Lexical Analysis by Finite Automata

4DV006 – Compiler Construction

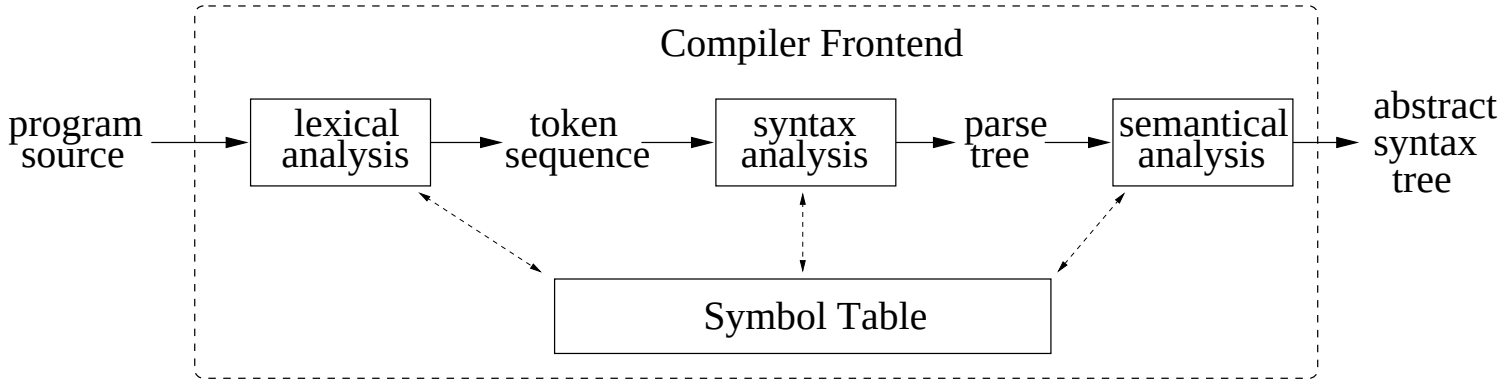
Dr Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

Slides are available in Moodle

26 oktober 2014

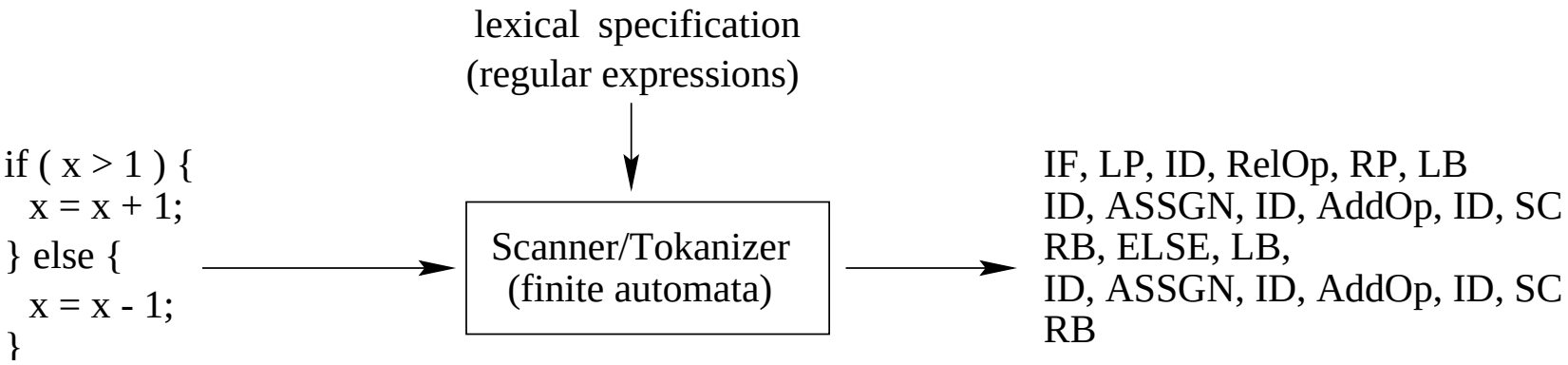
Frontend Overview



- **Lexical Analysis:** Identify atomic language constructs.
  Each type of construct is represented by a token.
  (e.g. `3.14` $\mapsto$ `FLOAT`, `if` $\mapsto$ `IF`, a $\mapsto$ `ID`).
- **Syntax Analysis:** Checks if the token sequence is correct with respect to the language specification.
- **Semantical Analysis:** Checks type relations + consistency rules.
  (e.g. if $type(lhs) = type(rhs)$ in an assignment $lhs = rhs$).

Each step involves a transformation from a program representation to another.

Lexical Analysis Overview



- Input program representation: Character sequence
- Output program representation: Token sequence
- Analysis specification: Regular expressions
- Recognizing (abstract) machine: Finite Automata
- Implementation: Finite Automata

Lexical Analysis Specification

Token	Patterns	Action
WS	$(blank\|tab\|newline)+$	skip
...	...	...
IF	if	`genToken();`
...	...	...
RelOp	$< \mid <= \mid = \mid >= \mid >$	`genToken(); addAttr();`
...	...	...
ID	$[a-zA-Z][a-zA-Z1-9]*$	`genToken(); updateSymTab();`
...	...	...

