

# Compiler / Interpreter

CS 208 - Bucknell University

Spring 2012

# Programming language Implementation

*Question:* What happens between the moment you finish writing your program and the moment you obtain its result ?

# Programming language Implementation

*Question:* What happens between the moment you finish writing your program and the moment you obtain its result ?

There are two main possibilities:

# Programming language Implementation

*Question:* What happens between the moment you finish writing your program and the moment you obtain its result ?

There are two main possibilities:

- **Interpreter** (Perl, Python, etc.).

1-step process:

1. The program is interpreted, its interpretation produces the output.

# Programming language Implementation

*Question:* What happens between the moment you finish writing your program and the moment you obtain its result ?

There are two main possibilities:

- **Interpreter** (Perl, Python, etc.).

1-step process:

1. The program is interpreted, its interpretation produces the output.

- **Compiler** (C, C++, etc.).

2-steps process:

1. The program is compiled into an executable.
2. The executable is run and it produces the output.

# The necessary components of Compiler/Interpreter

The following components are necessary whatever type of implementation a programming language uses (compiler/interpreter):

1. Lexical analysis. Lexer, tokenizer.  
Split the input program into tokens, identify the keywords, the particular symbols of the language, the comments, indentation. The usual tools are Lex, Flex, etc.

# The necessary components of Compiler/Interpreter

The following components are necessary whatever type of implementation a programming language uses (compiler/interpreter):

1. Lexical analysis. Lexer, tokenizer.  
Split the input program into tokens, identify the keywords, the particular symbols of the language, the comments, indentation. The usual tools are Lex, Flex, etc.
2. Syntactical analysis. Parser.  
Translate the Tokenized program into the Abstract Syntax Tree (aka AST). The AST is a Tree like structure for representing a program, nodes are the structure of the language (conditional, loop, function calls), leafs are particular constants, variables. The usual tools are Bison, Yacc.

# The necessary components of Compiler/Interpreter

The following components are necessary whatever type of implementation a programming language uses (compiler/interpreter):

1. Lexical analysis. Lexer, tokenizer.  
Split the input program into tokens, identify the keywords, the particular symbols of the language, the comments, indentation. The usual tools are Lex, Flex, etc.
2. Syntactical analysis. Parser.  
Translate the Tokenized program into the Abstract Syntax Tree (aka AST). The AST is a Tree like structure for representing a program, nodes are the structure of the language (conditional, loop, function calls), leafs are particular constants, variables. The usual tools are Bison, Yacc.
3. Check program validity.  
Check if a variable is declared. Is it the correct **type** ? Is this operation allowed ?

# The necessary components of Compiler/Interpreter

The following components are necessary whatever type of implementation a programming language uses (compiler/interpreter):

1. Lexical analysis. Lexer, tokenizer.  
Split the input program into tokens, identify the keywords, the particular symbols of the language, the comments, indentation. The usual tools are Lex, Flex, etc.
2. Syntactical analysis. Parser.  
Translate the Tokenized program into the Abstract Syntax Tree (aka AST). The AST is a Tree like structure for representing a program, nodes are the structure of the language (conditional, loop, function calls), leafs are particular constants, variables. The usual tools are Bison, Yacc.
3. Check program validity.  
Check if a variable is declared. Is it the correct **type** ? Is this operation allowed ?
4. The valid AST is translated into an output code or directly interpreted.

# 1 Lexical analysis. Tokenizer

A **tokenizer** (also called lexer) transforms a program, considered as a finite sequence of characters, into a sequence of tokens. Tokens are the smallest elements that are considered as units in a program.

# 1 Lexical analysis. Tokenizer

A **tokenizer** (also called lexer) transforms a program, considered as a finite sequence of characters, into a sequence of tokens. Tokens are the smallest elements that are considered as units in a program. Example:

```
/* this is the definition of the factorial function */
int fact(int n){
    int i = 1;
    int r = 1;    // local variable
    while (i <= n) {
        r = r * i;
        i++;
    }
    return r;
}
```

# 1 Lexical analysis. Tokenizer

```
/* this is the definition of factorial function */  
int fact(int n){  
    int i = 1;  
    int r = 1;    // local variable  
    while (i <= n) {  
        r = r * i;  
        i++;  
    }  
    return r;  
}
```

The tokenizer is transforming this program into the following sequence of tokens:

```
int fact ( int n ) { int i = 1 ; int r = 1 ; while  
( i <= n ) { r = r * i ; i ++ ; } return n ; }
```

## 2 Syntactical analysis. Parser

A parser takes as input the program tokenized and construct a tree representing the program, called the abstract syntax tree (AST):  
*drawing on the board*

# Activity

Draw the AST for the following program:

```
int factr(int n){
    if (n <= 0){
        return 1;
    }else{
        return n * factr(n-1);
    }
}
```

## Front-end and Back-end

The architecture of a compiler is separated in two parts: a **front-end** and a **back-end**.

