

# Introducció a la compilació

Jordi Petit



## Objectius

- Conèixer l'estruccional general d'un compilador, les seves principals etapes, i la seva organització.
- Conèixer l'existència d'eines per ajudar a crear compiladors (usarem [ANTLR](#)).
- A la pràctica, ens limitem a crear petits processadors de llenguatges:
  1. Definició del vocabulari,
  2. Definició de la gramàtica,
  3. Generació de l'arbre de sintaxi abstracta,
  4. Interpretació a través del recorregut de l'arbre.
- El curs de Compiladors aprofundeix molts més els continguts.
- El curs de Teoria de la Computació n'ofereix els fonaments teòrics.

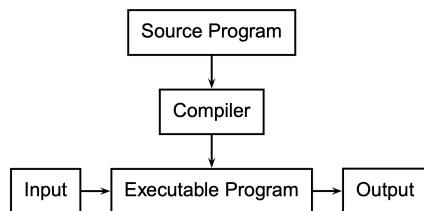
# Crèdits

Gran part del material d'aquestes diapositives sobre compilació s'ha extret de les que va elaborar el professor **Stephen A. Edwards** (Universitat de Columbia) per l'assignatura COMS W4115 (Programming Languages and Translators) i que el professor **Jordi Cortadella** (UPC) va adaptar per l'assignatura de Compiladors. També s'ha extret material de les transparències del professor **Fernando Orejas** (UPC).

# Visió general

## Processadors de llenguatges

### Compiladors



Un **compilador** és un programa que tradueix programes escrits en un LP d'alt nivell a codi màquina (o, en general, a codi de baix nivell).

Exemples: GCC, CLANG, go, ghc, ...

# Processadors de llenguatges

## Compiladors

Compilació en C

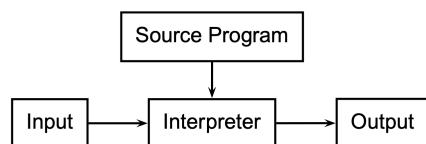
```
int func(int a, int b) {  
    a = a + b;  
    return a;  
}
```

```
gcc -S prova.c
```

```
_func:  
    pushq  %rbp  
    movq    %rsp, %rbp  
    movl    %edi, -4(%rbp)  
    movl    %esi, -8(%rbp)  
    movl    -4(%rbp), %eax  
    addl    -8(%rbp), %eax  
    movl    %eax, -4(%rbp)  
    movl    -4(%rbp), %eax  
    popq    %rbp  
    retq
```

# Processadors de llenguatges

## Intèrprets



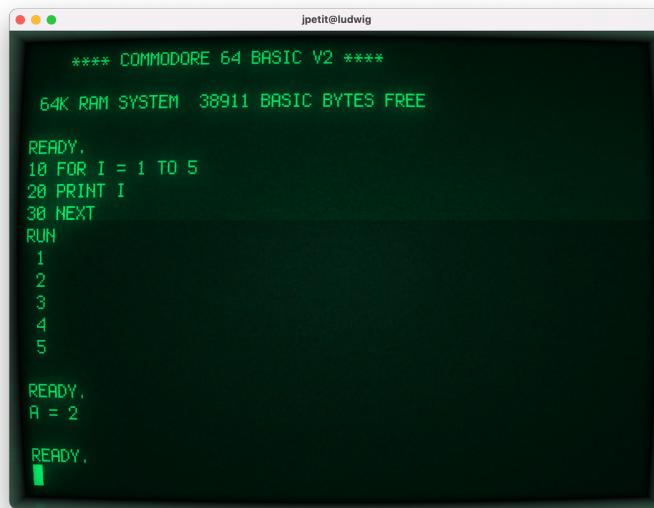
Un **intèpret** és un programa que executa directament instruccions escrites en un LP.

Exemples: Python, PHP, Perl, ghci, BASIC, Logo...

# Processadors de llenguatges

## Intèrprets

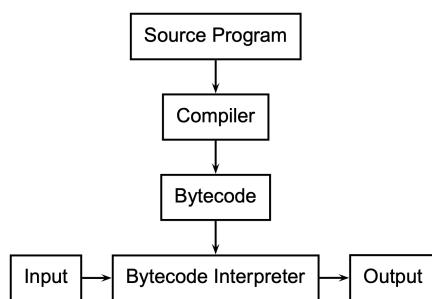
Sessió amb l'intèrpret de BASIC al Commodore 64 (emulat al Mac 🐹!)



```
***** COMMODORE 64 BASIC V2 *****  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
  
READY.  
10 FOR I = 1 TO 5  
20 PRINT I  
30 NEXT  
RUN  
1  
2  
3  
4  
5  
  
READY.  
R = 2  
  
READY.
```

# Processadors de llenguatges

## Intèrprets de bytecode



Variant entre els compiladors i els intèrprets.

- El **bytecode** és un codi intermedi més abstracte que el codi màquina.
- Augmenta la portabilitat i seguretat i facilita la interpretació.
- Una **màquina virtual** interpreta programes en bytecode.

Exemples: Java, Python, ...

# Processadors de llenguatges

Intèrprets de bytecode: CPython per a Python

```
>>> import dis # desensamblador
>>> dis.dis("a = a + b")
  1  0 LOAD_NAME      0 (a)
  2  LOAD_NAME      1 (b)
  4  BINARY_ADD
  6  STORE_NAME     0 (a)
  8  LOAD_CONST    0 (None)
 10 RETURN_VALUE
```

La VM de CPython usa tres tipus de piles:

- **Call stack:** Guarda l'estructura principal de l'execució d'un programa en Python.

Hi ha *frame* per a cada crida oberta a una funció. Cada crida a una funció empila un nou *frame* i cada sortida de funció el desempila. És on es desen les variables locals.

- A cada *frame*, hi ha un **evaluation stack**: És on es fa l'avaluació de les expressions, ficant paràmetres i extraient resultats.
- A cada *frame*, també hi ha un **block stack**: És on es realitza l'execució de les instruccions (condicionals, bucles, try/excepts, withs, continues, breaks, ...)

