



# *Item 0: Introduction and definitions*

Mari Paz Guerrero Lebrero

Grado en Ingeniería Informática

Curso 2019/2020

# Topics

1. Introduction
2. Translator / compiler concept and types
3. Compiler execution environment
4. Stages of a compiler
5. Scanning
6. Parsing
7. Semantic analysis
8. Code generation
9. Other stages

# Introduction

# Abstraction effort

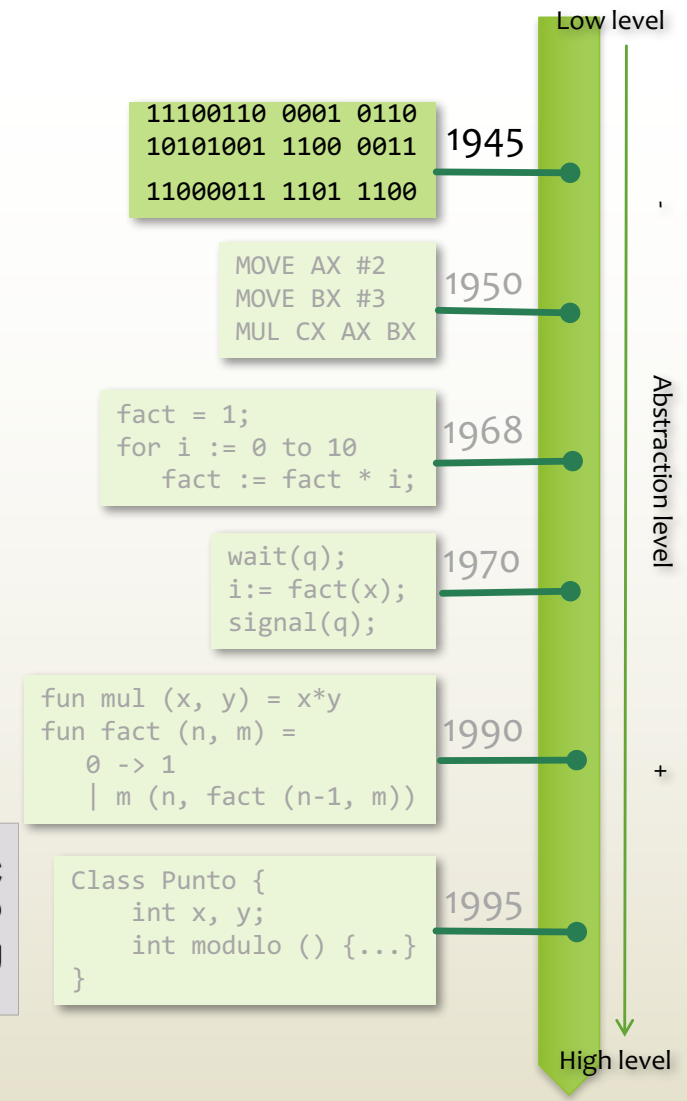
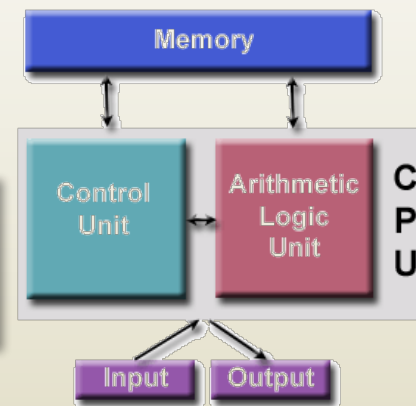
- ▶ Programming history can be described as a constant effort to bring the executable language of hardware architectures to a more close human language through successive steps of abstraction

# Abstraction effort

Von Neumann architectures

- Program representation as instructions in memory
- The control unit is sequentially reading the program
- Each instruction has an operation code and operands
- Arithmetic logic operations, comparative operations, jump operations, I/O operations ...

Operation code	Operating 1	Operating 2
11100110	0001	0110
10101001	1100	0011
11000011	1101	1100

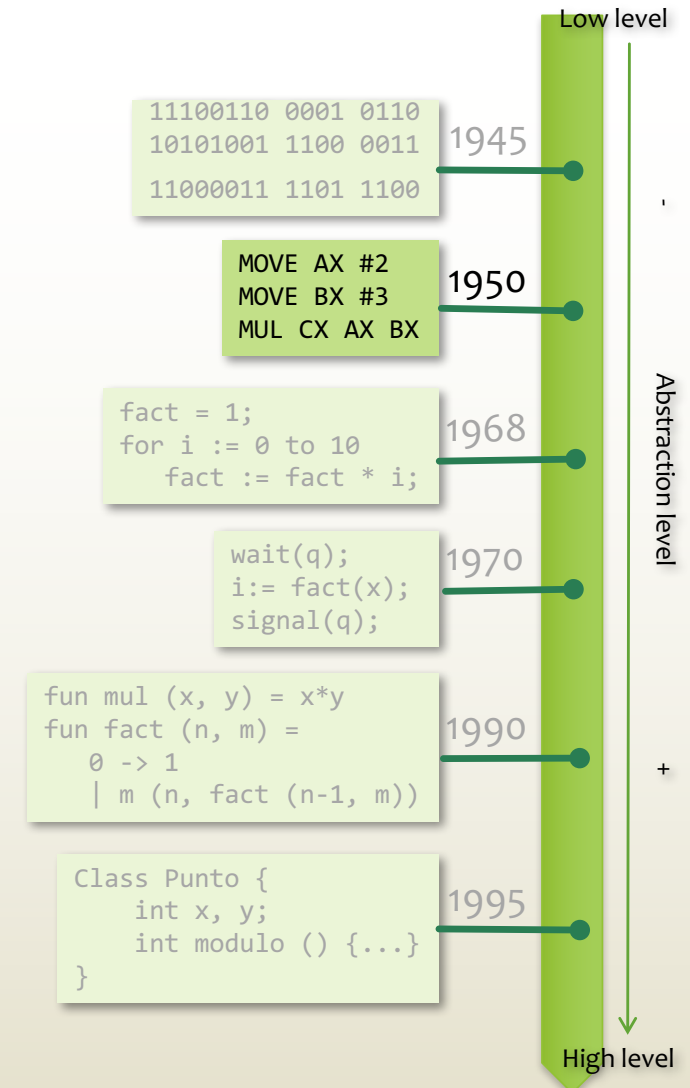


# Abstraction effort

## Assembler paradigm

- The operation codes are replaced by acronyms
- Operating are replaced by references to memory and records
- The instruction set remains the same

