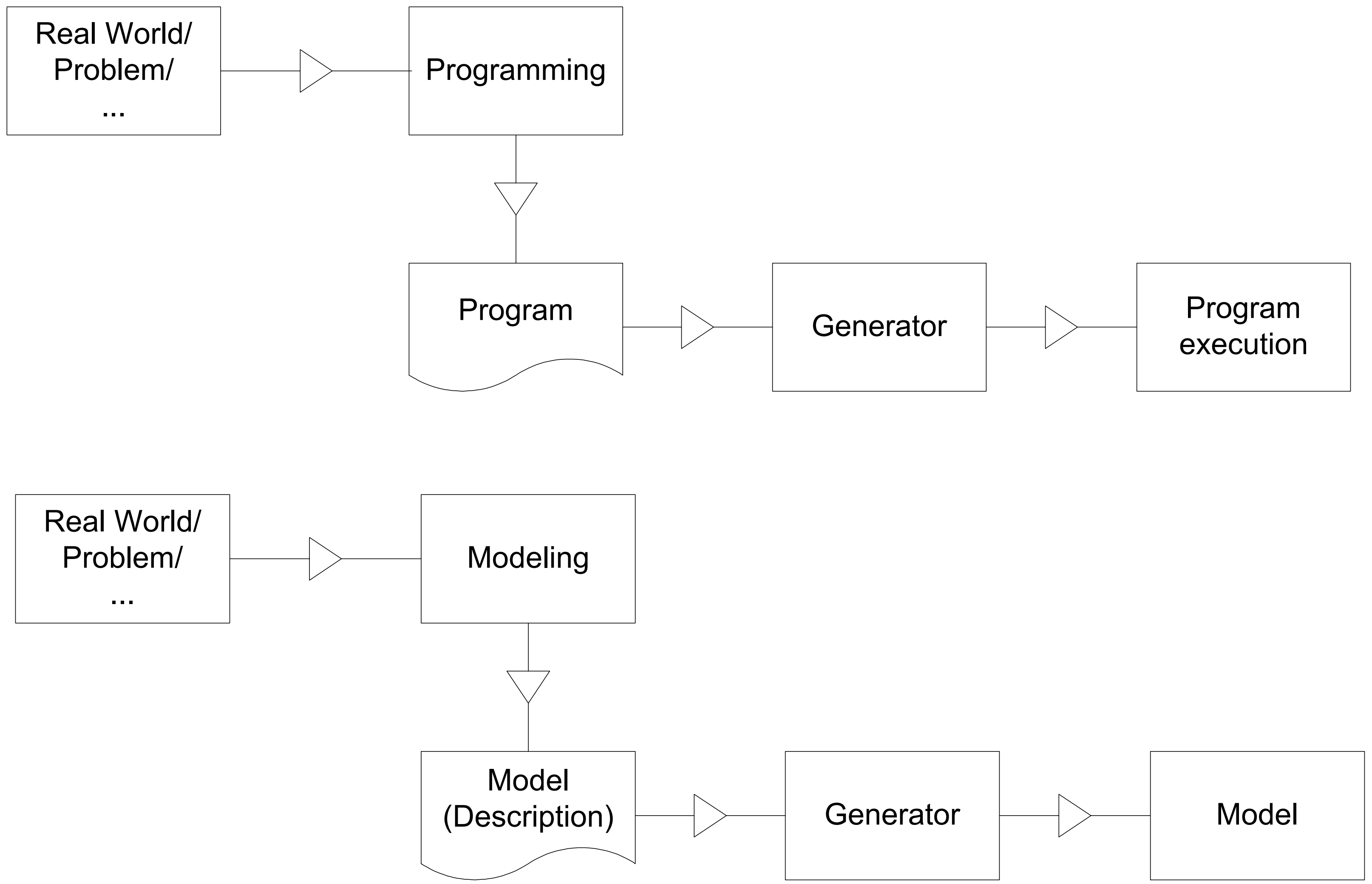
Syntax/semantics - I

- Program <> program execution
- Compiling/interpretation
- Syntax
  - Classes of languages
  - Regular languages
  - Context-free languages
- Meta models

