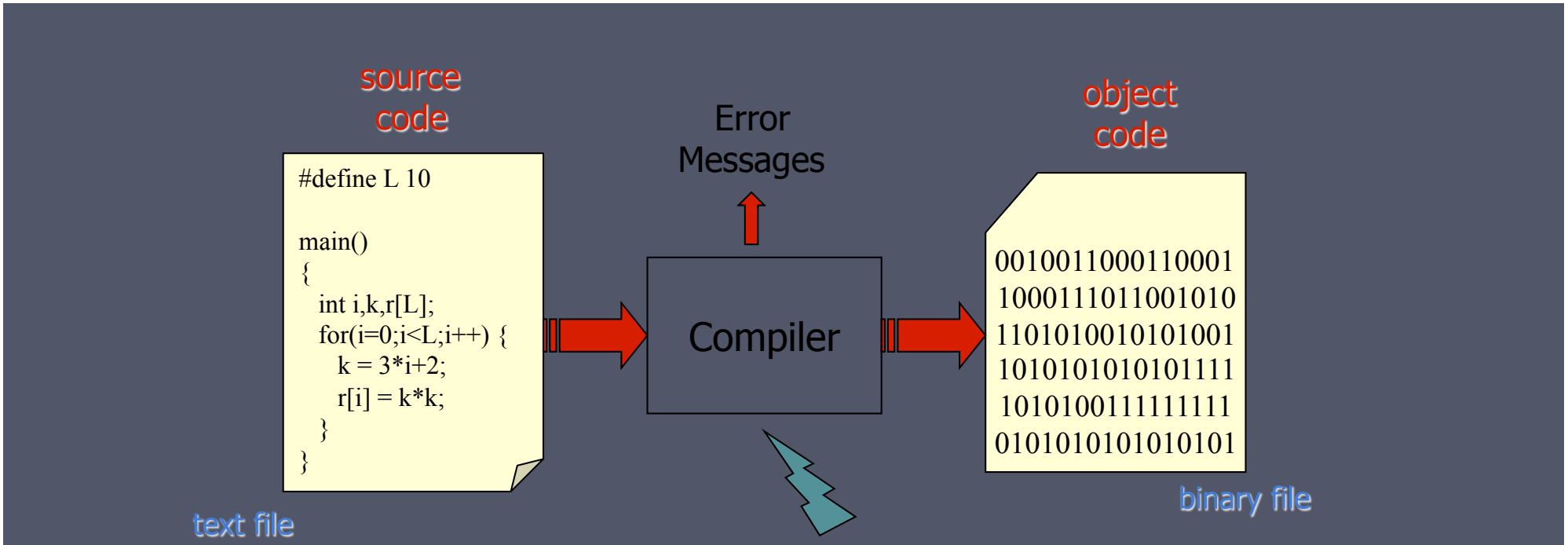



# Languages & Compilers

From source code to executable code

# The compiler

- It is a software program that
  - Translates a program written in a high level programming language into an equivalent object code
  - ..or.. reports the errors present in the source code
- In the '50: development of the first techniques to translate mathematical formulas into machine language
- The first Fortran compiler required 18 man-years of development (1957)
- Systematic techniques for the development of compilers have been devised

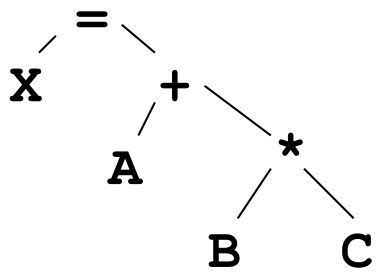
# Source and object languages

- There are… hundreds of programming languages
  - General purpose programming languages
    - C, C++, Pascal, Fortran, Java, Basic, Lisp, Prolog, perl….
  - Special purpose languages
    - Text formatting (Tex, Latex…)
    - Database management and querying (SQL)
- The compiler translates the source language into
  - Another high-level programming language
    - e.g. pascal -> C
  - The machine code for a given processor/architecture

