### 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 (3 address code):
* 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.
* Lecture 4
* Two Kinds of Intermediate Representations:

1. Trees, including parse tree and abstract syntax tree.
2. Linear representation (three address code).

* Single pass:
* A single-pass compiler processes the source code in one go, meaning it reads the code from beginning to end, translating it directly into machine code or intermediate code without revisiting any part of the code.
* Requires everything to be defined before being used in source program.
* Advantage:

1. Efficiency: It is faster and requires less memory since it only scans the code once.
2. Simple Structure: Ideal for simpler programming languages and small-scale applications.

* Disadvantage

1. Limited Optimization: It performs limited optimization since it does not analyze the code in multiple stages.

* Multi pass:
* A multi-pass compiler processes the source code in multiple stages or passes, where each pass performs a specific task, such as syntax analysis, semantic analysis, optimization, and code generation.
* compiler may have to keep entire program representation in memory
* Advantage:

1. Better Optimization: Allows for more thorough analysis and optimization of the code as each pass can refine or restructure the code for better performance.
2. Modular Design: Easier to handle complex languages and more sophisticated features.

* Disadvantage

1. Memory-Intensive: Requires more memory and computational resources since it processes the code multiple times.

* Compare between single and multi-pass:

|  |  |  |
| --- | --- | --- |
| Multi-Pass Compiler | Single-Pass Compiler | Aspect |
| Multiple | One | **Number of stages** |
| Slower | Faster | **Speed** |
| Higher | Lower | **Memory Usage** |
| Advanced | Limited | **Optimization** |
| More complex | Simple | **Complexity** |

* Compiler functions:

1. **Lexical Analysis:** Breaks down the source code into tokens.
2. **Syntax Analysis (Parsing):** Analyzes the sequence of tokens to check if they follow the rules of the programming language's grammar and builds a syntax tree.
3. **Semantic Analysis:** Ensures that the syntax tree adheres to the language's semantic rules, such as correct data types.
4. **Intermediate Code Generation:** Converts the syntax tree into an intermediate representation that is easier to optimize and is not dependent on the target machine.
5. **Optimization:** Enhances the intermediate code to make it more efficient, reducing resource consumption and execution time.
6. **Code Generation:** Transforms the optimized intermediate code into machine code or assembly language specific to the target hardware.
7. **Code Linking:** Combines the compiled code with external libraries and modules to create a complete executable program.
8. **Error Handling:** Identifies and reports errors encountered during different stages of compilation, providing useful messages to aid in debugging.

* Compiler-Construction Tools:

1. **Scanner generators:** that produce lexical analyzers from a regular-expression description of the tokens of a language.
2. **Parser generators:** that automatically produce syntax analyzers from a grammatical description of a programming language.
3. **Syntax-directed translation engines:** that produce collections of routines for walking a parse tree and generating intermediate code.
4. **Automatic Code-generator:** that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. **Data-flow analysis engines:** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. **Compiler-construction toolkits:** that provide an integrated set of routines for constructing various phases of a compiler.

* Lexical Analysis:

1. The lexemes of a programming language:

* They are the smallest units that lexical analyzer identified including numeric literals, operators, and special words, among others.

1. Token of a language:

* It is a category of its lexemes.
* It is a pair consisting of a token name and an optional attribute value.
* The token name is an abstract symbol representing a kind of lexical unit.
* Lexical analyzer roles:

1. Identification of lexemes.
2. Stripping out comments and whitespace.
3. Correlating error messages generated by the compiler with the source program.
4. The expansion of macros (preprocessor) may also be performed by the lexical analyzer.

* Lexical analyzer Processes:

1. **Scanning** consists of the simple processes that do not require tokenization of the input, such as deletion of comments and compaction of consecutive whitespace characters into one.
2. **Lexical analysis** proper produces tokens from the output of the scanner.

* Why Lexical Analysis is separated from Parsing:

1. Simplicity of design.
2. Compiler efficiency is improved.

* A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing, specialized buffering techniques for reading input characters can speed up the compiler significantly.

