


# FORMAL LANGUAGES AND COMPILERS – A COLLECTION OF NOTIONS FOR THE THEORY EXERCISES



luca gerin  
POLITECNICO DI MILANO

## Part 1 – Regular Expressions and Finite State Automata

Derive a string from a r.e.

At each step, make a choice:  $(a|b)^+[a] \rightarrow (a|b)(a|b)[a] \rightarrow (a)(a|b)[a] \rightarrow (a)(b)[a] \rightarrow (a)(b)$

Two r.e. are **equivalent** if they define the same language

A r.e. is **ambiguous** if a phrase can be obtained in more than one way through a different derivation, More formally, if the numbered r.e. generates a set of phrases that contains at least two phrases that are equal when the numbers are removed.

An automaton is **clean** if every state is accessible (reachable from initial state) and postaccessible (a final state can be reached from it). Every finite automaton admits an equivalent clean automaton, obtained by identifying and deleting useless states.

Two states are **distinguishable** if the set of labels on arcs outgoing from them are different and both of them are either final or non-final.

Forms of **nondeterminism** in automata:

- Alternative moves with same label from the same state
- Spontaneous moves ( $\varepsilon$ -moves)
- More than one initial state

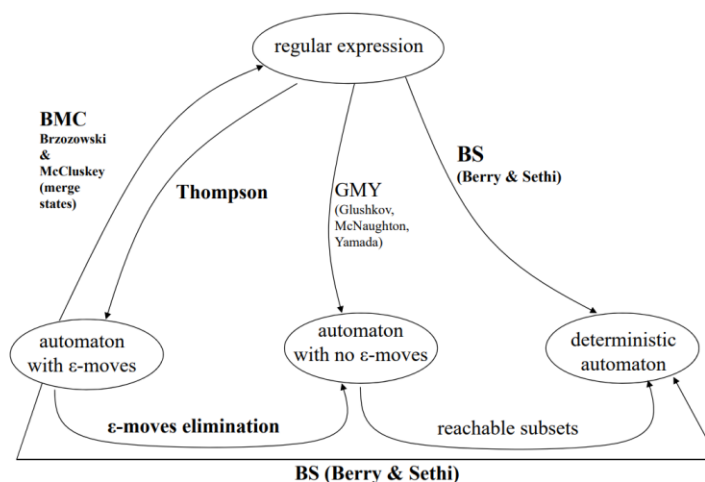
Note: states of a  $\varepsilon$ -loop can always be merged.

Obtain the automaton for the mirror language  $L^R$  from the automaton for  $L$ : switch initial and final states, reverse the arrows direction.

Obtain the automaton for the complement language  $\bar{L}$  from the automaton for  $L$ : add sink (error) state to make the automaton complete, then make the non-final states final and the final states non-final.

$$L(A) - L(B) = L(A) \cap \overline{L(B)}$$

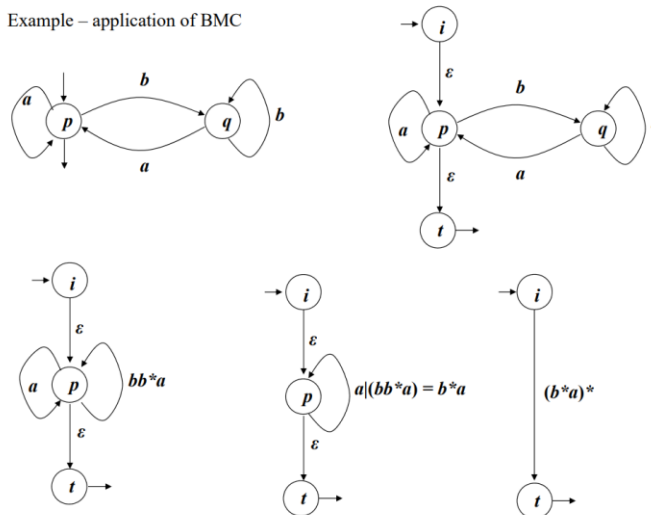
An automaton is **ambiguous** if it accepts a sentence with distinct computations.