Operation acronym	Operating 1	Operating 2
MOVE	AX	#2
MOVE	BX	[1305]
MUL	CX	AX BX

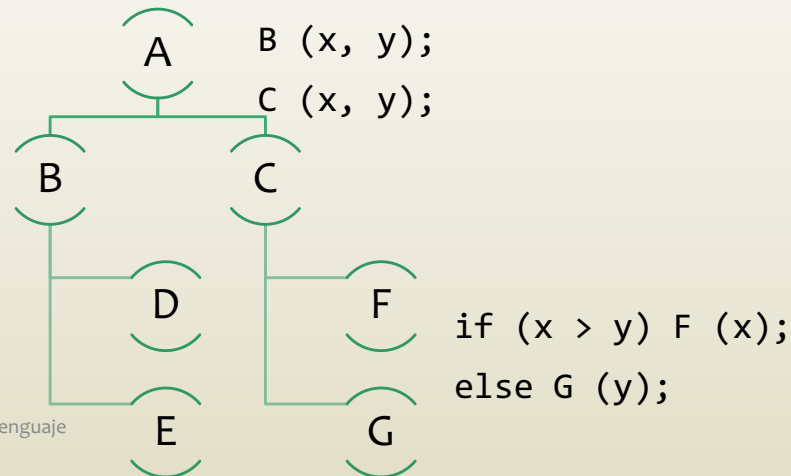


# Abstraction effort

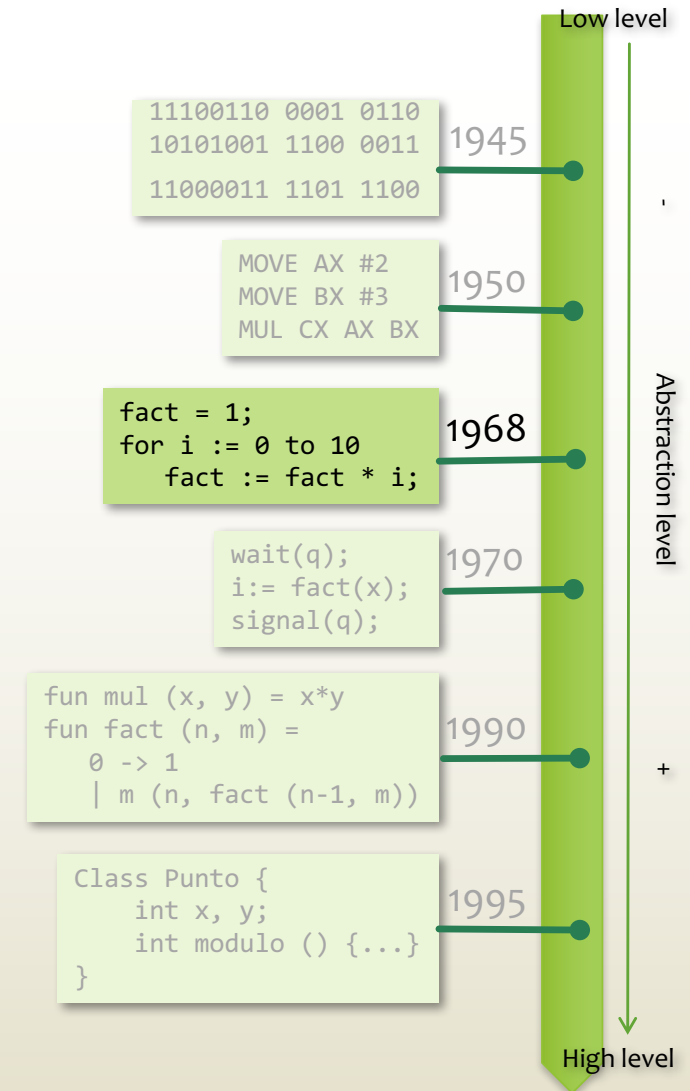
Structured imperative paradigm

- Sequential execution flow
- Data typification
- Conditional and iterative structures (not jump)
- Subroutines to modularize programs

```
while (x = y)
{
  D (x, y);
  E (x, y);
}
```



