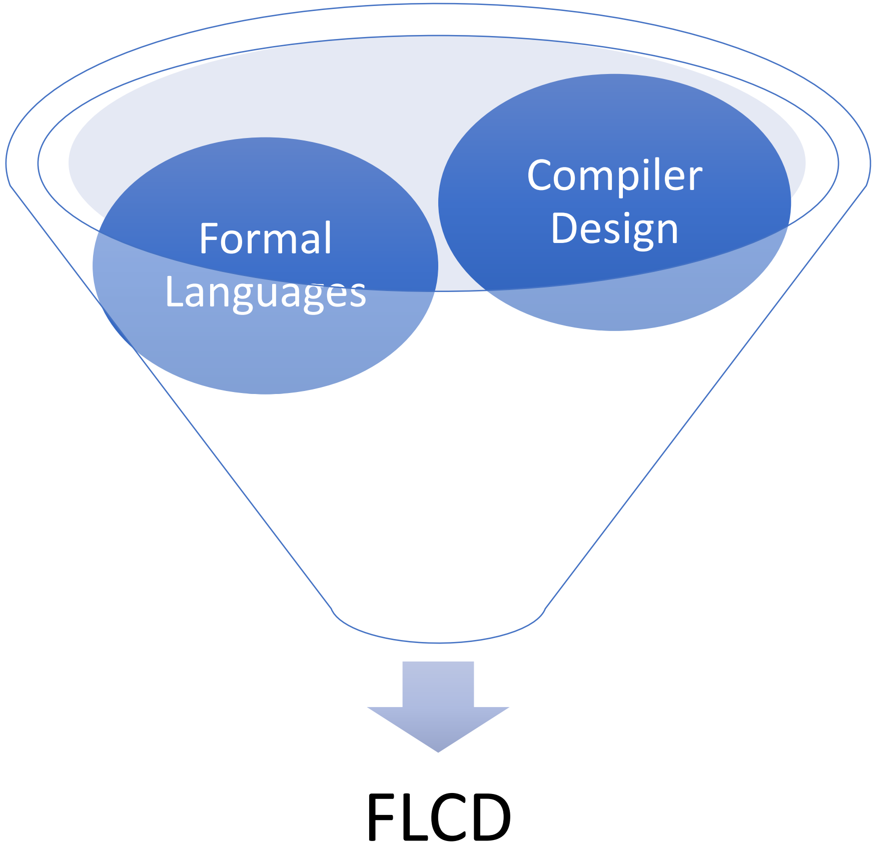
Formal Languages and Compiler Design

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

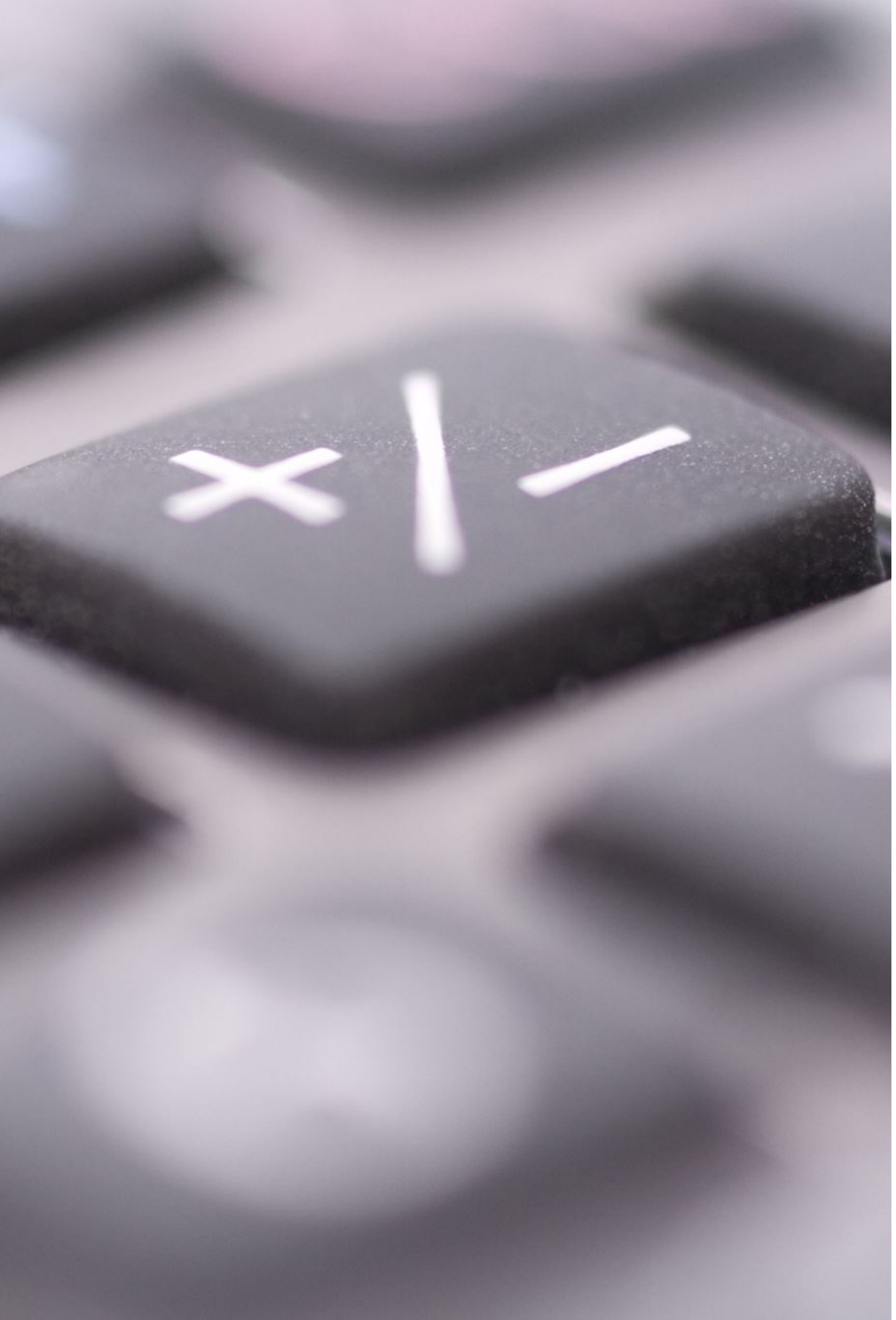
- Moodle:
  - All course resources
  - Homeworks
  - Assignments
  - Labs
  - Points / grades

- 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 fișa disciplinei

```python
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
```

ZBNKPRT1.CBL

```cobol
       IDENTIFICATION DIVISION.
       PROGRAM-ID.
           ZBNKPRT1.
       DATE-WRITTEN.
           September 2002.
       DATE-COMPILED.
           Today.

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT   SECTION.
         FILE-CONTROL.
           SELECT EXTRACT-FILE
                  ASSIGN       TO EXTRACT
                  ORGANIZATION IS SEQUENTIAL
                  ACCESS MODE  IS SEQUENTIAL
                  FILE STATUS  IS WS-EXTRACT-STATUS.
           SELECT PRINTOUT-FILE
                  ASSIGN       TO PRINTOUT
                  ORGANIZATION IS SEQUENTIAL
                  ACCESS MODE  IS SEQUENTIAL
                  FILE STATUS  IS WS-PRINTOUT-STATUS.

       DATA DIVISION.
       FILE SECTION.

       FD  EXTRACT-FILE
           RECORDING MODE IS V
```

```c
#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 = %lf\n", mean(data, sizeof(data) / sizeof(data[0])));
}
```

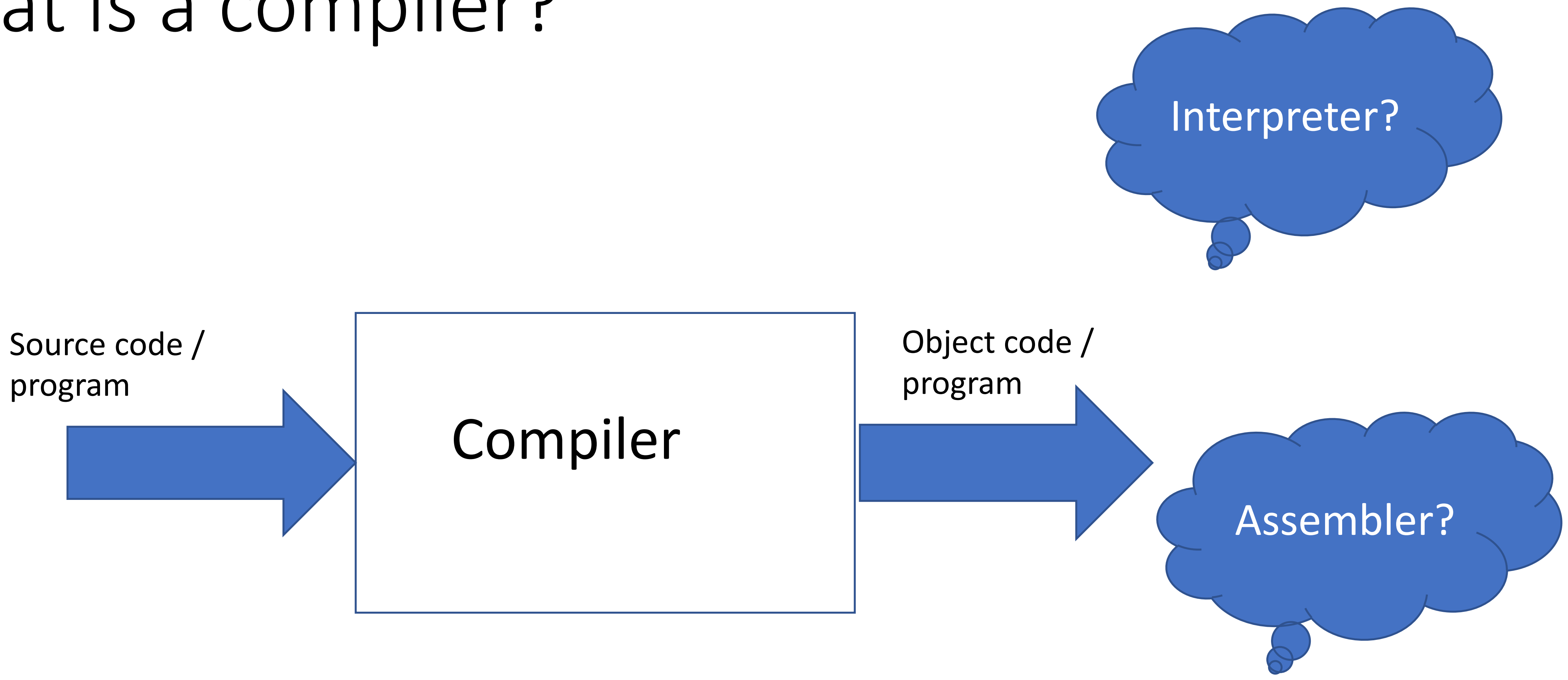
```java
package rentalStore;
import java.util.Enumeration;
import java.util.Vector;

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;
    }
```

```fortran
C
      PIN=0.02
      IF(DDT.NE.0.0)  THEN
      DT=DDT
      ELSE
      DT=PIN
      ENDIF
      WRITE(*,'(A)') '    PLEASE ENTER NAME OF OUTPUT FILE (FOR EXAMPLE
     *  B:ZZ.DAT)'
      READ(*,'(A)') FNAMEO
      OPEN(6,FILE=FNAMEO,STATUS='UNKNOWN')
      PV=WFLX/TH
      RS=NEQ*ROU*KD/TH
      C0=CS
C
      TIME=0.0D0
      EF=0.0D0
    5 CONTINUE
      GAMMA=DT/(2.D0*DX*DX)
      BETA=DT/DX
      IF((BETA*PV).GT.0.50D0) GO TO 7
      IF((GAMMA*D/(BETA*PV)).LT.0.5D0) GO TO 6
      GO TO 8
    6 DX=DX/2
      GO TO 5
    7 DT=DT/2
      GO TO 5
    8 CONTINUE
      N=COL/DX
      NM1=N-1
      NM2=N-2
      NP1=N+1
      GAMMA=DT/(2*DX*DX)
```

What is a compiler?



```python
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
```

ZBNKPRT1.CBL

```cobol
IDENTIFICATION DIVISION.
PROGRAM-ID.
    ZBNKPRT1.
DATE-WRITTEN.
    September 2002.
DATE-COMPILED.
    Today.

ENVIRONMENT DIVISION.
INPUT-OUTPUT   SECTION.
  FILE-CONTROL.
    SELECT EXTRACT-FILE
           ASSIGN       TO EXTRACT
           ORGANIZATION IS SEQUENTIAL
           ACCESS MODE  IS SEQUENTIAL
           FILE STATUS  IS WS-EXTRACT-STATUS.
    SELECT PRINTOUT-FILE
           ASSIGN       TO PRINTOUT
           ORGANIZATION IS SEQUENTIAL
           ACCESS MODE  IS SEQUENTIAL
           FILE STATUS  IS WS-PRINTOUT-STATUS.

DATA DIVISION.
FILE SECTION.

FD  EXTRACT-FILE
    RECORDING MODE IS V
```

```c
#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 = %lf\n", mean(data, sizeof(data) / sizeof(data[0])));
}
```

```java
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;
    }
```

```fortran
C
      PIN=0.02
      IF(DDT.NE.0.0)  THEN
      DT=DDT
      ELSE
      DT=PIN
      ENDIF
      WRITE(*,'(A)') '    PLEASE ENTER NAME OF OUTPUT FILE (FOR EXAMPLE
     *  B:ZZ.DAT)'
      READ(*,'(A)') FNAMEO
      OPEN(6,FILE=FNAMEO,STATUS='UNKNOWN')
      PV=WFLX/TH
      RS=NEQ*ROU*KD/TH
      C0=CS
C
      TIME=0.0D0
      EF=0.0D0
    5 CONTINUE
      GAMMA=DT/(2.D0*DX*DX)
      BETA=DT/DX
      IF((BETA*PV).GT.0.50D0) GO TO 7
      IF((GAMMA*D/(BETA*PV)).LT.0.5D0) GO TO 6
      GO TO 8
    6 DX=DX/2
      GO TO 5
    7 DT=DT/2
      GO TO 5
    8 CONTINUE
      N=COL/DX
      NM1=N-1
      NM2=N-2
      NP1=N+1
      GAMMA=DT/(2*DX*DX)
```

