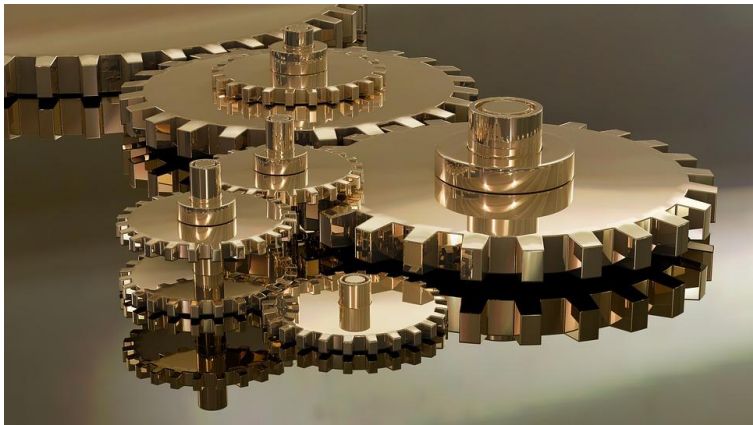
Limbaje formale și tehnici de compilare

Curs 1: introducere; compilatoare și translatoare; fazele unui compilator

Domeniul LFTC

- Limbajul de programare (LP) este unealta principală a unui programator. La materiile de programare și algoritmi se învață folosirea LP, iar la LFTC se învață alcătuirea și funcționarea sa internă.
- LFTC se ocupă cu algoritmii și teoria pentru definirea și implementarea LP și, la modul general, pentru procesarea de text structurat: compilatoare, translatoare, interpretoare, verificare formală, validare de date, data mining, mașini virtuale
- LFTC face posibilă învățarea mult mai rapidă și mai aprofundată a limbajelor de programare
- Cunoașterea modului de lucru a unui compilator permite scrierea de programe care să maximizeze folosirea resurselor disponibile (CPU, memorie, …)



Algoritmii de la LFTC fac posibile:

- Citirea fișierelor sau a altor date de intrare scrise în formate mult mai complexe decât cele accesibile prin folosirea expresiilor regulate și pentru care nu există biblioteci potrivite cu aplicația cerută
- Validarea datelor de intrare
- Extragerea informațiilor din datele de intrare
- Scrierea unor LP pentru un domeniu specific (DSL – Domain Specific Language) sau a unor LP generale
- Implementare suport de scripting în aplicații
- Verificarea formală a programelor
- Dezvoltarea unor aplicații de procesare a limbajului natural, de exemplu pentru traducere automată sau pentru agenți artificiali

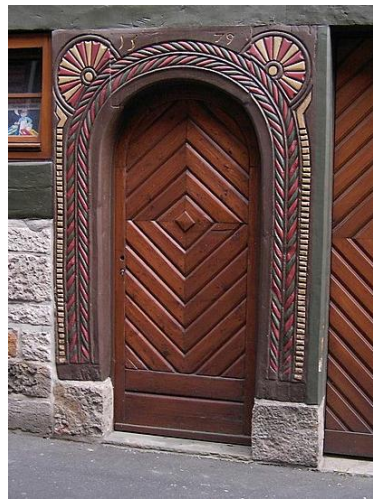
Limbaje de programare (1)

- Orice act de comunicare se bazează pe existența unui **limbaj** înțeles de ambele părți implicate în comunicare
- Limbajele pot fi de multe feluri: bazate pe cuvinte vorbite sau scrise, nonverbale, simbolice, …
- Comunicarea în limbajele uzuale în anumite situații este ambiguă și atunci este necesar să se facă apel la context pentru a se stabili sensul ei
- Și pentru comunicarea cu calculatorul este necesar să existe un limbaj comun: **limbajul de programare**

Ana vorbea cu Maria.
Ea era binedispusă.

Ești un erou!

Tocul este solid.

