The University for
world-class professionals

Manchester Metropolitan University

Compilers: Lexical Analysis

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

Where we are…

- ~~Admin and overview~~
- **Lexical analysis**
- Parsing
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review

Objectives

- Describe the role of a *lexical analyser* (otherwise known as a *scanner*)
- Define *tokens* in the context of lexical analysis
- Recognise and specify *regular expressions*
- Introduce *finite automata*

Example of a Java program

```
class SomeNumbers
{
    static int square (int x)
    {
        return x*x;
    }

    public static void main (String[] args)
    {
        int n=20;
        if (args.length > 0) // change default
            n = Integer.parseInt(args[0]);
        for (int i=0; i <= n; i++)
        {
            System.out.print("The square of " + i + " is ");
            System.out.println(square(i));
        }
    }
}
```

What is a lexical analyser?

- A lexical analyser (or scanner) is the first phase of the compilation process.
- Task is to read the input characters of the source program, group them into *lexemes* – the “words” that make up the program – and output a token stream (input to the next phase of the compiler).

Some definitions

- **Token**: a pair, (token name, attribute)
- **Lexeme**: a (valid) sequence of characters that identifies a token
- **Pattern**: a description of the form that the lexemes of a token may take

For example:

x = x + 1;

Lexeme	Type
x	identifier
=	assignment operator
x	identifier
+	addition operator
1	number
;	end of statement

printf(“total = %d”, score);

Lexeme	Type
printf	identifier
(	open parenthesis
“total = %d”	string literal
,	comma
score	identifier
)	close parenthesis
;	end of statement

Lexical errors

- Lexical analysis cannot identify many source code errors

fi (a == f(x)) …	Lexically correct (in C or Java)
5copy = test;	Lexical error (in C or Java)

Recovering from an error

- “Panic mode” – discard characters until a well-formed token is found
- Delete a single character
- Insert a missing character
- Replace one character by another
- Transpose two adjacent characters

ANTLR automatically recovers for you

(although not necessarily in the most elegant way)

Regular expressions

- Keywords “if” or “else” or “then”

Regular expressions

- Keywords “if” or “else” or “then”

’i’‘f’ | ‘e’’l’’s’’e’ | ’t’’h’’e’’n’

Regular expressions

- Keywords “if” or “else” or “then”

’i’‘f’ | ‘e’’l’’s’’e’ | ’t’’h’’e’’n’

‘if’ |‘else’ | ‘then’

Regular expressions

- Integer: a non empty string of digits

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Integer = digit*

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Integer = digit* -> zero or more digits

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Integer = digit digit*

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Integer = digit digit* -> one digit followed by zero or more digits

Regular expressions

- Integer: a non empty string of digits

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ’7’ | ‘8’ | ‘9’

Integer = digit digit* = digit$^+$

Regular expressions

- a simple Identifier: string of letters or digits, starting with a letter

letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ’h’ ....

Regular expressions

- a simple Identifier: string of letters or digits, starting with a letter

letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ’h’ ….

letter = [a-z]

Regular expressions

- a simple Identifier: string of letters or digits, starting with a letter

letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ’h’ ....

letter = [a-zA-Z]

Regular expressions

- a simple Identifier: string of letters or digits, starting with a letter

letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ’h’ ....

letter = [a-zA-Z]

identifier = letter (letter | digit)

Regular expressions

- a simple Identifier: string of letters or digits, starting with a letter

letter = ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ’h’ ….

letter = [a-zA-Z]

identifier = letter (letter | digit)*

Regular expressions

- Regular expressions are a concise way of specifying the valid forms of tokens in a language

  letter_ → [A-Za-z_]
  digit → [0-9]
  identifier → letter_ (letter_ | digit)*

In ANTLR, scanner rules are regular expressions. The left-hand side (LHS) of a rule **must** start with a capital letter

- There are many different forms of regular expressions
  – All have similar structures and operators

RE Operators

- Three basic operators:
  - * (known as *Kleene closure*) binds tightest, means *zero or more instances* of the symbol (or expression)
  - **Concatenation** is second tightest binding
  - **|** (alternation) has weakest binding
- REs can contain the empty string (ε)
- Sometimes include convenience operators:
  - **+** similar to * but one or more instances
  - **Range** [0-9] is equivalent to (0|1|2|3|4|5|6|7|8|9)
  - **?** zero or one instance