# Processadors de llenguatges

Intèrprets de bytecode: JVM per a Java

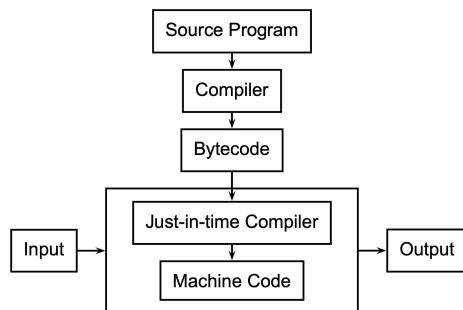
```
public static void func(int a, int b) {
    a = a + b;
}
```

```
javap -v prova.class
```

```
public static void func(int, int);
  descriptor: (II)V
  flags: (0x0009) ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=2, args_size=2
      0: iload_0
      1: iload_1
      2: iadd
      3: istore_0
      4: return
```

# Processadors de llenguatges

Compiladors *just-in-time*



La compilació **just-in-time** (JIT) compila fragments del programa durant la seva execució.

Un analitzador inspecciona el codi executat per veure quan val la pena compilar-lo.

Exemples: Julia, V8 per Javascript, JVM per Java, ...

# Processadors de llenguatges

Compiladors *just-in-time*

**Numba** tradueix funcions en Python a codi màquina i optimitzat usant LLVM en temps d'execució.

```
import numba
import random

@numba.jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

# Processadors de llenguatges

## Preprocessadors

Un **preprocessador** prepara el codi font d'un programa abans que el compilador el vegi.

- Expansió de macros
- Inclusió de fitxers
- Compilació condicional
- Extensions de llenguatge

Exemples: cpp, m4, ...

# Processadors de llenguatges

## Preprocessadors

El preprocessador de C

```
#include <stdio.h>
#define min(x, y) ((x)<(y))?(x):(y)
#ifndef DEFINE_BAZ
int baz();
#endif
void foo() {
    int a = 1;
    int b = 2;
    int c;
    c = min(a,b);
}
```

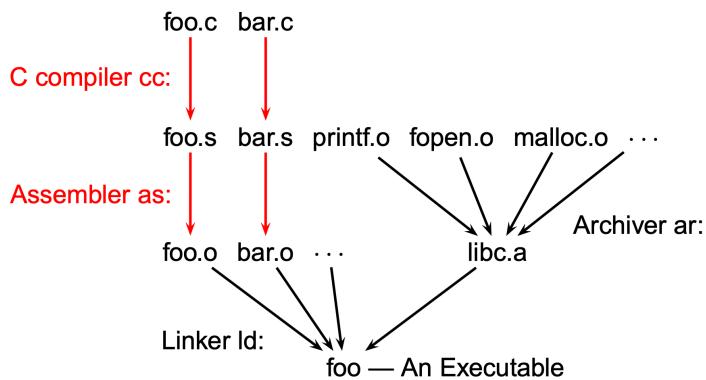
```
gcc -E programa.c
```

```
extern int printf(char*, ...);
/* moltes més línies de stdio.h
void foo() {
    int a = 1;
    int b = 2;
    int c;
    c = ((a)<(b))?(a):(b);
}
```

# Processadors de llenguatges

## Ecosistema

Els processadors de llenguatges viuen en un ecosistema gran i complex: preprocessadors, compiladors, enllaçadors, gestors de llibreries, ABIs (application binary interfaces), formats d'executables, ...



# Sintaxi

La **sintaxi** d'un llenguatge de programació és el conjunt de regles que defineixen les combinacions de símbols que es consideren construccions correctament estructurades.

Jove xef, porti whisky amb quinze glaçons d'hidrogen, coi!

➡ frase sintàcticament correcta en català, però no és un programa en Java.

```
class Foo {  
    public int j;  
    public int foo(int k) {  
        return j + k;  
    }  
}
```

➡ programa sintàcticament correcte en Java, però no és un programa en C.

# Sintaxi

Sovint s'especifica la sintaxi utilitzant una **gramàtica lliure de context** (*context-free grammar*).

Els elements més bàsics ("paraules") s'especifiquen a través d'**expressions regulars**.

Exemple per expressions algebraiques:

```
expr → NUM
      | '(' expr ')'
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr

NUM → [0-9]+ ( '.' [0-9]+ )
```

# Semàntica

La **semàntica** d'un LP descriu què significa un programa ben construit.

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

- ➡ La semàntica d'aquesta funció en Python és el càclul de l' $n$ -èsim nombre de Fibonacci.

# Semàntica

A vegades, les construccions sintàcticament correctes poden ser semànticament incorrectes.

L'arc de Sant Martí va saltar sobre el planeta pelut.

- ➡ sintàcticament correcta en català, però sense sentit.

```
class Foo {  
    int bar(int x) { return Foo; }  
}
```

- ➡ sintàcticament correcte en Java, però sense sentit.

# Semàntica

A vegades, les construccions sintàcticament correctes poden ser ambigües.

Han posat un banc a la plaça.

- ➡ sintàcticament correcta en català, però ambigüa.

```
class Bar {  
    public float foo() { return 0; }  
    public int foo() { return 0; }  
}
```

- ➡ sintàcticament correcte en Java, però ambigu.

# Semàntica

Hi ha bàsicament dues maneres d'especificar formalment la semàntica:

- **Semàntica operacional:** defineix una màquina virtual i com l'execució del programa canvia l'estat de la màquina.
- **Semàntica denotacional:** mostra com construir una funció que representa el comportament del programa (és a dir, una transformació d'entrades a sortides) a partir de les construccions del LP.

La majoria de definicions de semàntica per a LPs utilitzen una semàntica operacional descrita informalment en llenguatge natural.



# Flux de compilació

## Etapes

- Front end
  - preprocessador
  - analitzador lèxic (escàner)
  - analitzador sintàctic (parser)
  - analitzador semàntic
- Middle end
  - analitzador de codi intermedi
  - optimitzador de codi intermedi
- Back end
  - generador de codi específic
  - optimitzador de codi específic

# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

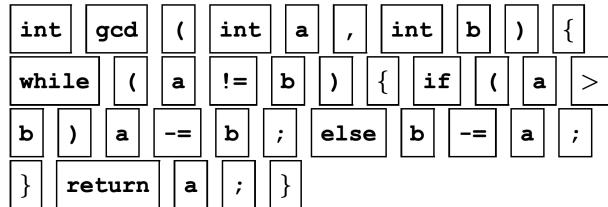
El compilador veu una seqüència de caràcters:

```
int gcd(int a, int b) { ↪
    while (a != b) { ↪
        if (a > b) a -= b; else b -= a; ↪
    } ↪
    return a; ↪
}
```

# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

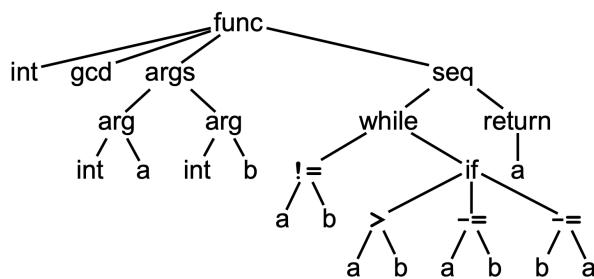
L'**analitzador lèxic (escàner)** agrupa els caràcters en "paraules" (tokens) i elimina blancs i comentaris.



# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

L'**analitzador sintàctic (parser)** construeix un **arbre de sintaxi abstracta (AST)** a partir de la seqüència de tokens i les regles sintàctiques.



Les paraules clau, els separadors, parèntesis i blocs s'eliminen.

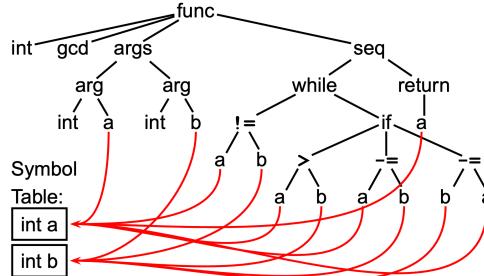
# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

L'analitzador semàntic recorre l'AST i:

- crea la **taula de símbols**,
- assigna memòria a les variables,
- comprova errors de tipus,
- resol ambigüitats.

El resultat és la taula de símbols i un AST decorat.



# Flux de compilació

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```

El **generador de codi** tradueix el programa a **codi de tres adreces** (ensamblador idealitzat amb infinitat de registres).

```
L0:    sne $1, a, b          # signed not equal
       seq $0, $1, 0          # signed equal
       btrue $0, L1            # while a != b
       sl $3, b, a             # signed less
       seq $2, $3, 0           #
       btrue $2, L4            #     if a < b
       sub a, a, b              #         a -= b
       jmp L5
L4:    sub b, b, a            #         b -= a
L5:    jmp L0
L1:    ret a                  # return a
```

# Flux de compilació

```

L0:    sne $1, a, b          # signed not equal
        seq $0, $1, 0         # signed equal
        btrue $0, L1           # while a != b
        sl $3, b, a           # signed less
        seq $2, $3, 0
        btrue $2, L4           #   if a < b
        sub a, a, b           #     a -= b
        jmp L5
L4:    sub b, b, a           #     b -= a
L5:    jmp L0
L1:    ret a                # return a

```

El **back end** tradueix i optimitza el codi de tres adreces a l'arquitectura desitjada:

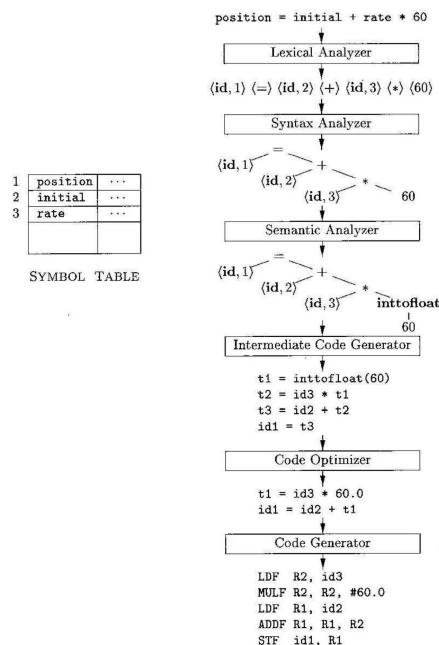
```