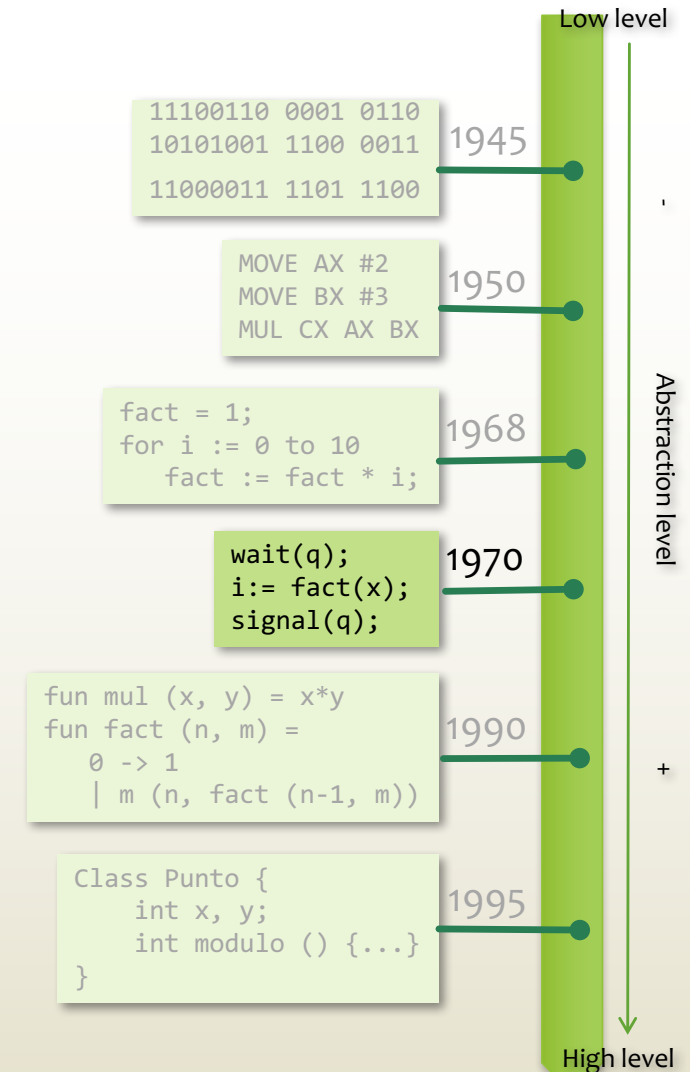
Procesadores del Lenguaje



# Abstraction effort

## Concurrent paradigm

- Several sequential execution flows associated with processes
- Take care concurrent access to resources
- Mechanisms for mutual exclusion and condition synchronization
- The algorithm remains imperative for each process
- There concurrent functional languages



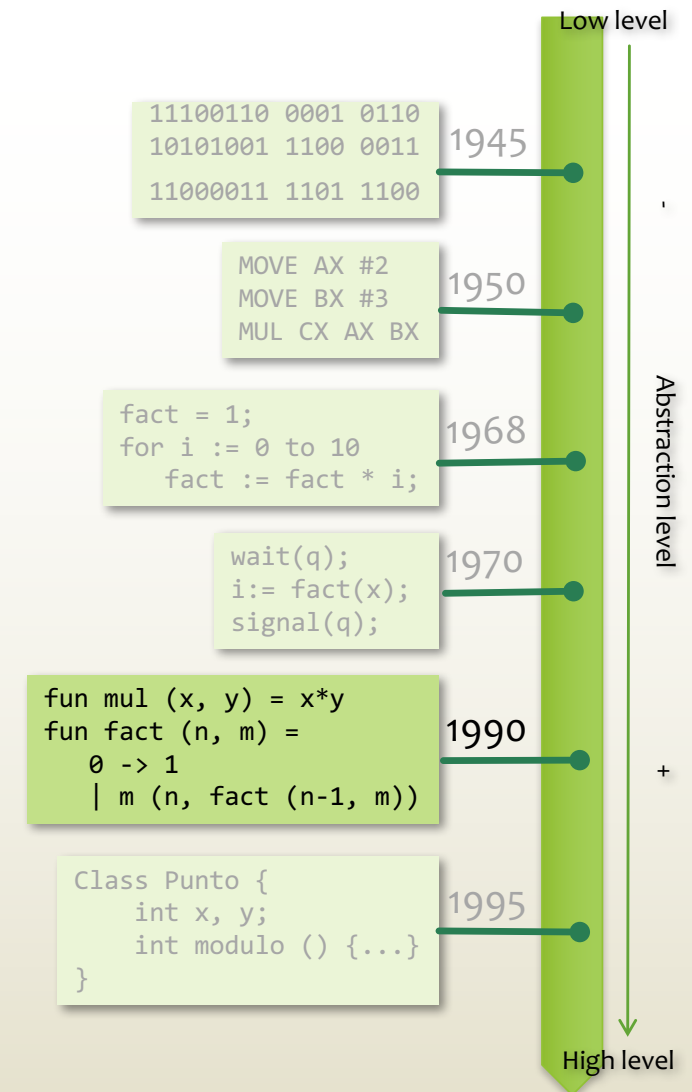
# Abstraction effort

Declarative functional paradigm

- Only function declaration
- Expression result depends on its sub-expressions
- No side effects in functional assessment
- No assignment or control structures
- Support is given to the functions recursive definition
- Functions are used like data
- Map / reduce operations

```
fun invertir (l) =
  [] -> []
  | (p: resto) -> invertir(resto): p
```

