

Paradigmas de la Programación

FaMAF 2015

sintaxis vs. semántica

las diferencias en forma...

C, C++, Java:

```
int fact (int n) { return (n == 0) ? 1 : n * fact (n-1); }
```

Scheme:

```
(define fact  
  (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))
```

ML:

```
fun fact n = if n=0 then 1 else n*fact(n-1);
```

Haskell:

```
fact :: Integer->Integer  
fact 0 = 1  
fact n = n*fact(n-1)
```

sintaxis vs. semántica

... no necesariamente implican diferencias en el contenido (los efectos en la máquina)

catálogo de herramientas

- teoría de computabilidad: el poder y las limitaciones de los programas
- teoría de tipos: sintaxis y semántica de los lenguajes

... pero todavía no tenemos una buena teoría matemática que dé cuenta de funciones de alto orden, transformaciones de estado y concurrencia :(

qué es computable

- [video del halting problem](#) (o la prueba de que las computadoras no pueden hacer todo)

programa: sintaxis y semántica

- un programa es la descripción de un proceso dinámico
 - **sintaxis**: texto del programa
 - **semántica**: cosas que hace
- la implementación de un lenguaje de programación debe transformar la sintaxis de un programa en instrucciones de máquina que se pueden ejecutar para que suceda la secuencia de acciones que se pretendía

transformación de sintaxis a semántica

un lenguaje de programación es un conjunto de abstracciones y empaquetamientos quizás sin correspondencia directa con la máquina

- es necesario **traducir** lenguaje de programación a instrucciones de máquina
- el **compilador** hace esa traducción
- un **intérprete** puede combinar traducción y ejecución

sintaxis

compilador (o intérprete)

- el compilador se encarga de procesar la sintaxis de los lenguajes de programación
- un intérprete incluye un compilador y la ejecución

usaremos las filminas de [Xin Yuan](#)

[http://www.cs.fsu.edu/~xyuan/cop4020/
compiler_phases.ppt](http://www.cs.fsu.edu/~xyuan/cop4020/compiler_phases.ppt)

si quieren saber más, pueden ver más filminas en la [página del curso](#)

qué es un compilador?

un programa que lee un programa escrito en un lenguaje origen y lo traduce a un programa equivalente (con el mismo significado) en un lenguaje destino

- dos componentes
 - entender el programa (asegurarse de que es correcto)
 - Reescribir el programa
- normalmente, el lenguaje origen es de alto nivel y el destino es de bajo nivel

fases de un compilador

- análisis léxico
- análisis sintáctico
- análisis semántico
- generación de código intermedio (independiente de máquina)
- optimización de código intermedio
- generación de código destino (dependiente de máquina)
- optimización de código destino