# Parsing and generation

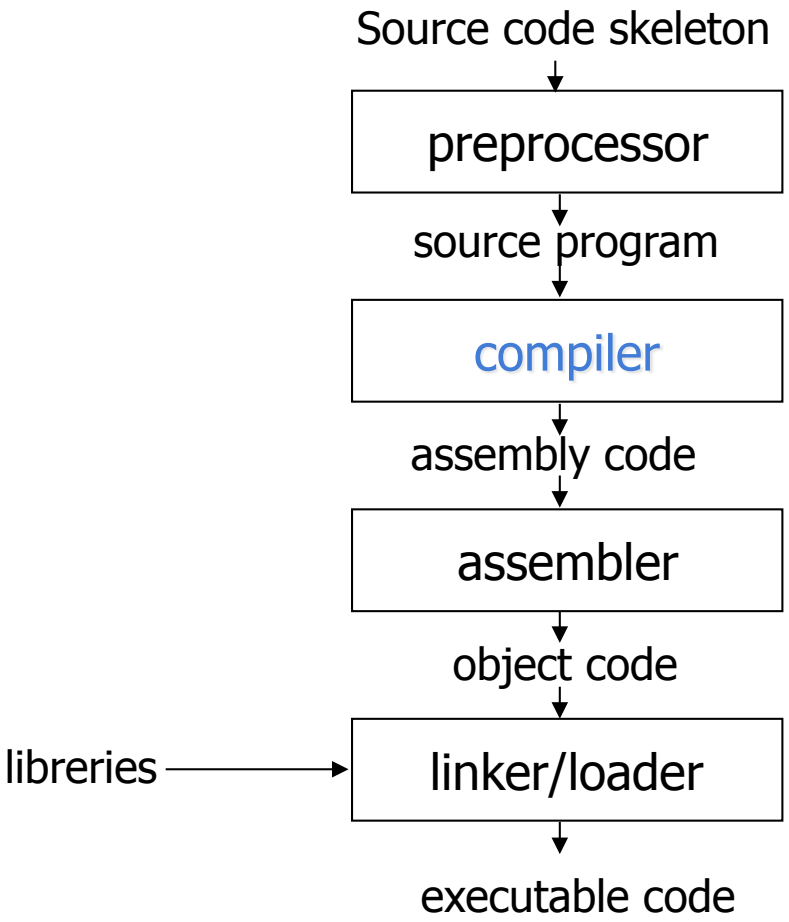
- Parsing
  - The source code is split into its components
  - An intermediate representation of the program structure is built in memory (Syntatic tree)

```
X = A+B*C
```



