Proposal for an Undergraduate Computer Science Curriculum for the 1980s

Part II: Detailed Course Descriptions

Mary Shaw, Steve Brookes, Marc Donner,
James Driscoll, Michael Mauldin, Randy Pausch,
Bill Scherlis, Alfred Spector

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213
20 October 1983

Abstract

The authors propose to the Carnegie-Mellon Computer Science Department a curriculum for undergraduate computer science. This Part contains the detailed course descriptions that support the curriculum proposal described in Part I.

The Curriculum Design Project is supported by general operating funds of the Carnegie-Mellon University Computer Science Department.

Table of Contents

Part I: Discussion

Part II: Detailed Course Descriptions

8. Course Descriptions

This chapter presents descriptions of the courses we propose to be the major components of an undergraduate computer science curriculum. We have tried to write descriptions that will indicate clearly the scope and emphasis we have in mind. However, a complete course design is a major undertaking, so most of these descriptions should be viewed as design sketches, not full designs. An overview of the course structure, including course names and prerequisites[2], is given in Figure 8-1.

Some courses that are shown would be offered by departments other than computer science. In some instances (as in COMBINATORIAL ANALYSIS [MATH 301 / CS 251]), these courses have computer science course numbers. Given more time and broader expertise, the Curriculum Design Project would have made detailed sketches on such courses; instead, information about similar courses currently offered at Carnegie-Mellon University has been provided, but only for purposes of exposition and completeness. We are neither endorsing nor criticizing the current curricula of these courses.

[2] The prerequisite structure is complete only for computer science courses.

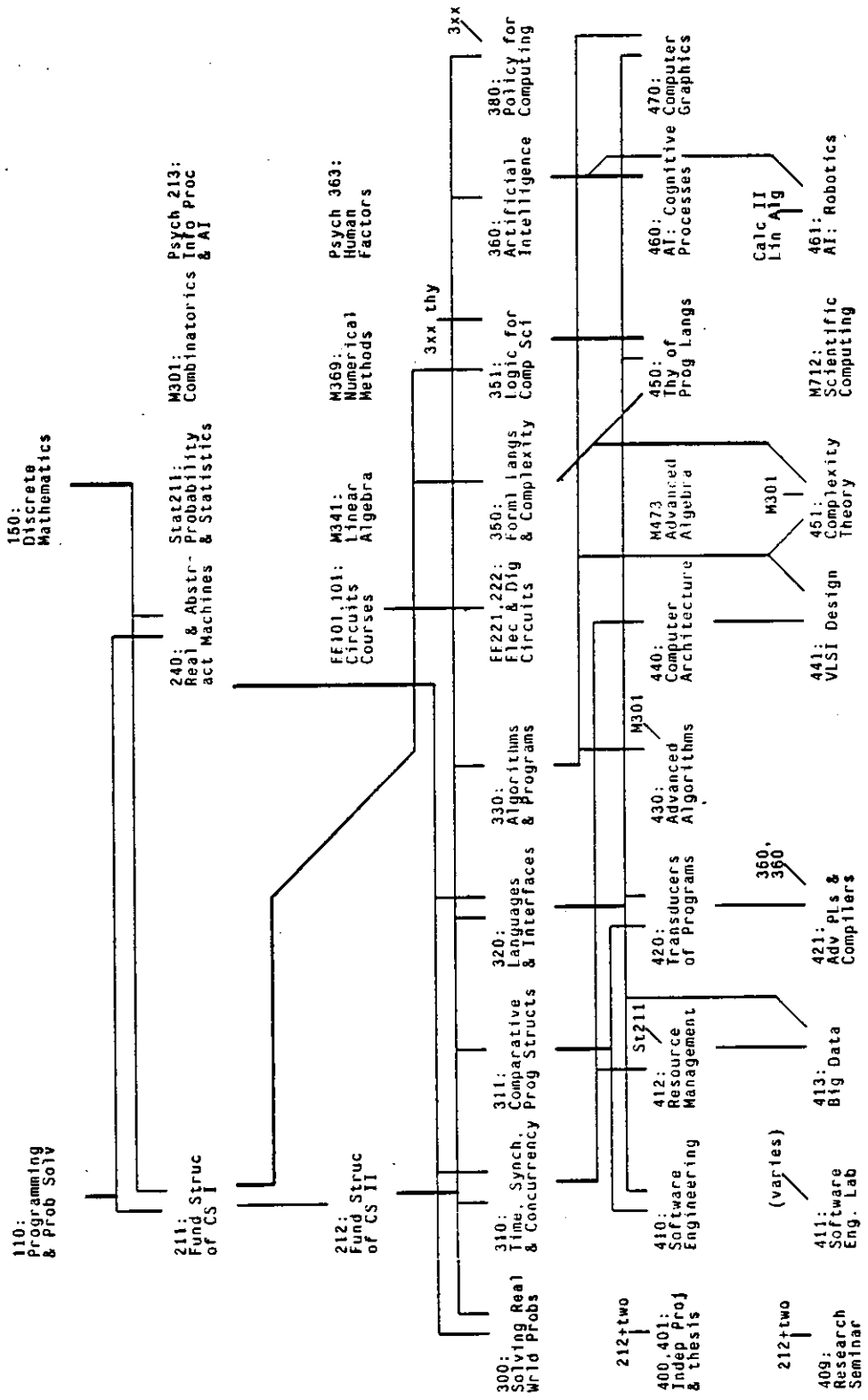

Figure 8-1: Course and Prerequisite Structure

8.1 Basic and Introductory Courses

Introductory computing courses serve all the communities described in Section 2.1. They exist to provide general computer literacy to the campus at large, to provide the background for students who must use computers in other disciplines, and to provide sufficient background for the basic computer science courses.

In order to establish the basis of our computer science curriculum, we give a brief description of two kinds of introductory course. The first of these is directed at the need for general literacy about computing. The second is directed at the need for the specific skills required for good programming, including problem-solving skills. The two courses are independent; we conceive of the former as a universal requirement.

8.1.1 Computers in Modern Society [100]

This course presents algorithmic thinking and the role of computing and technology in contemporary society. It covers

- Use of computing facilities, including important classes of programs such as text formatters, electronic message systems, interactive computation and planning systems, and public information utilities.
- Survey of classes of computers and applications, with emphasis on the diversity of the applications and the common elements of the successful ones.
- The style of precise, deductive reasoning and problem solving that characterizes science and engineering. One of the carriers of this idea will be an introduction to elementary computer programming.
- Ethical and social implications of widespread computing power.

There is an opportunity for this to become a course that teaches all students, particularly nontechnical students, about the nature of scientific reasoning. Such a course would include elements of formal logic, history and philosophy of science, and Western civilization. Hands-on experience with computers could provide the direct experience and tangible feedback that is often difficult to provide. The course would help balance the conventional view of liberal education that calls for scientists to take substantive courses in the humanities but not for humanists to take substantive courses in the sciences.

8.1.2 Programming and Problem Solving [110]

Techniques for solving problems with computers, including problem-solving and programming skills. This is the course for students who will take more advanced computer science courses. This course could use an introductory programming text and books such as the following:

- R.G. Dromey, *How to Solve It by Computers* [20].
- J.R. Hayes, *The Complete Problem Solver* [32].
- G. Polya, *How to Solve It* [63].
- M. Rubinstein, *Patterns of Problem Solving* [68].
- W.A. Wickelgren, *How to Solve Problems* [84].

8.1.3 Discrete Mathematics [150]

Prerequisites: NONE

Description: This course introduces and relates a variety of discrete mathematical themes and subjects. This course is intended to serve prospective computer science and mathematics majors, among others.

Rationale: The themes listed below provide the fabric that holds the course together. Although they are not mentioned explicitly in the subject listing, it is important that they be approached frequently in textbooks and in lectures.

Objectives: At the end of this course, a student will have a command of the basic ideas and techniques from discrete mathematics and will be able to apply them to problems outside mathematics, such as problems in computer science. Besides these skills, students will have begun to develop an appreciation of the nature and use of abstraction, an understanding of the roles of language and logic in mathematics, an understanding of the notion of mathematical structure, and an understanding of the nature of mathematical proof.

Ideas: This course will be the primary carrier of the following:

- ► Problem diagnosis
- ► Abstraction: how to go from particular to abstract
- ► Representation: making abstract structure concrete
- ► Mathematical reasoning and the notion of proof
- ► Operational reasoning and the notion of algorithm
- ► Recursion and induction; operational vs. relational reasoning
- ► Modeling
- ► Synthesis: building mathematical structures

It will reinforce or share responsibility for:

- ► A precise understanding of the notion of algorithm and an appreciation of the role of algorithms in mathematics.

Topic Outline:

1. Graphs
   - ► Fundamental ideas
   - ► Directed acyclic graphs and trees
   - ► Simple algorithms on graphs
2. Sets
   - ► Sets and set equality
   - ► Defining sets: extension and abstraction, paradoxes
   - ► Relations between sets, operations on sets
   - ► Infinite sets
   - ► Relations, mappings, and functions
3. Logic Skills
   - ► Propositions and truth functions
   - ► Individuals, predicates, and quantification
   - ► The language of logic
   - ► Expressing statements in the language of logic
   - ► Informal deduction in predicate logic

- ▶ A unifying structure: Boolean algebra

4. Induction
   - ▶ Elementary stepwise induction and complete induction
   - ▶ Induction over general structure and inductive definitions of sets
   - ▶ Recursive definitions (e.g., of sequences and functions)
5. A Brief Introduction to Logic and Mathematical Reasoning
   - ▶ Syntax: formal languages and inductive definitions
   - ▶ Deduction: axioms, rules of inference, and proofs
   - ▶ Informal and formal proofs
   - ▶ A glimpse at semantics: interpretations and soundness
6. Counting
   - ▶ Combinations, permutations
   - ▶ Binomial and multinomial theorem
   - ▶ Inclusion/exclusion
7. Relations, Equivalence Relations, and Order
   - ▶ Properties of relations, closures
   - ▶ Equivalence relations, partitions, equivalence classes
   - ▶ Examples of equivalence relations: divisibility of integers, modular arithmetic
   - ▶ Partial and linear order
   - ▶ Well-founded ordering
8. Retrospect
   - ▶ (Several lectures drawing on previous work to reinforce the themes of abstraction, proof, algorithm, etc. More than one unit of this type may be needed.)
9. Matrices
   - ▶ Matrix algebra
   - ▶ Linear systems, Gaussian elimination
   - ▶ Applications: incidence matrices, transitive closure, (and possibly Markov chains)
10. Algebraic Structures
    - ▶ Associative binary operations and semigroups
    - ▶ Examples of semigroups (e.g., tables, strings, composition of functions, matrices)
    - ▶ Algebras and structures
    - ▶ Monoids, groups, rings, and fields
    - ▶ Isomorphism and homomorphism
11. Recurrence Relations
    - ▶ Recursive definition of sequences
    - ▶ Differencing and summation
    - ▶ Solution of linear recurrence relations
    - ▶ Applications to algorithm analysis (e.g., Fibonacci, binary search)

References:

- ▶ G. Birkhoff and T.C. Bartee, *Modern Applied Algebra* [8].
- ▶ J.L. Gersting, *Mathematical Structures for Compuer Science* [25].
- ▶ I. Lakatos, *Proofs and Refutations: The Logic of Mathematical Discovery* [44].
- ▶ C.L. Liu, *Elements of Discrete Mathematics* [50].
- ▶ C.L. Liu, *Introduction to Combinatorial Mathematics* [49].
- ▶ D.F. Stanat and D.F. McAlister, *Discrete Mathematics in Computer Science* [73].
- ▶ H.S. Stone, *Discrete Mathematical Structures and their Applications* [74].
- ▶ J.P. Tremblay and R.P. Manohar, *Discrete Mathematical Structures with Applications to Computer Science* [78].

Resource Requirements:

Implementation Considerations and Concerns:

- Although this is a mathematics course, it should be taught with close attention to the abstractions of computer science.
- This course is also listed as INTRODUCTION TO APPLIED MATHEMATICS [MATH 127] in the Mathematics Department.
- This course should provide sufficient maturity for the student to continue with more advanced mathematics courses. If this one term course proves insufficient, it may need to be split into a two term sequence. In that event, course COMBINATORIAL ANALYSIS [MATH 301 / CS 251] would be involved in the redesign.

8.2 Elementary and Intermediate Computer Science Courses

These courses form a core that is germane to nonterminal, terminal, and joint-interest students. We believe that all those students need a foundation based on a balance between theory and practice. Divergence, if any, can come in the advanced courses.

In addition to the courses we define here, some of the content of computer science as described in Chapter 4 may be taught in departments other than computer science. These departments include mathematics, electrical engineering, psychology, and others. We have generally avoided designing courses that cover material taught at Carnegie-Mellon in these other departments.

To show more complete coverage of computer science, however, we list here the titles of courses that should be jointly listed by computer science and another department. Catalog descriptions for these courses appear in Chapter 9.

- ► COMBINATORIAL ANALYSIS [MATH 301 / CS 251]
- ► NUMERICAL METHODS [MATH 369 / CS 352]
- ► PROBABILITY AND APPLIED STATISTICS [STAT 211 / CS 250]
- ► LINEAR CIRCUITS [EE 101 / CS 241]
- ► ELECTRONIC CIRCUITS [EE 102 / CS 242]
- ► ELECTRONIC CIRCUITS II [EE 221 / CS 340]
- ► ANALYSIS AND DESIGN OF DIGITAL CIRCUITS [EE 222 / CS 341]

Some of the courses outlined in this section may also be jointly listed. In particular, DISCRETE MATHEMATICS [150] can be listed in the Mathematics Department and REAL AND ABSTRACT MACHINES [240] is very similar to INTRODUCTION TO DIGITAL SYSTEMS [EE 133].

8.2.1 Fundamental Structures of Computer Science I [211]

Prerequisites: PROGRAMMING AND PROBLEM SOLVING [110]
DISCRETE MATHEMATICS [150]

Description: This course introduces students to the fundamental scientific concepts that underlie computer science and computer programming. Software concepts such as abstraction, representation, correctness, and performance analysis are developed and are related to underlying mathematical concepts. Students are asked to apply these concepts to programming problems throughout the course.

Rationale: The ideas of abstraction and analysis are fundamental in computer science and should be introduced as early as possible in the curriculum — as soon as students are familiar with the activity of programming. The specific ideas and techniques introduced in this course serve as the basis for detailed development in later computer science courses.