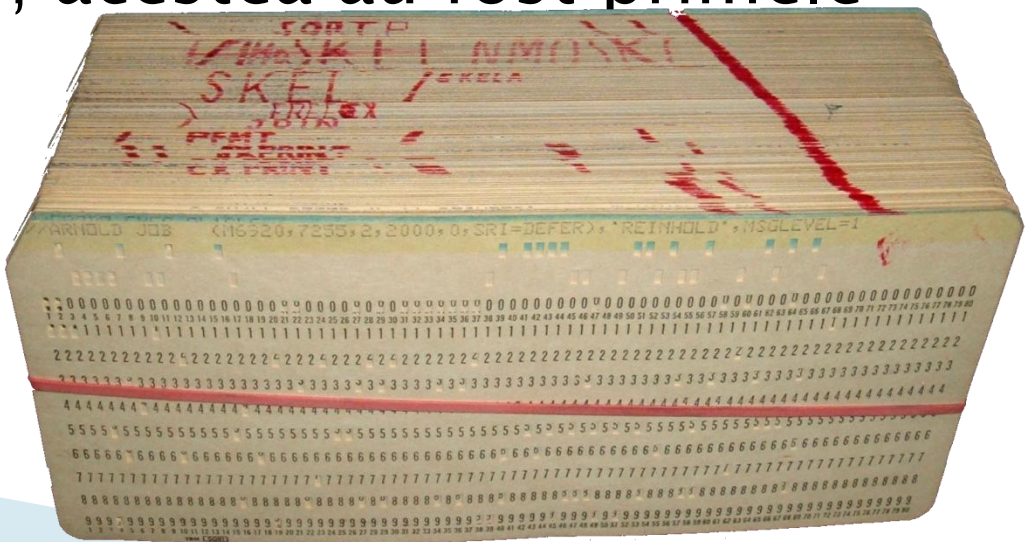


Limbaje de programare (2)

- Spre deosebire de limbajele obișnuite, un limbaj de programare (LP) trebuie să îndeplinească unele condiții:
  - **Să aibă o sintaxă complet definită** – alcătuirea și ordinea cuvintelor, folosirea semnelor de punctuație sau a operatorilor sunt specificate strict
  - **Să nu fie ambiguu** – orice comunicare prin LP trebuie să aibă strict o singură semnificație (semantică)

Compilatorul (1)

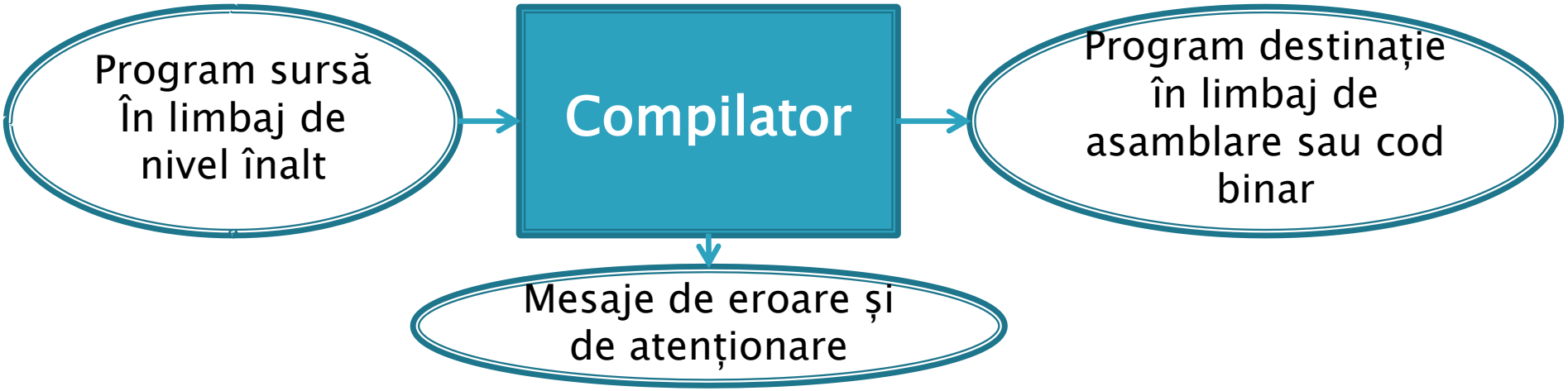
- Un CPU știe să interpreteze instrucțiuni în cod binar
- La început, deoarece nu existau limbaje de programare, primele calculatoare trebuiau programate folosind doar succesiuni de 0 și 1, ceea ce era foarte complicat
- Ulterior au început să apară programe al căror rol era să convertească un program dintr-o reprezentare mai accesibilă omului în codul binar necesar calculatoarelor; acestea au fost primele compilatoare

