a "dead state" (state 0), corresponding to erroneous input.
- Add transitions to the "dead state" for all erroneous input.
- Generate an "ERROR token" when the dead state is reached.

# Error handling



Conceptually (we typically don't draw this explicitly – too much clutter):

- Add a "dead state" (state 0), corresponding to erroneous input.
- Add transitions to the "dead state" for all erroneous input.
- Generate an "ERROR token" when the dead state is reached.

# Implementation alternatives for DFAs

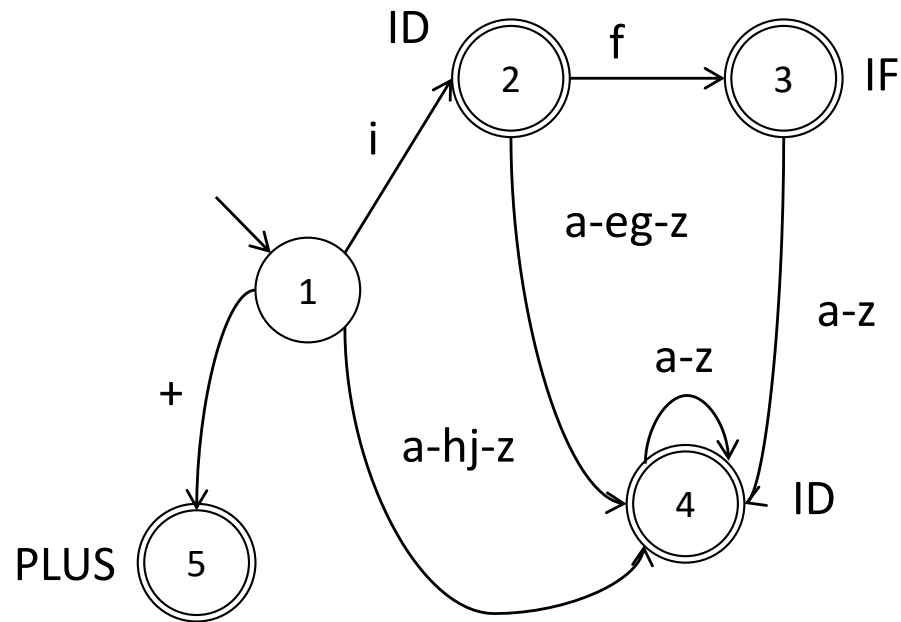
## Table-driven

- Represent the automaton by a table
- Additional table to keep track of final states and token kinds
- A global variable keeps track of the current state

## Switch statements

- Each state is implemented as a switch statement
- Each case implements a state transition as a jump (goto) to another switch statement
- The current state is represented by the program counter.