BMC (Brzozowski & McCluskey) method: from automata to regular expression

- If there is no unique initial state without incoming arcs, add a new initial state connected to the initial states, now not initial anymore
- If there is no unique final state without outgoing arcs, add a new final state connected to the final states, now not final anymore
- Progressively eliminate internal states, by adding compensating moves labeled by appropriate r.e. that preserve the language, until only an initial and a final state are left

Example – application of BMC



#### Elimination of nondeterminism:

1. Elimination of  $\epsilon$ -moves, substituted by scanning moves

- a. Transitive closure of  $\epsilon$ -moves



- b. backward propagation of scanning moves over  $\epsilon$ -moves



- c. new final states: backward propagation of the «finality condition» for final states reached by  $\epsilon$ -moves

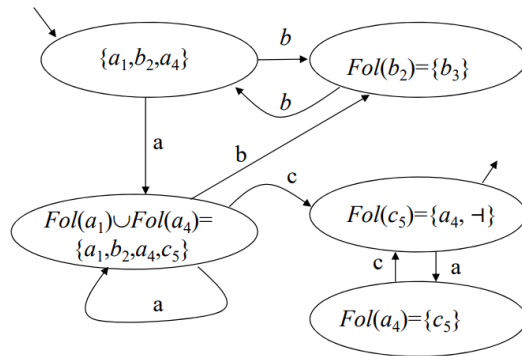


- d. clean-up of  $\epsilon$ -moves and of useless states

2. Replacement of several nondeterministic (scanning) transitions by a single one

Berry-Sethi (BS) deterministic recognizer: from r.e. to deterministic automaton or from non-deterministic to deterministic automaton

1. Write the numbered version of the r.e. or of the automaton
2. Find the set of initials terminal symbols
3. Find the followers of each terminal symbol
4. Create the initial state as the set of initials
5. Create subsequent states by using the followers set to generate them and transitions



$$e = (a \mid bb)^* (ac)^+$$

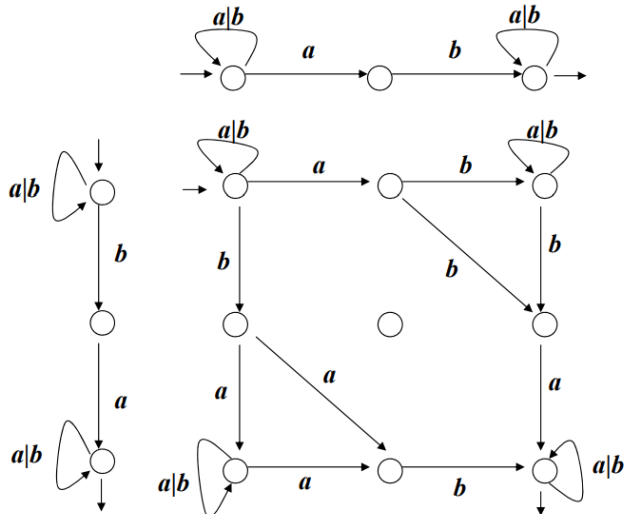
$$e^* \neg = (a_1 \mid b_2 b_3)^* (a_4 c_5)^+ \neg$$

$$Ini(e^* \neg) = \{a_1, b_2, a_4\}$$

$x$	$Fol(x)$
$a_1$	$a_1, b_2, a_4$
$b_2$	$b_3$
$b_3$	$a_1, b_2, a_4$
$a_4$	$c_5$
$c_5$	$a_4, \neg$

Recognizer for the intersection of two regular languages:

- Using De Morgan:  $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$
- Cartesian product automaton



## Part 2 – Free Grammars and Pushdown Automata

TYPES OF RULES (RP = right part, LP = left part)

<u><b>Terminal:</b></u> RP contains only terminals, or the empty string	$\rightarrow u \mid \varepsilon$
<u><b>Empty (or null):</b></u> RP is empty	$\rightarrow \varepsilon$
<u><b>Initial / Axiomatic:</b></u> LP is the axiom	$S \rightarrow$
<u><b>Recursive:</b></u> LP occurs in RP	$A \rightarrow \alpha A \beta$
<u><b>Left-recursive:</b></u> LP is prefix of RP	$A \rightarrow A \beta$
<u><b>Right-recursive:</b></u> LP is suffix of RP	$A \rightarrow \alpha A$
<u><b>Left- and right-recursive:</b></u> conjunction of two previous cases	$A \rightarrow A \beta A$
<u><b>Copy or categorization:</b></u> RP is a single nonterminal	$A \rightarrow B$
<u><b>Linear:</b></u> at most one nonterminal in RP	$\rightarrow u B v \mid w$
<u><b>Right-linear (type 3):</b></u> linear + nonterminal is suffix	$\rightarrow u B \mid w$
<u><b>Left-linear (type 3):</b></u> linear + nonterminal is prefix	$\rightarrow B v \mid w$
<u><b>Homogeneous normal:</b></u> $n$ nonterminals or just one terminal	$\rightarrow A_1 \dots A_n \mid a$
<u><b>Chomsky normal</b></u> (or homogeneous of degree 2): two nonterminals or just one terminal	$\rightarrow BC \mid a$
<u><b>Greibach normal:</b></u> one terminal possibly followed by nonterminals	$\rightarrow a \sigma \mid b$
<u><b>Operator normal:</b></u> two nonterminals separated by a terminal (operator); more generally, strings devoid of adjacent nonterminals	$\rightarrow A a B$

Derive a string from a grammar: follow the rules by derivations of the non-terminals.