Programming/Modeling

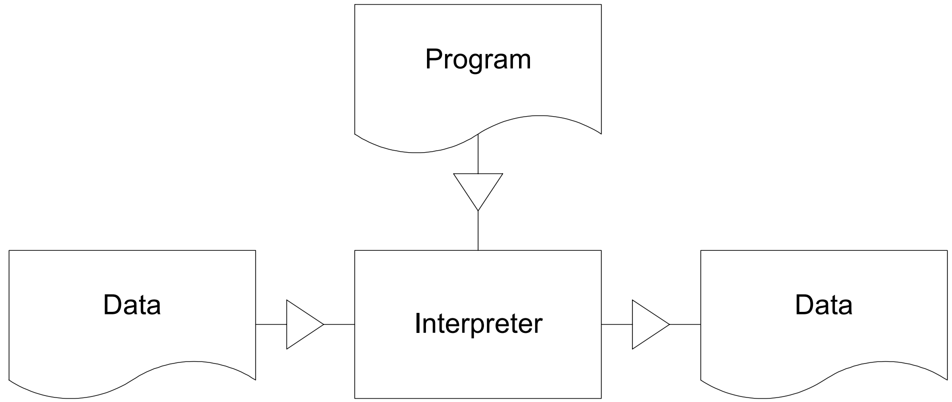


Syntax <> Semantics

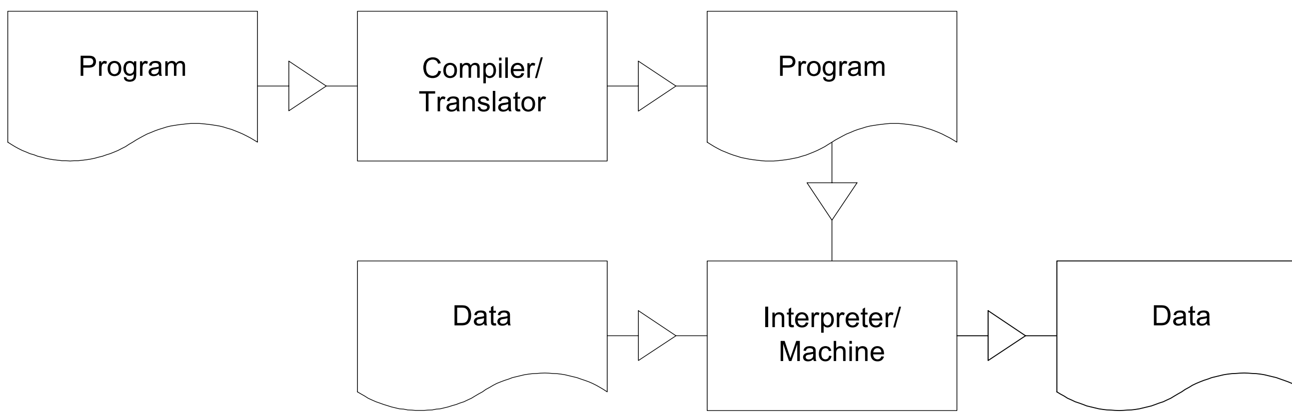
- A description of a programming language consists of two main components:
  - *Syntactic* rules: what *form* does a legal program have.
  - *Semantic* rules: what the sentences in the language *mean*.
    - *Static* semantics: rules that may be checked (by the compiler) *before* the execution of the program, e.g.:
      - All variables must be declared.
      - Declaration and use of variables coincide (type check).
    - *Dynamic* semantics: rules saying what shall hapen *during (as part of)* the execution of the program, e.g. in terms of an operational semantics, that is a semantics that describes the behaviour of a (idealised) abstract generator (prosessor/machine) executing a program.

Compiling/interpretation

- An interpreter reads a program and executes its operations.



- A compiler/ translator translates a program to another language, typically a machine language or to a language for a virtual machine.



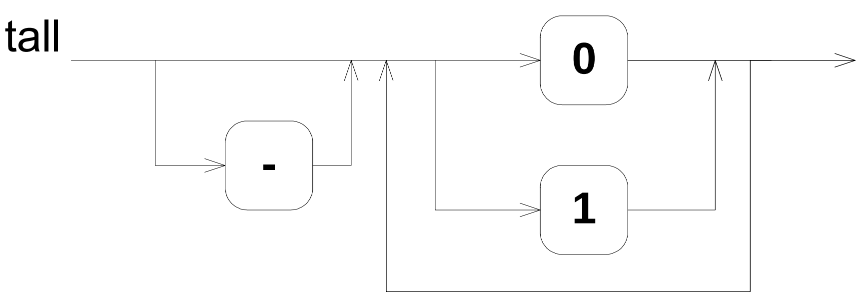
INF 3110/4110 - 2006

How to describe syntax – I

- BNF-grammar

<tall> → - <siffer>
<tall> → <sifffer>
<siffer> → 0 <sifre>
<siffer> → 1 <sifre>
<sifre> → 0 <sifre>
<sifre> → 1 <sifre>
<sifre> → ε

- Syntax diagram
  (‘railway diagram’)



Extended BNF

- Extended BNF has the following metasymbols (on the right side):