gcd:   pushl %ebp          # Save FP
       movl %esp,%ebp
       movl 8(%ebp),%eax      # Load a from stack
       movl 12(%ebp),%edx     # Load b from stack
.L8:   cmpl %edx,%eax
       je .L3                 # while a != b
       jle .L5                 #   if (a < b)
       subl %edx,%eax
       jmp .L8                 #     a -= b
.L5:   subl %eax,%edx
       jmp .L8                 #     b -= a
.L3:   leave                  # Restore SP, BP
       ret                     # return a

```

[asm 80386]

## Flux de compilació: sumari



# Eines

Per construir un compilador no es parteix de zero.

Hi ha moltes eines que donen suport.

Exemples:

- ANTLR, donades les especificacions lèxiques i sintàctiques del LP, construeix automàticament l'escàner, l'anàlitzador i l'AST.
- LLVM ofereix una col·lecció d'eines modulars reutilitzables pels backends dels compiladors.

## Anàlisi lèxica

# Anàlisi lèxica

L'**analitzador lèxic** (o **escàner**) converteix una seqüència de caràcters en una seqüència de **tokens**:

- identificadors,
- literals (nombres, textos, caràcters)
- paraules clau,
- operadors,
- puntuació...

`f o o = a + bar(2, q);`

ID	EQUALS	ID	PLUS	ID	LPAREN	NUM
COMMA	ID	LPAREN	SEMI			

Token	Lexemes	Pattern
EQUALS	=	an equals sign
PLUS	+	a plus sign
ID	a foo bar	letter followed by letters or digits
NUM	0 42	one or more digits

# Objectius

- Simplificar la feina de l'analitzador sintàctic.

El parser no té en compte els noms dels identificadors, només li preocuten els tokens (`supercalifragilisticexpialidocious` → ID).

- Descartar detalls irrelevants: blancs, comentaris, ...
- Els escàners són molt més ràpids que els parsers.

# Descripció de tokens

Per especificar els tokens s'utilitzen conceptes de la teoria de llenguatges:

**Alfabet:** un conjunt finit de símbols.

Exemples: {0, 1}, {A, B, ..., Z}, ASCII, Unicode, ...

**Paraula:** una seqüència finita de símbols de l'alfabet.

Exemples:  $\epsilon$  (la paraula buida), foo,  $\alpha\beta\gamma$ .

**Llenguatge:** Un conjunt de paraules sobre un alfabet.

Exemples:  $\emptyset$  (el llenguatge buit), { 1, 11, 111, 1111 }, tots els mots anglesos, tots els identificadors (textos que comencen amb una lletra seguida per lletres o díigits).

# Operacions sobre llenguatges

Exemple:  $L = \{ \epsilon, wo \}$ ,  $M = \{ man, men \}$

**Concatenació:** Una paraules d'un llenguatge seguida d'una paraula de l'altre llenguatge.

$L M = \{ man, men, woman, women \}$

**Unió:** Totes les paraules de cada llenguatge.

$L \cup M = \{ \epsilon, wo, man, men \}$

**Clausura de Kleene:** Zero o més concatenacions.

$M^* = \{ \epsilon \} \cup M \cup MM \cup MMM \cup \dots = \{ \epsilon, man, men, manman, manmen, menman, menmen, manmanman, manmanmen, manmenman, \dots \}$

# Expressions regulars

Les **expressions regulars** descriuen llenguatges a partir de tokens sobre un alfabet  $\Sigma$ .

1.  $\varepsilon$  és una expressió regular que denota  $\{\varepsilon\}$ .
2. Si  $a \in \Sigma$ ,  $a$  és una expressió regular que denota  $\{a\}$ .
3. Si  $r$  i  $s$  denoten els llenguatges  $L(r)$  i  $L(s)$ ,
  - $(r) | (s)$  denota  $L(r) \cup L(s)$
  - $(r)(s)$  denota  $\{tu : t \in L(r), u \in L(s)\}$
  - $(r^*)$  denota la clausura transitiva de  $L(r)$

# Expressions regulars

Exemples:

$$\Sigma = \{a, b\}$$

RE	Language
$a b$	$\{a, b\}$
$(a b)(a b)$	$\{aa, ab, ba, bb\}$
$a^*$	$\{\epsilon, a, aa, aaa, aaaa, \dots\}$
$(a b)^*$	$\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \dots\}$
$a a^*b$	$\{a, b, ab, aab, aaab, aaaab, \dots\}$

# Generadors d'escàners

Les expressions regulars s'usen en eines per crear compiladors:

- lex, ANTLR, ...

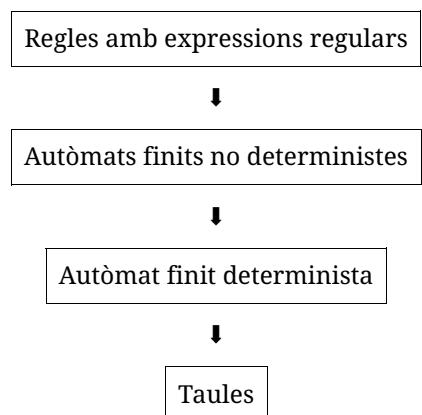
I també,

- en comandes del SO per tractar fitxers (grep, sed, ...)
- en llibreries d'LPs per tractar textos (re en Python, directament en Javascript, ...)

# Generadors d'escàners

A partir de la definició lèxica, l'escàner és un autòmat determinista que produceix com a sortida els tokens reconeguts.

Construcció:



# Analitzador lèxic d'ANTLR

```

RET      : 'return' ;
LPAREN  : '(' ;
RPAREN  : ')' ;
ADD     : '+' ;
SUB     : '-' ;
MUL     : '*' ;
DIV     : ';' ;
DIGIT   : '0'..'9' ;
LETTER  : [a-zA-Z] ;
NUMBER  : (DIGIT)+ ;
IDENT   : LETTER (LETTER | DIGIT)* ;
WS      : [ \t\n]+ -> skip ;

```

Referència

- els noms dels tokens han de ser en majúscules
- textos entre cometes representen aquell text
- ajuntar indica concatenació
- | indicaunió
- \* indica zero o més
- + indica un o més
- ? indica un o cap
- ... indica rangs
- es poden agrupar elements amb parèntesis
- l'skip no reporta el token

## grep

La comanda grep (*global regular expression print*) permet cercar patrons en texts.

grep 'jpetit' /etc/passwd

Imatge: Dave Child, cheatography.com

Anchors	Assertions	Groups and Ranges
^ Start of string, or start of line in multi-line pattern	?= Lookahead assertion	. Any character except new line (\n)
\A Start of string	?! Negative lookahead	(a b) a or b
\$ End of string, or end of line in multi-line pattern	?<= Lookbehind assertion	(...) Group
\Z End of string	?< = ?<! Negative lookbehind	(?...) Passive (non-capturing) group
\b Word boundary	?> Once-only Subexpression	[abc] Range (a or b or c)
\B Not word boundary	?() Condition [if then]	[^abc] Not (a or b or c)
\c Start of word	?()  Condition [if then else]	[a-q] Lower case letter from a to q
\d End of word	?# Comment	[A-Q] Upper case letter from A to Q
		[0-7] Digit from 0 to 7
		\x Group/subpattern number "x"
		Ranges are inclusive.
Character Classes	Quantifiers	Pattern Modifiers
\c Control character	* 0 or more [3] Exactly 3	g Global match
\s White space	+ 1 or more [3..] 3 or more	i Case-insensitive
\S Not white space	? 0 or 1 [3..5] 3..4 or 5	m Multiple lines
\d Digit	Add a ? to a quantifier to make it ungreedy.	s Treat string as single line
\D Not digit		x Allow comments and whitespace in pattern
\w Word		e Evaluate replacement
\W Not word		U Ungreedy pattern
\x Hexadecimal digit		* PCRE modifier
\o Octal digit		
POSIX	Escape Sequences	String Replacement
[upper:] Upper case letters	\ Escape following character	\$\ nth non-passive group
[lower:] Lower case letters	\Q Begin literal sequence	\$2 "xyz" in '/(abc xyz)\$/'
[alpha:] All letters	\E End literal sequence	\$1 "xyz" in '/(?:abc xyz)\$/'
[alnum:] Digits and letters	"Escaping" is a way of treating characters which have a special meaning in regular expressions literally, rather than as special characters.	\$ Before matched string
[digit:] Digits		\$ After matched string
[xdigit:] Hexadecimal digits	The escape character is usually \	\$+ Last matched string
[punct:] Punctuation		\$& Entire matched string
[blank:] Space and tab		Some regex implementations use \ instead of \$.
[space:] Blank characters		
[control:] Control characters	\n New line	
[graph:] Printed characters	\r Carriage return	
[print:] Printed characters and spaces	\t Tab	
[word:] Digits, letters and underscore	\v Vertical tab	
	\f Form feed	
	\xxx Octal character xxx	
	\xhh Hex character hh	

# re

El mòdul `re` de Python proporciona operacions sobre expressions regulars.

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p.match("7654386732(.)")
None
>>> m = p.match('unicorn')
<re.Match object; span=(0, 5), match='unicorn'>
>>> m.group()
'unicorn'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)

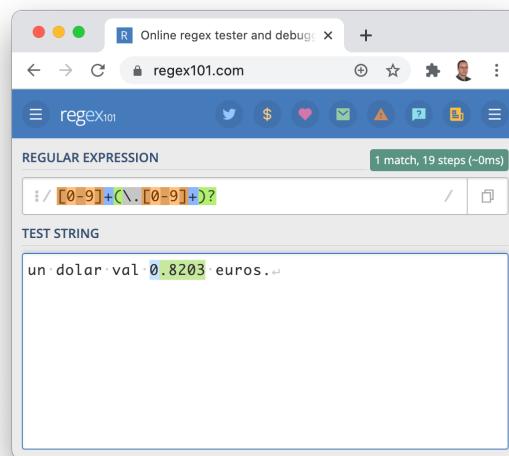
>>> re.search('casa', 'vaig a casa a dormir')
<re.Match object; span=(7, 11), match='casa'>
>>> re.sub('[0-9]+', 'molts', 'val 350 euros')
'val molts euros'
```