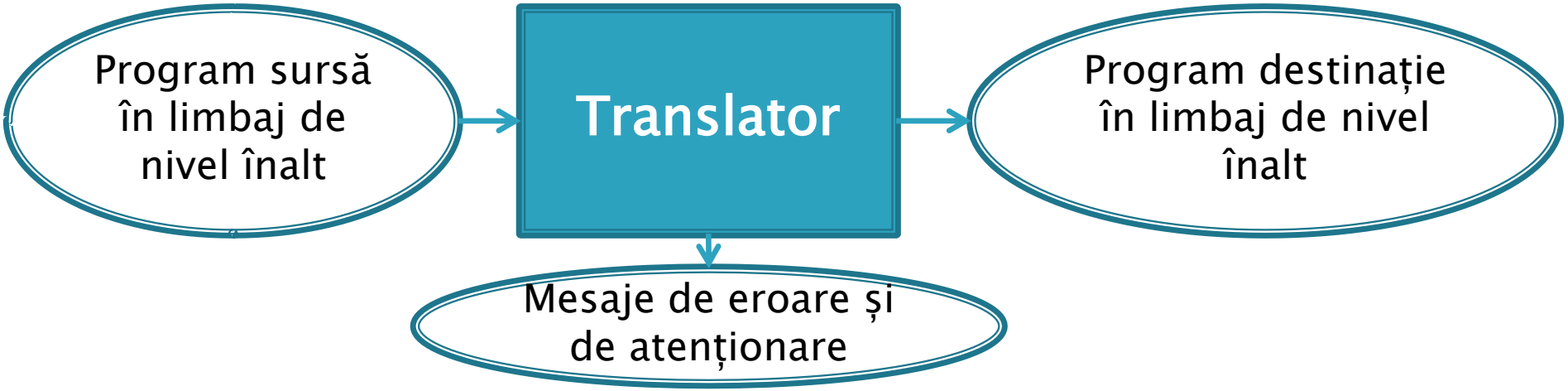


Compilatorul (2)

- Funcția de bază a unui compilator este să translateze un **program sursă**, scris într-un limbaj de nivel înalt (mai apropiat de reprezentarea umană), într-o reprezentare echivalentă (**program destinație**), accesibil unui CPU
- Ieșirea compilatorului poate fi un program în limbaj de asamblare (care ulterior va fi transformat în cod binar) sau se va genera direct cod binar
- Pe lângă această funcție de bază, un compilator mai trebuie să fie capabil să verifice dacă programul de compilat este corect (și dacă nu este, să emită mesaje de eroare sau atenționări)
- Unele compilatoare mai pot avea funcții de optimizare, pentru a genera variante mai eficiente ca timp de execuție sau ca necesar de memorie

Compilatorul (3)





- **Translatorul** transformă programul sursă într-un program destinație, care este tot într-un limbaj de nivel înalt
- Primele compilatoare de C++ au fost de fapt translatoare, care translatau din C++ în C, iar apoi se foloseau compilatoare obișnuite de C