# Table-driven implementation



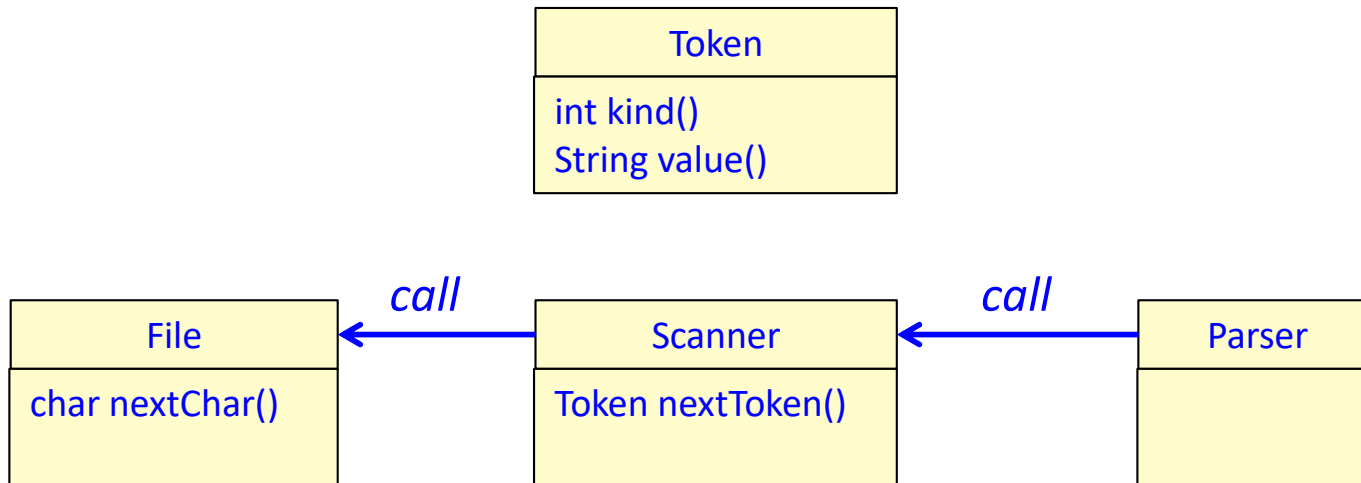
alphabet

	...	+	...	a	...	e	f	g	...	h	i	j	...	z	...	final	token kind
<b>0</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	true	ERROR
<b>1</b>	0	5	0	4	4	4	4	4	4	4	2	4	4	4	0	false	
<b>2</b>	0	0	0	4	4	4	3	4	4	4	4	4	4	4	0	true	ID
<b>3</b>	0	0	0	4	4	4	4	4	4	4	4	4	4	4	0	true	IF
<b>4</b>	0	0	0	4	4	4	4	4	4	4	4	4	4	4	0	true	ID
<b>5</b>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	true	PLUS

states



# Scanner implementation, design



# Scanner implementation, sketch

Idea: Scan the next token by

- starting in the start state
- scan characters until we reach a final state
- return a new token

```
Token nextToken() {
    state = 1; // start state
    while (! isFinal[state]) {
        ch = file.readChar();
        state = edges[state, ch];
    }
    return new Token(kind[state]);
}
```

# Scanner implementation, sketch

Idea: Scan the next token by

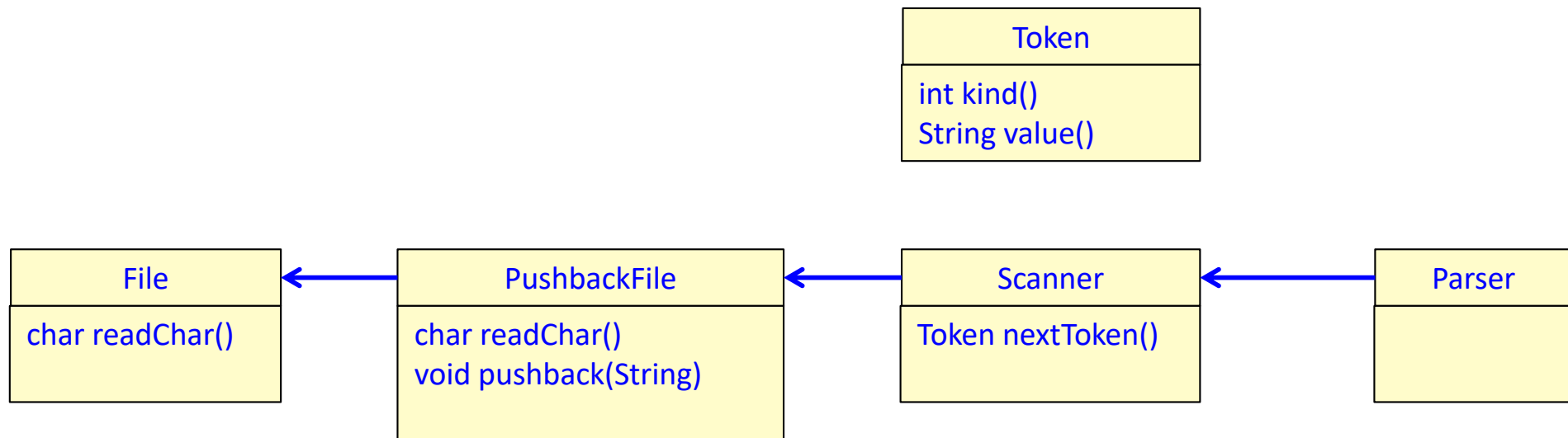
- starting in the start state
- scan characters until we reach a final state
- return a new token

```
Token nextToken() {
    state = 1; // start state
    while (! isFinal[state]) {
        ch = file.readChar();
        state = edges[state, ch];
    }
    return new Token(kind[state]);
}
```

Needs to be extended with handling of:

- longest match
- end of file
- non tokens (like whitespace)
- token values (like the identifier name)
- errors (no token could be matched)

# Extend to longest match, design



## Idea:

- When a token is matched, keep track of it, but don't stop scanning.
- When the error state is reached, return the last (=longest) token matched.
- Push read characters that are unused back into the file, so they can be scanned again.
- Use a PushbackFile to accomplish this.