proceso de compilación

programa origen, con macros

preprocesador

programa origen

compilador

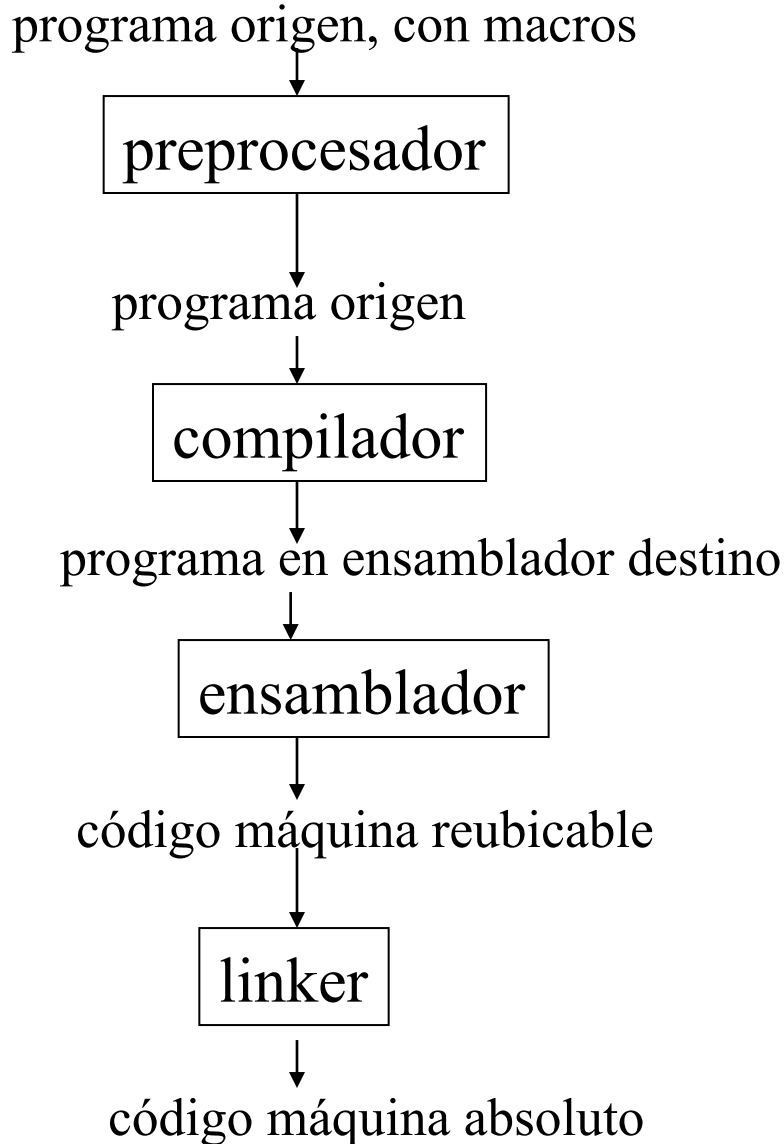