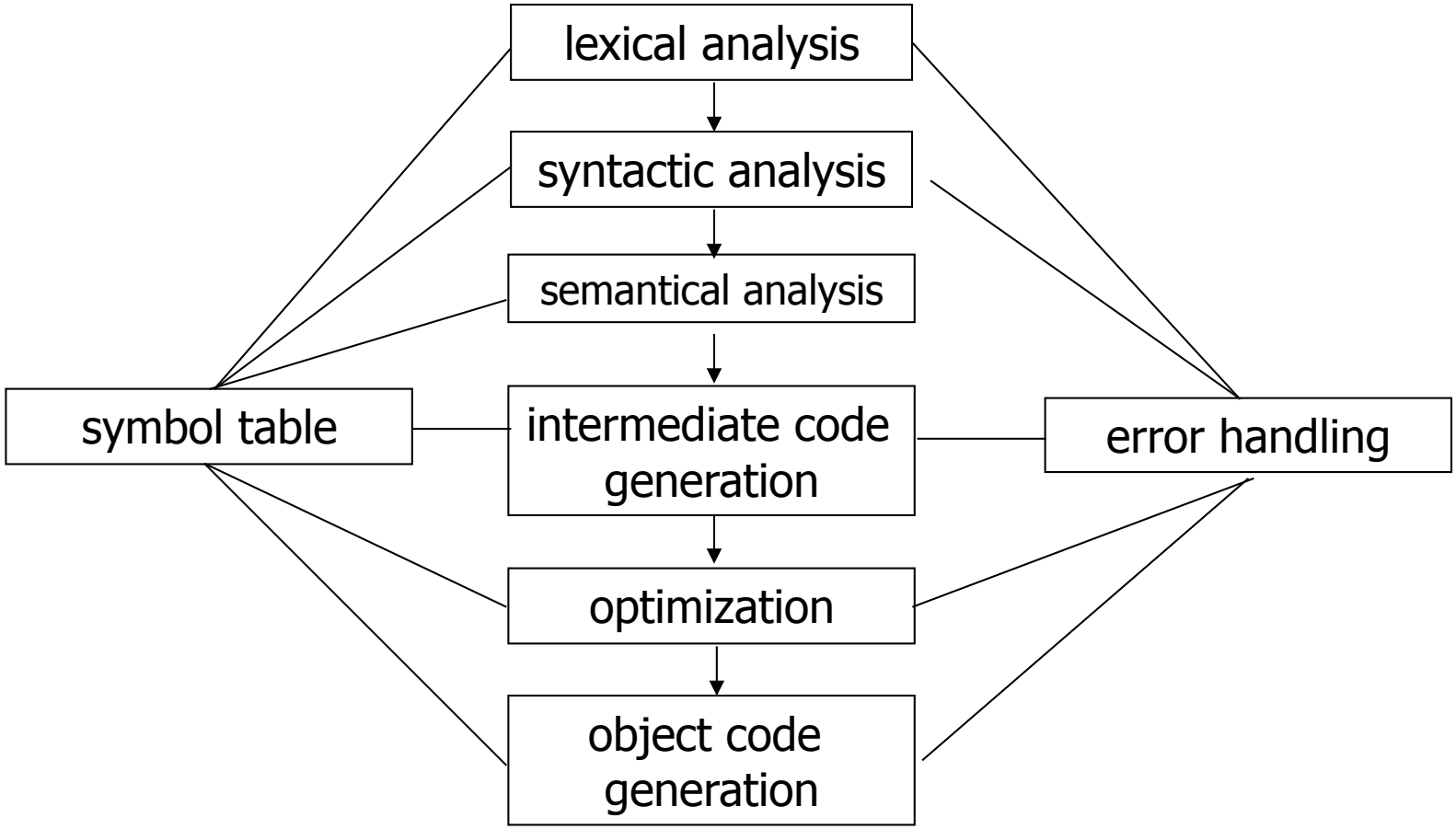

- Generation
  - The object code is obtained from the intermediate representation

# The compiler “context”




include, define (macro)

# Compiler structure



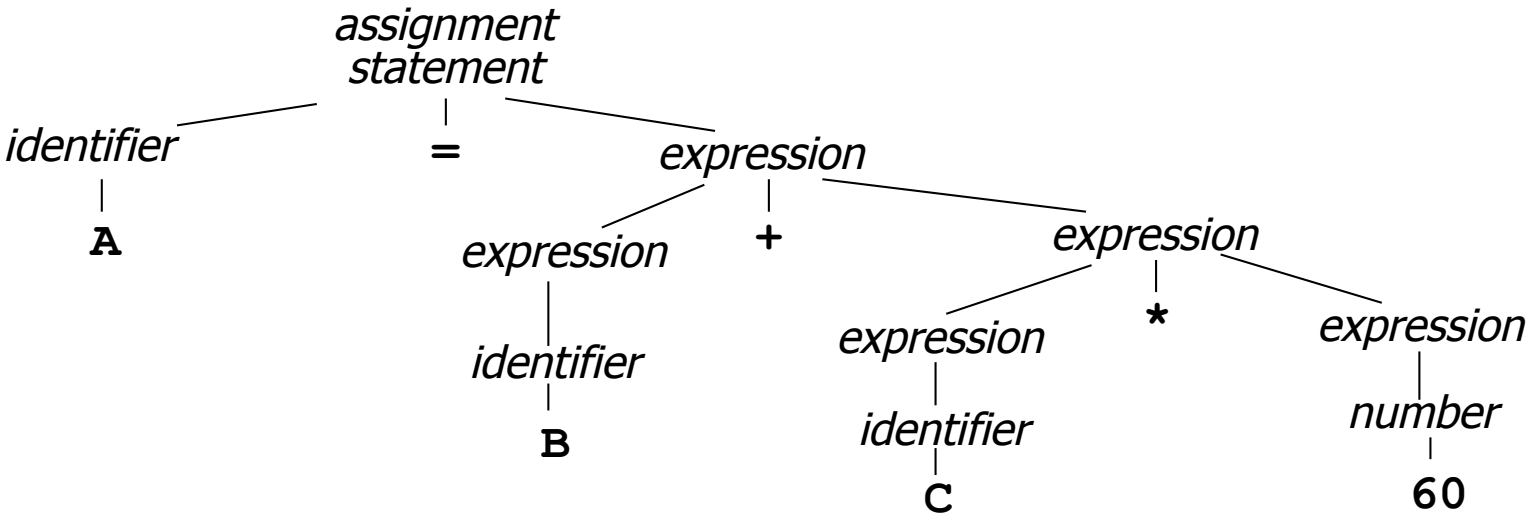
# Lexical analysi (scanning)

