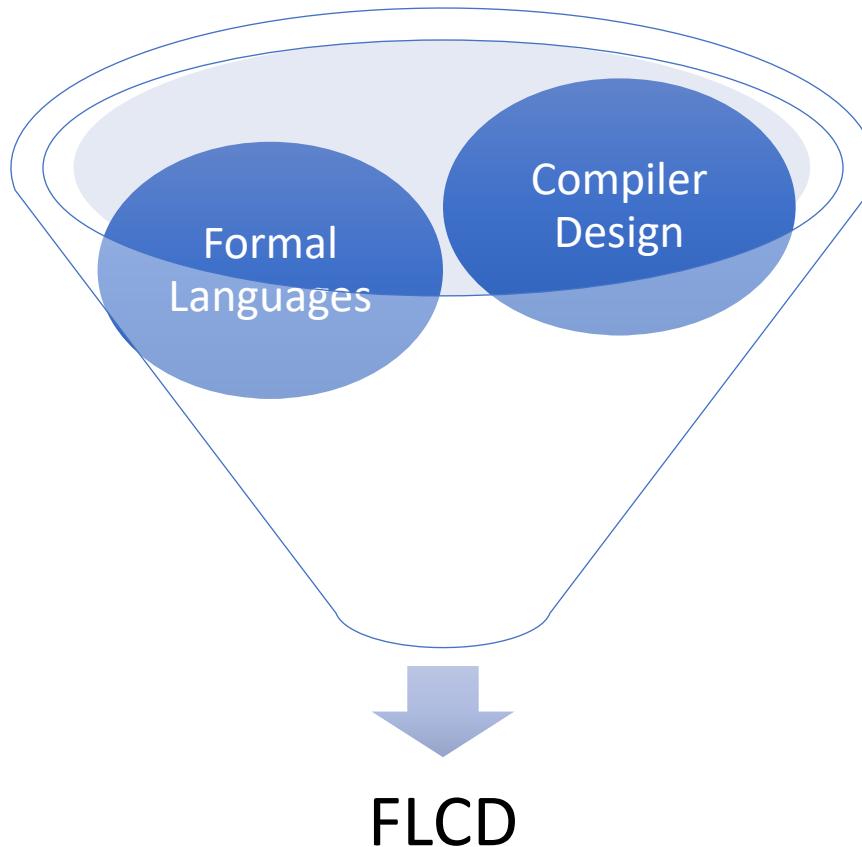


# Formal Languages and Compiler Design

*Simona Motogna*

# Why?

- Historical reasons
- Be a better programmer
- Performant algorithms



# Organization Issues

- Course – 2 h/ week
- Seminar – 2h/week
- Laboratory - 2 h/week

10 presences – seminar  
12 presences - lab

**PRESENCE IS MANDATORY**

# Most interesting stuff for students

- Moodle:
  - All course resources
  - Homeworks
  - Assignments
  - Labs
  - Points / grades
- MsTeams – labs (maybe)

# Minimal Conditions to Pass

- *Minimum 10 presences at seminar*
- *Minimum 12 presences at laboratory*
  
- *Minimum grade 6 at lab*
- *Minimum grade 5 at final exam*



# Final grade

60% final exam

+

30% lab

+

10% seminar

Bonus

# Lab work

- 10 laboratory tasks
- !!! Must be completed and loaded during lab hours

- Weighted grades:

Lab grade

*Bonus points:*

- “awesome” solutions
- Extra work

# I wish ...



Effective communication



Interactive experience



Learning fun

# References

- See fisa disciplinei

```

import time

def count(limit):
    result = 0
    for a in range(1, limit + 1):
        for b in range(a + 1, limit + 1):
            for c in range(b + 1, limit + 1):
                if c * c > a * a + b * b:
                    break

                if c * c == (a * a + b * b):
                    result += 1

    return result

```

```

001000 IDENTIFICATION DIVISION.
001000  PROGRAM-ID.
001000    ZBNKPRT1.
001000  DATE-WRITTEN.
001000    September 2002.
001000  DATE-COMPILED.
002000    Today.
002000  ENVIRONMENT DIVISION.
002000    INPUT-OUTPUT SECTION.
002000      FILE-CONTROL.
002000        SELECT EXTRACT-FILE
002000          ASSIGN TO EXTRACT
002000          ORGANIZATION IS SEQUENTIAL
002000          ACCESS MODE IS SEQUENTIAL
002000          FILE STATUS IS WS-EXTRACT-STATUS.
003000        SELECT PRINTOUT-FILE
003000          ASSIGN TO PRINTOUT
003000          ORGANIZATION IS SEQUENTIAL
003000          ACCESS MODE IS SEQUENTIAL
003000          FILE STATUS IS WS-PRINTOUT-STATUS.
003700  DATA DIVISION.
003700    FILE SECTION.
004000
004100 FD EXTRACT-FILE
004200 RECORDING MODE IS V

```

```

package rentalStore;
import java.util.Enumeration;
import java.util.Vector;

class Customer {
    private String _name;
    private Vector<Rental> _rentals = new Vector<Rental>();

    public Customer(String name) {
        _name = name;
    }
    public String getMovie(Movie movie) {
        Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE), 10);
        Movie m = rental._movie;
        return movie.getTitle();
    }
    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }
    public String getName() {
        return _name;
    }
}

```

```

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_squares; };

void stats_update(struct stats * s, int x, bool reset) {
    if (s == NULL) return;
    if (reset) * s = { 0, 0, 0 };
    s->count += 1;
    s->sum += x;
    s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
    struct stats s;
    for (int i = 0; i < len; ++i)
        stats_update(&s, data[i], i == 0);
    return ((double)s.sum) / ((double)s.count);
}

void main() {
    int data[] = { 1, 2, 3, 4, 5, 6 };
    printf("MEAN= %f\n", mean(data, sizeof(data) / sizeof(data[0])));
}

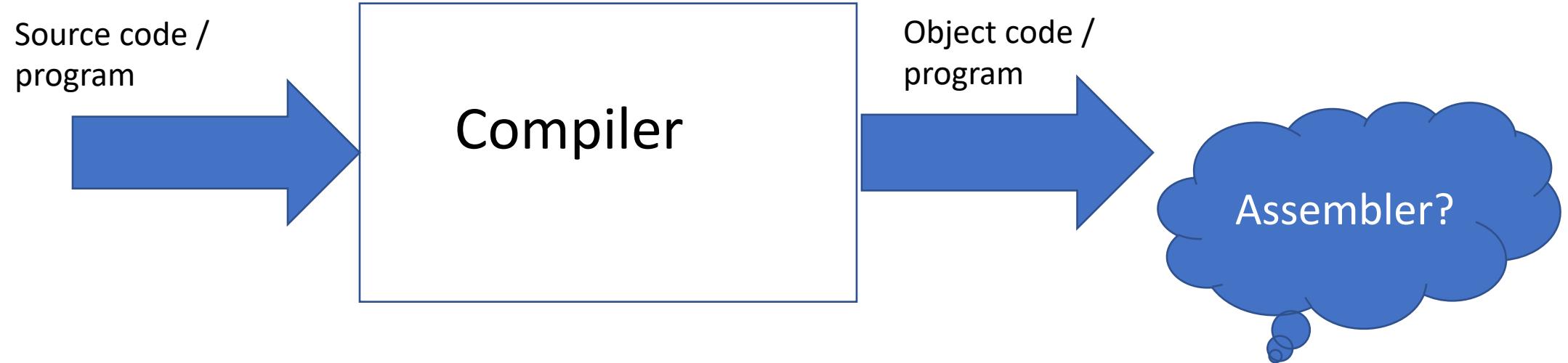
```

```

190   C
191   PIN=0.02
192   IF (DTT.NE.0.0) THEN
193   DTT=DTT
194   ELSE
195   DT=PIN
196   ENDIF
197   WRITE(*,'(A)') ' PLEASE ENTER NAME OF OUTPUT FILE (FOR EXAMPLE
198   * B:ZZ.DAT)'
199   READ(*,'(A)') FNAMEO
200   OPEN(6,FILE=FNAMEO,STATUS='UNKNOWN')
201   PV=WFLX/TH
202   RS=NSEQ*ROU*KD/TH
203   CO=CS
204
205   TIME=0.000
206   EF=0.000
207   5 CONTINUE
208   GAMMA=DT/(2.00*DX*DX)
209   BETA=DT/DX
210   IF ((BETA*PV).GT.0.50D0) GO TO 7
211   IF ((GAMMA*D/(BETA*PV)).LT.0.50D0) GO TO 6
212   GO TO 8
213   6 DX=DX/2
214   GO TO 5
215   7 DT=DT/2
216   GO TO 5
217   8 CONTINUE
218   N=COL/DX
219   NM1=N-1
220   NM2=N-2
221   NP1=N+1
222   GAMMA=DT/(2*DX*DX)

```

# What is a compiler?



```

import time

def count(limit):
    result = 0
    for a in range(1, limit + 1):
        for b in range(a + 1, limit + 1):
            for c in range(b + 1, limit + 1):
                if c * c > a * a + b * b:
                    break

                if c * c == (a * a + b * b):
                    result += 1
    return result

```

```

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

struct stats { int count; int sum; int sum_squares; };

void stats_update(struct stats * s, int x, bool reset) {
    if (s == NULL) return;
    if (reset) * s = (struct stats) { 0, 0, 0 };
    s->count += 1;
    s->sum += x;
    s->sum_squares += x * x;
}

double mean(int data[], size_t len) {
    struct stats s;
    for (int i = 0; i < len; ++i)
        stats_update(&s, data[i], i == 0);
    return ((double)s.sum) / ((double)s.count);
}

void main() {
    int data[] = { 1, 2, 3, 4, 5, 6 };
    printf("MEAN = %.1f\n", mean(data, sizeof(data) / sizeof(data[0])));
}

```

```

package rentalStore;
import java.util.Enumeration;
import java.util.Vector;

class Customer {
    private String _name;
    private Vector<Rental> _rentals = new Vector<Rental>();

    public Customer(String name) {
        _name = name;
    }

    public String getMovie(Movie movie) {
        Rental rental = new Rental(new Movie("", Movie.NEW_RELEASE), 10);
        Movie m = rental.movie;
        return movie.getTitle();
    }

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }

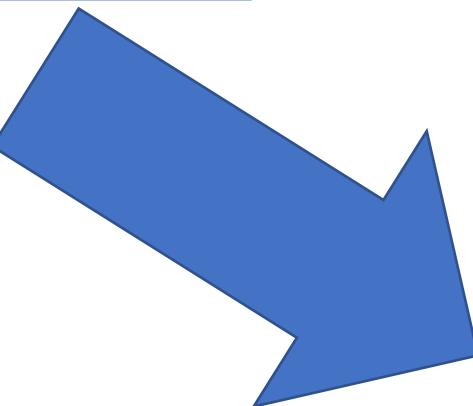
    public String getName() {
        return _name;
    }
}

```

```

190      C      PIN=0.02
191      IP(DOT,NE.0.0) THEN
192      DT=DT
193      ELSE
194      DT=DT
195      ENDIF
196      WRITE(*,'(A)')   * PLEASE ENTER NAME OF OUTPUT FILE (FOR EXAMPLE
197      * B.ZZ.DAT)
198      READ(*,*) FNAMBO
199      OPEN(6,FILE=FNAMBO,STATUS='UNKNOWN')
200      PV=MFLY/TB
201      RS=MFLY/RD/TR
202      CO=CD
203
204      TIME=0.000
205      EP=0.000
206      5  CONTINUE
207      GAMMA=D/(2.00*DX*DX)
208      BETA=D/(2.00*DX)
209      IF((BETA*PV).GT.0.500) GO TO 7
210      IF((GAMMA*D)/(BETA*PV)).LT.0.500) GO TO 6
211      GO TO 1
212
213      6  DX=DX/2
214      GO TO 5
215      7  DT=DT/2
216      GO TO 5
217      8  CONTINUE
218      N=COL/DX
219      NN=N-1
220      NN2=NN-2
221      NP1=NN+1
222      GAMMA=DT/(2*DX*DX)

```

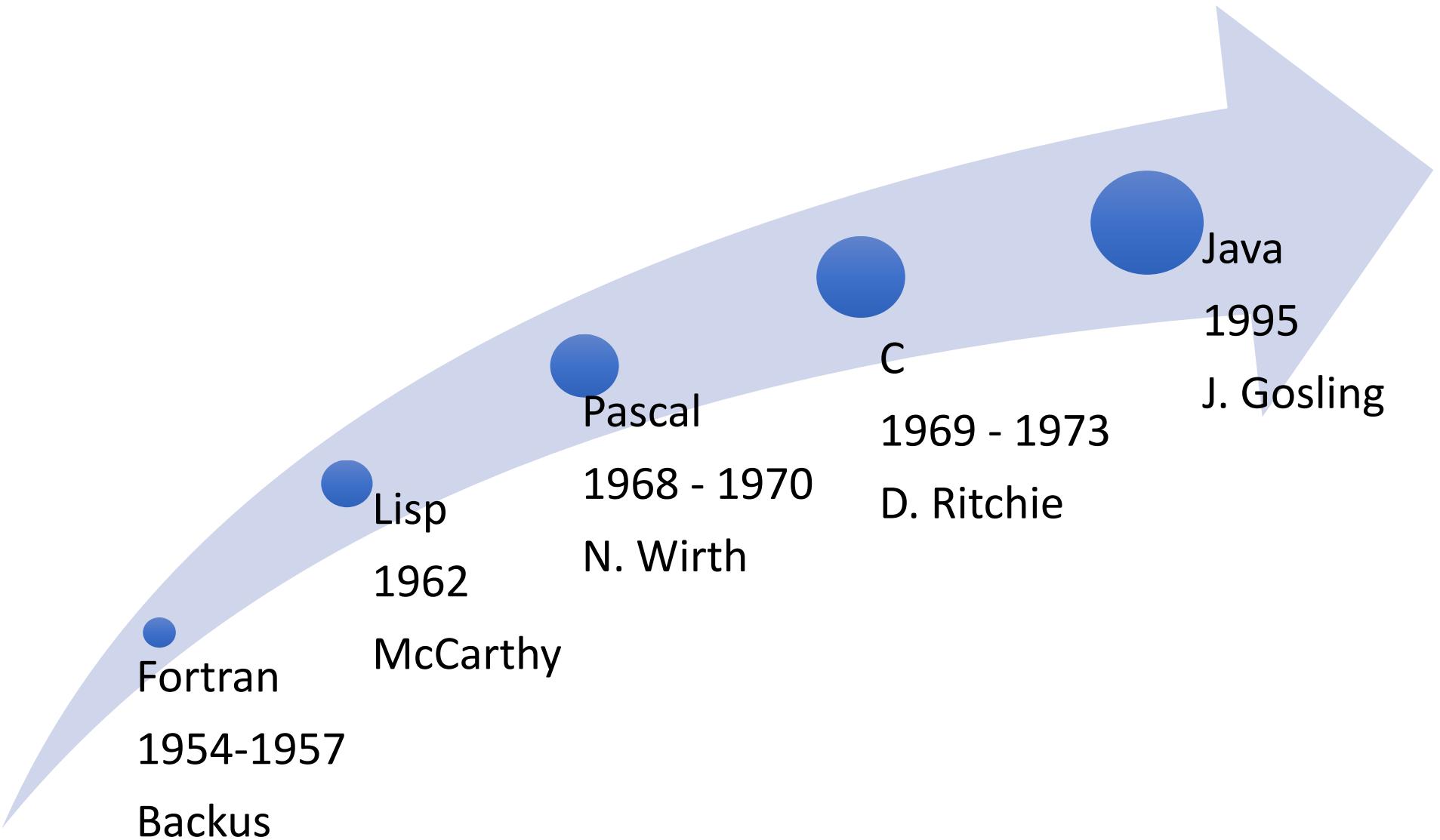


```

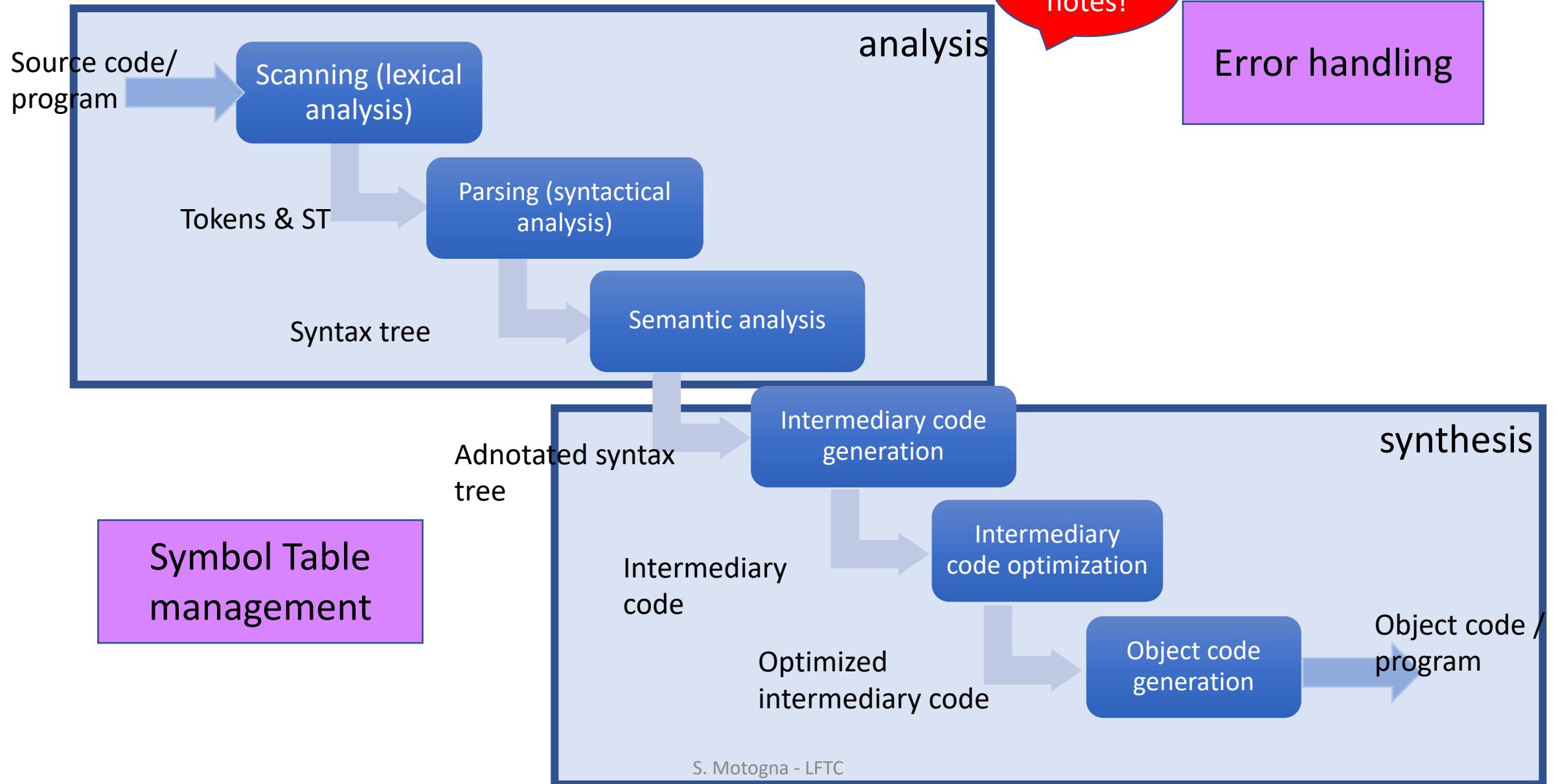
00000000 0000 0001 0001 1010 0010 0001 0004 0128
00000010 0000 0016 0000 0028 0000 0010 0000 0020
00000020 0000 0001 0004 0000 0000 0000 0000 0000
00000030 0000 0000 0000 0010 0000 0000 0000 0204
00000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
00000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
00000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
00000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
00000080 8888 8888 8888 288e be88 8888 8888
00000090 3b83 5788 8888 8888 7667 778e 8828 8888
000000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
000000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
000000c0 8a18 880c e841 c988 b328 6871 688e 958b
000000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
000000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
000000f0 8888 8888 8888 8888 8888 8888 8888 0000
00000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000 0000
000013e

```

# A little bit of history ...



# Structure of a compiler



# Chapter 1. Scanning

**Definition** = treats the source program as a sequence of characters, detect lexical tokens, classify and codify them

INPUT: source program  
OUTPUT: PIF + ST

*Algorithm Scanning v1*  
**while** (not(eof)) **do**  
    **detect(token);**  
    **classify(token);**  
    **codify(token);**  
**End\_while**

# Detect

Take  
notes!

```
I am a student. I    am  
          Simona
```

- Separators => ***Remark 1)***

```
if (x==y) {x=y+2}
```

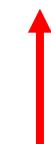
- Look-ahead => ***Remark 2)***

# Classify

- Classes of tokens:
  - Identifiers
  - Constants
  - Reserved words (keywords)
  - Separators
  - Operators
- If a token can NOT be classified => LEXICAL ERROR

# Codify

- May be codification table
- OR
- code for identifiers and constants
- Identifier, constant => Symbol Table (ST)
- PIF = Program Internal Form = array of pairs
- pairs (token, position in ST)



identifier, constant

## *Algorithm Scanning v2*

```
while (not(eof)) do
    detect(token);
    if token is reserved word OR operator OR separator
        then genPIF(token, 0)
        else
            if token is identifier OR constant
                then index = pos(token, ST);
                    genPIF(token, index)
                else message "Lexical error"
            endif
        endif
    endwhile
```

a=a+b

FIP

(id,1)

(=,0)

(id,1)

(+,0)

(id,2)

ST

1 a

2 b

## Remarks:

- genPIF = adds a pair (token, position) to PIF
- Pos(token,ST) – searches *token* in symbol table ST; if found then return position; if not found insert in SR and return position
- Order of classification (reserved word, then identifier)
- If-then-else imbricate => detect error if a token cannot be classified

# Example (sem?)

- <https://babeljs.io/docs/en/>
- <https://www.antlr.org/> and <https://github.com/antlr/antlr4>
- <https://www.programiz.com/python-programming/online-compiler/>
- [https://www.w3schools.com/python/python\\_compiler.asp](https://www.w3schools.com/python/python_compiler.asp)

# Course 2

## *Algorithm Scanning v2*

```
while (not(eof)) do
    detect(token);
    if token is reserved word OR operator OR separator
        then genPIF(token, 0)
        else
            if token is identifier OR constant
                then index = pos(token, ST);
                    genPIF(token_type, index)
                else message "Lexical error"
            endif
        endif
    endwhile
```

## Remarks:

- Also comments are eliminated
- Most important operations: SEARCH and INSERT

# Symbol Table

***Definition*** = contains all information collected during compiling regarding the symbolic names from the source program



identifiers, constants, etc.