programa en ensamblador destino

ensamblador

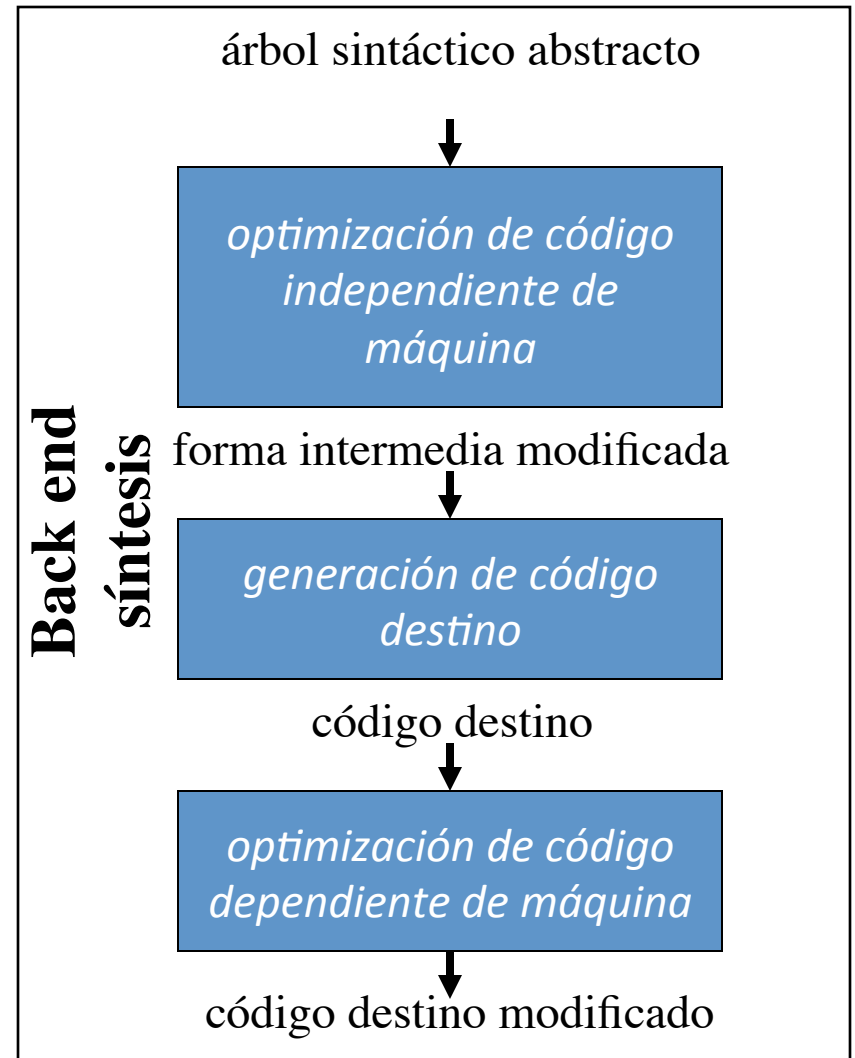
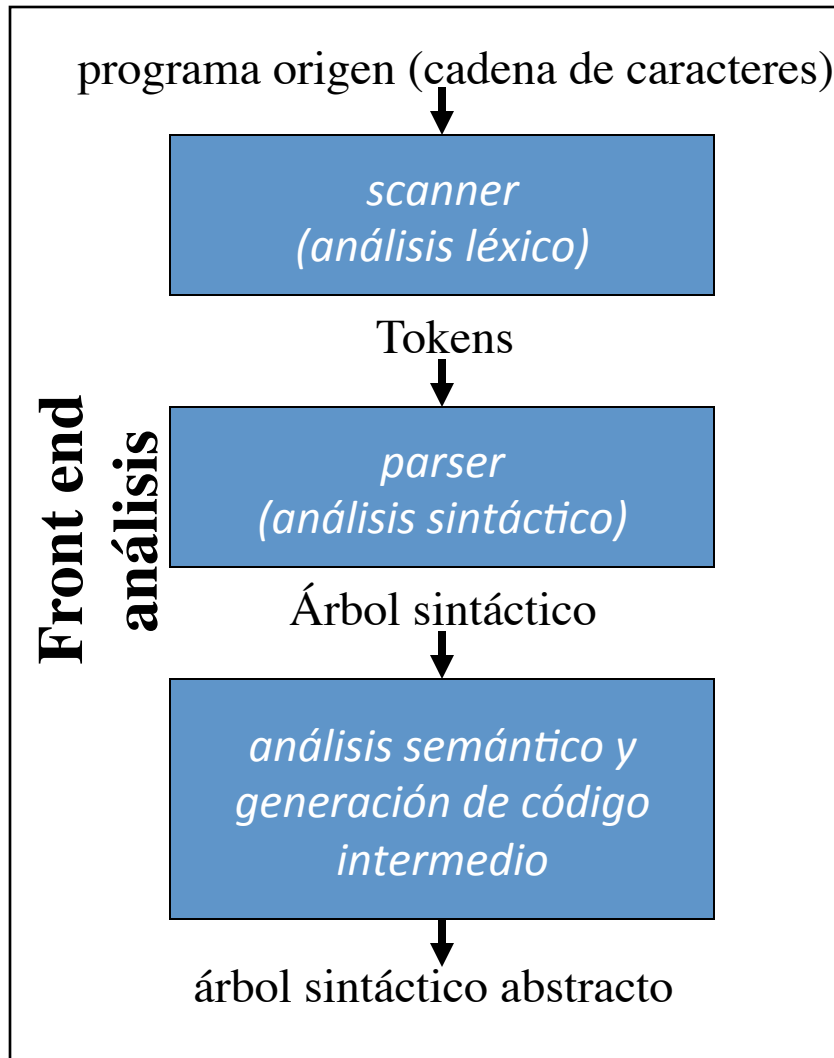
código máquina reubicable

