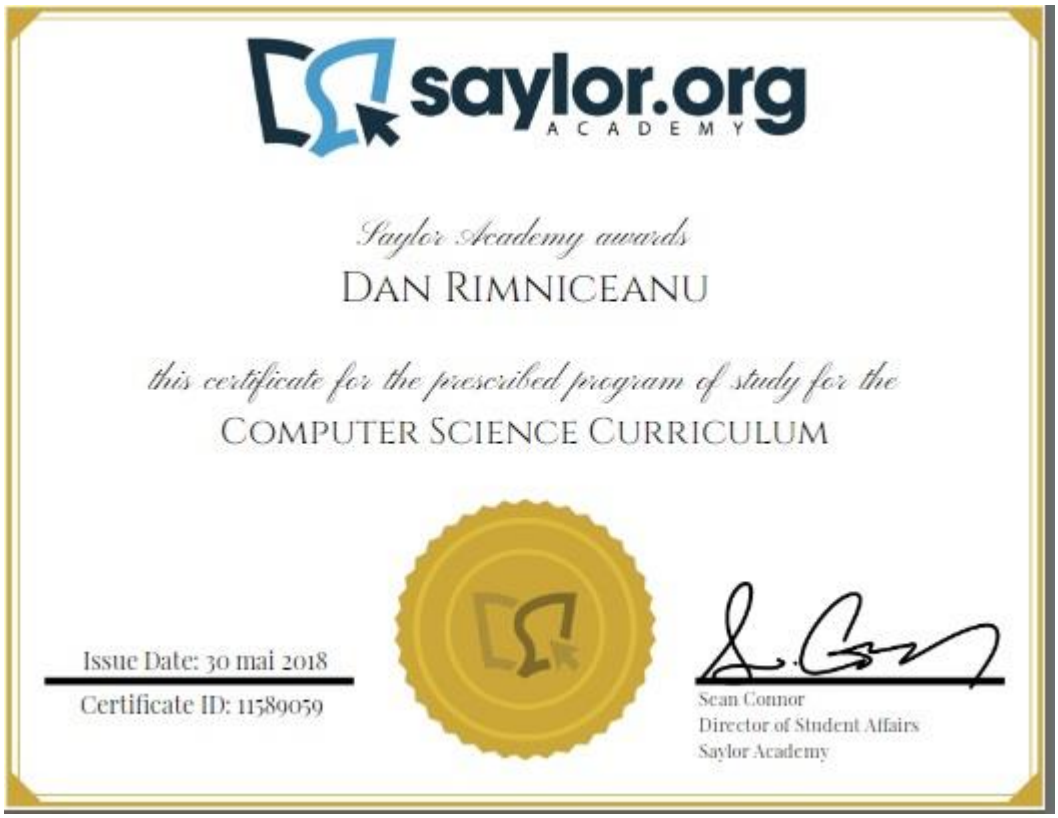
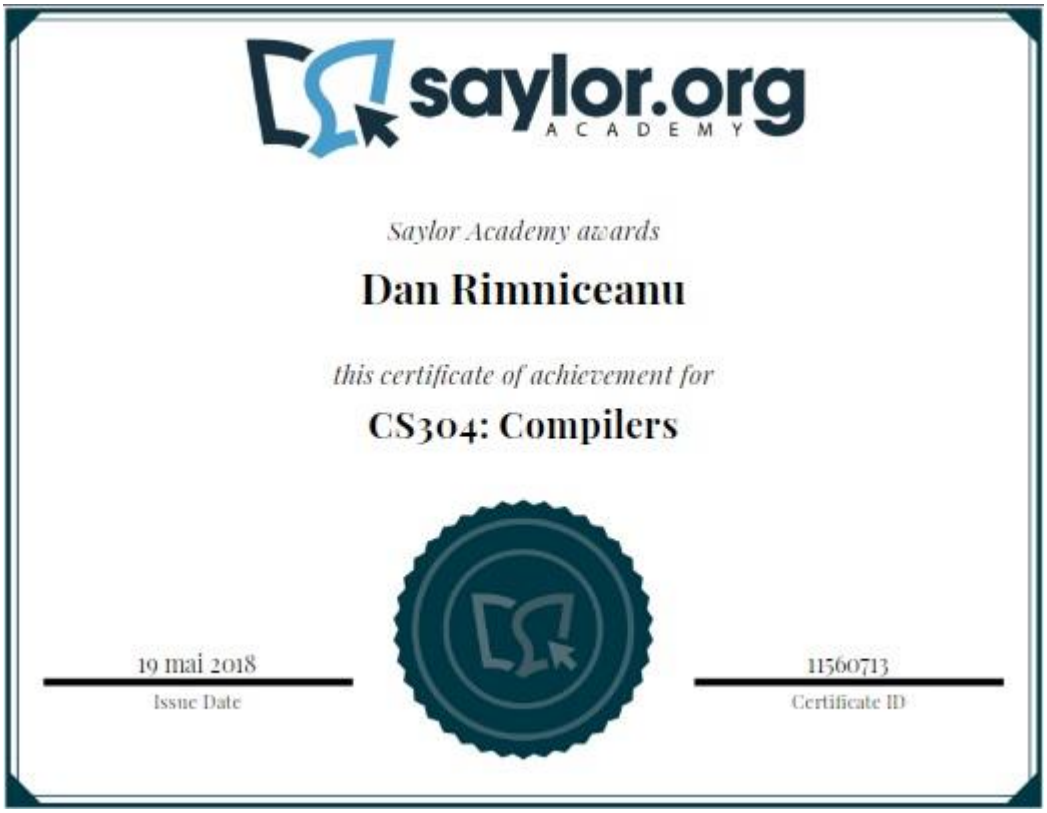

saylor.org
ACADEMY
Saylor Academy awards
DAN RIMNICEANU
this certificate for the prescribed program of study for the
COMPUTER SCIENCE CURRICULUM
Issue Date: 30 mai 2018
Certificate ID: 11589059
Sean Connor
Director of Student Affairs
Saylor Academy


saylor.org
ACADEMY
Saylor Academy awards
Dan Rimniceanu
this certificate of achievement for
CS304: Compilers
19 mai 2018
Issue Date
11560713
Certificate ID

Stanford | ONLINE
STATEMENT OF ACCOMPLISHMENT

June 2, 2019

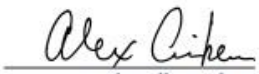
Dan Rimniceanu

has successfully completed a free online offering of

Compilers

with **Programming**.

In order to earn a Programming Statement of Accomplishment, participants were required to score a total score of 70% or higher across 4 programming assignments, 6 quizzes, 1 midterm exam and 1 final exam.



Alex Aiken, Ph.D.
Alcatel-Lucent Professor of Computer Science
Stanford University

PLEASE NOTE: SOME ONLINE COURSES MAY DRAW ON MATERIAL FROM COURSES TAUGHT ON-CAMPUS BUT THEY ARE NOT EQUIVALENT TO ON-CAMPUS COURSES. THIS STATEMENT DOES NOT AFFIRM THAT THIS PARTICIPANT WAS ENROLLED AS A STUDENT AT STANFORD UNIVERSITY IN ANY WAY. IT DOES NOT CONFER A STANFORD UNIVERSITY GRADE, COURSE CREDIT OR DEGREE, AND IT DOES NOT VERIFY THE IDENTITY OF THE PARTICIPANT.

Authenticity can be verified at **https://verify.lagunita.stanford.edu/SOA/7d9d27245a984b69a60976b71dda6dbf**

Stanford | ONLINE
STATEMENT OF ACCOMPLISHMENT

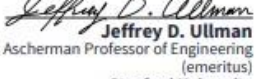
May 30, 2019

Dan Rimniceanu

has successfully completed a free online offering of

Automata Theory

This course covers the theory of automata and languages, including the various forms of finite automata, regular expressions, context-free grammars, Turing machines, undecidability, and NP-completeness. The Statement of Accomplishment is awarded to those scoring at least 50% of the marks, half of which are based on homeworks and half on a timed final exam.



Jeffrey D. Ullman
S. W. Ascherman Professor of Engineering (emeritus)
Stanford University

PLEASE NOTE: SOME ONLINE COURSES MAY DRAW ON MATERIAL FROM COURSES TAUGHT ON-CAMPUS BUT THEY ARE NOT EQUIVALENT TO ON-CAMPUS COURSES. THIS STATEMENT DOES NOT AFFIRM THAT THIS PARTICIPANT WAS ENROLLED AS A STUDENT AT STANFORD UNIVERSITY IN ANY WAY. IT DOES NOT CONFER A STANFORD UNIVERSITY GRADE, COURSE CREDIT OR DEGREE, AND IT DOES NOT VERIFY THE IDENTITY OF THE PARTICIPANT.