## Variants:

- Unique symbol table – contains all symbolic names
- distinct symbol tables: IT (identifiers table) + CT (constants table)

# ST organization

*Remark:* search and insert

1. Unsorted table – in order of detection in source code  $O(n)$
2. Sorted table: alphabetic (numeric)  $O(\lg n)$
3. Binary search tree (balanced)  $O(\lg n)$
4. Hash table  $O(1)$

# Hash table

- $K$  = set of keys (symbolic names)
- $A$  = set of positions ( $|A| = m$ ;  $m$  –prime number)

$h : K \rightarrow A$

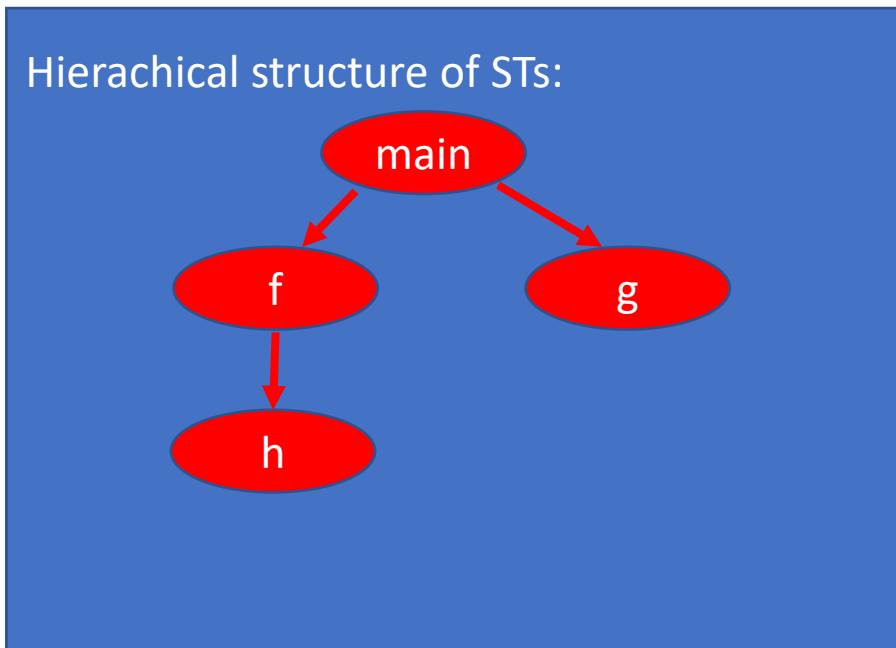
$$h(k) = (\text{val}(k) \bmod m) + 1$$

- Conflicts:  $k_1 \neq k_2$ ,  $h(k_1) = h(k_2)$

Toy hash function to use at  
lab:  
Sum of ASCII codes of chars

# Visibility domain (scope)

- Each scope – separate ST
- Structure -> inclusion tree



*Example:*

```
Int main(){  
... int a;  
  
void f()  
{float a;  
... int h() {...}  
}  
  
...  
void g()  
{char a;  
...  
}  
}
```

# Formal Languages

*- basic notions -*

# Examples of languages

- natural (ex. English, Romanian)
- programming (ex. C,C++, Java, Python)
- formal

A formal language is a set

Ex.:

$$L = \{a^n b^n \mid n > 0\} \quad L = \{ab, aabb, aaabbb, \dots\}$$

$$L' = \{01^n \mid n \geq 0\} \quad L' = \{0, 01, 011, \dots\}$$

# Example

a boy has a dog

$S \rightarrow PV$   
 $P \rightarrow a N$   
 $N \rightarrow boy \text{ or } N \rightarrow dog$   
 $V \rightarrow QC$   
 $Q \rightarrow has$   
 $C \rightarrow BN$   
 $B \rightarrow a$

- $A \rightarrow \alpha = \text{rule}$
- $S, P, V, N, Q, C, B = \text{nonterminal symbols}$
- $a, boy, dog, has = \text{terminal symbols}$

## Remarks

1. Sentence = word, sequence (contains only terminal symbols) ; denoted w.
2.  $S \Rightarrow PV \Rightarrow a NV \Rightarrow a NQC \Rightarrow a N \text{ has } C$  - sentential form  
In general :  $w = a_1 a_2 \dots a_n$
3. The rule guarantees syntactical correctness, but not the semantical correctness (*A dog has a boy*)

# Grammar

- **Definition:** A (formal) **grammar** is a 4-tuple:  $G=(N,\Sigma,P,S)$  with the following meanings:

- $N$  – set of nonterminal symbols and  $|N| < \infty$
- $\Sigma$  - set of terminal symbols (alphabet) and  $|\Sigma| < \infty$
- $P$  – finite set of productions (rules), with the property:  
$$P \subseteq (N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$
- $S \in N$  – start symbol/axiom

## Remarks :

1.  $(\alpha, \beta) \in P$  is a production denoted  $\alpha \rightarrow \beta$
2.  $N \cap \Sigma = \emptyset$

$A^*$  = transitive and reflexive closure =  
 $\{a, aa, aaa, \dots\} \{a^0\}$   
 $A = \{a\}$   
 $A^+ = \{a, aa, aaa, \dots\}$   
 $X^0 = \epsilon$

# Binary relations defined on $(N \cup \Sigma)^*$

- **Direct derivation**

$\alpha \Rightarrow \beta , \alpha, \beta \in (N \cup \Sigma)^* \text{ if } \alpha=x_1xy_1 , \beta=x_1yy_1 \text{ and } x \rightarrow y \in P$   
(x is transformed in y)

- **k derivation**

$\overset{k}{\alpha \Rightarrow \beta} , \alpha, \beta \in (N \cup \Sigma)^*$

sequence of k direct derivations  $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{k-1} \Rightarrow \beta , \alpha, \alpha_1, \alpha_2, \dots, \alpha_{k-1}, \beta \in (N \cup \Sigma)^*$

- **+ derivation**

$\overset{+}{\alpha \Rightarrow \beta} \text{ if } \exists k > 0 \text{ such that } \overset{k}{\alpha \Rightarrow \beta}$  (there exists at least one direct derivation)

- **\* derivation**

$\overset{*}{\alpha \Rightarrow \beta} \text{ if } \exists k \geq 0 \text{ such that } \overset{k}{\alpha \Rightarrow \beta}$  namely,  $\overset{*}{\alpha \Rightarrow \beta} \Leftrightarrow \overset{+}{\alpha \Rightarrow \beta} \text{ OR } \overset{0}{\alpha \Rightarrow \beta} (\alpha = \beta)$

**Definition:** Language generated by a grammar  $G=(N,\Sigma,P,S)$  is:

$$L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$$

**Remarks:**

1.  $S \xrightarrow{*} \alpha, \alpha \in (N \cup \Sigma)^*$  = sentential form  
 $S \xrightarrow{*} w, w \in \Sigma^*$  = word / sequence

2. Operations defined for languages (sets) :

$$L_1 \cup L_2, L_1 \cap L_2, L_1 - L_2, \overline{L} \text{ (complement)}, L^+ = \bigcup_{k>0} L^k, L^* = \bigcup_{k \geq 0} L^k$$

$$\text{Concatenation: } L = L_1 L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$$

3.  $|w|=0$  (empty word - denoted  $\epsilon$ )

$$L_1 = \{a, b, aa\}$$

$$L_2 = \{c, d, cd\}$$

$$L_1 L_2 = \{ac, ad, acd, bc, bd, bcd, aac, aad, aacd\}$$

**Definition:** Two grammar  $G_1$  and  $G_2$  are equivalent if they generate the same language

$$L(G_1) = L(G_2)$$

# Chomsky hierarchy(based on form $\alpha \rightarrow \beta \in P$ )

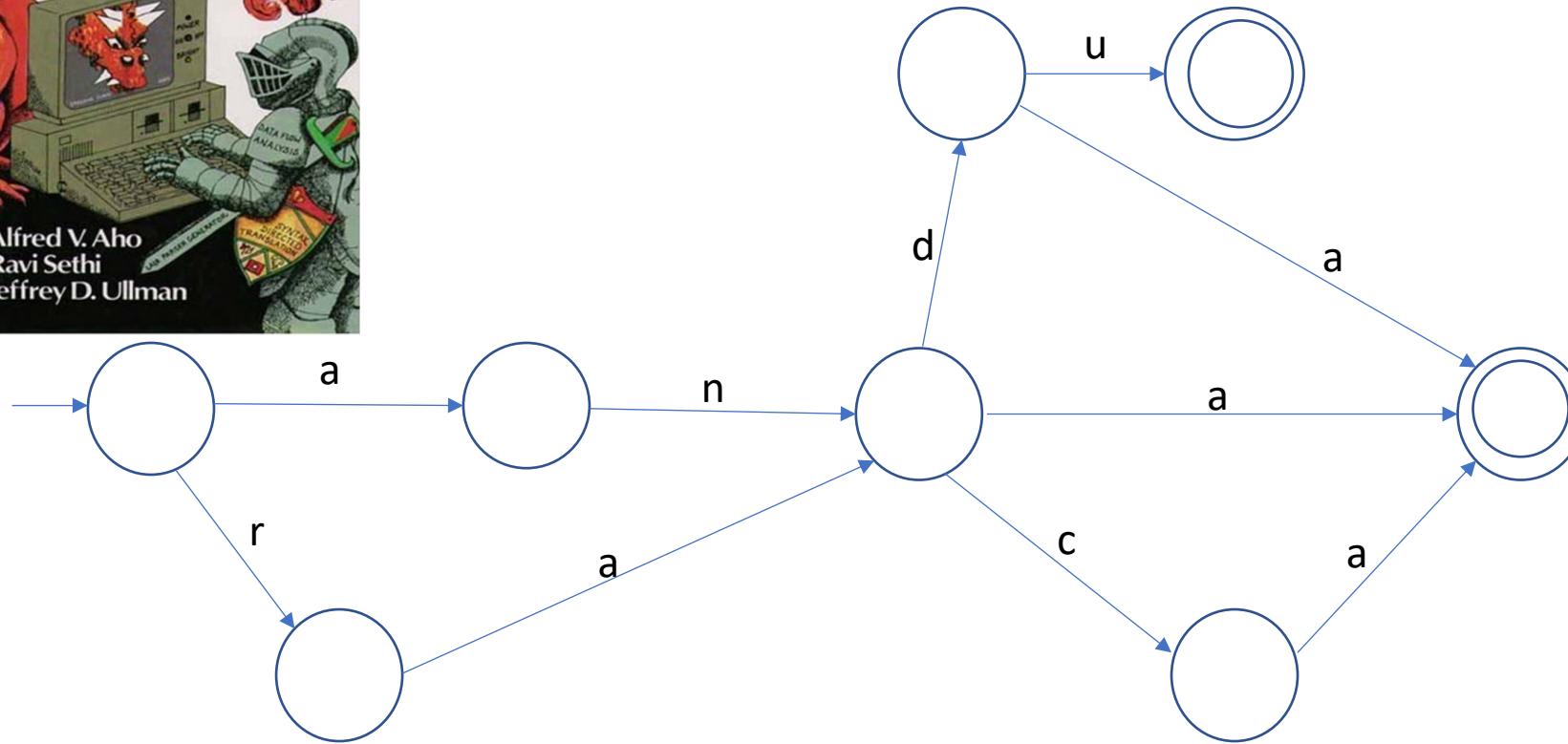
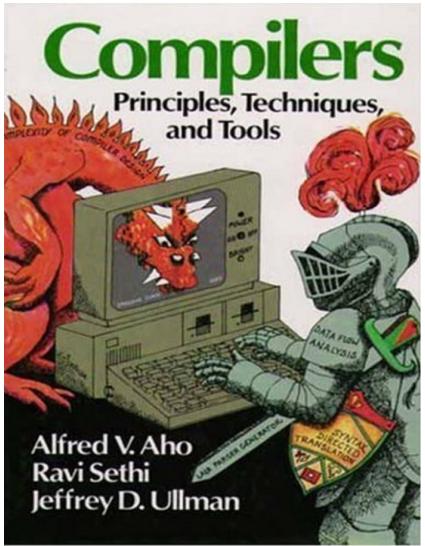
- type 0 : no restriction
- type 1 : context dependent grammar ( $x_1Ay_1 \rightarrow x_1\gamma y_1$ )
- type 2 : context free grammar ( $A \rightarrow \alpha \in P$  ,where  $A \in N$  and  $\alpha \in (N \cup \Sigma)^*$ )
- type 3 : regular grammar (  $A \rightarrow aB | a \in P$ )

***Remark :***

$$\text{type 3} \subseteq \text{type 2} \subseteq \text{type 1} \subseteq \text{type 0}$$

# Notations

- $A, B, C, \dots$  – nonterminal symbols
- $S \in N$  – start symbol
- $a, b, c, \dots \in \Sigma$  – terminal symbol
- $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$  - sentential forms
- $\epsilon$  – empty word
- $x, y, z, w \in \Sigma^*$  - words
- $X, Y, U, \dots \in (N \cup \Sigma)$  – grammar symbols (nonterminal or terminal)



**Problem:** The door to the tower is closed by the **Red Dragon**, using a complicated machinery. Prince Charming has managed to steal the plans and is asking for your help. Can you help him determining all the person names that can unlock the door

# Course 3&4

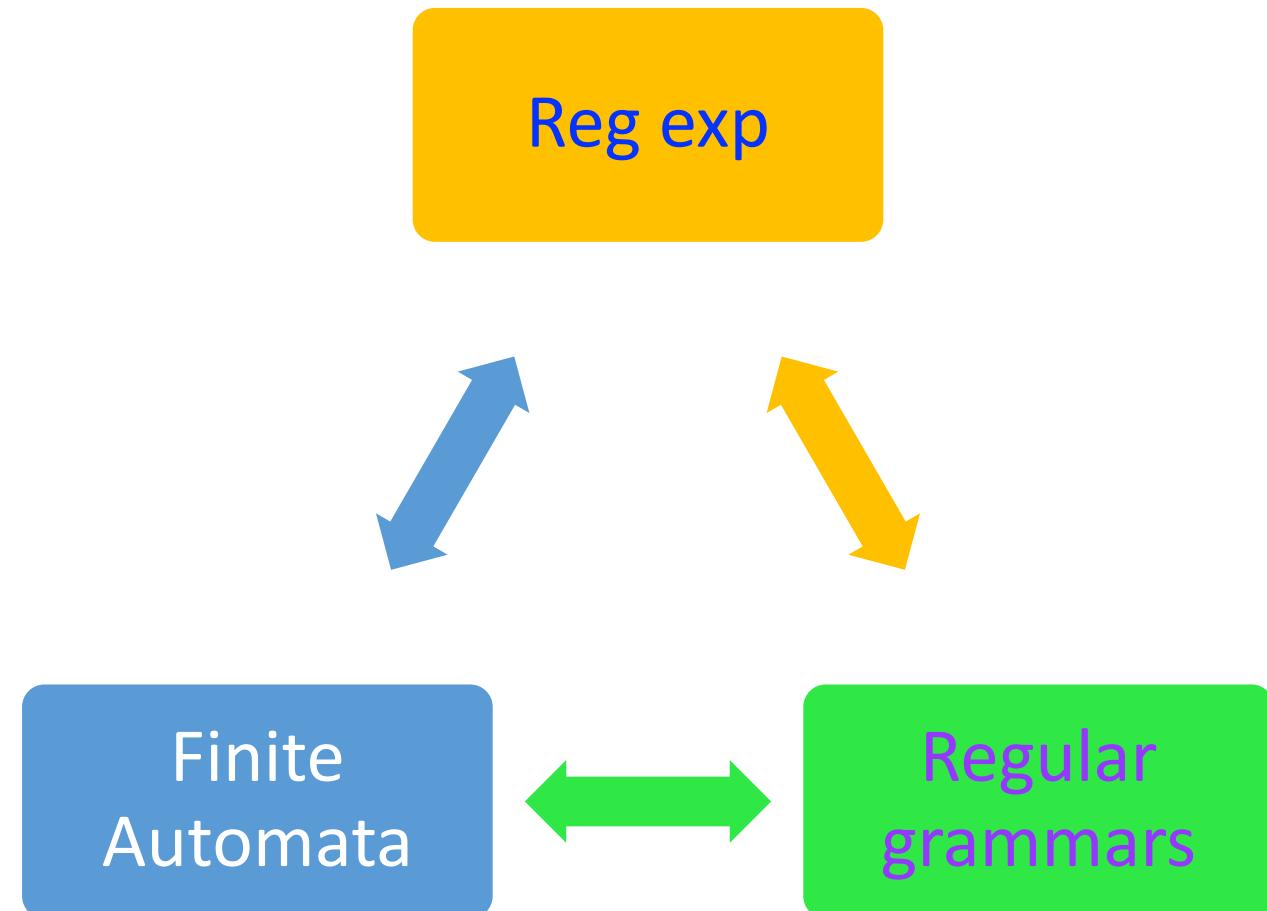
## Formal Languages

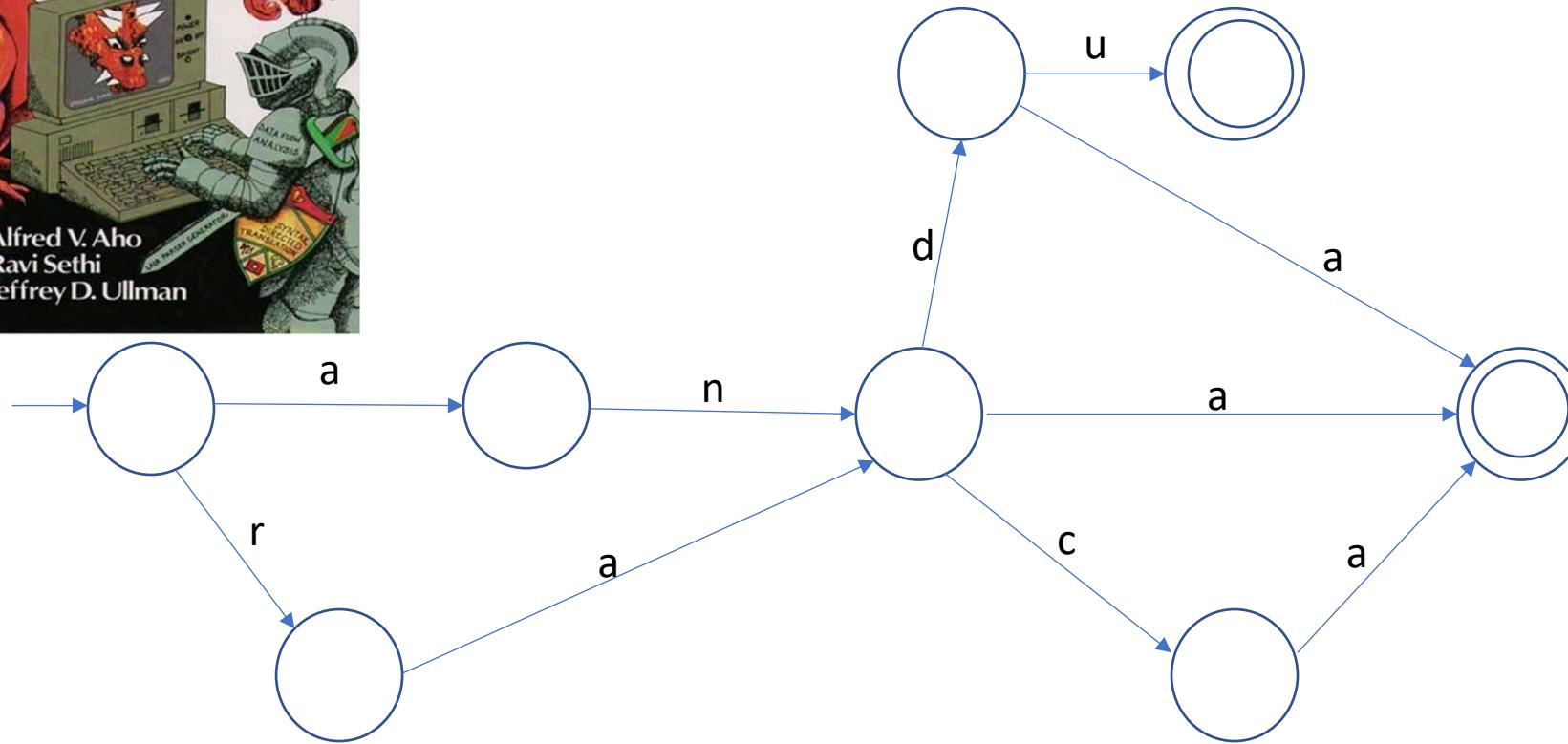
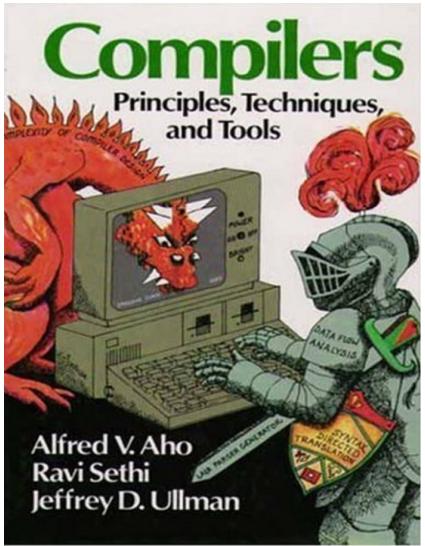
*- Basic notions -*

# Regular languages

# Why?

1. Search engine – success of Google
2. Unix commands
3. Programming languages – new feature

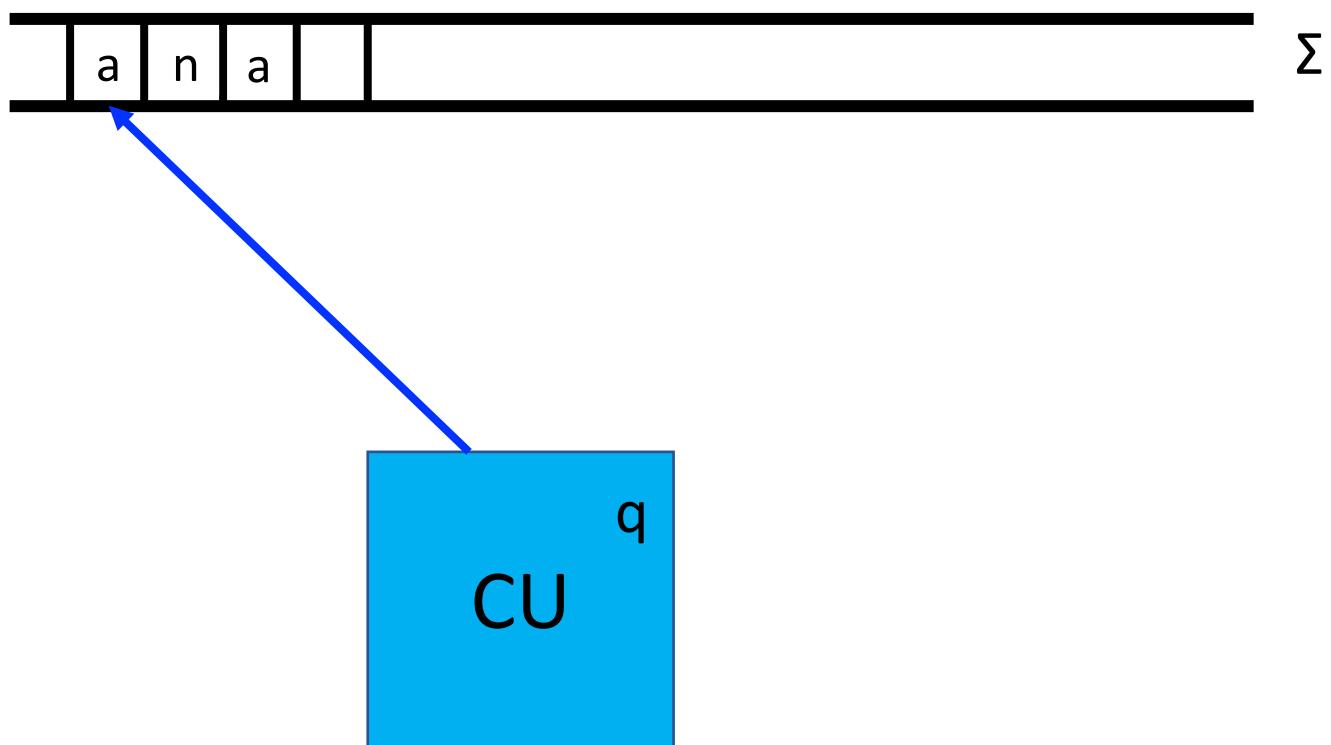




**Problem:** The door to the tower is closed by the **Red Dragon**, using a complicated machinery. Prince Charming has managed to steal the plans and is asking for your help. Can you help him determining all the person names that can unlock the door

# Finite Automata

- Intuitive model



**Definition:** A *finite automaton (FA)* is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- $Q$  - finite set of states ( $|Q| < \infty$ )
- $\Sigma$  - finite alphabet ( $|\Sigma| < \infty$ )
- $\delta$  – transition function :  $\delta: Q \times \Sigma \rightarrow P(Q)$
- $q_0$  – initial state  $q_0 \in Q$
- $F \subseteq Q$  – set of final states

## *Remarks*

1.  $Q \cap \Sigma = \emptyset$
2.  $\delta: Q \times \Sigma \rightarrow P(Q)$ ,  $\varepsilon \in \Sigma^0$  - relation  $\delta(q, \varepsilon) = p$  **NOT** allowed
3. If  $|\delta(q, a)| \leq 1 \Rightarrow$  deterministic finite automaton (DFA)
4. If  $|\delta(q, a)| > 1$  (more than a state obtained as result)  $\Rightarrow$  nondeterministic finite automaton (NFA)

**Property:** For any NFA  $M$  there exists a DFA  $M'$  equivalent to  $M$

## *Configuration C=(q,x)*

where:

- q state
- x unread sequence from input:  $x \in \Sigma^*$

Initial configuration :  $(q_0, w)$  , w - whole sequence

Final configuration:  $(q_f, \varepsilon)$  ,  $q_f \in F$ ,  $\varepsilon$  –empty sequence  
(corresponds to accept)

# Relations between configurations

- $\vdash$  **move / transition** (simple, one step)  
 $(q,ax) \vdash (p,x)$ ,  $p \in \delta(q,a)$
- $\vdash^k$  **k move** = a sequence of k simple transitions)  $C_0 \vdash C_1 \vdash \dots \vdash C_k$
- $\vdash^+$  **+ move**  
 $C \vdash^+ C' : \exists k > 0$  such that  $C \vdash^k C'$
- $\vdash^*$  **\* move (star move)**  
 $C \vdash^* C' : \exists k \geq 0$  such that  $C \vdash^k C'$

**Definition** : *Language* accepted by FA  $M = (Q, \Sigma, \delta, q_0, F)$  is:

$$L(M) = \{ w \in \Sigma^* \mid (q_0, w) \xrightarrow{*} (q_f, \varepsilon), q_f \in F \}$$

### Remarks

1. 2 finite automata  $M_1$  and  $M_2$  are equivalent if and only if they accept the same language

$$L(M_1) = L(M_2)$$

1.  $\varepsilon \in L(M) \Leftrightarrow q_0 \in F$  (initial state is final state)

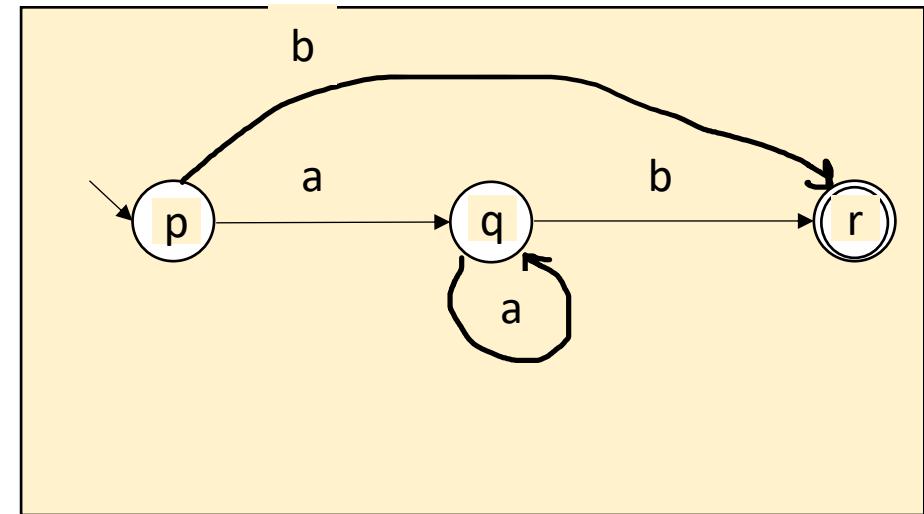
# Representing FA

1. List of all elements
2. Table
3. Graphical representation

$M = (Q, \Sigma, \delta, p, F)$   
 $Q = \{p, q, r\}$   
 $\Sigma = \{a, b\}$   
 $\delta(p, a) = q$   
 $\delta(q, a) = q$   
 $\delta(q, b) = r$   
 $\delta(p, b) = r$   
 $F = \{r\}$