- In theory: $(p_1|p_2|...|p_n)*$, where p_i are the above patterns, defines all lexically correct programs. $\Rightarrow$ the set of lexically correct programs is a regular language.
- In practice: we recognize each pattern individually.
- This type of specification is input to AntLR in Practical Assignment 1.

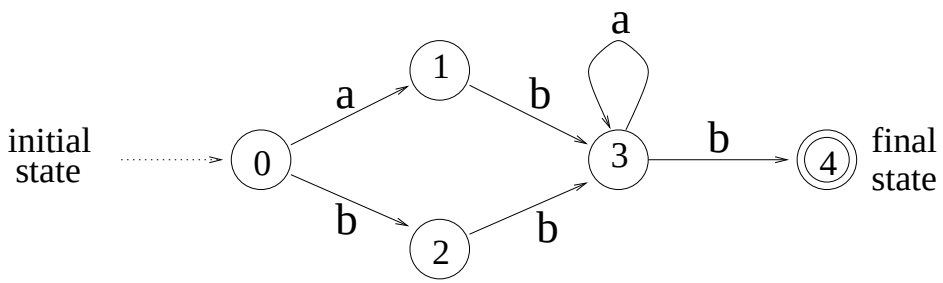
An AntLR Lexical Specification

- AntLR is a compiler generator tool
- It reads a specification and generates a parser.
  (Actually, it generates Java classes implementing a parser.)
- It will be used in the practical assignment
- Below: A very simple lexical specification

```
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
MULT: '*';
ASSIGN: '=';
PLUS: '+';
MINUS: '-';
```

String recognition using Finite Automata

A finite automaton is an abstract machine that can be used to identify strings specified by regular expressions.



- The above finite automata recognizes the pattern $(a|b)ba^*b$.
- Every input character (*a* or *b*) causes a *transition* from one *state* to another.
- A string is *accepted* if we end up in a *final state* once every character been processed.
- No possible transition $\Rightarrow$ *rejection* $\Rightarrow$ the string is not part of the language.

Finite Automata (FA)

- A *finite automata* M is a quintuple $M = \{Q, \Sigma, \delta, q_0, F\}$ where
  - Q is a finite set of states,
  - Σ is the input alphabet,
  - $\sigma : Q \times \Sigma \mapsto Q$ is the *transition function*,
  - $q_0 \in Q$ is the *initial state*,
  - F is the set of *final states*.
- The graph corresponding to a given FA is called the *transition graph*.
- **Notice:** We can have many final states but only one initial state.

Two kinds of Automata

We have to kinds of finite automata:

1. **Deterministic FA (DFA):**
   - at most one possible transition for each input
   - no $\varepsilon-$ transitions

2. **Nondeterministic FA (NFA):** If not DFA, then NFA.

An $\varepsilon-$ transition indicates that we are in two states at the same time.

The following slides show one example of each type.

NFA accepting $(a|b)^*abb$

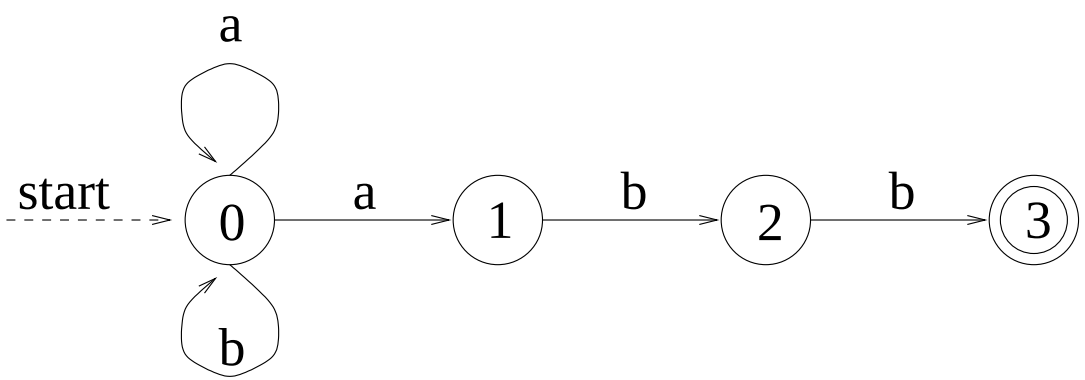
$M = \{Q, \Sigma, \delta, q_0, F\}$
$Q = \{0, 1, 2, 3\}$
$\Sigma = \{a, b\}$
$q_0 = 0$
$F = \{3\}$

δ	a	b
0	$\{0, 1\}$	$\{0\}$
1	$\emptyset$	$\{2\}$
2	$\emptyset$	$\{3\}$
3	$\emptyset$	$\emptyset$

Notice

- Multiple transitions possible for one input.
- Transition to $\emptyset \Rightarrow$ rejection.