Tutorial

# Testing d'expressions regulars

Some people, when confronted with a problem, think "I'll use regular expressions".  
Now they have two problems.

— Jamie Zawinski, 1997



# Exercicis

Definiu expressions regulars pels llenguatges següents:

1. Identificadors de C++: poden contenir lletres, dígits i subratllats però no poden començar per un dígit.

2. Nombres en coma flotant. Si s'utilitza un punt decimal, un dígit decimal és obligatori.

Exemples: 3.1416, -3e4, +1.0e-5, .567e+8, ...

3. Totes les paraules amb alfabet {a, b, c} en les quals la primera aparició de b sempre és precedida per, com a mínim, una aparició de a.

4. Totes les paraules amb minúscules que contenen les cinc vocals en ordre (cada vocal només pot aparèixer un sol cop).

Example: zfaehipojksuj

5. Totes les paraules amb minúscules en les quals les lletres es troben en ordre lexicogràfic ascendent.

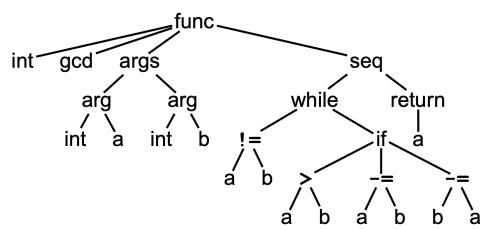
Examples: afhmñqsyz, abcdz, dgky, ... Contraexemple: bdeaz.

## Anàlisi sintàctica

# Anàlisi sintàctica

L'objectiu de l'**analitzador sintàctic** (o **parser**) és convertir una seqüència de tokens en un arbre de sintaxi abstracta que capture la jerarquia de les construccions.

```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a -= b; else b -= a;
    }
    return a;
}
```



→ Es descarta informació no rellevant com les paraules clau, els separadors, els parèntesis i els blocs.

→ Es facilita la feina dels propers estadis.

# Gramàtiques

La majoria dels LPs es descriuen a través de **gramàtiques incontextuals**, usant notació **BNF** (Backus–Naur form).

```
pgma → expr ; pgma
| ε

expr → expr + expr
| expr - expr
| expr * expr
| expr / expr
| ( expr )
| NUM
```

Les gramàtiques incontextuals permeten descriure llenguatges més amplis que els llenguatges regulars perquè són "recursives".

**Exemple:** Llenguatge dels mots capicues

- gramàtica incontextual
- expressió regular

→ La recursivitat permet definir jerarquies i niuar elements (parèntesis o blocs).

# Gramàtiques

Exemple: Gramàtica de C 

```
translation-unit      : {external-declaration}*
external-declaration   : function-definition
                        | declaration
function-definition    : {declaration-specifier}* declarator {declaration}* compound-statement
declaration-specifier  : storage-class-specifier
                        | type-specifier
                        | type-qualifier
storage-class-specifier : auto
                        | register
                        | static
                        | extern
                        | typedef
type-specifier         : void
                        | char
                        | short
                        | int
                        | long
                        | float
                        | double
                        | signed
                        | unsigned
                        | struct-or-union-specifier
                        | enum-specifier
                        | typedef-name
struct-or-union-specifier : struct-or-union identifier { {struct-declaration}+ }
                           | struct-or-union { {struct-declaration}+ }
                           | struct-or-union identifier
```

Font

# Dificultats

- Gramàtiques ambigües
- Prioritat dels operadors
- Associativitat dels operadors
- Analitzadors top-down vs. bottom-up
- Recursivitat per la dreta / per l'esquerra

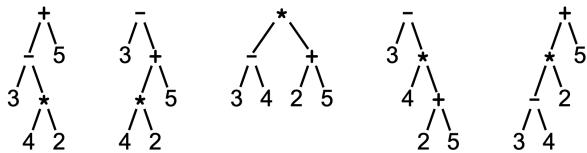
# Gramàtiques ambigües

Una gramàtica és **ambigua** si un mateix text es pot **derivar** (organitzar en un arbre segons la gramàtica) de diferents maneres.

Per exemple, amb

$\text{expr} \rightarrow \text{expr} + \text{expr} \mid \text{expr} - \text{expr} \mid \text{expr} * \text{expr} \mid \text{NUM}$

el fragment  $3 - 4 * 2 + 5$  es pot derivar d'aquestes maneres:



# Gramàtiques ambigües

Associar prioritat i associativitat als operadors sol permetre eliminar ambigüïtats.