$M = (Q, \Sigma, \delta, p, F)$   
 $F = \{r\}$

	a	b
p	q	r
q	q	r
r	-	-



$(p, aab) | -(q, ab) | -(q, b) | -(r, \epsilon) \Rightarrow aab$  accepted  
 $(p, aba) | -(q, ba) | -(r, a) \Rightarrow aba$  not accepted

# Remember

- Grammar

$$G = (N, \Sigma, P, S)$$

$$L(G) = \{ w \in \Sigma^* \mid S \xrightarrow{*} w \}$$

- Finite automaton

$$M = (Q, \Sigma, \delta, q_0, F)$$

$$L(M) = \{ w \in \Sigma^* \mid (q_0, w) \vdash (q_f, \epsilon), q_f \in F \}$$

# Regular grammars

- $G = (N, \Sigma, P, S)$  **right linear grammar** if

$\forall p \in P: A \rightarrow aB \text{ or } A \rightarrow b$ , where  $A, B \in N$  and  $a, b \in \Sigma$

- $G = (N, \Sigma, P, S)$  **regular grammar** if

- $G$  is right linear grammar  
and

- $A \rightarrow \epsilon \notin P$ , with the exception that  $S \rightarrow \epsilon \in P$ , in which case  $S$  does not appear in the rhs (right hand side) of any other production

- $L(G) = \{w \in \Sigma^* \mid S \xrightarrow{*} w\}$  - right linear language

S->aA | ε; A-> a reg  
S->aS | aA; A->bS | b reg  
S->aA; A->aA | ε NOT reg  
S->aA | ε; A->aS NOT reg

**Theorem 1:** For any regular grammar  $G=(N, \Sigma, P, S)$  there exists a FA  $M=(Q, \Sigma, \delta, q_0, F)$  such that  $L(G) = L(M)$

Proof: construct  $M$  based on  $G$

$$Q = N \cup \{K\}, K \notin N$$

$$q_0 = S$$

$$F = \{K\} \cup \{S \mid \text{if } S \xrightarrow{\epsilon} \epsilon \in P\}$$

$$\delta: \text{if } A \xrightarrow{a} B \in P \text{ then } \delta(A, a) = B$$

$$\text{if } A \xrightarrow{\epsilon} K \in P \text{ then } \delta(A, \epsilon) = K$$

Prove that  $L(G) = L(M)$  ( $w \in L(G) \Leftrightarrow w \in L(M)$ ):

$$S \xrightarrow{*} w \Leftrightarrow (S, w) \vdash^{*} (qf, \epsilon)$$

$$w = \epsilon: S \xrightarrow{*} \epsilon \Leftrightarrow (S, \epsilon) \vdash^{*} (S, \epsilon) - \text{true}$$

$$w = a_1 a_2 \dots a_n: S \xrightarrow{*} w \Leftrightarrow (S, w) \vdash^{*} (K, \epsilon)$$

$$S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \dots \Rightarrow a_1 a_2 \dots a_{n-1} A_{n-1} \Rightarrow a_1 a_2 \dots a_{n-1} a_n$$

$S \Rightarrow a_1 A_1$  exists if  $S \rightarrow a_1 A_1$  and then  $\delta(S, a_1) = A_1$

$A_1 \rightarrow a_2 A_2 : \delta(A_1, a_2) = A_2 \dots$

$A_{n-1} \rightarrow a_n : \delta(A_{n-1}, a_n) = K$

$$(S, a_1 a_2 \dots a_n) \vdash (A_1, a_2 \dots a_n) \vdash (A_2, a_3 \dots a_n) \vdash \dots \vdash (A_{n-1}, a_n) \vdash (K, \epsilon), K \in F$$

**Theorem 2:** For any FA  $M = (Q, \Sigma, \delta, q_0, F)$  there exists a right linear grammar  $G = (N, \Sigma, P, S)$  such that  $L(G) = L(M)$

Proof: construct  $G$  based on  $M$

$$N = Q$$

$$S = q_0$$

$P$ : if  $\delta(q, a) = p$  then  $q \rightarrow ap \in P$

if  $p \in F$  then  $q \rightarrow a \in P$

if  $q_0 \in F$  then  $S \rightarrow \epsilon$

Prove that  $L(M) = L(G)$

( $w \in L(M) \Leftrightarrow w \in L(G)$ ):

$P(i)$ :  $q \xrightarrow{i+1} x \Leftrightarrow (q, x) \vdash^i (q_f, \epsilon)$ ,  $q_f \in F$  -prove by induction

Apply  $P$ :  $q_0 \xrightarrow{i+1} w \Leftrightarrow (q_0, w) \vdash^i (q_f, \epsilon)$ ,  $q_f \in F$

If  $i=0$ :  $q \Rightarrow x \Leftrightarrow (q, x) \vdash^0 (q_f, \epsilon)$  ( $x = \epsilon, q = q_f$ )  $q \Rightarrow \epsilon \Leftrightarrow q_0 \rightarrow \epsilon$ ,  $q_0 \in F$

Assume  $\forall k \leq i$   $P$  is true

$q \xrightarrow{i+1} x \Leftrightarrow (q, x) \vdash^i (q_f, \epsilon)$

For  $q \in N$  apply " $\Rightarrow$ ":  $q \Rightarrow ap \xrightarrow{i} ax$

If  $q \Rightarrow ap$  then  $\delta(q, a) = p$ ; if  $p \xrightarrow{i} ax$  then  $(p, x) \vdash^{i-1} (q_f, \epsilon)$ ,  $q_f \in F$

THEN  $(q, ax) \vdash^i (q_f, \epsilon)$ ,  $q_f \in F$

# Regular sets

**Definition:** Let  $\Sigma$  be a finite alphabet. We define regular sets over  $\Sigma$  recursively in the following way:

1.  $\Phi$  is a regular set over  $\Sigma$  (empty set)
2.  $\{\epsilon\}$  is a regular set over  $\Sigma$
3.  $\{a\}$  is a regular set over  $\Sigma$ ,  $\forall a \in \Sigma$
4. If  $P, Q$  are regular sets over  $\Sigma$ , then  $P \cup Q$ ,  $PQ$ ,  $P^*$  are regular sets over  $\Sigma$
5. Nothing else is a regular set over  $\Sigma$

# Regular expressions

**Definition:** Let  $\Sigma$  be a finite alphabet. We define regular expressions over  $\Sigma$  recursively in the following way:

1.  $\Phi$  is a regular expression denoting the regular set  $\Phi$  (empty set)
2.  $\epsilon$  is a regular expression denoting the regular set  $\{\epsilon\}$
3.  $a$  is a regular expression denoting the regular set  $\{a\}$ ,  $\forall a \in \Sigma$
4. If  $p, q$  are regular expression denoting the regular sets  $P, Q$  then:
  - $p+q$  is a regular expression denoting the regular set  $P \cup Q$ ,
  - $pq$  is a regular expression denoting the regular set  $PQ$ ,
  - $p^*$  is a regular expression denoting the regular set  $P^*$
5. Nothing else is a regular expression

# Remarks:

## Examples

1.  $p^+ = pp^*$
2. Use parenthesis to avoid ambiguity
3. Priority of operations: \*, concat, + (from high to low)
4. For each regular set we can find at least one regular exp to denote it (there is an infinity of reg exp denoting them)
5. For each regular exp, we can construct the corresponding regular set
6. 2 regular expressions are **equivalent** iff they denote the same regular set

# Algebraic properties of regular exp

Let  $\alpha, \beta, \gamma$  be regular expressions.

$$1. \alpha + \beta = \beta + \alpha$$

$$2. \Phi^* = \epsilon$$

$$3. \alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma$$

$$4. \alpha(\beta\gamma) = (\alpha\beta)\gamma$$

$$5. \alpha(\beta + \gamma) = \alpha\beta + \alpha\gamma$$

$$6. (\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma$$

$$7. \alpha \epsilon = \epsilon \alpha = \alpha$$

$$8. \Phi\alpha = \alpha\Phi = \Phi$$

$$9. \alpha^* = \alpha + \alpha^*$$

$$10. (\alpha^*)^* = \alpha^*$$

$$11. \alpha + \alpha = \alpha$$

$$12. \alpha + \Phi = \alpha$$

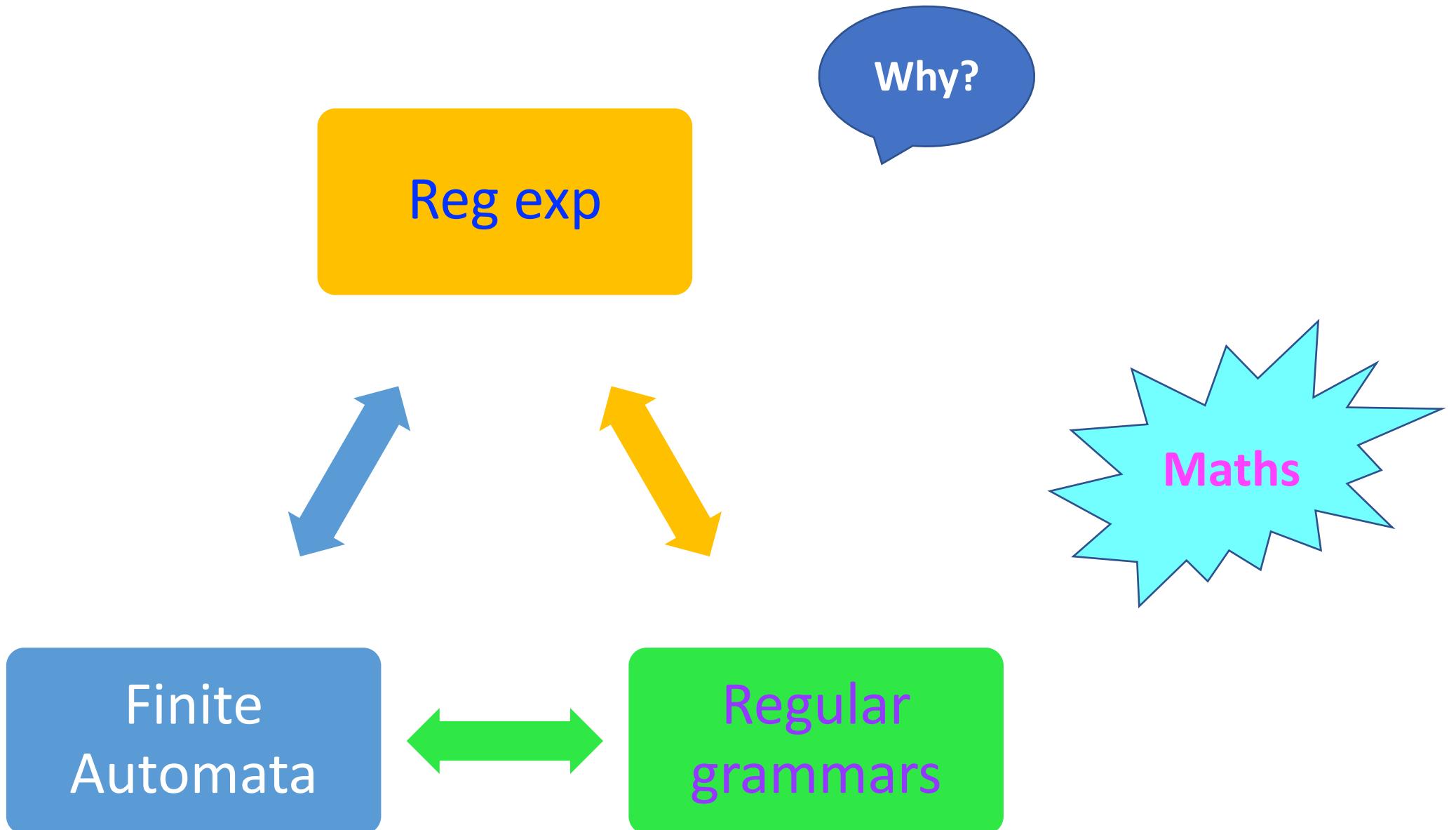
# Reg exp equations

- Normal form:  $X = aX + b$   
where  $a, b$  – reg exp

- Solution:  $X = a^*b$

$$a a^*b + b = (aa^* + \epsilon)b = a^*b$$

- System of reg exp equations:
$$\begin{cases} X = a_1X + a_2Y + a_3 \\ Y = b_1X + b_2Y + b_3 \end{cases}$$
- Solution: Gauss method (replace  $X_i$  and solve  $X_n$ )



**Prop:** Regular sets are right linear languages

**Lemma 1:**  $\Phi, \{\epsilon\}, \{a\}, \forall a \in \Sigma$  are right linear languages

Proof: constructive

- i.  $G = (\{S\}, \Sigma, \Phi, S)$  – regular grammar such that  $L(G) = \Phi$
- ii.  $G = (\{S\}, \Sigma, \{S \rightarrow \epsilon\}, S)$  – regular grammar such that  $L(G) = \{\epsilon\}$
- iii.  $G = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$  – regular grammar such that  $L(G) = \{a\}$

**Lemma 2:** If  $L_1$  and  $L_2$  are right linear languages then:

$L_1 \cup L_2$ ,  $L_1L_2$  and  $L_1^*$  are right linear languages.

Proof: constructive

$L_1, L_2$  right linear languages  $\Rightarrow \exists G_1, G_2$  such that

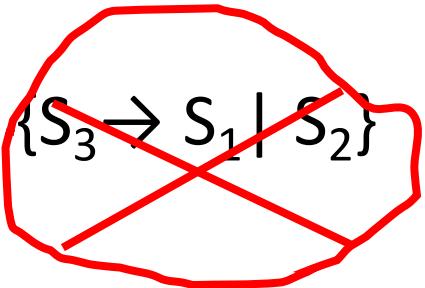
$G_1 = (N_1, \Sigma_1, P_1, S_1)$  and  $L_1 = L(G_1)$

$G_2 = (N_2, \Sigma_2, P_2, S_2)$  and  $L_2 = L(G_2)$       assume  $N_1 \cap N_2 = \emptyset$

i.  $G_3 = (N_3, \Sigma, P_3, S_3)$

$$N_3 = N_1 \cup N_2 \cup \{S_3\}; \Sigma_3 = \Sigma_1 \cup \Sigma_2$$

$$P_3 = P_1 \cup P_2 \cup \{S_3 \rightarrow S_1 | S_2\}$$



$$\{S_3 \rightarrow \alpha_1 | S_1 \rightarrow \alpha_1 \in P_1\} \cup \{S_3 \rightarrow \alpha_2 | S_2 \rightarrow \alpha_2 \in P_2\}$$

$G_3$  – right linear language

and

$$L(G_3) = L(G_1) \cup L(G_2)$$

**PROOF!!! Homework**

ii.  $G_4 = (N_4, \Sigma, P_4, S_4)$

$$N_4 = N_1 \cup N_2; S_4 = S_1; \Sigma_4 = \Sigma_1 \cup \Sigma_2$$

$$\begin{aligned} P_4 = & \{ A \rightarrow aB \mid \text{if } A \rightarrow aB \in P_1 \} \cup \\ & \{ A \rightarrow aS_2 \mid \text{if } A \rightarrow a \in P_1 \} \cup \\ & P_2 \cup \\ & \{ S_1 \rightarrow \alpha_2 \mid \text{if } S_1 \rightarrow \epsilon \in P_1 \text{ and } S_2 \rightarrow \alpha_2 \in P_2 \} \end{aligned}$$

$G_4$  – right linear language  
and

$$L(G_4) = L(G_1) L(G_2)$$

**PROOF!!! Homework**

iii.  $G_5 = (N_5, \Sigma_1, P_5, S_5)$

//IDEA: concatenate  $L_1$  with itself

$N_4 = N_1 \cup \{S_5\};$

$P_5 = P_1 \cup \{S_5 \rightarrow \epsilon\} \cup$   
 $\{S_5 \rightarrow \alpha_1 \mid S_1 \rightarrow \alpha_1 \in P_1\} \cup$   
 $\{A \rightarrow aS_1 \mid \text{if } A \rightarrow a \in P_1\}$

$G_5$  – right linear language  
and

$L(G_5) = L(G_1)^*$

**PROOF!!! Homework**

**Theorem:** A language is a regular set if and only if it is a right linear language

Proof:

=> Apply lemma 1 and lemma 2

<= construct a system of regular exp equations where:

- Indeterminants – nonterminals
- Coefficients – terminals
- Equation for A: all the possible rewritings of A

Example:  $G = (\{S, A, B\}, \{0, 1\}, P, S)$

P:  $S \rightarrow 0A \mid 1B \mid \epsilon$

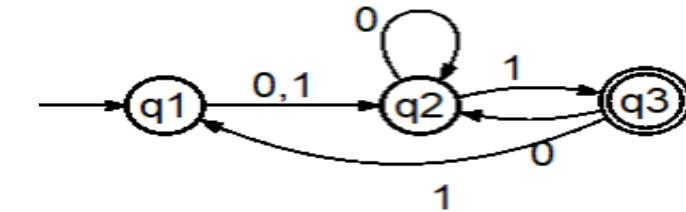
$A \rightarrow 0B \mid 1A$

$B \rightarrow 0S \mid 1$

$$\begin{cases} S = 0A + 1B + \epsilon \\ A = 0B + 1A \\ B = 0S + 1 \end{cases}$$

Regular exp = solution  
corresponding to S

*Theorem: A language is a regular set if and only if it is accepted by a FA*



Proof:

=> Apply lemma 1 and lemma 2 (to follow, similar to RG)

<= construct a system of regular exp equations where:

- Indeterminants – states
- Coefficients – terminals
- Equation for A: all the possibilities that put the FA in state A
- Equation of the form:  $X = Xa + b \Rightarrow$  solution  $X = ba^*$

$$\begin{cases} q_1 = q_30 + \epsilon \\ q_2 = q_10 + q_11 + q_20 + q_30 \\ q_3 = q_21 \end{cases}$$

**Regular exp = union of  
solutions corresponding  
to final states**

**Lemma 1':**  $\Phi, \{\epsilon\}, \{a\}, \forall a \in \Sigma$  are accepted by FA

Reg exp	FA
$\Phi$	$M = (Q, \Sigma, \delta, q_0, \Phi)$
$\epsilon$	$M = (Q, \Sigma, \Phi, q_0, \{q_0\})$
$a, \forall a \in \Sigma$	$M = (\{q_0, q_1\}, \Sigma, \{\delta(q_0, a) = q_1\}, q_0, \{q_1\})$

**Lemma 2'**: If  $L_1$  and  $L_2$  are accepted by a FA then:  
 $L_1 \cup L_2$ ,  $L_1L_2$  and  $L_1^*$  are accepted by FA

Proof:

$M_1 = (Q_1, \Sigma_1, \delta_1, q_{01}, F_1)$  such that  $L_1 = L(M_1)$

$M_2 = (Q_2, \Sigma_2, \delta_2, q_{02}, F_2)$  such that  $L_2 = L(M_2)$

$M_3 = (Q_3, \Sigma_{1\cup}, \delta_3, q_{03}, F_3)$

$Q_3 = Q_1 \cup Q_2 \cup \{q_{03}\}$ ;  $\Sigma_3 = \Sigma_1 \cup \Sigma_2$

$F_3 = F_1 \cup F_2 \cup \{q_{03} \mid \text{if } q_{01} \in F_1 \text{ or } q_{02} \in F_2\}$

$\delta_3 = \delta_1 \cup \delta_2 \cup \{\delta_3(q_{03}, a) = p \mid \exists \delta_1(q_{01}, a) = p\} \cup$   
 $\{\delta_3(q_{03}, a) = p \mid \exists \delta_2(q_{02}, a) = p\}$

$$L(M_3) = L(M_1) \cup L(M_2)$$

**PROOF!!! Homework**

$$M_4 = (Q_4, \Sigma_4, \delta_4, q_{04}, F_4)$$

$$Q_4 = Q_1 \cup Q_2; \quad q_{04} = q_{01};$$

$$F_3 = F_2 \cup \{q \in F_1 \mid \text{if } q_{02} \in F_2\}$$

$$\delta_3(q,a) = \delta_1(q,a), \text{ if } q \in Q_1 - F_1$$

$$\delta_1(q,a) \cup \delta_2(q_{02},a) \text{ if } q \in F_1$$

$$\delta_2(q,a), \text{ if } q \in Q_2$$

$$L(M_3) = L(M_1)L(M_2)$$

**PROOF!!! Homework**

$$M_5 = (Q_5, \Sigma_1, \delta_5, q_{05}, F_5)$$

//IDEA: concatenate with itself

$$Q_5 = Q_1; \quad q_{05} = q_{01}$$

$$F_5 = F_1 \cup \{q_{01}\}$$

$$\delta_5(q,a) = \delta_1(q,a), \text{ if } q \in Q_1 - F_1$$

$$\delta_1(q,a) \cup \delta_1(q_{01},a) \text{ if } q \in F_1$$

$$L(M_3) = L(M_1)^*$$

PROOF!!! Homework

# Course 5

# Pumping Lemma

- Not all languages are regular
- How to decide if a language is regular or not?
- Idea: pump symbols

Example:  $L = \{0^n 1^n \mid n \geq 0\}$

## **Theorem:** (Pumping lemma, Bar-Hillel)

Let  $L$  be a regular language.  $\exists p \in N$ , such that if  $w \in L$  with  $|w| > p$ , then

$w = xyz$ , where  $0 < |y| \leq p$

and

$xy^i z \in L, \forall i \geq 0$

# Proof

$L$  regular  $\Rightarrow \exists M = (Q, \Sigma, \delta, q_0, F)$  such that  $L = L(M)$

Let  $|Q| = p$

If  $w \in L(M)$ :  $(q_0, w) \xrightarrow{*} (q_f, \epsilon)$ ,  $q_f \in F$

and

$|w| > p$

] process at least  $p+1$  symbols  
                          ]  $p$  states

$\Rightarrow \exists q_1$  that appear in at least 2 configurations

$(q_0, xyz) \xrightarrow{*} (q_1, yz) \xrightarrow{*} (q_1, z) \xrightarrow{*} (q_f, \epsilon)$ ,  $q_f \in F \Rightarrow 0 \leq |y| \leq p$

# Proof (cont)

$(q_0, xy^i z) \xrightarrow{*} (q_1, y^i z)$   
 $\xrightarrow{*} (q_1, y^{i-1} z)$   
 $\xrightarrow{*} \dots$   
 $\xrightarrow{*} (q_1, yz)$   
 $\xrightarrow{*} (q_1, z)$   
 $\xrightarrow{*} (q_f, \epsilon), q_f \in F$

So, if  $w=xyz \in L$  then  $xy^i z \in L$ , for all  $i > 0$

If  $i=0$ :  $(q_0, xz) \xrightarrow{*} (q_1, z) \xrightarrow{*} (q_f, \epsilon), q_f \in F$

*Example:*  $L = \{0^n 1^n \mid n \geq 0\}$

Suppose  $L$  is regular  $\Rightarrow w = xyz = 0^n 1^n$

Consider all possible decomposition  $\Rightarrow$

Case 1.  $y = 0^k$

$$xyz = 0^{n-k} 0^k 1^n; xy^i z = 0^{n-k} 0^{ik} 1^n \notin L$$

Case 2.  $y = 1^k$

$$xyz = 0^n 1^k 1^{n-k}; xy^i z = 0^n 1^{ik} 1^{n-k} \notin L$$

**=>  $L$  is not regular**

Case 3.  $y = 0^k 1^l$

$$xyz = 0^{n-k} 0^k 1^l 1^{n-l}; xy^i z = 0^{n-k} (0^k 1^l)^i 1^{n-l} \notin L$$

Case 4.  $y = 0^k 1^k$

$$xyz = 0^{n-k} 0^k 1^k 1^{n-k}; xy^i z = 0^{n-k} 0^k 1^k 0^k 1^k \dots 1^{n-l} \notin L$$

# Context free grammars (cfg)

# Context free grammar (cfg)

- Productions of the form:  $A \rightarrow \alpha$ ,  $A \in N$ ,  $\alpha \in (N \cup \Sigma)^*$
- More powerful
- Can model programming language:  
 $G = (N, \Sigma, P, S)$  s.t.  $L(G) = \text{programming language}$

# Syntax tree

**Definition:** A syntax tree corresponding to a cfg  $G = (N, \Sigma, P, S)$  is a tree obtained in the following way:

1. Root is the starting symbol  $S$
2. Nodes  $\in N \cup \Sigma$ :
  1. Internal nodes  $\in N$
  2. Leaves  $\in \Sigma$
3. For a node  $A$  the descendants in order from left to right are  $X_1, X_2, \dots, X_n$  only if  $A \rightarrow X_1 X_2 \dots X_n \in P$

## Remarks:

- a) Parse tree = syntax tree – result of parsing (syntactic analysis)
- b) Derivation tree – condition 2.2 not satisfied
- c) Abstract syntax tree (AST)  $\neq$  syntax tree (semantic analysis)

# Syntax tree (cont)

**Property:** In a cfg  $G = (N, \Sigma, P, S)$ ,  $w \in L(G)$  if and only if there exists a syntax tree with frontier  $w$ .