Corresponding transition graph



An NFA *accepts* (or *recognizes*) an input string x iff there is some path from start to final such that the edge labels along this path spell out x.

DFA accepting $(a|b)^*abb$

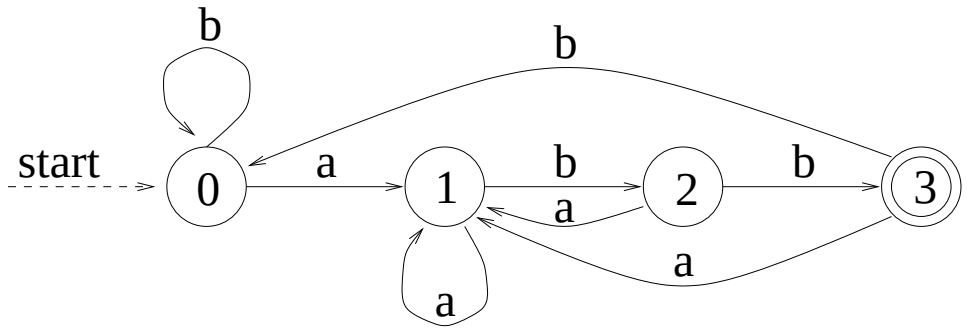
$M = \{Q, \Sigma, \delta, q_0, F\}$
$Q = \{0, 1, 2, 3\}$
$\Sigma = \{a, b\}$
$q_0 = 0$
$F = \{3\}$

δ	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Notice

- Only one possible transition for each input.
- No transition possible $\Rightarrow$ rejection.

Corresponding transition graph



A DFA *accepts* an input string x iff, after reading x, we are in a final state.

DFA simulation algorithm (Algorithm 1)

Item	Description
q_0	Initial state
F	Set of final states
nextChar()	Returns next symbol from the input string, stops with EOF.
move(q, c)	Returns the state to which their is a transition from q given input c.

$q := q_0$;
$c :=$ nextChar();
while $c \neq$ EOF **do**
  $q :=$ move(q, c);
  $c :=$ nextChar();
end while
if $q \in F$ **then**
  return "Accepted";
else
  return "Rejected";
end if

```
move(state,next) --> new_state
  switch (state) {
    case 0:
      if (next=a) return 1;
      else return 0;
    case 1:
      if (next=a) return 1;
      else return 2;
    case 2:
      if (next=a) return 1;
      else return 3;
    ...
```

Thus, implementing string recognition for a given DFA is straight forward.

Summary: Finite Automata

Known Theoretical Results

- **Kleene's Theorem:** A language is recognized by a FA iff it is regular
  $\Rightarrow$ For each regular expression their is a corresponding finite automata.
- For each NFA, there is a corresponding DFA.
- Given FA accepting strings defined by RE r, and an input string x.
  Simulation requires
  - **NFA:** Memory $O(|r|)$ and time $O(|r| \times |x|)$
  - **DFA:** Memory $O(2^{|r|})$ and time $O(|x|)$

  where $|r|$ is the number of symbols and operators in r.

Conclusion

- NFA's are easier to *construct* given an arbitrary regular expression.
- DFA *simulation* is both easier and faster.
- Hence, we construct an NFA, convert it into an DFA,
  and use the resulting DFA for simulation.

A Recipe for Scanner Construction

1. Identify what tokens t_i you are interested in.
2. For each token t_i, write a matching regular expression r_i.
3. Convert the regular expressions $r_1, r_2, ..., r_n$ to an NFA:s $N_1, N_2, \ldots, N_n$ using Algorithm 2
4. Put together the NFA:s $N_1, N_2, \ldots, N_n$ to a single NFA which final accepting states F corresponds to the above identified lexeme groups g_i.
5. Convert the NFA to a DFA using Algorithm 3.

The steps 3-5 are briefly presented in the following slides.
They are more thoroughly presented in the book by Aho, Ullman, and Sethi.

Example: Identifying INT, ID, and IF

Step 1 and 2: Lexical Specification

Token	Sample Patterns	Patterns
IF	if	if
INT	0, 78, 1367, 0067	d^+
ID	max, a , value33	$l(l\|d)^*$