Authenticity can be verified at **https://verify.lagunita.stanford.edu/SOA/3a8acf4ea1a949adbd4f00d1614e5d38**

Course Introduction

Because we have compiler programs, software developers often take the process of compilation for granted. However, as a software developer, you should cultivate a solid understanding of how compilers work in order to develop the strongest code possible and fully understand its underlying language. In addition, the compilation process comprises techniques that are applicable to the development of many software applications. As such, this course will introduce you to the compilation process, present foundational topics on formal languages and outline each of the essential compiler steps: scanning, parsing, translation and semantic analysis, code generation, and optimization. By the end of the class, you will have a strong understanding of what it means to compile a program, what happens in the process of translating a higher-level language into a lower-level language, and the applicability of the steps of the compilation process to other applications.

Unit 1: Introduction to Compilers

The compilation process is one of the steps in executing a program. Understanding how compilers work and what goes on "behind the scenes" will help you get better at developing software. This unit will first provide you with an introduction to the compiler, its history, compiler structure and design, and the types of compilers. By the end of this unit, you will be able to describe the steps of the compilation process.

Completing this unit should take you approximately 8 hours.

- Upon successful completion of this unit, you will be able to:
    - Describe the role and applications of compilers.
    - Define compiler, interpreter, and translator, and recount the early history of compilers.
    - Explain the process for development of a compiler in the context of a systems process.
    - Explain the structure of compilers and the corresponding steps in the compilation process.

- ### 1.1: Overview of Compilers
    - A compiler is a complex software system, and, as such, has a system life cycle that starts with a need, transitions to design and construction/implementation, undergoes testing, transitions to use in its intended environment, undergoes maintenance over its life, and ends with "disposal" and archival storage. The activities, various work products, and documentation that comprise the processes used during the system's life must be carefully planned. The part of the system life cycle that deals with the design and construction of the compiler is called the development life cycle. This course focuses on the development life cycle, but the overall system life cycle must be kept in mind.

The term "compiler process" is related to, but different from the system (life cycle) process. The "compiler process" refers to the processing that a compiler does when it performs its function of translation of a source program to a target program. It also refers to the approach taken to design a compiler. Design, in fact, is one activity in the system (life cycle) process.

- 1.1.1: Compiler Definitions, Terminology, and Anatomy of a Compiler
  -  Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Introduction to Computer Language Engineering"URL

    For a definition of a compiler and some terminology, study slides 13-26. For an anatomy of a compiler see slides 27-47. For examples of optimization see slides 48-76. These slides have good examples of compiler output for a given input and a lot of examples of optimizations. A compiler translates a high-level language to a low-level language.
  -  Stanford University: Keith Schwarz's "Introduction to Compilers"URL

    For a slightly different overview of compilers, study slides 8-47. These slides make an analogy between compilation and the equivalence-preserving transformations of an electrical circuit.
  - Wikipedia: "Compilers"URL

    Read the short paragraph on Related Techniques, which defines related terms: language translator, assembler, disassembler, and decompiler.
- 1.1.2: History
  - Wikipedia: "Compilers"URL

    Read the interesting history of the early compilers and computer pioneers, such as Grace Hopper, who worked on the first COBOL compiler.
  -  Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 1: Introduction"URL

    Read sections 1.1, 1.2, and 1.3. Note that "compile" means to translate from a high-level language, used by humans, to a low-level language used by a computer. A high-level language uses concepts, objects, and tasks performed by a human, whereas a low-level or machine language uses concepts, objects, and tasks performed by machines.

    The input language to a compiler, typically a programming language, is called the source language. The output language of a compiler is called the target language or object code, typically an assembly language or machine language.

    The author describes seven phases of a compiler. The middle phase is called Intermediate Code Generation. The intermediate code is independent of a

particular target machine. One of the back-end phases is called Machine Code Generation, which translates the machine-independent code to machine-dependent code for a particular computer.

The need for compilers arises from the need for high-level languages, which are more relevant for problem solving and software development by humans. Moreover, high-level languages are machine independent.

- 1.1.3: Other Applications
  -  Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 1: Introduction"URL

    Read section 1.4, which gives reasons for studying compilers. In the discussion, the author suggests other applications for compiler techniques and tools. He specifically mentions domain-specific languages. Other applications derive from techniques and methods used in compilers. For example, if the input to a software system can be described using a formal grammar, then lexical- and syntax-phase techniques may be the most effective and efficient to use for inputting, verifying, and representing the data for processing. Front-end techniques can also be applied via software tools to the enforcement of development and programming standards and procedures. Knowledge of compiler techniques and methods helps in the design of new programming languages, new hardware architectures, generation of software applications, and the design of software tools.

- 1.1.4: Hardware Compilation
  - Wikipedia: "Compilers"URL

    Read the short paragraph on Hardware Compilation. Note that hardware compilation refers to the translation of a software program to a hardware representation.

- 1.2: Compiler Design Overview
  - The need for compilers arises from the need for high-level languages, which are more relevant for problem solving and software development by humans. Moreover, high-level languages are machine independent.

    Since development of a compiler is a relatively complex system-development effort, typically having many users and developers, and will be maintained over a life of many years, a formal process should be used for its development. The development process should extend from requirements through verification and validation, and should include reviews, tests, analysis and measures, quality assurance, configuration control, and key documentation. The development process is a part of the overall system life cycle process, which additionally, includes deployment, maintenance, disposal and archival storage. The compiler development process should consist of procedures for writing and documenting the needs and requirements, the architecture and design, construction,

integration, and verification and validation of the compiler. Documentation should also include the formal foundations and techniques used, tradeoffs made, alternatives evaluated, and the chosen alternative for design or implementation. Full coverage of all of these in detail is beyond the scope of this course. As we proceed through this course, however, we include high-level needs, requirements, functions, performance considerations, and verification and validation issues for a compiler and its parts.

- 1.2.1: One-Pass vs. Multi-Pass
  - Wikipedia: "Compilers"URL

    Read the paragraph on one-pass and multi-pass compilers. This distinction was more relevant in the early days of computing, when computer memory was limited and processing time was much slower compared to current computers.

    A "pass" refers to reading and processing the input, i.e., source or source representation. A one-pass compiler is simpler to implement. However, more sophisticated processing, such as optimizations, may require multiple passes over the source representation.

    The compiler is composed of components, which perform the steps of the compilation process. Some of the components may make a separate pass over the source representation--thus, the name of multi-pass. Multi-pass compilers are used for translating more complex languages (for example, high-level language to high-level language), and for performing more complete optimizations. Moreover, multi-pass compilers are easier to verify and validate, because testing and proof of correctness can be accomplished for the smaller components. Multi-pass compilers are also used for translation to an assembler or machine language for a theoretical machine, and for translation to an intermediate representation such as bytecode. Bytecode, or pcode (portable code), is a compact encoding that uses a one-byte instruction, and parameters for data or addresses, to represent the results of syntax and semantic analysis. The bytecode or pcode is independent of a particular machine. It is executed by a virtual machine residing on the target machine.

- 1.2.2: Structure of a Compiler
  -  Stanford University: Keith Schwarz's "CS143 Course Overview"URL

    Read the CS143 Course Overview handout. You have studied the lecture "Introduction to Compilers" already. The handout gives an overview of the structure of a compiler with a good explanation.

    This overview describes the front-end process, or analysis stage, of the compilation process. The intermediate code used as an example is TAC, three-address code. The front end, or analysis, consists of lexical analysis, syntax analysis or parsing, semantic analysis, and intermediate code generation. Lexical analysis

may be preceded by preprocessing, depending on the language and the development environment.

It also explains the back-end, or synthesis state, of the compilation process, which includes intermediate code optimization, object code generation, and object code optimization. The symbol table and error-handling topics apply to both the front end and to the back end. The section on one-pass versus multi-pass complements the previous explanation of the number of "passes" that a compiler uses.

The handout also has a very good section on history, which helps us understand why the structure of a compiler is as described in previous sections. Read the Historical Perspective section, which shows the important influence of programming languages on the development of compilers.

Unit 2: Formal Languages and Formal Grammar

Formal languages and formal grammars are the theoretical foundation for computer languages and the compilation process. Formal languages are defined by their grammars, which specify the syntax of the languages.

Completing this unit should take you approximately 6 hours.

- Upon successful completion of this unit, you will be able to:
  - Define a formal language.
  - Give examples of a formal language.
  - Define a formal grammar and explain its purpose.
  - Define syntax and semantics of a formal language.

- 2.1: Motivation for Formal Languages
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Specifying Languages with Regular Expressions and Context-Free Grammars"URL

    Study slides 2 - 8, and slides 38 - 41. Regular and context-free languages are introduced. Also, read slides 68 - 70.