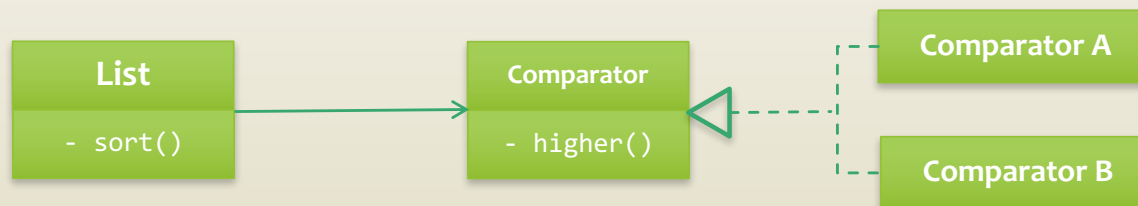
Procesadores del Lenguaje



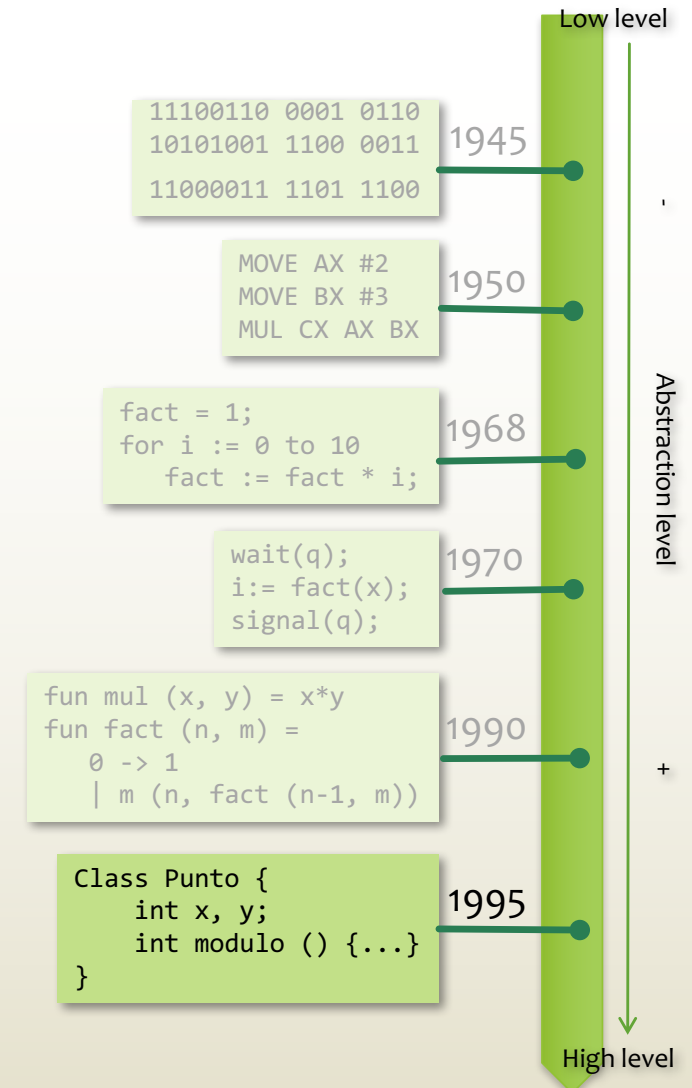
# Abstraction effort

Object-oriented paradigm

- Operations accompanying data structures
- Classes with low coupling and strong cohesion
- The algorithm is distributed in the collaboration between objects
- Inheritance, polymorphism, dynamic binding and genericity
- The objects are managed in the heap



Procesadores del Lenguaje

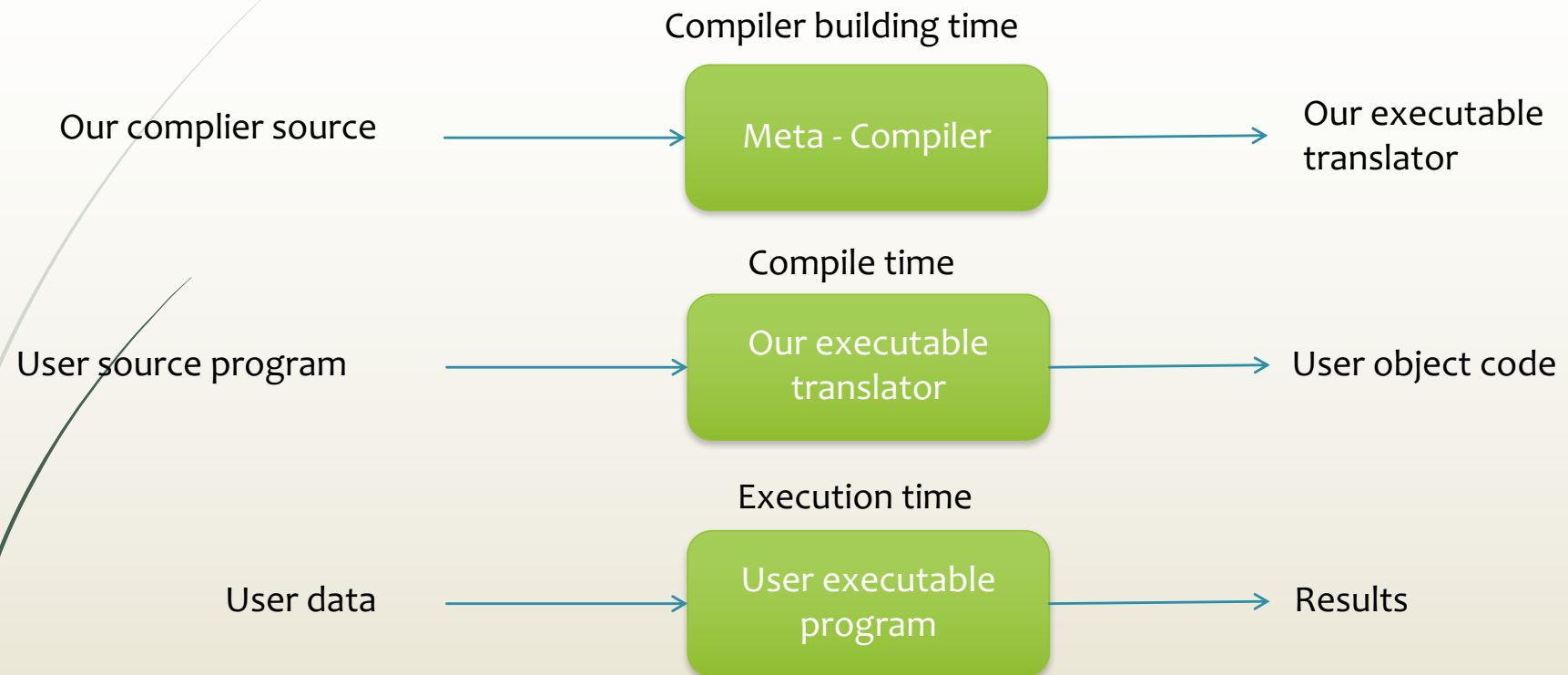


# Translator / compiler concept and types

# Translator concept

- ▶ Translator is a software that:
  - ▶ Translates a **source program** (program instructions in their original form or programming language)
  - ▶ To a **object/machine code** (lower level language)
  - ▶ Maintaining the original meaning
- ▶ We should distinguish between:
  - ▶ Compiler building time
  - ▶ Compile time
  - ▶ Execution time