linker

código máquina absoluto



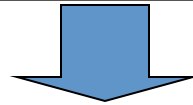
Front-end y Back-end



Scanner: análisis léxico

- se divide un programa (secuencia de caracteres) en palabras (*tokens*)

```
program gcd (input, output);  
var i, j : integer;  
begin  
  read (i, j);  
  while i <> j do  
    if i > j then i := i - j else j := j - i;  
  writeln (i)  
end.
```



program	gcd	(input	,	output)	;
var	i	,	j	:	integer	;	begin
read	(i	,	j)	;	while
i	<>	j	do	if	i	>	j
then	i	:=	i	-	j	else	j
:=	i	-	i	;	writeln	(i
)	end	.					

Scanner: análisis léxico

- qué tipo de errores se puede reportar en el análisis léxico?

A = b + @3;

Parser: análisis sintáctico

- comprueba si la secuencia de tokens conforma a la especificación gramatical del lenguaje y genera el árbol sintáctico
- la especificación gramatical suele representarse con una gramática independiente de contexto (*context free grammar*), que también le da forma al árbol sintáctico

gramáticas independientes de contexto

- se definen categorías de construcciones del lenguaje, por ejemplo:
 - Sentencias (Statements)
 - Expresiones (Expressions)
 - Declaraciones (Declarations)

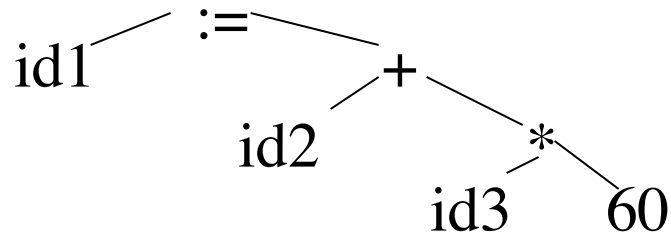
```
<statement> ::= <for-statement> | <if-statement> | <assignment>  
<for-statement> ::= for ( <expression> ; <expression> ; <expression> ) <statement>  
<assignment> ::= <identifier> := <expression>
```