Proof: HomeWork

Example:  $S \rightarrow aSbS \mid c$ ;  $w = aacbcabc$

**Leftmost derivations**

$S \Rightarrow aSbS \Rightarrow aaSbSbS \Rightarrow aacbSbS$   
 $\Rightarrow aacbcbS \Rightarrow aacbcabc$

**Rightmost derivations**

$S \Rightarrow aSbS \Rightarrow aSbc \Rightarrow aaSbSbc$   
 $\Rightarrow aaSbcabc \Rightarrow aacbcabc$

**Definition:** A cfg  $G = (N, \Sigma, P, S)$  is ambiguous if for a  $w \in L(G)$  there exists 2 distinct syntax tree with frontier  $w$ .

Example:

# Parsing (syntax analysis) modeled with cfg:

cfg  $G = (N, \Sigma, P, S)$ :

- $N$  – nonterminal: syntactical constructions: declaration, statement, expression, a.s.o.
- $\Sigma$  – terminals; elements of the language: identifiers, constants, reserved words, operators, separators
- $P$  – syntactical rules – expressed in BNF – simple transformation
- $S$  – syntactical construct corresponding to program

THEN

Program syntactically correct  $\Leftrightarrow w \in L(G)$

# Equivalent transformation of cfg

- Unproductive symbols
  - Inaccessible symbols
  - $\epsilon$  - productions
  - Single productions
1. Determine elements (symbols/productions): Greedy alg
  2. eliminate them: construct equivalent grammar

# Unproductive symbols

## Definition

A nonterminal A este **unproductive** in a cfg if does not generate any word:  $\{w \mid A \Rightarrow^* w, w \in \Sigma^*\} = \emptyset$ .

## **Algorithm 1: Elimination of unproductive symbols**

*input:*  $G = (N, \Sigma, P, S)$

*output:*  $G' = (N', \Sigma, P', S)$ ,  $L(G) = L(G')$

// idea: build  $N_0, N_1, \dots$  recursively (until saturation)

step 1:  $N_0 = \emptyset$ ;  $i := 1$ ;

step 2:  $N_i = N_{i-1} \cup \{A \mid A \rightarrow \alpha \in P, \alpha \in (N_{i-1} \cup \Sigma)^*\}$

step 3: if  $N_i \neq N_{i-1}$  then  $i := i + 1$ ; goto step 2

else  $N' = N_i$

step 4: if  $S \notin N'$  then  $L(G) = \emptyset$

else  $P' = \{A \rightarrow \alpha \mid A \rightarrow \alpha \in P \text{ and } A \in N'\}$

# Example

$G = (\{S, A, B, C, D\}, \{a, b, c\}, P, S)$

$P: \quad S \rightarrow aA \mid aC$

$A \rightarrow AB$

$B \rightarrow b$

$C \rightarrow aC \mid CD$

$D \rightarrow b$

# Inaccessible symbols

## Definition

A symbol  $X \in N \cup \Sigma$  is **inaccessible** in a cfg if  $X$  does not appear in any sentential form:  $\forall S \Rightarrow^* \alpha, X \notin \alpha$

## **Algorithm 2: Elimination of inaccessible symbols**

*input:*  $G = (N, \Sigma, P, S)$

*output:*  $G' = (N', \Sigma', P', S)$ ,  $L(G) = L(G')$  and

$$\forall X \in N \cup \Sigma \exists \alpha, \beta \in (N' \cup \Sigma')^* \text{ s.t. } S \Rightarrow^{*}_{G'} \alpha X \beta.$$

step 1:  $V_0 = \{S\}$ ;  $i := 1$ ;

step 2:  $V_i = V_{i-1} \cup \{X \mid \exists A \rightarrow \alpha X \beta \in P, A \in V_{i-1}\}$

step 3: if  $V_i \neq V_{i-1}$  then  $i := i + 1$ ; goto step 2

else  $N' = N \cap V_i$

$$\Sigma' = \Sigma \cap V_i$$

$$P' = \{A \rightarrow \alpha \mid A \rightarrow \alpha \in P, A \in N', \alpha \in (N \cup \Sigma)^*\}$$

# Example

$$G = (\{S, A, B, C, D\}, \{a, b, c, d\}, P, S)$$

$$P: \quad S \rightarrow aA \mid aC$$

$$A \rightarrow AB$$

$$B \rightarrow b$$

$$C \rightarrow aC \mid bCb$$

$$D \rightarrow bB \mid d$$

# $\epsilon$ -productions

## **Algorithm 3: Elimination of $\epsilon$ -productions**

*input: cfg  $G = (N, \Sigma, P, S)$*

*output: cfg  $G' = (N', \Sigma, P', S')$*

step 1: construct  $\bar{N} = \{A \mid A \in N, A \Rightarrow^+ \epsilon\}$

1.a.  $N_0 := \{A \mid A \rightarrow \epsilon \in P\};$   
 $i := 1;$

1.b.  $N_i := N_{i-1} \cup \{A \mid A \rightarrow \alpha \in P, \alpha \in N_{i-1}^*\}$   
1.c. **if**  $N_i \neq N_{i-1}$  **then**  $i := i + 1$ ; **goto** step 1.b  
**else**  $\bar{N} = N_i$

A->BC  
B-> $\epsilon$   
C-> $\epsilon$

## **Definition**

A cfg  $G = (N, \Sigma, P, S)$  is without  **$\epsilon$ -productions** if

1.  $P \not\ni A \rightarrow \epsilon$  ( $\epsilon$ -productions)

OR

2.  $\exists S \rightarrow \epsilon$  si  $S \notin \text{rhs}(p), \forall p \in P$

step 2: Let  $P'$  = set of productions built:

2.a. **if**  $A \rightarrow \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_k \alpha_k \in P, k >= 0$   
**and** for  $i := 1, k$   $B_i \in \bar{N}$   
and  $\alpha_j \notin \bar{N}, j := 0, k$

**then** add to  $P'$  all prod of the form

$A \rightarrow \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \dots X_k \alpha_k$   
where  $X_i$  is  $B_i$  or  $\epsilon$  (not  $A \rightarrow \epsilon$ )

2.b **if**  $S \in N'$  **then** add  $S'$  to  $N'$  and  $S' \rightarrow S | \epsilon$  to  $P'$   
**else**  $N' := N$ ;  $S' := S$ .

# Example

$$G = (\{S, A, B\}, \{a, b\}, P, S)$$

$$P: \quad S \rightarrow aA \mid aAbB$$

$$A \rightarrow aA \mid B$$

$$B \rightarrow bB \mid \epsilon$$

# Single productions

## Definition

O production of the form  $A \rightarrow B$  is called single production or renaming rule.

### **Algorithm 4 : Elimination of single productions**

*Input:* cfg G, without  $\epsilon$ -productions

*Output:*  $G'$  s.t.  $L(G) = L(G')$

For each  $A \in N$  build the set  $N_A = \{B \mid A \Rightarrow^* B\}$  :

1.a.  $N_0 := \{A\}$ ,  $i := 1$

1.b.  $N_i := N_{i-1} \cup \{C \mid B \rightarrow C \in P \text{ si } B \in N_{i-1}\}$

1.c. **if**  $N_i \neq N_{i-1}$  **then**  $i := i + 1$  **goto** 1.b.

**else**  $N_A := N_i$

$P'$ : **for** all  $A \in N$  **do**

**for** all  $B \in N_A$  **do**

**if**  $B \rightarrow \alpha \in P$  **and not** “single” **then**  $A \rightarrow \alpha \in P'$

$G' = (N, \Sigma, P', S)$

# Example

$$G = (\{E, T, F\}, \{a, (,), +, *\}, P, E)$$

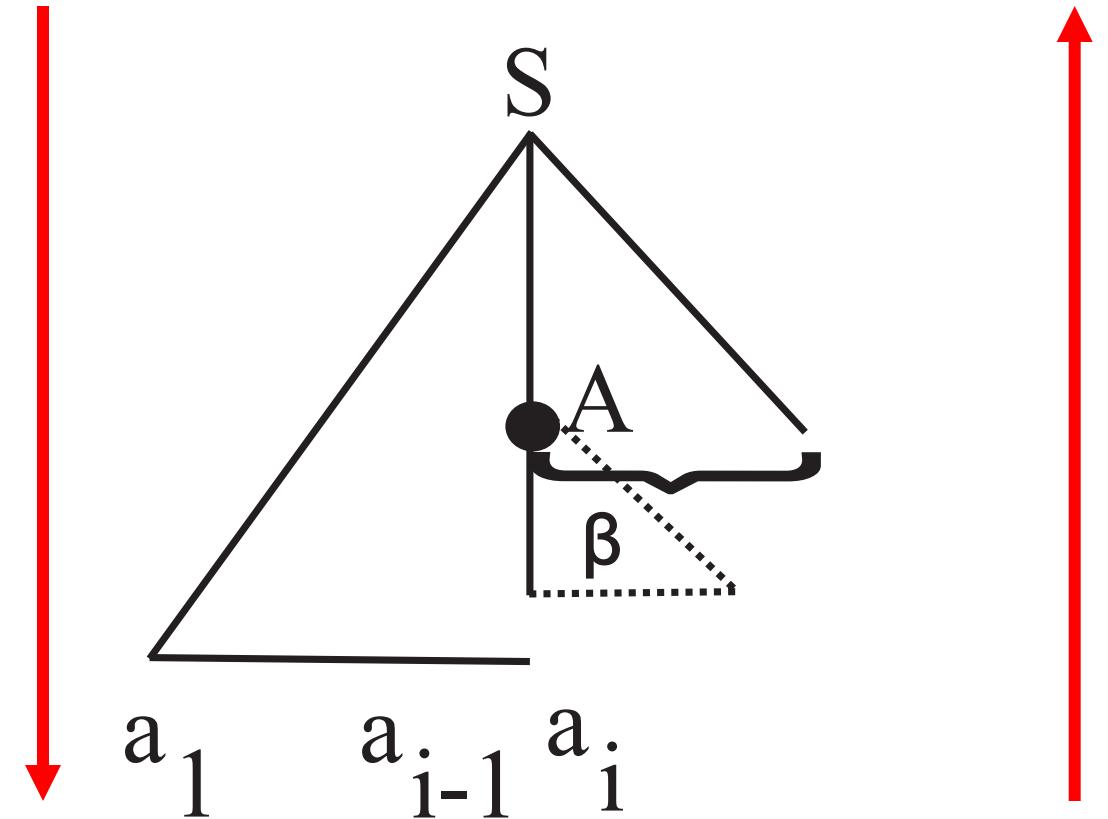
$$P: \quad E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid a$$

# Parsing

- Cfg  $G = (N, \Sigma, P, S)$  check if  $w \in L(G)$
- Construct parse tree
- How:
  1. Top-down vs. Bottom-up
  2. Recursive vs. linear



# Course 6

# Problem: Parsing (construct the parsee tree)

**if** the *source program is syntactically correct*

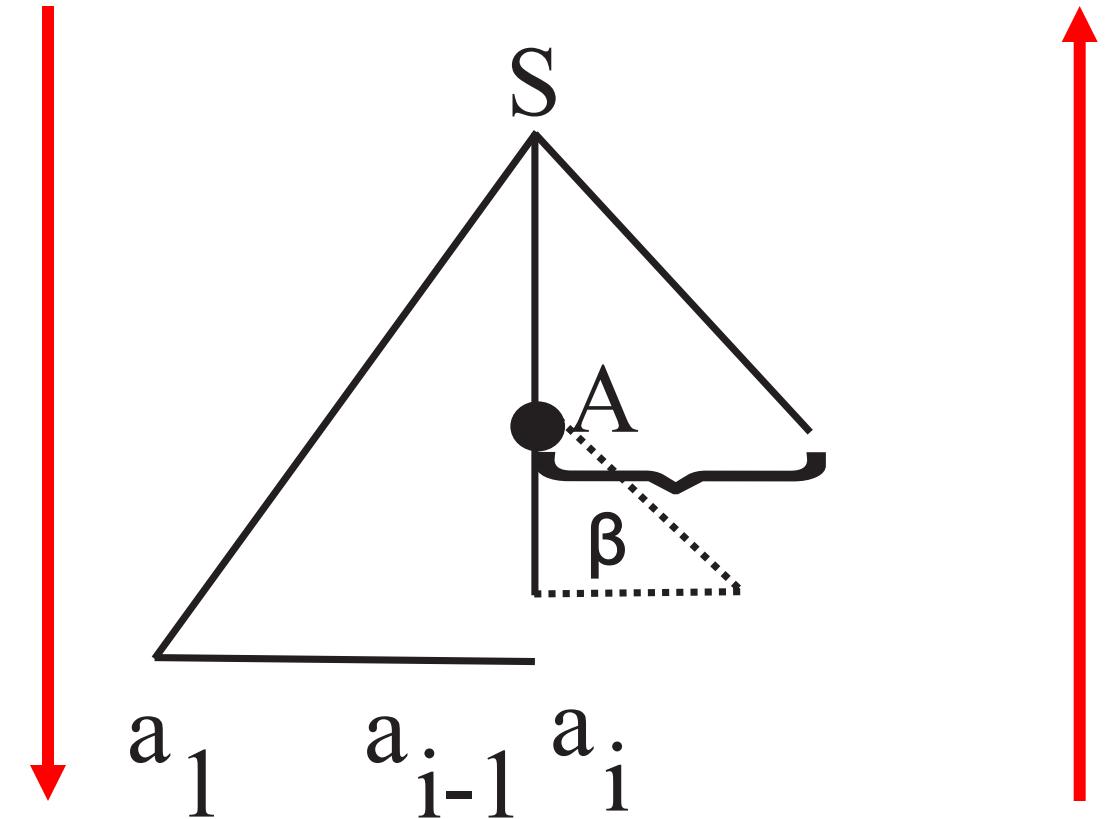
**then** construct syntax tree

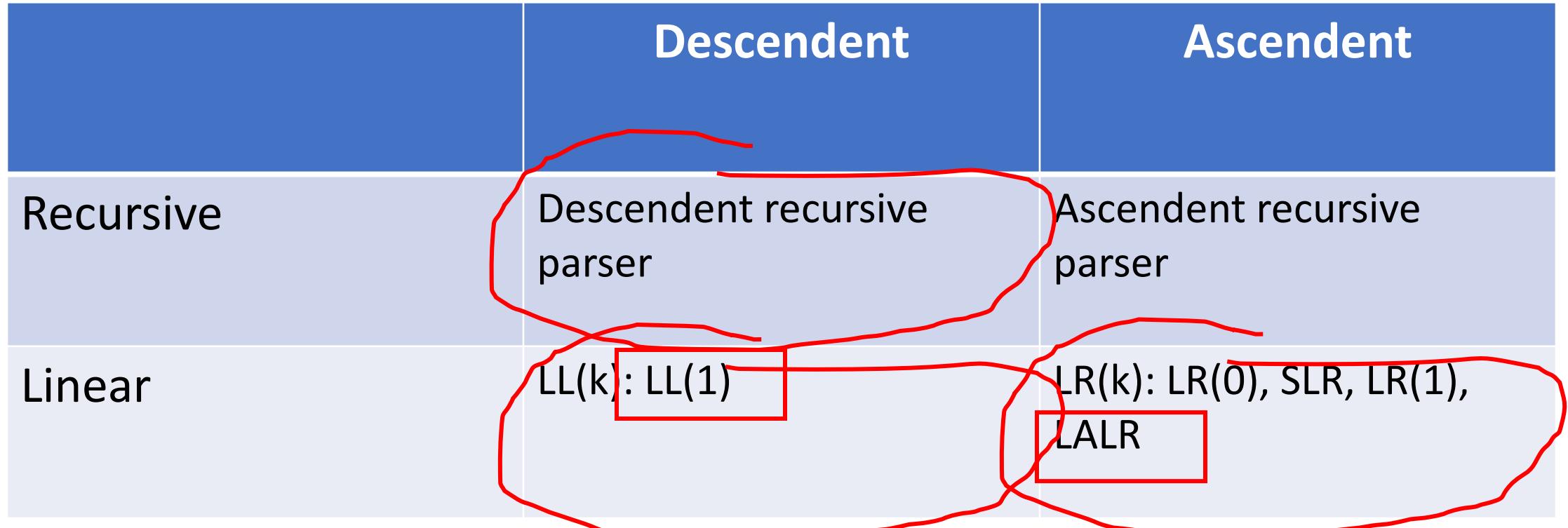
**else** "syntax error"

*source program is syntactically correct* =  $w \in L(G)$   $\Leftrightarrow S \xrightarrow{*} w$

# Parsing

- How:
  1. Top-down vs. Bottom-up
  2. Recursive vs. linear

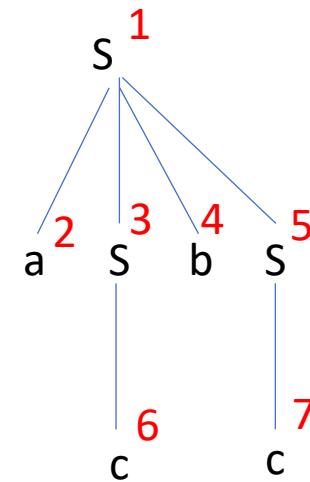




# Result – parse tree -representation

- Arbitrary tree – child sibling representation
- Sequence of derivations  $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n = w$
- String of production – index associated to prod – which prod is used at each derivation step: 1,4,3,...

index	Info	Parent	Right sibling
1	S	0	0
2	a	1	0
3	S	1	2
4	b	1	3
5	S	1	4
6	c	3	0
7	c	5	0



# Descendent recursive parser

- Example

$S \rightarrow aSbS \mid aS \mid c$

# Formal model

- Configuration

$$(s, i, \alpha, \beta)$$

Initial configuration:  
 $(q, 1, \varepsilon, S)$

where:

- $s$  = state of the parsing, can be:
  - $q$  = normal state
  - $b$  = back state
  - $f$  = final state - corresponding to success:  $w \in L(G)$
  - $e$  = error state – corresponding to insuccess:  $w \notin L(G)$
- $i$  – position of current symbol in input sequence  
 $w = a_1 a_2 \dots a_n, i \in \{1, \dots, n+1\}$
- $\alpha$  = working stack, stores the way the parse is built
- $\beta$  = input stack, part of the tree to be built

Define moves between configurations

Final configuration:  
 $(f, n+1, \alpha, \varepsilon)$

# Expand

WHEN: head of input stack is a nonterminal

$$(q,i, \alpha, A\beta) \vdash (q,i, \alpha A_1, \gamma_1 \beta)$$

where:

$A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots$  represents the productions corresponding to A

$\gamma_1$  = first prod of A

# Advance

WHEN: head of input stack is a terminal = current symbol from input

$$(q, i, \alpha, a_i \beta) \vdash (q, i+1, \alpha a_i, \beta)$$

# Momentary insuccess

WHEN: head of input stack is a terminal  $\neq$  current symbol from input

$$(q,i, \alpha, a_i \beta) \vdash (\textcolor{red}{b}, i, \alpha, a_i \beta)$$

# Back

WHEN: head of working stack is a terminal

$$(b, i, \alpha a, \beta) \leftarrow (b, i-1, \alpha, a\beta)$$

# Another try

WHEN: head of working stack is a nonterminal

$(b, i, \alpha A_j, \gamma_j \beta) \leftarrow (q, i, \alpha A_{j+1}, \gamma_{j+1} \beta)$ , if  $\exists A \rightarrow \gamma_{j+1}$

$(b, i, \alpha, A \beta)$ , otherwise with the exception

$(e, i, \alpha, \beta)$ , if  $i=1$ ,  $A=S$ , **ERROR**

# Success

$(q, n+1, \alpha, \varepsilon) \vdash (\textcolor{red}{f}, n+1, \alpha, \varepsilon)$

# Algorithm

## Algorithm Descendent Recursive

---

**INPUT:**  $G, w = a_1 a_2 \dots a_n$

**OUTPUT:** string of productions and message

config =  $(q, 1, \epsilon, S)$ ;

//initial configuration  $(s, i, \alpha, \beta)$

**while**  $(s \neq f)$  and  $(s \neq e)$  **do**

**if**  $s = q$

**then if**  $(i=n+1)$  and IsEmpty $(\beta)$

**then Success(config)**

**else**

**if Head** $(\beta) = A$

**then Expand(config)**

**else**

**if Head** $(\beta) = a_i$

**then Advance(config)**

**else MomentaryInsucces** $(config)$

**else**

**if**  $s = b$

**then**

**if Head** $(\alpha) = a$

**then Back(config)**

**else AnotherTry** $(config)$

**endWhile**

**if**  $s = e$  **then** message "Error"

**else** message "Sequence accepted";

BuildStringOfProd $(\alpha)$

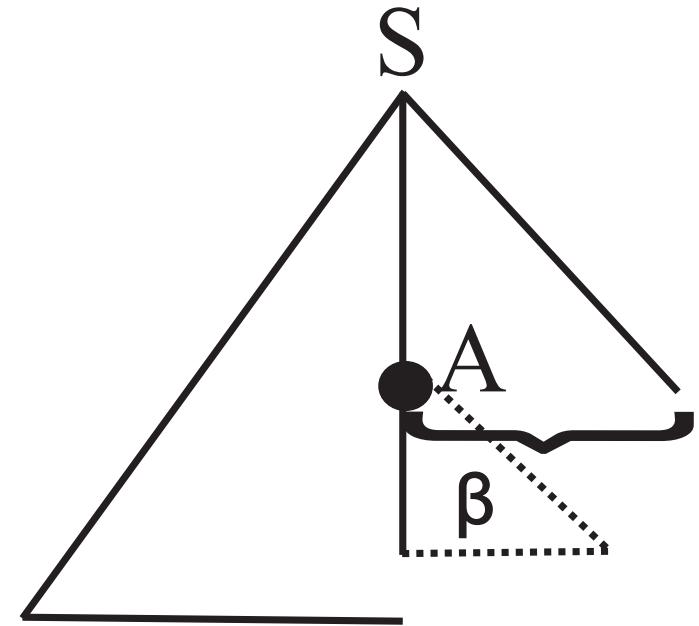
# $w \in L(G)$ - HOW

- Process  $\alpha$ :
  - From left to right (reverse if stored as stack)
  - Skip terminal symbols
  - Nonterminals – index of prod
- Example:  $\alpha = S_1 a S_2 a S_3 c b S_3 c$

# When the algorithm never stops?

- $S \rightarrow S\alpha$  – expand infinitely (left recursive)

# LL(1) Parser



$a_1 \quad a_{i-1} \quad a_i$

Linear algorithm

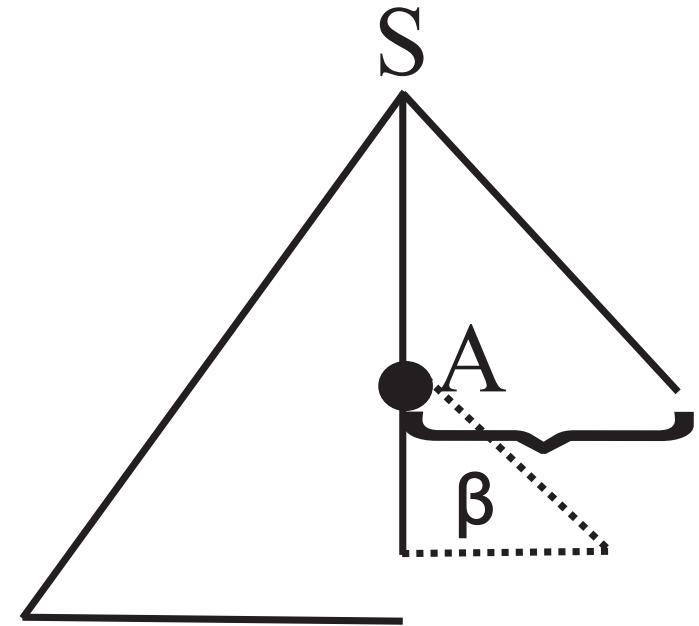
# FIRST<sub>k</sub>

- $\approx$  first k terminal symbols that can be generated from  $\alpha$
- **Definition:**

$$FIRST_k : (N \cup \Sigma)^* \rightarrow \mathcal{P}(\Sigma^k)$$

$$FIRST_k(\alpha) = \{u \mid u \in \Sigma^k, \alpha \xrightarrow{*} ux, |u| = k \text{ sau } \alpha \xrightarrow{*} u, |u| \leq k\}$$

# LL(1) Parser



$a_1 \quad a_{i-1} \quad a_i$

Linear algorithm

Operation:  $\oplus$  = concatenation of length 1

$$L_1 = \{aa, ab, ba\}$$

$$L_2 = \{00, 01\}$$

$$L_1 \oplus L_2 = \{a, 0\}$$

$$L_1 = \{a, \epsilon\}$$

$$L_2 = \{0, 1\}$$

$$L_1 \oplus L_2 = \{a, 0, 1\}$$

# FIRST<sub>k</sub>

- $\approx$  first k terminal symbols that can be generated from  $\alpha$
- **Definition:**

$$FIRST_k : (N \cup \Sigma)^* \rightarrow \mathcal{P}(\Sigma^k)$$

$$FIRST_k(\alpha) = \{u \mid u \in \Sigma^k, \alpha \xrightarrow{*} ux, |u| = k \text{ sau } \alpha \xrightarrow{*} u, |u| \leq k\}$$

# FIRST<sub>k</sub>

- Which are the first k terminal symbols that can be generated from A?
- <https://forms.office.com/r/kNHNGW7XtC>