- Prioritat:  $1 * 2 + 3 * 4$

\* té més prioritat que +

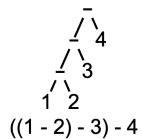


+ té més prioritat que \*

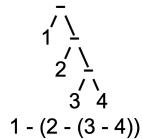


- Associativitat:  $1 - 2 - 3 - 4$

associativitat per l'esquerra



associativitat per la dreta



# Desambiguació de gramàtiques

Comencem amb:

```
expr → expr + expr  
| expr - expr  
| expr * expr  
| expr / expr  
| NUM
```

➡ És ambigua: no hi ha la prioritat ni associativitat.

# Desambiguació de gramàtiques

Podem assignar prioritats trencant en vàries regles, una per nivell:

```
expr → expr + expr  
| expr - expr  
| expr * expr  
| expr / expr  
| NUM
```

↓

```
expr → expr + expr  
| expr - expr  
| term
```

```
term → term * term  
| term / term  
| NUM
```

➡ Encara és ambigua: falta associativitat.

# Desambiguació de gramàtiques

Podem fer que un costat o altre afecti el següent nivell de prioritat:

```
expr → expr + expr  
| expr - expr  
| term
```

```
term → term * term  
| term / term  
| NUM
```



```
expr → expr + term  
| expr - term  
| term
```

```
term → term * NUM  
| term / NUM  
| NUM
```

La gramàtica ja no és ambigua.

# Desambiguació de gramàtiques

Podem fer que un costat o altre afecti el següent nivell de prioritat:

```
expr → expr + expr  
| expr - expr  
| term
```

```
term → term * term  
| term / term  
| NUM
```



```
expr → expr + term  
| expr - term  
| term
```

```
term → term * NUM  
| term / NUM  
| NUM
```

La gramàtica ja no és ambigua.

Però el - queda associat per la dreta...

# Generadors d'analitzadors sintàctics

Existeixen diferents tècniques per generar analitzadors sintàctics, amb propietats diferents.

- **Analitzadors descendents** (*top-down parsers*): reconeixen l'entrada d'esquerra a dreta tot buscant derivacions per l'esquerra expandint la gramàtica de l'arrel cap a les fulles.
- **Analitzadors ascendents** (*bottom-up parsers*): reconeixen primer sobre les unitats més petites de l'entrada analitzada abans de reconèixer l'estructura sintàctica segons la gramàtica.

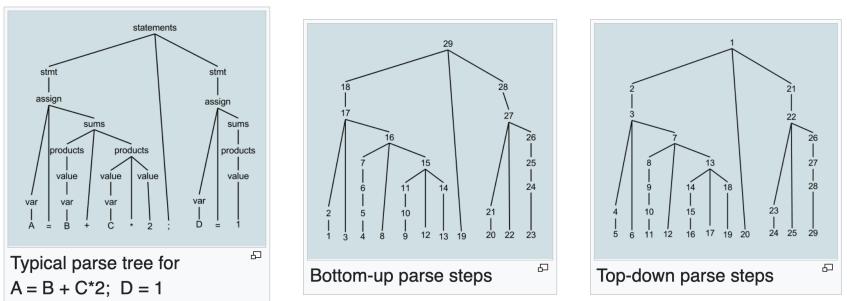


Figura: Wikipedia

# Generadors d'analitzadors sintàctics

## Analitzadors descendents $LL(k)$

- LL: Left-to-right, Left-most derivation
- $k$ : nombre de tokens que mira endavant

**Idea bàsica:** mirar el següent token per poder decidir quina producció utilitzar.

**Implementació:** Associar una funció a cada construcció del LP.

- Si la construcció està definida per una única regla:

La seva funció associada cridarà a les funcions associades a les construccions que apareixen en aquesta regla i comprovarà que els tokens que apareixen en la definició son els que apareixen en la seqüència donada.

- Si la construcció està definida per diverses regles:

La decisió sobre quina regla s'ha d'aplicar es basa en mirar els següents  $k$  tokens.

# Generadors d'analitzadors sintàctics

## Analitzadors descendents LL(1)

Exemple de gramàtica:

```
root   : stmt* 'end'  
;  
stmt   : 'if' expr 'then' stmt  
| 'while' expr 'do' stmt  
| expr ':=' expr  
;  
expr   : NUMBER  
| '(' expr ')' ;
```

Parser LL(1):

```
def stmt():  
    if current_token() == IF:  
        match(IF)  
        cond = expr()  
        match(THEN)  
        then = stmt()  
        return Node(IF, cond, then)  
    elif current_token() == WHILE:  
        match(WHILE)  
        cond = expr()  
        match(DO)  
        loop = stmt()  
        return Node(WHILE, cond, loop)  
    elif current_token() in [NUMBER, LPAREN]:  
        lvalue = expr()  
        match(ASSIGN)  
        rvalue = expr()  
        return Node(ASSIGN, lvalue, rvalue)  
    else:  
        SyntaxError()
```

**Exercici:** Implementeu `expr()` i `root()`.

# Generadors d'analitzadors sintàctics

## Analitzadors descendents LL(1)

Inconvenients principals:

- Les produccions no poden tenir prefixos comuns (no sabria quina triar).

```
expr → ID ( expr )  
      | ID = expr
```

- Les regles no poden tenir recursivitat per l'esquerra (es penjaria).

```
expr → expr + term  
      | term
```

# Generadors d'analitzadors sintàctics

## Analitzadors descendents LL(1)

Comencem amb:

```
expr → expr '+' term          • prefixos comuns
      | expr '-' term
      | term
```

↓ factoritzem els prefixos comuns

```
expr → expr ('+' term | '-' term)   • recursivitat per l'esquerra
      | term
```

↓ substituim recursivitat per l'esquerra per recursivitat per la dreta

```
expr → term expr2
expr2 → '+' term expr2
       | '-' term expr2
       | ε
```



## ANTLR

ANTLR és un analitzador descendente  $LL(k)$ .