  - Stanford University: Keith Schwarz's "Formal Grammars"URL

    Read pages 10 - 11, which give some history of FORTRAN and ALGOL.

- 2.2: Formal Grammars
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's Lecture 3: "Introduction to Shift-Reduce Parsing"URL

    Study slides 41-67.

  - Stanford University: Keith Schwarz's "Lexical Analysis"URL

Read pages 1 - 10. Formal languages are defined by formal grammars. Regular and context-free grammars are applied in scanning and parsing of programming languages. Formal grammars define the syntax of a formal language.

- 2.3: Syntax of Formal Languages
  - University of California, Berkeley: Paul Hilfinger's "Parsing"URL

    Study the slides.

- 2.4: Semantics of Formal Languages
  - University of California, Berkeley: Paul Hilfinger's "Static Semantics Overview"URL

    Study slides 1 - 7. Note that these slides are associated with the lecture below.

  - University of California, Berkeley: Paul Hilfinger's "Lecture 16"Page

    Watch this video from 12:12 to 15:00, where Professor Hilfinger talks about semantics.

Unit 3: Finite State Machines

Finite state machines (FSM), also called finite state automata (FSA), are conceptual models for recognizing, parsing, and generating strings in a formal language. An FSM can be used to recognize (i.e., determine) whether a string adheres to the syntax of a language. Moreover, an FSM can be used to build a syntax tree, which shows the derivation (i.e., how the string was constructed) of the string. This unit introduces (or reviews) FSMs, which are covered in detail in other courses (for example, CS202: Discrete Structures).

Completing this unit should take you approximately 6 hours.

- Upon successful completion of this unit, you will be able to:
  - Define the types of finite state machines (FSMs) used in the compilation process.
  - Identify the type of language and grammar recognized by each type of FSM.
  - Explain the use of FSA in the compilation process.

- 3.1: Definitions
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Specifying Languages with Regular Expressions and Context-Free Grammars"URL

    Study slides 9 - 33. These slides describe and give examples of regular languages/regular expressions and their corresponding finite state machine.

- University of California, Berkeley: Paul Hilfinger's "Lexical Analysis, Regular Expressions"URL

  Study these slides.

- University of California, Berkeley: Paul Hilfinger's "FSA"URL

  Study slides 1-5.

- 3.2: Some Results and Examples of FSAs
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Specifying Languages with Regular Expressions and Context-Free Grammars"URL

    Study slides 34 - 76. These slides give results and examples of conversion from a non-deterministic finite state automaton (NFA) to a deterministic finite state automaton (DFA); regular expressions or strings; context free grammars and pushdown automata (PDA); and context sensitive grammars and Turing machines. They also introduce parsing and abstract syntax trees.

  - University of California, Berkeley: Paul Hilfinger's "FSA"URL

    Study slides 6 and 7.

- 3.3: Some Applications of FSA
  - University of California, Berkeley: Paul Hilfinger's "FSA"URL

    Study slides 8 to the end. These notes repeat some of the material from section 3.2 and provide additional practice in applying FSAs.

- Unit 4: Scanning and Lexical Analysis

  Lexical analysis is performed by a scanner, one of the front-end components of a compiler. The foundation for lexical analysis is provided by regular grammars and finite state automata. This unit studies scanners and lexical analysis in terms of development process products: requirements, functions, design, construction, and test. The verification of a scanner is done through testing. Validation is based on the programming language specifications, and operation of the scanner as a component of the compiler or application system that uses it.

  Completing this unit should take you approximately 12 hours.

  - Upon successful completion of this unit, you will be able to:
    - Explain scanning and lexical analysis in the context of the compilation process.
    - Define token and lexeme.
    - Specify the functions of a scanner in performing lexical analysis.
    - Describe the handling of blanks, keywords, and comments.
    - Summarize the design and construction of a scanner.
- 4.1: Lexical Analysis Introduction and Overview

- University of California, Berkeley: Paul Hilfinger's "Lecture 2"Page

  Watch from the 4-minute mark to the 30-minute mark, and from the 37-minute mark to the end. This video gives you a glimpse into lexical analysis, which will be studied in more depth in the remaining sections of this unit.

- 4.2: Requirements for a Scanner
  - Stanford University: Keith Schwarz's "Lexical Analysis"URL

    Read the Basics on pages 1-2.

  - Stanford University: Keith Schwarz's "Lexical Analysis Notes"URL

    Read through slide 45.

- 4.3: Review of Regular Expressions, FSAs, and Regular Languages
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 2: Lexical Analysis"URL

    Look over Chapter 2 on Lexical Analysis. If you are comfortable with this material, just review it. If you feel you are somewhat uncomfortable with this material, read or study it to become comfortable with it; it is the foundation for much of our work on compilers.

    A regular expression has associated non-deterministic finite automata (NFA) that accept it.

    The recognition of a regular expression in a regular language can be done in several ways: by operating on the expression directly, using an NFA, or using a DFA. Also, an automaton can be transformed to an equivalent having fewer states. The size (that is, the number of states) and the speed of the automaton determine the most efficient way to recognize a regular expression. Note in section 2.7 of the reading the time estimates for processing an expression by a DFA and by an NFA.

    Regular languages include many languages and can be used for many applications, including scanners. Further, given regular languages, their union, concatenation, repetition, set difference, and intersection are also regular languages--we say that regular languages are closed under these operations. Regular languages can be expressed, equivalently, using regular expressions, NFAs, or DFAs. A key limiting characteristic of regular languages is seen from DFAs. A DFA is a finite automaton, and, thus, can remember only a finite number of symbols. Hence, a DFA cannot recognize a string of the type an b cn, for any n (because for any n it will require an infinite number of states to remember the number of a's). Most computer languages are not regular and we will need to use a larger formal language class to parse them.

- 4.4: Design of a Scanner
  - Stanford University: Keith Schwarz's "Lexical Analysis"URL

    Read from page 3 to the end, including Scanner Implementation 1 and 2, and the FORTRAN I Case Study.

- Stanford University: Keith Schwarz's "Lexical Analysis Notes"URL

  Read slides 46-216.

- 4.5: Construction of a Scanner
  - University of California, Berkeley: Paul Hilfinger's "Lexical Analysis, Regular Expressions"URL

    Read these slides.

  - University of California, Berkeley: Paul Hilfinger's "Lecture 2"Page

    Watch the rest of this lecture.

  - Stanford University: Keith Schwarz's "FLEX in a Nutshell"URL

    Read these notes. FLEX is a scanner generator. It produces a scanner, given a description of the patterns to be identified and actions to take for each token.

  - Stanford University: Keith Schwarz's "Introduction to FLEX"URL

    Read these slides.

  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 2: Lexical Analysis"URL

    Read section 2.9.1. Scanners are usually not written anew, but are generated by tools called scanner (or parser) generators.

  - The Flex Project: Vern Paxson, Will Estes, and John Millaway's The Flex ManualURL

    Review this manual, which contains details on the Flex scanner generator. The link will take you to the introduction section of the manual. Use the links at the top of each page to navigate to the next page of the resource, reading up through section "24 Limitations". For further information, feel free to browse other parts of the manual, including the "Additional Readings" and "FAQ" sections.

- 4.6: Verification and Validation of a Scanner
  - Tom Niemann's "Tutorial on Lex and Yacc"Page

    Scanners for production compilers or software systems must be verified (by proof, demonstration, review, or test, that shows that the requirements, design, and performance specifications are met) and validated (also by proof, demonstration, review, or test that shows that the scanner satisfies the needs of its users and its role in a larger containing system--either compiler or system application). The amount of effort expended in verifying and validating a scanner is dependent on the purpose and intended use of the scanner. For both verification and validation, the description of the input language must be shown to be correct. If the scanner is generated, then the quality of the scanner depends on the reliability and effectiveness of the generator.

In this exercise, you will be introduced to LEX and YACC, which stand for Lexer (short for lexical analyzer) and Yet Another Compiler-Compiler. They are generators, i.e. programs that generate other programs, in this case, a scanner and a parser. Our focus in this Unit 4 Assessment will be on LEX, actually, FLEX - which stands for Fast Lexer. Read this tutorial on LEX; it provides a concise description of LEX and YACC. Then, complete the exercises on this page.

Skip Other students also took...

Other students also took...

Unit 5: Parsing and Syntax Analysis

The next step of the compilation process is parsing. This step also has a foundation in formal languages and automata. Parsing takes input from the Lexical Analysis step and builds a parse tree, which will be used in future steps to develop the machine code. In this unit, we will define parsing and identify its uses. We will also discuss two parsing strategies, Top-Down Parsing and Bottom-Up Parsing, examining what it means to approach parsing from each standpoint and taking a look at an example of each. By the end of the unit, you will understand parsing techniques with regards to compilers, and be able to discuss each of the two main approaches.