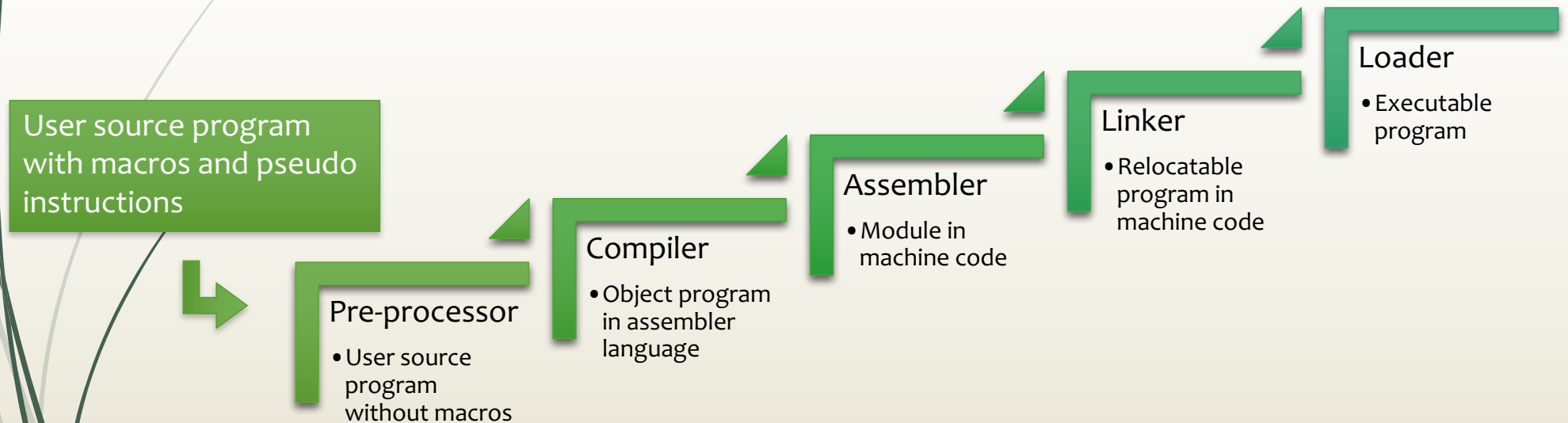
# Translator / compiler concept



# Translator types

- According to source and object languages:
  - Assembler (assembler source, object language in machine code)
  - Compiler (source in high level, object language in low level)
  - Translator (from C++ to C, ...)
- Incremental translators
- 1 o 2 steps translators
- Translator with optimization
- JIT (Just In Time)
- Etc.

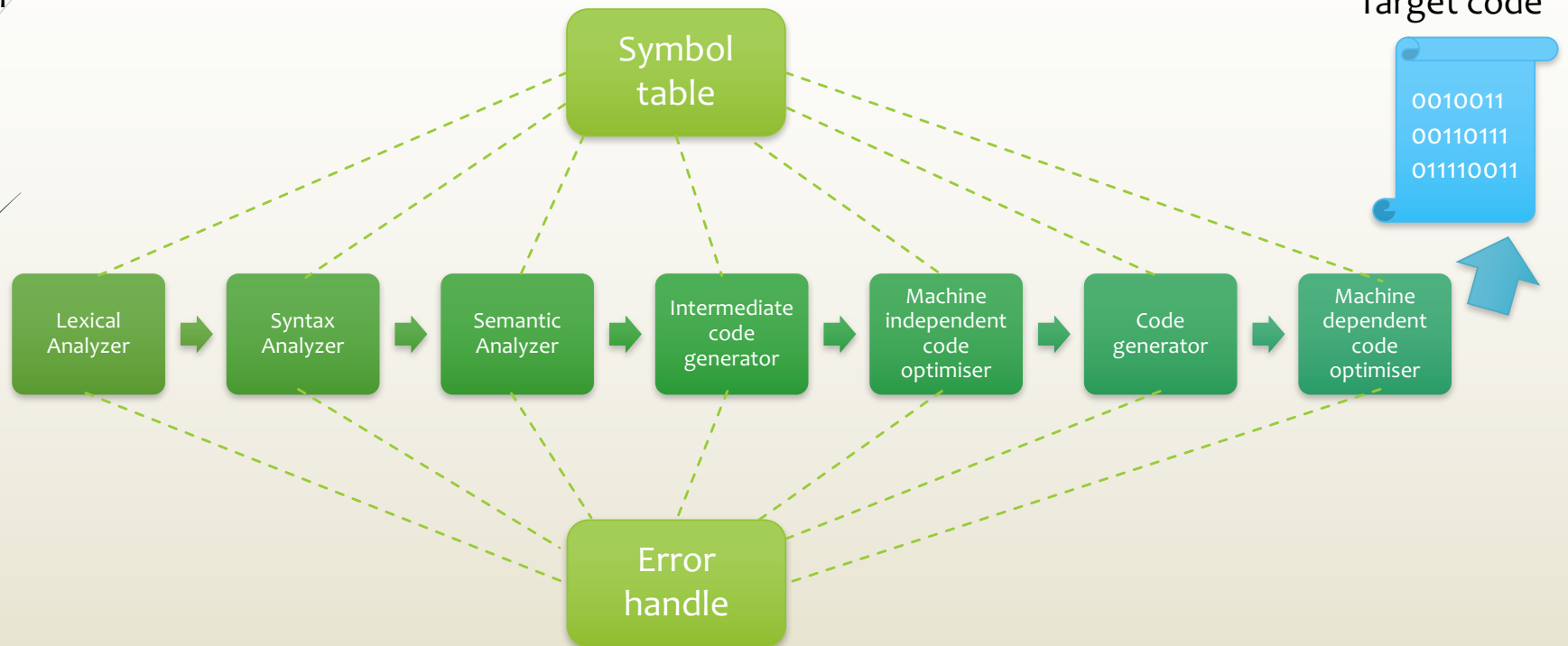
# Compilers execution environment



# Stages of a compiler

Source program

Target code



# Change in the internal representation: lexical analysis

“energy = total = quantity + 23;”