També permet usar \* i + a les regles sintàctiques.

```
expr → expr '+' term
      | expr '-' term
      | term
```

↓ s'escriu senzillament

```
expr : term ('+' term | '-' term) * ;
```

A més, la prioritat dels operadors ve donada per l'ordre d'escriptura:

```
expr : expr '*' expr
      | expr '+' expr
      | NUM ;
```

I es pot definir fàcilment l'associativitat:

```
expr : <assoc=right> expr '^' expr
      | NUM ;
```

# Exercicis

**P1:** Escriviu gramàtiques no ambigües pels llenguatges següents:

1. El conjunt de tots els mots amb as i bs que són palíndroms.
2. Mots que tenen el patró  $a^*b^*$  amb més as que bs.
3. Textos amb parèntesis i claudàtors ben aniuats.  
Exemple: ( [ [ ] ( ( ) [ ( ) ] [ ] ) ] ).
4. El conjunt de tots els mots amb as i bs tals que a cada a li segueix immediatament per, almenys, una b.
5. El conjunt de tots els mots amb as i bs amb el mateix nombre d'as que bs.
6. El conjunt de tots els mots amb as i bs amb un nombre diferent d'as que bs.
7. Blocs d'instruccions separades per ; a la Pascal. Exemple:  
`BEGIN instrucció ; BEGIN instrucció ; instrucció END ; instrucció END.`
8. Blocs d'instruccions acabades per ; a la C.  
Exemple: { instrucció ; { instrucció ; instrucció ; } ; instrucció ; }.

# Exercicis

**P2:** Especifiqueu les gramàtiques anteriors amb notació ANTLR, modificant la gramàtica si és necessari.

# Exercicis

**P3:** Sense utilitzar cap eina ni llibreria (ni `eval!`), escriviu en Haskell, Python o C++ un analitzador descendant LL(1) que llegeixi una seqüència d'expressions i escrigui el resultat de cadascuna d'elles. [TBD: problema pel Jutge! 😊]

- Entrada:

```
2
2 * 3
2 * 3 + 1
2 * (3 + 1)
10 - 3 - 2
13 / 3
```

- Sortida:

```
2
6
7
8
5
4
```

# Exercicis

**P4:** Sense utilitzar cap eina ni llibreria, escriviu en Haskell, Python o C++ un analitzador descendant LL(1) que llegeixi una seqüència d'expressions i construeixi i escrigui l'arbre de sintaxi abstracta de cadascuna. [TBD: problema pel Jutge! 😊]

- Entrada:

```
2 * (3 + 1)
(10 - 3 - 2) / 4
```

- Sortida:

```
(MUL 2 (SUM 3 1))
(DIV (SUB (SUB 10 3) 2) 4)
```

# Arbres de sintaxi abstracta

## Accions

- Amb un analitzador descendent, es poden executar accions durant el reconeixement de les regles.

Les accions poden aparèixer en qualsevol punt de la regla:

```
regla  :  { /* abans */   }
          regla1
          { /* durant */   }
          regla2
          { /* durant */   }
          regla3
          { /* després */  }
;
```

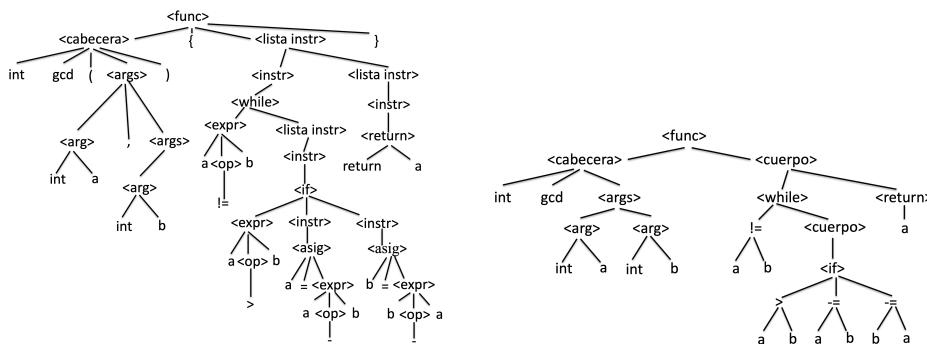
- La gramàtica esdevé "imperativa".
- Les accions s'entrellacen amb la gramàtica.
- És fàcil entendre què passa i quan passa.

- Amb un analitzador ascendent, només es poden executar accions després de reconèixer una regla.

# Arbres de sintaxi

Usualment, les accions construeixen un arbre de sintaxi concreta que segueix les regles de la gramàtica.

Aquest arbre es sol convertir en un arbre de sintaxi abstracta.



- Es separa l'anàlisi de la traducció.
- Es facilita les modificacions tot minimitzant les interaccions.
- Es permet que diferents parts del programa s'analitzin en ordres diferents.

# Arbres de sintaxi concreta vs abstracta

**Arbre de sintaxi concreta / de derivació:** Reflecteix exactament les regles sintàctiques.

**Arbre de sintaxi abstracta (abstract syntax tree, AST):** Representa el programa fidelment, però elimina i simplifica detalls sintàctics irrelevants.

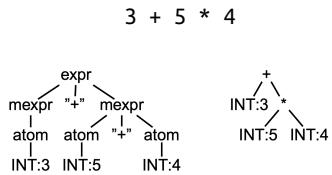
**Exemple:** Eliminar regles per desambiguar gramàtica.

**Exemple:** Aplanar una llista de paràmetres.

```

expr   : mexpr ('+' mexpr)* ;
mexpr  : atom ('*' atom)* ;
atom   : NUM ;

```

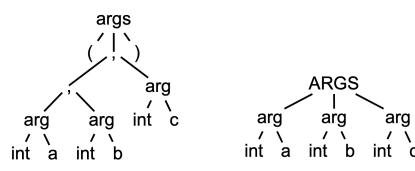


Concrete Parse Tree      Abstract Syntax Tree

```

int gcd(int a, int b, int c)

```



# Ús dels ASTs

Un cop construït l'AST, les etapes següents el recorren per a dur a terme les seves tasques:

- L'anàlisi semàntica verificarà l'ús correcte dels elements del programa.
- El generador de codi visitarà l'arbre i li aplicarà regles per generar codi intermedi.
- L'intèrpret es passejarà per l'arbre per dur a termes les seves instruccions.

