### اللهم علمنا ما ينفعنا، وانفعنا بما علمتنا، وزدنا علما "سُبْحَانَكَ لا عِلْمَ لَنَا إِلَّا مَا عَلَّمْتَنَا إِنَّكَ أَنْتَ الْعَلِيمُ الْحَكِيم"

Lecture 1

* Programing language
* Software running on all the computers was written in some programming language.
* Before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.
* What makes a language successful?

1. Expressive power:

* Writability (Easy creating programs)
* Readability (Easy understanding programs)
* Simplicity

1. Ease of use for the novice المبتدئين.
2. Cost:

* Generate the machine code very fast.
* Machine language:
* It’s the sequence of bits (0’s and 1’s) that directly controls a processor
* Disadvantages:
* Tedious task.
* Not suitable for large programs.
* More error-prone. اكثر عرضه للخطا
* Assembly language:
* It is a symbolic representation of machine code. Convert assembly to machine code by assembler.
* Disadvantages: Machine dependent.
* High level languages:
* Easier to learn. Machine independent.
* Programs written in a high-level language must be translated into machine language by a **compiler or interpreter**. FORTRAN is the first language.
* Compiler: is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language).
* Report any errors in the source program that it detects during the translation process.
* Advantages:
* Very fast program execution
* Interpreter:
* Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.
* Advantages:
* Easy implementation of many source-level debugging operations, because all run-time error messages can refer to source-level units.
* Give better error diagnostics than a compiler, because it executes the source program statement by statement.
* Disadvantages:
* Execution is slower than in compiled systems, because decoding of the high-level language statements are more complex than machine language instructions.
* Java language processors combine compilation and interpretation. A Java source program may first be compiled into an intermediate form called bytecodes.
* The bytecodes are then interpreted by a virtual machine.
* JIT (just-in-time) systems are widely used for java programs. JIT systems are delayed compilers.
* Uses of Compiler Technology:

1. translate a high-level program to object code
2. Optimizations for computer architectures
3. Performance instrumentation
4. Software productivity tool

Lecture 2

* The Analysis-Synthesis Model of Compilation:

1. *Analysis* (**Front-end**)
2. *Synthesis* (**Back-end**)

* Analysis (Front-end)
* determines the operations implied by the source program which are recorded in a tree structure
* Recognises legal and illegal programs and reports errors.
* “Understands” the input program and collects its semantics in an Intermediate Representation. Can be automated.
* **Front end in O(n)**
* Synthesis (Back-end)
* takes the tree structure and translates the operations therein into the target program
* Chooses instructions to implement each IR operation. Translates IR into target code.
* Automation has been less successful.
* Back end in NP-Complete
* All language specific knowledge must be encoded in the front-end
* All target specific knowledge must be encoded in the back-end
* Symbol table
* The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
* Allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.
* Used by all phases of the compiler.
* Lexical Analysis
* The first phase of a compiler is called lexical analysis or scanning.
* The lexical analyzer reads the stream of characters making up the source program and groups them into words (basic unit of syntax (tokens)). Speed is important.
* The output is called token and is a pair of the form ***<type, lexeme>*** or ***<token\_name, attribute>***

1. token-name is an abstract symbol that is used during syntax analysis
2. The second component attribute-value points to an entry in the symbol table for this token.

* Syntax (or syntactic) Analysis (Parsing):
* The second phase of the compiler is syntax analysis or parsing.
* The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
* A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.
* This hierarchical structure is usually expressed by recursive rules. Context-free grammars formalise these recursive rules.
* Abstract Syntax Tree (AST): is a more useful data structure for internal representation.
* Version of the **parse tree** as it summary of grammatical structure without details about its derivation
* ASTs are one form of IR.
* **Lecture 3**
* Lexical analyzer:
* Lexical analyzer allows a translator to handle multi-character constructs like identifiers (sequences of characters), but are treated as units called tokens during syntax analysis.
* The lexical analyzer allows numbers, identifiers, and white space (blanks, tabs, and newlines) to appear within expressions.
* Syntax analyzer: Context free grammar
* If the sentence does not match to the rules (syntax error, there will be no tree)
* Anything on the left side of the grammar is non-terminal
* Semantic Analysis (context handling):
* Output: annotated parse tree (modified intermediate representation)
* Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.

1. Type checking: report error if an operator is applied to an incompatible operand.
2. Check flow-of-controls.
3. Uniqueness or name-related checks.

* Tree evaluation: convert expression from infix to postfix (operand, operand, operator), the process called Syntax direct translation.
* Syntax vs semantics:
* Syntax: describes the proper form of its programs.
* For specifying syntax: context-free grammars or BNF (for Backus-Naur Form).
* Semantic: what its programs mean; that is, what each program does when it executes.
* For specifying semantics: informal descriptions and suggestive examples (much more difficult to describe than the syntax).
* Coercions (casting): type conversions
* If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number, the type checker in the semantic analyzer discovers it.
* Intermediate code generation:
* Translate language-specific constructs in the syntax tree or AST into more general constructs. (معادلات رياضية )
* A criterion for the level of “generality”: it should be straightforward to generate the target code from the intermediate representation chosen.
* This intermediate representation should have two important properties:

1. It should be easy to produce.
2. It should be easy to translate into the target machine.

* Code Optimisation:
* The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code.
* Optimisations can range from:

1. Trivial (constant folding): Constant folding is an optimization technique in which the expressions are calculated beforehand مسبقا to save execution time.
2. Highly sophisticated مطور لغايه (in-lining): inline expansion, or **in-lining**, is a manual or **compiler** optimization that replaces a function call site with the body of the called function.

* Code Generation Phase:
* Map the optimized code onto a linear list of target machine instructions in a symbolic form (target code).
* What is the difficulty in Code Generation Phase?

1. Instruction selection: a pattern matching problem.
2. Register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem.
3. Instruction scheduling: take advantage of multiple functional units (ALU): NP-Complete problem.

* The Grouping of Phases:

1. Analysis (*machine independent* front end):

* The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code.

1. Synthesis (*machine dependent* back end)

* The synthesis phase translates the intermediate code into the target program.