Languages & Compilers

From source code to executable code

```
#include L 10

main()
{
    int i,k,r[L];
    for(i=0;i<L;i++)
    {
        k = 3*i+2;
        r[i] = k*k;
    }
}
```
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 *(Syntactic tree)*

\[
X = A + B \times C
\]

• **Generation**
  ▫ The object code is obtained from the intermediate representation
The compiler “context”

Source code skeleton

- preprocessor
  - source program
  - compiler
    - assembly code
      - assembler
        - object code
          - linker/loader
            - libraries
              - executable code

include, define (macro)
Compiler structure

- lexical analysis
- syntactic analysis
- semantical analysis
- intermediate code generation
- optimization
- object code generation
- error handling
- symbol table
Lexical analysis (scanning)

- Groups character into words, numbers or symbols
  - The source text is mapped 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;
  ```
Syntactic Analysis (parsing)

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

```
A = B + C
   |  
   +------>
   |        
   +------>

A = B + C
   |  
   +------>
   |        
   +------>

A = B + C
   |  
   +------>
   |        
   +------>
```

```
A = B + C
   |  
   +------>
   |        
   +------>

A = B + C
   |  
   +------>
   |        
   +------>

A = B + C
   |  
   +------>
   |        
   +------>
```
Syntactic rules

- The hierarchical structure of a program is expressed by **recursive rules**
  
  - Each **identifier** is an **expression**
  - Each **number** is an **expression**
  - If **expr1** and **expr2** are **expressions** then also
    
    - **expr1 + expr2**
    - **expr1 * expr2**
    - *( expr1 )

  **base rules**

  **recursive rules**

  are **expressions**
Sematical analysis

- Yields the semantics associated to the syntactic 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;
```

```
*  
C  
|    
int-to-float
|    
D    

type conversion function
int -> float
```
Symbol table

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>float</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>double</td>
<td>8</td>
</tr>
<tr>
<td>I</td>
<td>int</td>
<td>16</td>
</tr>
<tr>
<td>J</td>
<td>int</td>
<td>20</td>
</tr>
</tbody>
</table>
Object code generation

internal intermediate representation

intermediate code generation

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

t1 = id3 * 60.0
id1 = id2 + t1

code optimization

object code generation

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