ejemplo: Micro Pascal

$\langle \text{Program} \rangle \quad ::= \text{program } \langle \text{id} \rangle (\langle \text{id} \rangle \langle \text{More_ids} \rangle); \langle \text{Block} \rangle .$
 $\langle \text{Block} \rangle \quad ::= \langle \text{Variables} \rangle \text{begin } \langle \text{Stmt} \rangle \langle \text{More_Stmts} \rangle \text{end}$
 $\langle \text{More_ids} \rangle \quad ::= , \langle \text{id} \rangle \langle \text{More_ids} \rangle$
 | ϵ
 $\langle \text{Variables} \rangle \quad ::= \text{var } \langle \text{id} \rangle \langle \text{More_ids} \rangle : \langle \text{Type} \rangle ; \langle \text{More_Variables} \rangle$
 | ϵ
 $\langle \text{More_Variables} \rangle ::= \langle \text{id} \rangle \langle \text{More_ids} \rangle : \langle \text{Type} \rangle ; \langle \text{More_Variables} \rangle$
 | ϵ
 $\langle \text{Stmt} \rangle \quad \quad ::= \langle \text{id} \rangle := \langle \text{Exp} \rangle$
 | **if** $\langle \text{Exp} \rangle$ **then** $\langle \text{Stmt} \rangle$ **else** $\langle \text{Stmt} \rangle$
 | **while** $\langle \text{Exp} \rangle$ **do** $\langle \text{Stmt} \rangle$
 | **begin** $\langle \text{Stmt} \rangle \langle \text{More_Stmts} \rangle$ **end**
 $\langle \text{Exp} \rangle \quad \quad ::= \langle \text{num} \rangle$
 | $\langle \text{id} \rangle$
 | $\langle \text{Exp} \rangle + \langle \text{Exp} \rangle$
 | $\langle \text{Exp} \rangle - \langle \text{Exp} \rangle$

ejemplo de análisis sintáctico

id1: = id2 + id3 * 60



análisis semántico

- el compilador trata de ver si un programa **tiene sentido** analizando su árbol sintáctico
- un programa sin errores gramaticales no siempre es correcto, puede haber problemas de tipo

`pos = init + rate * 60`

- qué pasa si *pos* es una clase y *init* y *rate* son enteros?
- el parser no puede encontrar este tipo de errores
- el análisis semántico encuentra este tipo de error

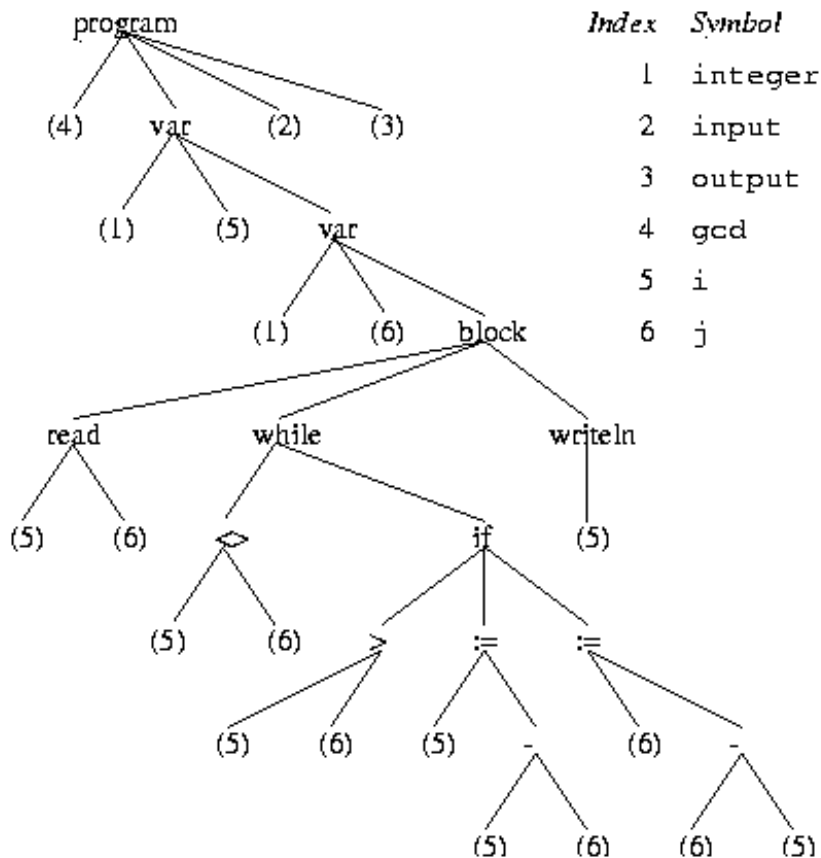
análisis semántico

- el compilador hace comprobaciones semánticas **estáticas** (*static semantic checks*)
 - comprobación de tipos
 - declaración de variables antes de su uso
 - se usan los identificadores en contextos adecuados
 - comprobar argumentos
 - si hay un fallo en compilación, se genera un **error**
- en **tiempo de ejecución** (*dynamic semantic checks*) se comprueba:
 - que los valores de los arreglos estén dentro de los límites
 - errores aritméticos (división por 0)
 - no se desreferencian los punteros si no apuntan a un objeto válido
 - se usan variables sin inicialización
 - si hay un fallo en ejecución, se levanta una **excepción**