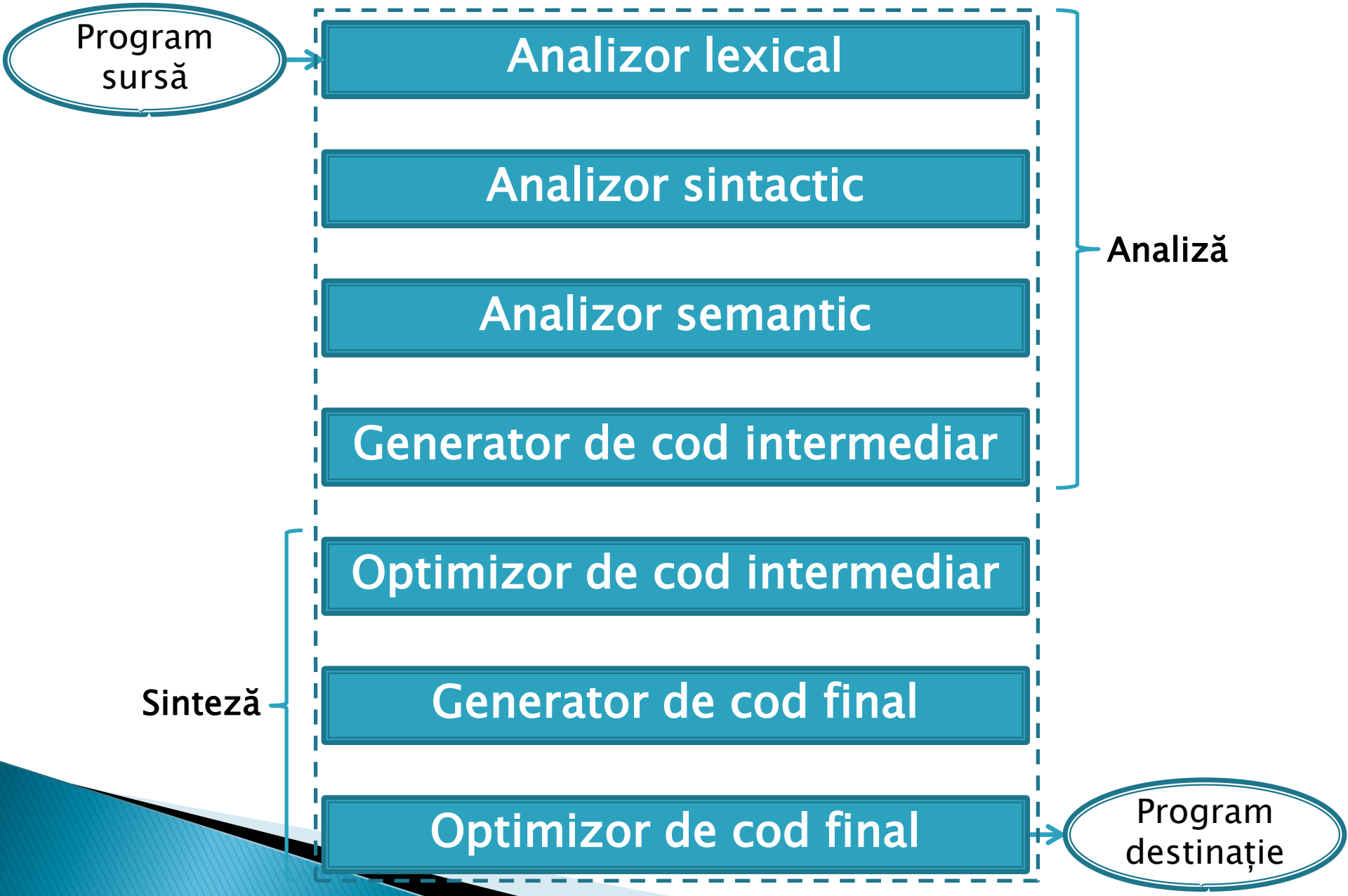
Compilatorul (4)

- Primul compilator a fost realizat în anul 1957, pentru limbajul de programare Fortran; dezvoltarea sa a durat 18 luni, deoarece nu exista o teorie a limbajelor formale și totodată a trebuit scris în limbaj de asamblare
- Dezvoltarea ulterioară a compilatoarelor a fost un proces iterativ, pe mai multe planuri:
  - Prin dezvoltarea părții teoretice s-a putut automatiza și optimiza o bună parte din procesul de dezvoltare
  - Având la dispoziție un limbaj de programare mai puternic, a fost mai ușor să se scrie programe mai complexe
- Au apărut compilatoare pentru limbaje de programare din ce în ce mai complexe: Pascal, C, Ada, C++, Java, C#

Modelul analiză-sinteză al compilării

- În principiu, un compilator are două mari funcții:
  - Analiza programului sursă (front-end)
  - Sinteza programului destinație (back-end)
- **Analiza programului sursă** – împarte programul sursă în componentele sale de bază și, pentru compilatoarele mai avansate, creează o reprezentare intermediară a programului sursă. Uneori prelucrarea codului intermediar se consideră ca fiind o funcție separată: **middle-end**
- **Sinteza programului destinație** – construiește programul destinație direct din informațiile de la faza de analiză sau pornind de la reprezentarea intermediară

Fazele unui compilator



Analizorul lexical

- Grupează caracterele de intrare în atomi lexicali
- Reține doar atomii lexicali utili
- Asociază atomilor lexicali informații conexe: linia din fișierul de intrare, caracterele constituente, valoarea numerică

```
// valoarea absolută
int abs(int v){
    if(a<0)return -a;
    return a;
    }
```

```
ID ::= [a-zA-Z_] [a-zA-Z0-9_]*
LACC ::= {
…
```