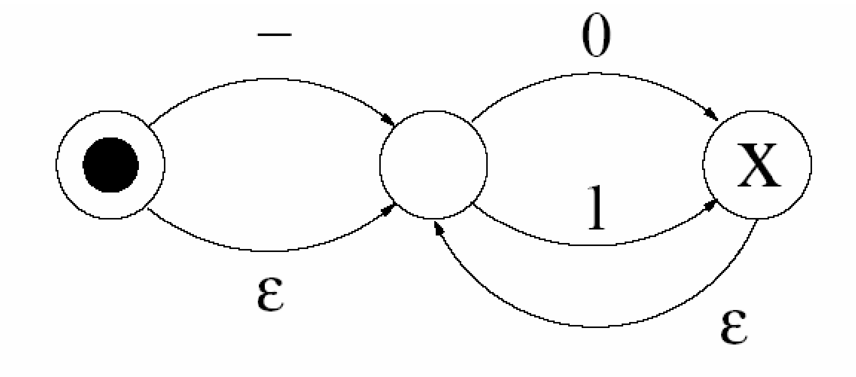
|	alternatives	
?	symbols may occur	0 or 1 time
*	symbols may occur	0 or several times
+	symbols may occur	1 or several times
{...}	groups symbols	

<tall> → - <siffer> | <sifffer>
<siffer> → 0 <sifre> | 1 <sifre>
<sifre> → 0 <sifre> | 1 <sifre> | ε

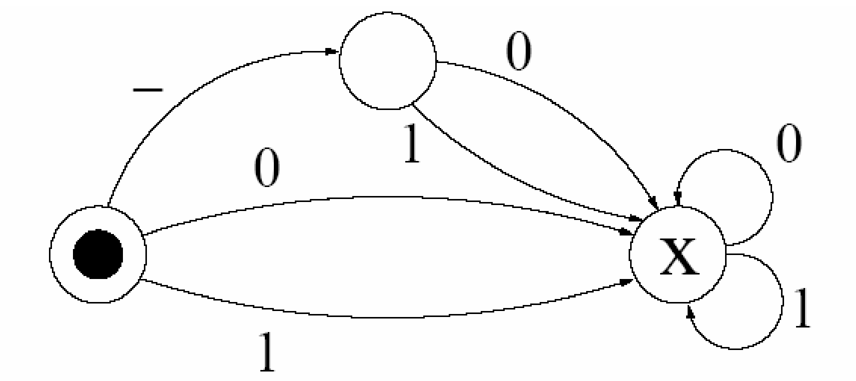
<tall> → {-}?{0|1}+

How to describe syntax – II

- Non-deterministic automaton



- Deterministic automaton



Example

- BNF-grammar

<uttrykk> ⟶ <uttrykk> + <term> | <term>

<term> ⟶ <term> * **navn** | **navn**

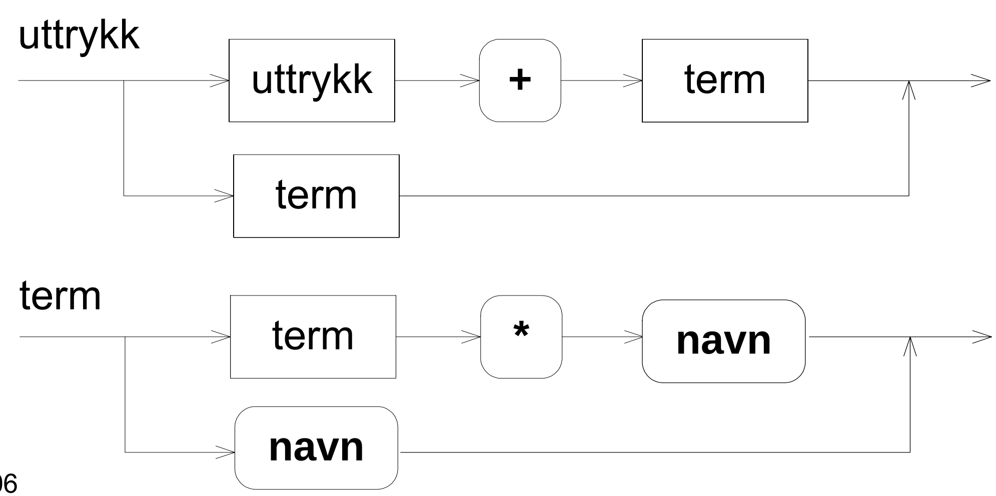
production, rule (produksjon)

left-recursion

non-terminals (metasymbol)

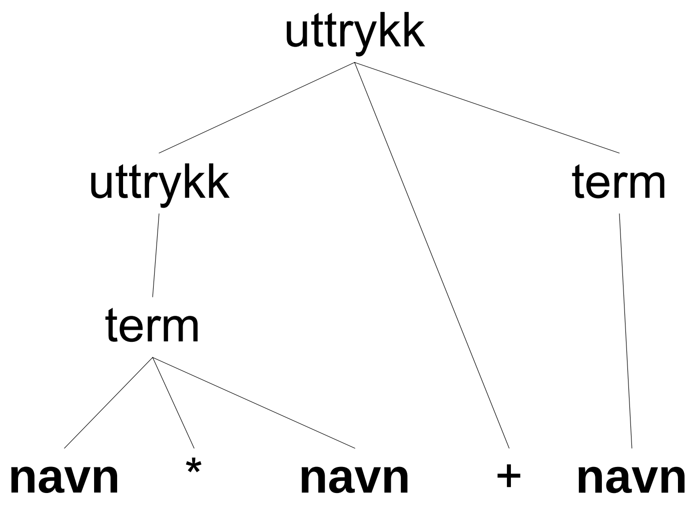
terminals (grunnsymbol)