# Construct FIRST

➤ FIRST<sub>1</sub> denoted FIRST

➤ Remarks:

- If  $L_1, L_2$  are 2 languages over alphabet  $\Sigma$ , then :  $L_1 \oplus L_2 = \{w|x \in L_1, y \in L_2, xy = w, |w| \leq 1 \text{ sau } xy = wz, |w| = 1\}$  and

- $FIRST(\alpha\beta) = FIRST(\alpha) \oplus FIRST(\beta)$

$$FIRST(X_1 \dots X_n) = FIRST(X_1) \oplus \dots \oplus FIRST(X_n)$$

Concatenation  
of length 1

---

### Algoritmul 3.3 FIRST

---

**INPUT:** G

**OUTPUT:**  $FIRST(X), \forall X \in N \cup \Sigma$

**for**  $\forall a \in \Sigma$  **do**

$F_i(a) = \{a\}, \forall i \geq 0$

**end for**

$i := 0;$

$F_0(A) = \{x | x \in \Sigma, A \rightarrow x\alpha \text{ sau } A \rightarrow x \in P\}; \{\text{initializare}\}$

**repeat**

$i := i + 1;$

**for**  $\forall X \in N$  **do**

**if**  $F_{i-1}$  au fost calculate  $\forall X \in N \cup \Sigma$  **then**

            {dacă  $\exists Y_j, F_{i-1}(Y_j) = \emptyset$  atunci nu se poate aplica}

$F_i(A) = F_{i-1}(A) \cup$

$\{x | A \rightarrow Y_1 \dots Y_n \in P, x \in F_{i-1}(Y_1) \oplus \dots \oplus F_{i-1}(Y_n)\}$

**end if**

**end for**

**until**  $F_{i-1}(A) = F_i(A)$

$FIRST(X) := F_i(X), \forall X \in N \cup \Sigma$

---

A  $\rightarrow$  BC

B  $\rightarrow$  DA

D  $\rightarrow$  a

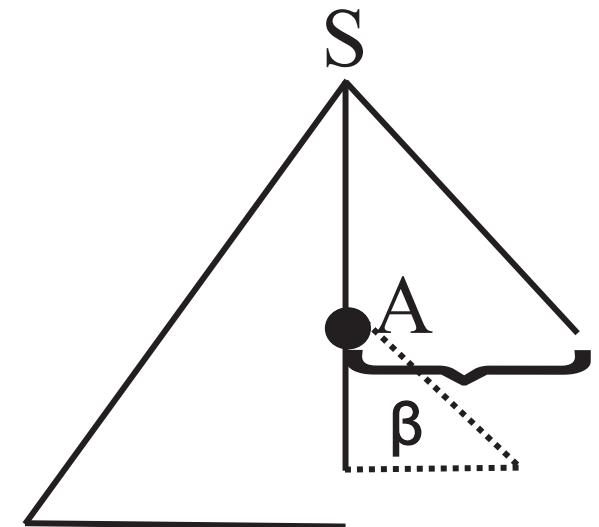
$F_0(A) = F_0(B) = \emptyset; F_0(D) = \{a\}$

$F_1(A) = F_0(A) \cup \{ \dots | A \rightarrow BC \} F_0(B) \oplus F(D) = \emptyset$

$F_1(B) = \{a\}$

# FOLLOW

$A \rightarrow \epsilon$



$a_1 \quad a_{i-1} \quad a_i$

➤ FOLLOW<sub>k</sub>(A)≈ next k symbols generated after/ following A

Follow(A)  
 $S \Rightarrow^* xBy \Rightarrow xaAy$   
What if  $B \Rightarrow uA$

$FOLLOW : (N \cup \Sigma)^* \rightarrow \mathcal{P}(\Sigma)$

$FOLLOW(\beta) = \{w \in \Sigma | S \xrightarrow{*} \alpha\beta\gamma, w \in FIRST(\gamma)\}$

## Algorithm FOLLOW

**INPUT:** G, FIRST(X),  $\forall X \in N \cup \Sigma$

**OUTPUT:** FOLLOW(A),  $\forall A \in N$

**for**  $A \in N - \{S\}$  **do**

{init}

$L_0(A) = \Phi;$

**endFor;**

$L_0(S) = \{\epsilon\};$

{init}

$i = 0;$

$S \Rightarrow^0 S // \epsilon \text{ after } S$

**repeat**

$i = i + 1;$

**for**  $B \in N$  **do**

**for**  $A \rightarrow \alpha By \in P$  **do**

**for**  $\forall a \in FIRST(y)$  **do**

**if**  $a = \epsilon$  **then**  $F_i(B) = F_i(B) \cup F_{i-1}(A)$

**else**  $F_i(B) = F_{i-1}(B) \cup FIRST(y)$

**endif**

**endFor**

**endFor**

**endfor**

**until**  $F_i(X) = F_{i-1}(X), \forall X \in N$

$FOLLOW(X) = F_i(X), \forall X \in N$

$S \Rightarrow aAc \Rightarrow abBc$   
 $A \rightarrow bB$

# FIRST

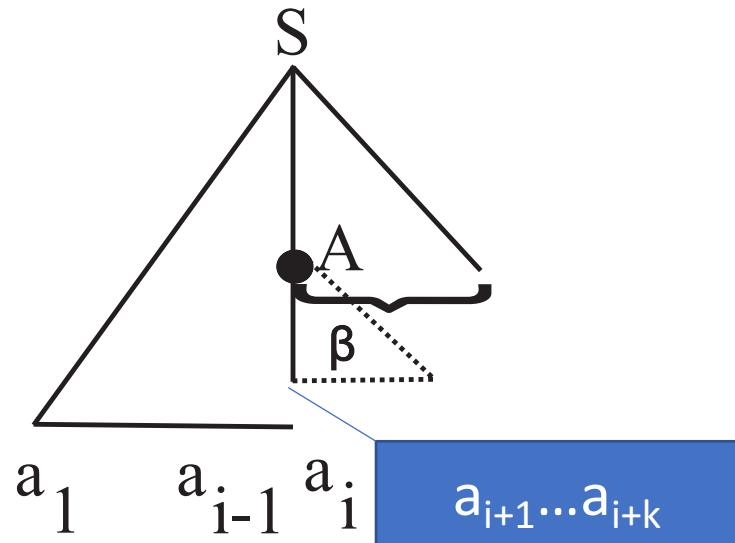
- $\approx$  first terminal symbols that can be generated from  $\alpha$

# FOLLOW

- $\approx$  next symbol generated after/ following A

## LL(k)

- L = left (sequence is read from left to right)
- L = left (use leftmost derivation)
- Prediction of length k



## LL(k) Principle

- In any moment of parsing, action is uniquely determined by:
  - Closed part ( $a_1 \dots a_i$ )
  - Current symbol A
  - Prediction  $a_{i+1} \dots a_{i+k}$  (length k)

# Definition

- A cfg is LL( $k$ ) if for any 2 leftmost derivation we have:

$$1. \ S \xrightarrow[st]{}^* wA\alpha \Rightarrow_{st} w\beta\alpha \xrightarrow[st]{}^* wx;$$

$$2. \ S \xrightarrow[st]{}^* wA\alpha \Rightarrow_{st} w\gamma\alpha \xrightarrow[st]{}^* wy;$$

such that  $\text{FIRST}_k(x) = \text{FIRST}_k(y)$  then  $\beta = \gamma$ .

# Theorem

*The necessary and sufficient condition for a grammar to be LL (k) is that for any pair of distinct productions of a nonterminal ( $A \rightarrow \beta$ ,  $A \rightarrow \gamma, \beta \neq \gamma$ ) the condition holds:*

$$\text{FIRST}_k(\beta\alpha) \cap \text{FIRST}_k(\gamma\alpha) = \emptyset, \forall \alpha \quad \text{such that} \quad S \xrightarrow{*} uA\alpha$$

**Theorem:** A grammar is LL(1) if and only if for any nonterminal A with productions  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ ,  $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$  and if  $\alpha_i \Rightarrow \varepsilon$ ,  $\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A) = \emptyset$ ,  $\forall i, j = 1, n, i \neq j$

# LL(1) Parser

- Prediction of length 1
- Steps:
  - 1) construct FIRST, FOLLOW
  - 2) Construct LL(1) parse table
  - 3) Analyse sequence based on moves between configurations

Executed 1 time

# Step 2: Construct LL(1) parse table

- Possible action depend on:
  - Current symbol  $\in N \cup \Sigma$
  - Possible prediction  $\in \Sigma$
- Add a special character “\$” ( $\notin N \cup \Sigma$ ) – marking for “empty stack”

= > table:

- One line for each symbol  $\in N \cup \Sigma \cup \{\$\}$
- One column for each symbol  $\in \Sigma \cup \{\$\}$

# Rules LL(1) table

1.  $M(A, a) = (\alpha, i), \forall a \in FIRST(\alpha), a \neq \epsilon, A \rightarrow \alpha$  production in P with number i  
 $M(A, b) = (\alpha, i), \text{ if } \epsilon \in FIRST(\alpha), \forall b \in FOLLOW(A), A \rightarrow \alpha$  production in P with number i
2.  $M(a, a) = pop, \forall a \in \Sigma;$
3.  $M($, $) = acc;$
4.  $M(x, a) = \text{err}$  (error) otherwise

## Remark

A grammar is LL(1) if the LL(1) parse table does NOT contain conflicts – there exists at most one value in each cell of the table  $M(A,a)$

# Step 3: Define configurations and moves

- INPUT:
  - Language grammar  $G = (N, \Sigma, P, S)$
  - LL(1) parse table
  - Sequence to be parsed  $w = a_1 \dots a_n$
- OUTPUT:

*If* ( $w \in L(G)$ )      **then string of productions**  
*else* **error & location of error**

# LL(1) configurations

$(\alpha, \beta, \pi)$

where:

- $\alpha$  = input stack
- $\beta$  = working stack
- $\pi$  = output (result)

Initial configuration:  
 $(w\$, S\$, \varepsilon)$

Final configuration:  
 $(\$, \$, \pi)$

# Moves

1. Push – put in stack

$(ux, A\alpha\$, \pi) \vdash (ux, \beta\alpha\$, \pi i), \quad \text{if } M(A, u) = (\beta, i);$   
(pop A and push symbols of  $\beta$ )

2. Pop – take off from stack (from both stacks)

$(ux, a\alpha\$, \pi) \vdash (x, \alpha\$, \pi), \quad \text{if } M(a, u) = \text{pop}$

3. Accept

$(\$, \$, \pi) \vdash acc$

4. Error - otherwise

# Algorithm LL(1) parsing

- INPUT:
  - LL(1) table with NO conflicts;
  - G –grammar (productions)
  - Input sequence  $w = a_1 a_2 \dots a_n$
- OUTPUT:
  - sequence accepted or not?
  - If yes then string of productions

# Algorithm LL(1) parsing (cont)

```
alpha := w$;beta := S$;pi := ε; config =(alpha,beta, pi)  
go := true;
```

```
while go do  
    if M(head(beta),head(alfa))=(b,i) then  
        ActionPush(config)  
    else  
        if M(head(beta),head(alfa))=pop then  
            ActionPop(config)  
        else  
            if M(head(beta),head(alfa))=acc then  
                go:=false; s:="acc";  
            else go:=false; s:="err";  
            end if  
        end if  
    end if  
end while
```

```
if s=="acc" then  
    write("Sequence accepted");  
    write(pi)  
else  
    write(" Sequence not accepted")
```

# Remarks

1) LL(1) parser provides location of the error

2) Grammars can be transformed to be LL(1)

example:

$I \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S \text{ else } S$       // is not LL(1)

$I \rightarrow \text{if } C \text{ then } S T$

$T \rightarrow \epsilon \mid \text{else } S$       // is LL(1)

# Play time!!!

- Menti.com cod: 42 60 49

# Curs 8

## LR( $k$ ) parsing

# Terms

Reminder:

rhp = right handside of production

lhp = left handside of production

- Prediction – see LL(1)
- Handle = symbols from the head of the working stack that form (in order) a rhp
- ***Shift – reduce*** parser:
  - **shift** symbols to form a handle
  - When a rhp is formed – **reduce** to the corresponding lhp

# LR(k)

- L = left – sequence is read from left to right
- R = right – use rightmost derivations
- k = length of prediction
- Enhanced grammar
  - $G = (N, \Sigma, P, S)$
  - $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$ ,  $S' \notin N$

S' does NOT appear in any rhp

# LR(k)

- Ascendent
- Linear – COST? – what we compute to obtain linear algorithm?

- **Definition 1:** If in a cfg  $G = (N, \Sigma, P, S)$  we have

$S \xrightarrow{*} \alpha Aw \Rightarrow_r \alpha\beta w$ , where  $\alpha \in (N \cup \Sigma)^*$ ,  $A \in N$ ,  $w \in \Sigma^*$ , then any prefix of sequence  $\alpha\beta$  is called *live prefix* in  $G$ .

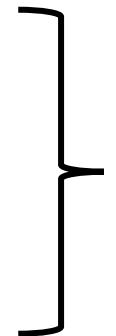
- **Definition 2:** *LR(k) item* is defined as  $[A \rightarrow \alpha.\beta, u]$ , where  $A \rightarrow \alpha\beta$  is a production,  $u \in \Sigma^k$  and describe the moment in which, considering the production  $A \rightarrow \alpha\beta$ ,  $\alpha$  was detected ( $\alpha$  is in head of stack) and it is expected to detect  $\beta$ .

- **Definition 3:** LR( $k$ ) item  $[A \rightarrow \alpha.\beta, u]$  is *valid for the live prefix*  $\gamma\alpha$  if:

$$\begin{aligned} & S \xrightarrow{*} \gamma Aw \Rightarrow_r \gamma\alpha\beta w \\ & u = \text{FIRST}_k(w) \end{aligned}$$

**Definition 4:** A cfg  $G = (N, \Sigma, P, S)$  is LR( $k$ ), for  $k \geq 0$ , if

1.  $S' \xrightarrow{^*} r \alpha Aw \Rightarrow_r \alpha \beta w$
2.  $S' \xrightarrow{^*} r \gamma Bx \Rightarrow_r \alpha \beta y$
3.  $\text{FIRST}_k(w) = \text{FIRST}_k(y)$



$\Rightarrow \alpha = \gamma \text{ AND } A = B \text{ AND } x = y$

- $[A \rightarrow \alpha\beta., u]$  – special case: prefix is all rhp - apply reduce
- Otherwise  $[A \rightarrow \alpha.\beta, u]$  – apply shift

**Consequence 1:** state is important –  
should be stored by parsing method

⇒ Working stack:

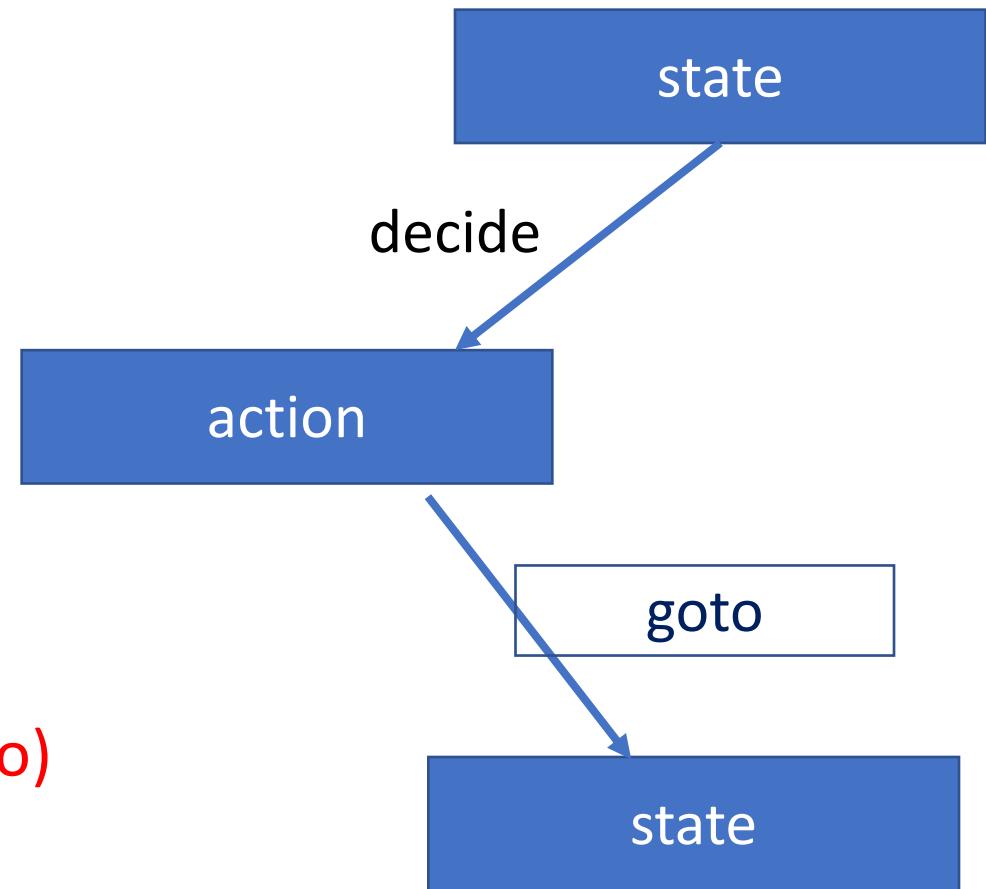
$$\$s_{\text{init}}X_1s_1 \dots X_ms_m$$

where: \$ - mark empty stack

$$X_i \in N \cup \Sigma$$

$s_i$  - states

**Consequence 2:** the action takes the  
parsing process to another state (goto)



# LR(k) principle

- Current state
- Current symbol
- prediction

uniquely determines:

- Action to be applied
- Move to a new state

=> LR(k) table – 2 parts: **action** part + **goto** part

# States

## What a state contains?

- LR items – all items corresponding to same live prefix
- *closure*

## How to go from one state to another state? How many states?

- *goto*
- *Canonical collection*

*What LR item will be in the same state?*

- $[A \rightarrow \alpha.B\beta, u]$  valid for live prefix  $\gamma\alpha \Rightarrow$

$$S \xrightarrow{*_{dr}} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w$$

$$u = FIRST_k(w)$$

- $B \rightarrow \delta \in P \Rightarrow S \xrightarrow{*_{dr}} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w \xrightarrow{*_{dr}} \gamma\alpha\delta w$ ,

- $\Rightarrow [B \rightarrow .\delta, u]$  valid for live prefix  $\gamma\alpha$

LR( $k$ ) parsing:

LR(0), SLR, LR(1), LALR

- Define item
  - Construct set of states
  - Construct table
- 
- Parse sequence based on moves between configurations

Executed 1 time

# LR(0) Parser

- Prediction of length 0 (ignored)
  1. LR(0) item:  $[A \rightarrow \alpha.\beta]$

## 2. Construct set of states

- What a state contains – Algorithm *closure\_LR(0)*
- How to move from a state to another – Function *goto\_LR(0)*
- Construct set of states – Algorithm *ColCan\_LR(0)*

Canonical collection

# Algorithm *Closure\_LR(0)*

---

**INPUT:** I-element de analiză; G'- gramatica îmbogățită

**OUTPUT:** C = closure(I);

$C := \{I\}$ ;

**repeat**

**for**  $\forall [A \rightarrow \alpha.B\beta] \in C$  **do**

**for**  $\forall B \rightarrow \gamma \in P$  **do**

**if**  $[B \rightarrow .\gamma] \notin C$  **then**

$C = C \cup [B \rightarrow .\gamma]$

**end if**

**end for**

**end for**

**until** C nu se mai modifică

---

# Function *goto\_LR(0)*

$\text{goto} : P(\mathcal{E}_0) \times (N \cup \Sigma) \rightarrow P(\mathcal{E}_0)$

where  $\mathcal{E}_0$  = set of LR(0) items

$\text{goto}(s, X) = \text{closure}(\{[A \rightarrow \alpha X . \beta] \mid [A \rightarrow \alpha . X \beta] \in s\})$

# Algorithm *ColCan\_LR(0)*

**INPUT:**  $G'$ - gramatica îmbogățită

**OUTPUT:**  $C$  - colecția canonică de stări

$\mathcal{C} := \emptyset;$

$s_0 := closure(\{[S' \rightarrow .S]\})$

$\mathcal{C} := \mathcal{C} \cup \{s_0\};$

**repeat**

**for**  $\forall s \in \mathcal{C}$  **do**

**for**  $\forall X \in N \cup \Sigma$  **do**

**if**  $goto(s, X) \neq \emptyset$  and  $goto(s, X) \notin \mathcal{C}$  **then**

$\mathcal{C} = \mathcal{C} \cup goto(s, X)$

**end if**

**end for**

**end for**

**until**  $\mathcal{C}$  nu se mai modifică

$S \rightarrow aS \mid bSc \mid dA$

$A \rightarrow dc$

$Goto(s_0, S)$

$Goto(s_0, A)$

$Goto(s_0, a)$

$Goto(s_0, b)$

$Goto(s_0, c) = \emptyset$

$Goto(s_0, d)$

### 3. Construct LR(0) table

- one line for each state
- 2 parts:
  - Action: one column (for a state, action is unique because prediction is ignored)
  - Goto: one column for each symbol  $X \in N \cup \Sigma$

# Rules LR(0) table

1. if  $[A \rightarrow \alpha.\beta] \in s_i$  then **action( $s_i$ )=shift**
2. if  $[A \rightarrow \beta.] \in s_i$  and  $A \neq S'$  then **action( $s_i$ )=reduce l**, where l = number of production  $A \rightarrow \beta$
3. if  $[S' \rightarrow S.] \in s_i$  then **action( $s_i$ )=acc**
4. if  $\text{goto}(s_i, X) = s_j$  then **goto( $s_i, X$ ) =  $s_j$**
5. otherwise = **error**

# Remarks

1) Initial state of parser = state containing  $[S' \rightarrow .S]$

2) No shift from accept state:

*if s is accept state then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .*

3) *If in state s action is reduce then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .*

4) Argument G': Let  $G = (\{S\}, \{a, b, c\}, \{S \rightarrow aSbS, S \rightarrow c\}, S)$

states  $[S \rightarrow aSbS.]$  and  $[S \rightarrow c.]$  – accept / reduce ?

# Remarks (cont)

5) A grammar is NOT LR(0) if the LR(0) table contains conflicts:

- shift – reduce conflict: a state contains items of the form  $[A \rightarrow \alpha.\beta]$  and  $[B \rightarrow \gamma.]$ , yielding to 2 distinct actions for that state
- reduce – reduce conflict: when a state contains items of the form  $[A \rightarrow \alpha\beta.]$  and  $[B \rightarrow \gamma.]$ , in which the action is reduce, but with distinct productions

## 4. Define configurations and moves

- INPUT:
  - Grammar  $G' = (NU\{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$
  - LR(0) table
  - Input sequence  $w = a_1 \dots a_n$
- OUTPUT:

*if* ( $w \in L(G)$ )      **then string of productions**  
*else* **error & location of error**

# LR(0) configurations

$(\alpha, \beta, \pi)$

where:

- $\alpha$  = working stack
- $\beta$  = input stack
- $\pi$  = output (result) stack

Initial configuration:  
 $(\$s_0, w \$, \varepsilon)$

Final configuration:  
 $(\$s_{acc}, \$, \pi)$

# Moves

## 1. Shift

if  $\text{action}(s_m) = \text{shift}$  AND  $\text{head}(\beta) = a_i$  AND  $\text{goto}(s_m, a_i) = s_j$  then

$$(\$s_0 x_1 \dots x_m \underline{s_m}, a_i \dots a_n \$, \pi) \vdash (\$s_0 x_1 \dots x_m \underline{s_m} \underline{a_i} \underline{s_j}, a_{i+1} \dots a_n \$, \pi)$$

## 2. Reduce

if  $\text{action}(s_m) = \text{reduce}$  AND (I)  $A \rightarrow x_{m-p+1} \dots x_m$  AND  $\text{goto}(s_{m-p}, A) = s_j$  then

$$(\$s_0 \dots \underline{x_m} \underline{s_m}, a_i \dots a_n \$, \pi) \vdash (\$s_0 \dots x_{m-p} \underline{s_{m-p}} \underline{A} \underline{s_j}, a_i \dots a_n \$, \pi)$$

## 3. Accept

if  $\text{action}(s_m) = \text{accept}$  then  $(\$s_m, \$, \pi) = \text{acc}$

## 4. Error - otherwise

# LR(0) Parsing Algorithm

## INPUT:

- LR(0) table – conflict free
- grammar  $G'$ : production numbered
  - sequence = Input sequence  $w = a_1 \dots a_n$

## • OUTPUT:

*if* ( $w \in L(G)$ )      **then string of productions**  
*else* **error & location of error**

# LR(0) Parsing Algorithm

```
state := 0;  
alpha := '$s0'; beta := 'w$'; phi := ""; end := false  
Config := (alpha, beta, phi);  
Repeat  
    if action(state) = 'shift' then  
        ActionShift(config)  
    else  
        if action(state) = 'reduce I' then  
            ActionReduce(config)  
        else  
            if action(state) = 'accept' then  
                write(" success"); write(phi);  
                end := true;  
            if action(state) = 'error' then  
                write(" error")  
                end := true  
Until end
```

# Course 9

LR( $k$ ) Parsing (cont.)