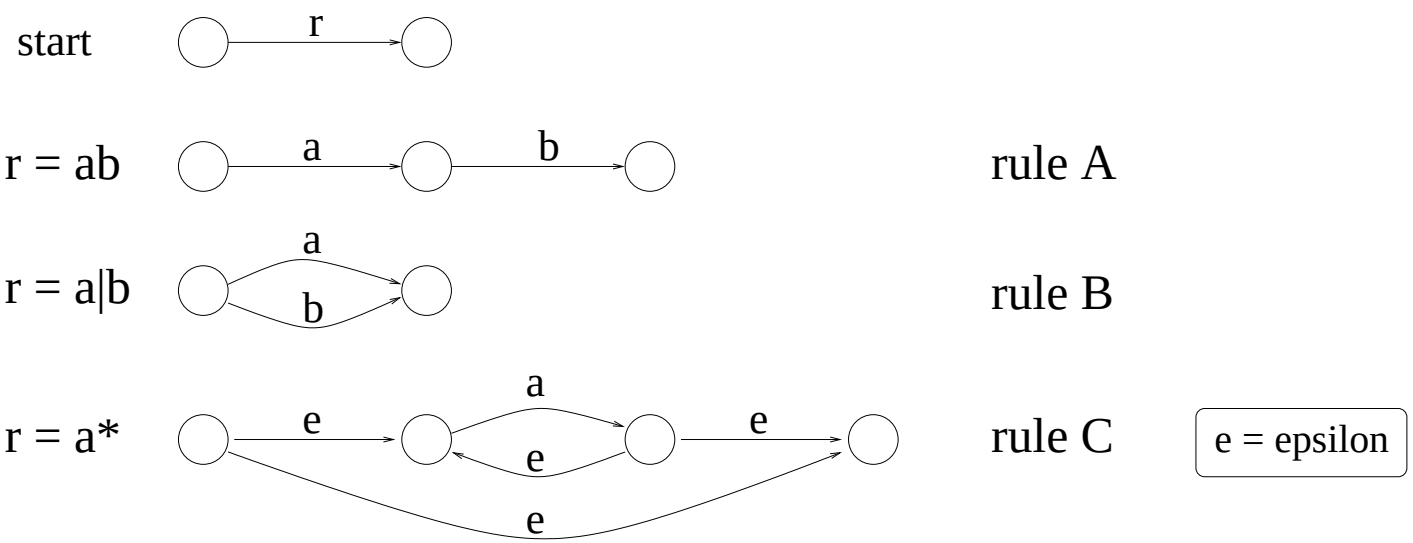
where d and l are character classes defined as:

- $d = [0 - 9]$
- $l = [a - zA - Z]$

Example available on course home page (see Reading Instructions)

Step 3: RegExp → NFA (Algorithm 2)

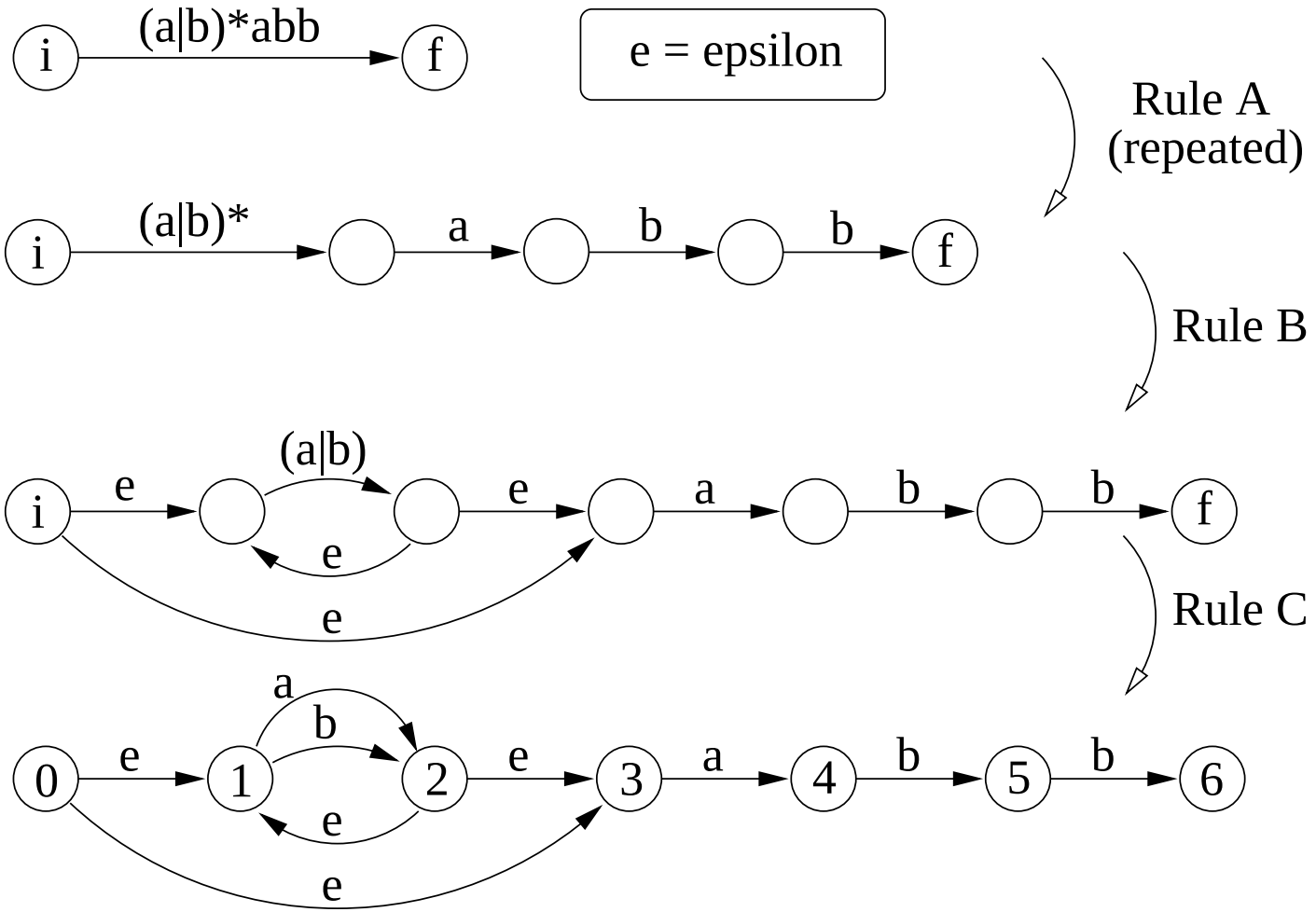
Input: A regular expression r over an alphabet Σ
Output: An NFA N accepting $L(r)$