Character string

Lexical Analyser

ID<sub>energy</sub> ASIGN ID<sub>total</sub> ASIGN ID<sub>quantity</sub> + CTEENT<sub>23</sub> ;

Token sequence

Token	Informal description	Lexeme
CTEENT	Decimal digits sequence. If it starts from zero, it is octal	23, 4356, 03472, 0
ID	Letters and digits sequence starting with a letter	Energy, total, quantity, x, y
ASIGN	The character '='	=
+	The character '+'	+

# Lexical description

Token	Informal description	Lexeme
CTEENT	Decimal digits sequence. If it starts from zero, it is octal	<code>digDec</code> <code>[0-9]</code> <code>digOctal</code> <code>[0-7]</code>  <code>{digDec}+ 0{digOctal}+</code>
ID	Letters and digits sequence starting with a letter	<code>[a-zA-Z_][a-zA-Z0-9_]*</code>
ASIGN	The character '='	<code>"="</code>
+	The character '+'	<code>"+"</code>

# Syntax description (grammar, BNF)

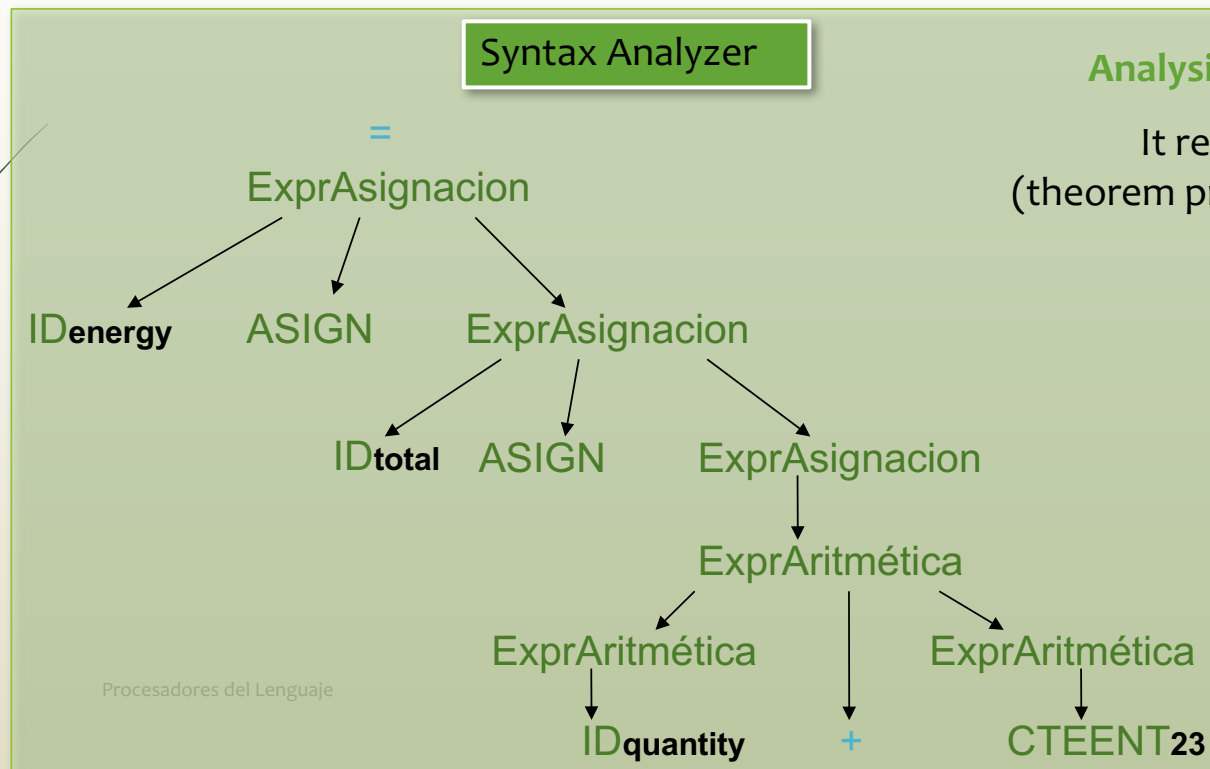
- ▶ `ExprAssign -> ID ASSIGN ExprAssign`
- ▶ `ExprAssign -> ExprAritmetica`
- ▶ `ExprAritmetica -> ExprAritmetica + ExprAritmetica`
- ▶ `ExprAritmetica -> ExprAritmetica - ExprAritmetica`
- ▶ `ExprAritmetica -> ExprFinal`
- ▶ `ExprFinal -> ID`
- ▶ `ExprFinal -> CTEENT`

These rules are independent of the lexemes value!!

# Change in the internal representation: Syntax analysis

`IDenergy ASIGN IDtotal ASIGN IDquantity + CTEENT23 ;`

Token sequence



Analysis tree(or concrete syntax tree)