## ASTs en ANTLR

A partir de la gramàtica, ANTLR pot generar un analitzador descendent amb accions que construeixin un AST.

L'AST es pot visitar a través de *visitors* (un patró de disseny).

ANTLR també genera la interfície dels visitadors, tot generant un esquelet de mètodes que podem heretar.

Cada mètode s'aplica sobre un tipus de node que correspon a cada regla de la gramàtica.

# Anàlisi semàntica

## Anàlisi semàntica

L'analitzador semàntic recorre l'AST, per obtenir tota la informació necessària per poder generar codi.

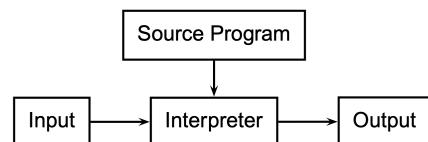
Objectius:

- Comprovar la correcció semàntica del programa (comprovació de tipus).
- Resoldre ambigüïtats.
- Assignar memòria.
- Construir la taula de símbols.

➡ En aquest curs no entrem en aquest vast tema, que deixem per l'assignatura de Compiladors (recomanada!). Però al tema "Inferència de tipus" veure com fer comprovació de tipus.

# Interpretació

## Interpretació



**Tutorial:** Escriure (en Haskell) un intèrpret per a l'AST d'un senzill llenguatge de programació (SimpleLP™).

# SimpleLP

## Definició del llenguatge

- Tipus de dades: enteros.
- Variables: siempre visibles, inicializadas a 0.
- Operadores aritméticos: + - \* /.
- Operadores relacionales: ≠ < (retornen 0 o 1).
- Instrucciones: asignación, composición secuencial, condicional y iteración.

## Exemple

```
# Algorisme d'Euclides per calcular el mcd de 105 i 252.
a ← 105
b ← 252
while a ≠ b do
    if a < b then
        b ← b - a
    else
        a ← a - b
    end
end
```

# SimpleLP

## Tipus de dades per l'AST

```
data Expr
= Val Int
| Var String
| Add Expr Expr
| Sub Expr Expr
| Neq Expr Expr
| Lth Expr Expr
    -- valor
    -- variable
    -- suma (+)
    -- resta (-)
    -- diferente-de (=)
    -- menor-que (<)

data Instr
= Ass String Expr
| Seq [Instr]
| Cond Expr Instr Instr
| Loop Expr Instr
    -- assignació
    -- composició seqüencial
    -- condicional
    -- iteració
```

# SimpleLP

## Tipus de dades per l'AST

Exemple: AST pel programa anterior:

```
Seq [  
  (Ass "a" (Val 105))  
  (Ass "b" (Val 252))  
  ,  
  (While  
    (Neq (Var "a") (Var "b"))  
    (Cond  
      (Lth (Var "a") (Var "b"))  
      (Ass "b" (Sub (Var "b") (Var "a")))  
      (Ass "a" (Sub (Var "a") (Var "b")))  
    )  
  )  
]
```

# SimpleLP

## Memòria

Descrivim el valor de les variables a través d'una memòria de tipus `Mem` amb aquestes operacions:

```
-- retorna una memòria buida  
empty :: Mem  
-- insereix (o canvia si ja hi era) una clau amb el seu valor  
update :: Mem -> String -> Int -> Mem  
-- consulta el valor d'una clau en una memòria  
search :: Mem -> String -> Maybe Int  
-- retorna la llista de claus en una memòria  
keys :: Mem -> [String]
```

La implementació seria amb qualsevol diccionari (BST, AVL, hashing, ...).  
[Si voleu provar-ho, useu `Data.Map`.]

# SimpleLP

## Avaluació de les expressions

```
--- avalia una expressió en un estat de la memòria
eval :: Expr -> Mem -> Int

eval (Val x) m = x
eval (Var v) m =
  case search v m of
    Nothing -> 0
    Just x -> x
eval (Add e1 e2) m = eval' e1 e2 m (+)
eval (Sub e1 e2) m = eval' e1 e2 m (-)
eval (Neq e1 e2) m = b2i $ eval' e1 e2 m (/=)
eval (Lth e1 e2) m = b2i $ eval' e1 e2 m (<)

eval' e1 e2 m op = op (eval e1 m) (eval e2 m)

b2i False = 0
b2i True = 1
```

# SimpleLP

## Interpretació de les instruccions

```
--- retorna l'estat final de la memòria després d'executar una instrucció
--- partint d'un estat inicial de la memòria
exec :: Instr -> Mem -> Mem

exec (Ass v e) m = update v (eval e m) m
exec (Seq []) m = m
exec (Seq (i:is)) m = exec (Seq is) (exec i m)
exec (Cond b i1 i2) m = exec (if eval b m /= 0 then i1 else i2) m
exec (Loop b i) m =
  if eval b m /= 0
    then exec (Loop b i) (exec i m)
  else m
```

# SimpleLP

## Execució del programa

Escriu el valor final de cada variable (en ordre lexicogràfic) després d'executar una instrucció partint d'una memòria buida.

```
run :: Instr -> IO ()  
  
run i = mapM_ printEntry $ sort $ keys m  
where  
    m = exec i empty  
    printEntry k = do  
        putStrLn k  
        putStrLn ":"  
        print $ fromJust $ search k m
```

Execució amb el programa anterior:

```
a : 21  
b : 21
```

## Exercicis

1. Modifiqueu l'AST i eval per tenir operadors de resta i producte.
2. Modifiqueu l'AST, exec i run per tal d'afegir a SimpleLP una nova instrucció print que escrigui el contingut d'una expressió. Ara run només ha d'executar el programa donat partint d'una memòria buida.
3. Afegiu una expressió read a SimpleLP que retorni el valor del següent enter de l'entrada.
4. Afegiu una instrucció del tipus for i ← a .. b.
5. Feu que print pugui escriure una llista de valors (print a, b, a + b per ex).
6. Escriviu l'AST d'aquest programa:

```
# factorial en SimpleLP  
n ← read  
f ← 1  
for i ← 2 .. n  
    f ← f * i  
end  
print n, f
```

# Laboratori

## Laboratori



Vegeu les transparències [Python 3: compiladors](#) de Gerard Escudero.