- It is basically a top-down approach where each regular subexpression is replaced by a *part* of an NFA.
- The construction is straight forward but the result is often an ugly NFA that needs to be rewritten.

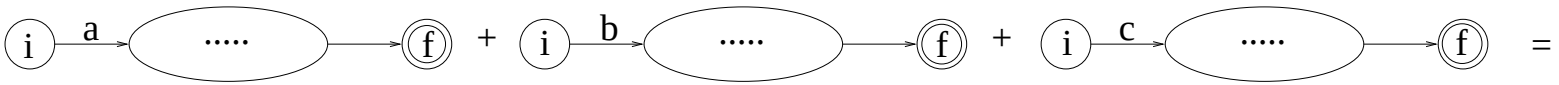
More information? See subsection "Construction of NFA from Regular Expression" in the book by Aho, Ullman, and Sethi.

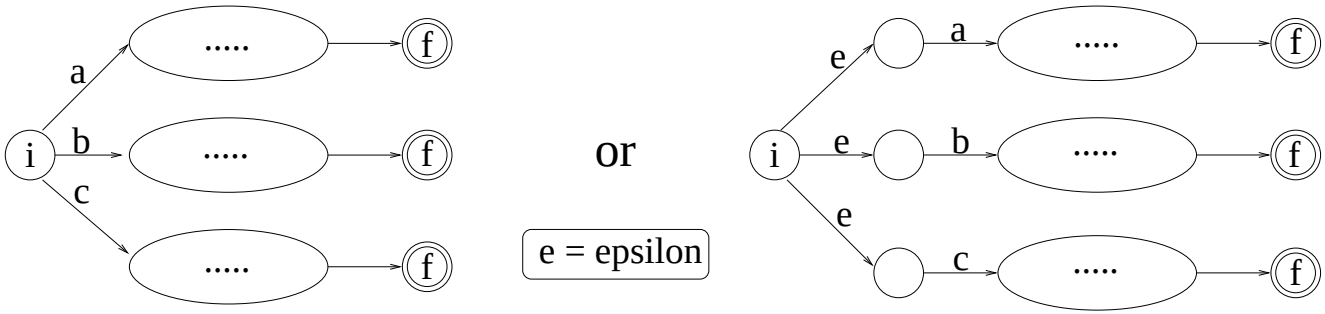
Alrorithm 2: $(a|b)^*abb$



Step 4: Adding NFA:s

Two possible, and equivalent, alternatives:





The new NFA accepts all the strings accepted by the three NFAs.

NFA $\Rightarrow$ DFA (Preparation)

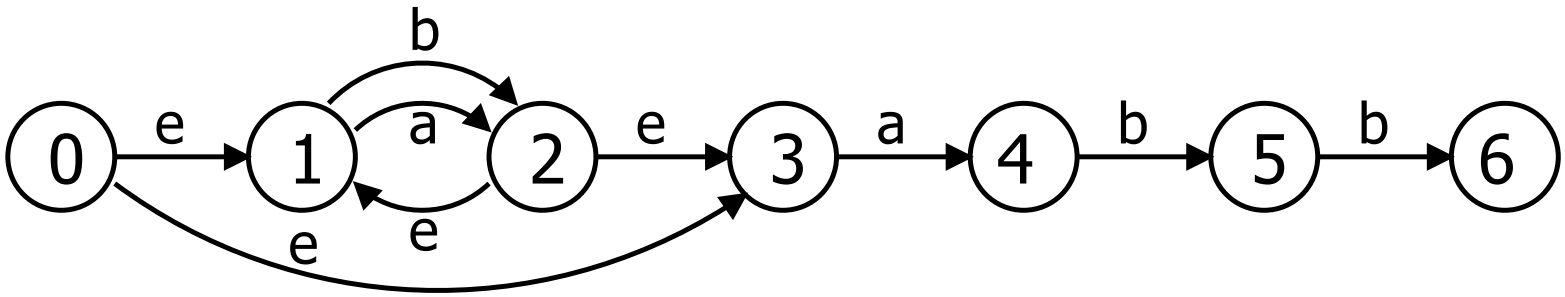
Now, assume that we have a single NFA recognizing all tokens we are interested in.