Completing this unit should take you approximately 28 hours.

- Upon successful completion of this unit, you will be able to:
  - Explain parsing in the context of the compilation process.
  - Describe several approaches or strategies used for parsing.
  - Specify the functions of a parser in performing syntax analysis.
  - Describe the handling of ambiguities.
  - Summarize the design and construction of a scanner.

- 5.1: Parser Introduction and Overview
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

    Read Chapter 3 through section 3.1.
  - University of California, Berkeley: Paul Hilfinger's "Parsing"URL

    Read these notes. This material overlaps some of the readings later in the class. However, they add additional information and have a practical perspective.

- 5.2: Requirements of a Parser
  - Wikipedia: "Parsing"URL

    Read the following three paragraphs: "Programming languages," "Overview of the process," and "Types of Parsers." Requirements of a parser include: build internal

representation of the input tokens (which come from the scanner); check that the input string of tokens is legal in the language of the compiler (i.e., that it can be derived from the start symbol); and determine the derivation steps for the input string of tokens. In addition, the parser should be reliable, efficient (how efficient depends on the intended use), user friendly (i.e., provide clear, accurate messages), and supportable (assuming that the parser will be used for a long time).

- 5.3: Functions of a Parser
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

    Read subsections 3.4 - 3.6. The functions of a parser include: building an internal representation of the derivation tree and related parser information, and resolving ambiguities of the language pertaining to the input string of tokens.
  - Stanford University: Keith Schwarz's "Top-Down Parsing"URL

    Read slide 4. The first bullet is a requirement statement and the third bullet is a function statement. An additional function is the output of meaningful and accurate messages, including error messages.

- 5.4: Formal Language Considerations
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

    This subsection is just a review of what has been covered in units 2 and 3. Read subsections 3.2 and 3.3. The main idea is that context free languages are used to build efficient parsers, but are supplemented with special techniques to resolve ambiguities.

- 5.5: Design of a Parser

  - 5.5.1: Top-Down Parsers
    - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

      Read sections 3.7 to 3.13.
    - Stanford University: Keith Schwarz's "Top-Down Parsing"URL

      Read these notes, as well as these slides from Lecture 3 and Lecture 4.
    - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Top-Down Parsing"URL

This material overlaps some of the previous readings. However, it presents the material using examples. Scan over the slides, studying those that you feel will benefit you.

- University of California, Berkeley: Paul Hilfinger's "Top-Down Parsers"URL

  This material overlaps previous readings, but provides a practical view. Look over the material and study the parts you feel will benefit you.

- University of California, Berkeley: Paul Hilfinger's "Lecture 7" and "Lecture 8"Page

  Watch both lectures.

- 5.5.2: Bottom-Up Parsers
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

    Read sections 3.14 to 3.18.

  - Stanford University: Keith Schwarz's "Bottom-Up Parsing"URL

    Read the following notes:

    - Notes 10
    - Notes 11
    - Lecture 4
    - Lecture 5
    - Lecture 6

  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Introduction to Shift-Reduce Parsing"URL

    This material overlaps some of the previous readings. However, it presents the material using examples. Scan over the slides, studying those that you feel will benefit you.

  - University of California, Berkeley: Paul Hilfinger's "Earley's Algorithm" and "Bottom-Up Parsing"URL

    This material overlaps previous readings, but provides a practical view. Look over the material and study the parts you feel will benefit you.

    Read these notes, as well as the notes for Lecture 12.

  - University of California, Berkeley: Paul Hilfinger's "Lectures 10-14"Page

    Watch these lectures.

- 5.6: Construction of a Parser

- Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 3: Syntax Analysis"URL

  Read sections 3.15 to the end of Chapter 3.

- Stanford University: Keith Schwarz's "Bottom-Up Parsing"URL

  Read Notes 12, 14, and 16. Read the slides for lectures 5 (264 to the end), 6 (48 to the end), and 7 (91 to the end).

- Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Parse Table Construction"URL

  This material overlaps, but also complements, some of the previous readings. Scan over the slides, studying those that you feel will benefit you. In particular, in the introduction to shift-reduce parsing, study slides 60 to the end.

- 5.7: Verification and Validation of a Parser
  - Tom Niemann's "Tutorial on LEX and YAC"Page

    In this exercise, you will be introduced to LEX and YACC, which stand for Lexer (short for lexical analyzer) and Yet Another Compiler-Compiler. They are generators, i.e. programs that generate other programs, in this case, a scanner and a parser. Our focus in this Assessment will be on YACC. Read the part of the tutorial on YACC. This tutorial explains YACC and how YACC and LEX interface. LEX and YACC are the original programs, and just as Flex is an open software version for LEX, Bison is an open software version for YACC.

Unit 6: Syntax Directed Translation and Semantic Analysis

Semantic Analysis takes input from the parsing process and prepares the code for the code-generation step. In this unit, we will discuss this process in detail, learning about scope and type-checking, type expression, type equations, and type inference.

Completing this unit should take you approximately 33 hours.

- Upon successful completion of this unit, you will be able to:
  - Explain semantic analysis in the context of the compilation process.
  - Describe scope checking and type checking.
  - Specify the functions of semantic analysis.
  - Solve type equations and make inferences in a type calculus.

- 6.1: Syntax-Directed Translation and Attribute Grammars
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 5: Interpretation"URL

Read Chapter 5 on interpretation, where execution of a program takes place as the derivation is produced.

- Stanford University: Keith Schwarz's "Syntax-Directed Translation"URL

  Study the definitions and examples. Syntax-directed translation and attribute grammars are techniques for using the parser to drive the translation directly. Attributes are properties of grammar symbols, and the attributes take on values. Rules associated with each grammar production specify how to compute the value of the attributes.

- 6.2: Intermediate Representation
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Intermediate Formats"URL

    Study the slides on intermediate representations. Data, as well as computations and flow of control, have intermediate representations.

  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 4: Scopes and Symbol Tables"URL

    Read Chapter 4 on scope of names and symbol tables.

- 6.3: Functions of Semantic Analysis
  - 6.3.1: Scope Checking of Names in a Program
    - Stanford University: Keith Schwarz's "Semantic Analysis"URL

      Read the notes and the slides for an overview of semantic analysis.

  - 6.3.2: Static vs. Dynamic Scope Checking
    - University of California, Berkeley: Paul Hilfinger's "Static Semantics Overview"URL

      Read these slides.

  - 6.3.3: Type Checking
    - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 6: Type Checking"URL

      Read Chapter 6, which presents an overview of type checking, nicely organized: symbol table environment, type checking for expressions, functions, and then, for programs.

- 6.3.3.1: Type Expressions, Type Equivalence, Type Inference, and What to Check

- Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Semantic Analysis"URL

  Study the slides on type systems and what to check for when building intermediate representations for various language constructs.

- University of California, Berkeley: Paul Hilfinger's "Types"URL

  Read slides 2 through 14 and use them as a review or supplement to the above readings. Slides 15 through 31 give examples of type inference for the languages Prolog, Java, and Python.

- University of California, Berkeley: Paul Hilfinger's "Lectures 18-21"Page

  These lectures correspond to the notes and are optional. They may be helpful in understanding some of the notes.

- 6.3.3.2: Type Systems as Proof Systems-Type Checking as Proofs
  - Stanford University: Keith Schwarz's "Type Checking"URL

    Study the very interesting presentation on type checking by proofs.

  - University of California, Berkeley: Paul Hilfinger's "Type Inference and Unification"URL

    Read slides 1 through 8. The slides are somewhat formal and present a "type calculus." Use these slides to add to your understanding of types.

  - University of California, Berkeley: Paul Hilfinger's "Lecture 22"Page

    This video corresponds to these notes and is optional. If it adds to the notes and helps you, watch it.

- 6.3.3.3: Applications of Type Proofs
  - Stanford University: Keith Schwarz's "Type Checking II"URL

    Study the application of the type proof system (introduced above) to the detection of type errors.

- 6.3.3.4: Type Equations, Unification and Binding of Type Expressions
  - University of California, Berkeley: Paul Hilfinger's "Type Inference and Unification"URL

    Study slides 8 through 19. The slides supplement the readings above with type examples. A binding is a substitution of a type expression for a type variable.

- 6.4: Verification and Validation of Semantic Analysis

- Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 6: Type Checking"URL

  Complete exercises 6.1 and 6.2 on pages 143 and 144.

  These exercises will give you some practice with semantic analysis and are based on Chapter 6, which covers type checking. Then check your answers against the Answer Key.