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

A little bit of history …

Fortran
1954-1957
Backus

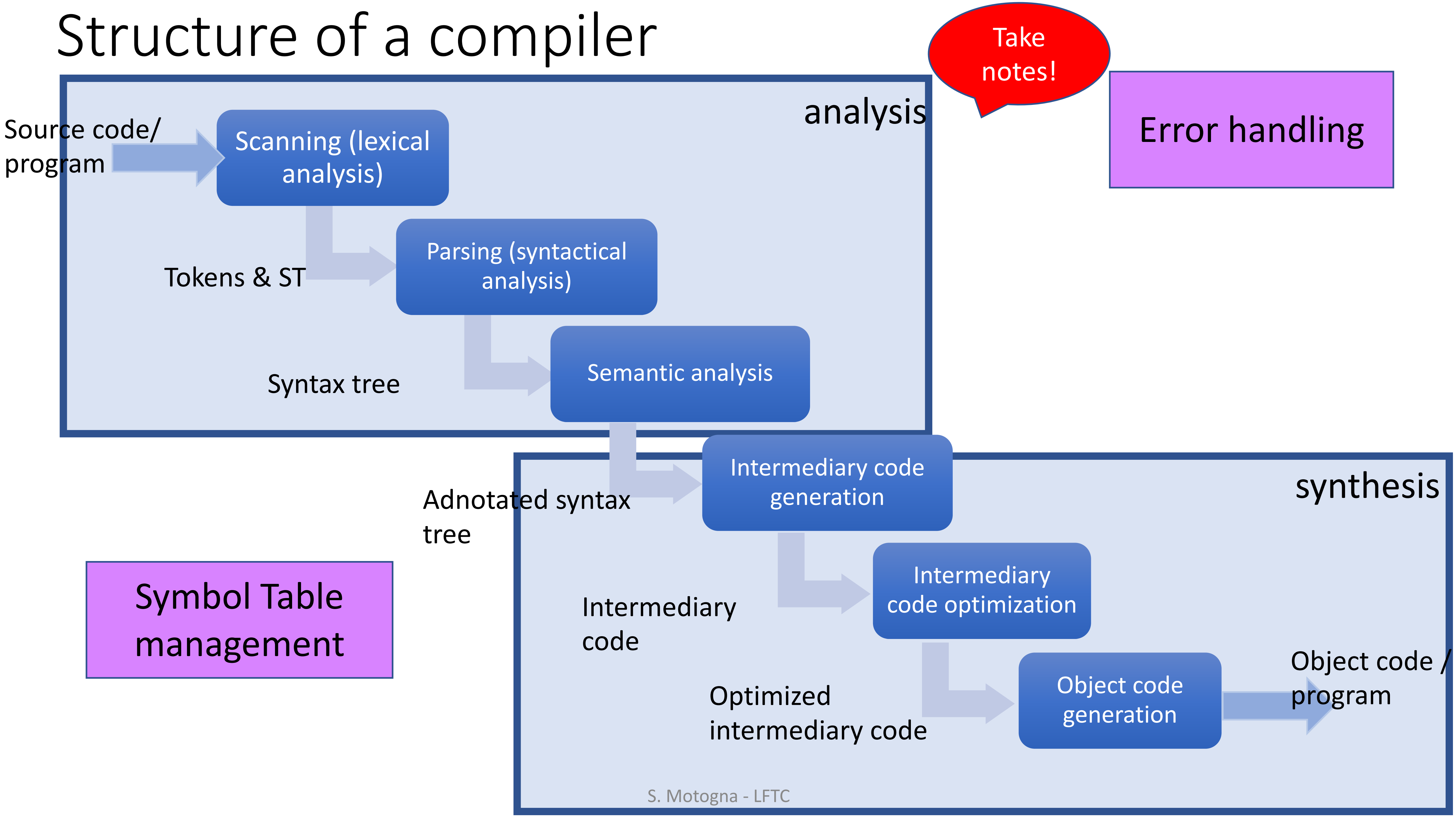
Lisp
1962
McCarthy

Pascal
1968 - 1970
N. Wirth

C
1969 - 1973
D. Ritchie

Java
1995
J. Gosling

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 S*T*; 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

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 <u>symbolic names</u> 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)

$h : K \rightarrow A$

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

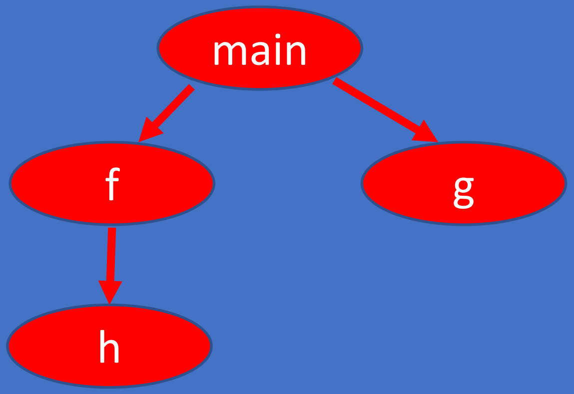
- Conflicts: $k_1 \neq k_2$, $h(k_1) = h(k_2)$

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

Visibility domain (scope)

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

Hierachical structure of STs:



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.:
$L = \{a^n b^n | \ n>0\}$ $L = \{ab, aabb, aaabbb, \ldots\}$
$L' = \{01^n | n>=0\}$ $L' = \{0, 01, 011, \ldots\}$

Example

a boy has a dog

S→PV
P→ *a* N
N→ *boy* or N→ *dog*
(N→*boy*|*dog*)
V → QC
Q → *has*
C → BN
B → *a*

- A→ α = **rule**
- S,P,V,N,Q,C,B = **nonterminal symbols**
- *a, boy,dog,has* = **terminal symbols**

Remarks

1. Sentence = word, sequence (contains only terminal symbols) ; denoted w.
2. S⇒PV⇒a NV⇒a NQC⇒a N has C - sentential form

   In general : $w=a_1a_2. . . a_n$

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 $|N| < \infty$
  - Σ - set of terminal symbols (alphabet) and $|\Sigma|<\infty$
  - P – finite set of productions (rules), with the propriety:

    $P\subseteq(N\cup\Sigma)^* N(N\cup\Sigma)^* \mathbf{x} (N\cup\Sigma)^*$
  - $S\in N$ – 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,…} {a^0}

A = {a}

A+ = {a,aa,aaa,…}

$X^0 = \varepsilon$

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

- ***Direct derivation***
  $\alpha \Rightarrow \beta$, $\alpha,\beta \in (N \cup \Sigma)^*$ ***if*** $\alpha = x1xy1$, $\beta = x1yy1$ ***and*** $x \rightarrow y \in P$

  (x is transformed in y)

- ***k derivation***

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

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

- ***+ derivation***

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

- **** derivation***

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

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

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

L1 = {a,b,aa}
L2 = {c,d,cd}
L1L2 = {ac,ad,acd,bc,bd,bcd,aac,aad,aacd}

Remarks:

1. $S\overset{*}{\Rightarrow} \alpha, \alpha\in(N\cup\Sigma)^*$ = sentential form
   $S\overset{*}{\Rightarrow} w, w\in\Sigma^*$ = word / sequence
2. Operations defined for languages (sets) :
   L1∪L2 , L1∩L2 , L1-L2 , $\overline{L}$ (complement) , $L^+=\bigcup_{k>0} L^k$, $L^*=\bigcup_{k\geq 0} L^k$
   Concatenation: $L=L_1L_2 = \{w_1w_2 \mid w_1\in L_1 , w_2\in L_2\}$
3. $|w|=0$ (empty word - denoted ε)

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 $\alpha \rightarrow \beta \in P$)

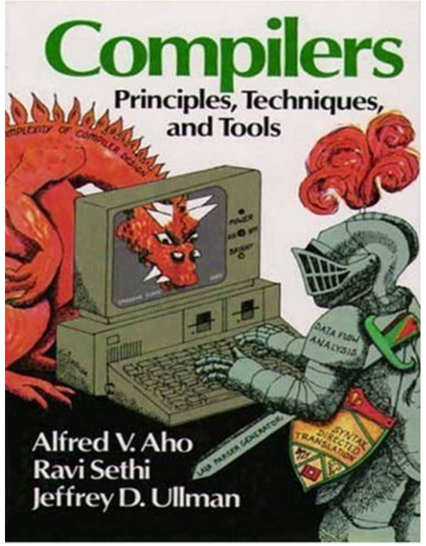
- type 0 : no restriction
- type 1 : context dependent grammar ($x_1Ay_1 \rightarrow x_1\gamma y_1$)
- type 2 : context free grammar ($A \rightarrow \alpha \in P$,where $A\in N$ and $\alpha\in(N \cup \Sigma)^*$)
- type 3 : regular grammar ($A \rightarrow aB|a \in P$)

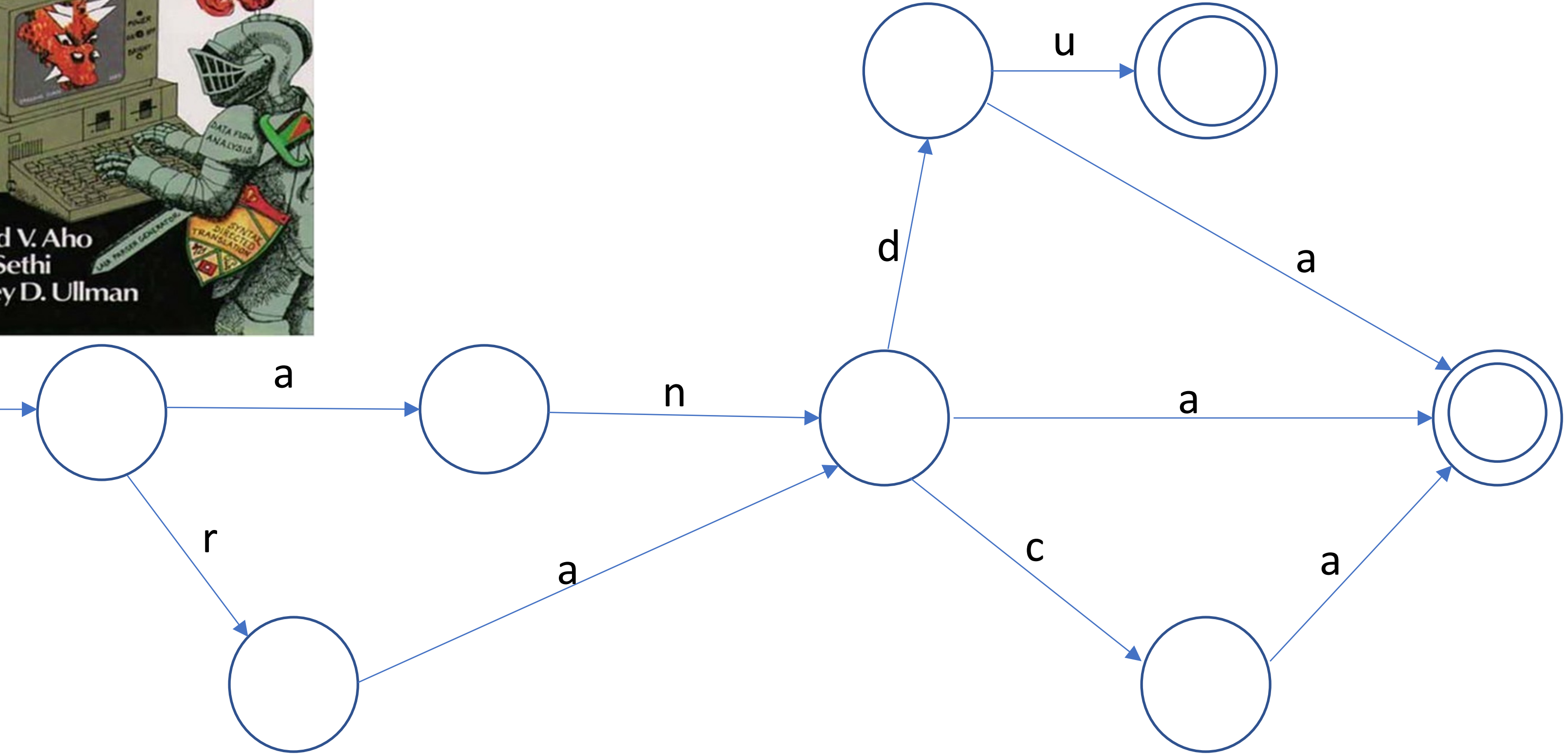
Remark :