Objectives: At the end of this course, a student will:

- Appreciate the central role of abstraction in computer science and programming.
- Be able to reason precisely about the correctness and performance of simple programs.
- Understand how a knowledge of analytical techniques can aid informal programming activity.
- Improve his programming skills through practice and analysis of existing code.
- Be aware of some of the basic program structures and programming techniques.

Ideas: This course will be the primary carrier of the following:

- The nature and use of abstraction in computer science.
- Basic techniques for reasoning about program correctness and analyzing program performance.
- Fundamental algorithms for searching and sorting in arrays.

Topic Outline:

1. Introduction: Understanding Programs
   - Abstraction
   - Specification and implementation
   - Analysis: correctness and performance
   - Search in an unordered array
   - Search in an ordered array: Binary search
2. Brief Review of Discrete Mathematics for Computer Science (review of DISCRETE MATHEMATICS [150])
   - Logic skills
   - Sets, relations, functions, graphs
   - Induction and recursive definition
   - Abstraction, language, and logic
3. Finite State Automata
   - Alphabets and languages
   - Describing languages: recognition and generation
   - The notion of state; abstract automata
   - Nondeterminism
   - Regular expressions
4. Programming Languages: Abstractions
   - Syntax: programming languages as formal languages
   - Flowchart programs and control structures

- ► Basic control structure abstractions: sequencing, conditionals, and iteration
- ► Procedures and function subprograms
- ► Identifiers, variables, binding, and assignment
- ► Parameter binding
- ► Scope, extent, and free-variable binding
- ► Recursion

5. Programming Languages: Pragmatics
   - ► Specifying the meanings of programs
   - ► Machine-level languages
   - ► Representation of high-level constructs
   - ► Translation and interpretation
6. Correctness of Programs
   - ► Program specification and programming language semantics
   - ► Test vs. proof
   - ► Assertions about programs
   - ► Hoare assertions and weakest preconditions
   - ► Loops and invariants
   - ► Specification, abstraction, and modularity
7. Performance of Programs
   - ► Resource utilization
   - ► Measuring input size, expressing cost
   - ► Experimental methods for cost estimation
   - ► Analytic methods
   - ► When and how to improve performance
8. Major Examples
   - ► Abstraction and analysis revisited
   - ► Sorting
   - ► Lexical Analysis

References:

- ► A.V. Aho, J.D. Hopcroft, and J.E. Ullman, *Data Structures and Algorithms* [3].
- ► O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* [15].
- ► W.A. Wulf, M. Shaw, P.N. Hilfinger, and L. Flon, *Fundamental Structures of Computer Science* [88].

Resource Requirements (Software):

- ► Example programs to work with
- ► Interpreters for micro-languages
- ► Simulators for finite-state automata
- ► Possibly program timing support — routines and test-bed
- ► Data sets for sorting and searching — tuned to best, worst cases for various algorithms

Implementation Considerations and Concerns:

- ► See [22, 34] for discussions of the course design.
- ► Because of the inexperience of the students and the large class sizes, this course is particularly sensitive to the problem of concentrating on the examples at the expense of the major underlying themes and principles.
- ► So as to provide an appropriate bridge from the programming done in the one-hundred

level course to the more theoretical matters that are the topic of this course, it is important to blend in a sufficient number of programming examples.

8.2.2 Fundamental Structures of Computer Science II [212]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]

Description: The course is a continuation of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]. It comprises five major parts: data abstraction, implementation of data types and corresponding algorithms, models of computation, topics in computer implementations, and a brief introduction to LISP. In addition to lectures on these areas, students are asked to complete a number of programming assignments.

The programming assignments are an integral part of the course. They are often the first programs that are large enough to force the student to deal with abstraction (by necessity), and they give the student an opportunity to apply algorithms and abstraction techniques that are presented in class. Students are asked to program and think about programming during the entire course. It is this emphasis that ties the course together.

Rationale: This course presents a breadth first cut across many topics in computer science. Taken as a last course in computer science, this course and FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211] provide an introduction to the central topics in the field. Taken as an introduction to the more advanced courses, this course provides students with material that is either prerequisite or introductory.

Objectives: At the end of this course, a student will have

- Enough programming skill to handle larger programming assignments and experience in both PASCAL and, to a lesser extent, LISP
- A basic knowledge of data abstraction and specification techniques, and the ability to implement a data structure to support a given specification
- Some background in computability theory
- Some background in the implementation of programming languages
- Some background in the design and analysis of algorithms.

Ideas: This course will be the primary carrier of the following:

- Data type specification and abstraction techniques
- Data structure design and analysis, including time/space tradeoffs
- The organization of systems via the use of layered abstractions
- LISP programming

It will reinforce or share responsibility for:

- Data representations and related algorithms
- Topics in the implementation of programming languages
- Computability theory
- Verification techniques

Topic Outline:

1. **Data Abstraction.** An example is presented almost immediately, so students can begin to program an abstract data type on their own.
   a. Course Introduction, Abstract Data Types
   b. Abstraction, An example of Directories as ADT implemented via Hashing w/Linear Probing
   c. Introduction to Formal Specification

d. Algebraic Specifications, Sequences, Positive Integers as examples
e. Abstract Models (Strings, Queues, Sets), Verification Considerations

2. **Abstract Data Types, their Implementation, and Corresponding Algorithms.** In general, abstract data types are semi-formally introduced before algorithms and representations are discussed. This material is presented in Pascal. It is specifically not intended that the introduction to LISP be merged with the introduction to data types.
   a. Basic Programming Abstractions
      - ► Variant Records, and Pascal Modules (Independent Compilation)
      - ► Pointers/References Introduced
      - ► Representation techniques (e.g., packing, encoding)
   b. Queues and Stacks
      - ► Stacks, Nested Abstractions
      - ► FIFO Queues
      - ► Static Implementations; e.g. via arrays
      - ► Implementations Involving Single Linking, Double Linking, Circular Structures
      - ► Modeling, Discrete Event Simulations
      - ► Bucket Hashing
      - ► Other Uses of Queues and Stacks
   c. Types often implemented with trees
      - ► Abstractions of Sets, Directories, Symbol Tables, Priority Queues, etc.
      - ► Introduction to Trees and Definitions: Trees as an Abstract Type
      - ► Tree Walks, Uses of Trees, Specification of Trees, Binary Tree
      - ► Inductive Proofs of Trees, Representation of Trees
      - ► Binary Search Trees, Recursive and Iterative Processing
      - ► Deletion in Binary Search Trees
      - ► Balancing Trees
      - ► Multi-way Trees, 2-3 Trees, Heaps
      - ► Copying Structures with Pointers/Recursion
      - ► Multi-key Retrieval, Database Queries
      - ► Interval Retrieval, Iterators
   d. Graph-like Types
      - ► Graphs Introduced, Defined, and Exemplified
      - ► Graph Traversal, Connectness Algorithms
      - ► Graph Representations and Transitive Closure Algorithms
      - ► Transitive Closure Algorithms Refined, Shortest Path Algorithms
      - ► Overview of Traveling Salesman and Spanning Tree Algorithms

3. **Models of Computation: An Introduction.**
   a. Turing Machines
   b. Other Models
   c. Church's Thesis and Computability
   d. The Halting Problem and Undecidability
   e. Context-Free Languages and the Chomsky Hierarchy

4. **Topics in Computer Implementations.**
   a. Introduction to Storage Allocation
   b. Stack-based storage allocation, static and dynamic scoping
   c. Non-Stack-Based Allocation, Freelists, Explicit Merging of Objects
   d. Buddy System Allocation, Marking In-use Objects and Garbage Collection

5. LISP
   a. Notion of Applicative Language
   b. Programming Environments and Interpreters
   c. Use of simple, yet powerful, primitives
   d. The power of recursion
   e. Different implementations of abstractions already seen

References:

► A.V. Aho, J.D. Hopcroft, J.E. Ullman, *Data Structures and Algorithms* [3]. Chapters/Sections 1, 2.1-2.4, 3.1-3.2, 5.1-5.2, much of 6 and 7.

- O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* [15].
- P.H. Winston and B.K.P. Horn, *LISP* [85].
- W.A. Wulf, M. Shaw, P.N. Hilfinger, and L. Flon, *Fundamental Structures of Computer Science* [88]. Chapters 7, 8, 9, 10.1-10.5, 11.1-11.2, 13, 14, 15, 16, 19.
- Other reading to be determined.

Resource Requirements (software):

- Library of data types to support many of the assignments
- Simulator for Turing Machines
- Some assignments can involve students with larger programs by giving them running versions of exemplar programs (whose design was presented in class) and asking for modifications.
- Note that FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212] requires much greater computing resources than FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211].

Implementation Considerations and Concerns:

- See [22, 34] for discussions of the course design.
- It is specifically intended that LISP be taught for approximately the last 30% of the course and that the data structure material be taught initially in Pascal. There are two reasons: First, one new thing is enough to learn at a time. Second, the students should see the data structures material from two points of view (that is, Pascal and then LISP), and the advantages and disadvantages of both should be made clear.
- Feedback to students is important. An extensive grading staff is required to provide substantive feedback to students on their programs with respect to algorithms abstraction techniques, and general programming style. This is especially true when students are supposed to be learning design techniques that require extensive individual feedback and for which answer sheets do not give enough guidance. This is such a course, and it must be adequately staffed.

8.2.3 Real and Abstract Machines [240]

Prerequisites: PROGRAMMING AND PROBLEM SOLVING [110]
DISCRETE MATHEMATICS [150]

Description: In this course the student is introduced simultaneously to the theoretical models and the hardware instances of machines that compute. The notions of layers of virtual machines is explored and their realization in various combinations of hardware and software are major themes. Beginning with primitive computations, the mathematical concept of function is used to capture the capabilities of combinatorial digital logic circuits. From that base, finite automata are introduced as tools for understanding, analyzing, and designing finite state machines. After that, Turing Machines and, more appropriately, register machines are introduced and related to the architectures of real computers. Finally, microcode, machine/assembly language, and general-purpose programming languages are positioned in this hierarchy.

The laboratory component of this course will require about three hours of lab work per week and will expose the student to simple instances of some of the machine types covered in the lectures. Sudents will simulate instances of several classes of machine and will design and construct simple combinatorial circuits and a simple finite state machine.

Rationale: Conventional teaching of computer architecture doesn't convey the sense that the capabilities of constructable systems can be described and reasoned about formally. While the strict identity between formal models and actual machines ends at the finite automata, there is nonetheless a great deal to be learned from exposure to both the formal models of more powerful machines and to the architectural ideas embodied in real machines. The purpose of this course is to expose students to the design and construction of various kinds of computing devices and to establish that there are formal techniques for reasoning about the mathematical properties of computing machines.

Objectives: At the end of this course, a student will be able to:

- Understand and describe the relationships between some formal models of machines and corresponding real machines
- Understand the notion of an interpreter of an instruction set and the layers of abstract machines that are present in all real systems
- Understand the use of a clock to impose the discrete time abstraction on the continuous time functions of real circuits
- Understand the circuit family abstraction that permits Boolean algebra to describe the behavior of real electronic devices
- Design and implement simple Finite State Machines

Ideas: This course will be the primary carrier of the following:

- Concept of machine as executor or interpreter of an instruction stream
- Elementary computer architecture
- Abstract machines, corresponding languages, and corresponding real machines; notion that there are different kinds of machines with different power and various realizations
- Discrete time, ordering of events

- ► Circuit family abstraction

It will reinforce or share responsibility for:

- ► Abstraction and representation
- ► Finite state automata, PDAs, Turing Machines
- ► Basic design levels of hardware (shared with COMPUTER ARCHITECTURE [440])
- ► Notion of algorithm
- ► Notion of state
- ► Boolean algebra

Topic Outline:

1. Function
   - ► Concept of Function
   - ► Circuit Family abstraction (with hardware lab)
   - ► Combinatorial Logic Circuits (with hardware lab)
   - ► Review of elementary Boolean logic (De Morgan's Theorem ...)
   - ► Minimization versus VLSI
   - ► PLA, PAL, PROM (with hardware lab)
2. Finite Automata
   - ► Registers and Latches (with hardware lab)
   - ► Combinatorial Logic with feedback
   - ► Clocks and discrete time (with hardware lab)
   - ► Regular Languages, Regular Expressions
   - ► Finite State Machines (with hardware lab)
   - ► Register Transfer Level description
   - ► Mealy and Moore Machines
   - ► One-Hot versus Encoded implementations (with hardware lab)
3. Push Down Automata
   - ► Context Free Grammars (with software lab)
   - ► Context Free Languages
   - ► BNF
   - ► Related real machines - B5000, HP3000, HP calculators (with software lab)
4. Machine models
   - ► Turing Machines (with software lab)
   - ► Register Machines (with software lab)
   - ► Von Neumann architecture and the Universal TM
   - ► Memory Devices
     - > RAM
     - > Disk
     - > Tape
     - > Memory hierarchies
5. Architecture Introduction
   - ► ISP and Executor/Interpreter model (with software lab)
   - ► Microcode
   - ► Machine Language (with software lab)
   - ► Assembly Language
   - ► Intermediate Language (e.g. Pcode)
   - ► Higher Level languages and their Virtual Machines

References:

► C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering* [4].

- M. Minsky, *Computation: Finite and Infinite Machines* [54].
- D.P. Siewiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples* [72].

Resource Requirements:

- Finite State Machine simulator
- Regular Expression to FSM converter
- PDA Simulator
- CFG to PDA converter
- TM Simulator
- ISP simulator
- Digital Electronics lab suitable for building simple digital circuits up to FSMs: TTL parts, breadboards, power supplies, signal generators, switches and displays, oscilloscopes, logic analyzer, etc.

Implementation Considerations and Concerns:

- It may be tricky to find faculty with the right mix of interests to teach this course with good balance between hardware, software, and theory.
- This course is very similar to INTRODUCTION TO DIGITAL SYSTEMS [EE 133], offered in the Electrical Engineering Department. The content is similar enough that the lab facilities might easily be set up in common and shared.

8.2.4 Solving Real Problems [300]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]

Description: This problem-oriented course provides students with an opportunity to solve real-world problems under the guidance of an instructor. Skills from a variety of areas both within and outside of computer science will need to be brought to bear on class examples and assignments posed as problems by the instructor. The emphasis is on the techniques used in obtaining the solution, rather than the solution per se. While proper software engineering techniques will, of course, be expected for all solutions involving software, it should be noted that the emphasis in the course is problem solving, **not** software engineering.