Unit 7: Runtime Environment

Runtime environment considerations include organization of the compiled program, storage, building blocks of a compiled program, and different runtime configurations. This unit has some overlap with Unit 8. The emphasis in this unit is on representation of needed data structures and techniques for objects and inheritance. The emphasis of Unit 8 is on instruction generation.

Completing this unit should take you approximately 14 hours.

- Upon successful completion of this unit, you will be able to:
  - Describe the role of intermediate representation and runtime environments in the compilation process.
  - Explain the encoding of data structures in runtime memory.
  - Explain the encoding of procedures, functions, and parameters at runtime.
  - Explain how objects, inheritance, and exceptions are encoded at runtime.

- 7.1: Overview of Runtime Environments
  - Stanford University: Keith Schwarz's "Runtime Environments"URL

    Read this handout, which discusses data representations and their organization in memory for a program. After semantic analysis, intermediate representations are encoded. This is the last step of the "front-end" of the compilation process. The relationship of front ends to back ends can be many-to-one or one-to-many. In the former case, a single back end is used for several languages, each handled by its own front end. In the latter case, one front end handles the input, source language, and the back end is used for several target machines, each having its own back end.

  - Stanford University: Keith Schwarz's "Runtime Environments Slides"URL

    Read the functions of IR generation on slides 6 and 7. Some requirements for IR generation are on slide 8. Encoding of primitive types and arrays are discussed on slides 19 through 37 for additional coverage.

    Scope of a variable is the lexical area of a program in which the variable can be used. Extent, or lifetime, is the period of time that a variable exists.

See slides 38 through 118 for a discussion of the stack, activation trees, closure and coroutines, and parameter passing.

- University of California, Berkeley: Paul Hilfinger's "Introduction to Runtime Organization"URL

  Read these notes. In this presentation, IR is treated as part of code generation, and it has a lot of detail on the run-time encoding for procedures and functions.

- University of California, Berkeley: Paul Hilfinger's "Lectures 25-27"Page

  Watch these lectures

- 7.2: More Complicated Representations: Objects and Inheritance
  - Stanford University: Keith Schwarz's "Runtime Environments II"URL

    Read these slides, which describe the encoding of objects, inheritance, and dynamic type checking, including difficulties and solutions.

- 7.3: Encoding of Exceptions and OOP (Object-Oriented Programs)
  - University of California, Berkeley: Paul Hilfinger's "Exceptions, OOP"URL

    Read these notes.

  - University of California, Berkeley: Paul Hilfinger's "Lectures 28-30"Page

    Scan over these and watch the parts that will benefit you.

Unit 8: Code Generation

This unit is closely related to Unit 7, where the emphasis was on representation of data structures needed for run-time. While there will be some overlap, the emphasis in this unit is on instruction-level intermediate code generation and from intermediate code to target code.

The last phase (or next to the last phase if there is a code optimization phase) of the compilation process is code generation, where the output from the previous steps is finally translated into machine code, ready to execute on the target platform. In this unit, we will start with a discussion of code generation in general before moving on to a more detailed description of the code generation process. This will include an in-depth discussion of three main areas: Instruction Selection, Instruction Scheduling, and Register Allocation. By the end of this unit, you will have a firm understanding of the code generation process.

Completing this unit should take you approximately 17 hours.

- Upon successful completion of this unit, you will be able to:

- Explain the use of an intermediate language.
- Identify the difficult aspects of code generation.
- Give examples of translation from simple source statements to intermediate code.
- Give examples of translation from intermediate language statements to assembler or machine code.

- 8.1: Introduction to Intermediate Code Generation
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 7: Intermediate-Code Generation"URL

    Read Chapter 7, which discusses generation to a relatively low-level intermediate language. Section 7.9 overlaps some of the previous readings.
  - Stanford University: Keith Schwarz's "Intermediate Representation"URL

    Intermediate representation of a source program may be part of the front end, may be part of the back end, depending on the design of the compiler and its intended use, or may be simply called the "middle part" of the compiler between the front end and back end. Handout 23 covers IR in the context of code generation.

- 8.2: Detailed Example: Three Address Code (TAC)
  - Stanford University: Keith Schwarz's "TAC Examples"URL

    TAC is a generic assembly language. Read this handout, and then the associated slides. Slides 1 through 149 give examples of TAC statements, function calls and encoding of objects.

- 8.3: Additional Intermediate Language Generation Examples
  - University of California, Berkeley: Paul Hilfinger's "Code Generation"URL

    These notes discuss generation of intermediate code for a stack machine, stack machine with accumulator, and for a TAC machine.

  - University of California, Berkeley: Paul Hilfinger's "Lecture 31"Page

    This lecture is optional, and is listed as an aid if you have any questions on the notes.

- 8.4: Generation of Machine Code
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 8: Machine-Code Generation"URL

    Read Chapter 8. It describes the translation from a low-level intermediate language to machine code. This translation addresses the handling of restrictions

on the machine language; for example, a finite number of registers is a restriction of a target machine and its machine language. The intermediate language assumes an unlimited number of registers, i.e., virtual register machine.

- University of California, Berkeley: Paul Hilfinger's "Registers, Functions, Parameters"URL

  Read these notes on generation of machine code from intermediate code.

- Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 9 and Chapter 10"URL

  Read Chapters 9 and 10. Chapter 9 overlaps the above reading, and is a supplementary discussion of the problem of mapping a large number of variables used in an intermediate language translation into a smaller number of registers, plus memory locations available on the target machine. Chapter 10 covers function calls using a stack, activation records, and frame pointers. Look over these chapters and read the parts that will add to your understanding of register allocation and function calls.

- University of California, Berkeley: Paul Hilfinger's "Lectures 32-34"Page

  Watch these videos.

- Stanford University: Keith Schwarz's "Register Allocation" and "Garbage Collection"URL

  Read these notes, as well as the ones on Garbage Collection. These are very well-done formal presentations and give a lot of detail. Register allocation, linear scan and Chaitin's algorithm are explained. Regarding garbage collection, reference counting, mark-and-sweep, and stop-and-copy are explained.

- 8.5: Verification and Validation of Code Generation
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 7: Intermediate-Code Generation"URL

    Do Exercise 7.1 on page 173 and Exercise 8.2 on page 189.

    These exercises will give you some practice with code generation and are based on Chapter 7, which covers intermediate code generation, and Chapter 8, which covers machine code generation. Intermediate code can be high level, i.e. close to the input language, or low level, i.e. close to the target machine language, and, hence, though Exercise #2 jumps to Chapter 8, the process of syntax directed translation is the same. These exercises illustrate the process for syntax directed translation to intermediate code or machine code. You can check your answers against the Answer Key.

Unit 9: Code Optimization

Simply compiling and executing a program is not enough to get the most out of your code. It is the optimization process that allows your code to run as effectively and efficiently as possible. In this unit, we will first take a look at optimization, learning what it is and why we are interested in it. Next, we will review different optimization categories, including Peephole, Local, Loop, Language Dependent, and Machine Dependent. We will conclude with a discussion of different optimization techniques. By the end of this unit, you will have a basic understanding of a wide range of optimization techniques and how they improve the effectiveness of your program.

Completing this unit should take you approximately 22 hours.

- Upon successful completion of this unit, you will be able to:
  - Define optimization.
  - Describe approaches and give examples of local optimizations.
  - Describe approaches and give examples of global optimizations.
  - Describe approaches and give examples of code optimizations.

- 9.1: The What and Why of Code Optimizations
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 11: Analysis and Optimisation"URL

    Read Chapter 11 through section 11.1.

- 9.2: Fundamentals of Code Optimization
  - University of California, Berkeley: Paul Hilfinger's "Local Optimization"URL

    Read slides 1 through 8.
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Introduction to Program Analysis and Optimization"URL

    Read these slides.

- 9.3: Local Intermediate Code Optimizations: Definitions and Examples
  - Stanford University: Keith Schwarz's "IR Optimization"URL

    Read pages 1 through 10 of the handout, and the associated lecture notes, which are an excellent unified formal treatment of the topic.

- 9.4: Global Intermediate Code Optimizations: Definitions and Examples
  - University of California, Berkeley: Paul Hilfinger's "Global Optimization"URL

    Read these slides. Global optimization uses forward analysis (e.g., constant propagation), which moves information forward, and backward analysis (e.g., liveness), which moves information backward. Note slide 5, which states that dynamic properties of a program are undecidable.

- Stanford University: Keith Schwarz's "Code Optimization", "Global Optimization", and "Global Optimization II"URL
  - Code Optimization
  - Global Optimization
  - Global Optimization II

  Read these pages, which cover the material in more detail and present it in a unified formal way using semilattices.
- Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Introduction to Dataflow Analysis" and "Foundations of Dataflow Analysis"URL
  - Introduction to Dataflow Analysis
  - Foundations of Dataflow Analysis

  Read these slides. They repeat some of the material from above readings. However, they are an excellent review, and go into detail on the formalism that underlies global optimizations.