A grammar is **clean** (or reduced) *iff*, for every nonterminal, that nonterminal is reachable (can be derived at some point starting from the axiom) and is defined (it does not generate the empty language, i.e. at some point a final string of terminals is produced).

Rule of thumb for defining languages:

- Precedence of operators: low precedence operators are derived first. High precedence operators are derived later
- Associativity of operators: left associative operators are modeled with left-recursive rules, while right associative operators with right-recursive rules

Two grammars are **weakly equivalent** if they generate the same language.

Two grammars are **strongly** or **structurally equivalent** if they are weakly equivalent and they have the same condensed skeleton trees.

The **Dyck language**:

$S \rightarrow aScS \mid \varepsilon$  where  $a$  is the left (opening) parenthesis and  $c$  is the right (closing) parenthesis

The **mirror language** is generated by the mirror grammar, obtained by reversing the right part of the rules.

A sentence of a grammar is **ambiguous** if it admits several distinct syntax trees. If such string exists, then the grammar is ambiguous.

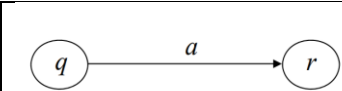
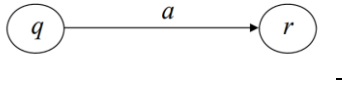
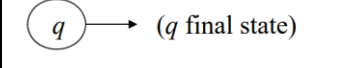
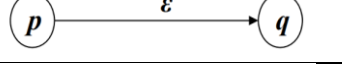
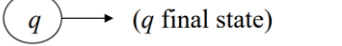
The **degree of ambiguity** of a sentence is the number of distinct syntax trees for that sentence, and the degree of ambiguity of a grammar is the maximum degree of ambiguity that can be found for a sentence of that grammar.

Ambiguous forms:

- Ambiguity from bilateral recursion:  $A \rightarrow A \dots A$  or  $A \rightarrow A \dots \mid \dots A$ 
  - To solve it, force an order in the derivations
- Ambiguity from language union: if the two languages are not disjoint

- To solve it: provide a disjointed set of rules, one for one of the grammars and one for the other without but the intersection
- Inherent ambiguity of a language: all the grammars of a language are ambiguous
- Ambiguity from concatenation of two languages
  - To solve it: insert a terminal symbol to act as separator

A grammar is **unilinear** iff its rules are either all right-linear ( $A \rightarrow uB$ ) or all left-linear ( $A \rightarrow Bv$ ).

Obtain a unilinear grammar from an automaton		Obtain an automaton from a unilinear grammar	
	$Q \rightarrow aR$		
	$Q \rightarrow \varepsilon$		
			

<b>Regular family:</b> Can be recognized by FSA.	<b>Context-free family:</b> I need at least a pushdown automaton to recognize a language.
---	--

## Part 3 - Syntax Analysis and Parsing Methodologies

ELR(1) conditions:

- No Shift-Reduce conflicts (lookaheads of final state contains same label of outgoing transition)
- No Reduce-Reduce conflicts (more final states with non-disjointed lookaheads)
- No conflictual converging arcs (two states go into the same, with the same input, producing non-disjointed lookaheads)

ELL(1) conditions:

- ELR(1)
- No left recursive derivations  $S \rightarrow S \dots$
- Simple transitions property (STP): Pilot has no multiple transitions, i.e. each macro state has the base part composed of only one state

Alternative way to check ELL(1):

- Draw guidesets on all arcs, exit arcs and call arcs
- The guidesets in bifurcations must be disjointed

To check ELL(k): utilize guide-sets with “lookaheads” of length k and verify that they are disjointed in bifurcations.