- Syntax diagram



Derivation of sentences

- The sentences in a language defined by a BNF-grammar are exactly those that may be produced by the following procedure:
  1. Start with the start symbol.
  2. For each meta symbol, substitute this with one of alternatives on the right hand side in the production rule.
  3. Repeat step 2 until only terminals.
- This is called a *derivation* from the start symbol to a complete sentence, and it may be represented by a *syntax tree*

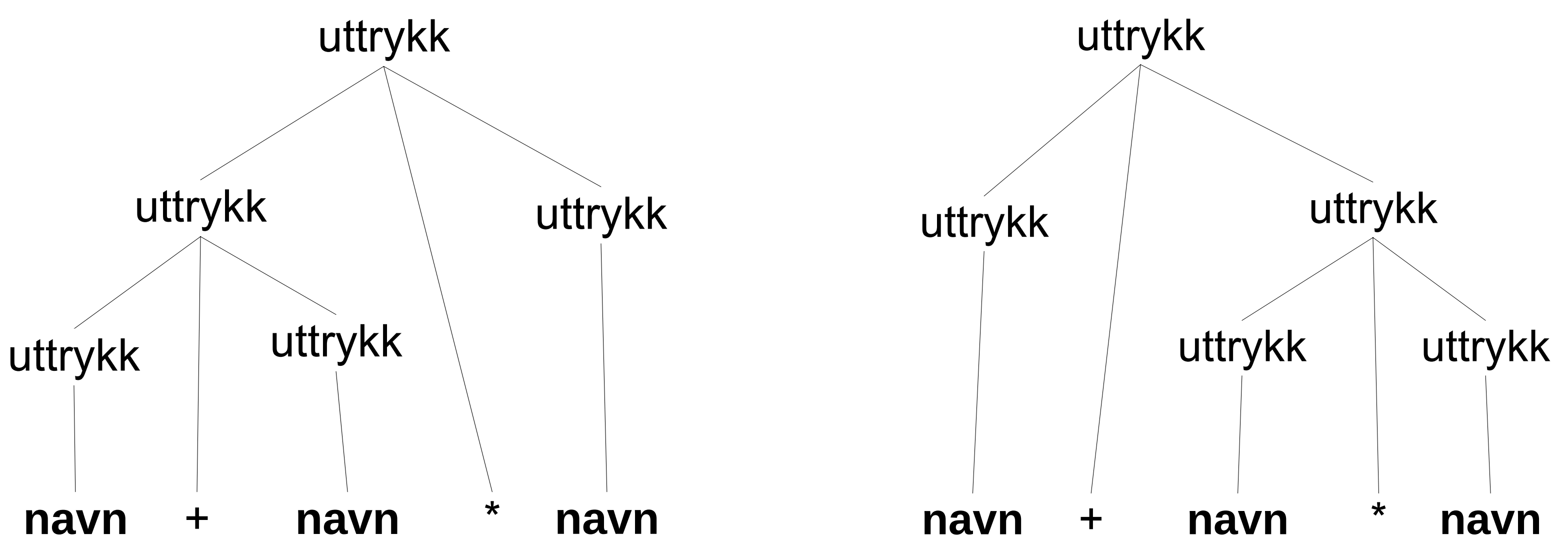


- *Abstract syntax tree*
  - Removing terminals that are not needed in order to give the meaning

Unambiguous/ambiguous grammars

- If every sentence in a language can be derived by one and only one syntax tree, then the grammar is *unambiguous*,
otherwise it is *ambiguous*.

<uttrykk> ⟶ **navn** |
<uttrykk> + <uttrykk> |
<uttrykk> * <uttrykk>

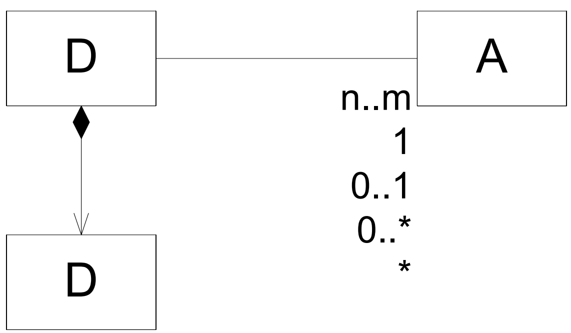


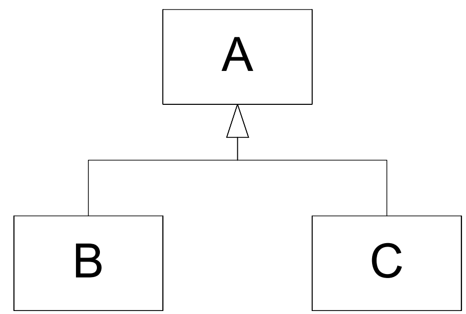
Types of languages

- Regular languages (type 3)
  - A BNF-grammar with one meta symbol to the left and only terminals to the right, possibly with a metasymbol as the last symbol
  - May be analyzed with non-deterministic and deterministic automata
- Context free languages (type 2)
  - A BNF-grammar with just one meta symbol on the left hand side
  - Almost all programming languages have grammars of this type
  - May be analyzed with parsers
- Type 1 languages («context-sensitive») require that the righthand side is of the same length as the lefthand side. This makes it possible to cover name bindings and type checks
- Type 0 languages have no restrictions
  - One of theoretical interest