Rationale: Traditional courses provide particular knowledge and skills, but usually the problems posed in such courses focus narrowly on the topic of the course. Rarely does a student encounter a problem with the real-world characteristic of requiring a non-trivial combination of acquired skills. In addition, academic assignments often make broad assumptions that make the task much cleaner than actual problems tend to be. This course should help prepare students for the realistic, thorny sorts of problems that they will have to encounter after graduation.

Objectives: At the end of this course, a student will be able to:

- ► Critically examine a task and define the real issues in solving a problem.
- ► Form a well-organized attack on a problem.
- ► Implement a solution, cognizant of possible error or oversight.
- ► Evaluate a completed solution, and learn and generalize from it.

Ideas: This course will be the primary carrier of the following:

- ► Problem analysis, definition, and decomposition.
- ► Coping with external constraints not necessarily inherent in the problem.
- ► Application of knowledge and technique in novel ways.
- ► Critical evaluation of a finished solution.

Topic Outline:

1. Basic stages of problem solving
   a. Problem definition
   b. Plan of attack
   c. Execution of a plan
   d. Check for correctness of solution
   e. Evaluation of a finished solution
2. Path to a solution as the desired results
   a. Working backwards from a final goal
   b. Establishment of stable substructures
3. Reductionism vs. Holism
4. Knowledge vs. skill
5. Epistemology
6. Models and modelling
7. Analogies and metaphors
8. Verbalizing and expressing a problem or solution
9. Well-structured vs. Ill-structured problems

10. Overcoming conceptual blocks
11. Defining and narrowing a problem domain
12. Reduction to a known problem
13. Partial Solutions
    a. Giving up on hard cases
    b. Approximate results: coping

References:

- J.R. Hayes, *The Complete Problem Solver* [32].
- I. Lakatos, *Proofs and Refutations* [44].
- G. Polya, *How to Solve It* [63].
- G. Polya, *Mathematical Discovery* [62]
- M. Rubinstein, *Patterns of Problem Solving* [68].
- W.A. Wickelgren, *How to Solve Problems* [84].

Resource Requirements:

- A healthy number of class examples of real problems of an inter-disciplinary flavor.

Implementation Considerations and Concerns:

- This course is modelled after existing "Analysis, Synthesis, and Evaluation" courses taught in engineering curricula, particularly the course described in [61]. Stanford's graduate course 204 was also studied [13, 81, 42].

8.2.5 Time, Concurrency, and Synchronization [310]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]
PHYSICS I [PHYS 121] (MECHANICS)

Description: This course conveys the fundamental notions of flow of time and control of temporal behavior in computer systems, both at the hardware and the software level. The fundamental issues of synchronization, deadlock, contention, metastable states in otherwise multistable devices and related problems are described. Solutions that have been evolved, like handshaking, synchronization with semaphores, and others are described and analyzed so that the fundamental similarities between the software and hardware techniques are exposed. This course has a significant laboratory component.

Rationale: Computer systems, both hardware and software, depend heavily upon synchronization and concurrency control. This is because they deal with implementations in which there is real parallelism. This course makes precise many of the terms (e.g., simultaneous, parallel) that we use to talk about time. These terms hide many implicit assumptions about the temporal behavior of computer systems that we make when thinking about hardware and software. Some of these assumptions are correct, while others are convenient simplifications designed to make intractable problems manageable. There are essential difficulties with our concepts of how systems behave in time, some of which are due to deep mismatches between our intuitions about time and the reality as modelled by physicists.

Objectives: At the end of this course, a student will be able to:

- ► Design, implement, and reason about software with synchronization
- ► Design, implement, and reason about hardware synchronization circuits

Ideas: This course will be the primary carrier of the following:

- ► Concepts of concurrency
- ► Concepts of nondeterminism
- ► Cooperating processes
- ► Synchronization (handshaking, semaphores, monitors, etc)
- ► Asynchronous and self-timed systems
- ► Metastable state problem, deadlock, contention

It will reinforce or share responsibility for:

- ► Abstract machine models
- ► Combinatorial circuits with feedback, memory circuits
- ► Addressing, data representation, and storage

Topic Outline:

In addition to the formal content, this course is intended to teach students how to evaluate systems and ideas.

1. Time
   a. Continuous time, Physics
      - ► D.C. circuits
      - ► Propagation delay
      - ► Transmission lines
   b. Discrete time
      - ► Clocks

- ► Events and orderings on events

c. Simultaneity
d. Concurrency
e. Formal Models
- ► Nondeterministic automata
- ► Temporal Logic
- ► Linear Time
- ► Branching Time

2. Hardware and time
a. Synchronous, asynchronous, self-timed
b. Metastable states and deadlock
c. Handshaking, synchronization
d. Interrupts
e. Multiprocessor organization
f. Indivisible instructions (test and set, compare and swap ...)
g. Clock generation and distribution
h. Interfacing and data communication protocols

3. Software and time
a. Cooperating Processes
b. P and V
c. Transactions and atomicity
d. Deadlock, livelock - spaghetti-eating philosophers
e. Blocking, semaphores
f. Spin Locks
g. Monitors
h. Ada synchronization constructs
i. Time Clocks
j. I/O and Data Communication
k. Inter Process Communication
l. Network Communication
- ► Name
- ► Address
- ► Route

References:

- ► C.G. Bell and J.C. Mudge, "The Evolution of the PDP-11"; Chapter 16 of C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering: A DEC View of Hardware Systems Design* [4].
- ► M. Ben-Ari, *Principles of Concurrent Programming* [5].
- ► E.W. Dijkstra, *Cooperating Sequential Processes* [18].
- ► A.N. Habermann, *Introduction to Operating System Design* [30].
- ► C.A.R. Hoare, *Communicating Sequential Processes* [35].
- ► R.C. Holt, G.S. Graham, E.D. Lazowska, and M.A. Scott, *Structured Concurrent Programming with Operating System Applications* [36].
- ► C. Seitz, *System Timing*; Chapter 7 of C. Mead and L. Conway, *Introduction to VLSI Systems* [53].

Implementation Considerations and Concerns: The best presentation of the material in this course depends on a careful balance between hardware, software, and theoretical issues. This requires a great deal of breadth from the instructor, something that may be difficult to find.

8.2.6 Comparative Program Structures [311]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]

Description: This course covers a variety of common program organizations and program development techniques that should be in the vocabulary of a competent software engineer. The student learns advanced methods for programming-in-the-small including implementation of modules to given specifications and some common program organizations. The course also covers techniques for reusing previous work (e.g., transformation techniques and generic definitions) and elementary design and specification.

Rationale: In previous courses, students have studied data structures, some programming languages, and some particular ways to organize and develop programs by putting individual statements together to make procedures. In Software Engineering courses they will study ways to put modules together to form systems. This course fills in the middle ground — ways to put code fragments together to make modules. This course thus presents a methodology for medium-scale program development. In the same way that FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212] and ALGORITHMS AND PROGRAMS [330] teach comparative data structures and LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320] teaches comparative programming languages, this course teaches comparative program organizations — the program skeletons that good programmers carry in their heads.

Objectives: At the end of this course, a student will be able to:

- Select an appropriate program organization for a problem of moderate size (5-10 pages) and implement a program competently
- Use pre-existing definitions and development tools to expedite the development of such programs
- Implement a module to a given specification

Ideas: This course will be the primary carrier of the following:

- Standard program organizations
- Program development methodology for medium-scale programs
- Systematic methods for creating and connecting software components

It will reinforce or share responsibility for:

- Understanding that programs can be constructed or modified by other programs
- Engineering concerns in software construction

Topic Outline:

1. Ideas
   - Notion of a program organization paradigm
   - Advanced programming techniques and methodology
   - Devising and evaluating alternative implementations
   - Creating software by modifying software (when large-grain transformations emerge, they go here)
   - Engineering concerns: reliability, reasoning about correctness and efficiency, informed selection among alternative implementations
2. Program Organizations: Examples drawn from:
   - (Abstract) data types (use for connection to FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212])
   - Pipes/filters
   - Table-driven interpreters

- ► Pattern-matching systems: production systems
- ► State machines
- ► Command-language processors (e.g. editors)
- ► Constraint systems
- ► Cooperating concurrent processes
- ► Object-oriented programming (message-passing systems)

3. Methodology
   - ► Evaluation and selection of implementation alternatives
   - ► Specification (formal and informal)
   - ► Generic definitions and macros
   - ► Transformation systems
   - ► Reusable software
   - ► Program development systems

References:

- ► O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* [15].
- ► D. Gries, *The Science of Programming* [28].
- ► P. Hibbard, A. Hisgen, J. Rosenberg, M. Shaw, and M. Sherman, *Studies in Ada Style* [33].
- ► B.W. Kernighan and P.J. Plauger *Software Tools in Pascal* [39].

Resource Requirements (software):

- ► Templates and worked-out examples of the various program organizations included in the course.
- ► A software development environment to make assignment of large programs feasible.
- ► If case studies are used, the software being studied should be available for modification or measurement.

Implementation Considerations and Concerns:

- ► For the time being, this should be a lab course. It might, for example, be organized as a set of case studies, much in the style of data structure courses, with sample program organizations from the list above, abstract specifications and implementation alternatives for each, and evaluations of the result. As the formal theories that support these organizations grow, it should become more of a lecture course.
- ► The course should cover cases in both Pascal and LISP.

8.2.7 Languages, Interfaces, and their Processors [320]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]

Description: This course examines the nature of programming languages and the programs that implement them. It covers the basic elements of programming language organization and implementation; it also touches on the design of interactive interfaces. The emphasis is on the elements of general-purpose programming languages that are common to many programming languages and on ideas that are also applicable to specialized systems. Implementation techniques covered include lexical analysis, simple parsing, semantic analysis including symbol tables and types, and interpretation for elementary arithmetic expressions. Programming projects include a simple interpreter and an interactive program.

Rationale: The traditional courses on programming languages are a "comparative languages" course and a "compiler" course; the compiler course also served as an example of a medium-sized system with a well-understood structure. Usually, however, the more fundamental objectives of understanding languages and system organization get lost in the press of, for example, learning three new programming languages or constructing a complete compiler for an Algol-class language. In addition, these courses omit a number of topics that are now of increasing importance to computer science. Such topics include the use of coherent systems of software development tools, human factors considerations for interfaces, engineering considerations concerning usability and reliability, and improved theoretical approaches to specifications of computations and policies. In addition, there has been a shift in the needs of the students: an increasing number of them end up creating programs to be used by laymen rather than modules that will be incorporated in large software systems.

We propose a three-course sequence that substantially revises the previous pair of courses:

- LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320] deals with the structure and organization of programming languages and with the interface between programs and people. Since it is intended for a wide spectrum of students, it emphasizes techniques applicable to specialized interfaces as well as to general-purpose languages.
- TRANSDUCERS OF PROGRAMS [420] centers on the notion that programs should be manipulated by other programs as well as by people. The major examples are drawn from compilers, but tools for constructing compilers and a variety of techniques for re-using code are explored. Examples include code generation, macro/generic definition expansion, test data generation, use of integrated editors and program development data bases.
- ADVANCED PROGRAMMING LANGUAGES AND COMPILERS [421] is concerned with programming language topics of specialized interest. These include comparative study of programming languages, optimization techniques, and the interaction between language design and implementation.

This section describes the first course (320) of this programming language sequence. The theme of this course is the description of computations. These descriptions are used both by humans and by computers. They may be either static, as in a conventional programming language, or dynamic, as in an interactive interface. The course covers both notations and software systems that process the notations. Notations of interest include programming languages, specification formalisms, software interfaces such

as command languages, and interactive systems such as graphics processors. Students are assumed to enter knowing Pascal and LISP plus several specialized systems such as operating system command languages, editors, text formatters, and electronic message systems, so there is a base of common experience to provide examples.

Specialized languages and packages account for an increasing share of modern software. They are especially important to naive users, who may use general-purpose languages rarely or not at all. As a result, there is a premium on good design and reliable implementation of these specialized systems. Design and implementation techniques developed for general-purpose programming languages apply as well to the specialized ones; the transfer is not, however, so obvious that students will make it without help. We believe that the changing style of computing justifies a shift in emphasis in the courses. Further, a shift to smaller languages will provide a large set of examples whose size is more manageable than many of the examples now used in compiler courses. The emergence of software development tools for constructing parts of language-like systems is also an advantage.

Objectives: At the end of this course, a student will be able to:

- ► Learn new programming languages or system interfaces with reasonable investment of effort
- ► Design and implement usable, reliable interfaces for small systems such as editors or data-management programs
- ► Be able to evaluate language or interface designs
- ► Program competently in the programming languages taught in the courses

Ideas: This course will be the primary carrier of the following:

- ► General structure and organization of programming languages
- ► Criteria for evaluating languages, including human factors concerns
- ► Implementation: data structures and algorithms for lexical analysis, symbol tables, and simple parsing

It will reinforce or share responsibility for:

- ► Abstraction methodologies
- ► The impact of notations on approaches to problems
- ► Introduction to several general-purpose languages of rather different character (Snobol, APL, etc)

Topic Outline:

This course includes a comparison of several general-purpose languages, the general principles of language and interface design, evaluation criteria and human factors concerns.