If it is ELL(1) then it's possible to write a recursive descent procedure:

```
program PARSER
  cc=next
  call s
  if cc∈ {-} then
    accept
  else
    error
end program

procedure X
0x:  if cc∈{<guideset of call arc>} then
      call A
      goto 1x
    if cc∈{<guideset of normal arc>} then
      [write <terminal>]          \\if it is a translator
      cc=next
      goto 2x
    if cc∈{<guideset of exit arc>} then
      return
    error
1x:  ...
2x:  ...
end procedure
```

## Part 4 - Language Translation and Semantic Analysis

The translation is **many-valued**, or not-single-valued, or **ambiguous** if it can assign more different translations to the same source string.

A translation is ambiguous only if the source grammar is ambiguous.

Representations of expressions:

**parenthesized functional:**  $add(i1, mult(i2, i3))$

**infix:**  $i1 + (i2 \times i3)$

**polish:**            **prefix:**             $add\ i1, mult\ i2, i3$

**postfix:**             $i1, i2, i3\ mult\ add$

A variable is **live** at some program point (i.e., on an arc) if some instruction that could be executed subsequently uses the value that the variable has at that point.

DEFINITION: a variable  $a$  is **live on an arc**, input or output of a program node  $p$ , if the graph admits a path from  $p$  to a node  $q$  such that

- instruction  $q$  uses  $a$ , that is,  $a \in use(q)$  **AND**
- the path does not traverse an instruction  $r$ ,  $r \neq q$  that defines  $a$ , that is, such that  $a \in def(r)$

Data flow equations:

for every final node  $p$  :

$$live_{out}(p) = \emptyset$$

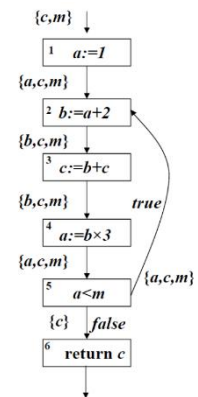
for every other node  $p$ :

$$live_{in}(p) = use(p) \cup (live_{out}(p) \setminus def(p))$$

$$live_{out}(p) = \bigcup_{\forall q \in succ(p)} live_{in}(q)$$

If two variables are never simultaneous live, they do not interfere and can be stored in the same memory cell or register.

An instruction defining a variable is useless if the variable is not live out of the instruction.



### Reaching definitions:

DEFINITION: A definition of  $a$  in instruction  $q$ ,  $a_q$ , reaches the entrance of instruction  $p$  (not necessarily distinct from  $q$ ) if there exists a path from  $q$  to  $p$  that does not traverse any node, distinct from  $q$ , where  $a$  is defined

Data flow equations:

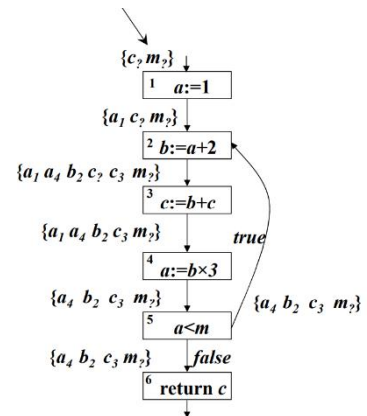
For the initial node 1 :

$$(1) \quad in(1) = \emptyset$$

For every other node  $p \in I$  :

$$(2) \quad out(p) = def(p) \cup (in(p) \setminus sup(p))$$

$$(3) \quad in(p) = \bigcup_{\forall q \in pred(p)} out(q)$$





**One-sweep** (depth first) condition:

- The dependency graph has no circuits
- In the dependency graph, there are no paths from a left attribute to a right attribute of the same node
- In the dependency graph, there are no paths from a left attribute of a father node to a right attribute of a child node
- The siblings graph has no circuits

**Condition L:**

- One-sweep
- The sibling graph for each rule  $D_0 \rightarrow D_1 \dots D_r$  is such that it is possible to choose a Topological Order of Siblings (TOS) that is the “natural” sequence  $D_1 D_2 \dots D_r$  (i.e., the sibling graph contains only dependencies from left to right)

If a grammar is LL(k) and satisfies the L condition, then it is possible to build a deterministic recursive descent parser that also evaluates the attributes.

Write the procedure of the semantic analyzer for a non-terminal X:

```
procedure X(l0: in, r0: out)    \\ right of father as input and left as output
  var l1, l2, r1, r2          \\ declare the attributes of children
  switch rule do
    case 1: X->AX do          \\rule previously defined as rule #1
      \\right attributes of children
      r1=l0+1
      r2=...
      \\recursive calls of children
      call A(...)
      call X(l2, r2)
      \\left attributes of father
      r0=...
    case 2: X->... do
      ...
    otherwise do
      error
  end procedure
```