type 3 ⊆ type 2 ⊆ type 1 ⊆ type 0

Notations

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

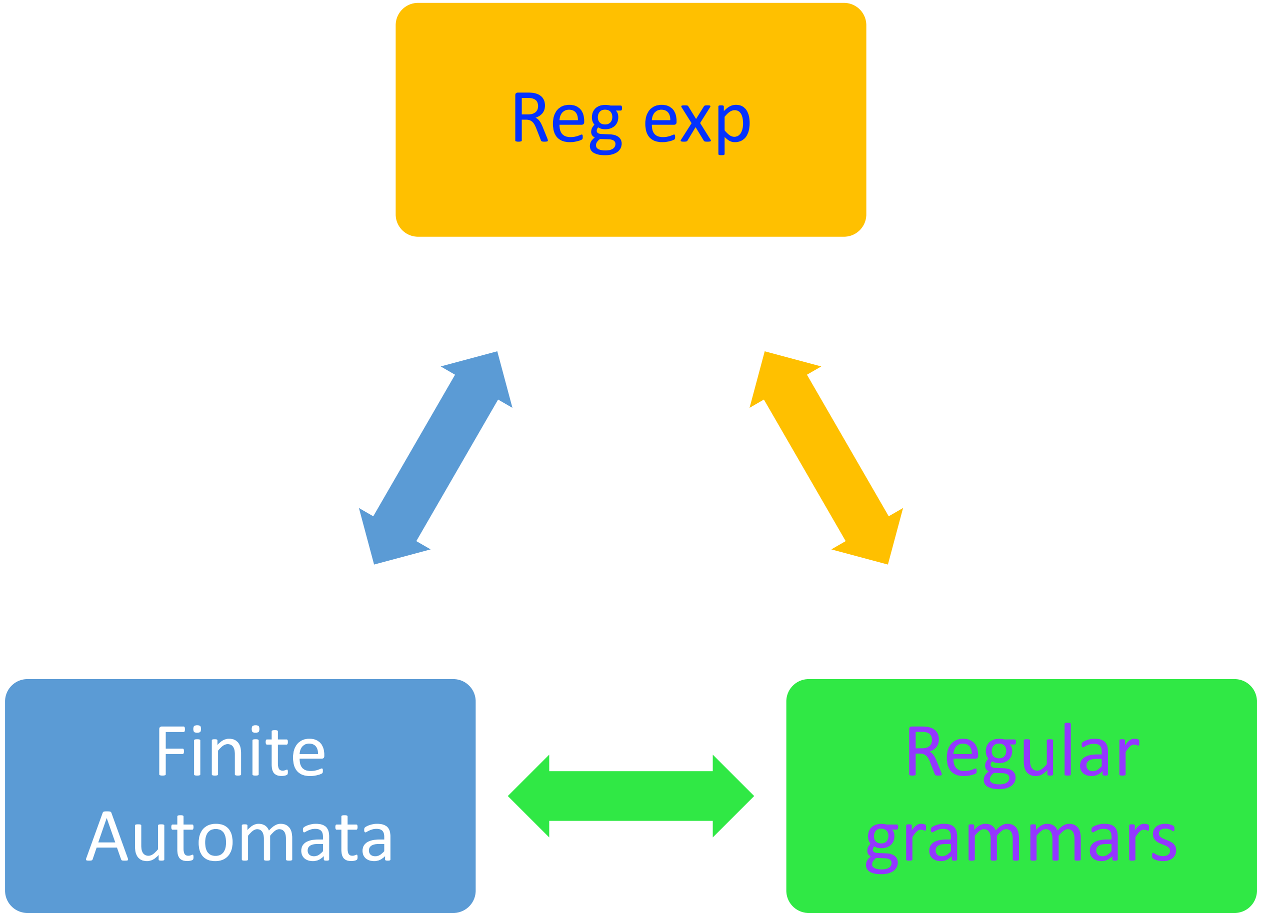
Formal Languages

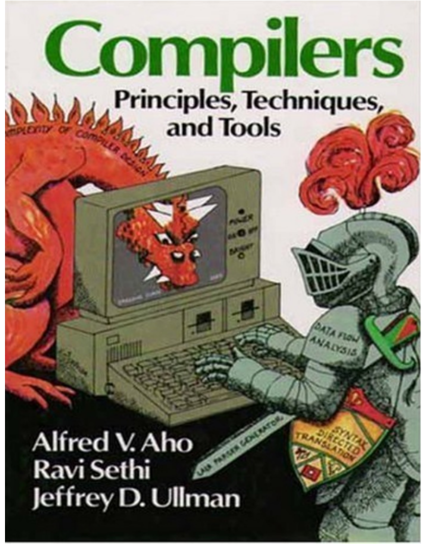
- Basic notions -

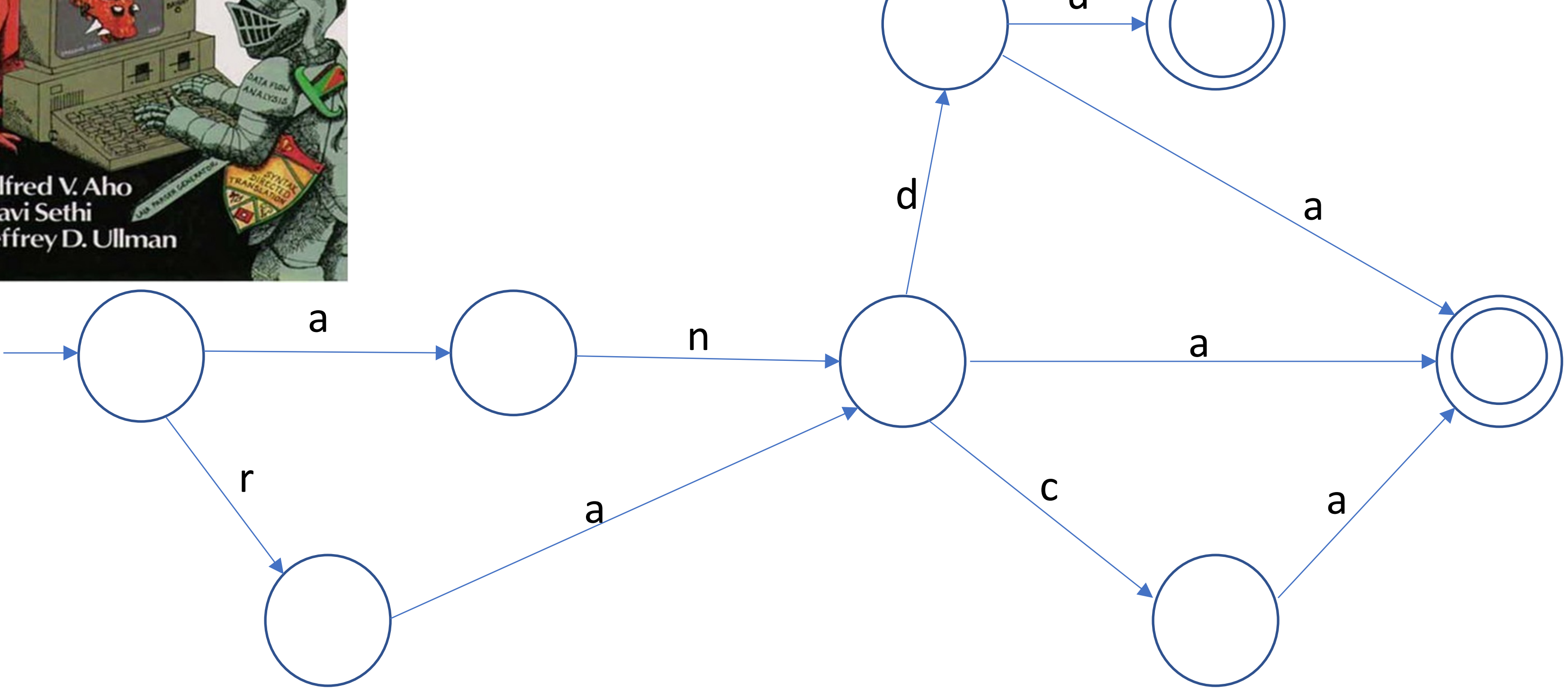
Regular languages

Why?

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


Reg exp
Finite Automata
Regular grammars

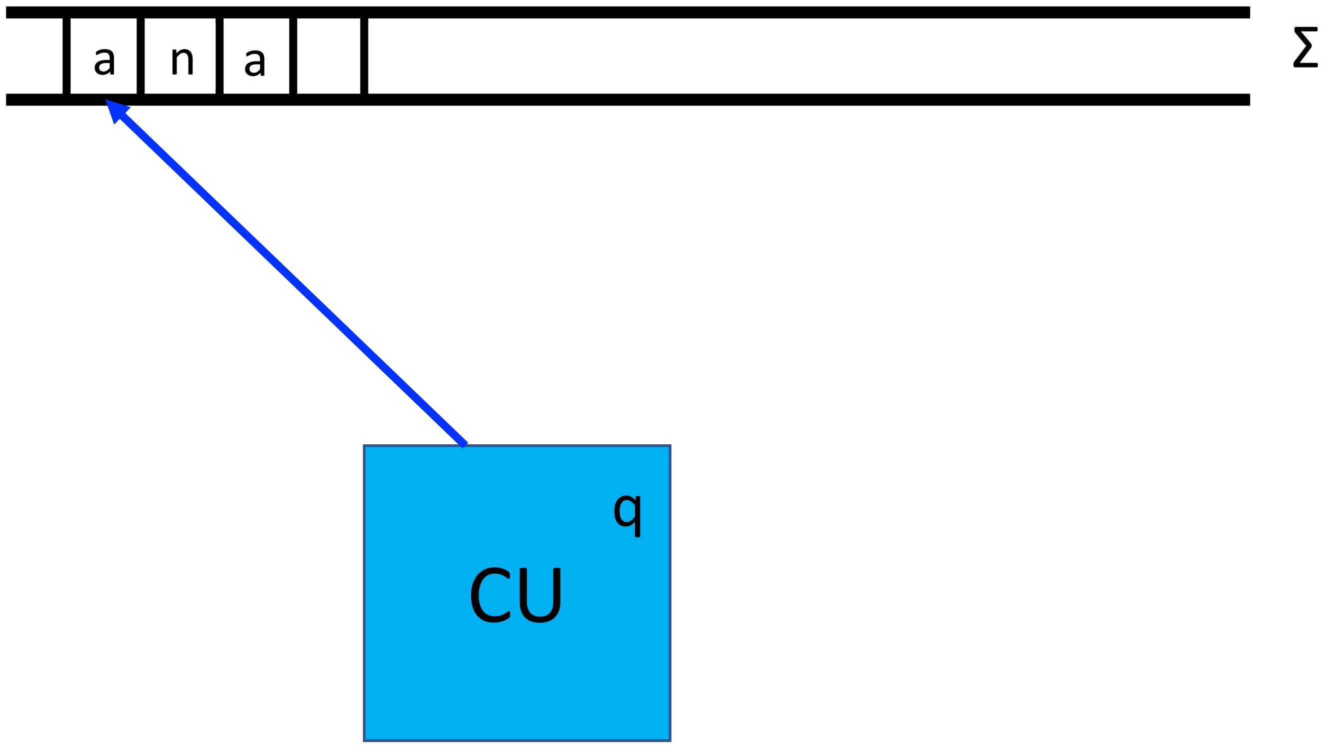




Problem: The door to the tower is closed by the Red Dragon, using a complicated machinery. Prince Charming has managed to steal the plans and is asking for your help. Can you help him determining all the person names that can unlock the door

Finite Automata

- Intuitive model



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

$$M = (Q,\Sigma,\delta,q0,F)$$

where:

- Q - finite set of states ($|Q|<\infty$)
- Σ - finite alphabet ($|\Sigma|<\infty$)
- δ – transition function : $\delta:Q\times\Sigma\to P(Q)$
- 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 $|\delta(q,a)| \leq 1$ => deterministic finite automaton (DFA)
4. If $|\delta(q,a)| > 1$ (more than a state obtained as result) => 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 ,ε) , $q_f \in F$, ε –empty sequence

(corresponds to accept)

Relations between configurations

- $\vdash$ **move / transition** (simple, one step)
  $(q,ax) \vdash (p,x)$, $p \in \delta(q,a)$

- $\overset{k}{\vdash}$ **k move** = a sequence of k simple transitions) $C_0 \vdash C_1 \vdash \ldots \vdash C_k$
- $\overset{+}{\vdash}$ **+ move**
  $C \overset{+}{\vdash} C'$: $\exists$ $k>0$ such that $C \overset{k}{\vdash} C'$
- $\overset{*}{\vdash}$ *** move (star move)**
  $C \overset{*}{\vdash} C'$: $\exists$ $k\geq 0$ such that $C \overset{k}{\vdash} C'$

Definition : ***Language*** accepted by FA M = (Q,Σ,δ,q0,F) is:

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

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)

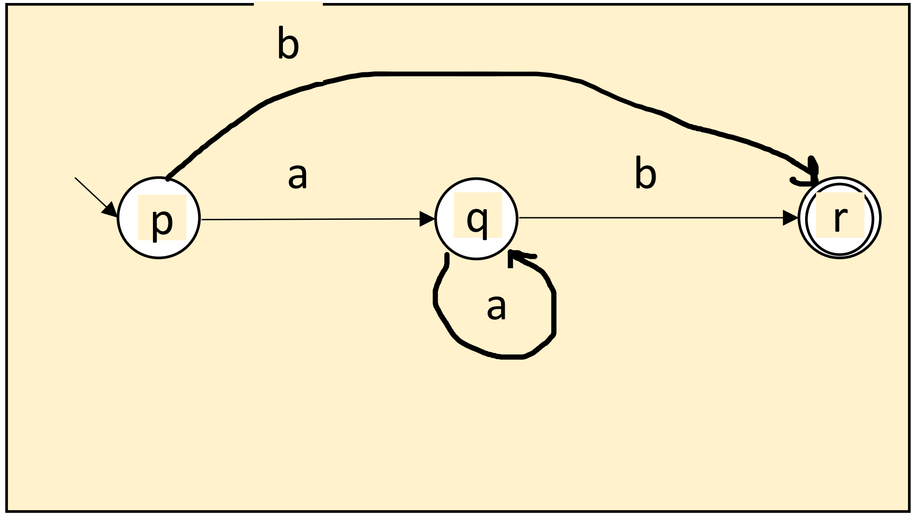
Representing FA

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	-	-



$(p,aab)|\!-(q,ab)|\!-(q,b)|\!-(r,\varepsilon)$ => aab accepted
$(p,aba)|\!-(q,ba)|\!-(r,a)$ => aba not accepted

Remember

- **Grammar**

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

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

- Finite automaton

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

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

Regular grammars

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

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

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

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

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

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

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

Proof: construct M based on G

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

$q_0 = S$

$F = \{K\} \cup \{S \mid \text{if } S \rightarrow \boldsymbol{\varepsilon} \in P\}$

δ: if $A \rightarrow aB \in P$ then $\delta(A,a) = B$

if $A \rightarrow a \in P$ then $\delta(A,a) = K$

Prove that L(G) = L(M)   (w∈L(G) ⇔ w∈L(M)):

$S \stackrel{*}{\Rightarrow} w \Leftrightarrow (S,w) \stackrel{*}{\vdash} (qf, \boldsymbol{\varepsilon})$

$w= \boldsymbol{\varepsilon}$: $S \stackrel{*}{\Rightarrow} \boldsymbol{\varepsilon} \Leftrightarrow (S, \boldsymbol{\varepsilon}) \stackrel{*}{\vdash} (S, \boldsymbol{\varepsilon})$ – true

$w=a_1a_2...a_n$: $S \stackrel{*}{\Rightarrow} w \Leftrightarrow (S,w) \stackrel{*}{\vdash} (K, \boldsymbol{\varepsilon})$

$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow ... \Rightarrow a_1a_2...a_{n-1}A_{n-1} \Rightarrow a_1a_2...a_{n-1}a_n$

$S \Rightarrow a_1A_1$ exists if $S \rightarrow a_1A_1$ and then $\delta(S,a_1)=A_1$

$A_1 \rightarrow a_2A_2$: $\delta(A_1,a_2)=A_2$ . . .

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

$(S,a_1a_2...a_n) \vdash (A_1,a_2...a_n) \vdash (A_2,a_3...a_n) \vdash ... \vdash (A_{n-1},a_n) \vdash (K, \boldsymbol{\varepsilon})$, $K \in F$

Theorem 2: For any FA $M=(Q, \boldsymbol{\Sigma}, \delta, q_0, F)$ there exists a right linear grammar $G=(N, \boldsymbol{\Sigma}, P, S)$ 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 $S \rightarrow \boldsymbol{\varepsilon}$

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

P(i): $q \overset{i+1}{\Rightarrow} x \Leftrightarrow (q,x) \overset{i}{\vdash} (q_f , \boldsymbol{\varepsilon})$, $q_f \in F$ -prove by induction

Apply P : $q_0 \overset{i+1}{\Rightarrow} w \Leftrightarrow (q_0,w) \overset{i}{\vdash} (q_f , \boldsymbol{\varepsilon})$, $q_f \in F$

If i=0: $q \Rightarrow x \Leftrightarrow (q,x) \overset{0}{\vdash} (q_f , \boldsymbol{\varepsilon})$ $(x= \boldsymbol{\varepsilon}, q=q_f)$ $q \Rightarrow \boldsymbol{\varepsilon} \Leftrightarrow q_0 \rightarrow \boldsymbol{\varepsilon}$, $q_0 \in F$

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

$q \overset{i+1}{\Rightarrow} x \Leftrightarrow (q,x) \overset{i}{\vdash} (q_f , \boldsymbol{\varepsilon})$

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

If $q \Rightarrow ap$ then $\delta(q,a)= p$; if $p \overset{i}{\Rightarrow} ax$ then $(p,x) \overset{i-1}{\vdash} (q_f , \boldsymbol{\varepsilon})$, qf$\in$F

THEN $(q,ax) \overset{i}{\vdash} (q_f , \boldsymbol{\varepsilon})$, qf$\in$F

Regular sets

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

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

Regular expressions

Definition: Let Σ be a finite alphabet. We define regular expressions over Σ recursively in the following way:

1. $\boldsymbol{\Phi}$ is a regular expression denoting the regular set $\boldsymbol{\Phi}$ (empty set)
2. $\boldsymbol{\varepsilon}$ is a regular expression denoting the regular set $\{\boldsymbol{\varepsilon}\}$
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 paranthesis 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 $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ be regular expressions.