1. The concept of language
   - ► Syntax vs semantics vs pragmatics
   - ► Language as a communication/interface medium
   - ► Language (notation) as a means of shaping ideas
2. Introduction to a third programming language
   - ► This language should be fairly different from Pascal and LISP
   - ► Teach a characteristic core subset in a week
   - ► Overlap anatomy of languages lectures with actually learning the language (1-2 weeks)
3. Defining programming languages (review of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211])
   - ► Regular expressions for lexical structure

- ► BNF for syntax
- ► Existence of formal semantics methods

4. Classical anatomy of programming languages
   - ► Note that this level (sophomore/junior), one enumerates the design alternatives rather than expecting the comparison and evaluation to really sink in. In a certain sense, this is elementary anatomy; comparative anatomy comes in ADVANCED PROGRAMMING LANGUAGES AND COMPILERS [421], and genuine depth in language design is a graduate issue.
   - ► Structure of algorithmic languages
   - ► Simple statements: statement sequencing; iteration and recursion; conditionals
   - ► Data structures and declarations
   - ► Addressing mechanisms (virtual addresses, indirection): variables, names vs values, scope, binding, extent; storage allocation — implicit and explicit, structure and management
   - ► Types (and what is typed: values, variables,...): abstract data types
   - ► Abstraction mechanisms: functions, procedures, and exception handlers
5. Evaluation criteria
   - ► Simplicity, orthogonality, abstraction, etc
   - ► Language complexity vs implementation complexity
   - ► Human factors — emphasis on the sorts of interfaces students use and create, not just on programming languages. This means graphics and human factors, among other things.
6. Effect of programming language on program organization
   - ► Structured programming
   - ► Recursion and list structures
   - ► Applicative programming
   - ► Shifting program organization paradigms with shift in language
7. Interactive program organization
   - ► Screen handling
   - ► Simple 2-dimensional interface design
8. Special-purpose languages as languages
   - ► Compare structure (control, data, etc) with general-purpose languages.
   - ► Relation between complexity of language and implementation, ease of use
   - ► Examples, from: spreadsheet program, robot control language, word/text processing language, database query language, editor, etc
9. Project: build an interpreter with full-screen display, or an interface for a client application provided by the instructor. Study implementation techniques while students work on project.
10. Processors and implementation techniques (survey)
    - ► Compilers, interpreters, linkers
    - ► Lexical analysis, parsing, symbol tables, display management
    - ► Expression evaluation
    - ► Run-time representations, structures, and types
    - ► Storage management, including reference counts and garbage collection
    - ► Code generation; role of optimization
    - ► Macro processors, pseudo-operations, cross-references, other good assembler techniques
11. Specific implementation techniques
    - ► These are selected because of their applicability outside the world of compilers for general-purpose languages.
    - ► Level of aspiration is complete treatment of regular languages plus the interpretation of arithmetic expressions as a special case of context-free.
    - ► Lexical analysis (review of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211])
    - ► Simple BNF (review of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211])
    - ► Parser (generated with a tool)
    - ► Elementary semantic analysis (symbol table, types on nodes of parse tree)

References:

► A.V. Aho and J.D. Ullman, *Principles of Compiler Design* [2]. (The "Dragon Book")

- R.E. Griswold and M.T. Griswold, *A Snobol Primer* [29].
- H. Ledgard and M. Marcotty, *The Programming Language Landscape* [47].
- B.J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation* [51].
- J.E. Nicholls, *The Structure and Design of Programming Languages* [58].
- S. Pakin, *APL\360 Reference Manual* [59].
- T.W. Pratt, *Programming Languages: Design and Implementation* [64].
- R.D. Tennent, *Principles of Programming Languages* [77].
- N. Wirth, *Algorithms + Data Structures = Programs* [86]. (especially Chapter 5)
- Text on command languages, human engineering, and interactive systems

Resource Requirements (software):

- Sample systems
- Compiler-construction tools
- Compiler lab: modules for lexer, symbol table, ... that can be composed to make a complete compiler. Ditto for components of an interpreter.

Implementation Considerations and Concerns:

- It is very hard to generalize about languages without at least 3 in hand. Pascal and LISP come from the 212 prerequisite; assembler comes from 240. Students should already know several special-purpose languages, such as the operating system command language, the editor, the text formatter, the mailer. Examples should draw heavily on these. If time permits, another interactive system such as VisiCalc could be taught.
- In addition, 240 is a prerequisite in order to ensure that students can appreciate the language-as-abstract-machine viewpoint and to provide a feeling for the role of the representation shift between a high-level language and a machine language.
- The emergence of program development tools affects us in two ways: first, they allow for larger, more realistic examples and introduce students to the tools of the real world; second, they make it possible to use the effects of the tools in the first course and defer the mechanism (e.g., parsing) to a later course.
- Balance should be 50% what a language is, 50% broadly useful implementations.
- See notes in topic outline

8.2.8 Algorithms and Programs [330]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]

Description: An introduction to abstract algorithms and to their design, analysis, and realization. The goal of the course is to develop skill with practical algorithm design and analysis techniques and to develop the ability to apply these techniques to the construction of real systems.

Rationale: The treatment of algorithms begins in FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212] with the algorithms that manipulate data structures; it continues through ALGORITHMS AND PROGRAMS [330] with a pragmatic view of the application of algorithmic ideas to reals systems and concludes with an abstract treatment of algorithms in ADVANCED ALGORITHMS [430]. This sequence provides a solid grounding in algorithm design and analysis.

In ALGORITHMS AND PROGRAMS [330] the student is presented with a collection of useful algorithms and with design and analysis techniques. The context is realistic enough to require meaningful choices about the application of these techniques. The point of view here is that algorithms (the abstractions) provide models that can be imposed on nasty real problems. Like all models, they do not match the real problems exactly, and some skill is required to use them well. Students need to learn a number of these models to use as tools; they also need practice in applying them to real problems.

Course ADVANCED ALGORITHMS [430], on the other hand, takes a more abstract view; it is directed towards teaching the fundamental ideas of problem diagnosis and algorithm design. This division of responsibilities is intended to provide all students with good problem solving skills for concrete algorithmic problems and to enable interested students to pursue topics in abstract algorithms in substantial depth.

Objectives: At the end of this course, a student will be able to

- ► Choose algorithms appropriate for many common computational problems.
- ► Analyze the use of computational resources by programs.
- ► Exploit constraints and structure to design good algorithms.
- ► Apply algorithmic ideas to write fast programs.
- ► Select appropriate tradeoffs for speed, space, and reliability.

Ideas: This course will be the primary carrier of the following:

- ► Algorithm design principles
- ► Analysis techniques for algorithms
- ► Pertinence of abstract algorithms to program construction

Topic Outline:

1. Data structures and algorithms (review of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212])
   - ► Queues, stacks, graphs, heaps, balanced binary trees, priority queues
2. Analysis of algorithms
   - ► What to analyze
   - ► Order arithmetic
   - ► Software timing and monitoring tools
3. Problem assessment and algorithm design techniques
   - ► Weak methods: local search, heuristic search, evaluation functions

- ► Exploiting structure
- ► Constraints
- ► Problem reformulation: time vs. space, precomputation, dynamic data updating
- ► Search: connected components, shortest paths
- ► Divide-and-Conquer: binary search, sorting, selection
- ► Greedy Method: Dijkstra's algorithm, spanning trees
- ► Dynamic Programming: path algorithms, traveling salesman
- ► Probabilistic algorithms

4. Implementation considerations
   - ► Choosing representations
   - ► Pragmatic constraints: speed vs. maintainability
   - ► Improving performance: bottlenecks, profiling, gross estimates
5. NP-completeness
   - ► Satisfiability, clique, hamiltonian circuits, etc.
6. Particular Algorithms. Examples will be selected from the following classes:
   - ► Mathematical Algorithms: arithmetic, random numbers, polynomials, Gaussian elimination, curve fitting, integration
   - ► Sorting: elementary sorting methods, Quicksort, radix sorting, priority queues, selection and merging, external sorting
   - ► Searching: elementary searching methods, balanced trees, hashing, radix searching, external searching
   - ► String Processing: string searching, pattern matching, parsing, file compression, cryptology
   - ► Geometric Algorithms: elementary geometric methods, finding the convex hull, range searching, geometric intersection, closest point problems
   - ► Graph Algorithms: elementary graph algorithms, connectivity, weighted graphs, directed graphs, network flow, matching
7. Advanced Topics: A selection from
   - ► Algorithm machines: general approaches, perfect shuffles, systolic arrays
   - ► The Fast Fourier Transform: evaluate, multiply, interpolate, complex roots of unity, evaluation and interpolation at the roots of unity, implementation
   - ► Dynamic Programming: knapsack problem, matrix chain product, optimal binary search trees, shortest paths, time and space requirements
   - ► Linear Programming: linear programs, geometric interpretation, the simplex method, implementation
   - ► Parallel algorithms: sorting, searching, in parallel
   - ► Exhaustive Search: exhaustive search in graphs, backtracking, permutation generation, approximation algorithms
   - ► NP-complete problems: deterministic and nondeterministic polynomial-time algorithms, NP-completeness, Cook's theorem, some NP-complete problems

References:

- ► A.V. Aho, J.D. Hopcroft, and J.E. Ullman, *Data Structures and Algorithms* [3].
- ► A.V. Aho, J.D. Hopcroft, and J.E. Ullman, *The Design and Analysis of Computer Algorithms* [1].
- ► J.L. Bentley, *Writing Efficient Programs* [6].
- ► J.L. Bentley, *Programming Pearls* [7].
- ► D. Gries, *The Science of Programming* [28].
- ► D.E. Knuth, *The Art of Computer Programming* [40, 43, 41].
- ► B. Lampson, *Notes on System Design* [45].
- ► E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms* [65].
- ► R. Sedgewick, *Algorithms* [70].

Resource Requirement (software):

- ► Library of data types and implementations

- ► Test bed for timing and program development
- ► Timing support
- ► Sample data sets

Implementation Considerations and Concerns:

- ► This course would be required for a major because, of all the 300-level courses, it most clearly captures the interplay of theoretical ideas with practical programming problems. Since so many traditional algorithms courses already exist, there will be a tendency for this course to drift toward those models. It is important to resist that drift.
- ► The algorithms listed in the outline are presented as a menu of examples. It is not possible to cover them all in one course, and the integrative material should not be slighted in favor of a few extra algorithms.
- ► Some algorithms may be covered in other courses such as numerical linear algebra or graph theory.

8.2.9 Formal Languages, Automata, and Complexity [350]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]

Description: This course introduces the basic ideas of formal languages, computability and complexity theory. It contains only the more fundamental material on complexity, to give the student an overall feel for the topic; the more advanced aspects are covered in an advanced course COMPLEXITY THEORY [451]. Some introductory material will be assumed from the pre-requisite course FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211].

This course begins with an introduction to finite state automata and their relationship to classes of formal languages. A finite automaton is a mathematical model of a finite-state system; computer science has many examples of finite state systems. Formal languages are of great importance, notably in defining programming languages and in formalizing the notion of parsing. The material of this course is primarily concerned with the relationship between the various classes of language and various types of automaton. Thus it is shown that particular classes of automata recognize particular types of formal language. Since this is a first course dealing in detail with these concepts, it is important to emphasize these ideas in a strongly applied context, to bring out the connections with areas such as software support (parsers for programming languages, simulators for automata, for example).

Computability is concerned with characterizing the class of problems that can be solved, in a well defined sense, by a computer. In complexity theory the interest lies in how much space or how much time is required to solve a problem (relative to the size of the problem); the recognition problem for various formal languages serves to provide examples of problems of various degrees of complexity. Again, it is important to emphasize the practical applications of the results.

Rationale: Automata and the related notion of computability by an automaton are fundamental to many branches of Computer Science. Likewise, formal languages underpin much work on parsing, programming language theory and practice. This course can be organized around the theme of formal languages, their generation by grammars and their recognition by finite state machines. Problems associated with formal languages, such as ambiguity, can be used to illustrate the notions of decidability and undecidability; the various recognition problems for languages serve to illustrate the problems of various degrees of complexity.

Objectives: At the end of this course, a student will have a feeling for the theoretical limitations of computers, and how restrictions on working space and running time affect the capability of computers to solve problems. He will have an idea of how formal languages are used in theory and in practice. This will help in later courses such as ADVANCED PROGRAMMING LANGUAGES AND COMPILERS [421].

Ideas: This course will be the primary carrier of the following:

- ► Formal languages
- ► Automata of various kinds
- ► Equivalences between machines and corresponding languages
- ► Impact of notation on the way we think
- ► Recognition and generation problems
- ► Computability by abstract devices

- ▸ Elementary aspects of Complexity theory

It will reinforce or share responsibility for:

- ▸ Computability and decidability
- ▸ Inductive definitions and inductive arguments

Topic Outline:

1. Regular Languages and Finite-State Automata:
   - ▸ Recognition of a language by an automaton
   - ▸ Regular sets as the languages recognized by finite automata
   - ▸ Regular expressions, finite automata
   - ▸ Equivalence of deterministic and nondeterministic finite automata
   - ▸ Minimization of a finite automaton
   - ▸ Algorithm for equivalence of finite automata (decidable problem)
   - ▸ The Pumping Lemma and its use in proving non-regularity
   - ▸ Closure properties of regular sets
   - ▸ Algebraic characterization of regular sets
   - ▸ Myhill-Nerode theorem and its uses
2. Context-free languages as the languages recognized by pushdown automata
   - ▸ Context-free grammars, pushdown automata
   - ▸ Examples of CFLs which are not regular
   - ▸ Undecidability of equivalence problem for CFLs
   - ▸ Undecidability of ambiguity problem for CFLs
   - ▸ Closure properties of CFLs
   - ▸ Properties of grammars: emptiness, ambiguity, LL, LR
3. Computability
   - ▸ Algorithms: intuitive notion of algorithm as effective procedure
   - ▸ Formalization of the notion of algorithm
   - ▸ Turing machines
   - ▸ Register machines
   - ▸ Computable functions, sets
   - ▸ Computable by Turing machine iff computable by register machine
   - ▸ Church's thesis
4. Recursive function theory
   - ▸ Recursive functions and sets
   - ▸ Recursively enumerable sets
5. Decidable and undecidable problems
   - ▸ Halting problem
   - ▸ Post correspondence problem
   - ▸ Rice's theorem
   - ▸ Reduction of a problem to an undecidable problem to show undecidability
   - ▸ Diagonal arguments
   - ▸ Examples drawn from context free languages (CFLs) (e.g., equivalence problem for CFLs and ambiguity problem for CFLs)
6. Universality and recursion
   - ▸ Godel numbering
   - ▸ Universal Turing machines
   - ▸ Kleene's T-predicate
   - ▸ The s-m-n theorem
   - ▸ The recursion theorems
   - ▸ Use of universal machines (e.g., to show computability of Ackermann's function)
7. Complexity Theory:

- ► Distinction between computability and complexity
- ► Space complexity
- ► Time complexity
- ► Complexity relative to deterministic and nondeterministic computation
- ► Survey of the time and space hierarchies: PTIME, NP, RPTIME, co-NP, PSPACE, etc.
- ► Cook's theorem
- ► Examples of problems known to lie in each hierarchical level: graph isomorphism, recognition problems

Primary References:

- ► N.J. Cutland, *Computability: An Introduction to Recursive Function Theory* [14].
- ► J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* [37].
- ► H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation* [48].