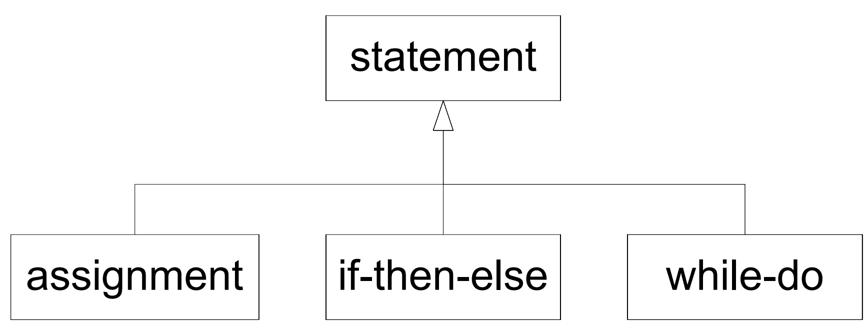
Meta models

- Alternative to grammars and syntax trees
- Object model representing the program (*not* the execution)





<statement> → <assignment> | <if-then-else> | <while-do>

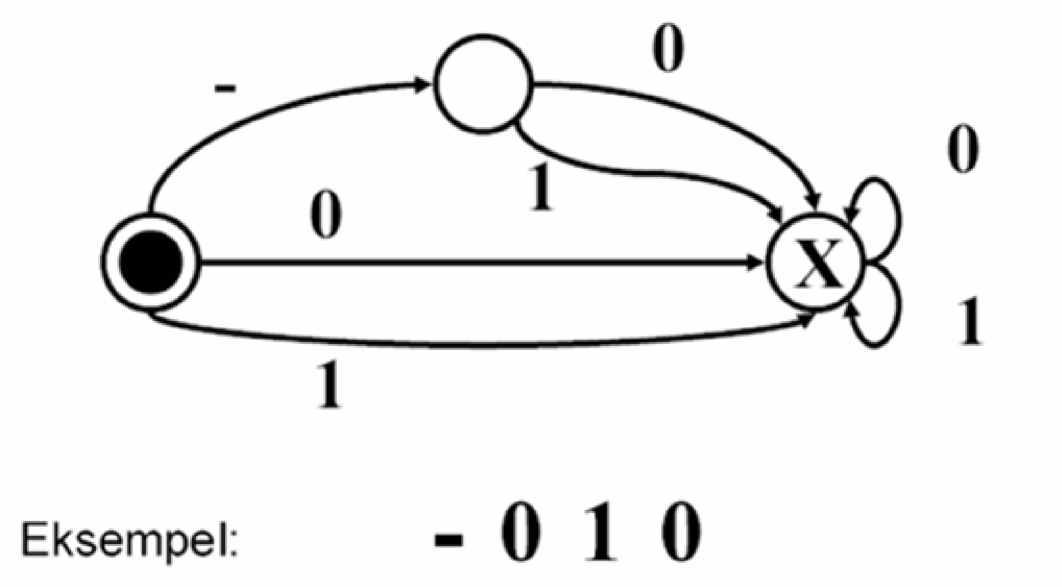


Why meta models?

- Inspired by abstract syntax trees in terms of object structures, interchange formats between tools
- Not all modeling/programming tools are grammar (parser)-based (e.g. wizards)
- Growing interest in domain specific languages, often with a mixture of text and graphics