NFA $\Rightarrow$ DFA (Basic Idea)

- **Basic idea:** Set of NFA states $\Rightarrow$ new DFA state
- **Worst case:** N NFA states $\Rightarrow 2^N$ DFA states
  (Usually not the case)
- **Algorithm outline:**
  1. DFA state 0 = all ε - equivalent states of NFA 0
  2. Which states can be reached from DFA 0 on input $x, y, z, \ldots$?
     $\Rightarrow$ new DFA states 1, 2, 3, ...
  3. Repeat (2) with DFA states 1, 2, 3,... until no more DFA states are created.

Example: $(a|b)^*abb$

NFA

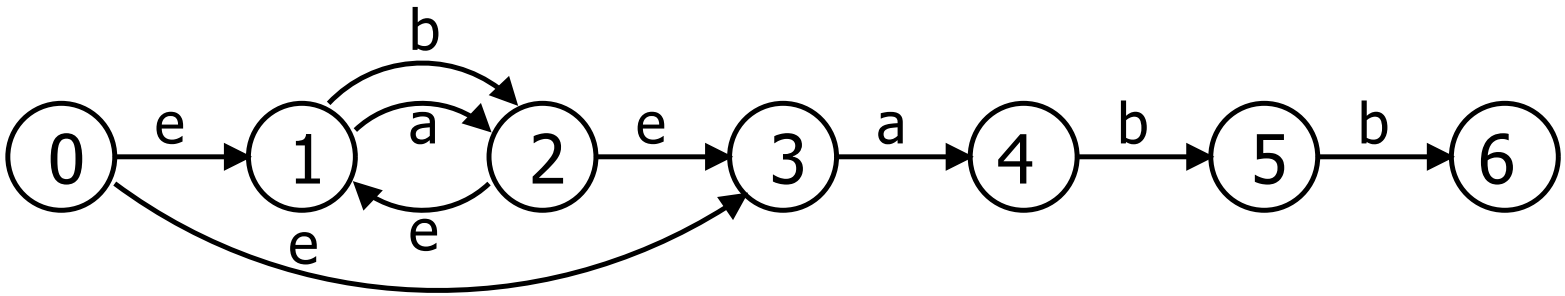


DFA Construction

DFA	NFA	a	b
0	{0,1,3}		