Secondary References:

- ► M. Minsky, *Computation: Finite and Infinite Machines* [54].
- ► H. Rogers, *Theory of Recursive Functions and Effective Computability* [67].

Resource Requirements:

- ► Grammar support tools: parser generators for various classes of grammars, drivers for testing grammars

Implementation Considerations and Concerns:

- ► The outline contains a lot of material, possibly too much for a one-semester course. It may be necessary to extract the more advanced material on computability and design an advanced course covering this, leaving only the basic material on computability here in FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350].

8.2.10 Logic for Computer Science [351]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]
One 300-level mathematics or theoretical computer science course

Description: The basic results and techniques of Logic are presented and related to fundamental issues in computer science.

Rationale: Logic provides essential foundations for our activity in computer science. It teaches us to distinguish between abstraction and realization and, in particular, between language and meaning. Programming languages are formal languages; the techniques we use to give them meaning and to reason about them find their foundations in logic. Logic also teaches us how to reason about the world by manipulating symbols; this is directly analogous to the activity of computation. Finally, the basic results of logic reveal the inherent limitations on our activity of formal reasoning.

Objectives: At the end of this course, a student will be able to:

- ► Understand the role of formalization and formal reasoning in computer science.
- ► Be familiar with the basic techniques and results of mathematical logic.

Ideas: This course will be the primary carrier of the following:

- ► Fundamental concepts and results from logic
- ► The notion of formal reasoning

It will reinforce or share responsibility for:

- ► Syntax and semantics
- ► Computability
- ► Reasoning about programs

Topic Outline:

1. Syntactical Structures and Computability
   - ► Lists and Functions — The basis for a simple model of computation
   - ► An analysis of the notion of computation
   - ► Syntactic structures in logic and programming
2. Formal reasoning
   - ► Rules of inference and recursive enumeration
     - > Natural deduction
     - > Hoare's logic
   - ► Propositional and predicate calculus
   - ► The formalization of mathematical reasoning
     - > Reasoning about programs
3. Semantics and completeness
   - ► Structures and truth
   - ► Completeness of first-order logic
   - ► Church's thesis revisited
   - ► Semantics and reasoning about programs
4. Incompleteness and undecidability
   - ► Decidability and undecidability
   - ► Presburger arithmetic
5. The incompleteness of certain systems

References:

- G.S. Boolos and R.C. Jeffrey, *Computability and Logic* [11].
- H. Enderton, *A Mathematical Introduction to Logic* [21].
- D. van Dalen, *Logic and Structure* [82].

Resource Requirements:

Implementation Considerations and Concerns:

8.2.11 Introduction to Artificial Intelligence [360]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
or INFORMATION PROCESSING PSYCHOLOGY AND AI [PSY 213]

Description: This course teaches the fundamentals of artificial intelligence, including problem solving techniques, search, heuristic methods, and knowledge representation. Ideas are illustrated by sample programs and systems drawn from various branches of AI. Small programming projects will also be used to convey the central ideas of the course.

Rationale: This course provides a single jumping off point for students in ARTIFICIAL INTELLIGENCE — COGNITIVE PROCESSES [460] and ARTIFICIAL INTELLIGENCE — ROBOTICS [461], in order to familiarize the student with both sides of the simulation/performance issue. It provides students with an overview of the field without requiring the math background used in the Robotics courses or the psychological emphasis used in the Cognitive Processes course.

Objectives: At the end of this course, a student will be able to

- ► Program large systems in Lisp
- ► Use AI techniques to solve difficult problems
- ► Read and understand AI literature

Ideas: This course will be the primary carrier of the following:

- ► Advanced Lisp techniques
- ► Weak methods and problem solving
- ► Knowledge representation

It will reinforce or share responsibility for:

- ► Lisp programming beyond the level of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
- ► Production systems and embedded languages (also covered in COMPARATIVE PROGRAM STRUCTURES [311] and LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320])
- ► Knowledge representation (also covered in BIG DATA [413])
- ► Search (also covered in ALGORITHMS AND PROGRAMS [330])

Topic Outline:

1. Introduction
   - ► History
   - ► Intellectual Issues
2. Implementation
   - ► ATNs
   - ► Agenda Control Structures
   - ► Data Driven Programming
   - ► Discrimination Nets
   - ► Frames
   - ► Lisp
   - ► Semantic Nets
3. Techniques
   - ► Exploiting Constraints
   - ► Heuristic Programming

- ► Inference and Inheritance
- ► Knowledge Representation
- ► Minimax and alpha-beta
- ► Production Systems
- ► Weak Methods

4. Applications
   - ► Analysis
   - ► Data Base
   - ► Design
   - ► Diagnosis
   - ► Game Playing
   - ► Natural Language
   - ► Speech Recognition
   - ► Theorem Proving
   - ► Vision (esp. Waltz line labelling)
   - ► Knowledge-based systems
5. Concepts
   - ► Discovery
   - ► Learning
   - ► Planning

References:

- ► E. Rich, *Artificial Intelligence* [66]
- ► P.H. Winston and B.K.P. Horn, *LISP* [85].

Resource Requirements:

- ► Copies of the example programs (Eliza, Mycin, *etc.*)
- ► Lisp programming environment
- ► Lisp cycles
- ► Lisp cycles
- ► Lisp cycles

Implementation Considerations and Concerns:

- ► The instructor has a responsibility to provide a *broad* overview of AI.
- ► Lisp coverage in FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212] may be spotty, review might be necessary.

8.3 Advanced Computer Science Courses

These courses are of specialized interest to computer scientists. They are suitable for a Master's program as well as for advanced students in a Bachelor's program.

In addition to the courses we define here, some of the content of computer science as described in Chapter 4 may be taught in departments other than computer science. These departments include mathematics, electrical engineering, psychology, and others. We have generally avoided designing courses that cover material taught at Carnegie-Mellon in these other departments.

To show more complete coverage of computer science, however, we list here the titles of courses that should be jointly listed by computer science and another department. Catalog descriptions for these courses appear in Chapter 9.

- MODERN ALGEBRA [MATH 473 / CS 452]
- LARGE-SCALE SCIENTIFIC COMPUTING [MATH 712 / CS 453]

Some of the courses outlined in this section may also be jointly listed. In particular, COMPUTER ARCHITECTURE [440] can be listed in the Electrical Engineering Department, ADVANCED ALGORITHMS [430] resembles APPLIED GRAPH THEORY [MATH 484], and COMPLEXITY THEORY [451] resembles THEORY OF ALGORITHMS [MATH 451].

8.3.1 Independent Project [400]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with Bs or better
Instructor's permission, based on acceptance of project proposal

Description: This is an independent project laboratory for the most advanced students. The student will design and construct a substantial software or hardware system under the supervision of the Project Lab faculty. Before construction of the project may proceed, a detailed design proposal must be submitted to and accepted by the faculty member running the course. A design review with the lab faculty and TAs will be held at mid-term time. A final review of the functioning system and its supporting documentation will be held at the end of the semester. The intent is to permit the best students to exercise their design skills in the construction of a real system, so good design practice and good documentation are mandatory. The production of a functioning but undocumented system will not be sufficient. The instructor may accept projects intended to last two semesters, in which case the review at the end of the first semester will be another major design review.

Rationale: Computer Scientists going out to graduate school or to the practice of programming need to be able to design and construct good systems. This means an appreciation of the difficulties of building a production-quality system, difficulties that go beyond the scope of toy systems built as part of lower level courses. This course will provide the advanced student with the opportunity to design and build a significant piece of hardware or software on his own, with experienced system builders available as instructors and TAs to consult, advise, and criticize.

Objectives: At the end of this course, a student will be able to:

- ► Design a real system
- ► Document a real system
- ► Construct a real system

Ideas: This course will be the primary carrier of the following:

- ► Independent formulation and execution of projects

It will reinforce or share responsibility for:

- ► Software Design principles
- ► Hardware Design principles

References: F. Brooks, *The Mythical Man-Month* [12]

Resource Requirements:

- ► A substantial host machine, plus access to others
- ► A well outfitted hardware/software lab
- ► A lab bench for each student

Implementation Considerations and Concerns:

- ► This course is intended to have serious intellectual content. It should not be permitted to deteriorate into a simple home for hacking.
- ► It is important that the students in this course have access to experienced system builders. The teaching of this course will be expensive in all domains, including people and inanimate resources. It would be better not to offer it than to reduce the quality.

8.3.2 Undergraduate Thesis [401]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with B's or better
Instructor's permission, base on acceptance of proposal

Description: This is an independent study and research course for the most advanced students. The student will write an undergraduate thesis or carry out a program of directed reading. Objectives for the course of study will be established by the student and a faculty advisor. With concurrence of a faculty advisor, an undergraduate thesis project may be planned for two semesters

Rationale: This course provides students with the opportunity to pursue in depth the study of a topic that is not part of the general curriculum. It is similar to INDEPENDENT PROJECT [400], but the end result is is a document rather than a system. It is expected that the student will work closely with a faculty advisor.

Objectives: At the end of this course a student will be able to:

- ► Organize the description of a collection of scientific findings and report the result in an expository technical paper
- ► Use the library for background research
- ► Do independent research

Ideas: This course will be the primary carrier of the following:

- ► Independent formulation and execution of a planned program of study.

Resource requirements:

- ► A pool of faculty to supervise students

8.3.3 Research Seminar [409]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with Bs or better

Description: Students attend the regular research seminars of the Computer Science Department and submit short written summaries.

Rationale: The Computer Science Department conducts a rich and varied set of public seminar series throughout the academic year. Undergraduates with sufficient maturity and experience in the field can benefit from attending, even if they do not completely understand the material presented. Attending these seminars is a good way to learn about very current ideas and to appreciate the scope and excitement of the field.

Objectives: At the end of this course, a student will:

- Be acquainted with some of the new ideas in computer science.
- Be able to write a short summary of a presentation on a technical topic.

Ideas: Students select seminars to suit individual preferences.

Topic Outline:

Certain regular seminar series plus selected individual seminars will be approved for this course. Each student should plan to attend an average of one seminar per week. After the seminar, the student prepares a one-page (250-500 word) summary and critical appraisal of the seminar in his or her own words. Ten of these summaries -- from any combination of approved seminars -- are required for completion of the course. An introduction to writing short technical summaries will be presented at the beginning of the course.

Resource Requirements:

- Doughnuts

Implementation Considerations and Concerns:

- This course requires an ongoing seminar series. Research seminars are fine; an undergraduate can get a sense of the nature of research and creativity without completely understanding the material.
- This course could carry about a third of the credit of a normal course. It is intended to require two to three hours per week.

8.3.4 Software Engineering [410]

Prerequisites: COMPARATIVE PROGRAM STRUCTURES [311]
LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]

Description: The student studies the nature of the program development task when many people, many modules, many versions, or many years are involved in designing, developing, and maintaining the system. The issues are both technical (e.g., design, specification, version control) and administrative (e.g., cost estimation and elementary management). The course will consist primarily of working in small teams on the cooperative creation and modification of software systems.

Rationale: This course extends the advanced program structures course by broadening the scope of attention to large-scale systems. This yields a natural progression from individual elements (statements or data structures) in FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212] through module-sized elements in COMPARATIVE PROGRAM STRUCTURES [311] to large systems. Analysis and evaluation techniques are included throughout, but the emphasis on estimation and overall efficiency is greatest here. In addition, issues of reliability, testing, and implementation and documentation of a substantial user interface will be addressed here.

Objectives: At the end of this course, a student will be able to:

- ► Understand the issues in large-scale software development
- ► Participate as a team member in such a development
- ► Write specifications for simple modules that will be combined with other modules
- ► Implement a program or module that satisfies such a specification

Ideas: This course will be the primary carrier of the following:

- ► Complexity of large-scale software and tools for dealing with it

It will reinforce or share responsibility for:

- ► Significance of tools for developing software

Topic Outline:

1. Elementary management
2. Cost estimation (of routines and larger code units)
3. Multiple people, versions, years, modules, modifications
4. Advanced design and specification; decomposition into modules
5. Programming-in-the-large
6. Properties of systems
   - ► Reliability
   - ► Generality
   - ► Efficiency
   - ► Complexity
   - ► Compatibility
   - ► Modularity
   - ► Sharing
7. System design and development principles
   - ► Design tradeoffs
   - ► Computer system reliability, speed, capacity, cost
   - ► Development methodologies and tools
   - ► Design automation

- ► Program specification
- ► Maintenance and release policy (test sites, etc.)
- ► Rapid prototyping and partial evaluation
- ► Protection and security
- ► Resource allocation
- ► System evaluation and development aids

8. Modification, planning for modification
9. Making implementation meet specifications
10. Models and modelling
    - ► What models are and how to use/construct them
    - ► Empirical vs analytic models
    - ► Validation
    - ► Specific models (at this level, introduction only)
        - > Queueing-theoretic models for operating systems and hardware
        - > Productivity and life-cycle models (esp. their limitations)
11. Monitoring tools and techniques for improving efficiency
12. Human factors, user interfaces
13. Examples of systems
    - ► Large software systems, some involving concurrency issues
    - ► Distributed systems
    - ► Compilers, operating systems
    - ► Batch vs. time-sharing systems
    - ► File management
    - ► System accounting
    - ► The multiprogramming executive (MPX) operating system
    - ► Process control
14. Current state of the art: APSEs, Gandalf, etc
15. Software systems
    - ► Systems and Utility Programs
    - ► System structure
    - ► Parallelism in operating systems
        - > Mutual exclusion
        - > Synchronization
16. Programming style and techniques
    - ► Table-driven schemes
17. Management, Societal, Economic, and Legal Aspects
    - ► Computing Economics: Acquisition and Operation
    - ► Copyrights and Patents, computer crime
18. Documentation
19. Software Systems
    - ► Memory management
    - ► File Systems
    - ► Directories
    - ► Backup and recovery
    - ► Permanent and transient data: caching, buffering, atomic transactions, stable storage
    - ► Redundancy, encoding, encryption
    - ► Data base management systems (DBMS)

References:

- ► B. W. Boehm, *Software Engineering Economics* [9].
- ► F. Brooks, *The Mythical Man-Month* [12].
- ► G. Myers, *Composite/Structured Design* [56].