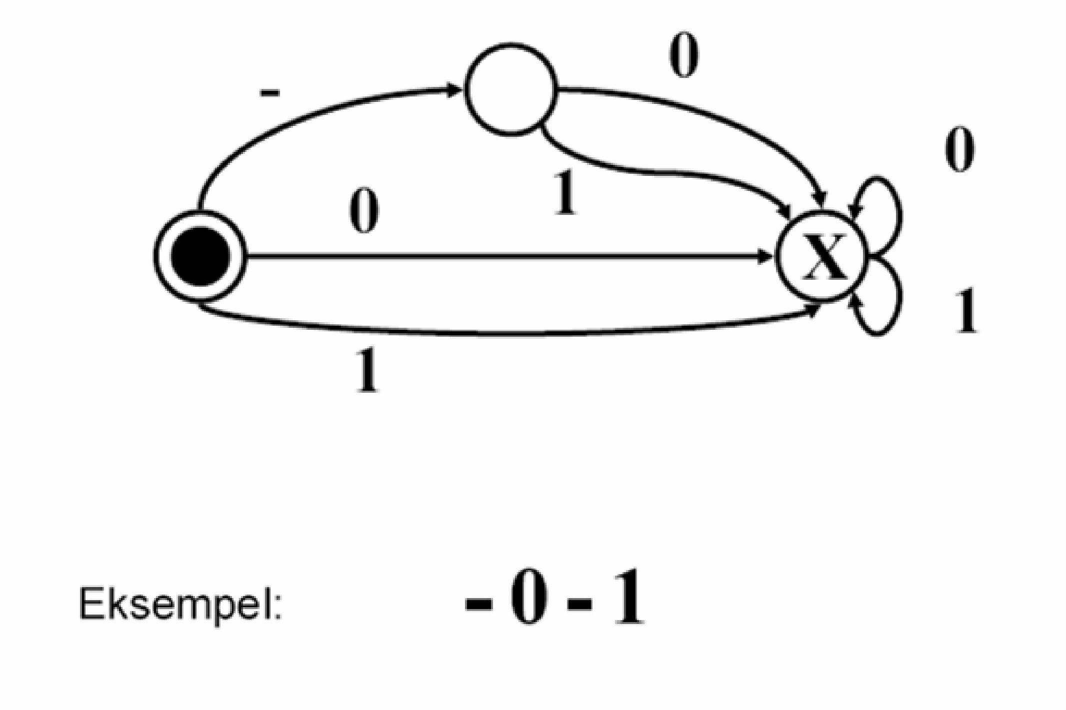
Use of deterministic automata

- To check if a given string is part of the regular language or not:



Use of deterministic automata

- What if the string is not in the language?



Howe to make a deterministic automaton?

- A deterministic automaton (D-automat) is easy to use, but not necessarily so intuitive and not so easy to make.
- From a regular expression (or a syntax diagram) it is, however, easy to make a *none-deterministic* automaton (ID-automat).

May have:

- Empty transitions (so-called ε - transitions).
- More transitions from same state with the same symbol.

- Then we can use an algorithm to make a deterministic automaton from the *none-deterministic* automaton.

Example

<tall> ⟶ **0** <FP> | **1** <IFP>

<IFP> ⟶ **1** <IFP> | **0** <IFP> | <FP>

<FP> ⟶ ε | **.** <EP>

<EP> ⟶ **0** | **1** | **0** <EP> | **1** <EP>

<tall> ⟶ { **0** | **1** {**0** | **1** }* } {**.**{ **0** | **1** }+ }?

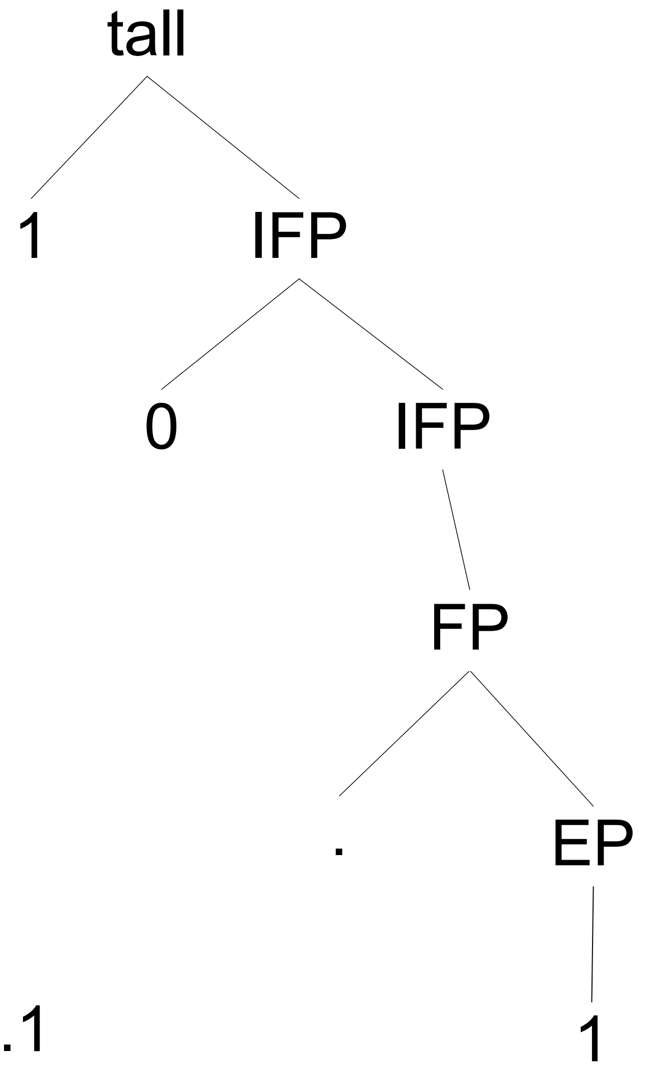
Allowed words are

0 1 101 0.10 100.1010 10.1

However, not allowed with leading 0 or ”decimalpoint” without preceeding or following ciffers, so the following is not allowed:

001 10. .01

Parse/syntax tree

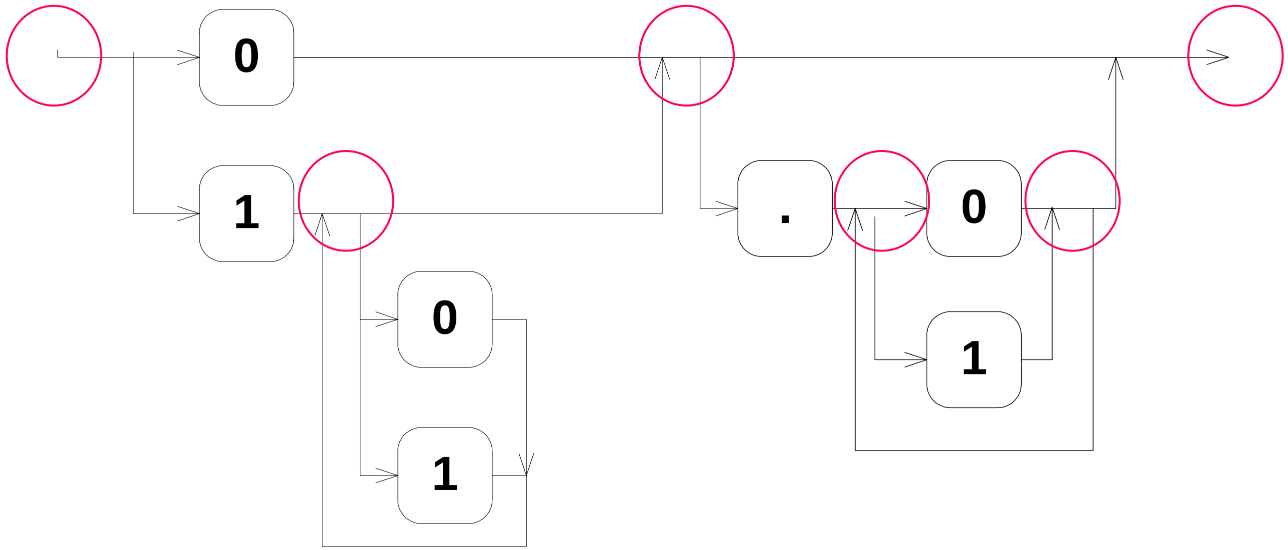


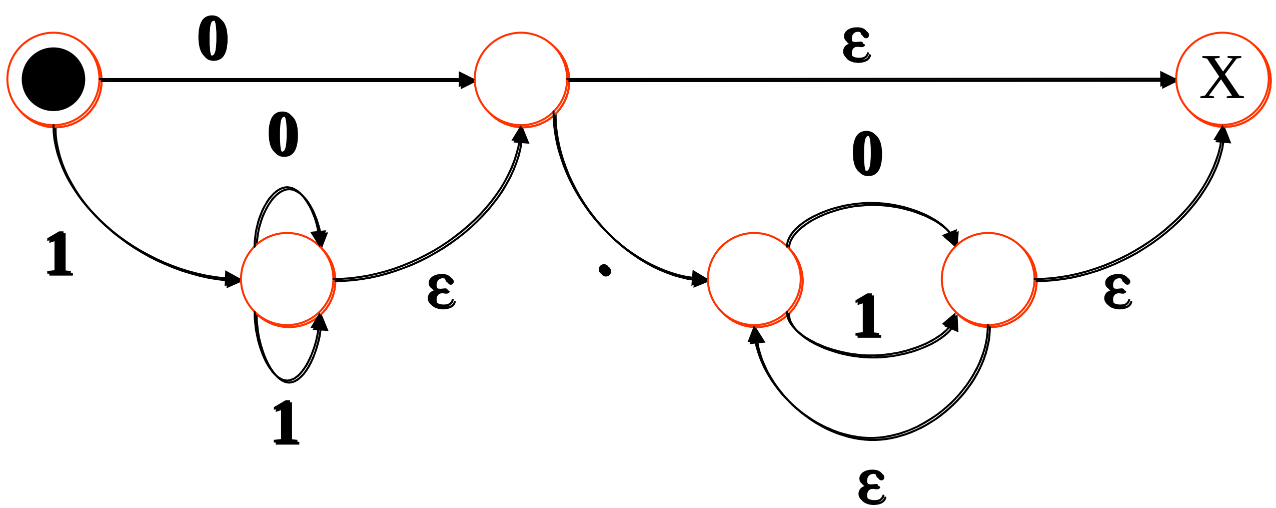
10.1

From syntax diagram to none-deterministic automaton

1. Every ”switch” becoms a node in the automaton
2. The lines (with symbols) become transitions between the nodes
   Some transitions may get an empty symbol (ε)
3. Mark start node and end node(s)