Example: $(a|b)^*abb$

NFA

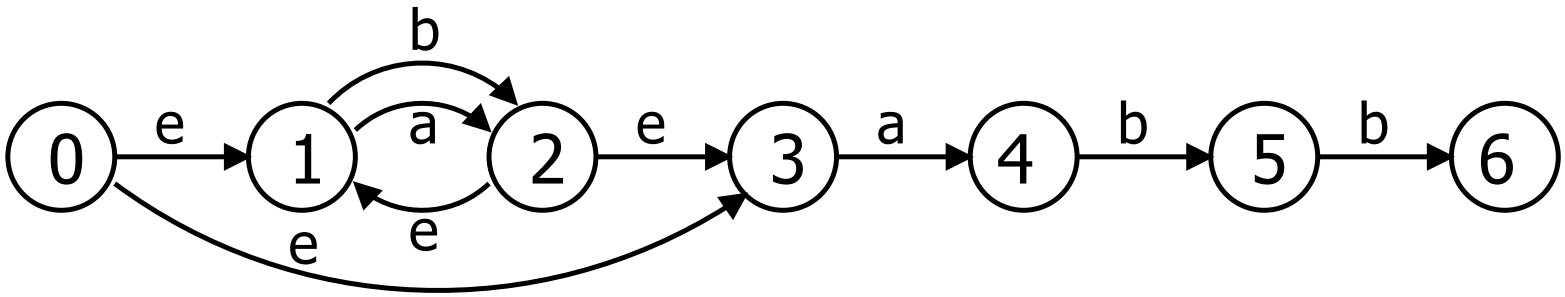


DFA Construction

DFA	NFA	a	b
0	{0,1,3}	{2,4,1,3}[1]	
1	{1,2,3,4}		

Example: $(a|b)^*abb$

NFA

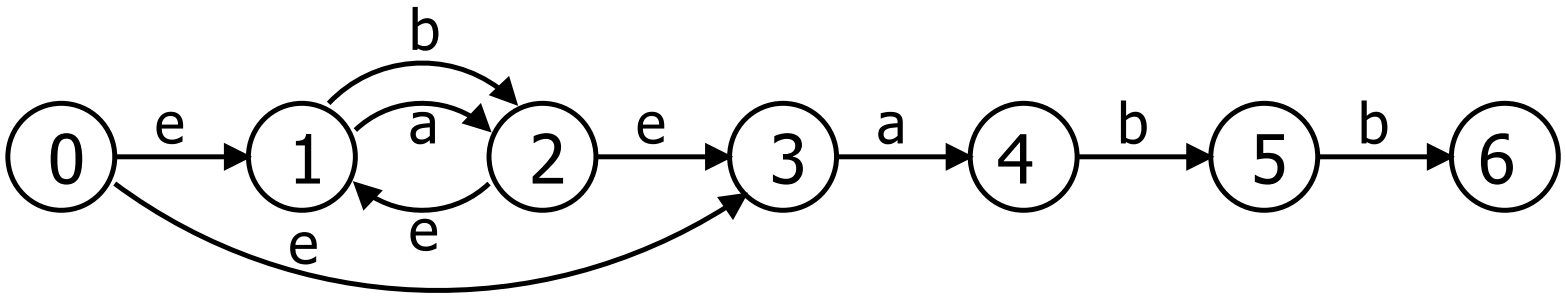


DFA Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}		
2	{1,2,3}		

Example: $(a|b)^*abb$

NFA

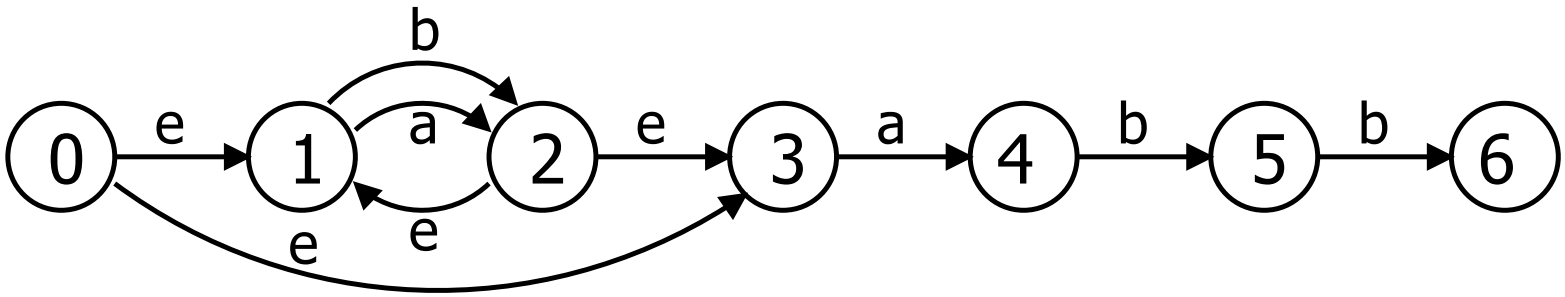


DFA Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	
2	{1,2,3}		

Example: $(a|b)^*abb$

NFA

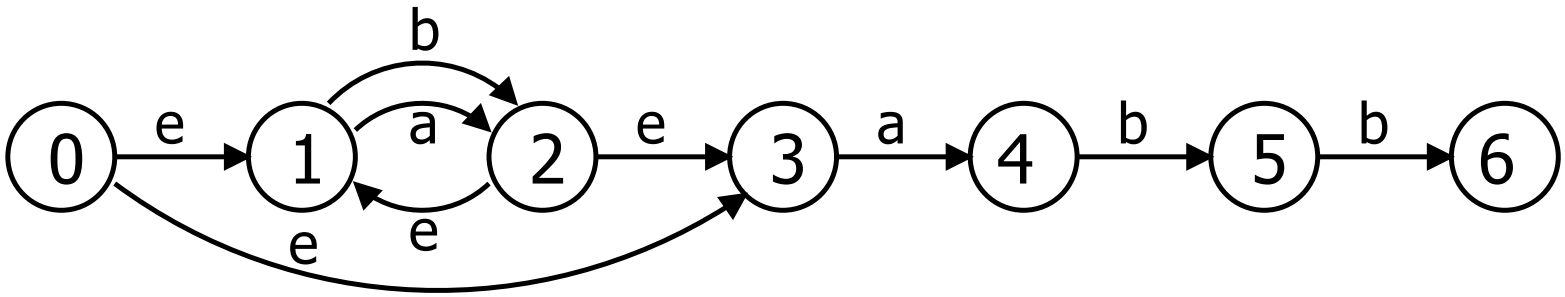


DFA Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	$\{2,5,1,3\}^3$
2	{1,2,3}		
3	{1,2,3,5}		