1. Compiler portability is enhanced.

* Input Buffering:
* **input buffering** is a technique used to manage and streamline the reading of source code for lexical analysis
* Two input buffer techniques are used: **buffer pairs** and **Sentinels**.
* Buffer Pairs
* Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
* Using one system read command, we can read N characters into a buffer, rather than using one system call per character.
* If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the **end of the source file.**
* Two pointers to the input are maintained:

1. Pointer lexemeBegin: marks the beginning of the current lexeme, whose extent we are attempting to determine.
2. Pointer forward: scans ahead until a pattern match is found.

* Sentinels:
* **Sentinels** are special markers placed at the end of each buffer half to signal boundaries, making it easier for the lexical analyzer to handle transitions and detect the end of the input stream.
* Natural choice is the character **(eof)**
* Lecture 6
* A Finite State Machine (FSM): is a computational model used to design systems with a finite number of states.

1. States: A finite set of conditions or configurations a system can be in.
2. Alphabet (Inputs): A set of symbols or inputs that trigger transitions between states.
3. Transitions: Rules that define movement from one state to another based on inputs.
4. Start State: The initial state of the FSM.
5. Accept/Final States: Specific states where the system halts or indicates a successful process.
6. Transition Function: A function 𝛿 that maps the current state and input symbol to the next state: **𝛿:𝑄×Σ→𝑄**, where 𝑄 is the set of states and Σ is the input alphabet.

* **Regular expressions = specification**
* **Finite automata = implementation**
* Deterministic Finite Automaton (DFA):

1. For each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

2. For each state and input, there is exactly one possible next state.

3. No ambiguity in transitions.

* Non-Deterministic Finite Automaton (NFA):

1. Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges.

2. Allows multiple transitions for the same state and input.

3. Can include **ε-transitions** (transitions without consuming input).

* Key Limitations of Finite Automata

1. Lack of Memory for Arbitrary Count: Finite Automata cannot count or remember an arbitrary number of inputs since they have no memory other than their states.
2. Inability to Handle Nested Structures: FA cannot handle nested or recursive patterns in strings.
3. Non-Regular Languages.

* Push Down Automata
* Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.
* A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar.
* A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
* A pushdown automaton has three components

1. An input tape: The input tape is divided in many cells or symbols. The input head is read-only and may only move from left to right, one symbol at a time
2. A control unit: The finite control has some pointer which points the current symbol which is to be read.
3. A stack with infinite size: a data structure in which we can push and remove the items from one end only, stores the items temporarily.

* A Pushdown Automata (PDA) can be defined as :

1. Q is the set of states
2. ∑ is the set of input symbols
3. Γ is the set of pushdown symbols (which can be pushed and popped from the stack)
4. q0 is the initial state
5. Z is the initial pushdown symbol (which is initially present in the stack)
6. F is the set of final states
7. δ is a transition function that maps **Q x {Σ ∪ ∈} x Γ into Q x Γ\*.**

* Lecture 7
* A context-free grammar has four components:

1. A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.
2. A set of nonterminal, sometimes called "syntactic variables." Each non- terminal represents a set of strings of terminals, in a manner we shall describe.
3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or nonterminal, called the body or right side of the production
4. A designation of one of the nonterminal as the start symbol.

* Syntax Analysis

1. BNF (Backus Naur Form): is a notation technique for expressing the grammar of a language in a formal.
2. CFG (Context Free Grammar): is a formal system used to define the syntax of programming languages and natural languages.
3. V (Variables or Non-Terminals: Symbols that can be replaced by other symbols.
4. T (Terminals)
5. P (Production Rules): Define how variables can be replaced with combinations of variables and terminals.
6. S (Start Symbol): A special variable from which the derivation begins.

* A meta-language: It is a language that is used to describe another language.
* **BNF is a meta-language for programming languages.**
* Two ways to know that string follow the grammar:

1. Derivations
2. Parsing: Parsing is the problem of taking a string of terminals and showing how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string.