1. $\boldsymbol{\alpha + \beta = \beta + \alpha}$
2. $\boldsymbol{\Phi^* = \varepsilon}$
3. $\boldsymbol{\alpha + (\beta + \gamma) = (\alpha + \beta) + \gamma}$
4. $\boldsymbol{\alpha(\beta\gamma) = (\alpha\beta)\gamma}$
5. $\boldsymbol{\alpha (\beta + \gamma) = \alpha\beta + \alpha\gamma}$
6. $\boldsymbol{(\alpha + \beta)\gamma = \alpha\gamma + \beta\gamma}$
7. $\boldsymbol{\alpha \varepsilon = \varepsilon \alpha = \alpha}$
8. $\boldsymbol{\Phi\alpha = \alpha\Phi = \Phi}$
9. $\boldsymbol{\alpha^* = \alpha + \alpha^*}$
10. $\boldsymbol{(\alpha^*)^* = \alpha^*}$
11. $\boldsymbol{\alpha + \alpha = \alpha}$
12. $\boldsymbol{\alpha + \Phi = \alpha}$

Reg exp equations

- Normal form: **X = aX + b**

  where a,b – reg exp

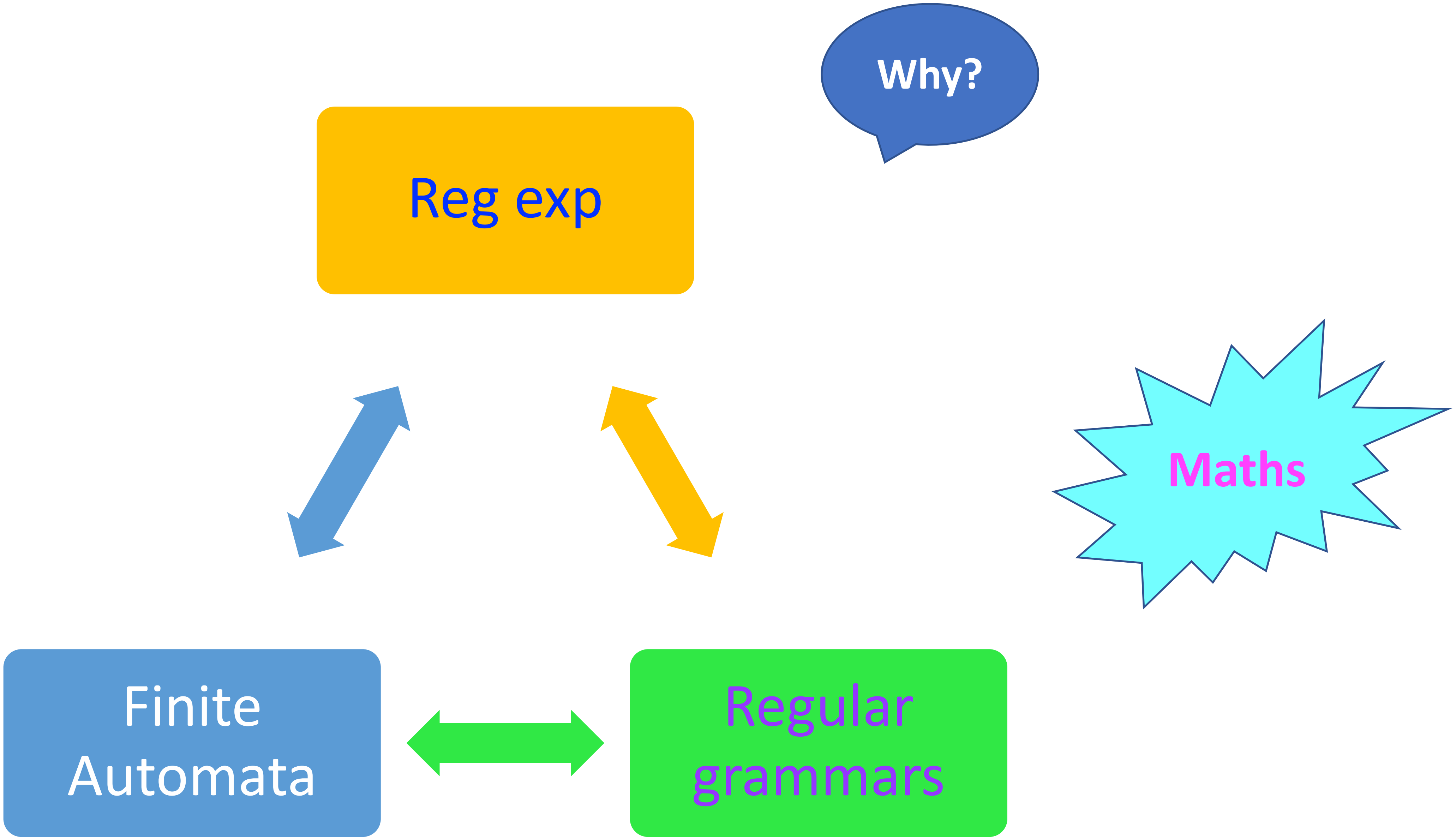
- Solution: **X = a*b**

a a*b + b = (aa* +**ε**)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)


Why?
Reg exp
Maths
Finite Automata
Regular grammars

Prop:*Regular sets are right linear languages*

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

Proof: constructive

i. G = ({S}, $\boldsymbol{\Sigma}$, $\boldsymbol{\Phi}$, S) – regular grammar such that L(G) = $\boldsymbol{\Phi}$

ii. G = ({S}, $\boldsymbol{\Sigma}$,{S→$\boldsymbol{\varepsilon}$}, S) – regular grammar such that L(G) ={$\boldsymbol{\varepsilon}$}

iii. G = ({S}, $\boldsymbol{\Sigma}$,{S→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 => $\exists G_1, G_2$ such that

$G_1 = (N_1, \boldsymbol{\Sigma}_1, P_1, S_1)$ and $L_1 = L(G_1)$

$G_2 = (N_2, \boldsymbol{\Sigma}_2, P_2, S_2)$ and $L_2 = L(G_2)$ assume $N_1 \cap N_2 = \emptyset$

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

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

$P_3 = P_1 \cup P_2 \cup \cancel{\{S_3 \to S_1 \mid S_2\}}$

$\{S_3 \to \alpha_1 \mid S_1 \to \alpha_1 \in P_1\} \cup \{S_3 \to \alpha_2 \mid S_2 \to \alpha_2 \in P_2\}$

G_3 – right linear language

and

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

PROOF!!! Homework

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

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

$P_4 =$ $\{A \rightarrow aB \mid \text{if } A \rightarrow aB \in P_1\} \cup$
$\{A \rightarrow aS_2 \mid \text{if } A \rightarrow a \in P_1\} \cup$
$P_2 \cup$
$\{S_1 \rightarrow \boldsymbol{\alpha}_2 \mid \text{if } S_1 \rightarrow \boldsymbol{\varepsilon} \in P_1 \text{ and } S_2 \rightarrow \boldsymbol{\alpha}_2 \in P_2\}$

G_4 – right linear language
and
$L(G_4) = L(G_1)\, L(G_2)$

PROOF!!! Homework

iii. $G_5 = (N_5, \boldsymbol{\Sigma}_1, P_5, S_5)$

//IDEA: concatenate L_1 with itself

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

$P_5 = \quad P_1 \cup \{S_5 \rightarrow \boldsymbol{\varepsilon}\} \cup$

$\{S_5 \rightarrow \boldsymbol{\alpha}_1 \mid S_1 \rightarrow \boldsymbol{\alpha}_1 \in P_1\} \cup$

$\{A \rightarrow aS_1 \mid \text{if } A \rightarrow a \in P_1\}$

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 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: $S \to 0A \mid 1B \mid \boldsymbol{\varepsilon}$

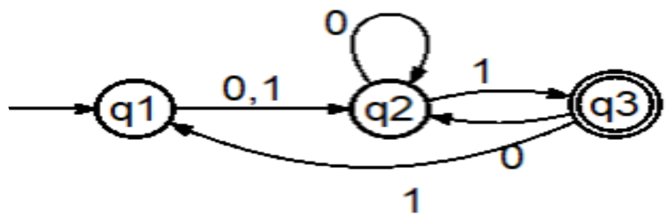
$A \to 0B \mid 1A$

$B \to 0S \mid 1$

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

Regular exp = solution corresponding to S

Theorem: *A language is a regular set if and only if 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 => solution X=ba*

$$\begin{cases} q_1 = q_3 0 + \boldsymbol{\varepsilon} \\ q_2 = q_1 0 + q_1 1 + q_2 0 + q_3 0 \\ q_3 = q_2 1 \end{cases}$$

Regular exp = union of solutions corresponding to final states

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

Reg exp	FA
$\boldsymbol{\Phi}$	$M = (Q, \boldsymbol{\Sigma}, \delta, q_{0,} \boldsymbol{\Phi})$
$\boldsymbol{\varepsilon}$	$M = (Q, \boldsymbol{\Sigma}, \boldsymbol{\Phi}, q_{0,} \{q_0\})$
a,$\forall a \in \boldsymbol{\Sigma}$	$M = (\{q_0,q_1\}, \boldsymbol{\Sigma}, \{\delta(q_0,a) = q_1\}, q_{0,} \{q_1\})$

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:

$M_1 = (Q_1, \boldsymbol{\Sigma}_1, \delta_1, q_{01}, F_1)$ such that $L_1 = L(M_1)$

$M_2 = (Q_2, \boldsymbol{\Sigma}_2, \delta_2, q_{02}, F_2)$ such that $L_2 = L(M_2)$

$M_3 = (Q_3, \boldsymbol{\Sigma}_{1U}, \delta_3, q_{03}, F_3)$

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

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

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

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

PROOF!!! Homework

$M_4 = (Q_4, \boldsymbol{\Sigma}_4, \delta_4, q_{04}, F_4)$

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

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

$$\delta_3(q,a) = \begin{cases} \delta_1(q,a), \text{ if } q \in Q_1 - F_1 \\ \delta_1(q,a) \cup \delta_2(q_{02},a) \text{ if } q \in F_1 \\ \delta_2(q,a), \text{ if } q \in Q_2 \end{cases}$$

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

PROOF!!! Homework

$M_5 = (Q_5, \boldsymbol{\Sigma}_1, \delta_5, q_{05}, F_5)$ *//IDEA: concatenate with itself*

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

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

$$\delta_5(q,a) = \begin{cases} \delta_1(q,a), \text{ if } q \in Q_1\text{-}F_1 \\ \delta_1(q,a) \cup \delta_1(q_{01},a) \text{ if } q \in F_1 \end{cases}$$

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

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: $L = \{0^n1^n \mid n >= 0\}$

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 $0<|y|<=p$

and

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

Proof

L regular => $\exists$ M = $(Q, \boldsymbol{\Sigma}, \boldsymbol{\delta}, q_0, F)$ such that L= L(M)

Let |Q| = p

$$\left.\begin{array}{l} \text{If } w \in L(M)\text{: } (q_0,w) \vdash^{*} (q_f\,, \boldsymbol{\varepsilon})\,,\ q_f \in F \\ \qquad\quad \text{and} \\ |w|>p \end{array}\right\} \begin{array}{l} \text{process at least p+1 symbols} \\ \\ \text{p states} \end{array}$$

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

$(q_0,xyz) \vdash^{*} (q1,yz) \vdash^{+} (q1,z) \vdash^{*} (q_f\,, \boldsymbol{\varepsilon})\,,\ q_f \in F \Rightarrow 0 <= |y| <= p$

Proof (cont)

$(q_0, xy^i z) \vdash^* (q_1, y^i z)$

$\vdash^* (q_1, y^{i-1} z)$

$\vdash^* \ldots$

$\vdash^* (q_1, yz)$

$\vdash^* (q_1, z)$

$\vdash^* (q_f, \boldsymbol{\varepsilon}), q_f \in F$

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

If $i = 0$: $(q_0, xz) \vdash^* (q_1, z) \vdash^* (q_f, \boldsymbol{\varepsilon}), q_f \in F$

Example: $L = \{0^n1^n \mid n >= 0\}$

Suppose L is regular => $w = xyz = 0^n1^n$

Consider all possible decomposition =>

Case 1. $y = 0^k$

$$xyz = 0^{n-k}0^k1^n;\ xy^iz = 0^{n-k}0^{ik}1^n \notin L$$

Case 2. $y = 1^k$

$$xyz = 0^n1^k1^{n-k};\ xy^iz = 0^n1^{ik}1^{n-k} \notin L$$

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

$$xyz = 0^{n-k}0^k1^l1^{n-l};\ xy^iz = 0^{n-k}(0^k1^l)^i1^{n-l} \notin L$$

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

$$xyz = 0^{n-k}0^k1^k1^{n-k};\ xy^iz = 0^{n-k}0^k1^k0^k1^k\ldots1^{n-l} \notin L$$

=> L is not regular

Context free grammars (cfg)

Context free grammar (cfg)

- Procdutions of the form: A $\rightarrow \alpha$, A∈N, α∈(N∪$\boldsymbol{\Sigma}$)*

- More powerful

- Can model programming language:

  G = (N, $\boldsymbol{\Sigma}$,P,S) s.t. L(G) = 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 $X_1, X_2, \ldots, X_n$ only if $A \rightarrow X_1X_2\ldots X_n \in P$

Remarks:

a) Parse tree = syntax tree – result of parsing (syntatic analysis)

b) Derivation tree – condition 2.2 not satisfied

c) Abstract syntax tree (AST) ≠ syntax tree (semantic analysis)

Syntax tree (cont)

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

Proof: HomeWork

Example: S-> aSbS | c; w = aacbcbc

Leftmost derivations	**Rightmost derivations**
S => aSbS => aaSbSbS => aacbSbS => aacbcbS => aacbcbc	S => aSbS => aSbc => aaSbSbc => aaSbcbc => aacbcbc

Definition: A cfg G = (N, Σ,P,S) is ambigous if for a w ∈ L(G) there exists 2 distinct syntax tree with frontier w.

Example:

Parsing (syntax analysis) modeled with cfg:

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

- N – nonterminal: syntactical constructions: declaration, statement, expression, a.s.o.
- $\boldsymbol{\Sigma}$ – 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 <=> w ∈ L(G)

Equivalent transformation of cfg

- Unproductive symbols
- Inaccesible symbols

- ε - productions
- Single productions

1. Determine elements (symbols/ productions): Greedy alg
2. eliminate them: construct equivalent grammar

Unproductive symbols

<u>Definition</u>
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 $N_0, N_1, \ldots$ recursively (until saturation)

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

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

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

else $N' = N_i$

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

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

Example

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

P: $S \rightarrow aA \mid aC$

$A \rightarrow AB$

$B \rightarrow b$

$C \rightarrow aC \mid CD$

$D \rightarrow b$

Inaccesible symbols

Definition

A symbol $X \in N \cup \Sigma$ is ***inaccesible*** 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*

$\forall X \in N \cup \Sigma\ \exists \alpha, \beta \in (N' \cup \Sigma')^*$ 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 $V_i <> V_{i-1}$ 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: $S \to aA \mid aC$

$A \to AB$

$B \to b$

$C \to aC \mid bCb$

$D \to bB \mid d$

ε-productions

Definition

A cfg G=(N,Σ,P,S) is without ***ε-productions*** if

1. $P \not\ni A \to \varepsilon$ (ε-productions)

OR

2. $\exists\ S \to \varepsilon$ si $S \notin rhs(p), \forall p \in P$

Algorithm 3: Elimination of ε-productions

input: cfg G = (N,Σ,P,S)

output: cfg G' = (N',Σ,P',S')

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

1.a. $N_0 := \{A \mid A \to \varepsilon \in P\}$;

$i := 1$;

1.b. $N_i := N_{i-1} \cup \{A \mid A \to \alpha \in P, \alpha \in N^*_{i-1}\}$

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

else $\overline{N} = N_i$

A->BC

B->ε

C->ε

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

2.a. **if** $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \ldots B_k \alpha_k \in P$, k>=0

and for i := 1,k $B_i \in \overline{N}$

and $\alpha_j \notin \overline{N}$, j:=0,k

then add to P' all prod of the form

$A \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2 \ldots X_k \alpha_k$

where X_i is B_i or ε (not $A \to \varepsilon$)

2.b **if** $S \in \overline{N}$' **then** add S' to N' and $S' \to S \mid \varepsilon$ to P

else N' := N; S' := S.