- G. Myers, *Software Reliability: Principles and Practices* [55].
- M. Shooman, *Software Engineering* [71]
- E. Yourdon and L.L. Constantine, *Structured Design* [89].
- M.V. Zelkowitz, A.C. Shaw, and J.D. Gannon, *Principles of Software Engineering and Design* [90].

Resource Requirements (software):

- A program development environment will be essential.

Implementation Considerations and Concerns:

- It is imperative that students actually use the *best available tools* for version control, text editing, etc. Students will invariably draw on their experiences in actual system development rather than on what they have read or heard in lectures.
- Since the majority of learning in this course is by doing, a traditional course format may not be best. The instructor should spend time counselling teams and walking them through code-reading sessions, etc., in addition to the lectures. A known problem with many software engineering courses taught in the past is that students become involved with the project they are implementing, and ignore the material in lecture. This has been partially addressed by including a large number of (hopefully) interesting topics not usually taught in software engineering courses.
- The software engineering course currently taught was studied as a basis [38].

8.3.5 Software Engineering Lab [411]

Prerequisites: vary with the individual arrangement
SOFTWARE ENGINEERING [410]

Description: This course is intended to provide a vehicle for real-world software engineering experience. Students will work on existing software that is or will soon be in service. In a work environment, a student will experience first-hand the pragmatic arguments for proper design, documentation, and other software practices that often seem to have hollow rationalizations when applied to code that a student writes for an assignment and then never uses again. Projects and supervision will be individually arranged.

Rationale: Software engineering issues arise in software that involves many months, many programmers, many versions, and many modules. These issues are extremely hard to raise in a one-semester course; they are easier to appreciate by working with real-world projects. This course is intended to provide an opportunity for training similar to a clinical practice course in a medical school. This will require closer cooperation between the industrial work site and the university than an ordinary work-study program would need. Evaluation of students will be shared between univeristy faculty and the individual(s) managing them in the industrial organization.

Objectives: At the end of this course, a student will be able to:

- Apply software engineering principles to large, long-term projects
- Work effectively in a programming team

Ideas: This course will be the primary carrier of the following:

- Complexity of real-world systems
- Tools for dealing with that complexity

Resource Requirements:

Implementation Considerations and Concerns:

- Getting good projects and good supervision. We must be able to select the people who serve as faculty for this course.
- We must be careful not to have this become a "mindless programming for credit" course; the students must work on challenging projects that will force them to work with practicing software engineers.

8.3.6 Resource Management [412]

Prerequisites: TIME, CONCURRENCY, AND SYNCHRONIZATION [310]
PROBABILITY AND APPLIED STATISTICS [STAT 211 / CS 250]

Description: This course provides a synthesis of many of the ideas that students have learned in earlier courses. The vehicle for this synthesis is the exploration of at least one instance of a real operating system in great detail. Taking the view that an operating system is a resource manager, we will explore some resource issues and how they may be handled. The primary issues are resource classes, properties, and management policies. This course has a substantial programming laboratory component in which an existing operating system will serve as an experimental testbed.

Rationale: Many systems, such as operating systems, file systems, and data base systems, are resource management systems. Every system that involves the control of any finite resource presents the designer with resource management problems and every computer scientist will be faced with such systems many times in his career. Operating Systems are particularly rich examples of resource management systems and this course uses operating systems as examples for this reason.

Objectives: At the end of this course, a student will be able to:

- ► Understand resources and their classifications
- ► Understand and apply techniques for using, allocating, scheduling, and naming resources
- ► Understand techniques for making reliable resources from unreliable ones
- ► Understand and apply security and protection principles
- ► Analyze and evaluate the performance of resource management systems

Ideas: This course will include units that synthesize previous examples of the following ideas with examples that appear in operating systems. It bears the responsibility of showing how these ideas appear in slightly different form in a variety of areas. It will be the primary carrier of the following:

- ► Resource classification
- ► Resource allocation
- ► Binding (e.g., of decisions as well as names)
- ► Performance Evaluation

This course will share responsibility for or reinforce the following:

- ► Multilevel naming
- ► Management of concurrency.

Topic Outline:

1. Naming and Addressing
   - ► Models
     - > Distributed
   - ► Types
   - ► Scope
     - > Space Scope
     - > Extent (Time Scope)
   - ► Aliasing
2. Resource classification
   - ► Real Time

- ► CPU Time
- ► Memory
  - > Memory Hierarchy
- ► Disk
- ► Logical
- ► Real
- ► Pre-emptible
- ► Non-pre-emptible

3. Resource related activities
   - ► Allocation
     - > Paging Secondary Storage
   - ► Synchronization
   - ► Scheduling and Concurrency
     - > CPU Paging
   - ► Reliability
     - > Redundancy
     - > Atomicity
   - ► Security, Protection and Authentication
   - ► Analysis
   - ► Resource Managers
     - > Spooling
     - > Servers for networks
4. Relationship to Architecture
5. Relationship to Operating System Kernel
6. Performance Evaluation and Tuning
   - ► Review of elementary Queuing Theory
   - ► Models
     - > Analytic
     - > Simulation

References:

- ► A.N. Habermann, *Introduction to Operating Systems* [30].
- ► A. Tanenbaum, *Computer Networks* [76].

Resource Requirements (software):

- ► operating system components for software laboratory: schedulers, storage allocators, etc.
- ► Driver and simulated load and timing apparatus

Implementation Considerations and Concerns:

8.3.7 Big Data [413]

Prerequisites: LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
RESOURCE MANAGEMENT [412]

Description: The central theme of this course is the storage of large amounts of data. Topics include user data models, underlying data storage techniques, data representations, algorithms for data retrieval, specialized data manipulation languages, and techniques for providing reliability and security. Systems that permit the storage and retrieval of large amounts of data are exemplified.

Rationale: Although the topics in this course could be distributed among an algorithms course (external data storage and data representations), an operating systems course (reliability and security techniques), and a language course (data manipulation languages and models), we have chosen to incorporate them into one course for three reasons: First, the central theme of all these topics is the storage and manipulation of large collections of data. Second, the storage and manipulation of large quantities of data represents one of the major applications of computers. Third, a unified course on these topics provides an opportunity for students to focus on large systems and some well understood techniques for their organization.

Objectives: At the end of this course, a student will be able to:

- ► Understand the goals of systems that deal with large quantities of data
- ► Use certain example systems
- ► Understand some of the algorithms and data structures used to organize such systems

Ideas: This course will be the primary carrier of the following:

- ► Security techniques
- ► Reliability techniques
- ► Algorithms and data structures for external data storage

It will reinforce or share responsibility for:

- ► Explaining the use of specialized high level languages
- ► Presenting layered abstractions
- ► Naming, binding, addressing

Topic Outline:

1. Files & access methods
   - ► Sorting & searching
     - > BTrees
     - > Multi-level Storage Structures
     - > Memory Hierarchy
     - > Hashing
     - > Multi-key Organizations
   - ► Other aspects of file organization
     - > Physical allocation
     - > Organizations for availability
     - > Performance issues in file storage
2. Classical database management:
   - ► modeling at user level
     - > Utility of this level of abstraction
     - > Classical data models and languages

- ≫ Relational
- ≫ Hierarchical
- ≫ Network

> Example Models

► Detail Study of Relational Model

> Example Language: SQL
> Embedding of SQL within procedural language
> Example SQL Application

3. Topics in the storage of data

► Approaches to data integrity and reliability

> Use of redundancy
> Old master/new master schemes

► Sharing/concurrent access: consistency

> Synchronization aspects
> Recovery considerations
> Role of transactions

► Security, Privacy, and Authentication

> Capabilities vs authorization lists
> Access protection and file security
> Administrative concerns
> Role Of Encryption
- ≫ Public Key Encryption
- ≫ Private Key Encryption

4. Non-traditional databases

► Storage of "knowledge"
► Issues in Knowledge Representation
► Non-uniform data in databases

5. Emerging Public information utilities

► Library Search
► Electronic Publishing
► Teletext-type systems
► Community Bulletin Boards

References:

► C.J. Date, *An Introduction to Database Systems, Volumes I and 2* [16, 17]

► J.D. Ullman, *Principles of Database Systems* [79].

Resource Requirements:

► One or more production-quality data base systems for use and comparison.

Implementation Considerations and Concerns:

8.3.8 Transducers of Programs [420]

Prerequisites: COMPARATIVE PROGRAM STRUCTURES [311]
LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]

Description: This course studies ways to gain leverage on the software development process by using programs to create or modify other programs, by reusing previously-created software, and by using automated tools to manage the software development process. Examples are drawn from the tools locally available. Students use these tools in projects that lead to useful software components. Special emphasis is placed on the use of integrated systems of compatible tools.

Rationale: As programming is usually taught, students often form the impression that programs are always created from scratch, by hand. The major theme of this course is that programs are frequently created by and from other programs, and that this leverage is important in increasing productivity and in transmitting good techniques in the form of working software, not just by word of mouth. More specifically, a system of any size can often be factored into segments, some of which have a structure so standard that they can be build from a specification by a specialized tool.

For example, parser generators are used as tools in the prerequisite course LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320] with only a cursory introduction to the techniques encapsulated in the tool. In this course, such tools are the objects of study and, for example, the practical parsing theory needed to understand, modify, or even construct a parser or parser-generator is included. In similar fashion, this course covers libraries (both design and administration), program development environments, smart editors, and other mechanisms for using programs to construct or maintain programs. More compiler components are studied, and many of the examples are drawn from the class of tools that can be easily integrated in a system surrounding a parse-tree representation of programs.

Objectives: At the end of this course, a student will be able to:

- ► Use automated tools effectively in software development.
- ► Describe the organization of a compiler and make minor modifications to one.
- ► Add compatible tools to a unified program development system, taking advantage of existing components and using interface representations correctly.

Ideas: This course will be the primary carrier of the following:

- ► Internal representations for compilation and the possibility of using them as an interface medium
- ► Tools for program development, especially tool-building tools
- ► Relation between complexity of language and complexity of implementation; interaction of language design and system issues
- ► Compiler organization as example of medium-large system
- ► Parsing and code generation

It will reinforce or share responsibility for:

- ► Various classes of languages and their power (and the cost of processing them)
- ► Relation between syntax and semantics
- ► Practical application of formal theories

Topic Outline:

1. Re-usable software
   - ► Libraries and integrated packages — strengths and weaknesses
     - > Pascal string manipulation package
     - > I/O packages, such as Unix curses
     - > matrix manipulation and mathematical function libraries

     Certified software (e.g., the math software)
   - ► Specifications
   - ► Evolution: building programs by modifying similar programs
   - ► Program transformation
2. Tools for operating on programs
   - ► Classes of tools
     - > Tools that help you program (editors, cross-referencers, etc)
     - > Tools that help you organize programs into systems (filters, system modellers, etc)
     - > Tools that build programs (parser generators, etc)
   - ► Programmable editors and filters
   - ► Generic definitions
   - ► Program transformations
3. Use of integrated tools
   - ► Examples: whatever's available locally from tools in the spirit of Gandalf, programmer's workbench, etc. Most likely, this will be a set of tools that operate on the parse-tree representation of a program
   - ► Tools: editors, program development data base, documentation generators,
4. Construction of integrated tools
   - ► What goes on inside a front-end generator?
     - > Example connects to previous course LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
     - > Practical parsing theory — why the limitations on the grammars arise
     - > How that theory is used in (and affects) the implementation
   - ► Bootstrapping
   - ► More about compilers: code generation, linkers
   - ► Test data generation
   - ► Verification condition generation
5. Projects:
   - ► Students should have a project in which they must factor a system into segments that can be generated by tools.
   - ► Students should write a simple tool to produce code from specifications for simple factored segments.

References:

- ► A.V. Aho and J.D. Ullman, *Principles of Compiler Design* [2]. (The "Dragon Book")
- ► D. Gries, *Compiler Construction for Digital Computers* [27].
- ► B.W. Kernighan and P.J. Plauger, *Software Tools in Pascal* [39].

Resource Requirements:

- ► Software development tools to form a laboratory that is both rich enough to illustrate the principles and simple enough for undergraduate course projects.
- ► A demonstration compiler for students to modify.
- ► Examples of useful subroutine libraries.

Implementation Considerations and Concerns:

- ► This course is at present a bit speculative. We believe that enough material already exists to teach it now, but this part of the discipline is moving rapidly. It is important to be sure that the course remains flexible for a few years so that the best of current understanding can be included.

8.3.9 Advanced Programming Languages and Compilers [421]

Prerequisites: FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
TRANSDUCERS OF PROGRAMS [420]

Description: This course is intended for students seriously interested in the construction of compilers for general-purpose programming languages. The student studies an optimizing compiler as an example of a well-organized system program, studies algorithms and data structures appropriate to the optimization process, examines code generators, optimizers, and their interactions. The student also studies comparative programming languages with emphasis on the interaction between language design and implementation considerations. Compiler-generator technology is used to build a compiler, thereby demonstrating the use of system-building tools.

Rationale: This is the third course that contains material from the traditional compiler course (the rational for the sequence is given in the description of LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]). Techniques that are broadly useful for interfaces to interactive programs have been moved into LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]. Techniques for which good automated tools exist have been at least introduced in TRANSDUCERS OF PROGRAMS [420]. This course addresses the techniques that are specialized to optimizing compilers. In addition to the traditional content, it will cover the use of automated tools for compiler construction and advanced language design topics.