RE Examples

- (a|b)*abb
  - All strings of **a** and **b** that end in **abb**
- (aa* | bb*)
  - All strings consisting entirely of **a** or entirely of **b**
- [A-Z][a-z]*
  - All strings that start with an uppercase letter, followed by zero or more lowercase letters

Creating REs

- Write a regular expression for an integer number of metric distance units (take into account only millimetres, centimetres, metres and kilometres)
  - 0|([1-9][0-9]*)(m|c|k)?m
- Write a regular expression that describes floating point numbers, e.g. 2.76, -5., .42, 5e+4, 11.22e-3
  - ‘-’?(0|([1-9][0-9]*))?’.’[0-9]*(e(‘+’|’-’) [1-9][0-9]*)?

Creating REs

- Write a regular expression for an integer number of metric distance units (take into account only millimetres, centimetres, metres and kilometres)
  - 0|([1-9][0-9]*)(m|c|k)?m   e.g. 0m or 15km or 333mm, etc.
- Write a regular expression that describes floating point numbers, e.g. 2.76, -5., .42, 5e+4, 11.22e-3
  - ‘-’?(0|([1-9][0-9]*))?’.’[0-9]*(e(‘+’|’-’) [1-9][0-9]*)?

2 or 20 or 33, etc.

m or mm or cm or km

- Write a regular expression for an integer number of metric distance units (take into account only millimetres, centimetres, metres and kilometres)
  - 0|([1-9][0-9]*)(m|c|k)?m
- Write a regular expression that describes floating point numbers, e.g. 2.76, -5., .42, 5e+4, 11.22e-3
  - ‘-’?(0|([1-9][0-9]*))?’.’[0-9]*(e(‘+’|’-’) [1-9][0-9]*)?

Finite Automata (FA)

- To implement a scanner, a regular expression is translated into a finite automaton, which in turn is implemented as a program
  - This is what tools like *lex* or *ANTLR* do. Here we are looking at the theory of how this is done.

Example:

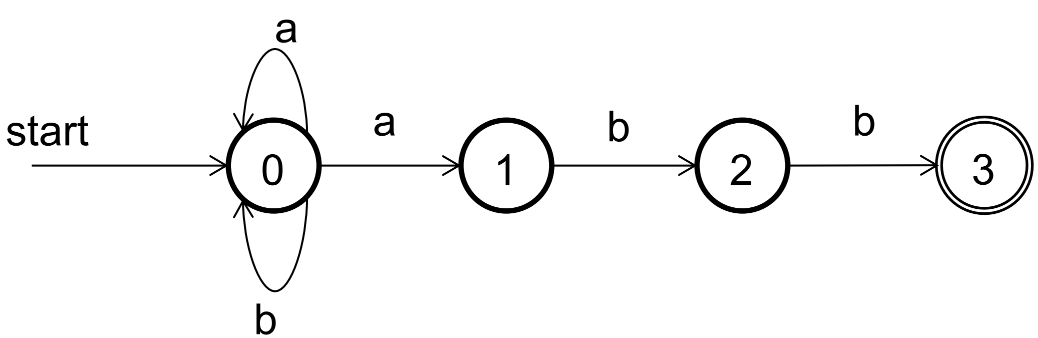
- RE: (a|b)*abb

aaabb
babb
abb
abababab b

Example:

- RE: (a|b)*abb

aaabb
babb
abb
abababb



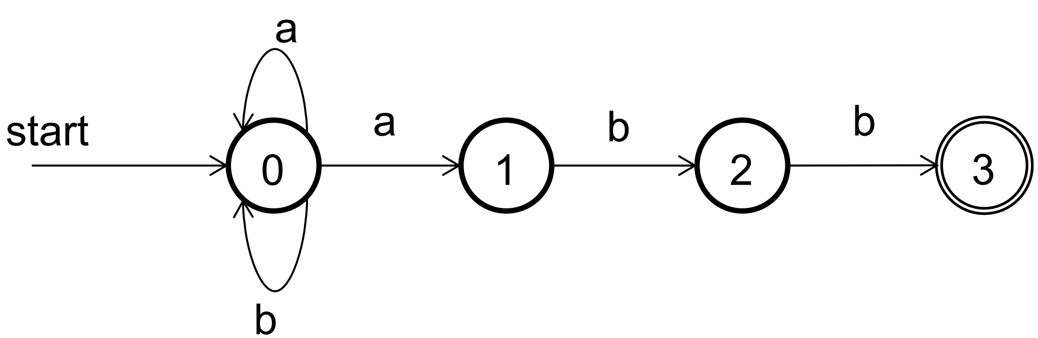
Nondeterministic Finite Automata (NFA)