tipado fuerte

- un lenguaje tiene tipado fuerte si siempre se detectan los errores de tipo
 - en tiempo de compilación o de ejecución
 - tipado fuerte: Ada, Java, ML, Haskell
 - tipado débil: Fortran, Pascal, C/C++, Lisp
 - *duck typing*: Python
- el tipado fuerte hace que el lenguaje sea más seguro y fácil de usar sin errores, pero potencialmente más lento por las comprobaciones dinámicas
- en algunos lenguajes algunos errores de tipo se detectan tarde, lo que los hace poco fiables (Basic, Lisp, Prolog, lenguajes de scripting)

Code Generation and Intermediate Code Forms



- A typical intermediate form of code produced by the semantic analyzer is an abstract syntax tree (AST)
- The AST is annotated with useful information such as pointers to the symbol table entry of identifiers

Example AST for the gcd program in Pascal

código intermedio

el código intermedio está cerca de la máquina pero sigue siendo fácil de manipular, para poder implementar optimizaciones. Por ejemplo:

```
temp1 = 60
temp2 = id3 + temp1
temp3 = id2 + temp2
id1 = temp3
```

se puede optimizar (independientemente de máquina):

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```


código destino

- de la forma independiente de máquina se genera ensamblador:

MOVF id3, R2

MULF #60.0, R2

MOVF id2, R1

ADDF R2, R1

MOVF R1, id1

- este código específico de máquina se optimiza para explotar características de hardware específicas

ejemplo de compilación de una expresión

```
float position, initial, rate;  
position = initial + rate * 60;
```

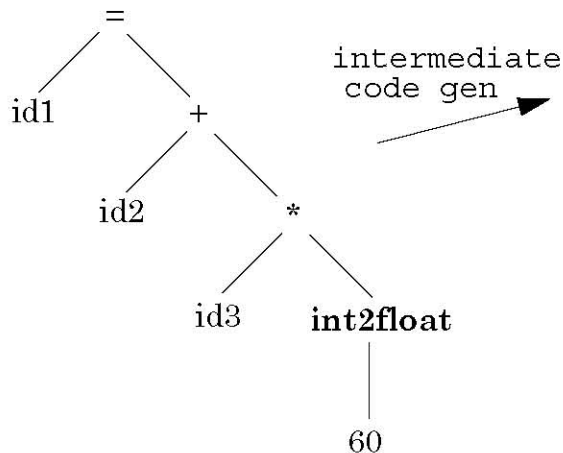
↓ análisis léxico (scanner)

```
[ID, "position"] [ASSIGN, '=' ] [ID, "initial"] [PLUS, '+' ] [ID, "rate"] [MULT, '*' ] [NUM, 60] [SEMICOLON, ';' ]
```

expresión tokenizada:

id1 = id2 + id3 * 60 conversión de tipos implícita

↓ parser



intermediate
code gen

intermediate code

```
temp1 = int2float(60)  
temp2 = mult(id3, temp1)  
temp3 = add(id2, temp2)  
id1 = temp3
```

optimizer

optimized interm. code

```
temp1 = mult(id3, 60.0)  
id1 = add(id2, temp1)
```

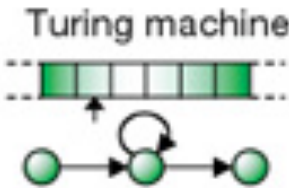

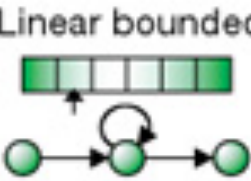

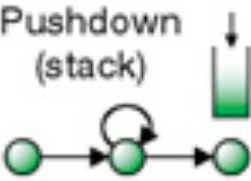



code
generator

assembly code

```
movf id3, fp2  
mulf #60.0, fp2  
movf id2, fp1  
addf fp2, fp1  
movf fp1, id1
```

gramáticas

jerarquía de Chomsky

Language	Automaton	Grammar	Recognition
Recursively enumerable languages	<p>Turing machine</p> 	<p>Unrestricted</p> $Baa \rightarrow A$	<p>Undecidable</p> 
Context-sensitive languages	<p>Linear bounded</p> 	<p>Context sensitive</p> $At \rightarrow aA$	<p>Exponential?</p> 
Context-free languages	<p>Pushdown (stack)</p> 	<p>Context free</p> $S \rightarrow gSc$	<p>Polynomial</p> 
Regular languages	<p>Finite-state automaton</p> 	<p>Regular</p> $A \rightarrow cA$	<p>Linear</p> 

gramáticas independientes de contexto

vamos a estar usando gramáticas
independientes de contexto (CFG, *context-free
grammars*), aunque algunas propiedades de los
lenguajes de programación escapan a su
expresividad

gramáticas y árboles sintácticos

una gramática...