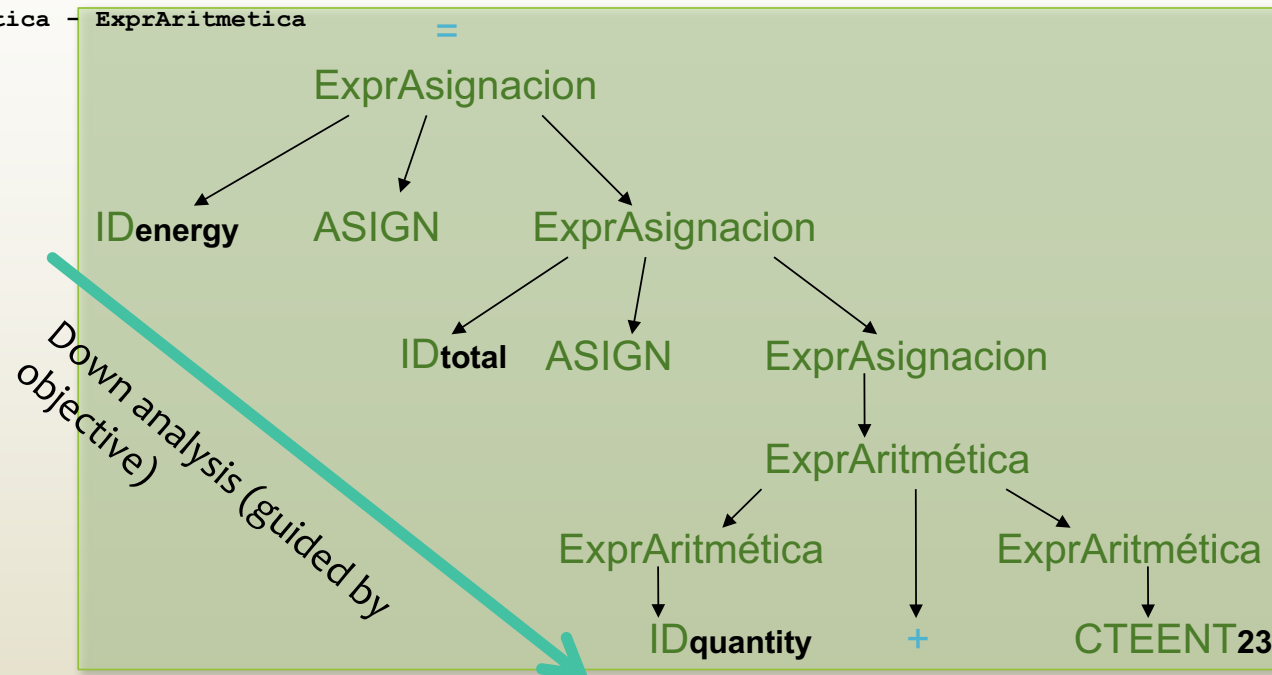
It represents the steps the analyser (theorem prover) to discover the structure of input from your grammar

Symbol table

energy	....
total	....
quantity	....

# Steps during de parsing: Ascending and descending parsing

```
ExprAssign -> ID ASSIGN ExprAssign  
ExprAssign -> ExprAritmetica  
ExprAritmetica -> ExprAritmetica + ExprAritmetica  
ExprAritmetica -> ExprAritmetica - ExprAritmetica  
ExprAritmetica -> ID  
ExprAritmetica -> CTEENT
```



# Change in the internal representation: first intermediate code

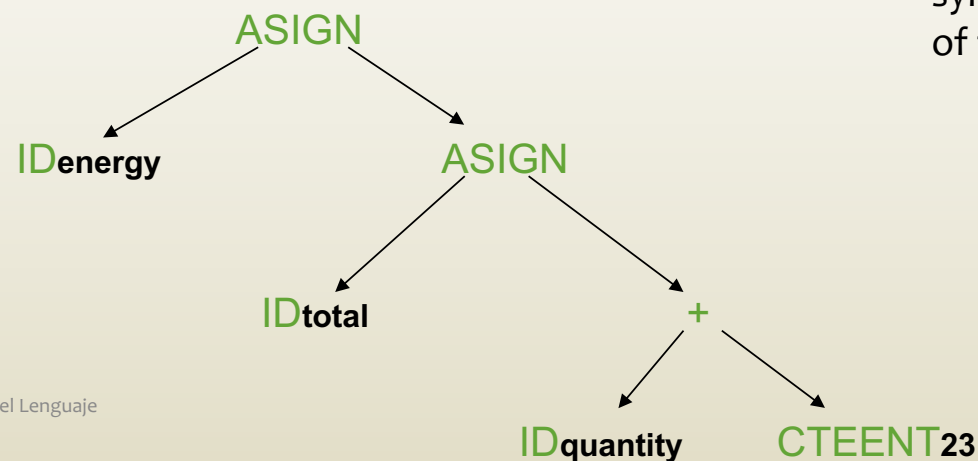
`IDenergy ASIGN IDtotal ASIGN IDquantity + CTEENT23 ;`

Token sequence

Symbols table

energy	....
total	....
quantity	....

Parser that follows the steps of the analysis tree



Syntax tree (or abstract syntax tree)

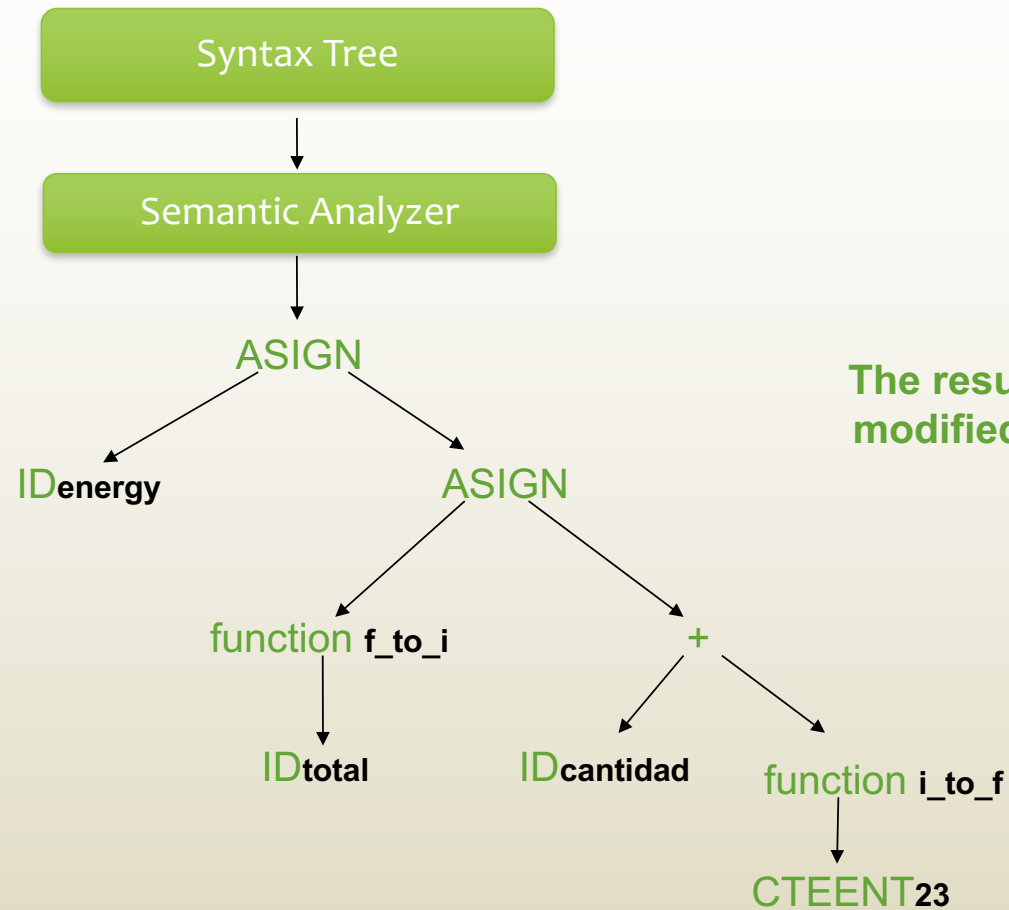
Pointers made structure in memory that represents the syntactic structure without details of the analysis method used