- A *nondeterministic finite automaton* (NFA) consists of
  - A finite set of states *S*
  - A set of input symbols ∑
  - A state s_0, the *initial state*
  - A set of states *F*, the *final states*
- It is fairly straightforward to generate a NFA from a regular expression.

  (We'll look at the details of *how* you do this next week.)

Example:

- RE: (a|b)*abb
- NFA:



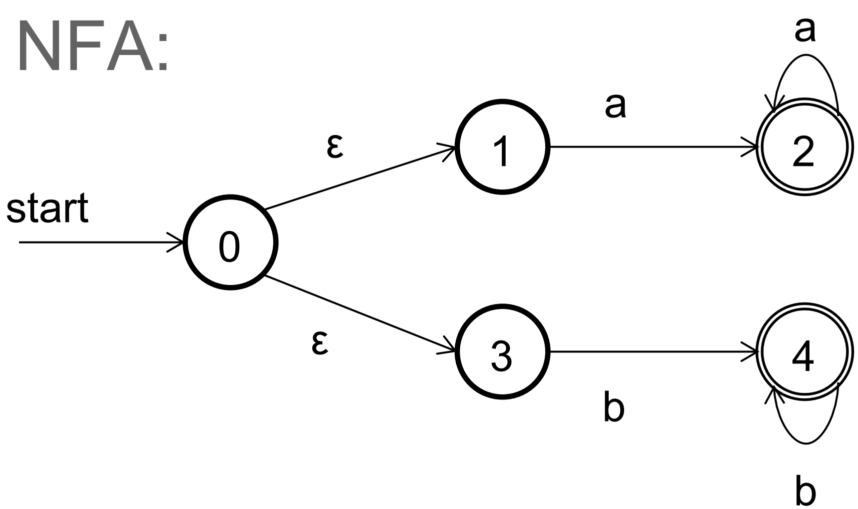
Important Features:

- *Start state is clearly indicated*
- *Edges are labelled to show the symbols that they consume*
- *Each state has a unique label*
- *End state (acceptance state) is clearly labelled (by the double circle on this state)*

Example:

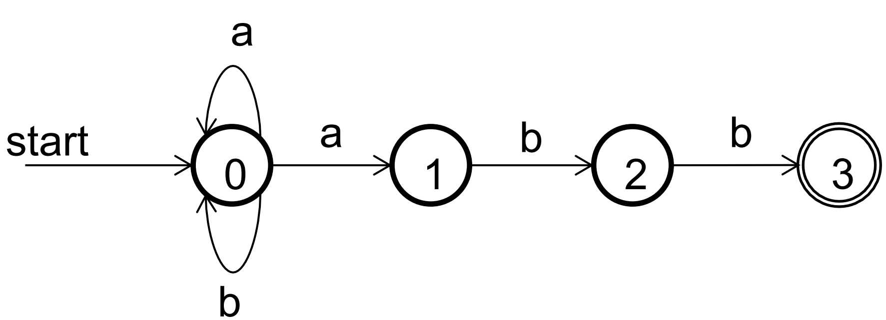
We'll look in detail at how you go about constructing NFA in next week's lecture