Example: $(a|b)^*abb$

NFA

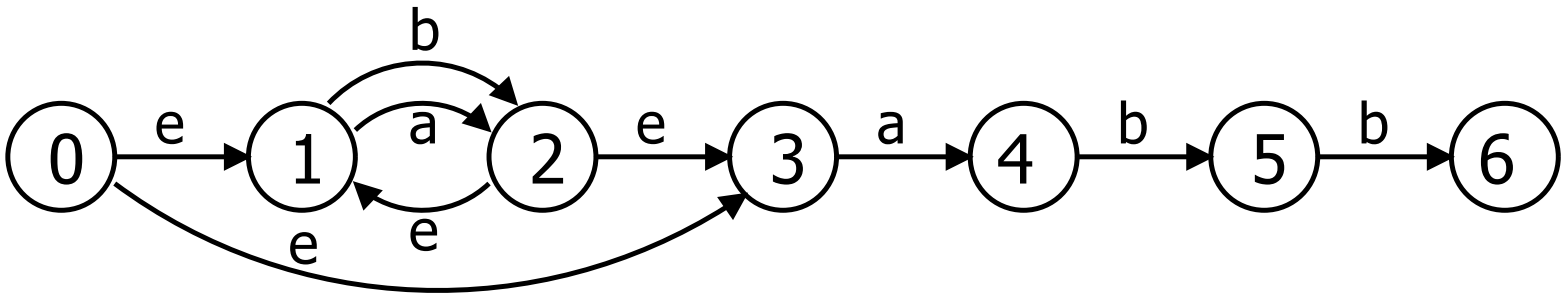


DFA Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	$\{2,5,1,3\}^3$
2	{1,2,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
3	{1,2,3,5}		

Example: $(a|b)^*abb$

NFA



DFA Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	$\{2,5,1,3\}^3$
2	{1,2,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
3	{1,2,3,5}	$\{2,4,1,3\}^1$	$\{2,6,1,3\}^4$
4	{1,2,3,6}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$

Step 5: NFA $\rightarrow$ DFA (Algorithm 3)

Operation	Description
closure(s)	States reachable from s on ε-transitions - s included
move(T, a)	ε-equivalent states reachable from state set T after input a
δ_{DFA}, Q_{DFA}	transition table and set of states for the DFA to be constructed

add closure(q_0) as an unmarked state to Q_{DFA}
while there is an unmarked state T in Q_{DFA} **do**
  mark T
  for each input symbol a **do**
    $U :=$ move(T, a)
    if $U \notin Q_{DFA}$ **then**
      add U as an unmarked state to Q_{DFA}
    end if
    $\delta_{DFA}[T, a] := U$
  end for
end while

DFA for $(a|b)^*abb$

Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	$\{2,5,1,3\}^3$
2	{1,2,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
3	{1,2,3,5}	$\{2,4,1,3\}^1$	$\{2,6,1,3\}^4$
4	{1,2,3,6}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$

Result

DFA	a	b	
0	1	2	start
1	1	3	
2	1	2	
3	1	4	
4	1	2	final

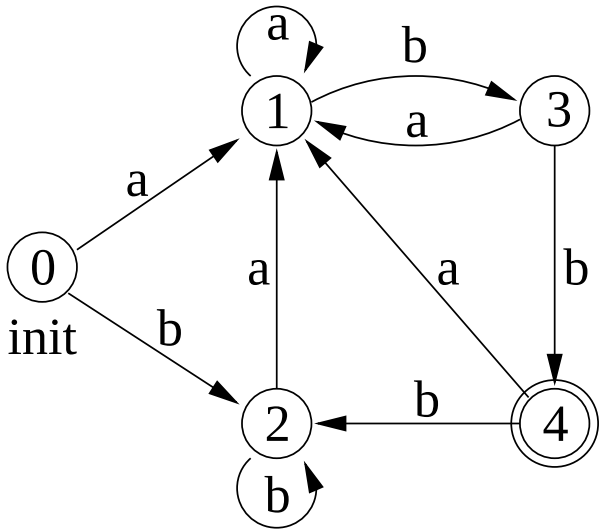
DFA for $(a|b)^*abb$

Construction

DFA	NFA	a	b
0	{0,1,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
1	{1,2,3,4}	$\{2,4,1,3\}^1$	$\{2,5,1,3\}^3$
2	{1,2,3}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$
3	{1,2,3,5}	$\{2,4,1,3\}^1$	$\{2,6,1,3\}^4$
4	{1,2,3,6}	$\{2,4,1,3\}^1$	$\{2,1,3\}^2$

Result

DFA	a	b	
0	1	2	start
1	1	3	
2	1	2	
3	1	4	
4	1	2	final



Scanner Summary

Scanner Construction:

1. Identify relevant (token,RegExp) pairs
2. Convert all RegExp:s to a single NFA
3. Convert the NFA to a DFA
4. Implement a scanner for the DFA

Notice: The steps 2-4 can be written as algorithms
$\Rightarrow$ A scanner can be generated for given set of RegExp:s
$\Rightarrow$ This is what happens inside AntLR.

Written assignment (WA1)

Construct a DFA accepting signed *integers* and *decimals*.

Item	Valid Expressions	Erroneous Expressions
Integers	34, 0, −12346	02, +67, −0
Floats	3.14, 0.02, −12.23, .47, -0.2	02.45, 23. , +2.1

Notice: We accept .47 for 0.47 but not 23. for 23.0.

- Use the algorithms presented at this lecture.
  1. Identify relevant RegExps r_{int}, r_{float} (No Algorithm!)
  2. Convert r_{int}, r_{float} to NFAs N_{int} and N_{float}
  3. Add N_{int} and N_{float} to a single NFA recognizing both ints and floats
  4. Convert the NFA to a DFA
- Act like a machine that executes the algorithms without thinking
  $\Rightarrow$ No clever rewritings in between the steps.
- More information available at the course web site.

Deadline: 2014-11-16

Good Luck!