Example

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

P: $S \rightarrow aA \mid aAbB$

$A \rightarrow aA \mid B$

$B \rightarrow bB \mid \varepsilon$

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 ε-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:

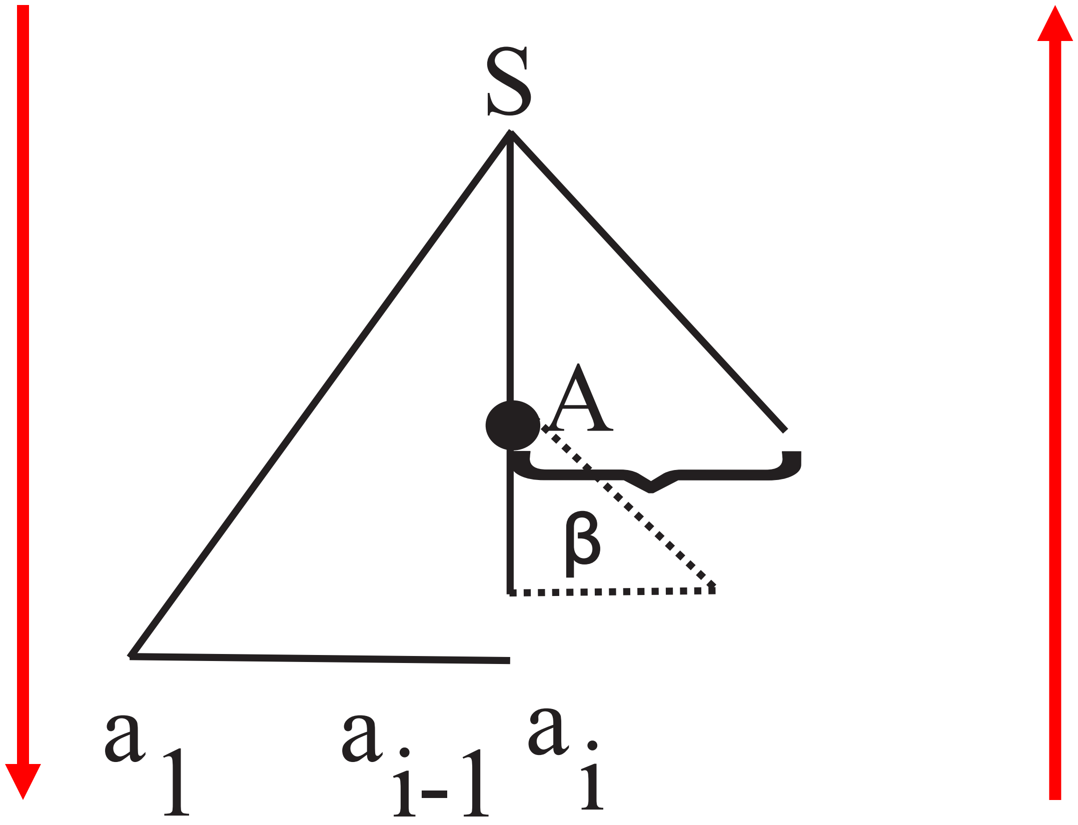
$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid a$

Parsing

- Cfg G = (N, Σ, P,S) check if w ∈ 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 sintactically correct*

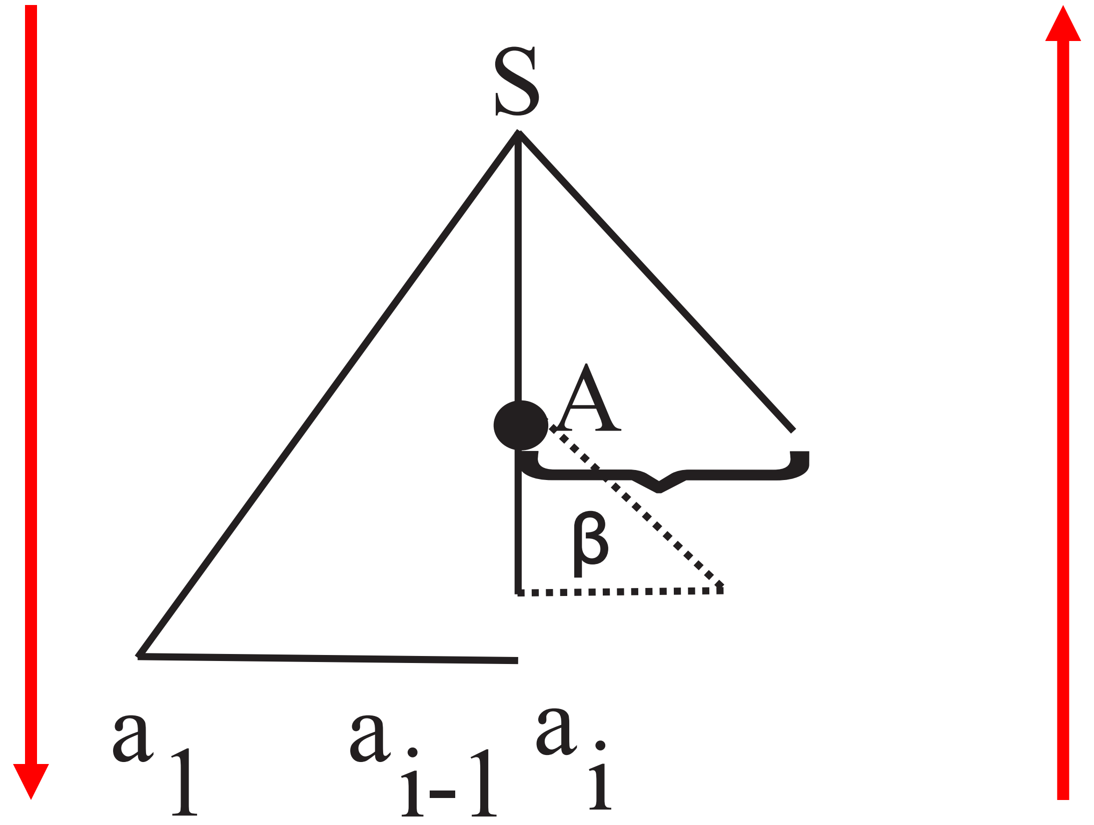
then construct syntax tree

else ”syntax error”

source program is sintactically correct = $w \in L(G) \Leftrightarrow S \overset{*}{=>} w$

Parsing

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

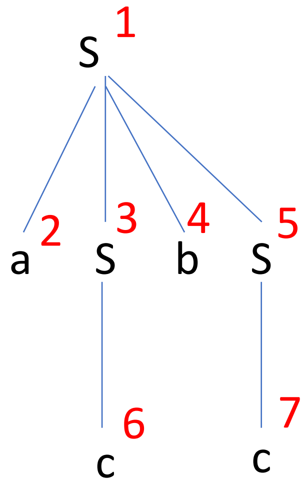


	Descendent	Ascendent
Recursive	Descendent recursive parser	Ascendent recursive parser
Linear	LL(k): $\boxed{\text{LL(1)}}$	LR(k): LR(0), SLR, LR(1), $\boxed{\text{LALR}}$

Result – parse tree -representation

- Arbitrary tree – child sybling representation
- Sequence of derivations S => $\boldsymbol{\alpha}_1$ => $\boldsymbol{\alpha}_2$ =>… => $\boldsymbol{\alpha}_n$ = 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 -> aSbS | aS | c

Formal model

- Configuration

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

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
  $w = a_1 a_2 \ldots a_n$, $i \in \{1,\ldots,n+1\}$
- α = working stack, stores the way the parse is built
- β = input stack, part of the tree to be built

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

Define moves between configurations

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

Expand

WHEN: head of input stack is a nonterminal

$(q,i, \boldsymbol{\alpha}, A\boldsymbol{\beta}) \vdash (q,i, \boldsymbol{\alpha}A_1, \boldsymbol{\gamma}_1\boldsymbol{\beta})$

where:

$A \rightarrow \boldsymbol{\gamma}_1 \mid \boldsymbol{\gamma}_2 \mid \ldots$ represents the productions corresponding to A

1 = first prod of A

Advance

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

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

Momentary insuccess

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

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

Back

WHEN: head of working stack is a terminal

$(b,i, \boldsymbol{\alpha}a, \boldsymbol{\beta}) \vdash (b,i-1, \boldsymbol{\alpha}, a\boldsymbol{\beta})$

Another try

WHEN: head of working stack is a nonterminal

$(b,i,\boldsymbol{\alpha}\, A_j, \boldsymbol{\gamma}_j\, \boldsymbol{\beta}) \vdash$

$(q,i,\boldsymbol{\alpha} A_{j+1}, \boldsymbol{\gamma}_{j+1}\boldsymbol{\beta})$, if $\exists\ A \rightarrow \boldsymbol{\gamma}_{j+1}$

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

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

Success

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

Algorithm

Algorithm Descendent Recursive

INPUT: G, w =$a_1a_2\ldots a_n$
OUTPUT: string of productions and message

```
config = (q,1, ε,S);                                   //initial configuration (s,i,α,β)
while (s ≠ f) and (s ≠ e) do
   if s = q
     then if (i=n+1) and IsEmpty(β)
             then Success(config)
             else
                  if Head(β) = A
                     then Expand(config)
                     else
                        if  Head(β) = a_i
                            then Advance(config)
                            else MomentaryInsuccess(config)
     else
         if s = b
            then
                 if Head(α) = a
                   then Back(config)
                   else AnotherTry(config)
endWhile
if s = e  then message"Error"
         else message "Sequence accepted";
               BuildStringOfProd(α)
```

w ∈ L(G) - HOW

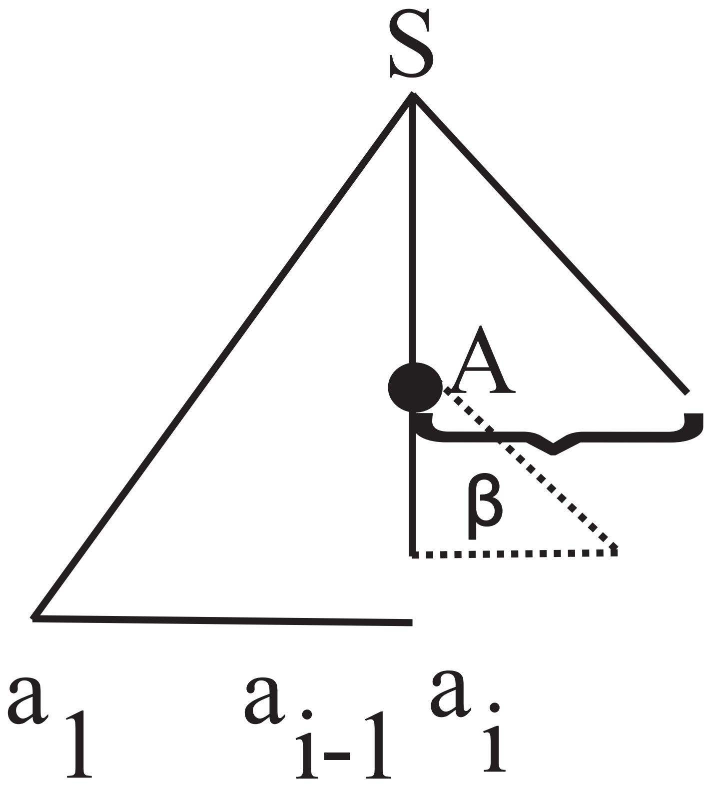
- Process $\boldsymbol{\alpha}$:
  - From left to right (reverse if stored as stack)
  - Skip terminal symbols
  - Nonterminals – index of prod

- Example: $\boldsymbol{\alpha} = S_1 \text{ a } S_2 \text{ a } S_3 \text{ c b } S_3 \text{ c}$

When the algorithm never stops?

- S->S**α** – expand infinitely (left recursive)

LL(1) Parser



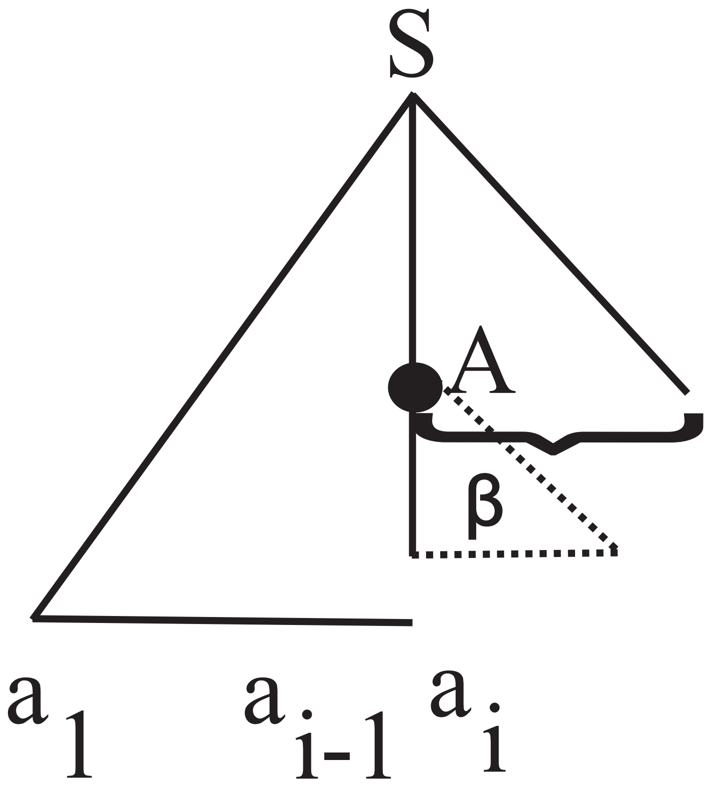
Linear algorithm

FIRST_k

- ≈ first k terminal symbols that can be generated from α
- **Definition**:

$$FIRST_k : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma^k)$$
$$FIRST_k(\alpha) = \{u | u \in \Sigma^k, \alpha \overset{*}{\Rightarrow} ux, |u| = k \text{ sau } \alpha \overset{*}{\Rightarrow} u, |u| \leq k\}$$

LL(1) Parser



Linear algorithm

Operation: $\oplus$ = concatenation of length 1

L1 = {aa,ab,ba}

L2 = {00,01}

L1$\oplus$L2 = {a,0}

L1={a, $\boldsymbol{\varepsilon}$}

L2={0,1}

L1$\oplus$L2 ={a,0,1}

FIRST$_k$

- ≈ first k terminal symbols that can be generated from α
- **Definition**:

$$FIRST_k : (N \cup \Sigma)^* \rightarrow \mathcal{P}(\Sigma^k)$$
$$FIRST_k(\alpha) = \{u | u \in \Sigma^k, \alpha \stackrel{*}{\Rightarrow} ux, |u| = k \text{ sau } \alpha \stackrel{*}{\Rightarrow} u, |u| \leq k\}$$

$FIRST_k$

- Which are the first k terminal symbols that can be generated from A?