Objectives: At the end of this course, a student will be able to:

- ► Participate competently in the construction of production-quality compilers using modern compiler-construction techniques and tools
- ► Identify language features and combinations of features that constrain or simplify implementation

Ideas: This course will be the primary carrier of the following:

- ► Understanding implementation techniques for programming languages
- ► Using new data structures introduced in the course
- ► Applying new tools introduced in the course

It will reinforce or share responsibility for:

- ► Understanding the components of a medium-sized system and how they interact
- ► Programming language design issues
- ► Applying theoretical techniques in practice

Topic Outline:

1. Compiler as an example of a complex medium-sized system
2. Intermediate representations for processing programs
3. Compiler-compiler technology
4. Implementation issues for programming languages
   - ► Lexical analysis, parsing, and semantic analysis (revisited)
   - ► Code generation
   - ► Global program analysis and optimization
   - ► Optimization
   - ► Interpretation

- ► Storage allocation, garbage collection
- ► Input-output

5. Techniques applied
   - ► AI search techniques
   - ► graph theory
   - ► data flow
   - ► others reflecting current research
6. Advanced topics in programming language organization and design
   - ► Interaction among design decisions [parameter binding rules, rules for assignment, etc.]
   - ► Interaction between language design decisions and implementations
   - ► Kinds of programming languages (survey)
     - > General-purpose programming languages
     - > Applicative vs imperative
     - > Assemblers, macros
     - > Very high-level languages
     - > Systems implementation languages
     - > Special-purpose languages
     - > Production systems
     - > Object-oriented languages
     - > Query languages
     - > Graphical interaction
     - > Special-purpose and application-based systems

References:

- ► A.V. Aho and J.D. Ullman, *Principles of Compiler Design* [2]. (The "Dragon Book")
- ► D. Gries, *Compiler Construction for Digital Computers* [27].
- ► R.L. Wexelblat, *History of Programming Languages* [83].
- ► W. Wulf et al, *The Design of an Optimizing Compiler* [87]

Resource Requirements (software):

- ► Automatic generators for compiler components such as lexical analyzers and parsers.
- ► Instances of other components of a compiler (symbol table module, various optimization modules)
- ► The objective is a software lab similar to a physics lab: the student "checks out" selected apparatus for an experiment, then assembles it and measures the result.

Implementation Considerations and Concerns:

8.3.10 Advanced Algorithms [430]

Prerequisites: ALGORITHMS AND PROGRAMS [330]
COMBINATORIAL ANALYSIS [MATH 301 / CS 251]

Description: A second course in the design and analysis of algorithms.

Rationale: This course is intended to familiarize the student with the unifying principles and underlying concepts of algorithm design and analysis. It extends and refines the algorithmic concepts introduced in ALGORITHMS AND PROGRAMS [330]. Here a more abstract view is taken, with emphasis on the fundamental ideas of problem diagnosis, design of algorithms, and analysis. The course assumes familiarity with material on combinatorial analysis.

Objectives: At the end of this course, a student will be able to:

- ► Design efficient algorithms
- ► Analyze the performance of algorithms

Ideas: This course will be the primary carrier of the following:

- ► Lower bounds
- ► Optimization

It will reinforce or share responsibility for:

- ► Analysis of algorithms
- ► Complexity Theory

Topic Outline:

1. Data structures
   - ► Lower bound arguments
   - ► Recurrences
   - ► Union-find
2. Graph Algorithms
   - ► Topological sort
   - ► Biconnectivity
   - ► Matching
   - ► Maximum Flow
3. Algebraic algorithms
   - ► Strassen's algorithm
   - ► Transitive closure
   - ► Chinese remainder
   - ► Four Russian's algorithm
   - ► Fast fourier transform
   - ► Power series multiplication/division
   - ► Lower bound arguments
4. Linear Programming
5. Complexity Theory
6. Approximation Algorithms

References:

- ► A.V. Aho, J.D. Hopcroft, and J.E. Ullman, *The Design and Analysis of Computer Algorithms* [1].
- ► J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications* [10].

- F. Harary, *Graph Theory* [31].
- D.E. Knuth, *The Art of Computer Programming* [40, 43, 41].
- E.L. Lawler, *Combinatorial Optimization* [46].
- C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization* [60].

Implementation Considerations and Concerns:

- In addition to the ALGORITHMS AND PROGRAMS [330] prerequisite, this course also requires students to possess mathematical maturity. This requirement should be aided by the prerequisite COMBINATORIAL ANALYSIS [MATH 301 / CS 251].
- There is considerable overlap between this course the material of APPLIED GRAPH THEORY [MATH 484].

8.3.11 Computer Architecture [440]

Prerequisites: REAL AND ABSTRACT MACHINES [240]
or INTRODUCTION TO DIGITAL SYSTEMS [EE 133]

Description: This course teaches the important concepts in computer system hardware design. System architecture is the focus of this course, so the technological details of the components from which such systems are constructed are avoided except where they are crucial to design goals like capacity and performance. The topics that are taught include design models including the Register Transfer Level model, Instruction Set Processor model, and PMS model. Analytic tools taught include notions of quantity of data based on Information Theory, Queueing Theory concepts, and Performance Evaluation techniques.

Rationale: A computer scientist ought to understand the design decisions that are embodied in the computers that he uses for the same reasons that an automobile driver ought to understand his vehicle: a user who understands his tool can make better use of its capabilities. As a course that focuses exclusively on hardware, this course will teach the computer scientist things about his machines that a simple understanding of computability and complexity does not provide. As a first exposure to machine architecture, this course will prepare the machine architect for more complex concepts in computer engineering.

Objectives: At the end of this course, a student will be able to:

- ► Understand and apply architectural techniques in design and analysis of systems

Ideas: This course will be the primary carrier of the following:

- ► Machine Architecture design techniques: RTL, ISP, PMS
- ► System resources: disks, tapes, drums, memory, I/O devices
- ► data communication - coding, quantity of information
- ► performance evaluation

It will reinforce or share responsibility for:

- ► Finite State Machines
- ► Addressing, Data Representation, and Storage
- ► Analysis, synthesis, and evaluation

Topic Outline:

1. Assembly language
   - ► Instruction set (68000 as sample)
   - ► Instruction format
   - ► Addressing Schemes
   - ► *Some assembler programming project*
2. ALU Design
   - ► Addition and Subtraction
   - ► Multiplication and Division
   - ► Other ALU functions (Masks, Flags, etc.)
   - ► Floating Point Representations (Add, Subtract, Multiply, Divide, Fast ALUs, Multiplier units)
3. Central Processor Design
   - ► Register schemes (stack, one address, two address, three address)
   - ► Instruction format

- ► Pipelining
- ► Lookahead and parallelism

4. Memory
   - ► Primary Memory design
   - ► Interleaved memory
   - ► Secondary memory
   - ► Associative memory
5. Memory Management
   - ► Memory Hierarchies
   - ► Paging Systems
   - ► Segmented Systems
   - ► Replacement Algorithms
   - ► Cache Memories
6. The Control Unit
   - ► Microprogramming
   - ► Hardwired control
7. I/O
   - ► Memory Mapped vs. Programmed I/O
   - ► DMA
   - ► Channel I/O
   - ► I/O Modelling
8. Some Design examples
   - ► PDP-11
   - ► IBM 370
   - ► HP 3000
9. Data Communication and Information Theory
   - ► Quantity of Information, Entropy
   - ► Signals and Noise
   - ► Shannon's Theorem
   - ► Error Correcting Codes
10. Performance Evaluation
    - ► Queuing models
    - ► Markov chains
    - ► Simulation, measurement

References:

- ► C.G. Bell, J.C. Mudge, and J.E. McNamara, *Computer Engineering* [4].
- ► D.P. Siewiorek, C.G. Bell, and A. Newell, *Computer Structures: Principles and Examples* [72].

Implementation Considerations and Concerns:

- ► This course is also listed as INTRODUCTION TO COMPUTER ARCHITECTURE [EE 247] in the Electrical Engineering Department.

8.3.12 VLSI Systems [441]

Prerequisites: COMPUTER ARCHITECTURE [440]
ALGORITHMS AND PROGRAMS [330]

Description: This course introduces the technology of VLSI and its use in system design. A broad survey of current technologies and simple design methodologies is given. The emphasis throughout is on practical issues, and the student will learn how to design projects and implement them on a chip. Some ideas of the potentials and limitations of VLSI design will be given, and special-purpose VLSI designs for a number of application areas will illustrate these points.

Rationale: VLSI technology is assuming increasing importance as an aid to high performance, low cost system design. Computer scientists should be familiar with the advantages, possibilities and limitations of such an important technology.

Objectives: At the end of this course, a student will be able to do VLSI designs.

Ideas: This course will be the primary carrier of the following:

- ► VLSI technology, NMOS, CMOS
- ► Fabrication and design of chips
- ► Clocked and self-timed systems

It will reinforce or share responsibility for:

- ► design techniques for computer hardware
- ► hardware synchronization circuits
- ► finite state machines

Topic Outline:

1. NMOS transistors, ratios
2. Fabrication and design rules
3. CIF
4. Clocked logic and shift registers
5. Combinatorial logic between latches
6. Type D static latches
7. Programmable logic arrays
8. Design tools
9. Finite State machines
10. Delay and System Timing
11. Clocks and clock generators
12. Self-timing
13. Testability and testing
14. Systolic algorithms
15. Design in CMOS

References:

- ► C. Mead and L. Conway, *Introduction to VLSI Systems* [53].
- ► J. D. Ullman, *Computational Aspects of VLSI* [80].

Resource Requirements:

- ► Locally accessible on-line design tools

- Access to fabrication facilities

Implementation Considerations and Concerns:

- The best way to learn to do VLSI design is to do VLSI design. Therefore, the life blood of the course should be design projects. Two would be typical: one that is fairly simple such as a flip flop or shift register, and one that is more advanced.

8.3.12 VLSI Systems [441]

Prerequisites: COMPUTER ARCHITECTURE [440]
ALGORITHMS AND PROGRAMS [330]

Description: This course introduces the technology of VLSI and its use in system design. A broad survey of current technologies and simple design methodologies is given. The emphasis throughout is on practical issues, and the student will learn how to design projects and implement them on a chip. Some ideas of the potentials and limitations of VLSI design will be given, and special-purpose VLSI designs for a number of application areas will illustrate these points.

Rationale: VLSI technology is assuming increasing importance as an aid to high performance, low cost system design. Computer scientists should be familiar with the advantages, possibilities and limitations of such an important technology.

Objectives: At the end of this course, a student will be able to do VLSI designs.

Ideas: This course will be the primary carrier of the following:

- ► VLSI technology, NMOS, CMOS
- ► Fabrication and design of chips
- ► Clocked and self-timed systems

It will reinforce or share responsibility for:

- ► design techniques for computer hardware
- ► hardware synchronization circuits
- ► finite state machines

Topic Outline:

1. NMOS transistors, ratios
2. Fabrication and design rules
3. CIF
4. Clocked logic and shift registers
5. Combinatorial logic between latches
6. Type D static latches
7. Programmable logic arrays
8. Design tools
9. Finite State machines
10. Delay and System Timing
11. Clocks and clock generators
12. Self-timing
13. Testability and testing
14. Systolic algorithms
15. Design in CMOS

References:

- ► C. Mead and L. Conway, *Introduction to VLSI Systems* [53].
- ► J. D. Ullman, *Computational Aspects of VLSI* [80].

Resource Requirements:

- ► Locally accessible on-line design tools

- Access to fabrication facilities

Implementation Considerations and Concerns:

- The best way to learn to do VLSI design is to do VLSI design. Therefore, the life blood of the course should be design projects. Two would be typical: one that is fairly simple such as a flip flop or shift register, and one that is more advanced.

8.3.12 VLSI Systems [441]

Prerequisites: COMPUTER ARCHITECTURE [440]
ALGORITHMS AND PROGRAMS [330]

Description: This course introduces the technology of VLSI and its use in system design. A broad survey of current technologies and simple design methodologies is given. The emphasis throughout is on practical issues, and the student will learn how to design projects and implement them on a chip. Some ideas of the potentials and limitations of VLSI design will be given, and special-purpose VLSI designs for a number of application areas will illustrate these points.

Rationale: VLSI technology is assuming increasing importance as an aid to high performance, low cost system design. Computer scientists should be familiar with the advantages, possibilities and limitations of such an important technology.

Objectives: At the end of this course, a student will be able to do VLSI designs.

Ideas: This course will be the primary carrier of the following:

- ► VLSI technology, NMOS, CMOS
- ► Fabrication and design of chips
- ► Clocked and self-timed systems

It will reinforce or share responsibility for:

- ► design techniques for computer hardware
- ► hardware synchronization circuits
- ► finite state machines

Topic Outline:

1. NMOS transistors, ratios
2. Fabrication and design rules
3. CIF
4. Clocked logic and shift registers
5. Combinatorial logic between latches
6. Type D static latches
7. Programmable logic arrays
8. Design tools
9. Finite State machines
10. Delay and System Timing
11. Clocks and clock generators
12. Self-timing
13. Testability and testing
14. Systolic algorithms
15. Design in CMOS

References:

- ► C. Mead and L. Conway, *Introduction to VLSI Systems* [53].
- ► J. D. Ullman, *Computational Aspects of VLSI* [80].

Resource Requirements:

- ► Locally accessible on-line design tools

- Access to fabrication facilities

Implementation Considerations and Concerns:

- The best way to learn to do VLSI design is to do VLSI design. Therefore, the life blood of the course should be design projects. Two would be typical: one that is fairly simple such as a flip flop or shift register, and one that is more advanced.

8.3.12 VLSI Systems [441]

Prerequisites: COMPUTER ARCHITECTURE [440]
ALGORITHMS AND PROGRAMS [330]

Description: This course introduces the technology of VLSI and its use in system design. A broad survey of current technologies and simple design methodologies is given. The emphasis throughout is on practical issues, and the student will learn how to design projects and implement them on a chip. Some ideas of the potentials and limitations of VLSI design will be given, and special-purpose VLSI designs for a number of application areas will illustrate these points.

Rationale: VLSI technology is assuming increasing importance as an aid to high performance, low cost system design. Computer scientists should be familiar with the advantages, possibilities and limitations of such an important technology.

Objectives: At the end of this course, a student will be able to do VLSI designs.

Ideas: This course will be the primary carrier of the following:

- ► VLSI technology, NMOS, CMOS
- ► Fabrication and design of chips
- ► Clocked and self-timed systems

It will reinforce or share responsibility for:

- ► design techniques for computer hardware
- ► hardware synchronization circuits
- ► finite state machines

Topic Outline:

1. NMOS transistors, ratios
2. Fabrication and design rules
3. CIF
4. Clocked logic and shift registers
5. Combinatorial logic between latches
6. Type D static latches
7. Programmable logic arrays
8. Design tools
9. Finite State machines
10. Delay and System Timing
11. Clocks and clock generators
12. Self-timing
13. Testability and testing
14. Systolic algorithms
15. Design in CMOS

References:

- ► C. Mead and L. Conway, *Introduction to VLSI Systems* [53].
- ► J. D. Ullman, *Computational Aspects of VLSI* [80].