- es un método para
 - definir conjuntos infinitos de expresiones
 - procesar expresiones
- consiste de
 - símbolo inicial
 - no terminales
 - terminales
 - producciones

gramática del lenguaje de expresiones numéricas

$e ::= n \mid e + e \mid e - e$

$n ::= d \mid nd$

$d ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

expresiones:

0

1 + 3 - 5

2 - 28 + 347598

gramáticas como método

- no terminales
 - forma adecuada de describir la composicionalidad de las expresiones
 - no pueden formar parte de una expresión, siempre se tienen que substituir por terminales
- derivación: secuencia de sustituciones que termina en una cadena de terminales

$e \rightarrow n \rightarrow nd \rightarrow dd \rightarrow 2d \rightarrow 25$

$e \rightarrow e-e \rightarrow e-e+e \rightarrow \dots \rightarrow n-n+n \rightarrow \dots \rightarrow 10-15+12$

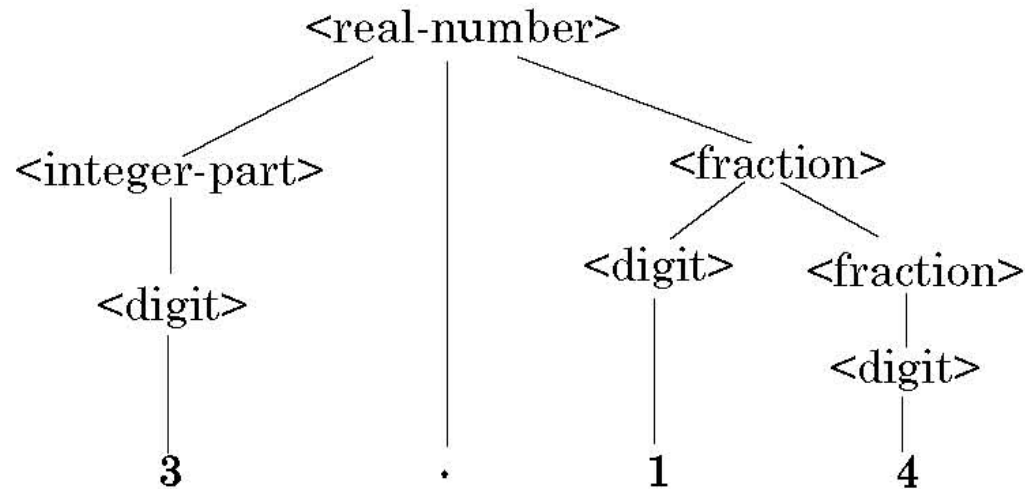
CFG para decimales

```
<real-number> ::= <integer-part> '.' <fraction-part>  
<integer-part> ::= <digit> | <integer-part> <digit>  
<fraction> ::= <digit> | <digit> <fraction>  
<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

::= stands for production rule; **<...>** are non-terminals;

| represents alternatives for the right-hand side of a production rule

Sample parse tree:



CFG para paréntesis balanceados

$\langle \text{balanced} \rangle ::= (\langle \text{balanced} \rangle) \mid \langle \text{empty} \rangle$

ejemplo de derivación: $\langle \text{balanced} \rangle \Rightarrow (\langle \text{balanced} \rangle)$
 $\Rightarrow ((\langle \text{balanced} \rangle))$
 $\Rightarrow ((\langle \text{empty} \rangle))$
 $\Rightarrow (())$

CFG para cifras

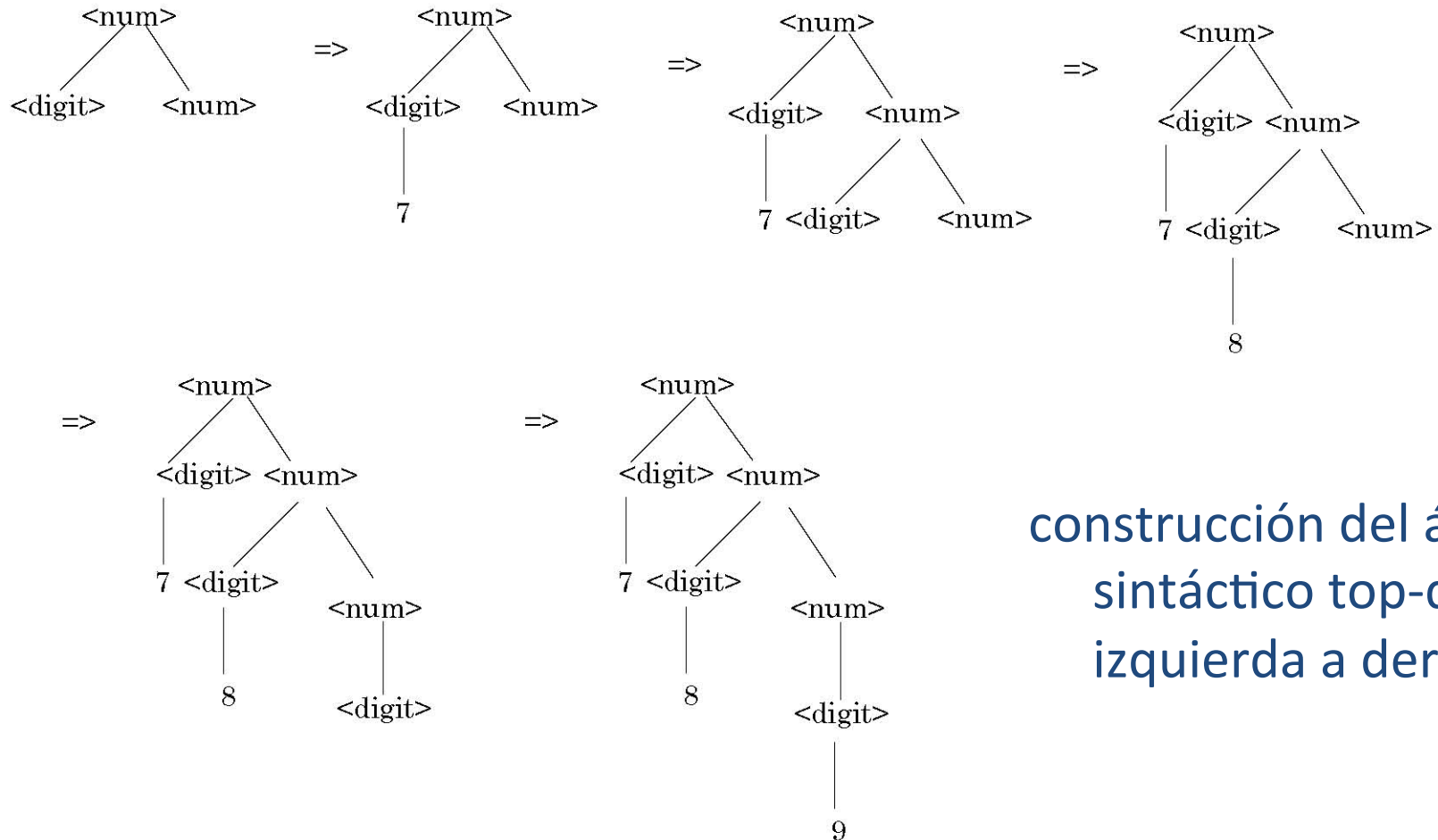
$\langle \text{num} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{num} \rangle$
 $\langle \text{digit} \rangle ::= '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'$

recursiva a la derecha

ejemplo de
derivación
top-down
izquierda - derecha

$\langle \text{num} \rangle \Rightarrow \langle \text{digit} \rangle \langle \text{num} \rangle$
 $\Rightarrow 7 \langle \text{num} \rangle$
 $\Rightarrow 7 \langle \text{digit} \rangle \langle \text{num} \rangle$
 $\Rightarrow 7 8 \langle \text{num} \rangle$
 $\Rightarrow 7 8 \langle \text{digit} \rangle$
 $\Rightarrow 7 8 9$

parsing recursivo descendiente



construcción del árbol
sintáctico top-down,
izquierda a derecha

- <http://www.garshol.priv.no/download/text/bnf.html>
- <http://cui.unige.ch/isi/bnf/JAVA/BNFindex.html>
- <http://cui.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html>

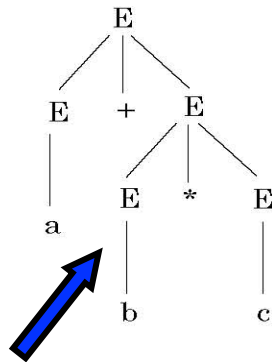
ambigüedad sintáctica

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid a \mid b \mid c$

Cómo parsear $a+b*c$ usando esta gramática?

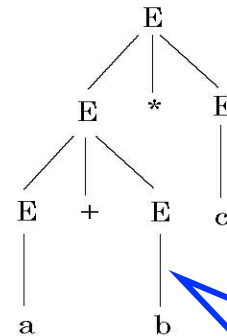
esta gramática es ambigua

Parse Tree from a rightmost derivation starting from $\langle \text{expr} \rangle + \langle \text{expr} \rangle$



