EDO UNIVERSITY IYAMHO
Department of Computer Science
CSC 312 Compiler Construction

Instructor: *Dr. Bunakiye Japheth*, email:japheth.bunakiye@edouniversity.edu.ng
Lectures: Tuesday, 8am – 12.10 pm, LT1, phone: (+234) 8061324564
Office hours: Wednesday, 2.30 to 3.30 PM (just before class), Office: ICT Floor2 Rm 4

Teaching Assistants: *Mr. Uddin Osemengbe*.

General overview of lecture: This course is intended to give the students a thorough knowledge of compiler design techniques and tools for modern computer programming languages. This course covers advanced topics such as data-flow analysis and control-flow analysis, code generation and program analysis and optimization.

Prerequisites: Students should be familiar with the concepts in theory of computation (*e.g.*, regular sets and context-free grammars); design and analysis of algorithms (*e.g.*, asymptotic complexity, divide and conquer and dynamic-programming techniques); and have strong programming skills using dynamic, pointer-based data structures either in C or C++ programming languages. Students should also be familiar with basic concepts of imperative programming languages such asscoping rules, parameter passing disciplines and recursion.

Learning outcomes: At the completion of this course, students are expected to:

i. to better understand the mathematical foundations of computer science,
ii. to gain experience with source code compilation and interpretation,
iii. to understand the relationships between grammars and parsers,
iv. to understand parsing algorithms and techniques,
v. to understand parse generation software and their activities in Linux and Windows os,
vi. to gain an appreciation for applications of compiler design management

Assignments: We expect to have 5 individual homework assignments throughout the course in addition to a Mid-Term Test and a Final Exam. Home works are due at the beginning of the class on the due date. Home works are organized and structured as preparation for the midterm and final exam, and are meant to be a studying material for both exams. There will also be 3 individual programming projects in this class. The goal of these projects is to have the students experiment with very practical aspects of compiler construction and program analysis.

Grading: We will assign 10% of this class grade to homeworks, 10% for the programming projects, 10% for the mid-term test and 70% for the final exam. The Final exam is comprehensive.

Textbook: The recommended textbook for this class are as stated:

Title: *Compilers: Principles, Techniques and Tools*
Authors: A. Aho, M. Lam, R. Sethi and J. Ullman
Publisher: Addison-Wesley 2nd edition
ISBN-13: 978-0321547989

Title: *Engineering a Compiler*
Authors: Keith Cooper and Linda Torczon,
Publisher: Morgan-Kaufman Publishers
ISBN: 1-558600-698-X

Title: *Advanced Compiler Design and Implementation*
Author: S. Muchnick,
Publisher: Morgan-Kaufmann Publishers
ISBN: 1-558600-320-4

Main Lecture: Below is a description of the contents. We may change the order to accommodate the materials you need for the projects.

Introduction to Compiler Design
Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by computer. The software systems that do this translation are called *compilers.*

Language Processors
Simply stated, a 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; an important role of the compiler is to report any errors in the source program that it detects during the translation process. If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs. An *interpreter* is another common kind of language processor. 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.
The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs. An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.
Example 1: Java language processors combine compilation and interpretation, as shown. A Java source program may first be compiled into an intermediate form called *bytecodes.* The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the byte codes into machine language immediately before they run the intermediate program to process the input.

The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this boxa little, we see that there are two parts to this mapping: analysis and synthesis. The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table,* which is passed along with the intermediate representation to the synthesis part. The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end.* If we examine the compilation process in more detail, we see that it operates as a sequence of *phases,* each of which transforms one representation of the source program to another. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.

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 the characters into meaningful sequences called *lexemes.* For each lexeme, the lexical analyzer produces as output a *token* of the form *(token-name, attribute-value)* that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation. For example, suppose a source program contains the assignment statement p o s i t i o n = i n i t i a l + r a t e * 60

Syntax Analysis

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. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in. This tree shows the order in which the operations in the assignment p o s i t i o n = i n i t i a l + **rate * 60**are to be performed. The tree has an interior node labeled * with **(id, 3)** as its left child and the integer **60** as its right child. The node **(id, 3)** represents the identifier **rate.** The node labeled * makes it explicit that we must first multiply the value of **r a t e** by **60.**

Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is *type checking,* where the compiler checks that each operator has matching operands. For example, many programming

language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

Names, Identifiers, and Variables

Although the terms "name" and "variable," often refer to the same thing, we use them carefully to distinguish between compile-time names and the run-time locations denoted by names. An *identifier* is a string of characters, typically letters or digits, which refers to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not all names are identifiers. Names can also be expressions. For example, the name *x.y* might denote the field *y* of a structure denoted by *x.* Here, *x* and *y* are identifiers, while *x.y* is a name, but not an identifier. Composite names like *x.y* are called *qualified* names. A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable. Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

Procedures, Functions, and Methods

To avoid saying "procedures, functions, or methods," each time we wantto talk about a subprogram that may be called, we shall usually refer to all of them as "procedures." The exception is that when talking explicitly of programs in languages like C that have only functions, we shall refer to them as "functions." Or, if we are discussing a language like Java that has only methods, we shall use that term instead. A function generally returns a value of some type (the "return type"), while a procedure does not return any value. C and similar languages, which have only functions, treat procedures as functions that have a special return type "void," to signify no return value. Object-oriented languages like Java and C + + use the term "methods." These can behave like either functions or procedures, but are associated with a particular class.

Declarations and Definitions

The apparently similar terms "declaration" and "definition" for programming-language concepts are actually quite different. Declarations tell us about the types of things, while definitions tell us about their values. Thus, i n t i is a declaration of *i,* while i = 1 is a definition of *i.* The difference is more significant when we deal with methods or other procedures. In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method. The method is then defined, i.e., the code for executing the method is given, in another place. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

Definition of Grammars

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 *nonterminals,* sometimes called "syntactic variables." Each nonterminal 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

Tokens versus Terminals

In a compiler, the lexical analyzer reads the characters of the source program, groups them into lexically meaningful units called lexemes, and produces as output tokens representing these lexemes. A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names *terminals,* since they appear as terminal symbols in the grammar for a programming language. The attribute value, if present, is a pointer to the symbol table that contains additional information about the token. This additional information is not part of the grammar, so in our discussion of syntax analysis, often we refer to token sand terminals synonymously.

Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar. The ten productions for the nonterminal *digit* allow it to stand for any of the terminals **0 , 1 , . . . ,9.** From production **(2.3),** a single digit by itself is a list. Productions **(2.1)** and **(2.2)** express the rule that any list followed by a plus or minus sign and then another digit makes up a new list.

Tree Terminology

Tree data structures figure prominently in compiling.

• A tree consists of one or more *nodes.* Nodes may have *labels,* which in this book typically will be grammar symbols. When we draw a tree, we often represent the nodes by these labels only.

• Exactly one node is the *root.* All nodes except the root have a unique *parent;* the root has no parent. When we draw trees, we place the parent of a node above that node and draw an edge between them. The root is then the highest (top) node.

• If node *N* is the parent of node *M,* then *M* is a *child* of *N.* The children of one node are called *siblings.* They have an order, *from the left,* and when we draw trees, we order the children of a given node in this manner.