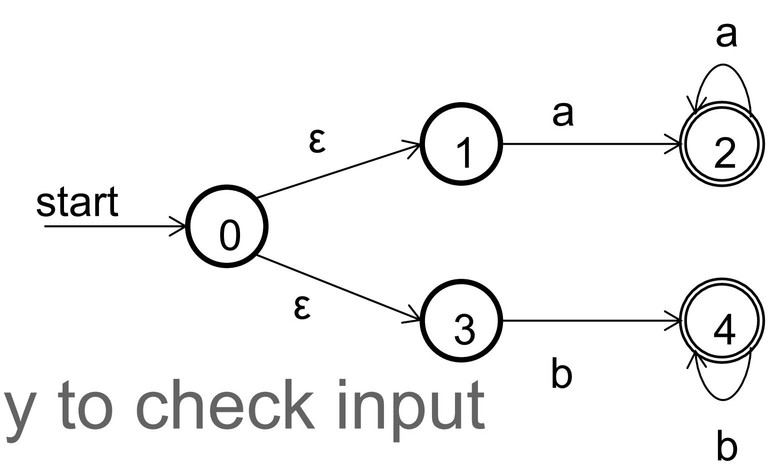
- RE: (aa* | bb*)
- NFA:



The Problem with NFAs

- The problem with NFAs is the non-determinism





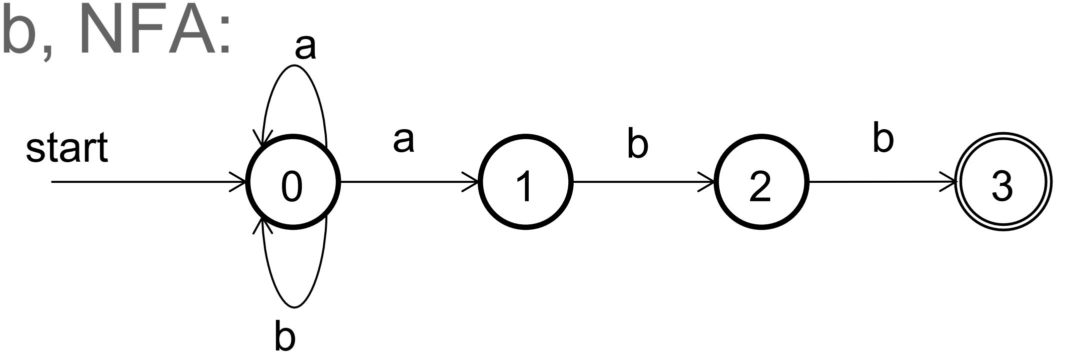
- Program needs a better way to check input

Deterministic Finite Automata (DFA)

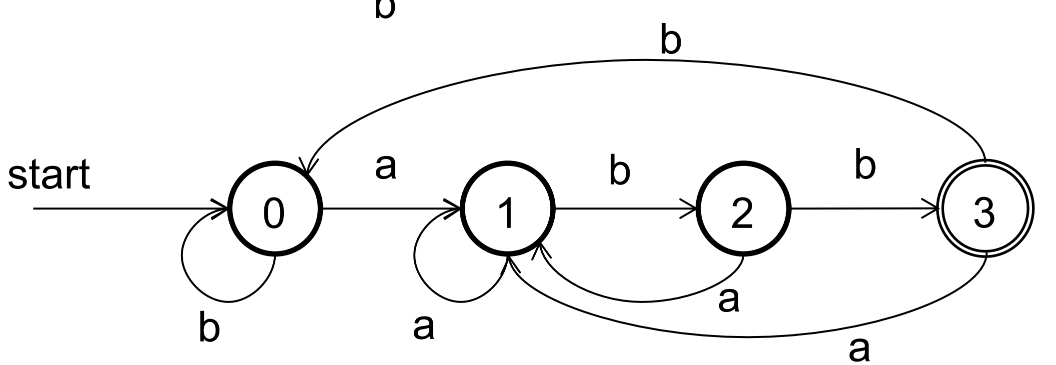
- Special case of an NFA:
  - There are no moves on input ε
  - For each state *s* and input symbol *a*, there is at most one edge out of *s* labeled *a*
- For every NFA there is an equivalent DFA
- Every DFA *is* a NFA

Example:

- RE: (a|b)*abb, NFA:

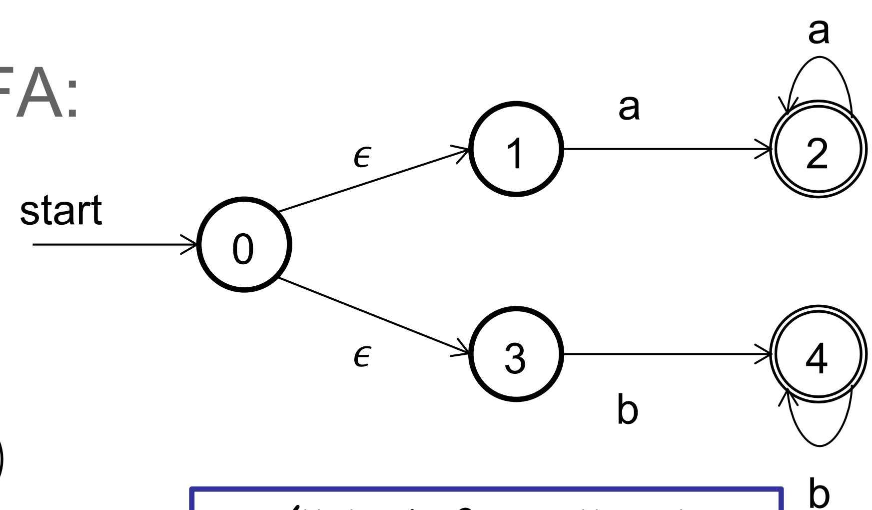


- DFA:

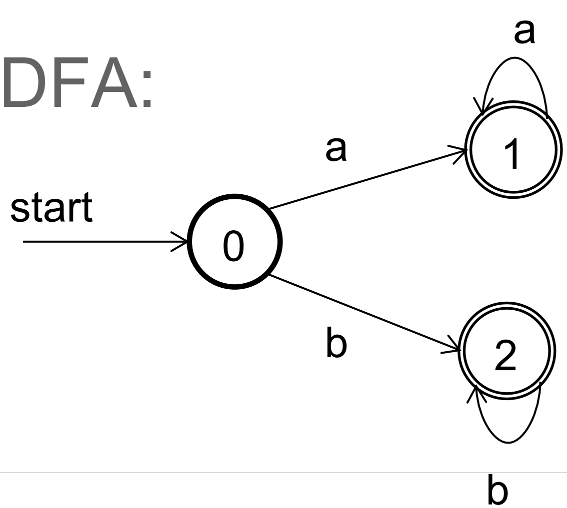


Example:

- RE: (aa* | bb*), NFA:



- DFA:

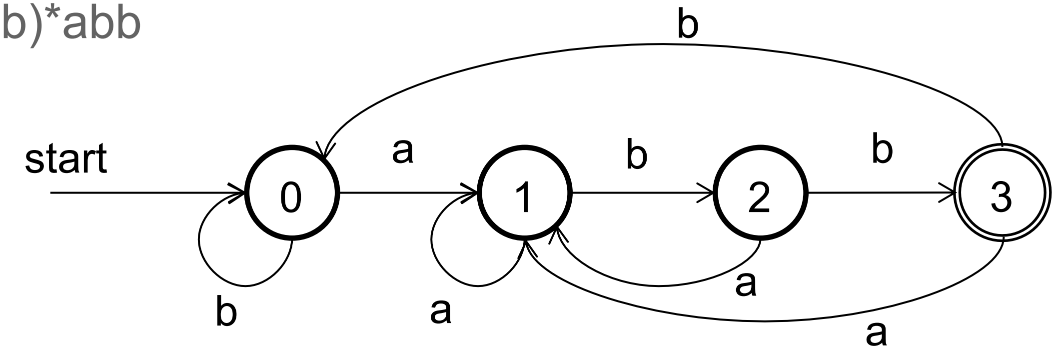


We'll look formally at how you convert from NFA to DFA next week.

Table-driven scanning

- A DFA can be represented as a table. This is a quick and efficient way to encode the DFA.

(a|b)*abb



	a	**b**	**other**
$\mathbf{s_0}$	s_1	s_0	s_e
$\mathbf{s_1}$	s_1	s_2	s_e
$\mathbf{s_2}$	s_1	s_3	s_e
$\mathbf{s_3}$	s_1	s_0	s_e

- Input is accepted *iff* it ends in final state (s_3)
- Transitions on other inputs go to error state

Summary

- The purpose of a lexical analyser is…
- A *token* is…
- Regular expressions are made up of…
- A NFA is…
- A DFA is…

Where we are…

- ~~Admin and overview~~
- **Lexical analysis**
- Parsing
- Semantic analysis
- Machine-independent optimisation
- Code generation
- Hardware architectures
- Machine-dependent optimisation
- Review