- 9.5: Code Optimization
  - Massachusetts Institute of Technology: S. Amarasinghe and M. Rinard's "Notes for Lectures 13-18"URL
    - Introduction to code optimization: instruction scheduling
    - Loop optimization: instruction scheduling
    - More loop optimizations
    - Register allocation
    - Parallelization
    - Memory optimization

    Read these slides. If you find a part that is helpful and adds to your knowledge of optimizations, read it. These slides repeat some of the material from above readings. However, they are an excellent review, concise and clear, and reinforce key points. They also provide additional examples.

- 9.6: Verification and Validation of Code Optimization
  - Torben Ægidius Mogensen's "Basics of Compiler Design, Chapter 11: Analysis and Optimisation"URL

    Do Exercise 11.1 parts a) and b) on page 254.

    This exercise will give you some practice with code optimization and is based on Chapter 11, which covers analysis and optimization. The exercise uses common

subexpression elimination as an example of optimization. You can check your answers against the Answer Key.

Unit 10: Compiler Verification and Validation

The verification and validation (V&V) of a compiler consists of the V&V of its parts: scanner or lexical analyzer, syntax analyzer, semantic analyzer, Intermediate Code Generator, Intermediate Code Optimizer, Code Generator, and Code Optimizer. See the V&V notes for each of these. V&V is based on a careful plan, a well-defined compiler process (including version control or more extensive configuration control, and efficient and effective documentation), peer reviews, measurements, (including defect analysis), component tests, formal proofs, and integration tests. It can utilize known correct tools, such as parser generators, code generation templates, and other compilers and compiler parts. In addition, a formal certification of the compiler can be done by an independent organization.

Remember that verification pertains to correctness, which means satisfaction of specifications; and validation pertains to user needs, which means satisfaction of user operational requirements. Verification addresses, for example, correctness of results of scanning, semantic analysis, code generation, and code optimization. Validation includes meeting operational, functional, performance, and physical requirements for processing time and memory space, and also, the "ilities" -- reliability, availability, maintainability, and usability.

V&V depends on the complications of the source and target programming languages, the intended use of the compiler, number of front ends to back ends, available tools, run-time environments and machine architecture, and the system that it will interface with.

Finally, certification of a compiler -- i.e., compliance to a standard -- may be necessary for safety or security of critical programs.

Completing this unit should take you approximately 1 hour.

- Upon successful completion of this unit, you will be able to:
  - Describe verification and validation of a compiler.

- 10.1: Compiler Verification and Validation
  - Wikipedia: "ANSI C"URL

    Read this article about certification of ANSI C compilers.

  - Unit 10 AssignmentPage

    Review verification and validation of lexical analyzers, parsers, semantic analyzers, code generators, and code optimizers. In this unit (Unit 10) and in these exercises,

we combine the ideas of those sections and look at an integrated approach to the verification and validation of the compiler as a whole.

Unit 11: Compiler Summary and Next Steps

This unit concludes our course study of compilers. We summarize the topics we have studied in the course and look ahead to further study to build upon what we have learned in this course. The topics covered may at first seem limited in their application to the development of compilers. However, their application is much broader, and helps us write better programs. The techniques, abstractions, and data structures are applicable to many other applications, including safety, security, high-performance applications, data and control flow analysis, internal or intermediate representation and encoding of structures, and optimization for external dependencies.

Completing this unit should take you approximately 3 hours.

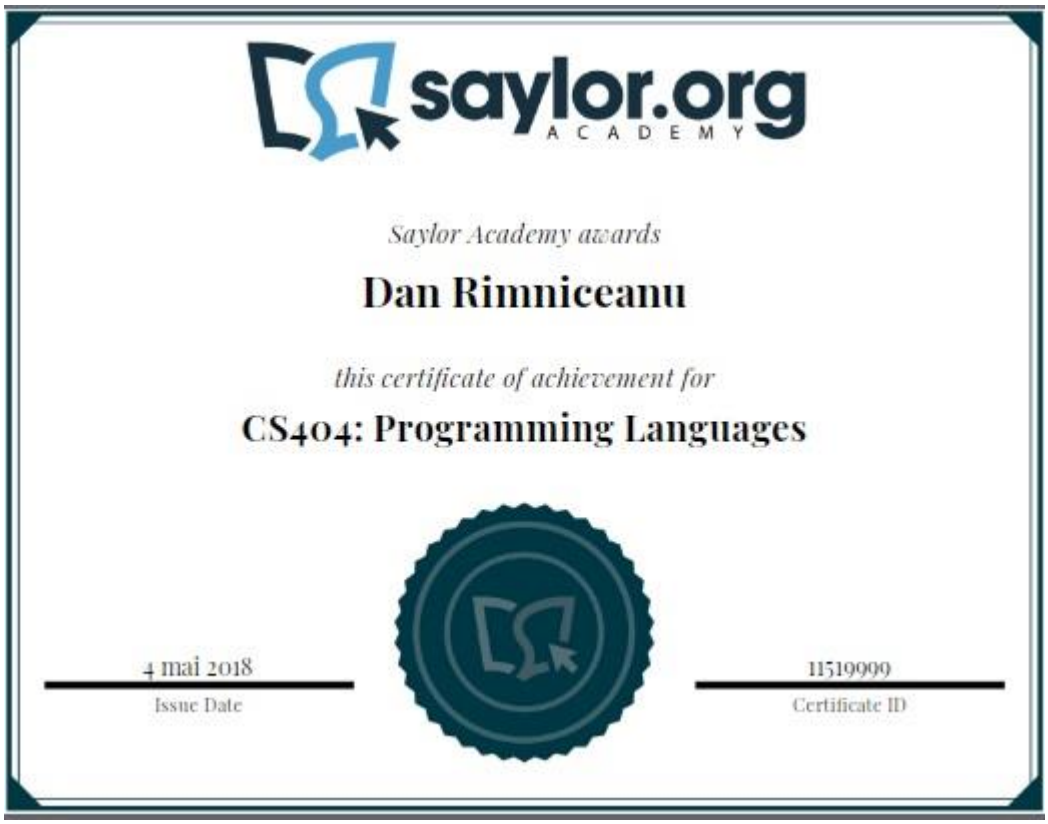
- Upon successful completion of this unit, you will be able to:
  - Summarize this course in terms of topics covered.
  - Explain the usefulness of the topics covered to non-compiler applications.
  - Identify areas of further study to build upon the foundation provided by this course.

- 11.1: Compiler Summary and Next Steps
  -  Stanford University: Keith Schwarz's "Code Optimization"URL

    Read slides 226 through 234, which provide a concise outline of the topics covered in our course.
  -  University of California, Berkeley: Paul Hilfinger's "Notes for Lecture 39"URL

    Read these slides.
  - University of California, Berkeley: Paul Hilfinger's "Lecture 39"Page

    Watch this video.