LR( $k$ ) parsing:

LR(0), SLR, LR(1), LALR

- Define item
  - Construct set of states
  - Construct table
- 
- Parse sequence based on moves between configurations

Executed 1 time

# Algorithm *ColCan\_LR(0)*

---

**INPUT:**  $G'$ - gramatica îmbogățită

**OUTPUT:**  $C$  - colecția canonică de stări

$\mathcal{C} := \emptyset;$

$s_0 := closure(\{[S' \rightarrow .S]\})$  // state corresponding to prod. of  $S'$  = initial state

$\mathcal{C} := \mathcal{C} \cup \{s_0\}$ ; //initialize collection with  $s_0$

**repeat**

**for**  $\forall s \in \mathcal{C}$  **do**

**for**  $\forall X \in N \cup \Sigma$  **do**

**if**  $goto(s, X) \neq \emptyset$  and  $goto(s, X) \notin \mathcal{C}$  **then**

$\mathcal{C} = \mathcal{C} \cup goto(s, X)$  //add new state

**end if**

**end for**

**end for**

**until**  $\mathcal{C}$  nu se mai modifică

---

# Algorithm *Closure*

I = LR(0) item of the form [A-> $\alpha.\beta$ ]

---

**INPUT:** I-element de analiză; G'- gramatica îmbogățită

**OUTPUT:** C = closure(I);

$C := \{I\}$ ; //initialize Closure with the LR(0) item

**repeat**

**for**  $\forall[A \rightarrow \alpha.B\beta] \in C$  **do** //search productions with dot in front of nonterminal

**for**  $\forall B \rightarrow \gamma \in P$  **do** //search productions of that nonterminal

**if**  $[B \rightarrow .\gamma] \notin C$  **then**

$C = C \cup [B \rightarrow .\gamma]$  //adds item formed from production with dot in  
                //front of right hand side of the production

**end if**

**end for**

**end for**

**until** C nu se mai modifică

---

# Function *goto*

$\text{goto} : P(\mathcal{E}_0) \times (N \cup \Sigma) \rightarrow P(\mathcal{E}_0)$  //creates new states

where  $\mathcal{E}_0$  = set of LR(0) items

$\text{goto}(s, X) = \text{closure}(\{[A \rightarrow \alpha X . \beta] \mid [A \rightarrow \alpha . X \beta] \in s\})$

goto(s,X): in state **s**, search LR(0) item that has dot in front of symbol **X**.  
Move the dot after symbol **X** and call closure for this new item.

# SLR Parser

- SLR = Simple LR

Prediction = next symbols on  
input sequence

- Remark:

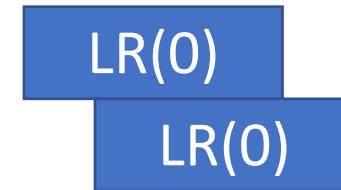
LR(0) – lots of conflicts – solved if considering prediction

=>

1. LR(0) canonical collection of states – prediction of length 0
2. Table and parsing sequence – prediction of length 1

# SLR Parsing:

- define item
- Construct set of states
- Construct table
- Parse sequence based on moves between configurations



# Construct SLR table

Remarks:

1. Prediction = next symbol from input sequence => FOLLOW
  - see LL(1)
2. Structure – LR( $k$ ):

- Lines - states
- action + goto

action – a column for each prediction  $\in \Sigma$

goto – a column for each symbol  $X \in N \cup \Sigma$

Optimize table structure:  
merge *action* and *goto*  
columns for  $\Sigma$

**Remark (LR(0) table):**

- if  $s$  is accept state then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .
- If in state  $s$  action is reduce then  $\text{goto}(s, X) = \emptyset, \forall X \in N \cup \Sigma$ .

# SLR table

And goto

	Action		GOTO			
	$a_1$	$\dots$	$a_n$	$B_1$	$\dots$	$B_m$
$s_0$						
$s_1$						
$\dots$						
$s_k$						

$a_1, \dots, a_n \in \Sigma$   
 $B_1, \dots, B_m \in N$   
 $s_0, \dots, s_k$  - states

# Rules for SLR table

1. If  $[A \rightarrow \alpha.\beta] \in s_i$  and  $\text{goto}(s_i, a) = s_j$  then **action( $s_i, a$ )=shift  $s_j$**   
*// dot is not at the end*
2. if  $[A \rightarrow \beta.] \in s_i$  and  $A \neq S'$  then **action( $s_i, u$ )=reduce l**, where l – number of production  $A \rightarrow \beta$ ,  $\forall u \in \text{FOLLOW}(A)$   
*//dot is at the end, but not for  $S'$*
3. if  $[S' \rightarrow S.] \in s_i$  then **action( $s_i, \$$ )=acc**  
*// dot is at the end, prod. of  $S'$*
4. if  $\text{goto}(s_i, X) = s_j$  then **goto( $s_i, X$ ) =  $s_j$** ,  $\forall X \in N$
5. otherwise **error**

# Remarks

1. Similarity with LR(0)
2. A grammar is SLR if the SLR table does not contain conflicts (more than one value in a cell)

# Parsing sequences

- INPUT:
  - Grammar  $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$
  - SLR table
  - Input sequence  $w = a_1 \dots a_n$
- OUTPUT:

*if* ( $w \in L(G)$ )      **then string of productions**  
*else* **error & location of error**

SLR = LR(0) configurations

$(\alpha, \beta, \pi)$

Initial configuration:  
 $(\$s_0, w \$, \varepsilon)$

where:

- $\alpha$  = working stack
- $\beta$  = input stack
- $\pi$  = output (result)

Final configuration:  
 $(\$s_{acc}, \$, \pi)$

# Moves

head( $\beta$ ) = prediction

## 1. Shift

**if** action( $s_m, a_i$ ) = **shift**  $s_j$  **then**

$$(\$s_0x_1 \dots x_m s_m, a_i \dots a_n \$, \pi) \vdash (\$s_0x_1 \dots x_m s_m a_i s_j, a_{i+1} \dots a_n \$, \pi)$$

## 2. Reduce

**if** action( $s_m, a_i$ ) = **reduce**  $t$  AND ( $t$ )  $A \rightarrow x_{m-p+1} \dots x_m$  AND goto( $s_{m-p}, A$ ) =  $s_j$

**then**

$$(\$s_0 \dots x_m s_m, a_i \dots a_n \$, \pi) \vdash (\$s_0 \dots x_{m-p} s_{m-p} A s_j, a_i \dots a_n \$, t \pi)$$

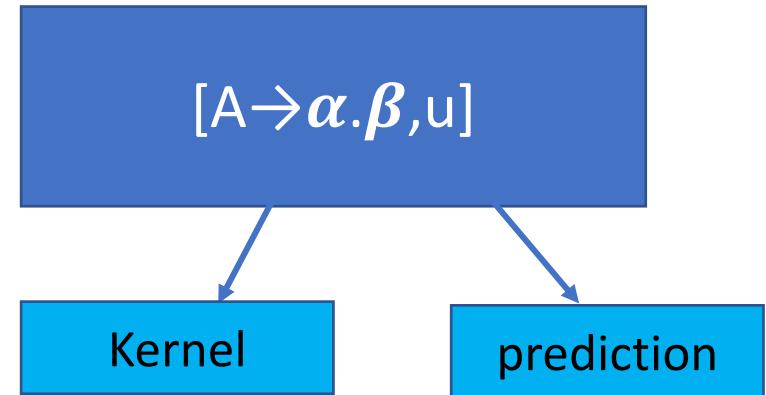
## 3. Accept

**if** action( $s_m, \$$ ) = **accept** **then**  $(\$s_m, \$, \pi) = acc$

## 4. Error - otherwise

# LR(1) Parser

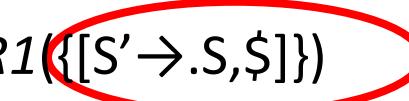
1. Define item
2. Construct set of states
3. Construct table
4. Parse sequence based on moves between configurations



# Construct LR(1) set of states

- Alg *ColCan\_LR1*
- Function *goto\_LR1*
- Alg *Closure\_LR1*

# Algorithm *ColCan\_LR1*

**INPUT:**  $G'$  – enhanced grammar  
**OUTPUT:**  $\mathcal{C}_1$  – canonical collection of states  
 $\mathcal{C}_1 = \emptyset$   
 $S_0 = Closure_{LR1}(\{[S' \rightarrow .S, \$]\})$    
 $\mathcal{C}_1 := \mathcal{C}_1 \cup \{s_0\}$   
**Repeat**  
    **for**  $\forall s \in \mathcal{C}_1$  **do**  
        **for**  $\forall X \in N \cup \Sigma$  **do**  
             $T = goto_{LR1}(s, X)$   
            **if**  $T \neq \emptyset$  **and**  $T \notin \mathcal{C}_1$  **then**  
                 $\mathcal{C}_1 = \mathcal{C}_1 \cup T$   
            **endif**  
        **endfor**  
    **endfor**  
**Until**  $\mathcal{C}_1$  *unchanged*

## Function *goto\_LR1*

$Goto\_LR1 : P(\mathcal{E}_1) \times (N \cup \Sigma) \rightarrow P(\mathcal{E}_1)$

where  $\mathcal{E}_1$  = set of LR(1) items

$Goto\_LR1(s, X) = Closure\_LR1(\{[A \xrightarrow{\cdot} \alpha X . \beta, u] \mid [A \xrightarrow{\cdot} \alpha . X \beta, u] \in s\})$

## Algorithm *Closure\_LR1*

- $[A \rightarrow \alpha.B\beta, u]$  valid for live prefix  $\gamma\alpha \Rightarrow$

$$S \xrightarrow{*_{dr}} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w$$

$$u = FIRST_k(w)$$

- $[B \rightarrow .\delta, smth] \in P \Rightarrow S \xrightarrow{*} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w \Rightarrow_{dr} \gamma\alpha\delta\beta w.$

$\Rightarrow [B \rightarrow .\delta, b]$  valid for live prefix  $\gamma\alpha,$

$$\forall b \in FIRST(\beta u) \quad // First(\beta w) = First(\beta u)$$

# Algorithm *Closure\_LR1*

**INPUT:** I-element de analiză; G' - gramatica îmbogățită;

$FIRST(X), \forall X \in N \cup \Sigma;$

**OUTPUT:**  $C_1 = \text{closure}(I);$

$C_1 := \{I\};$

**repeat**

**for**  $\forall [A \rightarrow \alpha.B\beta, a] \in C_1$  **do**

**for**  $\forall B \rightarrow \gamma \in P$  **do**

**for**  $\forall b \in FIRST(\beta a)$  **do**

**if**  $[B \rightarrow .\gamma, b] \notin C_1$  **then**

$C_1 = C_1 \cup [B \rightarrow .\gamma, b]$

**end if**

**end for**

**end for**

**end for**

**until**  $C_1$  nu se mai modifică

# Construct LR(1) table

- Structure – SLR
- Rules:
  1. if  $[A \rightarrow \alpha.\beta, u] \in s_i$  and  $\text{goto}(s_i, a) = s_j$  then **action( $s_i, a$ )=shift  $s_j$**
  2. if  $[A \rightarrow \beta., u] \in s_i$  and  $A \neq S'$  then **action( $s_i, u$ )=reduce l**, where l – number of production  $A \rightarrow \beta$
  3. if  $[S' \rightarrow S., \$] \in s_i$  then **action( $s_i, \$$ )=acc**
  4. if  $\text{goto}(s_i, X) = s_j$  then **goto( $s_i, X$ ) =  $s_j$** ,  $\forall X \in N$
  5. otherwise = **error**

# Remarks

1. A grammar is LR(1) if the LR(1) table does not contain conflicts
2. Number of states – significantly increase

## 4. Define configurations and moves

- INPUT:
  - Grammar  $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$
  - LR(1) table
  - Input sequence  $w = a_1 \dots a_n$
- OUTPUT:

*if* ( $w \in L(G)$ )      ***then string of productions***  
*else error & location of error*

# LR(1) configurations

$(\alpha, \beta, \pi)$

where:

- $\alpha$  = working stack
- $\beta$  = input stack
- $\pi$  = output (result)

Initial configuration:  
 $(\$s_0, w \$, \varepsilon)$

Final configuration:  
 $(\$s_{acc}, \$, \pi)$

# Moves

head( $\beta$ ) = prediction

## 1. Shift

**if** action( $s_m, a_i$ ) = shift  $s_j$  **then**

$$(\$s_0x_1 \dots x_m s_m, a_i \dots a_n \$, \pi) \vdash (\$s_0x_1 \dots x_m s_m a_i s_j, a_{i+1} \dots a_n \$, \pi)$$

## 2. Reduce

**if** action( $s_m, a_i$ ) = reduce  $t$  AND ( $t$ )  $A \rightarrow x_{m-p+1} \dots x_m$  AND goto( $s_{m-p}, A$ ) =  $s_j$

**then**

$$(\$s_0 \dots x_m s_m, a_i \dots a_n \$, \pi) \vdash (\$s_0 \dots x_{m-p} s_{m-p} A s_j, a_i \dots a_n \$, t \pi)$$

## 3. Accept

**if** action( $s_m, \$$ ) = accept **then**  $(\$s_m, \$, \pi) = acc$

## 4. Error - otherwise

# LALR Parser

- LALR = Look Ahead LR(1)
- why?

# LALR principle

$[A \rightarrow \alpha.\beta, u] \in s_i$

$[A \rightarrow \alpha.\beta., u] \in s_i$  apply reduce (k) then  $\text{goto}(s_i, A) = s_m$   
 $[A \rightarrow \alpha.\beta., v] \in s_j$  apply reduce (k) then  $\text{goto}(s_j, A) = s_n$

$\Rightarrow [A \rightarrow \alpha.\beta, u | v] \in s_{i,j}$

$[A \rightarrow \alpha.\beta, v] \in s_j$

- Merge states with the same kernel, conserving all predictions, if **no conflict** is created

# LALR Parsing

- Same as LR(1)
- Number of LALR states = number of SLR / LR(0) states
- How? - LR(1) states

# LR(k) Parsers

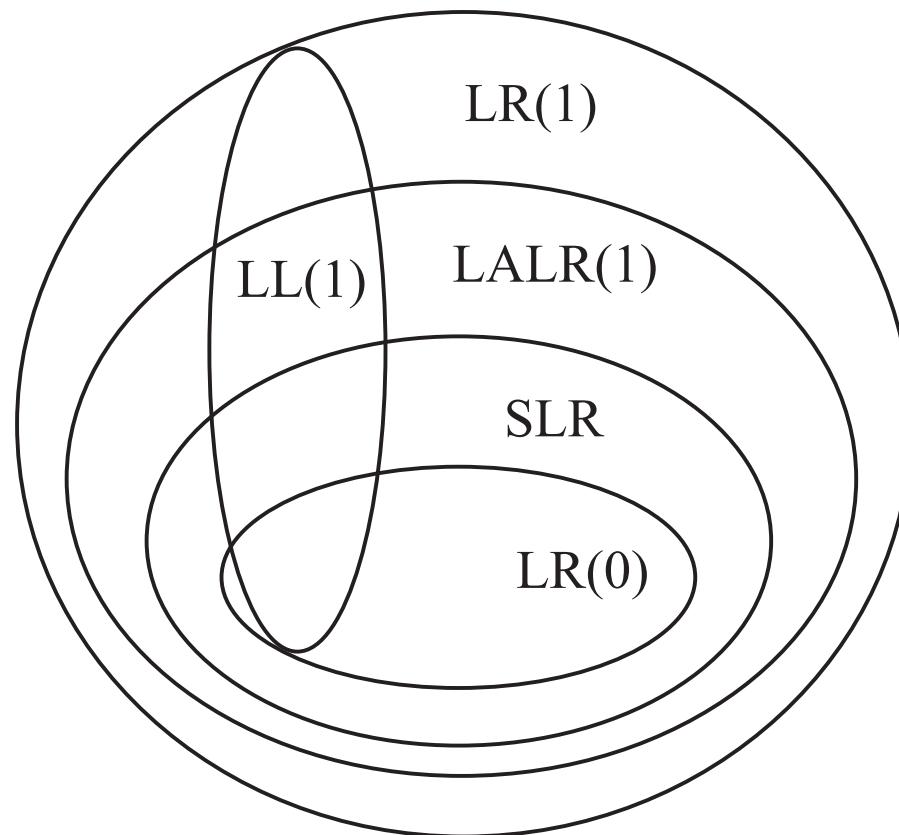
- LR(0):
  - Items ignore prediction
  - Reduce can be applied only in singular states (contain one item)
  - Lot of conflicts
- SLR:
  - Use same items as LR(0)
  - When reduce consider prediction
  - Eliminate several LR(0) conflicts (not all)
- LR(1):
  - Performant algorithm for set of states
  - Generate few conflicts
  - Generate lot of states
- LALR:
  - Merge LR(1) states corresponding to same kernel
  - Most used algorithm (most performant)

# Quiz time

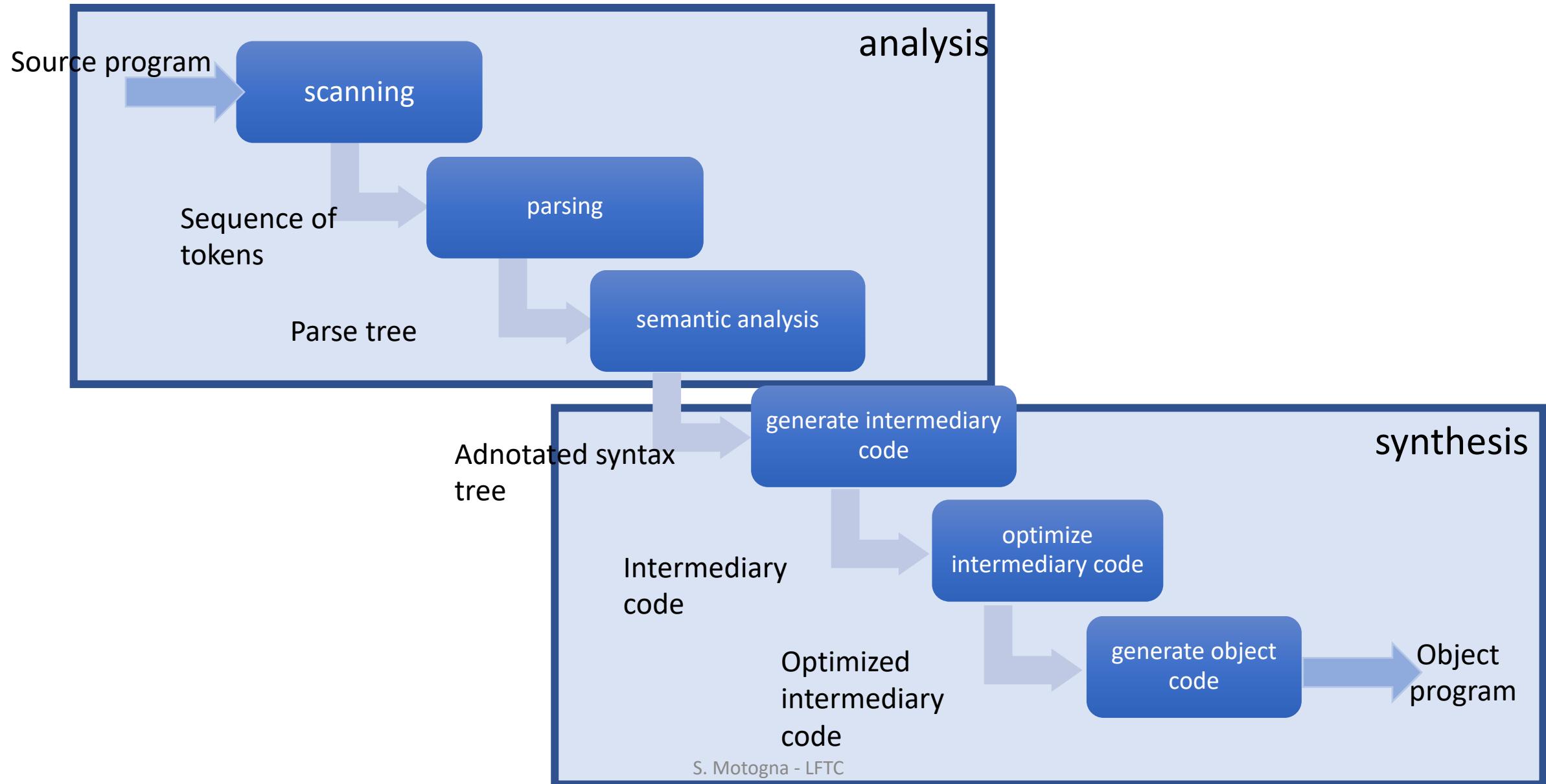
# Parsing - recap

	<b>Descendent</b>	<b>Ascendent</b>
Recursive	Descendent recursive parser	Ascendent recursive parser
Linear	LL(1)	LR(0), SLR, LR(1), LALR

# Parsing - recap



# Structure of compiler



# Course 10

# Important notice

➤ 9.12.2021

7.30 - Course Formal Languages and Compiler Design

9.20 - Course Formal Languages and Compiler Design

➤ 16.12.2021

7.30 – Course Parallel and Distributed Programming

9.20 – Course Parallel and Distributed Programming

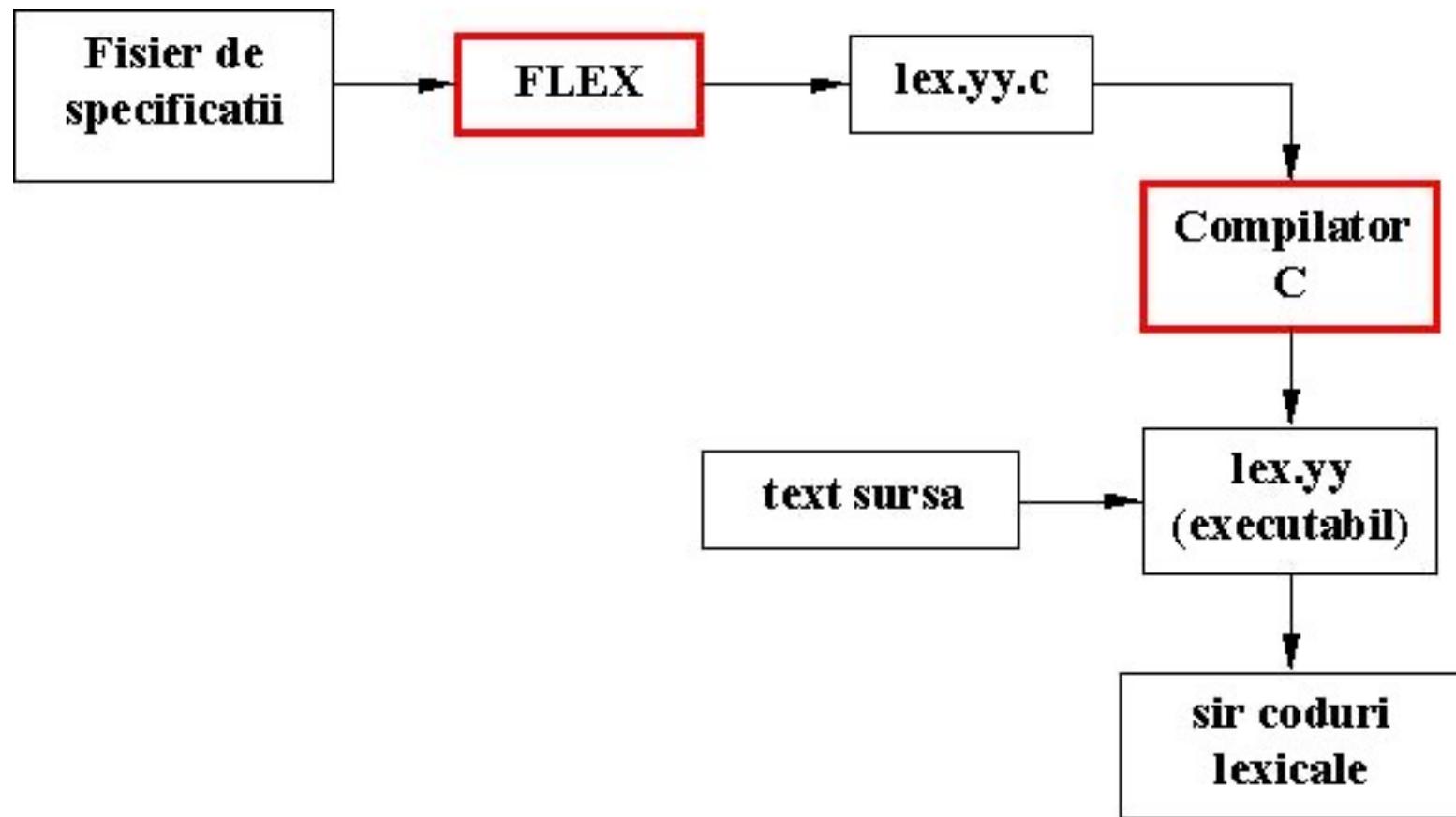
# LEX & YACC

1. Have you heard about these tools?
2. Have you used any of them?

# Scanning & Parsing Tools

- Scanning => lex
- Parsing => yacc

# Lex – Unix utility (flex – Windows version)



## INPUT FILE FORMAT

- The file containing the specification is a text file, that can have any name. Due to historic reasons we recommend the extension **.lxi**.
- Consists of 3 sections separated by a line containing %%:

definitions

%%

rules

%%

user code

## *Example 1:*

%%

```
username printf( "%s", getlogin() );
```

**specifies a scanner that, when finding the string  
“username”, will replace it with the user login name**

## Definition Section:

- C declarations
- +
  - declarations of simple *name definitions* (used to simplify the scanner specification), of the form
    - name definition
  - where:
    - **name** is a word formed by one or more letters, digits, '\_' or '-', with the remark that the first character MUST be letter or '\_' and must be written on the FIRST POSITION OF THE LINE.
    - **definition** is a regular expression and is starting with the first nonblank character after name until the end of line.
    - declarations of *start conditions*.