- Groups character into words, numbers or symbols
  - The source text is mappend into a sequence of lexical elements (token)
    - language reserved words (keywords) [if - for - while - class ....]
    - user defined identifiers [variable, procedure, function names ...]
    - costants [numbers, strings, ... ]
    - Logic an arithmetic operators [+ * ...]
    - statement separator characters [; , ....]

```
int somma, diff = 0.3;
```

# Syntatic Analyis (parsing)

- Groups tokens into grammatical phrases
  - The syntatic tree represents the program structure
    - leaves contain tokens
    - internal nodes represent syntactical categories



# Syntatic rules

- The hierachical structure of a program is expressed by recursive rules
  - Each *identifier* is an *expression*
  - Each *number* is an *expression*

  } base rules

  - If *expr1* and *expr2* are *expressions* then also
    - *expr1* + *expr2*
    - *expr1* * *expr2*
    - ( *expr1* )

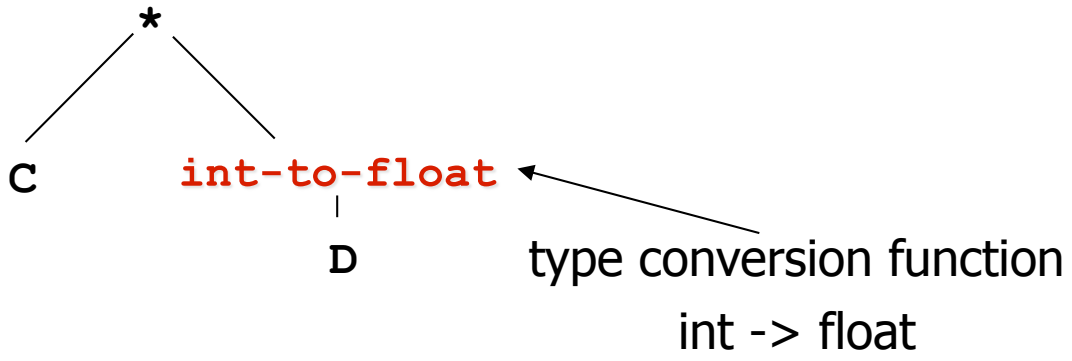
    } recursive rules

  are *expressions*

# Sematical analysis

- Yields the sematics associated to the syntatic structure
  - It verifies that the usage rules of the language are satisfied
    - identifier declarations (e.g. duplicate definitions,…)
    - type check (compatibility of types in expressions, automatic type conversion, type check for vector indexes, ecc..)

```
int D;
float C;
```



# Symbol table

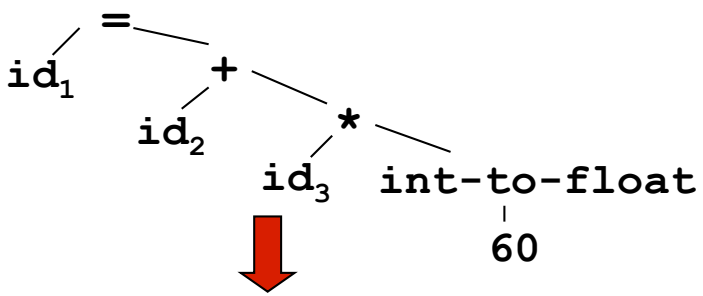
- It memorizes the identifiers and their associated attributes
  - memory allocation
  - type
  - visibility scope
  - number and type of function/procedure arguments

memory allocation

| Name | Type | offset |
|---|---|---|
| A | int | 0 |
| B | float | 4 |
| C | double | 8 |
| I | int | 16 |
| J | int | 20 |

# Object code generation

internal intermediate representation




intermediate code generation

```
t1 = int-to-float(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

code optimization

```
t1 = id3 * 60.0
id1 = id2 + t1
```

object code generation

```
MOVF id3, R2
MULF #60.0, R2
.............
```