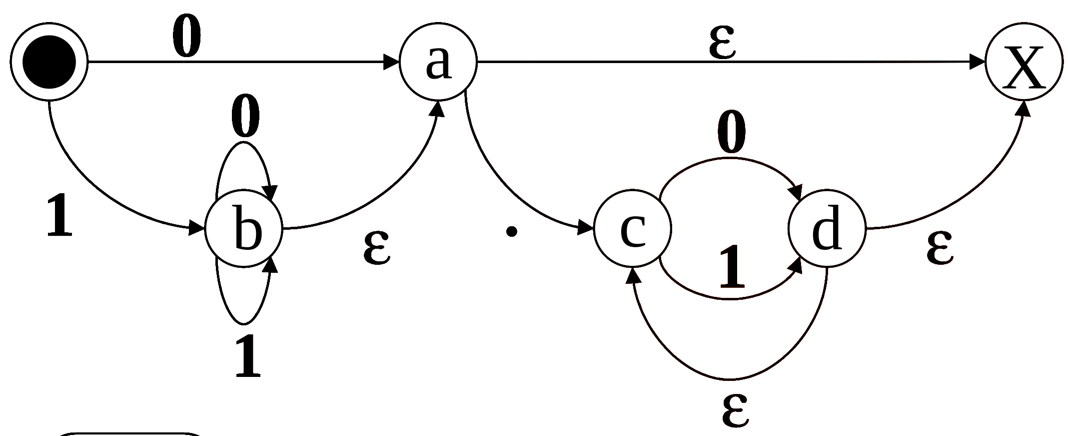
Example



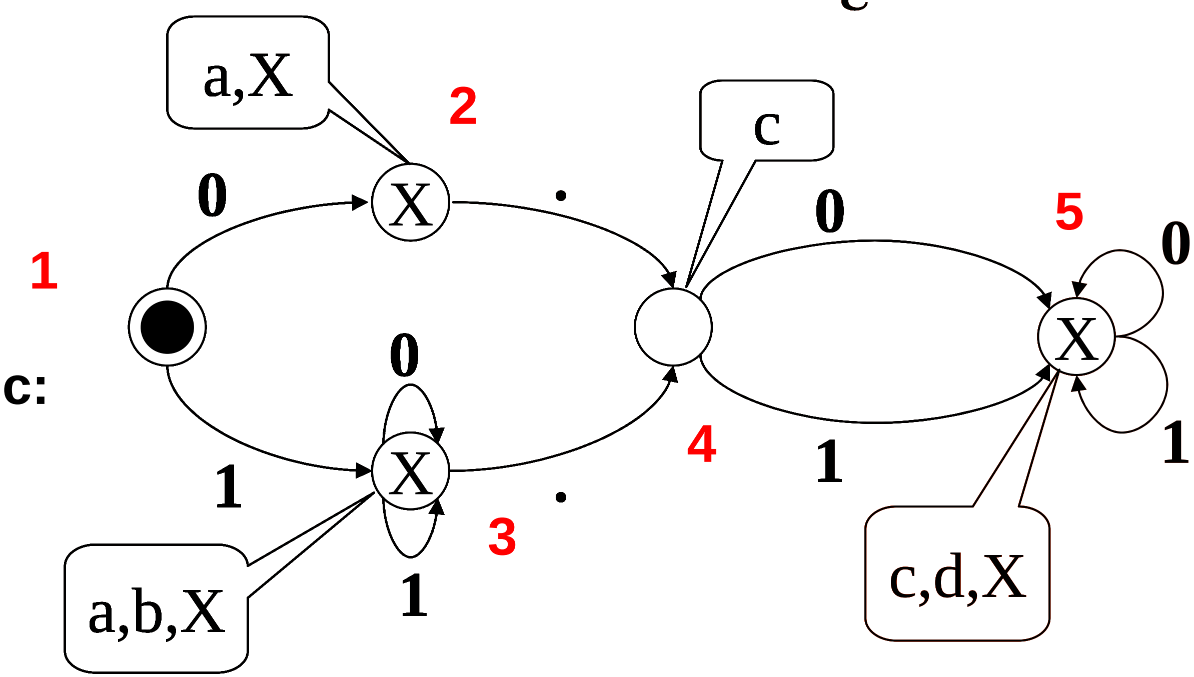


From non-deterministic to deterministic automaton

non-deterministic:



deterministic:



As a table

	1	2	3	4	5	error
0	2	error	3	5	5	
1	3	error	3	5	5	
.	error	4	4	error	error	
end		ok	**ok**		ok	

Syntax checking algorithm

- Given such a table t, the following algorithm will check a given string:

```
state := 1;
while <more symbols> do begin
     c := <next symbol>;
     state  := t(state,c);
end while;
if ok(state) then <OK>
else <Not OK>;
```

- Summary - How to make a syntax checking program:
  - Make a non-deterministic automaton for the regular expression
  - Make a deterministic automaton from the non-deterministic automaton
  - Make the table t
  - Use the above algorithm