## Rules Section

- to associate semantic actions with regular expressions. It may also contain user defined C code, in the following way:

**pattern action**

where:

- **pattern** is a regular expression, whose first character MUST BE ON THE FIRST POSITION OF THE LINE;
- **action** is a sequence of one or more C statements that MUST START ON THE SAME LINE WITH THE PATTERN. If there are more than one statements they will be nested between {}. In particular, the action can be a void statement.

## User Defined Code Section:

- Is optional (if is missing, then the separator %% following the rules section can also miss). If it exists, then its containing user defined C code is copied without any change at the end of the file lex.yy.c.
- Normally, in the user defined code section, one may have:
  - function *main()* containing call(s) to *yylex()*, if we want the scanner to work autonomously (for ex., to test it);
  - other called functions from *yylex()* (for ex. *yywrap()*) or functions called during actions); in this case, the user code from definitions section must contain: either prototypes, either **#include** directives of the headers containing the prototypes

Launching the execution:

**lex [option] [name\_specification\_file]**

where *name\_specification\_file* is an input file (implicitly, stdin)

\$ lex spec.lxi

\$ gcc lex.yy.c -o your\_lexer

\$ your\_lexer<input.txt

**options:** <http://dinosaur.compilertools.net/flex/manpage.html>

# Example

yacc

# Parsing (syntax analysis) modeled with cfg:

cfg  $G = (N, \Sigma, P, S)$ :

- $N$  – nonterminal: syntactical constructions: declaration, statement, expression, a.s.o.
- $\Sigma$  – terminals; elements of the language: identifiers, constants, reserved words, operators, separators
- $P$  – syntactical rules – expressed in BNF – simple transformation
- $S$  – syntactical construct corresponding to program

THEN

Program syntactical correct  $\Leftrightarrow w \in L(G)$

# yacc – Unix tool (Bison – Window version)

- **Yet Another Compiler Compiler**
- LALR
- C code

A yacc grammar file has four main sections

```
%{  
C declarations  
%}
```

**yacc declarations**

```
%%
```

**Grammar rules**

```
%%
```

**Additional C code**

contains declarations that define terminal and nonterminal symbols, specify precedence, and so on.

## The grammar rules section

- contains one or more yacc grammar rules of the following general form:

*result:* *components...* { *C statements* }

;

*exp:* *exp '+' exp*  
;

*result:* *rule1-components...*  
| *rule2-components...*

...

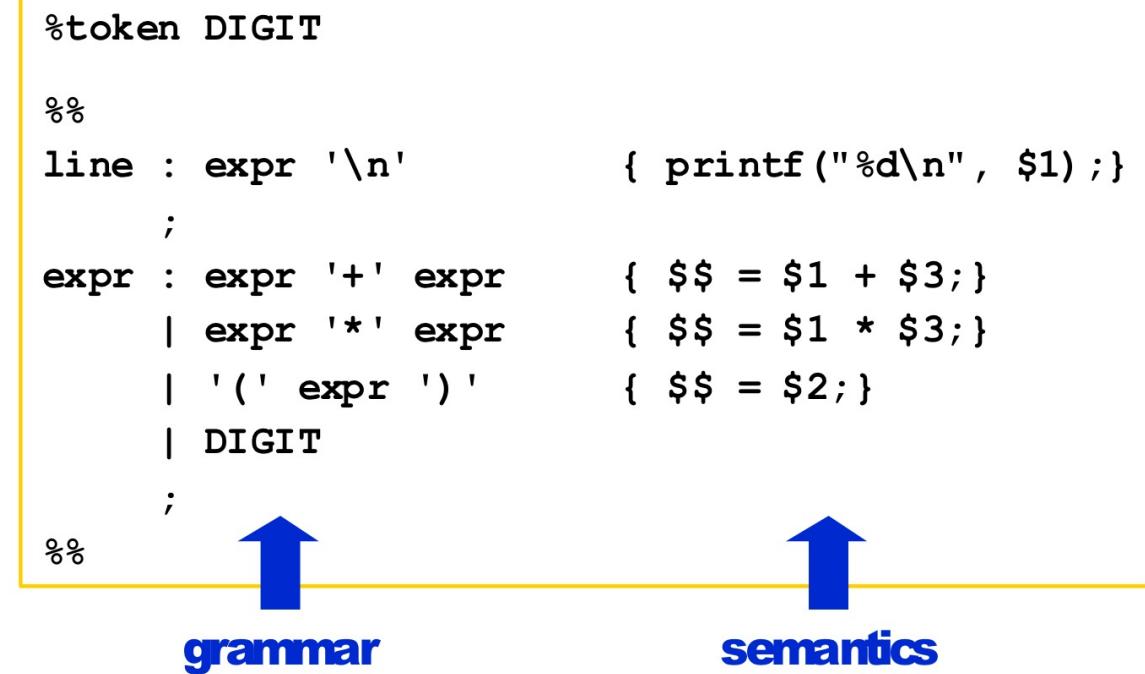
;

*result:* /\*empty \*/  
| *rule2-components...*  
;

# Example: expression interpreter

- input

```
%token DIGIT  
  
%%  
line : expr '\n'          { printf("%d\n", $1); }  
      ;  
expr : expr '+' expr     { $$ = $1 + $3; }  
      | expr '*' expr     { $$ = $1 * $3; }  
      | '(' expr ')'      { $$ = $2; }  
      | DIGIT  
      ;  
%%
```



- Yacc has a stack of values - referenced '\$i' in semantic actions

- Input file (desk0)

```
%%
line : expr '\n'          { printf("%d\n", $1); }
;
expr : expr '+' expr     { $$ = $1 + $3; }
| expr '*' expr         { $$ = $1 * $3; }
| '(' expr ')'          { $$ = $2; }
| DIGIT
;
```

```
> make desk0
bison -v desk0.y
desk0.y contains 4 shift/reduce conflicts.
gcc -o desk0 desk0.tab.c
>
```

# Conflict resolution in yacc

- Conflict **shift-reduce** – prefer **shift**
- Conflict **reduce-reduce** – chose first production

```
%%
line : expr '\n'          { printf("%d\n", $1); }
;
expr : expr '+' expr     { $$ = $1 + $3; }
| expr '*' expr         { $$ = $1 * $3; }
| '(' expr ')'          { $$ = $2; }
| DIGIT
;
%%
```

- Run yacc
- Run desk0

```
> desk0
2*3+4
```

14

# Operator priority in yacc

- From low to great

```
%token DIGIT
%left '+'
%left '*'

%%
line : expr '\n'          { printf("%d\n", $1); }
      ;
expr : expr '+' expr    { $$ = $1 + $3; }
      | expr '*' expr    { $$ = $1 * $3; }
      | '(' expr ')'     { $$ = $2; }
      | DIGIT
      ;
%%
```

- Use

```
>lex spec.lxi  
>yacc -d spec.y  
>gcc lex.yy.c y.tab.c -o result -lfl  
>result<InputProgram
```

- More on

<http://catalog.compilertools.net/lexparse.html>

## Example

# Course 11

## Push-Down Automata (PDA)

# Intuitive Model

# Definition

- A push-down automaton (APD) is a 7-tuple  $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$  where:
  - $Q$  – finite set of states
  - $\Sigma$  - alphabet (finite set of input symbols)
  - $\Gamma$  – stack alphabet (finite set of stack symbols)
  - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$  –transition function
  - $q_0 \in Q$  – initial state
  - $Z_0 \in \Gamma$  – initial stack symbol
  - $F \subseteq Q$  – set of final states

# Push-down automaton

Transition is determined by:

- Current state
- Current input symbol
- Head of stack

Reading head -> input band:

- Read symbol
- No action

Stack:

- Zero symbols => pop
- One symbol => push
- Several symbols => repeated push

# Configurations and transition / moves

- Configuration:

$$(q, x, \alpha) \in Q \times \Sigma^* \times \Gamma^*$$

where:

- PDA is in state  $q$
- Input band contains  $x$
- Head of stack is  $\alpha$
- Initial configuration  $(q_0, w, Z_0)$

# Configurations and moves(cont.)

- Moves between configurations:

$p, q \in Q, a \in \Sigma, Z \in \Gamma, w \in \Sigma^*, \alpha, \gamma \in \Gamma^*$

$(q, aw, Z\alpha) \vdash (p, w, \gamma Z\alpha)$  iff  $\delta(q, a, Z) \ni (p, \gamma Z)$

$(q, aw, Z\alpha) \vdash (p, w, \alpha)$  iff  $\delta(q, a, Z) \ni (p, \varepsilon)$

$(q, aw, Z\alpha) \vdash (p, aw, \gamma Z\alpha)$  iff  $\delta(q, \varepsilon, Z) \ni (p, \gamma Z)$

$(\varepsilon\text{-move})$

- $\vdash^k, \vdash^\dagger, \vdash^*$

# Language accepted by PDA

- Empty stack principle:

$$L_{\varepsilon}(M) = \{w \mid w \in \Sigma^*, (q_0, w, Z_0) \xrightarrow{*} (q, \varepsilon, \varepsilon), q \in Q\}$$

- Final state principle:

$$L_f(M) = \{w \mid w \in \Sigma^*, (q_0, w, Z_0) \xrightarrow{*} (q_f, \varepsilon, \gamma), q_f \in F\}$$

# Representations

- Enumerate
- Table
- Graphic

# Construct PDA

- $L = \{0^n 1^n \mid n \geq 1\}$
- States, stack, moves?

## 1. States:

- Initial state:  $q_0$  – beginning and process symbols ‘0’
- When first symbol ‘1’ is found – move to new state  $\Rightarrow q_1$
- Final: final state  $q_2$

## 2. Stack:

- $Z_0$  – initial symbol
- $X$  – to count symbols:
  - When reading a symbol ‘0’ – push  $X$  in stack
  - When reading a symbol ‘1’ – pop  $X$  from stack

# Exemple 1 (enumerate)

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{Z_0, X\}, \delta, q_0, Z_0, \{q_2\})$$

$$\delta(q_0, 0, Z_0) = (q_0, XZ_0)$$

$$\delta(q_0, 0, X) = (q_0, XX)$$

$$\delta(q_0, 1, X) = (q_1, \epsilon)$$

$$\delta(q_1, 1, X) = (q_1, \epsilon)$$

~~$$\delta(q_1, \epsilon, Z_0) = (q_2, Z_0)$$~~

$$\delta(q_1, \epsilon, Z_0) = (q_1, \epsilon)$$

$$(q_0, 0011, Z_0) \vdash (q_0, 011, XZ_0) \vdash (q_0, 11, XXZ_0) \vdash (q_1, 1, XZ_0) \vdash (q_1, \epsilon, Z_0) \vdash (q_2, \epsilon, Z_0)$$

Empty stack

$\vdash (q_1, \epsilon, \epsilon)$

Final state

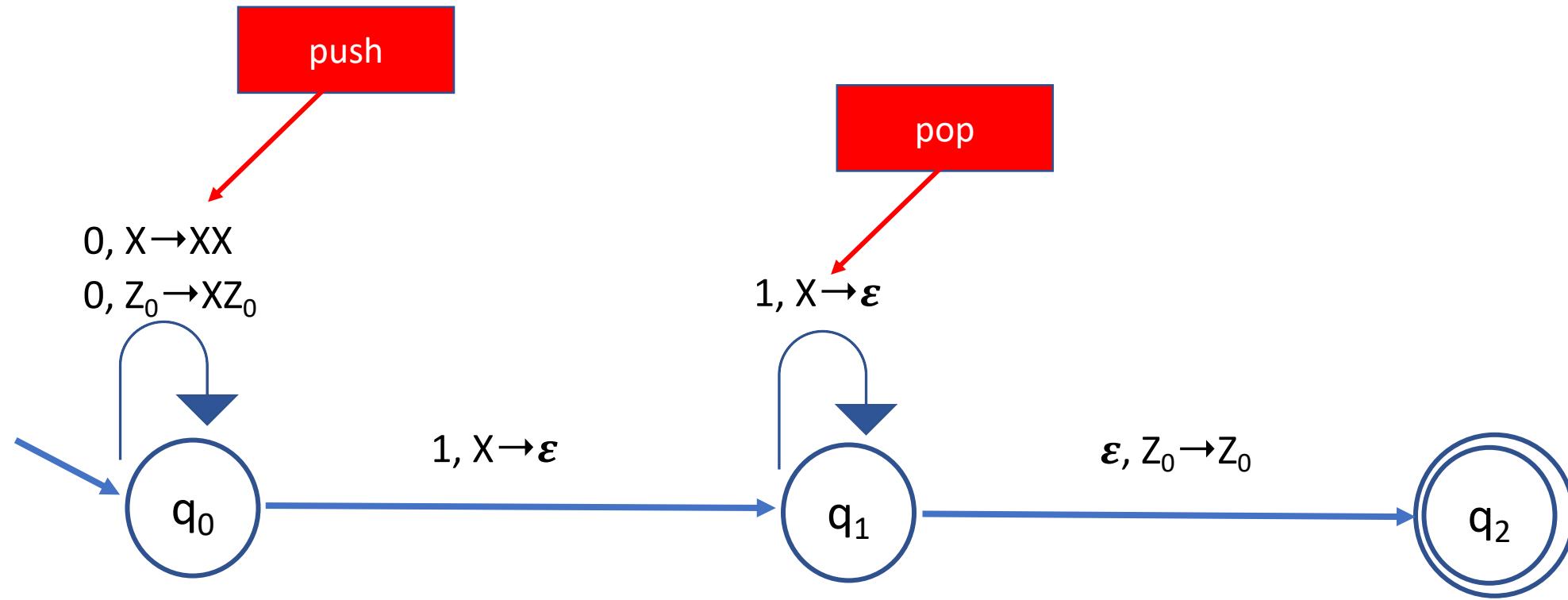
# Exemple 1 (table)

		0	1	$\epsilon$
	Z <sub>0</sub>	q <sub>0</sub> , XZ <sub>0</sub>		
q <sub>0</sub>	X	q <sub>0</sub> , XX	q <sub>1</sub> , $\epsilon$	
q <sub>1</sub>	Z <sub>0</sub>			q <sub>2</sub> , Z <sub>0</sub>
	X		q <sub>1</sub> , $\epsilon$	(q <sub>1</sub> , $\epsilon$ )
q <sub>2</sub>	Z <sub>0</sub>			
	X			

(q0,0011,Z0) |- (q0,011,XZ0) |- (q0,11,XXZ0) |- (q1,1,XZ0)  
 |- (q1,  $\epsilon$ ,Z0) |- (q2,  $\epsilon$ ,Z0) q2 final seq. is acc based on final state

(q0,0011,Z0) |- (q0,011,XZ0) |- (q0,11,XXZ0) |- (q1,1,XZ0)  
 |- (q1,  $\epsilon$ ,Z0) |-(q1,  $\epsilon$ ,  $\epsilon$ ) seq is acc based on empty stack

# Exemple 1 (graphic)



# Properties

**Theorem 1:** For any PDA  $M$ , there exists a PDA  $M'$  such that

$$L_{\varepsilon}(M) = L_f(M')$$

**Theorem 2:** For any PDA  $M$ , there exists a context free grammar such that

$$L_{\varepsilon}(M) = L(G)$$

**Theorem 3:** For any context free grammar there exists a PDA  $M$  such that

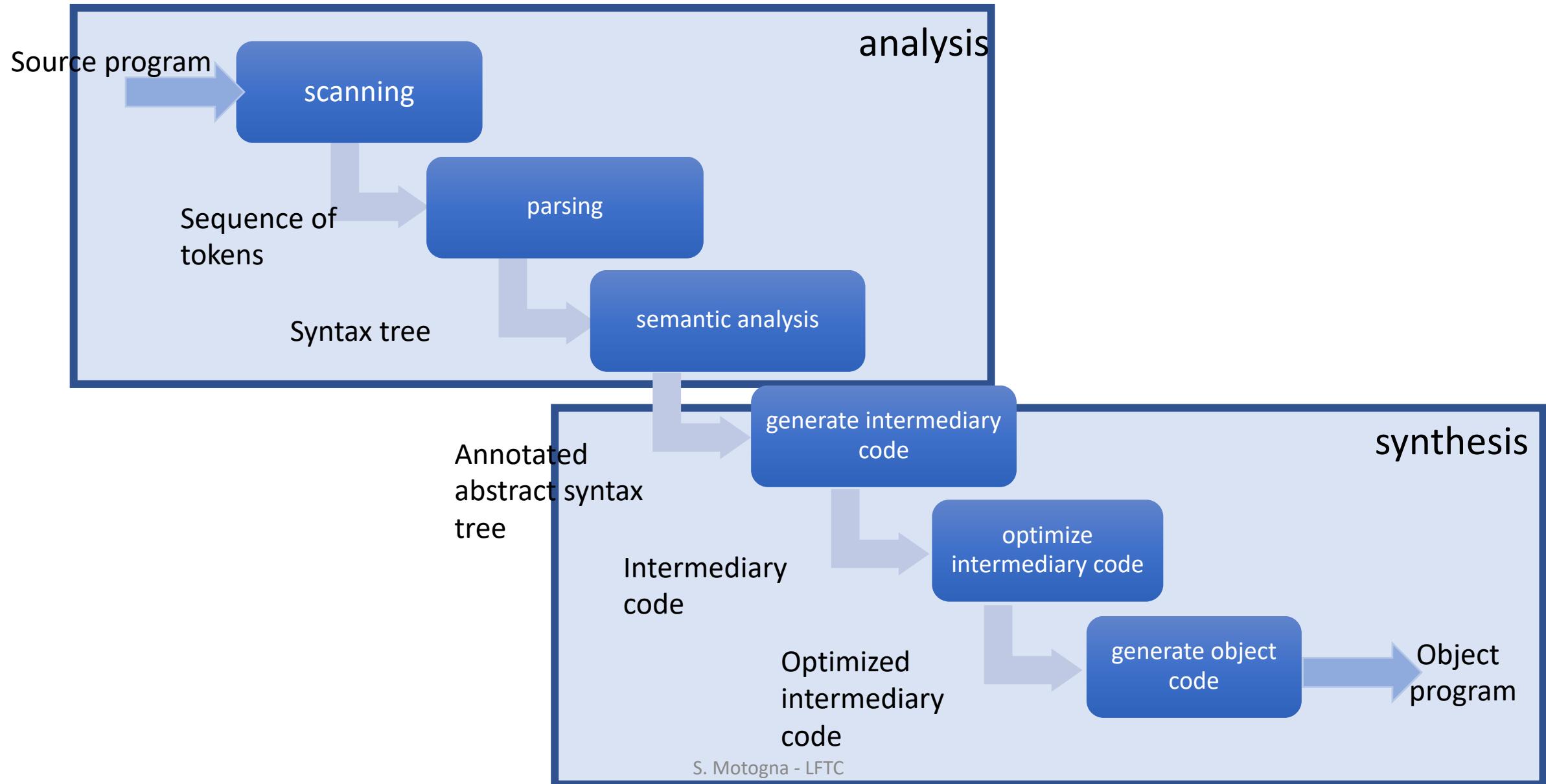
$$L(G) = L_{\varepsilon}(M)$$

# HW

- Parser:
  - Descendent recursive
  - LL(1)
  - LR(0), SLR, LR(1)

Corresponding PDA

# Structure of compiler



# Semantic analysis

- Parsing – result: syntax tree (ST)
- Simplification: abstract syntax tree (AST)
- Annotated abstract syntax tree (AAST)
  - Attach semantic info in tree nodes

Example

# Semantic analysis

- Attach meanings to syntactical constructions of a program
- What:
  - Identifiers -> values / how to be evaluated
  - Statements -> how to be executed
  - Declaration -> determine space to be allocated and location to be stored
- Examples:
  - Type checkings
  - Verify properties
- How:
  - **Attribute grammars**
  - Manual methods

# Attribute grammar

- Syntactical constructions (nonterminals) – attributes

$$\forall X \in N \cup \Sigma: A(X)$$

- Productions – rules to compute/ evaluate attributes

$$\forall p \in P: R(p)$$

# Definition

$AG = (G, A, R)$  is called ***attribute grammar*** where:

- $G = (N, \Sigma, P, S)$  is a context free grammar
- $A = \{A(X) \mid X \in N \cup \Sigma\}$  – is a finite set of attributes
- $R = \{R(p) \mid p \in P\}$  – is a finite set of rules to compute/evaluate attributes

# Example 1

- $G = (\{N, B\}, \{0, 1\}, P, N)$

P:

$$\begin{array}{l} N_1 \rightarrow N_2 \\ \underline{N \rightarrow B} \\ B \rightarrow 0 \\ B \rightarrow 1 \end{array}$$

$$\begin{aligned} N_1.v &= 2 * N_2.v + B.v \\ \underline{N.v = B.v} \\ B.v &= 0 \\ \underline{B.v = 1} \end{aligned}$$

Attribute – value of number = v

- **Synthesized attribute: A(lhp) depends on rhp**
- **Inherited attribute: A(rhp) depends on lhp**

# Evaluate attributes

- Traverse the tree: can be an infinite cycle
- Special classes of AG:
  - L-attribute grammars: for any node the depending attributes are on the “*left*”;
    - can be evaluated in one left-to-right traversal of syntax tree
    - Incorporated in top-down parser (LL(1))
  - S-attribute grammars: synthesized attributes
    - Incorporated in bottom-up parser (LR)

# Steps

- What? - decide what you want to compute (type, value, etc.)
- Decide attributes:
  - How many
  - Which attribute is defined for which symbol
- Attach evaluation rules:
  - For each production – which rule/rules

# Example 2 (L-attribute grammar)

Decl -> DeclTip ListId

ListId -> Id

ListId -> ListId, Id

ListId.type = DeclTip.type  
Id.type = ListId.type  
ListId<sub>2</sub>.type = ListId<sub>1</sub>.type  
Id.type = ListId<sub>1</sub>.type

Attribute – type

int i,j

# Example 3 (S-attribute grammar)

ListDecl -> ListDecl; Decl

ListDecl -> Decl

Decl -> Type ListId

Type -> int

Type -> long

ListId -> Id

ListId -> ListId, Id

$\text{ListDecl}_1.\text{dim} = \text{ListDecl}_2.\text{dim} + \text{Decl.dim}$

$\text{ListDecl.dim} = \text{Decl.dim}$

$\text{Decl.dim} = \text{Type.dim} * \text{ListId.no}$

$\text{Type.dim} = 4$

$\text{Type.dim} = 8$

$\text{ListId.no} = 1$

$\text{ListId}_1.\text{no} = \text{ListId}_2.\text{no} + 1$

Attributes – dim + no – **for which symbols**

int i,j; long k

# Proposed problems (HW):

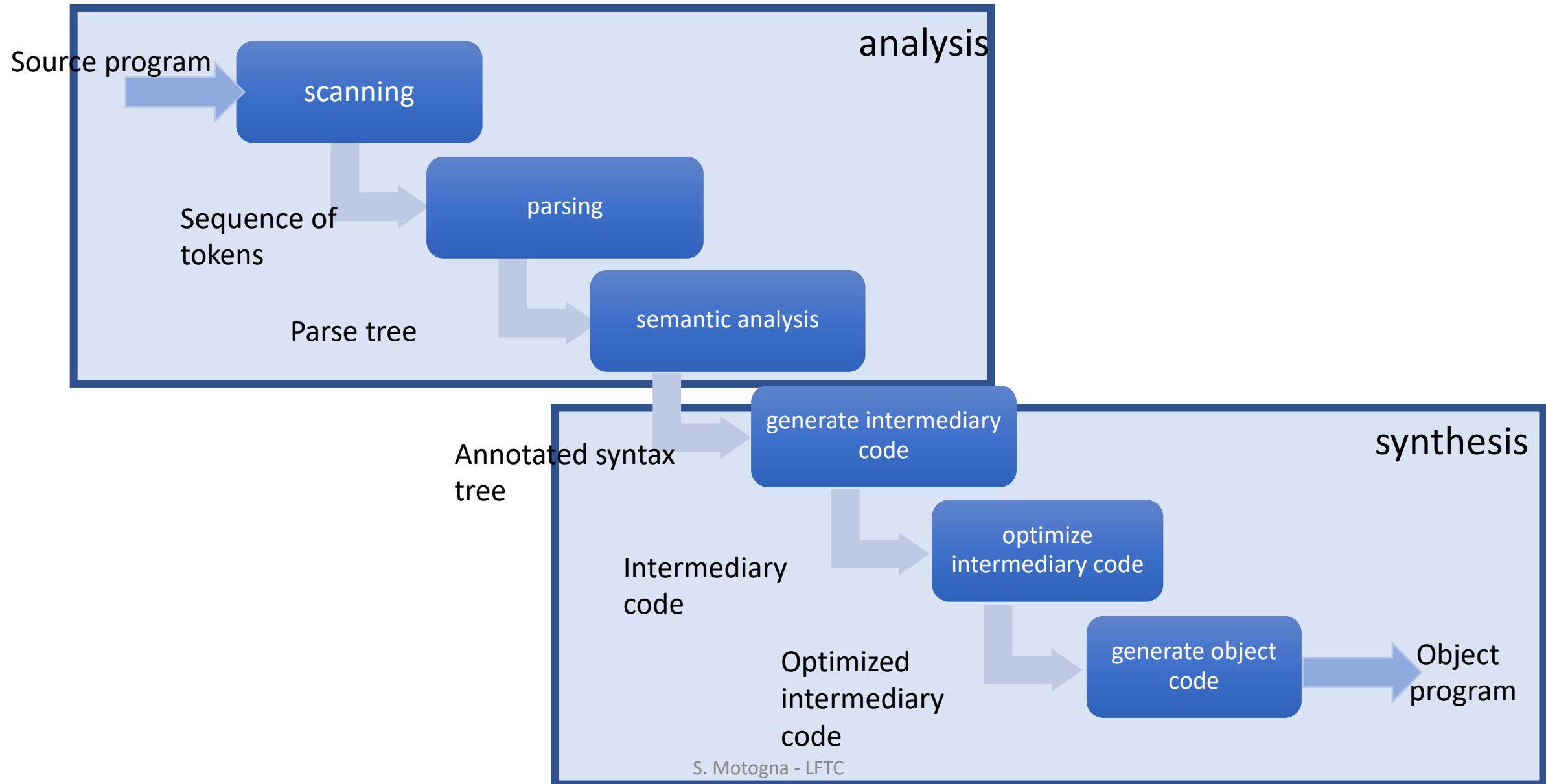
- 1) Define an attribute grammar for arithmetic expressions
- 2) Define an attribute grammar for logical expressions
- 3) Define an attribute grammar for if statement