# Extend to handle longest match, sketch

- When a token is matched (a final state reached), don't stop scanning.
- Keep track of the currently scanned string, `str`.
- Keep track of the latest matched token (`lastFinalState`, `lastTokenValue`).
- Continue scanning until we reach the error state.
- Restore the input stream using `PushBackFile`.
- Return the latest matched token.
- (or return the ERROR token if there was no latest matched token)

```
Token nextToken() {
    state = 1;
    str = "";
    lastFinalState = 0; lastTokenValue = "";
    while (state != 0) {
        ch = pushbackfile.readChar();
        str = str + ch;          // In Java, StringBuilder would be more efficient
        state = edges[state, ch];
        if (isFinal[state]) {
            lastFinalState = state;
            lastTokenValue = str;
        }
    }
    pushbackfile.pushback(str.substring(lastTokenValue.length));
    return new Token(kind[lastFinalState], lastTokenValue);
}
```

# Handling End-of-file (EOF) and non-tokens

## EOF

- construct an explicit EOF token when the end of the file is reached

## Non-tokens (Whitespace & Comments)

- view as tokens of a special kind
- scan them as normal tokens, but don't create token objects for them
- loop in next() until a real token has been found

## Errors

- construct an explicit ERROR token to be returned when no valid token can be found.

# Specifying EOF and ERROR in JFlex

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"                   { return new Token("IF"); }
"="                   { return new Token("ASSIGN"); }
"<"                   { return new Token("LT"); }
"<="                  { return new Token("LE"); }
[a-zA-Z]+             { return new Token("ID", yytext()); }
...
<<EOF>>               { return new Token("EOF"); }
[^]                   { return new Token("ERROR"); }
```

# Specifying EOF and ERROR in JFlex

```
package lang;           // the generated scanner will belong to the package lang
import lang.Token;     // Class for tokens
...

// ignore whitespace
" " | \t | \n | \r | \f { /* ignore */ }

// tokens
"if"                   { return new Token("IF"); }
"="                   { return new Token("ASSIGN"); }
"<"                   { return new Token("LT"); }
"<="                  { return new Token("LE"); }
[a-zA-Z]+             { return new Token("ID", yytext()); }
...
<<EOF>>               { return new Token("EOF"); }
[^]                   { return new Token("ERROR"); }
```

<<EOF>> is a special regular expression in JFlex, matching end of file.

[^] means any character. Due to rule priority, this will match any character not matched by previous rules.



# Example scanner generators

tool	author	generates
lex	Schmidt, Lesk. 1975	C-code
flex ("fast lex")	Paxon. 1987	C-code
jlex		Java code
jflex		Java code
...		

# Limitations of regular expressions for scanning

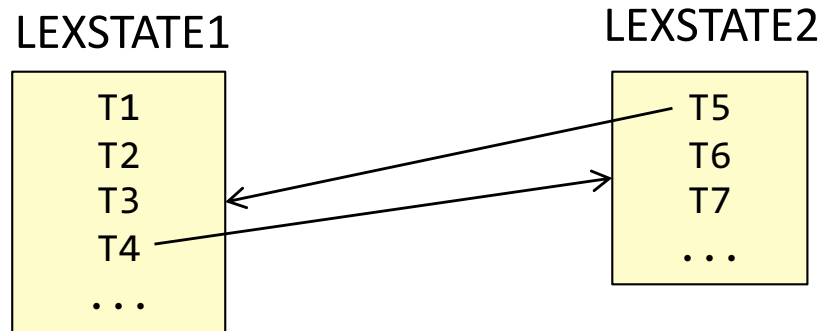
- Nested comments?
- Layout-sensitive syntax?
- Context-sensitive token definitions?  
For example, multi-language documents.

# Limitations of regular expressions for scanning

- Nested comments?
- Layout-sensitive syntax?
- Context-sensitive token definitions?  
For example, multi-language documents.
  
- Two mechanisms in scanner generators for workarounds:
  - **Lexical actions:**  
do more than create a token, e.g., count nesting levels of comments.
  - **Lexical states:**  
switch between different sets of token definitions.

# Lexical states

- Some tokens are difficult or impossible to define with regular expressions.
- *Lexical states* (sets of token rules) give the possibility to switch token sets (DFAs) during scanning.



- Useful for multi-line comments, HTML, scanning multi-language documents, etc.
- Supported by many scanner generators (including JFlex)

# Example: multi-line comments

Would like to scan the complete comment as one token:

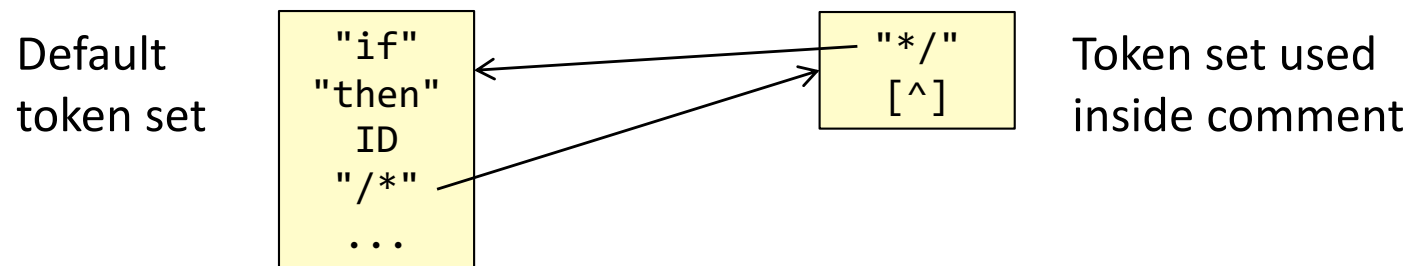
```
/*  
int m() {  
    return 15 / 3 * 4 * 2;  
}  
*/
```

# Example: multi-line comments

Would like to scan the complete comment as one token:

```
/*  
int m() {  
    return 15 / 3 * 4 * 2;  
}  
*/
```

Can be solved easily with lexical states:

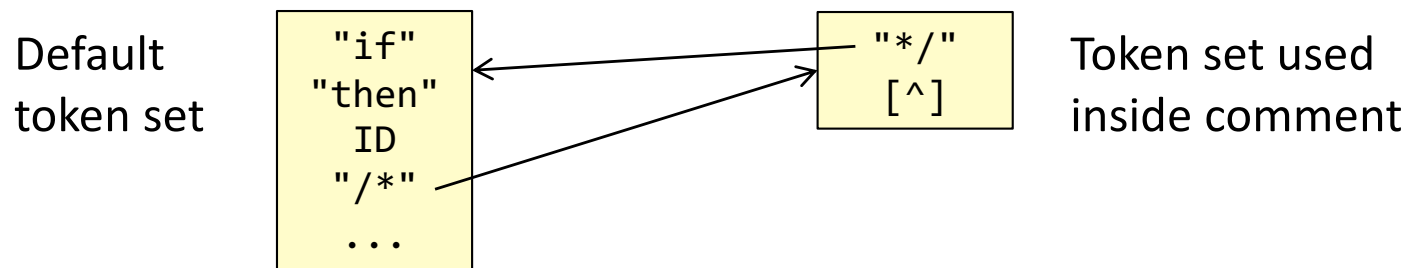


# Example: multi-line comments

Would like to scan the complete comment as one token:

```
/*  
int m() {  
    return 15 / 3 * 4 * 2;  
}  
*/
```

Can be solved easily with lexical states:



Writing an ordinary regular expression for this is difficult:

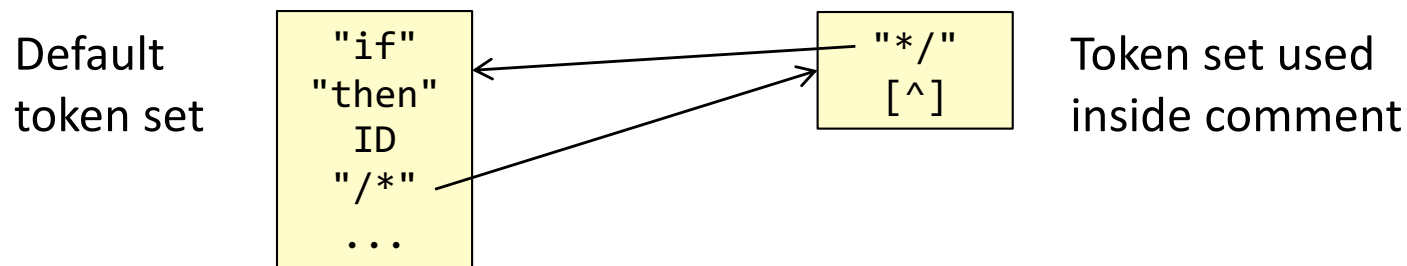
```
"/*"((\*+[^\*])|([^\*]))*\***/"
```

# Example: multi-line comments

Would like to scan the complete comment as one token:

```
/*  
int m() {  
    return 15 / 3 * 4 * 2;  
}  
*/
```

Can be solved easily with lexical states:



Writing an ordinary regular expression for this is difficult:

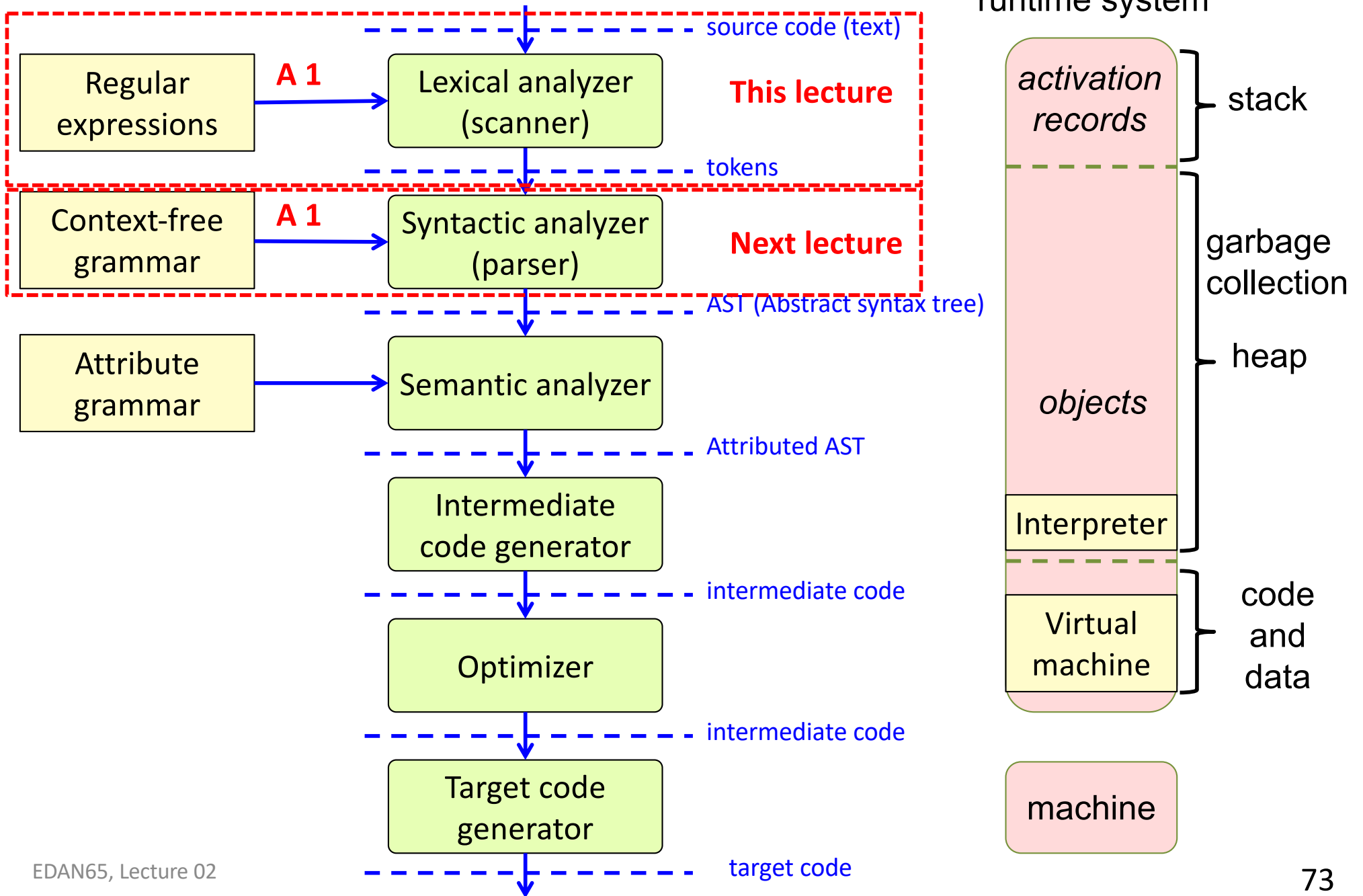
```
"/*"((\*+[^\*])|([^\*]))*\***/"
```

However, some scanner generators, like JFlex, has the special operator *upto* (~) that can be used instead:

```
"/*" ~"*/" { /* Comment */ }
```



# Course overview



# Summary questions

- What is a formal language?
- What is a regular expression?
- What is meant by an ambiguous lexical definition?
- Give some typical examples of ambiguities and how they may be resolved.
- What is a lexical action?
- Give an example of how to construct an NFA for a given lexical definition
- Give an example of how to construct a DFA for a given NFA
- What is the difference between a DFA and an NFA?
- Give an example of how to implement a DFA in Java.
- How is rule priority handled in the implementation? Longest match? EOF? Whitespace? Errors?
- What are lexical states? When are they useful?

You can start on Assignment 1 now. But you will have to wait until the next lecture for the parts about parsing.