## Change in the internal representation: Semantic analysis (Or contextual constraints)

Symbols table

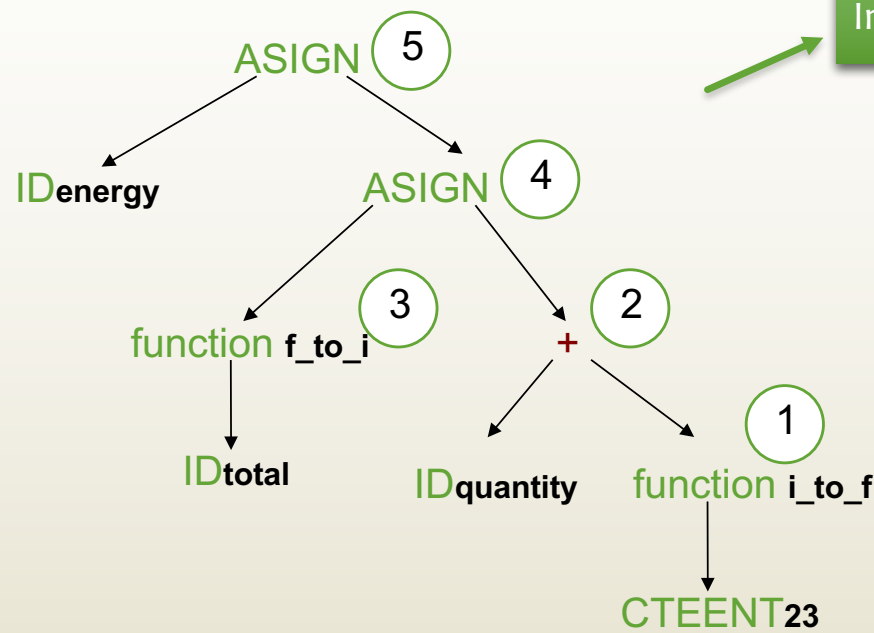
energy	Float
total	Integer
quantity	Float

Procesadores del Lenguaje



The result is another modified syntax tree

# Intermediate code generator: 3-way code



Intermediate code generator

```
Temp1 = i_to_f(23)
Temp2 = quantity + Temp1
Temp4 = Temp2
Temp3 = f_to_i(Temp4)
total = Temp3
Temp5 = Temp4
energy = Temp5
```

# Intermediate code optimizer

- Constant propagation
- Copy propagation
- Algebraic simplifications
- Common subexpression elimination
- Etc.

```
Temp1 = i_to_f(23)
Temp2 = quantity + Temp1
Temp4 = Temp2
Temp3 = f_to_i(Temp4)
total = Temp3
Temp5 = Temp4
energy = Temp5
```

Intermediate code optimizer

```
Temp2 = quantity + 23.0
total = f_to_i(Temp2)
energy = Temp2
```

Temp2 = cantidad + 23.0

total = f\_to\_i(Temp2)

energia = Temp2

Code generator

```
.file "p.c"
.section .rodata
.LC0:
.long 1102577664 # almacenamiento estatico para constante 23.0

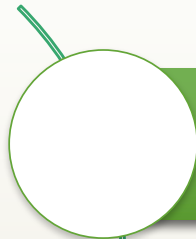
.text
.globl main
.type main, @function
main:
...
flds cantidad # variable "cantidad" a la pila
flds .LC0 # constante "23.0" a la pila
faddp %st, %st(1) # Sumar los elementos de la pila
fstps -8(%ebp) # almacenar el resultado en memoria (T2)

flds -8(%ebp) # Temporal (T2) a la pila
fnstcw -22(%ebp) # salvar palabra control FPU en memoria
movzwl -22(%ebp), %eax
movb $12, %ah # modificar el redondeo por abajo
movw %ax, -24(%ebp) # volver a cambiar la palabra de control de FPU
fldcw -24(%ebp)
fistpl -28(%ebp) # convertir a %st(0) a entero y almacenarlo
fldcw -22(%ebp) # restaurar la palabra de control de FPU
movl -28(%ebp), %eax
movl %eax, total # almacenar resultado en variable "total"

movl -8(%ebp), %eax # Temporal (T2) al registro acumulador
movl %eax, energia # acumulador a la variable "energia"
...

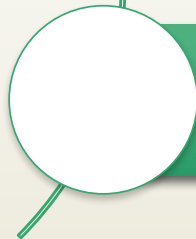
.comm cantidad,4,4 # almacenamiento estático para las variables
.comm energia,4,4
.comm total,4,4
```

## Other phases



Symbol table management

- During the other phases.



Error detection and error messages issues

# Grouping the analysis and synthesis phases

- Sometimes phases of lexical, syntactic and semantic analysis are brought together to form the **front end**(depending on the source language).
- The generation and optimization phases are joined to form the **back end** (language dependent object).

Source program