- https://forms.office.com/r/kNHNGW7XtC

Construct FIRST

Concatenation of length 1

- $FIRST_1$ denoted FIRST
- Remarks:
  - If L_1, L_2 are 2 languages over alphabet Σ, 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
  - $FIRST(\alpha\beta) = FIRST(\alpha) \oplus FIRST(\beta)$

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

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;

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

repeat

  i := i+1;

A

  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*$\}$

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

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

    end if

  end for

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

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

A -> BC

B -> DA

D -> a

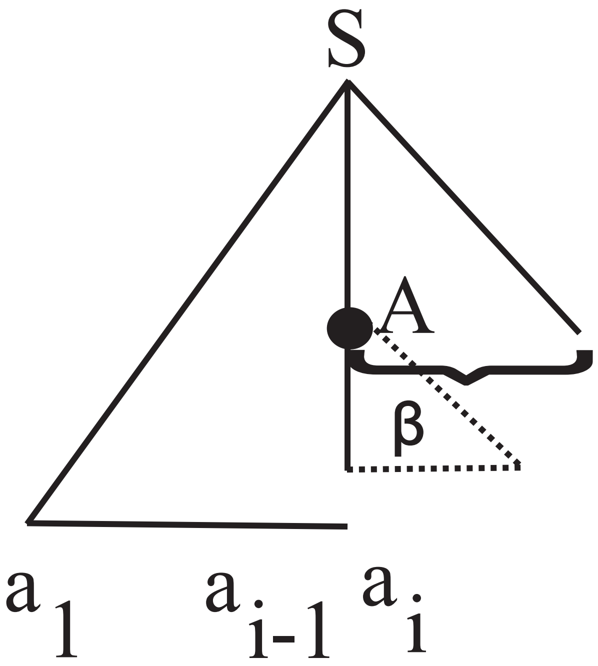
F0(A)=F0(B)=Ø; F0(D)={a}

F1(A) =F0(A) U {…| A->BC F0(B)⨁F(D)}= Ø

F1(B) ={a}

FOLLOW

$A \to \varepsilon$



- $FOLLOW_k(A) \approx$ next k symbols generated after/ following A

Follow(A)
S=>* xBy => xaAy
What if B->uA

$$FOLLOW : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma)$$
$$FOLLOW(\beta) = \{w \in \Sigma | S \stackrel{*}{\Rightarrow} \alpha\beta\gamma, w \in FIRST(\gamma)\}$$

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;
$L_0(S) = \{\varepsilon\}$; {init}
i = 0;
repeat
    i = i+1;
    for $B \in N$ **do**
        for $A \to \alpha B y \in P$ **do**
            for $\forall a \in FIRST(y)$ **do**
                if $a = \varepsilon$ **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 $Fi(X) = Fi\text{-}1(X)$, $\forall X \in N$
$FOLLOW(X) = Fi(X)$, $\forall X \in N$

$S \Rightarrow^0 S$ // ε after S

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

FIRST

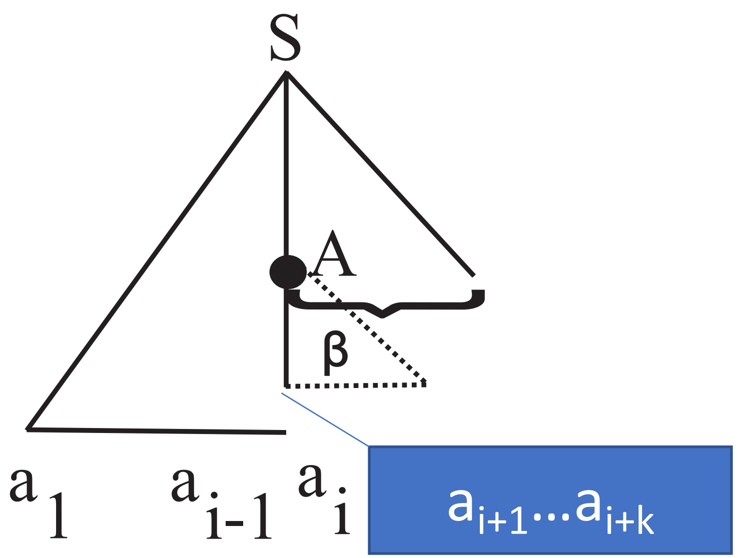
- ≈ first terminal symbols that can be generated from α

FOLLOW

- ≈ 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, <u>acțion is uniquely determinde</u> 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. $S \overset{*}{\Rightarrow}_{st} wA\alpha \Rightarrow_{st} w\beta\alpha \overset{*}{\Rightarrow}_{st} wx;$

2. $S \overset{*}{\Rightarrow}_{st} wA\alpha \Rightarrow_{st} w\gamma\alpha \overset{*}{\Rightarrow}_{st} wy;$

such that $FIRST_k(x) = FIRST_k(y)$ *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:

$$\mathrm{FIRST}_k(\beta\alpha) \cap \mathrm{FIRST}_k(\gamma\alpha) = \Phi, \forall\alpha \quad \text{such that} \quad S \overset{*}{=>} uA\alpha$$

Theorem: A grammar is LL(1) if and only if for any nonterminal A with productions $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$, FIRST(α_i) ∩ FIRST(α_j) = Ø and if $\alpha_i \Rightarrow \varepsilon$, FIRST($\alpha_i$) ∩ FOLLOW(A)= Ø, ∀i,j = 1,n,i≠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 ∈ **N**∪**Σ**
  - Possible prediction ∈ **Σ**
- Add a special character "\$" (∉ **N**∪**Σ**) – marking for "empty stack"

= > table:

- One line for each symbol ∈ **N**∪**Σ** ∪{\$}
- One column for each symbol ∈ **Σ** ∪{\$}

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

   $M(A,b) = (\alpha, i),$ 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)=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: Definire configurations and moves

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

  If (w ∈L(G)) *then* **string of productions**

  else **error** & **location of error**

LL(1) configurations

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

where:

- α = input stack
- β = working stack
- π = output (result)

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

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

Moves

1. Push – put in stack

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

(pop A and push symbols of β)

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

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

3. Accept

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

4. Error - otherwise

Algorithm LL(1) parsing

- INPUT:
  - LL(1) table with NO conflicts;
  - G –grammar (productions)
  - Input sequence $w = a_1a_2 \ldots a_n$
- 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 -> if C then S | if C then S else S // is not LL(1)

    I -> if C then S T

    T -> ε | 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

- <u>Enhanced grammar</u>

- $G = (N, \Sigma, P, S)$
- $G' = (N \cup \{S'\}, \Sigma, P \cup \{S' \rightarrow S\}, S')$, $S' \notin N$

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

  $S \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta w$, 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 $A \rightarrow \alpha\beta$, α was detected (α is in head of stack) and it is expected to detect β.

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

  $S \overset{*}{\Rightarrow}_r \gamma A w \Rightarrow_r \gamma\alpha\beta w$
  $u = FIRST_k(w)$

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

1. $S' \overset{*}{\Rightarrow}_r \alpha A w \Rightarrow_r \alpha\beta w$
2. $S' \overset{*}{\Rightarrow}_r \gamma B x \Rightarrow_r \alpha\beta y$
3. $FIRST_k(w) = FIRST_k(y)$

=> $\alpha = \gamma$ *AND* A =B *AND* x=y

- [A → αβ.,u] – special case: prefix is all rhp - apply reduce

- Otherwise [A → α.β,u] – apply shift

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

⇒Working stack:

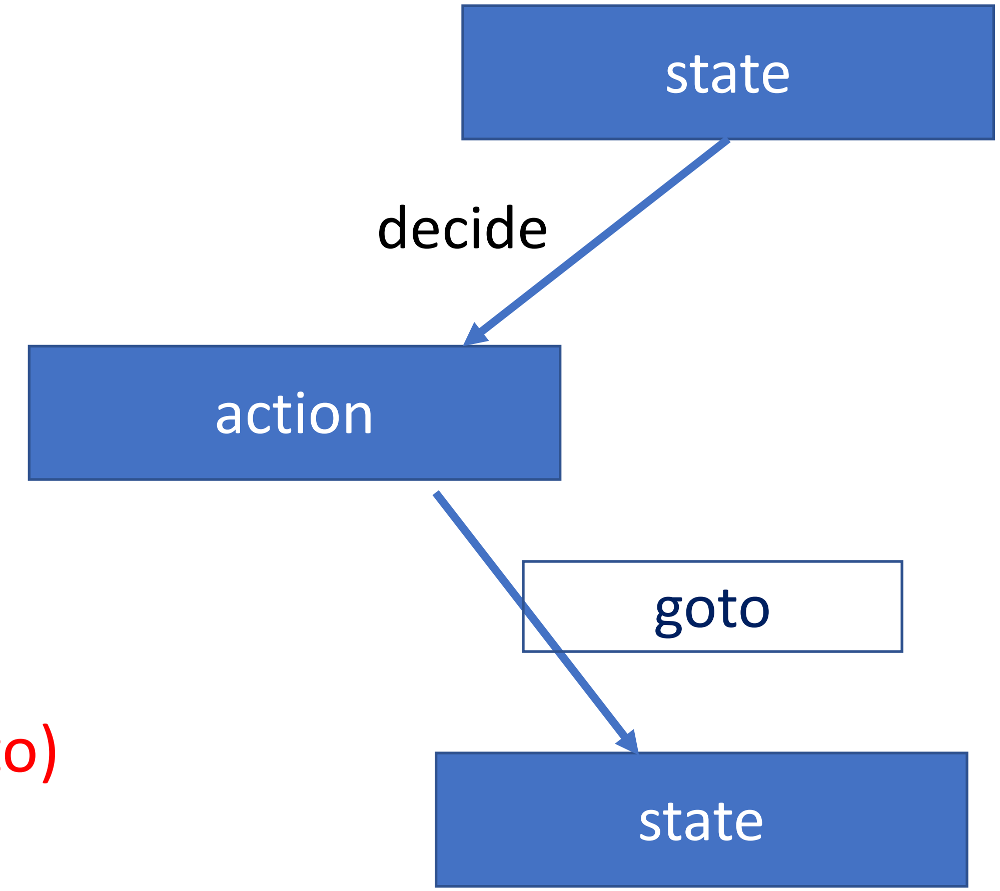
$\$s_{init}X_1s_1 \ldots X_ms_m$

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 → α.Bβ,u] valid for live prefix γα =>

$$S \stackrel{*}{\Rightarrow}_{dr} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w$$

$$u = FIRST_k(w)$$

- B → δ ∈P => $S \stackrel{*}{\Rightarrow}_{dr} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w \stackrel{*}{=>}_{dr} \boldsymbol{\gamma\alpha\delta} w'$

=> [B → .δ,u] valid for live prefix γα

LR(k) parsing:
LR(0), SLR, LR(1), LALR

- Define item
- Construct set of states
- Construct table

Executed 1 time

- Parse sequence based on moves between configurations

LR(0) Parser

- Prediction of length 0 (ignored)

1. LR(0) item: $[A \to \alpha.\beta]$

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 îmbogăţită
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)*

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

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

goto(s, X) = closure({[A → αX.β] | [A → α.Xβ] ∈ s})

Algorithm *ColCan_LR(0)*

INPUT: G'- gramatica îmbogăţită
OUTPUT: C - colecţia canonică de stări

$\mathcal{C} := \emptyset;$
$s_0 := closure(\{[S' \to .S]\})$
$\mathcal{C} := \mathcal{C} \cup \{s_0\};$
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-> aS|bSc|dA
A -> dc

Goto(s0,S)
Goto(s0,A)
Goto(s0,a)
Goto(s0,b)
Goto(s0,c) =Ø
Goto(so,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* $[A \to \alpha.\beta] \in s_i$ *then* **action(s_i)=shift**
2. *if* $[A \to \beta.] \in s_i$ and $A \neq S'$ *then* **action(s_i)=reduce l**, where l = number of production $A \to \beta$
3. *if* $[S' \to S.] \in s_i$ *then* **action(s_i)=acc**
4. *if* 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' → .S]
2) No shift from accept state:

   if s is accept state *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.

3) *If* in state ***s*** action is reduce *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.
4) Argument G': Let G = ({S},{a,b,c},{S → aSbS,S → c},S)

   states [S → aSbS.] and [S → 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'}, $\boldsymbol{\Sigma}$, P U {S'->S},S')
  - LR(0) table
  - Input sequence $w = a_1 \ldots a_n$
- OUTPUT:

  if (w ∈L(G)) *then* **string of productions**

  else **error & location of error**

LR(0) configurations

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

where:

- α = working stack
- β = input stack
- π = output (result) stack

Initial configuration:
($\$s_0$,w$,ε)

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

Moves

1. Shift

if action(s_m)= shift AND head(β)=a_i AND goto(s_m,a_i)=S_j **then**

($\$s_0x_1 \ldots x_m s_m$,$a_i \ldots a_n\$$, π) ⊢ ($\$s_0x_1 \ldots x_m s_m a_i s_j$,$a_{i+1} \ldots a_n\$$, π)

2. Reduce

if action(s_m) = reduce l AND (l) A → $x_{m-p+1} \ldots x_m$ AND goto(s_{m-p},A) = s_j **then**

($\$s_0 \ldots x_m s_m$,$a_i \ldots a_n\$$, π) ⊢ ($\$s_0 \ldots x_{m-p} s_{m-p} A s_j$,$a_i \ldots a_n\$$,l π)

3. Accept

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

4. Error - otherwise

LR(0) Parsing Algorithm

INPUT:

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

- 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 l" 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

Executed 1 time

- Parse sequence based on moves between configurations

Algorithm *ColCan_LR(0)*

INPUT: G'- gramatica îmbogăţită
OUTPUT: C - colecţia 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->$\boldsymbol{\alpha}.\boldsymbol{\beta}$]

INPUT: I-element de analiză; G'- gramatica îmbogăţită
OUTPUT: C = closure(I);

$C := \{I\};$ //initialize Closure with the LR(0) item
repeat
  for $\forall[A \to \alpha.B\beta] \in C$ **do** //search productions with dot in front of nonterminal
    for $\forall B \to \gamma \in P$ **do** //search productions of that nonterminal
      if $[B \to .\gamma] \notin C$ **then**
        $C = C \cup [B \to .\gamma]$ //adds item formed from production with dot in
      end if //front of right hand side of the production
    end for
  end for
until C nu se mai modifică

Function *goto*

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

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

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

Prediction = next symbols on input sequence

- SLR = Simple LR

- 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

LR(0)

LR(0)

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 $\in \boldsymbol{\Sigma}$

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

Optimize table structure: merge *action* and *goto* columns for Σ

Remark (LR(0) table):

- *if* s is accept state *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.
- *If* in state ***s*** action is reduce *then* goto(s, X) = ∅, ∀X ∈ N ∪ Σ.

SLR table

And goto

	Action		GOTO	
	a_1 …	a_n	B_1 …	B_m
s_0				
s_1				
…				
s_k				

$a_1,\ldots,a_n \in \boldsymbol{\Sigma}$
$B_1,\ldots,B_m \in N$
$s_0,\ldots,s_k$ - states

Rules for SLR table

1. If $[A \rightarrow \alpha.\beta] \in s_i$ and 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 $\in$ 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 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' = (NU{S'}, $\boldsymbol{\Sigma}$, P U {S'->S},S')
  - SLR table
  - Input sequence w =$a_1 \ldots a_n$
- OUTPUT:

  if (w ∈L(G)) *then* **string of productions**

  else **error & location of error**

SLR = LR(0) configurations

$$(\boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\pi})$$

where:

- α = working stack
- β = input stack
- π = output (result)

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

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

Moves

head(β) = 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) $A \to x_{m-p+1} \dots x_m$ AND goto(s_{m-p},A) = s_j **then**

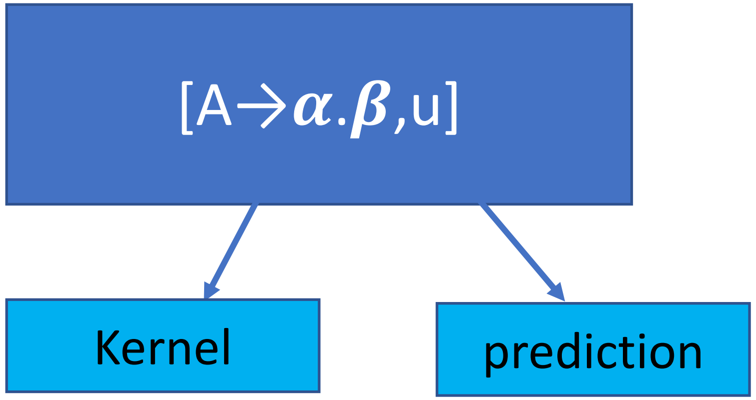
$$(\$s_0 \dots x_m s_m, a_i \dots a_n\$, \pi) \vdash (\$s_0 \dots x_{m-p} s_{m-p} A s_j, a_i \dots a_n\$, t\ \pi)$$

3. **Accept**

if action(s_m,\$) = accept **then** **(**$\$s_m$,\$, π)=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: $\mathcal{C}_1$ – cannonical collection of states

$\mathcal{C}_1 = \emptyset$

S0 = *Closure_LR1*({[S'→.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 ofLR(1) items

Goto_LR1(s, X) = *Closure_LR1*$(\{[A \rightarrow \alpha X.\beta,u] \mid [A \rightarrow \alpha.X\beta,u] \in s\})$

Algorithm *Closure_LR1*

- [A → α.Bβ,u] valid for live prefix γα =>

$$S \overset{*}{\Rightarrow}_{dr} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w$$

$$u = FIRST_k(w)$$

- [B → .δ, *smth*]∈P => $S \overset{*}{\Rightarrow} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w \Rightarrow_{dr} \gamma\alpha\delta\beta w.$

=> [B → .δ,b] valid for live prefix γα,

∀b ∈ FIRST(βu) *// First(βw) = First(βu)*

Algorithm *Closure_LR1*

INPUT: I-element de analiză; G'- gramatica îmbogăţită;
$FIRST(X), \forall X \in N \cup \Sigma$;
OUTPUT: C_1 = closure(I);

$C_1 := \{I\}$;
repeat
  for $\forall [A \to \alpha.B\beta, a] \in C_1$ **do**
    for $\forall B \to \gamma \in P$ **do**
      for $\forall b \in FIRST(\beta a)$ **do**
        if $[B \to .\gamma, b] \notin C_1$ **then**
          $C_1 = C_1 \cup [B \to .\gamma, b]$
        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 $goto(s_i,a) = s_j$ 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 $goto(s_i, X) = s_j$ 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' = (NU{S'}, $\boldsymbol{\Sigma}$, P U {S'->S},S')
  - LR(1) table
  - Input sequence w =$a_1 \ldots a_n$
- OUTPUT:

  if (w ∈L(G)) *then* **string of productions**

  else **error & location of error**

LR(1) configurations

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

where:

- α = working stack
- β = input stack
- π = output (result)

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

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

Moves

head(β) = prediction

1. **Shift**

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

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

2. **Reduce**

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

then

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

3. **Accept**

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

4. **Error** - otherwise

LALR Parser

- LALR = Look Ahead LR(1)

- why?