- The front-end is composed of 1 and 2 (Lexer, parser).
- The back-end is composed of 3 and 4 (typechecker, code generation).

## Front-end and Back-end

The architecture of a compiler is separated in two parts: a **front-end** and a **back-end**.

- The front-end is composed of 1 and 2 (Lexer, parser).
- The back-end is composed of 3 and 4 (typechecker, code generation).

The front-end is the *tip of the iceberg*, what the user sees the most directly but also the smaller part. Whereas the back-end is the hidden part, the most important one, invisible to the user.



## Front-end and Back-end

The architecture of a compiler is separated in two parts: a **front-end** and a **back-end**.

- The front-end is composed of 1 and 2 (Lexer, parser).
- The back-end is composed of 3 and 4 (typechecker, code generation).

The front-end is the *tip of the iceberg*, what the user sees the most directly but also the smaller part. Whereas the back-end is the hidden part, the most important one, invisible to the user.



The front-end is well understood (most of the techniques were developed in the 70's), whereas the back-end is constantly evolving, with richer and richer type-systems, new architectures (processors), optimization techniques.

# Match

## *Compiler vs Interpreter*

# Source code protection

Pro compiler: program can be shared safely without providing the source code (if you don't want to reveal the source code). One might be reluctant to publish his source code for several reasons:

- commercial application
- security
- drivers

# Efficiency

- Pro Compiler: You don't do the lexing/parsing everytime you run the program.
- Pro Compiler: you target the assembly language of the machine, you can perform heavy optimisations.

You definitely don't want your OS to be interpreted !

# Portability

- Pro Interpreter: Usually the interpreter is tailoring to match the hardware and OS so the program can be portable and run on any configuration.
- Cons Compiler: Traditionnally compiled language have low level feature that make the program dependent of the architecture (by definition this is non-portable.)

## Java: a mix of compiler/interpreter

- Java **compiles** to a portable low-level code assembly language, called the **Java-bytecode** (not as low level as in C), which is independent of the computer configuration. The bytecode produced is optimized enough.

```
javac ReadTest.java
```

## Java: a mix of compiler/interpreter

- Java **compiles** to a portable low-level code assembly language, called the **Java-bytecode** (not as low level as in C), which is independent of the computer configuration. The bytecode produced is optimized enough.

```
javac ReadTest.java
```

- The Java virtual machine **interprets** the Java-bytecode (avoid the parsing and tokenizing phase).

```
java ReadTest.class
```

## Java: a mix of compiler/interpreter

- Java **compiles** to a portable low-level code assembly language, called the **Java-bytecode** (not as low level as in C), which is independent of the computer configuration. The bytecode produced is optimized enough.

```
javac ReadTest.java
```

- The Java virtual machine **interprets** the Java-bytecode (avoid the parsing and tokenizing phase).

```
java ReadTest.class
```

Advantage: portable and fast.

- Q: Why C is not interpreted ?

- Q: Why C is not interpreted ?
- A: This is not the idea of the language, it requires to access low-level structures, it has to be super efficient.

- Q: Why C is not interpreted ?
- A: This is not the idea of the language, it requires to access low-level structures, it has to be super efficient.
- Q: Why Perl is not compiled ?

- Q: Why C is not interpreted ?
- A: This is not the idea of the language, it requires to access low-level structures, it has to be super efficient.
- Q: Why Perl is not compiled ?
- A: This is not the idea of the language, you should write quick-dirty script, execute them in a click.

# Cross Compilation

Usually a compiler targets the assembly language of the computer it is running on, but not always ! When the compiler produces on purpose an output in another assembly language, this is called **cross-compilation**.

For example, this is the case in the development of embedded systems.

# Bootstrapping

A compiler is said to be bootstrapped if the compiler is written in the programming language it is supposed to compile.

- C is written in C
- OCaml is written in OCaml
- ...
- Perl needs Perl, but the compiler is written in C.