* Ambiguity: A grammar is an ambiguous if this grammar can used to build more than one parse tree for a given string of terminals.
* Two main method to solve grammar ambiguity:

1. Associativity of Operators
2. Precedence of Operators

* Left Recursion: where the leftmost symbol of the body is the same as the nonterminal at the head of the production.
* Lecture 8
* Left factoring: In top-down parsing when it is not clear which production to choose for expansion of a symbol defer the decision till we have seen enough input.
* Parsing

1. Top down parsing: can be viewed as finding a leftmost derivation for an input string.
2. Button up parsing

* There are two types of Top-Down Parser

1. Recursive Descent Parser: Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.

* Recursive Descent Parser depend on back-tracking.

1. Predictive Parser (without backtracking): It is called "predictive" because it uses lookahead tokens to predict which production rule to apply at each step.

* This approach is most effective for LL(1) grammars
* Non Ambiguous Grammar: A Predictive parser cannot be made for ambiguous grammar.
* Non Left Recursive Grammar: Left Recursive Grammar are not suitable for this Parser.
* Left Factoring: Make the grammar deterministic, so the parser can proceed without guessing.
* LL(1) Grammar
* The first L in LL(1) refers to the fact that the input is processed from left to right.
* The second L refers to the fact that LL(1) parsing determines a leftmost derivation for the input string.
* The 1 in parentheses implies that LL(1) parsing uses only one symbol of input to predict the next grammar rule that should be used.
* Parsing table: It helps determine the next action to take based on the current input symbol and the top of the stack.
* First(): If there is a variable (non terminal): if we try to drive all the strings then the beginning Terminal Symbol is called the First.
* Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.
* A Simple Syntax-Directed Translator:
* Syntax Directed translation translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs.
* A translation scheme is a notation for attaching program fragments (semantic actions) to the productions of a grammar.
* The program fragments are executed when the production is used during syntax analysis.
* Using Synthesized Attributes for evaluation.
* Static checking: Static checking in programming and compiler design refers to error detection techniques that happen at compile-time, without executing the code.

1. Syntactic Checking: This checks whether the source code follows the correct syntax as per the language grammar.
2. Type Checking: Type checking verifies that variables and expressions are used consistently according to their types.

* avoiding operations between incompatible types
* Type checking can be static (at compile time) or dynamic (at runtime), but in most statically-typed languages, the checks occur during compilation, improving reliability by catching type errors before execution
* Code optimization: The goal is to improve performance while preserving the program's semantics

1. Machine-independent: These techniques are independent of the target machine's architecture and focus on improving the intermediate code.
2. Constant Folding: Replace constant expressions with their evaluated values at compile time. (int x = 3+5; int x = 8;)
3. Dead Code Elimination: Remove code that does not affect the program's output (unreachable or unused code), (int x=10; x=20;)
4. Common Subexpression Elimination (CSE): Identify and eliminate repeated calculations by reusing their results. (int x= y\*z+3; int c= y\*z+5;)
5. Strength Reduction: Replace expensive operations with computationally cheaper alternatives. (int x= x\*2; int x = x+x;)
6. Loop Optimization: Loop Unrolling: Reduce loop control overhead by increasing the number of operations per iteration.

For(int i=0;i<2;i++)

A[i] = B[i] + C[i];

Replace by:

A[0] = B[0] + C[0];

A[1] = B[1] + C[1];

1. Code Motion: Move computations outside loops if they yield the same result in each iteration.

For(int i=0;i<2;i++){

A[i] = B[i] + C[i];

Int a = b+4;}

1. Inline Expansion: Replace function calls with the function's body to avoid call overhead for small functions.
2. Copy Propagation: Replace variables with their values if possible.

(int a=b; int c= a+y;)

1. Machine-dependent methods: These techniques are tailored to the specific architecture of the target machine, focusing on efficient use of hardware resources.
2. Register Allocation
3. Instruction Scheduling
4. Peephole Optimization: Remove redundant (repeated) instructions.
5. Vectorization: Use SIMD (Single Instruction, Multiple Data) instructions to operate on multiple data points simultaneously.