# Manual methods

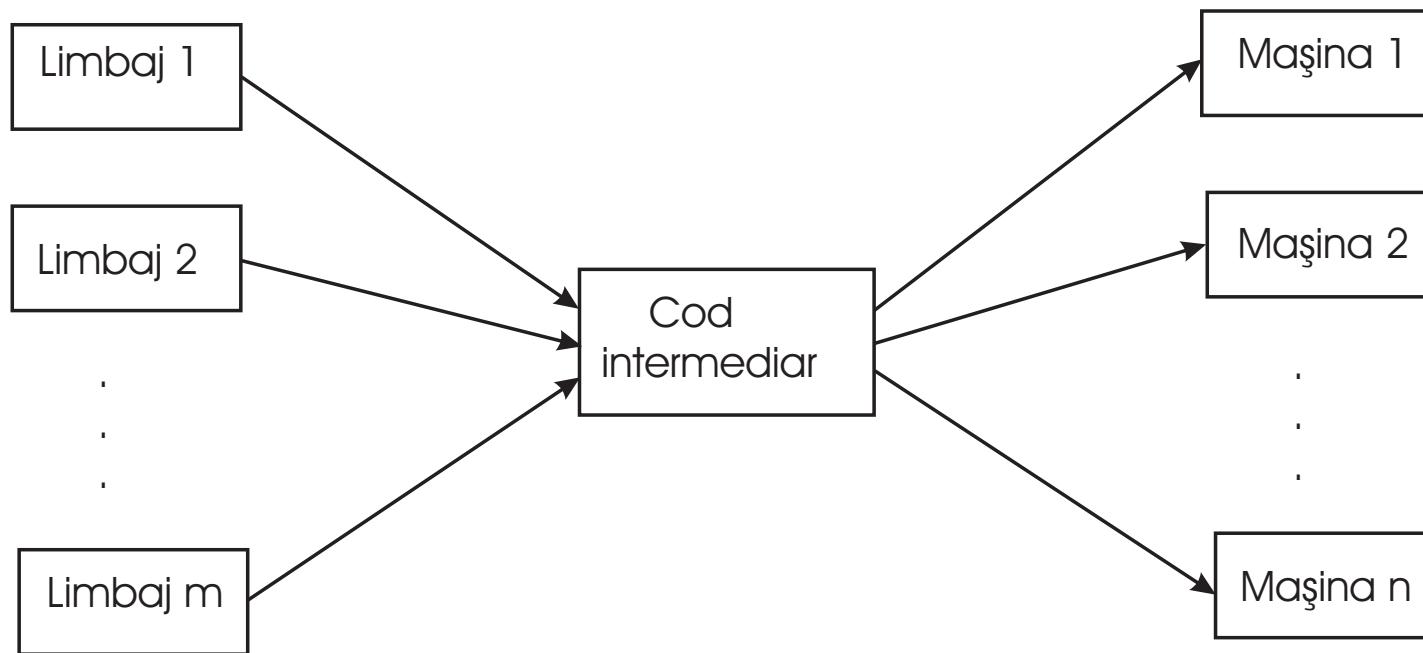
- Symbolic execution
  - Using control flow graph, simulate on stack how the program will behave
  - [Grune – Modern Compiler Design]
- Data flow equations
  - Data flow – associate equations based on data consumed in each node (statement) of the control flow graph: In, Out, Generated, Killed
  - [Grune – Modern Compiler Design], [[Kildall](#)], [[course](#)]

# Course 12

# Structure of compiler



# Generate intermediary code



# Forms of intermediary code

- Java bytecode
  - source language: Java
  - machine language (dif. platforms)
- MSIL (Microsoft Intermediate Language)
  - source language: C#, VB, etc.
  - machine language (dif. platforms)
- GNU RTL (Register Transfer Language)
  - source language: C, C++, Pascal, Fortran etc.
  - machine language (dif. platforms)

# Representations of intermediary code

- Annotated tree: intermediary code is generated in semantic analysis
- Polish postfix form:
  - No parenthesis
  - Operators appear in the order of execution
  - Ex.: MSIL

$$\text{Exp} = a + b * c$$

$$\text{Exp} = a * b + c$$

$$\text{Exp} = a * (b + c)$$

$$\text{ppf} = abc*+$$

$$\text{ppf} = ab*c+$$

$$\text{ppf} = abc+*$$

- 3 address code

# 3 address code

= sequence of simple format statements, close to object code, with the following general form:

**< result >=< arg1 >< op >< arg2 >**

Represented as:

- Quadruples
- Triples
- Indirected Triples

- Quadruples:

< op > < arg1 > < arg2 > < result >

- Triples:

< op > < arg1 > < arg2 >

(considered that the triple is storing the result)

# Special cases:

1. Expressions with unary operator: < result >=< op >< arg2 >
2. Assignment of the form **a := b** => the 3 addresss code is **a = b** (no operatorand no 2<sup>nd</sup> argument)
3. Unconditional jump: statement is **goto L**, where L is the label of a 3 address code
4. Conditional jump: **if c goto L**: if **c** is evaluated to **true** then unconditional jump to statement labeled with L, else (if c is evaluated to false), execute the next statement
5. Function call p(x1, x2, ..., xn) – sequence of statements: **param x1, param x2 , param xn, call p, n**
6. Indexed variables: < arg1 >, < arg2 >, < result > can be array elements of the form **a[i]**
7. Pointer, references: **&x, \*x**

Example:  $b*b-4*a*c$

op	arg1	arg2	rez
*	b	b	t1
*	4	a	t2
*	t2	c	t3
-	t1	t3	t4

nr	op	arg1	arg2
(1)	*	b	b
(2)	*	4	a
(3)	*	(2)	c
(4)	-	(1)	(3)

# Example 2

If ( $a < 2$ ) then  $a = b$  else  $a = b * b$

# Optimize intermediary code

- Local optimizations:
  - Perform computation at compile time – constant values
  - Eliminate redundant computations
  - Eliminate inaccessible code – if...then...else...
- Loop optimizations:
  - Factorization of loop invariants
  - Reduce the power of operations

# Eliminate redundant computations

Example:

$$D := D + C * B$$

$$A := D + C * B$$

$$C := D + C * B$$

(1)	*	C	B
(2)	+	D	(1)
(3)	:=	(2)	D
(4)	*	C	B
(5)	+	D	(4)
(6)	:=	(5)	A
(7)	*	C	B
(8)	+	D	(7)
(9)	:=	(8)	C

# Determine redundant operations

- Operation (j) is redundant to operation (i) with  $i < j$  if the 2 operations are identical and if the operands in (j) did not change in any operation between  $(i+1)$  and  $(j-1)$
- Algorithm [Aho]

# Factorization of loop invariants

What is a loop invariant?

```
for(i=0, i<=n,i++)  
{ x=y+z;  
a[i]=i*x}
```

x=y+z;  
**for**(i=0, i<=n,i++)  
{ a[i]=i\*x}

# Challenge

Consider  $n$ , and  $a[i]$   $i=0,n$  the coefficients of a polynomial  $P$ .

```
V1:  
P = a[0]  
For i=1 to n  
    P = P + a[i]*v^i
```

```
V2:  
P = a[0]  
Q=v  
For i=1 to n  
    P = P + a[i]*Q  
    Q = Q*v
```

Given  $v$ , write an algorithm that computes the value of  $P(v)$

3 solutions

```
V3  
P=a[n]  
For i=1 to n  
    P = P*v + a[n-i]
```

$$P(x) = a[n]*x^n + \dots + a[1]*x + a[0] = (a[n]*x^{n-1} + \dots + a[1])*x + a[0]$$

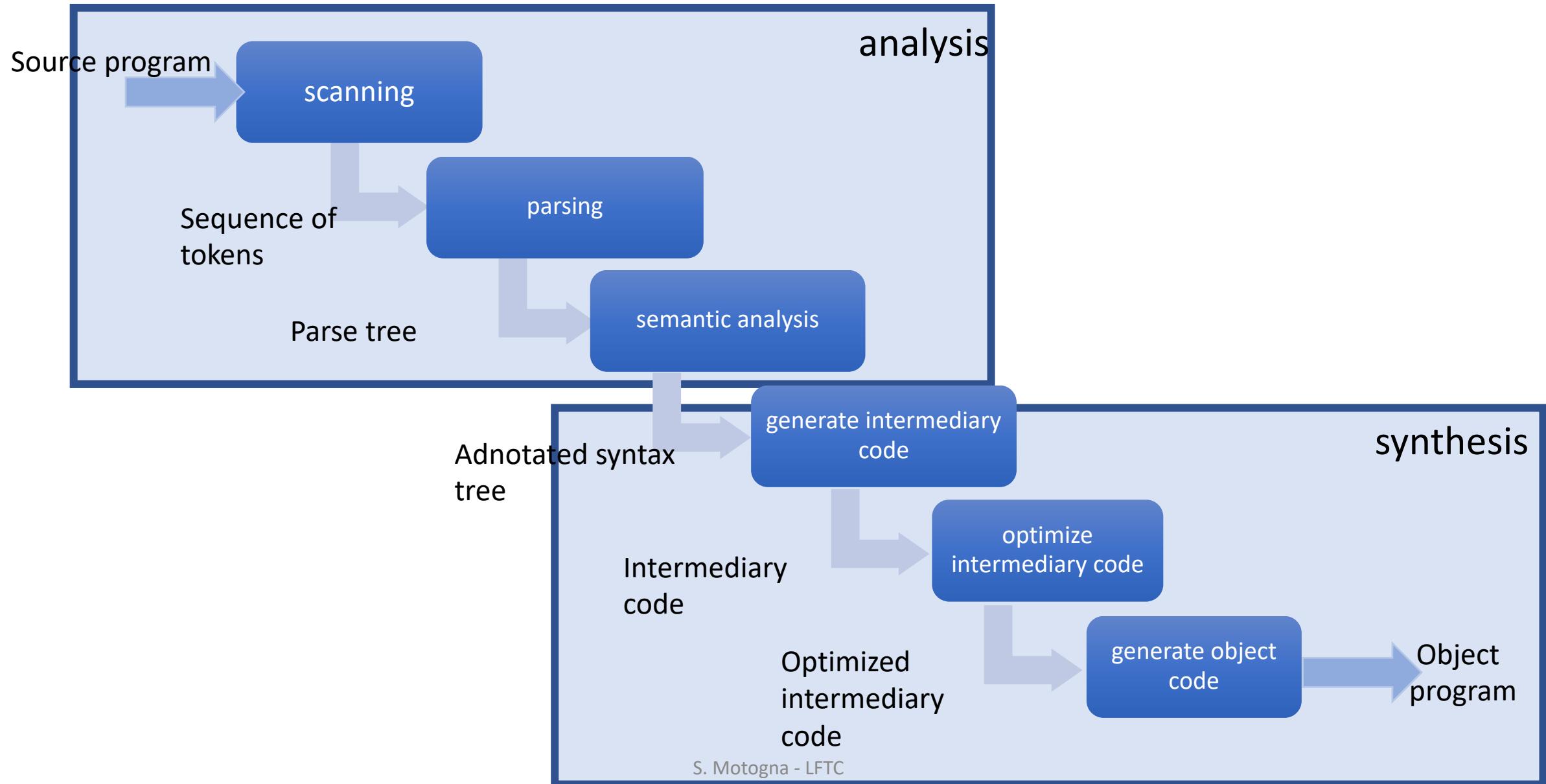
# Reduce the power of operations

```
for(i=k, i<=n,i++)  
{ t=i*v;  
. . .}
```

```
t1=k*v;  
for(i=k, i<=n,i++)  
{ t=t1;  
t1=t1+v;...}
```

# Course 13

# Structure of compiler



# Generate object code

= translate intermediary code statements into statements of object code (machine language)

- Depend on “machine”: architecture and OS

# Computer with accumulator

- A **stack machine** consists of:
- a stack for storing and manipulating values (store subexpressions and results)
- Accumulator – to execute operation
- 2 types of statements:
  - move and copy values in and from head of stack to accumulator
  - Operations on stack head, functioning as follows: operands are popped from stack, execute operation and then put the result in stack

# Example: $4 * (5+1)$

Code	acc	stack
acc $\leftarrow$ 4	4	<>
push acc	4	<4>
acc $\leftarrow$ 5	5	<4>
push acc	5	<5,4>
acc $\leftarrow$ 1	1	<5,4>
acc $\leftarrow$ acc + head	6	<5,4>
pop	6	<4>
acc $\leftarrow$ acc * head	24	<4>
pop	24	<>

# Computer with registers

- Registers +
- Memory
- Instructions:
  - LOAD v,R – load value **v** in register **R**
  - STORE R,v – put value **v** from register **R** in memory
  - ADD R1,R2 – add to the value from register **R1**, value from register **R2** and store the result in **R1** (initial value is lost!)

## 2 aspects:

- Register allocation – way in which variable are stored and manipulated;
- Instruction selection – way and order in which the intermediary code statements are mapped to machine instructions

## Remarks:

1. A register can be available or occupied =>  
 $\text{VAR}(R)$  = set of variables whose values are stored in register R
2. For every variable, the place (register, stack or memory) in which the current value of the value exists=>  
 $\text{MEM}(x)$ = set of locations in which the value of variable x exists (will be stored in Symbol Table)

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		$\text{VAR(R0)} = \{\}$ $\text{VAR(R1)} = \{\}$	
(1) $T1 = A * B$			
(2) $T2 = C + B$			
(3) $T3 = T2 * T1$			
(4) $F := T1 - T3$			

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) $T1 = A * B$	LOAD A, R0 MUL R0, B	VAR(R0) = {A} VAR(R0) = {T1}	MEM(T1) = {R0}
(2) $T2 = C + B$			
(3) $T3 = T2 * T1$			
(4) $F := T1 - T3$			

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) $T1 = A * B$	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) $T2 = C + B$	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) $T3 = T2 * T1$			
(4) $F := T1 - T3$			

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) $T1 = A * B$	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) $T2 = C + B$	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) $T3 = T2 * T1$	MUL R1,R0	VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) $F := T1 - T3$			

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) $T1 = A * B$	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) $T2 = C + B$	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) $T3 = T2 * T1$	MUL R1,R0	VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) $F := T1 - T3$	SUB R0,R1 STORE R0, F	VAR(R0) = {F} VAR(R1) = {}	MEM(T1) = {} MEM(F) = {R0, F}

# More about Register Allocation

- Registers – **limited resource**
- Registers – perform operations / computations
- Variables **much more** than registers

IDEA: *assigning a large number of variables to a reduced number of registers*

# Live variables

- Determine the number of variables that are live (used)

Example:

$$a = b + c$$

$$d = a + e$$

$$e = a + c$$

	op	op1	op2	rez
1	+	b	c	a
2	+	a	e	d
3	+	a	c	e

	1	2	3
a	x	x	x
b	x		
c	x	x	x
d		x	
e		x	x

# Graph coloring allocation (Chaitin a.o. 1982)

- Graph:
  - nodes = live variables that should be allocated to registers
  - edges = live ranges simultaneously live

Register allocation = graph coloring: colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color

## **Disadvantage:**

- NP complete problem

# Linear scan allocation (Poletto a.o., 1999)

- determine all live range, represented as an interval
- intervals are traversed chronologically
- greedy algorithm

**Advantage:** speed – code is generated faster (speed in code generation)

**Disadvantage:** generated code is slower (NO speed in code execution)

# Instruction selection

Example:  $F := A * B - (C + B) * (A * B)$

Intermediary code	Object code	VAR	MEM
		VAR(R0) = {} VAR(R1) = {}	
(1) $T1 = A * B$	LOAD A, R0 MUL R0, B	VAR(R0) = {T1}	MEM(T1) = {R0}
(2) $T2 = C + B$	LOAD C, R1 ADD R1, B	VAR(R1) = {T2}	MEM(T2) = {R1}
(3) $T3 = T2 * T1$	MUL R1,R0	STORE R0,T1 VAR(R1) = {T3}	MEM(T2) = {} MEM(T3) = {R1}
(4) $F := T1 - T3$	LOAD T1,R1		

Decide which register to use for an instruction

# Turing Machines

# Alan Turing

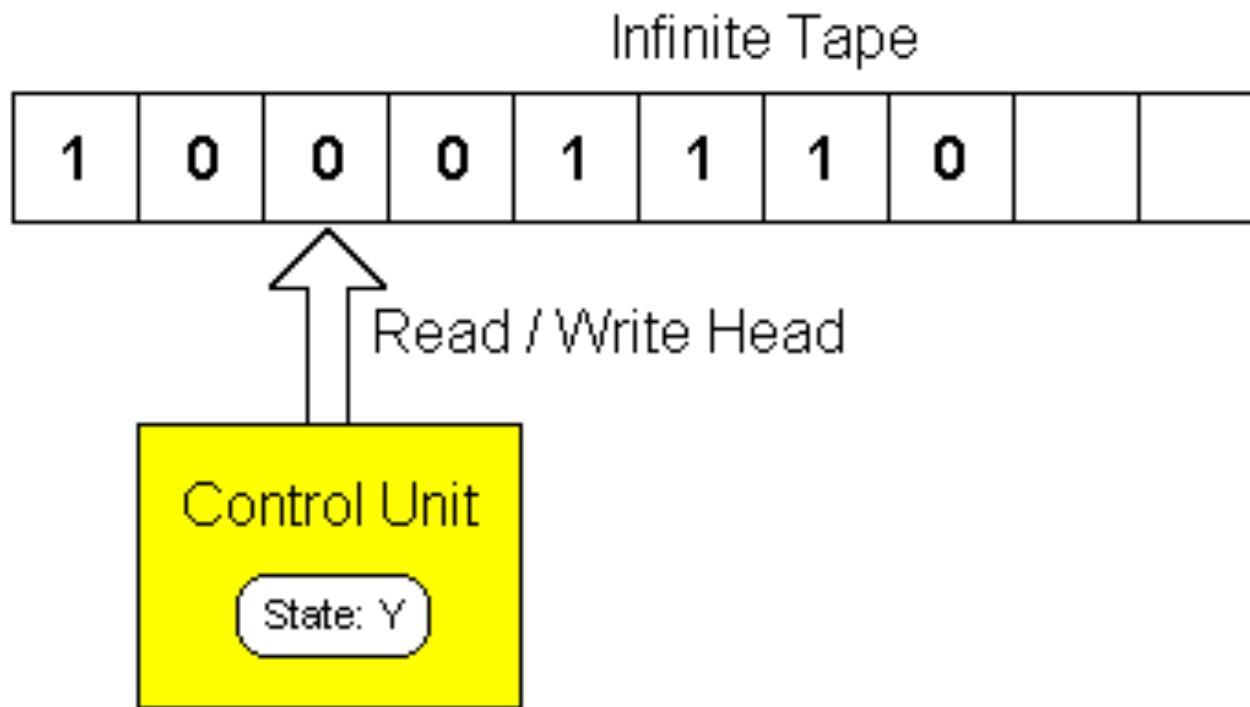
- Enigma (criptography)
- Turing test
- Turing machine (1937)



# Turing Machine

- Mathematical model for computation
- Abstract machine
- Can simulate any algorithm

# Turing Machine



- Input band (infinite)
- Reading head
- Control Unit: states
- Transitions / moves

# Turing machine – definition

7-tuple  $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$  where:

- $Q$  – finite set of states
- $\Gamma$  - alphabet (finite set of band symbols)
- $b \in \Gamma$  - blank (symbol)
- $\Sigma \subseteq \Gamma \setminus \{b\}$  – input alphabet
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  – transition function
- $q_0 \in Q$  – initial state
- $F \subseteq Q$  – set of final states

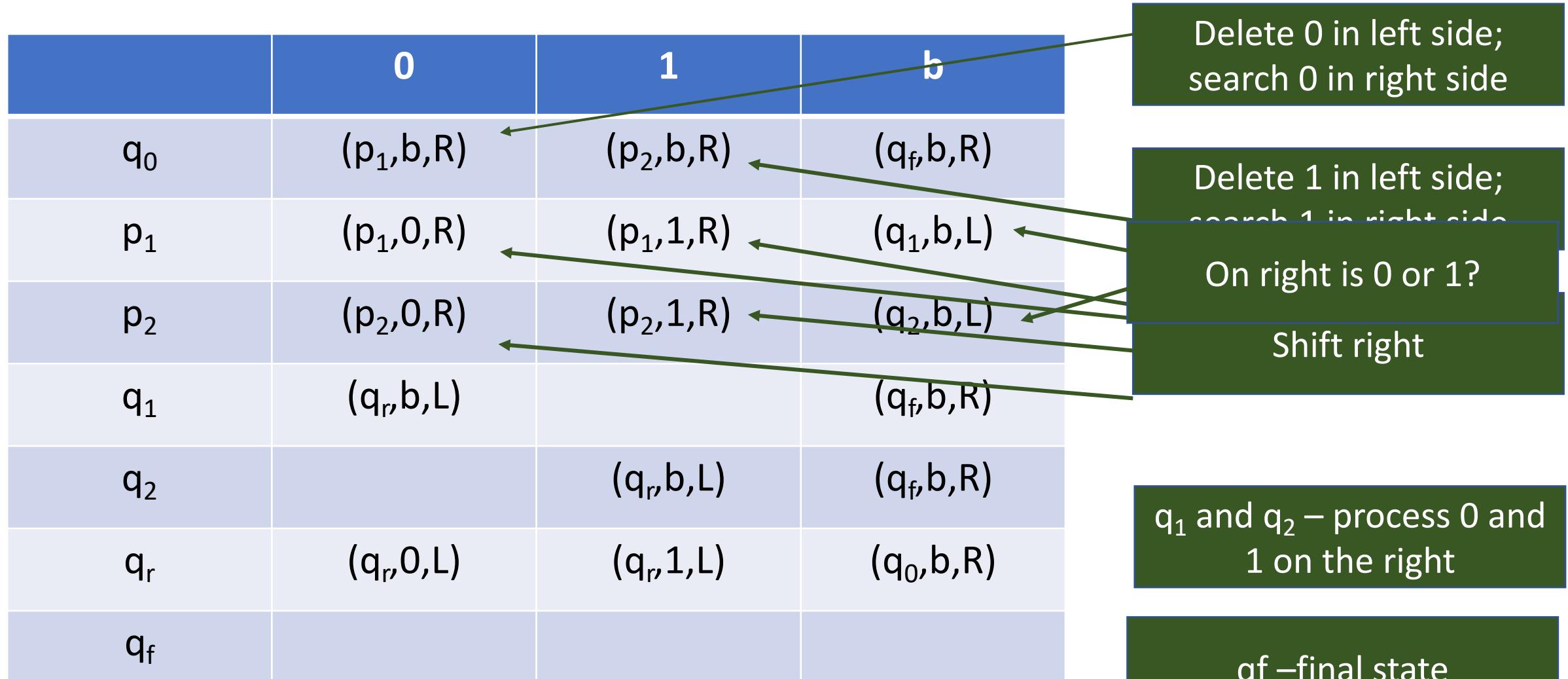
L = left  
R = right

# Example – palindrome over {0,1}

- 001100, 00100, 101101 a.s.o. accepted
- 00110, 1011 a.s.o. not accepted

001100

# Example – palindrome over {0,1}



0110

0	1	1	0	
	1	1	0	
	1	1	0	
	1	1	0	
	1	1	0	
	1	1	0	
	1	1	0	
	1	1	0	

	1	1		
	1	1		
	1	1		
	1	1		
	1	1		
		1		
•	•	•		

$$(q_0, \underline{0}110) \mid\!- (p_1, \underline{1}10) \mid\!- (p_1, 1\underline{1}0)$$

$$\mid\!- (p_1, 11\underline{0}) \mid\!- (p_1, 110\underline{b}) \mid\!- (q_1, 11\underline{0})$$

$$\mid\!- (q_r, 1\underline{1}) \mid\!- (q_r, \underline{1}1) \mid\!- (q_r, \underline{b}11)$$

$$\mid\!- (q_0, \underline{1}1) \mid\!- \dots$$

	<b>0</b>	<b>1</b>	<b>b</b>
$q_0$	$(p_1, b, R)$	$(p_2, b, R)$	$(q_f, b, R)$
$p_1$	$(p_1, 0, R)$	$(p_1, 1, R)$	$(q_1, b, L)$
$p_2$	$(p_2, 0, R)$	$(p_2, 1, R)$	$(q_2, b, L)$
$q_1$	$(q_r, b, L)$		$(q_f, b, R)$
$q_2$		$(q_r, b, L)$	$(q_f, b, R)$
$q_r$	$(q_r, 0, L)$	$(q_r, 1, L)$	$(q_0, b, R)$
$q_f$			

<https://turingmachinesimulator.com>