LALR principle

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

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

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

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

- 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

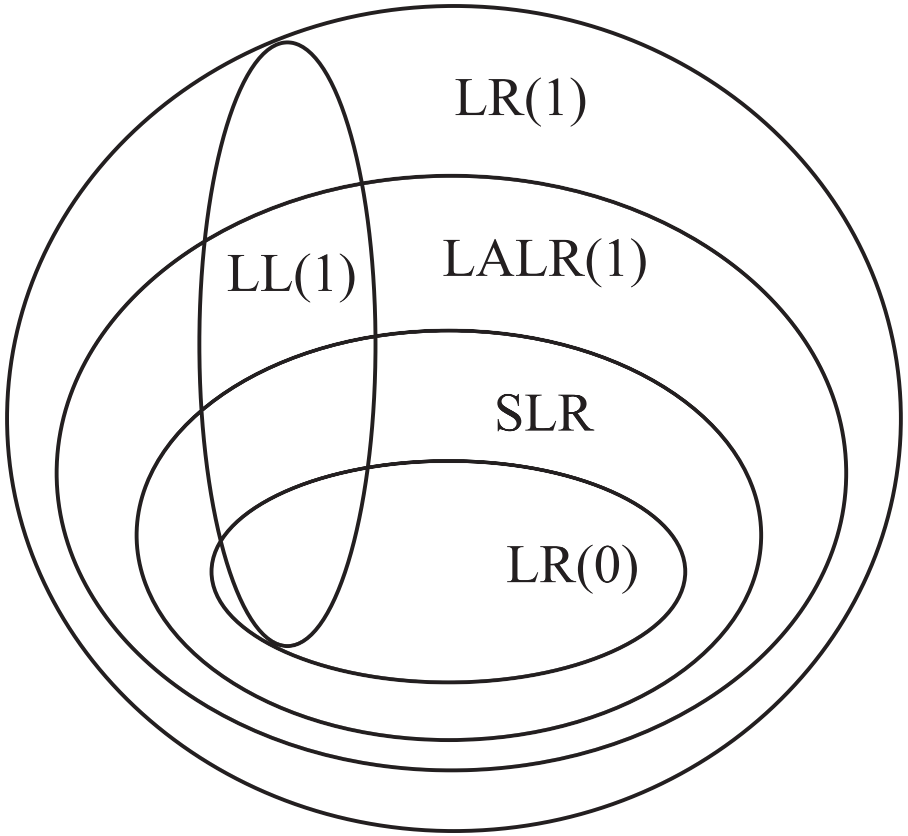
- LR(0):
  - Items ignore prediction
  - Reduce can be applied only in singular states (contain one item)
  - Lot of conflicts
- SLR:
  - Use same items as LR(0)
  - When reduce consider prediction
  - Eliminate several LR(0) conflicts (not all)
- LR(1):
  - Performant algorithm for set of states
  -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

Quiz time

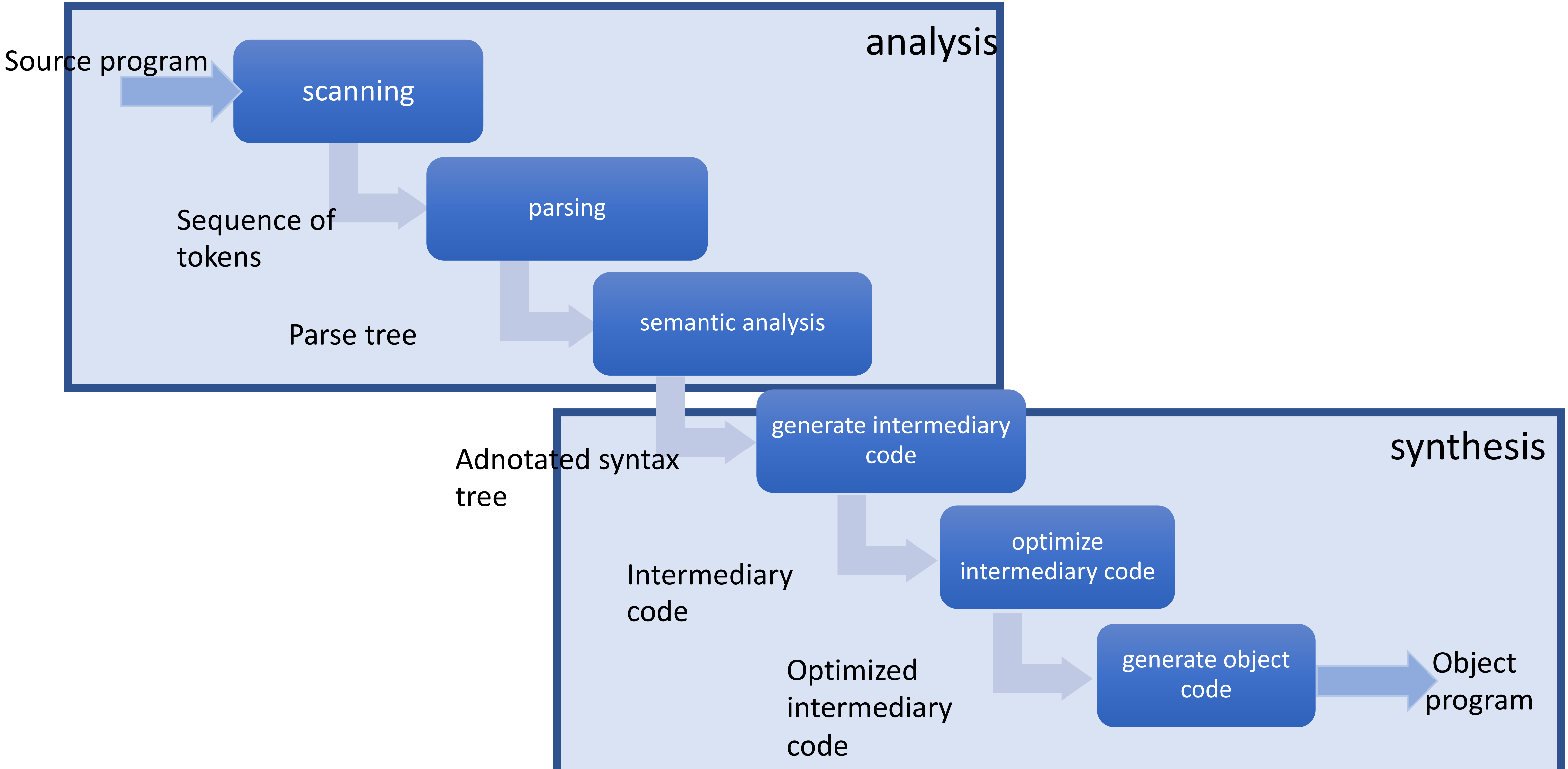
Parsing - recap

	Descendent	Ascendent
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

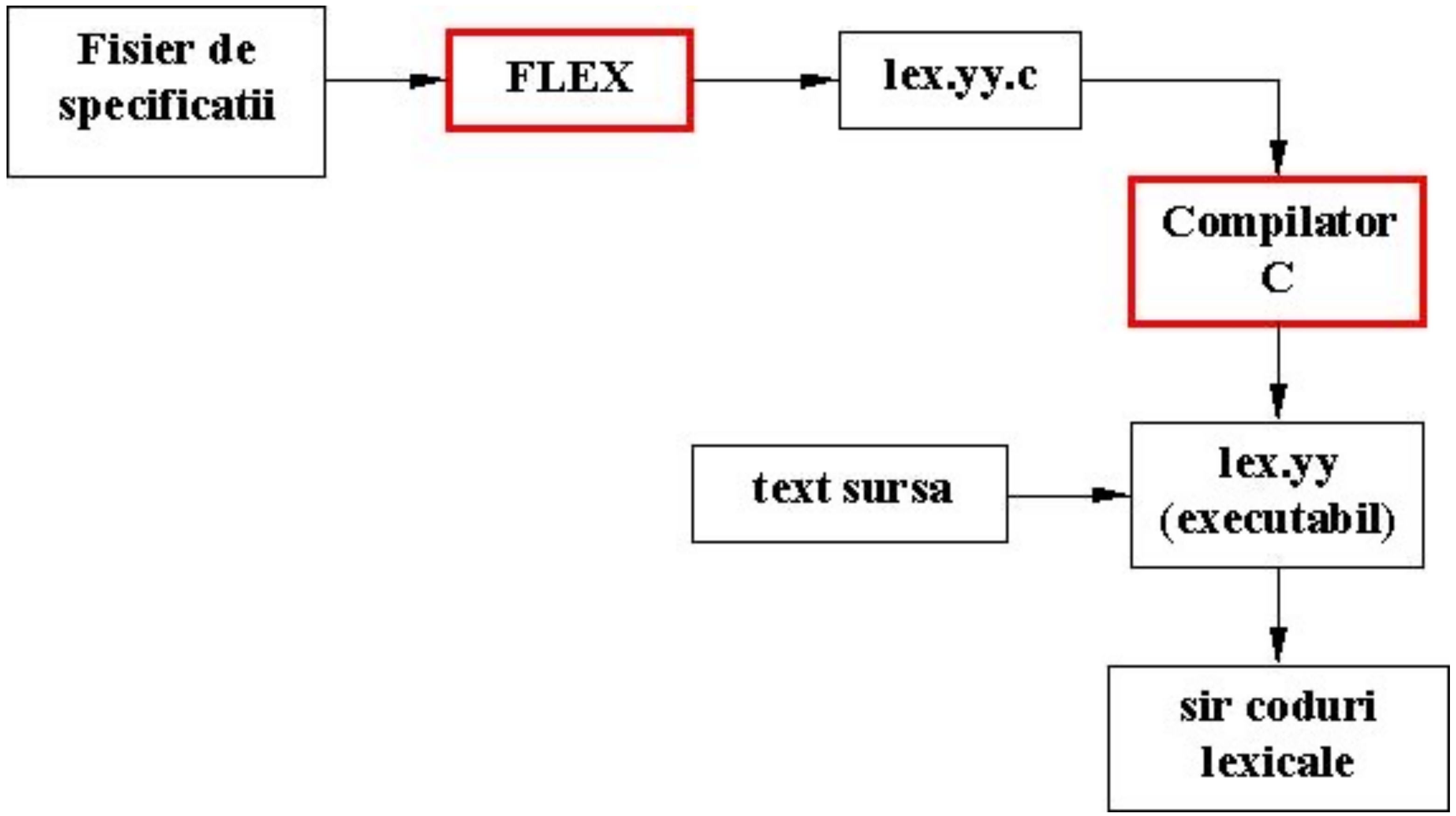
1. Have you heard about these tools?

2. Have you used any of them?

Scanning & Parsing Tools

- Scanning => lex
- Parsing => yacc

Lex – Unix utilitary (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_lex

$ your_lex<input.txt

options: http://dinosaur.compilertools.net/flex/manpage.html

Example

yacc

Parsing (syntax analysis) modeled with cfg:

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

- N – nonterminal: syntactical constructions: declaration, statement, expression, a.s.o.
- $\boldsymbol{\Sigma}$ – 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 <=> w ∊ 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
     ;
%%
```

grammar semantics

- 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 $M = (Q,\boldsymbol{\Sigma},\boldsymbol{\Gamma},\boldsymbol{\delta},q_0,Z_0,F)$ where:
  - Q – finite set of states
  - $\boldsymbol{\Sigma}$ - alphabet (finite set of input symbols)
  - $\boldsymbol{\Gamma}$ – stack alphabet (finite set of stack symbols)
  - $\boldsymbol{\delta} : Q \times (\boldsymbol{\Sigma} \cup \{\boldsymbol{\varepsilon}\}) \times \boldsymbol{\Gamma} \to \mathcal{P}(Q \times \boldsymbol{\Gamma}^*)$ –transition function
  - $q_0 \in Q$ – initial state
  - $Z_0 \in \boldsymbol{\Gamma}$ – 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:

$$(\mathbf{q}, \mathbf{x}, \boldsymbol{\alpha}) \in Q \times \boldsymbol{\Sigma}^* \times \boldsymbol{\Gamma}^*$$

where:

- PDA is in state **q**
- Input band contains **x**
- Head of stack is $\boldsymbol{\alpha}$

- Initial configuration (q_0, w, Z_0)

Configurations and moves(cont.)

- Moves between configurations:

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

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

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

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

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

- $\overset{k}{\vdash}$, $\overset{+}{\vdash}$, $\overset{*}{\vdash}$

Language accepted by PDA

- Empty stack principle:

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

- Final state principle:

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

Representations

- Enumerate
- Table
- Graphic

Construct PDA

- L = $\{0^n1^n \mid n \geq 1\}$
- States, stack, moves?

1. States:
   - Initial state:q_0 – beginning and process symbols ‘0’
   - When first symbol ‘1’ is found – move to new state => 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)

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

$\boldsymbol{\delta}(q_0,0,Z_0) = (q_0,XZ_0)$

$\boldsymbol{\delta}(q_0,0,X) = (q_0,XX)$

$\boldsymbol{\delta}(q_0,1,X) = (q_1,\boldsymbol{\varepsilon})$

$\boldsymbol{\delta}(q_1,1,X) = (q_1,\boldsymbol{\varepsilon})$

$\cancel{\boldsymbol{\delta}(q_1,\boldsymbol{\varepsilon},Z_0) = (q_2,Z_0)}$

$\boxed{\boldsymbol{\delta}(q_1,\boldsymbol{\varepsilon},Z_0) = (q_1, \boldsymbol{\varepsilon})}$

Empty stack

$\vdash (q_1, \boldsymbol{\varepsilon}, \boldsymbol{\varepsilon})$

$(q_0,0011,Z_0) \vdash (q_0,011,XZ_0) \vdash (q_0,11,XXZ_0) \vdash (q_1,1,XZ_0) \vdash (q_1, \boldsymbol{\varepsilon}, Z_0) \vdash (q_2, \boldsymbol{\varepsilon}, Z_0)$

Final state

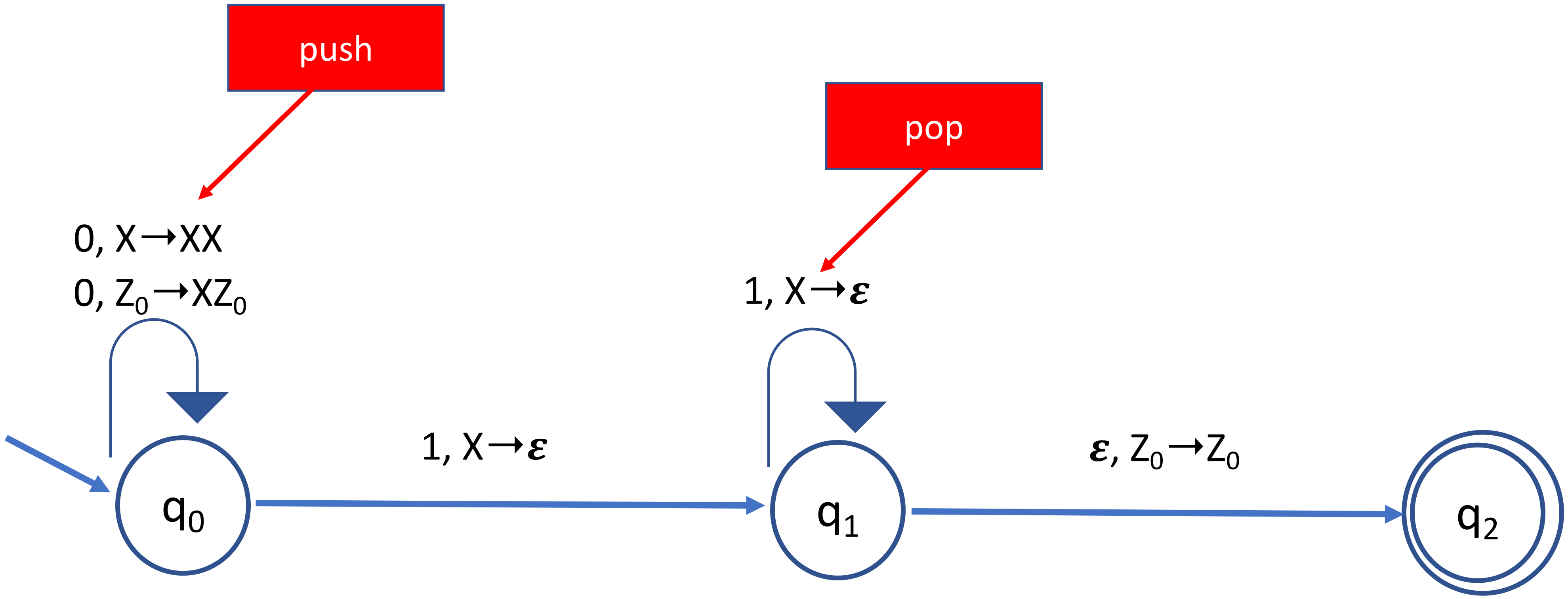
Exemple 1 (table)

		0	1	ε
q_0	Z_0	q_0,XZ_0		
	X	q_0,XX	q_1,ε	
q_1	Z_0			q_2,Z_0 (q_1, ε)
	X		q_1,ε	
q_2	Z_0			
	X			

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

(q0,0011,Z0) |- (q0,011,XZ0) |- (q0,11,XXZ0) |- (q1,1,XZ0)
|- (q1, ε,Z0) |-(q1, ε, ε) 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

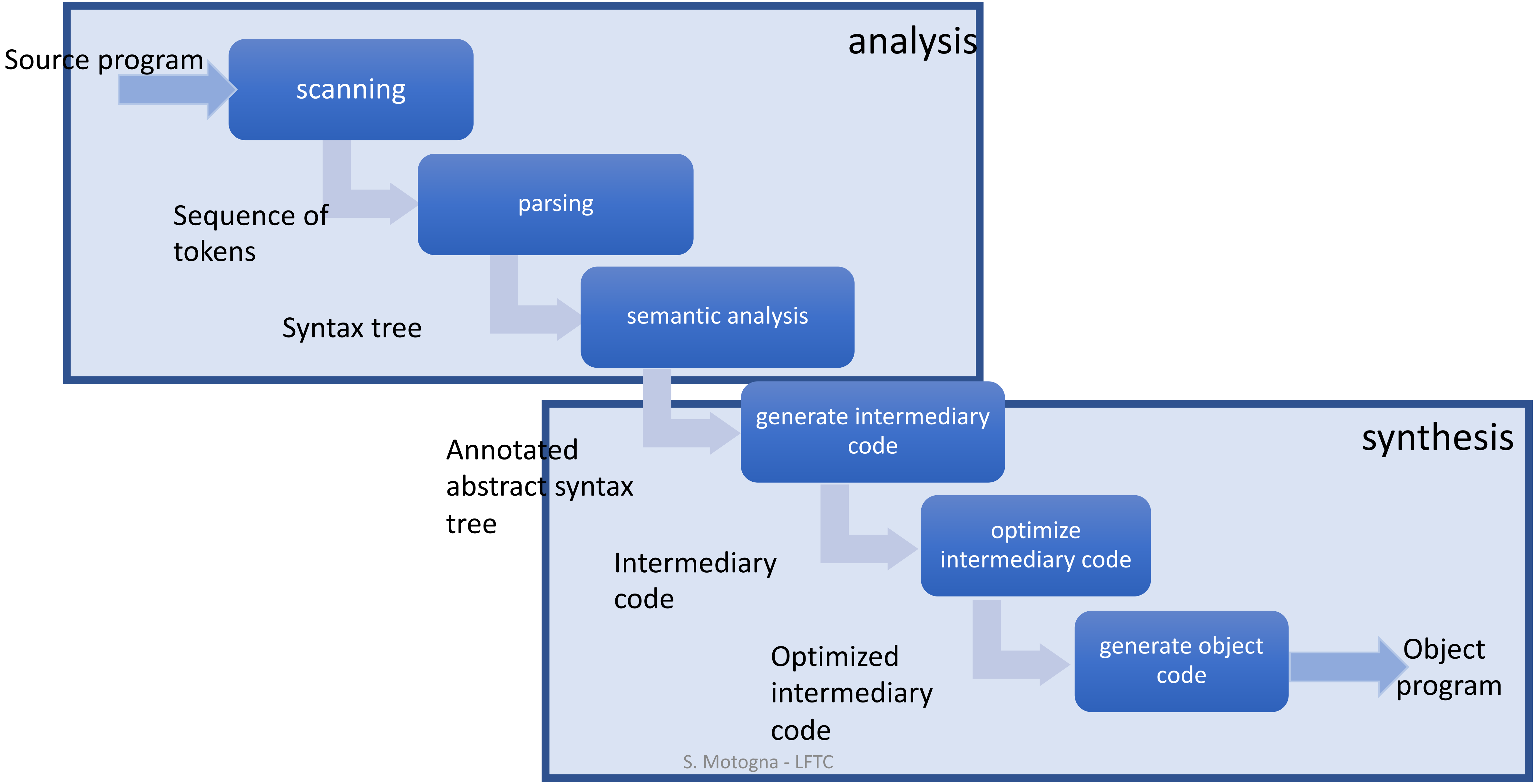
$$L(G) = L_{\varepsilon}(M)$$

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

Example 1

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

P: $N_1 \rightarrow N_2 B$

$N \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

$N_1.v = 2 * N_2.v + B.v$

$N.v = B.v$

$B.v = 0$

$B.v = 1$

Attribute – value of number = **v**

- **Synthetized 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: synthetized 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

$\text{ListId}_2\text{.type} = \text{ListId}_1\text{.type}$

$\text{Id.type} = \text{ListId}_1\text{.type}$

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

$ListDecl_1.dim = ListDecl_2.dim + Decl.dim$

$ListDecl.dim = Decl.dim$

$Decl.dim = Type.dim * ListId.no$

$Type.dim = 4$

$Type.dim = 8$

$ListId.no = 1$

$ListId_1.no = ListId_2.no + 1$

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

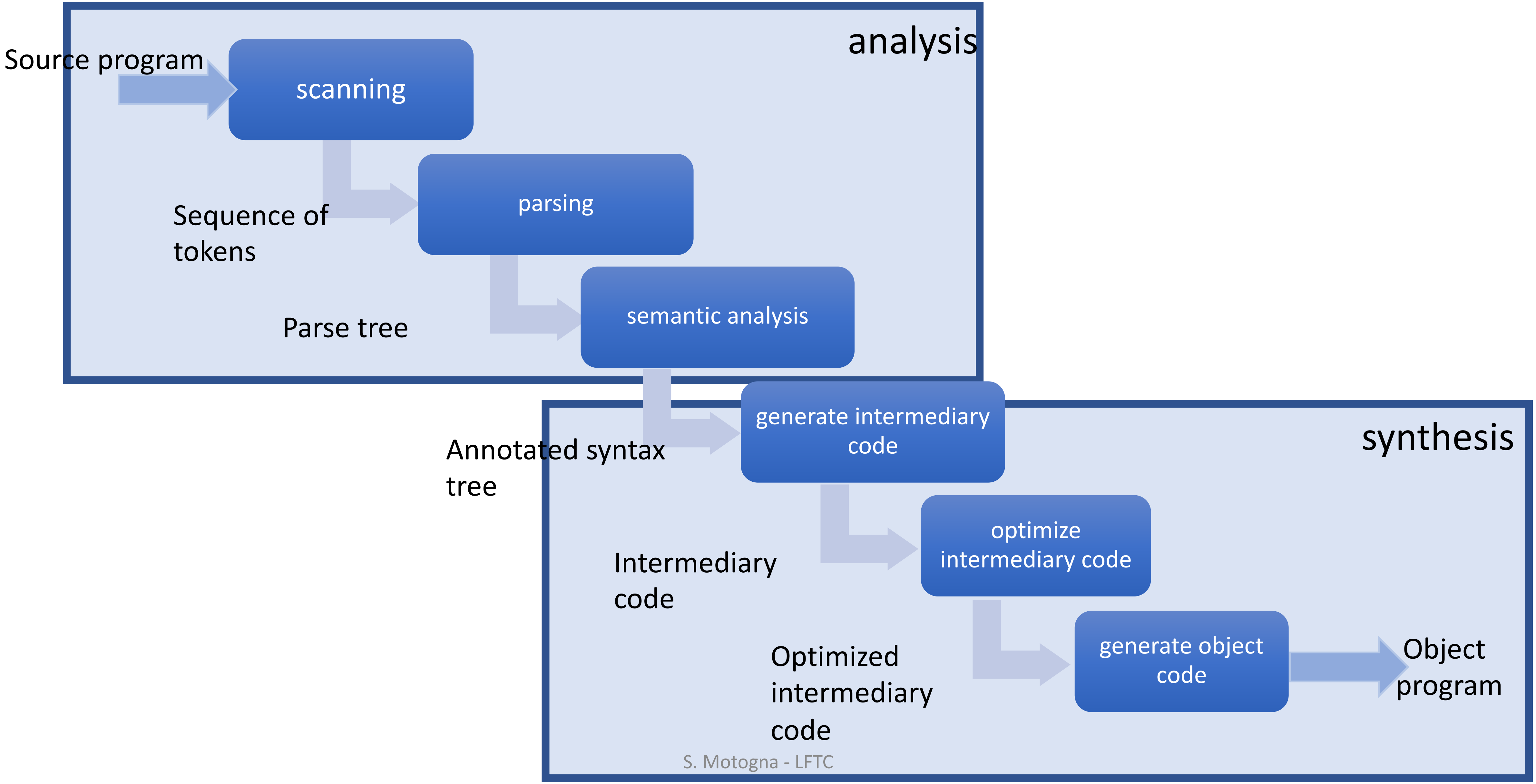
Manual methods

- Symbolic execution
  - Using control flow graph, simulate on stack how the program will behave
  - [Grune – Modern Compiler Design]

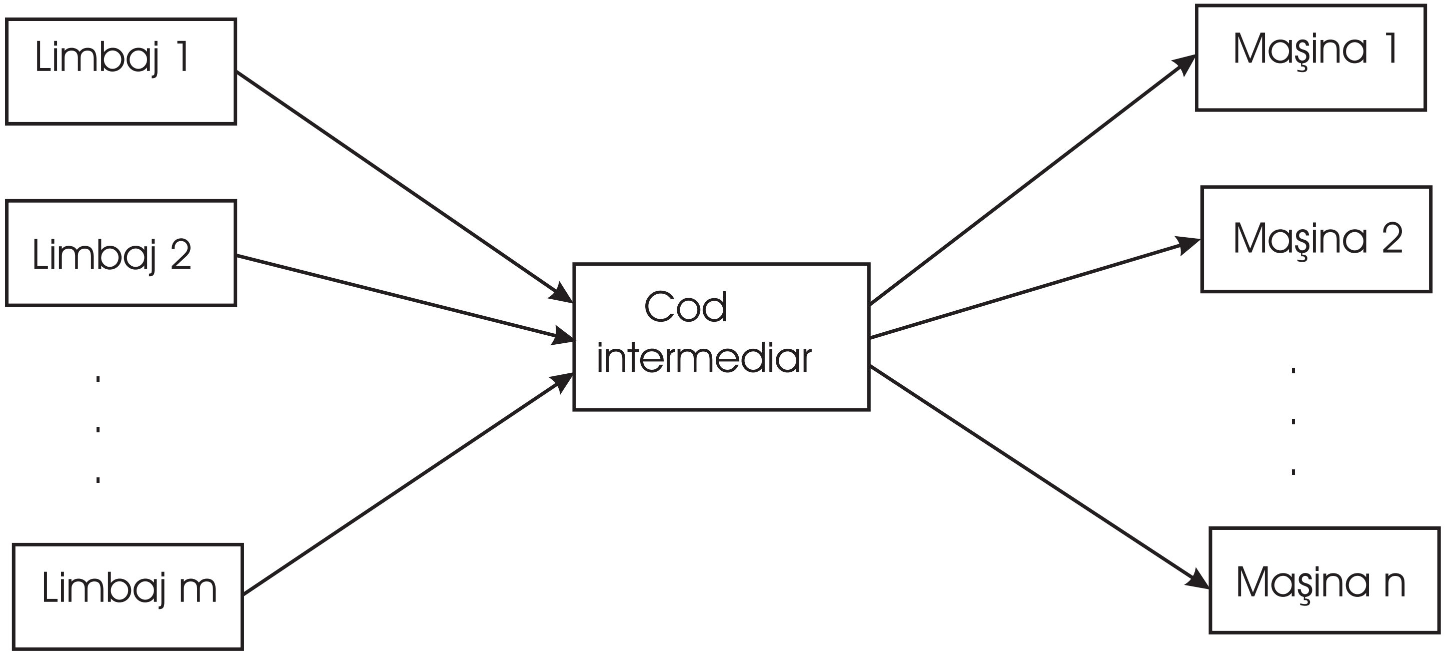
- Data flow equations
  - Data flow – associate equations based on data consumed in each node (statement) of the control flow graph: In, Out, Generated, Killed
  - [Grune – Modern Compiler Design], [Kildall], [course]

Course 12

Structure of compiler



Generate intermediary code



Forms of intermediary code

- Java bytecode – source language: Java
  – machine language (dif. platforms) JVM
- MSIL (Microsoft Intermediate Language)
  – source language: C#, VB, etc.
  – machine language (dif. platforms) Windows
- 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

Exp = a + b * c	ppf = abc*+
Exp = a * b + c	ppf = ab*c+
Exp = a * (b + c)	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 2nd 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 redudant 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?

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

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

Challenge

```
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
```

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

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

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

3 solutions

P(x) = a[n]*x^n+ … + a[1]*x + a[0] = (a[n]*x^(n-1)+ … + 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;...}