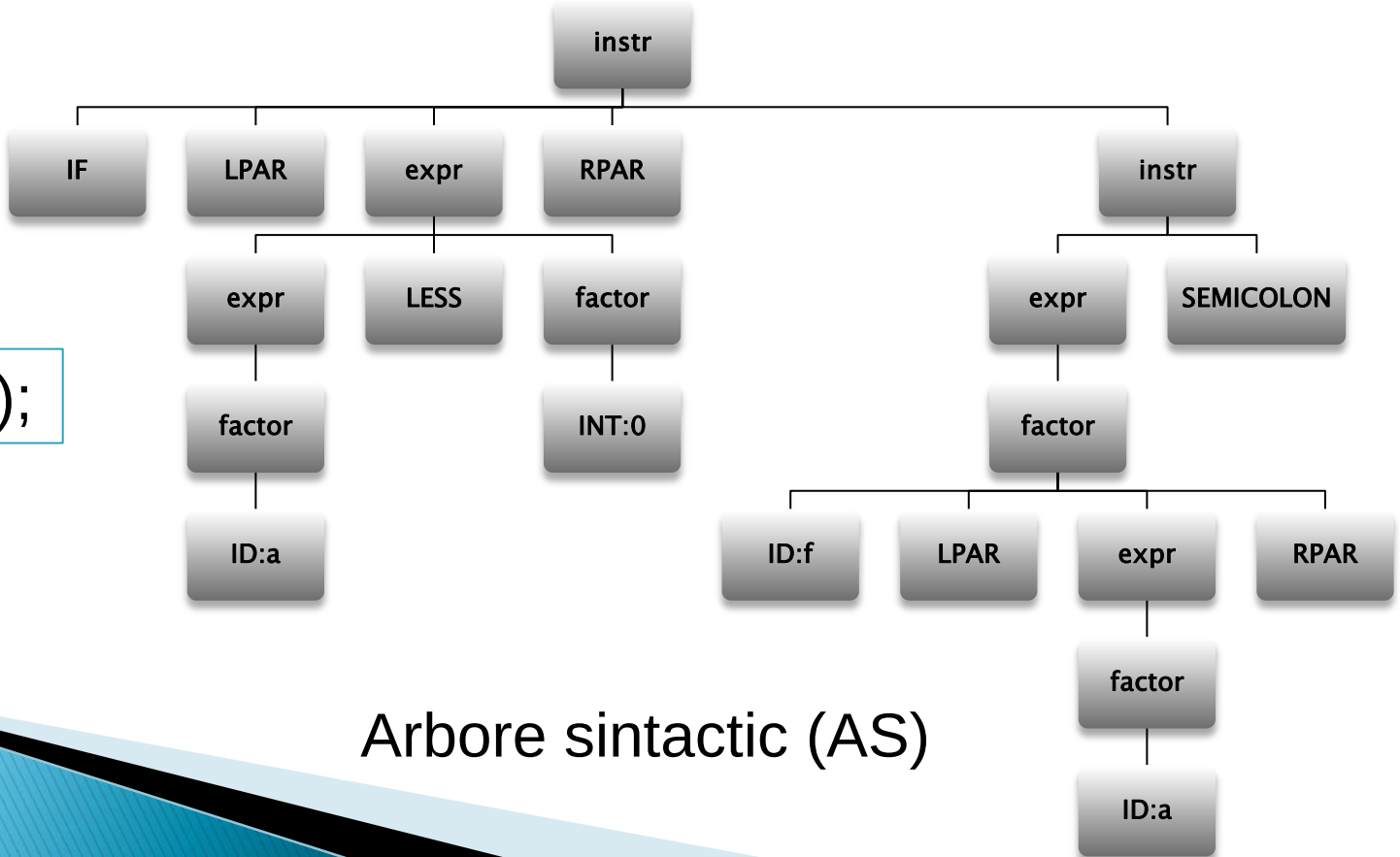
```
TYPE_INT ID:abs LPAR TYPE_INT ID:v RPAR LACC IF LPAR
ID:a LESS INT:0 RPAR RETURN SUB ID:a SEMICOLON
RETURN ID:A SEMICOLON RACC
```

Analizorul sintactic

- Grupează atomii lexicali în secvențe specifice limbajului: expresii, declarații, instrucțiuni, …

```
instr ::= IF LPAR expr RPAR instr ( ELSE instr )? | expr SEMICOLON
expr ::= expr LESS factor | factor
factor ::= ID ( LPAR ( expr ( COMMA expr )* )? RPAR )? | INT
```

```
if(a<0)f(a);
```



Arbore sintactic (AS)