saylor.org
ACADEMY

Saylor Academy awards

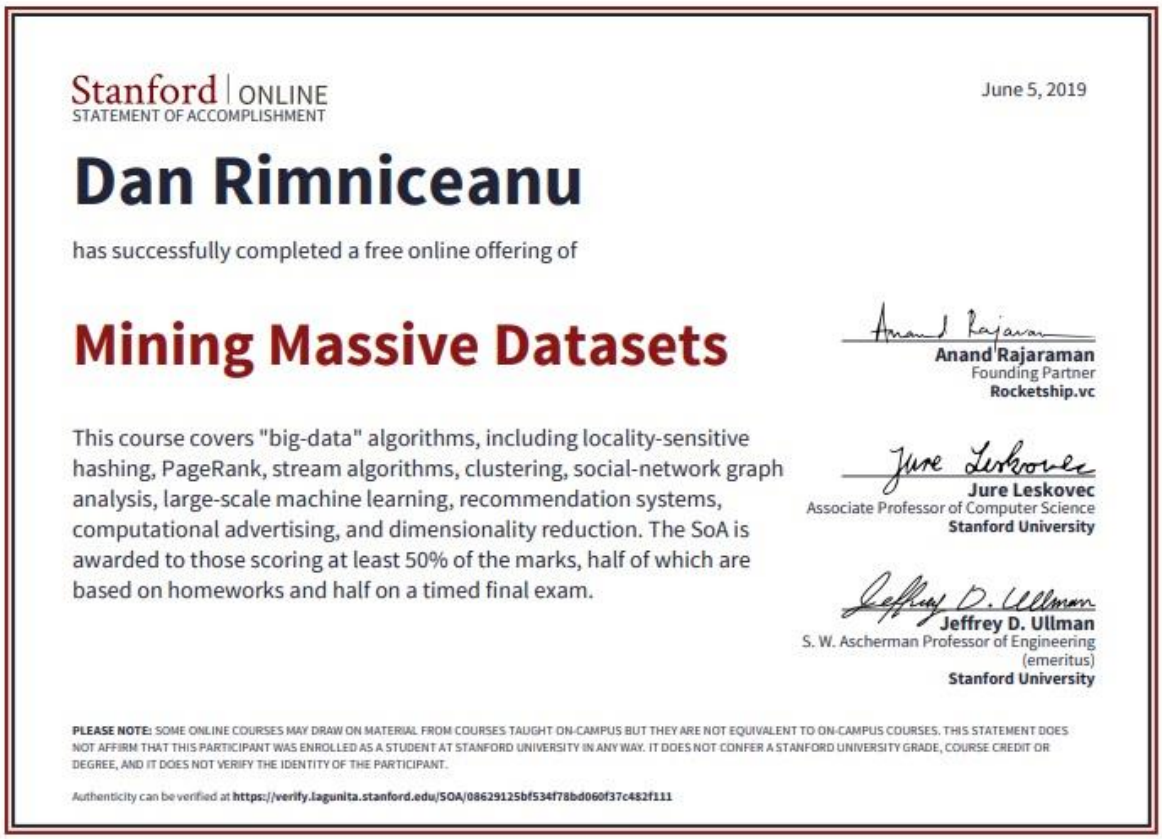
Dan Rimniceanu

this certificate of achievement for

CS404: Programming Languages

4 mai 2018
Issue Date

11519999
Certificate ID



Stanford | ONLINE
STATEMENT OF ACCOMPLISHMENT

June 5, 2019

Dan Rimniceanu

has successfully completed a free online offering of

Mining Massive Datasets

This course covers "big-data" algorithms, including locality-sensitive hashing, PageRank, stream algorithms, clustering, social-network graph analysis, large-scale machine learning, recommendation systems, computational advertising, and dimensionality reduction. The SoA is awarded to those scoring at least 50% of the marks, half of which are based on homeworks and half on a timed final exam.

Anand Rajaraman
Founding Partner
Rocketship.vc

Jure Leskovec
Associate Professor of Computer Science
Stanford University

Jeffrey D. Ullman
S. W. Ascherman Professor of Engineering (emeritus)
Stanford University

PLEASE NOTE: SOME ONLINE COURSES MAY DRAW ON MATERIAL FROM COURSES TAUGHT ON-CAMPUS BUT THEY ARE NOT EQUIVALENT TO ON-CAMPUS COURSES. THIS STATEMENT DOES NOT AFFIRM THAT THIS PARTICIPANT WAS ENROLLED AS A STUDENT AT STANFORD UNIVERSITY IN ANY WAY. IT DOES NOT CONFER A STANFORD UNIVERSITY GRADE, COURSE CREDIT OR DEGREE, AND IT DOES NOT VERIFY THE IDENTITY OF THE PARTICIPANT.

Authenticity can be verified at **https://verify.lagunita.stanford.edu/SOA/08629125bf534f78bd060f37c482f111**

CS404: Programming Languages

Course Introduction

This course is an upper division computer science course that studies the design of programming languages. While most of the industry uses either procedural or object-oriented programming languages, there are entire families of other languages with certain strengths and weaknesses that make them attractive to a variety of problem domains. It is important to know about these less well-known yet powerful languages if you find yourself working in an area that could utilize their strengths. In this course, we will discuss the entire programming language family, starting with an introduction to programming languages in general and a discussion of the features and functionality that make up the modern programming language. From there, each unit will discuss a different family of programming languages, including Imperative, Object-Oriented, Functional, Scripting, and, Logical. For each language, you will learn about its computational model, syntax, semantics, and pragmatic considerations that shape the language. By the end of this course, you will be able to intelligently discuss each of these programming paradigms, their respective strengths and weaknesses, and the reasons why you would opt to use one over the others in a given situation. You will also have opportunities to delve into the details of the design and evolution of several specific programming languages, including Scheme, Haskel, Java, C++, C#, Perl, Python, and Prolog.

Unit 1: Introduction to Programming Languages

Programming languages are not very different from spoken languages. Learning any language requires an understanding of the building blocks and the grammar that govern the construction of statements in that language. This unit will serve as an introduction to programming languages, taking you through the history of programming languages. We will also learn about the various universal properties of all programming languages and identify distinct design features of each programming language. By the end of this unit, you will have a deeper understanding of what a programming language is and the ability to recognize the properties of programming languages.

Completing this unit should take you approximately 17 hours.

- Unit 1 Learning OutcomesPage

- 1.1: Evolution of Programming Languages
  - Wright State University: T.K. Prasad's "Evolution of Programming Languages"URL

Read these slides, which provide a history of development of programming languages along the three major paradigms: Imperative, Functional, and Object-oriented. It also briefly discusses modern scripting languages.

- Johns Hopkins University: Mike Grant, Zachary Palmer, and Scott Smith's "Principles of Programming Languages"URL

Read chapter 1, which provides an overview of the pre-history and the early history of programming languages and an introduction to lambda calculus.

- Indian Institute of Technology, Delhi: S. Arun Kumar's "Introduction to Programming Languages"Page

Watch this lecture, which provides a detailed introduction to programming languages.

Write a few paragraphs describing the impacts of Internet on the evolution of programming languages. Which programming languages were developed specifically for the Web?

- 99 Bottles of BeerURL

This website collects code fragments for the song "99 bottles of beer" in 1500 different programming languages and variations. Write your own code to generate the lyrics for the song in a programming language that you know (for example, C or Java). The lyrics of the song can be found here. Check your code against what was submitted to the library for the language that you used. Compare that code with what was written for Prolog, Pascal, Scala, Scheme, Lisp, ML, C#, F#, and Haskel.

- 1.2: Lambda Calculus
  - University of Texas at Austin: Thomas Dillig's "Basic Lambda Calculus"URL

Read these slides, which will introduce you to lambda calculus. Skip the first 8 slides, which are about the organization of the CS 312 course. What are the four expressions in lambda calculus?

- New York University: Chris Baker's "Lambda Tutorial"URL

This is a great tutorial on lambda calculus. Browse to "Examples and practice," which have several exercises on lambda calculus. Guess the result of each expression before clicking "Reduce." Feel free to explore the rest of the page as well as the various recommended resources on lambda calculus.

- 1.3: Syntax as a Language's Form
  - Wright State University: T.K. Prasad's "Syntax Specification: Grammars"URL

Read these slides, which will introduce you to syntax as a language. The first few slides provide an overview of compiler, interpreter, and lexical analyzer.

- Indian Institute of Technology, Delhi: S. Arun Kumar's "Syntax and Grammar"Page

  Watch these lectures.

- 1.4: Semantics as the Meaning
  - University of Texas at Austin: Thomas Dillig's "Operational Semantics I"URL

    Read these slides. While there are several forms of language semantics (axiomatic, denotational, and operational), we will focus on operational semantics in this course. Make sure that you understand the difference between eager versus lazy evaluation, and call-by-name vs call-by-value.
  - University of Texas at Austin: Thomas Dillig's "Operational Semantics II"URL

    Read these slides.
  - Indian Institute of Technology, Delhi: S. Arun Kumar's "Semantics"Page

    Watch this video, which explains semantics and highlights several basic principles underlying semantics.

Unit 2: Types

In this unit, you will learn about types, a method of enforcing levels of abstraction in programs. Data in programs come in many types: real number, integers, characters, lists, etc. A type error occurs when an operation is applied to an inappropriate data type. A type system consists of a set of types, and a set of programs to analyze types and type judgment. You will also learn about the basics of static typing, type checking and type inference.

Completing this unit should take you approximately 19 hours.

- Unit 2 Learning OutcomesPage

- 2.1: Principle of Typing
  - University of Texas at Austin: Thomas Dillig's "Principles of Typing"URL

    Read these slides, which will define typing, and explain why we need typing and how types compute. What is the difference between dynamic and static typing? Do you know any language with dynamic typing?

- Johns Hopkins University: Scott F. Smith's "Principles of Programming Languages"URL

  Read Chapter 6, which discusses type systems.