ambos árboles son sintácticamente válidos

Parse Tree from a leftmost derivation starting with $\langle \text{expr} \rangle * \langle \text{expr} \rangle$



sólo este árbol es semánticamente correcto (la precedencia de operadores y la asociatividad son **semánticos**, no sintácticos)

este árbol es sintácticamente correcto, pero semánticamente incorrecto

eliminar a

no siempre podemos eliminar la ambigüedad de esta forma

- definir un símbolo distinto para cada nivel de precedencia de cada operador
- definir la parte derecha de las reglas de forma que se fuerce la asociatividad adecuada

$$\begin{array}{lcl} E & ::= & E + T \mid E - T \mid T \\ T & ::= & T * F \mid T / F \mid F \\ F & ::= & (E) \mid \text{id} \mid \text{num} \end{array}$$

esta gramática es inambigua

$E ::= E + T \mid E - T \mid T$
$T ::= T * F \mid T / F \mid F$
$F ::= (E) \mid \text{id} \mid \text{num}$

Leftmost:

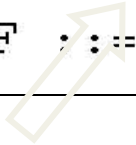
$E \Rightarrow \underline{E} + T$
 $\Rightarrow \underline{T} + T$
 $\Rightarrow \underline{F} + T$
 $\Rightarrow \text{id} + \underline{T}$
 $\Rightarrow \text{id} + \underline{T} * F$
 $\Rightarrow \text{id} + \underline{F} * F$
 $\Rightarrow \text{id} + \text{id} * F$
 $\Rightarrow \text{id} + \text{id} * \text{id}$

Rightmost:

$E \Rightarrow E + \underline{T}$
 $\Rightarrow E + T * \underline{F}$
 $\Rightarrow E + \underline{T} * \text{id}$
 $\Rightarrow E + \underline{F} * \text{id}$
 $\Rightarrow \underline{E} + \text{id} * \text{id}$
 $\Rightarrow \underline{T} + \text{id} * \text{id}$
 $\Rightarrow \underline{F} + \text{id} * \text{id}$
 $\Rightarrow \text{id} + \text{id} * \text{id}$

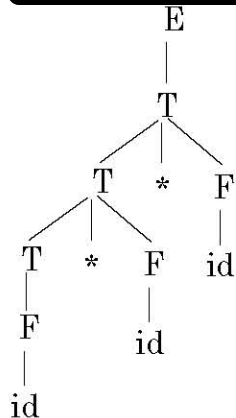
recursión a izquierda y a derecha

$E ::= E + T \mid E - T \mid T$
$T ::= T * F \mid T / F \mid F$
$F ::= (E) \mid \text{id} \mid \text{num}$




Leftmost non-terminal on the RHS of production is the same as the LHS

Left recursive parse tree for $\text{id} * \text{id} * \text{id}$ with **left-associativity**

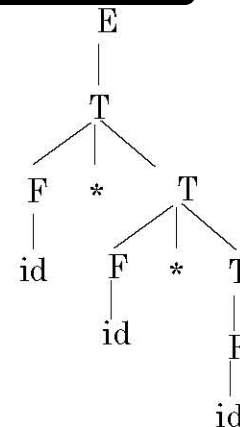


$E ::= T + E \mid T - E \mid T$
$T ::= F * T \mid F / T \mid F$
$F ::= (E) \mid \text{id} \mid \text{num}$



Right-recursive grammar

Right recursive parse tree for $\text{id} * \text{id} * \text{id}$ with **right associativity**



Can you think of any operators that are right-associative?

gramática de expresiones Yacc

- Yacc es un generador automático de parsers
- permite especificar explícitamente la precedencia y asociatividad de los operadores, sin modificar la gramática

```
%left PLUS MINUS          /* lowest precedence*/
%left MULT DIV
%nonassoc UNARY            /* highest precedence */
...
%%
...
expr:    LPAREN expr RPAREN    { $$ = $2; }
        | expr MULT expr      { $$ = $1 * $3; }
        | expr DIV expr       { $$ = $1 / $3; }
        | expr PLUS expr      { $$ = $1 + $3; }
        | expr MINUS expr     { $$ = $1 - $3; }
        | MINUS expr %prec UNARY { $$ = -$2; }
        | num
```