Analiza semantică

- Testează dacă programul respectă **regulile semantice**
- În general se compune din:
  - **Analiză de domeniu** – memorează simbolurile din declarații și verifică să nu existe redefiniri de simboluri
  - **Analiză de tip** – testează dacă într-o expresie componentele sunt folosite conform tipurilor lor și deduce tipul expresiei
- Se bazează pe **tabela de simboluri**

```
// analiză de domeniu
int a;
int a;  // eroare: a este redefinit
```

```
// analiză de tip
a(5);  // eroare: un întreg nu poate fi apelat ca o funcție
printf("%g",7.8+a);  // double + int => double
```

Generatorul de cod intermediar

Codul intermediar permite unui compilator să utilizeze același procesor de cod și back-end-uri pentru oricâte front-end-uri


```
for(i=0;i<n;i++)
    printf("%d",i*2);
```

```
        i=0
L1:     T1=i<n
        if not T1 goto L2
        T2=i*2
        printf("%d",T2)
        i=i+1
        goto L1
L2:
```

Optimizorul de cod intermediar

Optimizarea codului implică aplicarea unor transformări care măresc eficiența codului din diverse puncte de vedere (timp de execuție, necesar de memorie, …), dar îi păstrează neschimbată semantica

```
for(i=0;s[i]!=0;i++)
    if(islower(s[i]))s[i]+='A'-'a';
```

```
        i=0
L1:     T1=s[i]
        T2=T1!=0
        if not T2 goto L3
        T3=s[i]
        T4=islower(T3)
        if not T4 goto L2
        T5= 'A'-'a'
        T6=s[i]
        T7=T6+T5
        s[i]=T7
L2:     i=i+1
        goto L1
L3:
```

```
        i=0
L1:     T1=s[i]
        if not T1 goto L3
        T4=islower(T1)
        if not T4 goto L2
        T7=T1-32
        s[i]=T7
L2:     i=i+1
        goto L1
L3:
```

Generatorul de cod final

Este generat codul programului destinație. Generarea se poate face în limbaj de asamblare sau direct în cod binar.

```
        i=0
L1:     T1=s[i]
        if not T1 goto L3
        T4=islower(T1)
        if not T4 goto L2
        T7=T1-32
        s[i]=T7
L2:     i=i+1
        goto L1
L3:
```

```
        mov edi,0
L1:     mov bl,[s+edi]
        cmp bl,0
        jz L3
        movsx ecx,bl
        push ecx
        call islower
        add esp,4
        cmp eax,0
        jz L2
        sub bl,32
        mov [s+edi],bl
L2:     inc edi
        jmp L1
L3:
```

Optimizorul de cod final

Aplică pe codul final optimizări care sunt specifice unei anumite platforme și, din acest motiv, nu se pot include în optimizările generale de cod intermediar

```
do{...}while(--i>0);
```

```
L1:     ...
        dec edi
        cmp edi,0
        jnz L1
```

```
L1:     ...
        dec edi
        jnz L1
```

cmp edi,0 *nu este necesară, deoarece* dec edi *setează deja EFLAGS cu valorile corecte*

Unelte pentru dezvoltarea compilatoarelor

- **ANTLR** – generator de analizoare lexicale și sintactice pentru mai multe limbaje de programare
- **Bison** – generator de analizoare sintactice pentru C/C++/Java. Este considerat ca un succesor a lui Yacc.
- **Flex** – generator de analizoare lexicale pentru C/C++
- **LLVM** – o reprezentare intermediară de cod, cu unele dintre cele mai bune optimizări și multe back-end-uri. Se poate folosi din mai multe limbaje de programare.
- **re2c** – generator de analizoare lexicale pentru C/C++
- **Yacc** – generator de analizoare sintactice pentru C/C++
- Există asemenea unelte pentru multe limbaje de programare: Python, PHP, Ruby, OCaml, Haskell, …
- Mai pot fi utile programe/biblioteci pentru: expresii regulate, generare cod mașină din asembler, …

Bibliografie

- **”Compilers – Principles, Techniques, & Tools”**, Aho A.V., Lam M.S., Sethi R., Ullman J.D., 2nd edition, 2007
- **”Limbaje formale și translatoare”**, Ciocârlie H., 2017
- **”Limbaje formale și tehnici de compilare – laboratoare pentru programele de studii Calculatoare și Informatică”**, Aciu R.
- **”Engineering a Compiler”**, Cooper K.D., Linda T., 2nd edition, 2012