- 2.2: Type Checking
  - University of Texas at Austin: Thomas Dillig's "Basic Typing Rules and Proofs"URL

    Read these slides, which discuss how to construct type systems for multiple languages and prove/show soundness of a type system.
  - University of Virginia: Wes Weimer's "Type Checking and Static Semantics"URL

    Read these slides, which discuss typed programming languages.
  - Indian Institute of Technology, Delhi: S. Arun Kumar's "Type Checking"Page

    Watch this lecture, which discusses type checking.
- 2.3: Polymorphic Typing and Type Inference
  - University of Texas at Austin: Thomas Dillig's "Polymorphic Tpying and Type Inference"URL

    Read these slides.

    - Advanced Typing Rules and Proof
    - Type Inference I
    - Type Inference II
    - Type Inference in L

Unit 3: Functional Programming

Functional programming is not used very frequently in the industry, yet it is very powerful. Functional programming treats computation as the evaluation of mathematical functions. Functional programming languages are deeply rooted in lambda calculus.

Whereas older Functional Programming languages were typically designed with a specific purpose in mind, newer Functional Programming languages are more "general purpose" and are more widely applicable. In this unit, we will discuss Functional Programming's place in the programming languages world, first taking a look at exactly what constitutes a Functional Programming language. We will conclude the unit with a discussion of some of the more prevalent features in Functional Programming. By the end of this unit, you will be able to identify Functional Programming languages and, more importantly, instances in which a Functional Programming language would be most beneficial.

Completing this unit should take you approximately 21 hours.

- Unit 3 Learning OutcomesPage

- 3.1: Overview
  - Haskell Wiki: "Functional Programming"URL

    Read this article for an overview of functional programming.

  - University of California, Berkeley: Brian Harvey's "Functional Programming"Page

    Watch these lectures for an overview of functional programming.

- 3.2: Higher-order Functions
  - University of California, Berkeley: Brian Harvey's "Higher-Order Procedures"Page

    Watch this lecture for examples of higher-order functions and procedures in functional programming.

- 3.3: Pure Functional Functions
  - Paul Hudak, John Peterson, and Joseph Fasel's "Values, Types, and Other Goodies"URL

    Read this page about values and types in Haskell.

  - Paul Hudak, John Peterson, and Joseph Fasel's "Functions"URL

    Read this page about functions in Haskell.

- 3.4: Real Languages: Haskell
  - University of Texas at Austin: Thomas Dillig's "Real programming languages: Haskell"URL

    Read this overview of Haskell, a major functional programming language.

- 3.5: MapReduce
  - Ralf Lammel's "Google's MapReduce Programming Model - Revisited"URL

    Read this article, which gives an overview of the famous MapReduce algorithm invented by Google and its implementation in Haskell.

Unit 4: Imperative Programming

An imperative language uses a series of statements, contained in blocks or functions, to control a created state and, ultimately, produce a desired output. Imperative languages and, by extension, procedural languages, are extremely common and are frequently used to teach novices how to program. This unit will define Imperative Programming and identify the language's key properties before moving on to address common control structures such as conditionals, loops, and case statements. By the end of this unit, you will understand how imperative programming languages work and be able to identify their common properties (and the ways in which those properties are employed in imperative programming).

Completing this unit should take you approximately 4 hours.

- Unit 4 Learning OutcomesPage

- 4.1: Introduction to Imperative Programming
  - University of Texas at Austin: Thomas Dillig's "Introduction to Imperative Languages"URL

    Read these slides. What are the advantages and disadvantage of GOTO statements?

- 4.2: Pointers
  - University of Texas at Austin: Thomas Dillig's "Operational Semantics of an Imperative Language with Pointers"URL

    Read these slides.

Unit 5: Object-Oriented Programming

Object-Oriented programming languages are widely used in both government and industry, thanks to the popularity of Java and its older brethren, C++. Object-Oriented Programming has many practical advantages over other programming paradigms. It is considered an upgrade over the once-dominant Procedural Programming scheme. This unit will present an overview of Object-Oriented Programming, including examples of the two most popular languages mentioned above. Next, we will discuss some of the distinctive properties of Object-Oriented Programming, discussing the ways in which it differs from other schemes and why it is considered an improvement over older designs. By the end of this unit, you will be able to describe Object-Oriented Programming and identify the main properties of and advantages to using Object-Oriented Programming.

Completing this unit should take you approximately 14 hours.

- Unit 5 Learning OutcomesPage

- 5.1: Introduction and Fundamental Features of Object Oriented Programming
    - University of Texas at Austin: Thomas Dillig's "Object-Oriented Languages"URL

      Read these slides.
    - Johns Hopkins University: Scott F. Smith's "Principles of Programming Languages"URL

      Read Chapter 5, which discusses object-oriented languages and why they have become popular.
    - University of California, Berkeley: Brian Harvey's "Object-Oriented Programming"Page

      Watch these lectures for an introduction to object-oriented programming. The second lecture discusses the implementation of object-oriented programming in SmallTalk, as well as several basic concepts in object-oriented programming languages, such as inheritance, class, superclass, exceptions, simulation, message sending and type declaration. The third lecture discusses the specific and fundamental features in object-oriented programming languages.
- 5.2: Object Types and Subtyping
    - Johns Hopkins University: Scott F. Smith's "Principles of Programming Languages"URL

      Read Chapter 6 on Type Systems for a discussion of statically-typed object-oriented languages and the similarities and differences between C++ and Java implementations of object system.
- 5.3: Java Virtual Machine
    - Young Telecaster's "Java tutorial-Lecture 3-Basic Introduction 3"URL

      Watch this brief overview of Java Virtual Machine, method lookup, verifier analysis, and Java security. What is the difference between a complier and a virtual machine?
- 5.4: Templates and Generics
    - Chris Szalwinski's "Overview of Polymorphism"URL

      Read this introduction to parametric polymorphism for object-oriented programming through in-depth analysis of C++ Templates and Java Generics.

Unit 6: Scripting Languages

Scripts are everywhere in computer science. Any job in the computer science field will require you to write a script at some point or another, whether that script sets up the runtime environment for an application or automates the build process. Scripting languages are primarily designed for "gluing", i.e. connecting components. We will first learn the basics of scripting, learning in particular how scripts are useful. You will then learn about the wide-ranging family of scripting languages and the properties that those languages tend to share. By the end of this unit, you will be able to define scripting and identify situations in which a script would be most useful.

Completing this unit should take you approximately 27 hours.

- Unit 6 Learning OutcomesPage

- 6.1: What Is a Scripting Language?
  - John K. Ousterhout's "Scripting: Higher Level Programming for the 21st Century"URL

    Read this introduction to scripting languages. You will learn to distinguish between system programming languages and scripting languages (e.g strong versus weak typing, compiled versus interpreted).

  - University of Maryland, Baltimore County: Daniel J. Hood's "Introduction to Scripting Languages"URL

    Read this brief overview of scripting languages, including short scripts in major scripting languages such as Ruby, Perl, JavaScript, PHP, and Python.

- 6.2: Regular Expressions
  - University of Maryland, Baltimore County: Daniel J. Hood's "Regular Expressions"URL

    Read each of the pages under "Notes". Regular expressions are sets of symbols and syntactic elements used to match patterns of text.

- 6.3: Ruby
  - University of Maryland, Baltimore County: Daniel J. Hood's "Ruby"URL

    Read each of the pages under "Notes" for examples relating to Ruby, a popular programming language.

- 6.4: Python
  - University of Maryland, Baltimore County: Daniel J. Hood's "Python"URL

Read each of the pages under "Notes" for examples relating to the Python scripting language.

- 6.5: JavaScript
  - University of Maryland, Baltimore County: Daniel J. Hood's "JavaScript"URL

    Read the pages under "Notes" for examples relating to JavaScript.

Unit 7: Logical Programming

The relationship between mathematical logic and computer science runs deep. As such, there are a number of programming language tools that enable you to rapidly code efficient logic systems for deployment in a variety of high-tech arenas, such as Artificial Intelligence, Textual Analysis, and so on.

In this unit, you will learn about Logical Programming (also known as Declarative Programming) and why you would use it to solve these sorts of problems. We will discuss the fundamental features of Logical Programming and identify what distinguishes Logical Programming from other programming paradigms. By the end of this unit, you will be able to identify the problem domain that Logical Programming covers and recognize when you should take advantage of Logical Programming's considerable power.

Completing this unit should take you approximately 9 hours.

- Unit 7 Learning OutcomesPage

- 7.1: Overview
  - University of Oxford: Michael Spivey's "An Introduction to Logic Programming through Prolog"URL

    Read Chapter 1 for an introduction to logic programming.

  - Indian Institute of Technology, Kharagpur: P. Dasgupta's "Prolog"Page

    Watch this lecture for an overview of logic programming, using Prolog as an example.

- 7.2: Basic Features and Examples
  - University of California, Berkeley: Brian Harvey's "Logic Programming"Page

    Watch these lectures for an overview of the fundamental features of logic programming using several Scheme examples.