Resource Requirements:

- ► Locally accessible on-line design tools

- Access to fabrication facilities

Implementation Considerations and Concerns:

- The best way to learn to do VLSI design is to do VLSI design. Therefore, the life blood of the course should be design projects. Two would be typical: one that is fairly simple such as a flip flop or shift register, and one that is more advanced.

8.3.13 Theory of Programming Languages [450]

Prerequisites: LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
LOGIC FOR COMPUTER SCIENCE [351]

Description: This course brings together fundamental material on the theory of programming languages. Techniques for assigning mathematical meanings to programs and for reasoning precisely about program functionality and behavior are described. Some indication is given of the influence of formal methods on programming methodology and programming language design.

Rationale: Programs are rarely verified formally in practice, but there is much to be learned — both about programming techniques and about programming language design — from a study of precise methods for reasoning about programs. Indeed, we cannot reason precisely about programs unless we have a sound mathematical basis for such reasoning; this course is intended to provide that foundation.

Objectives: The student should gain from this course an understanding of the variety of approaches and techniques to reasoning precisely about programs. In particular, students should appreciate the potential for automation of these techniques, the ways in which they might be applied in practice, and their theoretical limitations.

Ideas: This course will be the primary carrier of the following:

- ► Formal reasoning about programs.
- ► Semantics of programming languages.
- ► Assertions about programs.

It will reinforce or share responsibility for:

- ► Programming methodology.
- ► Specifications of programs.
- ► Programming language design.

Topic Outline:

1. Introduction to semantics of programming languages
   - ► Syntax, semantics, and pragmatics: the distinctions
   - ► Abstract syntax and formal semantics
   - ► Assigning meanings to programs
   - ► Operational semantics
     - > Compilers and interpreters
     - > Labelled transition systems
     - > Operational semantics for simple sequential language with loops
   - ► Denotational semantics
     - > Basic idea: semantics given by structural induction
     - > Foundations: domains, continuous functions, and fixed points
     - > Semantics of a simple language
     - > Congruence between operational and denotational semantics
   - ► Axiomatic semantics
     - > Hoare-style axioms
     - > Weakest preconditions and predicate transformers
     - > Axiomatic semantics for simple language
     - > Consistency with respect to denotational or operational semantics
     - > Elementary ideas of soundness and relative completeness

2. The variety of programming languages
   - ▶ The distinction between imperative and applicative programs
   - ▶ The distinction between environment and store
   - ▶ Lazy evaluation and infinite structures
   - ▶ Object-oriented programming languages
   - ▶ Very-high-level programming languages
3. Semantic treatment of more complicated programming constructs:
   - ▶ Procedures:
     - > Parameterless
     - > Recursion
     - > Methods of parameter passing
   - ▶ Jumps (goto statement, breaks, etc.)
   - ▶ Nondeterminism (e.g., Dijkstra's guarded commands)
   - ▶ Parallelism:
     - > Treatment as nondeterministic interleaving of actions
     - > Concurrent processes (eg. CSP, ADA)
     - > Coroutines
   - ▶ Continuation semantics
   - ▶ Relational semantics
   - ▶ Elementary ideas of powerdomain semantics
4. Reasoning about programs
   - ▶ Inductive proof techniques
     - > Structural induction
     - > Well-founded induction
     - > Computational induction
   - ▶ Partial correctness
     - > Flowcharts and inductive assertions
     - > Hoare-style assertions
     - > Weakest preconditions
   - ▶ Total correctness of sequential programs
     - > Proving termination: examples of well-founded sets
     - > The sometime method
     - > Weakest liberal preconditions
   - ▶ Fixed-point properties of recursive programs
   - ▶ Temporal logic
     - > Continuously-operating programs
   - ▶ Dynamic logic
5. Manipulating Programs
   - ▶ Equivalence of programs
   - ▶ Program transformations and their correctness

Primary References:

- ▶ Z. Manna, *The Mathematical Theory of Computation* [52].
- ▶ J.E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory* [75].

Supplementary References:

- ▶ E.W. Dijkstra, *A Discipline of Programming* [19].
- ▶ M.J.C. Gordon, *The Denotational Description of Programming Languages.* [26].
- ▶ R.D. Tennent, *Principles of Programming Languages* [77].

Resource Requirements:

Implementation Considerations and Concerns:

- We have included a long list of important topics, from which it is possible to draw a variety of particular courses tailored to special needs or interests. At this advanced level, it is appropriate to allow some freedom in the selection of appropriate course material, especially since the area we are covering here is not yet static.
- This type of course would benefit greatly from a computer-aided facility for semantic test-bedding of formal definitions, such as symbolic execution. Although not necessary for the implementation of the course, such tools would help.

8.3.14 Complexity Theory [451]

Prerequisites: ALGORITHMS AND PROGRAMS [330]
FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
COMBINATORIAL ANALYSIS [MATH 301 / CS 251]

Description: This course extends in much more detail the material first introduced in FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]. After a quick review of the basic ideas of complexity theory, the course introduces some of the advanced results and open questions of abstract complexity theory, and the techniques used in proving these results. Emphasis is made on relating these results and open questions to their theoretical and practical implications for Computer Science; the study of computability leads to theoretical limitations on what a computer can in principle (given enough time and space) do, while the study of complexity yields limitations on what is feasibly computable: if we are restricted to using only a limited amount of time or space, the class of problems solvable by computer is restricted. There is some similarity of course content with THEORY OF ALGORITHMS [MATH 451].

Rationale: The theory of complexity is an interesting area in which many important problems remain to be solved. This course serves the purpose of engaging the student's interest and equipping him with the background material and ideas necessary for tackling research in this area.

Objectives: At the end of this course, a student will have a feeling for the theoretical limitations of computers, and how restrictions on working space and running time affect the capability of computers to solve problems. He will have seen enough of the methods and results of this subject to enable him to tackle research in this growing area.

Ideas: This course will be the primary carrier of the following:

- ► Time and Space hierarchies
- ► Notions of reducibility
- ► Complete sets for problem classes
- ► Implications of the P=NP problem

It will reinforce or share responsibility for:

- ► Time and space tradeoffs
- ► Diagonalization arguments
- ► Algorithms

Topic Outline:

1. Review of elementary Complexity Theory:
   - ► Distinction between computability and complexity
   - ► Space complexity and time complexity
   - ► Complexity relative to deterministic and nondeterministic computation
2. The time and space hierarchies: P, NP, co-NP, PSPACE, etc.
3. Time vs. space trade-offs
4. Location of known problems in the hierarchy: graph isomorphism, recognition problems, etc.
5. Notions of reducibility:
   - ► Turing reducibility
   - ► Polynomial reducibility
   - ► Logspace reducibility
   - ► Use of reductions to show complexity properties

6. Complete sets for NP: 3CNF, Clique, Hamiltonian circuits
7. Complete sets for PSPACE: QBF
8. Conditions that would imply P = NP
9. Implications for Computer Science of the P = NP question: what is the class of "feasibly computable" problems?
10. Computability and complexity relative to an oracle

References:

- M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* [24].
- J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation* [37].
- H. Rogers, *Theory of Recursive Functions and Effective Computability* [67].

Resource Requirements:

Implementation Considerations and Concerns:

- The material covered in this course overlaps with the content of THEORY OF ALGORITHMS [MATH 451].

8.3.15 Artificial Intelligence — Cognitive Processes [460]

Prerequisites: INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
or INFORMATION PROCESSING PSYCHOLOGY AND AI [PSY 213]

Description: Covers more advanced aspects of the cognitive side of AI, including natural language processing, use of knowledge sources, and learning and discovery. The use of computer programs as psychological models will also be discussed. Students will implement a large AI system as a semester project.

Rationale: This course covers the more symbolic side of AI, and allows for interaction between computer science students and psychology students. The semester project is here rather than in the prerequisite course, because the prerequisite, INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360], has a lot of ground to cover and little time in which to do it, and because most of the prerequisite material should be digested before the student spends too much time on a project.

Objectives: At the end of this course, a student will be able to:

- ► Use computer programs to model psychological phenomena
- ► Write large AI systems

Ideas: This course will be the primary carrier of the following:

- ► Cognitive Simulation
- ► Learning
- ► Natural Language Processing

It will reinforce or share responsibility for:

- ► Knowledge Representation (shared with BIG DATA [413])

Topic Outline:

1. Techniques
   - ► Exploiting Constraints
   - ► Heuristic Programming
   - ► Production Systems
2. Knowledge Representation
   - ► Declarative Knowledge
   - ► Inference and Inheritance
   - ► Procedural Knowledge
   - ► Scripts
   - ► Semantic Nets
3. Natural Language
   - ► ATN parsing
   - ► Expectation based parsing
   - ► Generation
4. Expert Systems
   - ► Design
   - ► Engineering Analysis
   - ► Medical Diagnosis
5. Cognitive Processes
   - ► Concept Acquisition
   - ► Discovery
   - ► Learning

- Planning

References:

- E. Rich, *Artificial Intelligence* [66]
- R.C. Schank and C.K. Riesbeck, *Inside Computer Understanding* [69].

Resource Requirements:

- Online versions of McEli, McSam, etc.
- Lisp programming environment
- Lisp cycles

Implementation Considerations and Concerns:

8.3.16 Artificial Intelligence — Robotics [461]

Prerequisites: INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
LINEAR ALGEBRA [MATH 341]
CALCULUS II [MATH 122] (MULTIVARIATE CALCULUS)

Description: Covers Artificial Intelligence systems which deal in some way with the physical world, either through visual, acoustic, or tactile means. Topics include vision, speech recognition, manipulation, and robotics.

Rationale: Students entering this course have a basic grounding in AI (which may not help directly in this course, but does provide a context), and by saving the more advanced material for this separate course, additional math can be required.

Objectives: At the end of this course, a student will be able to:

- ► Understand the fundamental approaches used.
- ► Read and understand literature on vision, speech, and manipulation.

Ideas: This course will be the primary carrier of the following:

- ► Perception
- ► Three Dimensional Modelling
- ► Control of Physical Systems

Topic Outline:

1. Manipulation
   - ► Kinematics and Dynamics
   - ► Trajectory Planning
   - ► Control and Control Languages
   - ► Programming
   - ► Spatial Planning
   - ► Shape Representation
2. Vision
   - ► Image formation
   - ► Shape from shading
   - ► Shape from range data
   - ► Stereo vision
   - ► Edge finding
   - ► Motion
   - ► Scene analysis
   - ► Model-based vision
3. Speech Recognition
   - ► Signal Processing
   - ► Feature Extraction
   - ► Search
4. Locomotion
   - ► Gait analysis
   - ► Dynamic balance

References:

Implementation Considerations and Concerns:

- ► This course is a part of the Artificial Intelligence group, and must be taught in such a way that it does not become a course in "Robot Engineering" or "Robot Math".

- We would have preferred to organize this course by technique and method, rather than by application.
- Also, the linear algebra course listed as a prerequisite may be too advanced for the purpose of preparing a student for Robotics. A lower level course covering the same material might be sufficient, for example METHODS OF APPLIED MATHEMATICS I [MATH 259].

8.3.17 Interactive Graphics Techniques [470]

Prerequisites: LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
ALGORITHMS AND PROGRAMS [330]

Description: A course in the creation and use of graphical information and user-interfaces.

Rationale: Although relatively young, the field of graphics has consolidated enough to warrant a semester course centering on the use of graphical, rather than textual, interaction with computers. A fair amount of background is required, however, since the students will have to apply a fair amount of previously learned material, such as language models of interaction and various sorts of algorithms. As graphical display devices become more widespread, knowledge of how to take advantage of them effectively become increasingly vital.

Objectives: At the end of this course, a student will be able to:

- ► Create interactive interfaces for computer applications.
- ► Understand the basic implementation and use of graphic support packages.
- ► Evaluate ergonomic aspects of user interfaces.

Ideas: This course will be the primary carrier of the following:

- ► The concept of graphical, vs. textual, interaction.
- ► Graphical interface creation principles.
- ► A knowledge basis to judge the merits of existing graphical tools.

Topic Outline:

1. History of computer graphics
2. Current applications
3. Graphics hardware
   - ► Vector graphics vs. raster graphics
   - ► Input devices (logical and real)
   - ► Possible future developments
4. Fundamental graphics operations
   - ► Coordinate system specification and mappings
   - ► Scan-conversion of lines and splines
   - ► Clipping
   - ► Transformation and homogeneous coordinates
5. Intermediate description formats for graphical information
6. Graphics packages
   - ► Device independence
   - ► CORE
   - ► GKS
7. Interaction techniques
   - ► Menu driven systems
   - ► FSA model, table-driven applications
   - ► Prompting, confirmation, error-checking, undo, redo, consistency
   - ► Prefix, postfix, infix operations
   - ► Window-based systems
8. User-computer dialogue
   - ► Language considerations
   - ► Human factors
9. Three (and greater) dimensional viewing

- ► Specification
- ► Implementation
- ► Solid-modelling systems

10. Graphical algorithms
    - ► Scan-conversion
    - ► Hidden line/surface removal, shading, lighting models
11. Color models and the use of color
12. Hard-copy graphics output
    - ► phototypesetters
    - ► bitmapped printers
    - ► isometric plots
    - ► half-toning

References:

- ► J.D. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics* [23].
- ► W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics* [57].

Resource Requirements:

- ► Various sorts of graphical devices for both display and input
- ► Implementations of existing graphics packages
- ► example interfaces
- ► computational support for some fairly compute-intensive operations

Implementation Considerations and Concerns:

9. Related Courses

This chapter contains descriptions of related courses in mathematics, statistics, electrical engineering, cognitive psychology, management, and public policy. They are included here because of their close ties to courses described in the previous chapter. In addition, they may be of interest for constructing concentrations in fields allied to computer science.

Some of the courses in this chapter are not currently offered, but are rather sketches of courses that sound interesting to us. These should be interpreted as proposals for discussion.

9.1 Mathematics Courses

9.1.1 Introduction to Applied Mathematics [Math 127 / CS 150]

An outline for this course is given in Section 8.1.3.

9.1.2 Calculus I [Math 121]

Description: Functions, limits, derivatives of algebraic, trigonometric, exponential and logarithmic functions, curve sketching, related rate and maximum-minimum problems, definite and indefinite integrals with applications. 3 hrs. lec., 2 hrs. rec. [Course 21-121 per CMU 1982-84 catalog]

9.1.3 Calculus II [Math 122]

Description: Techniques of integration, improper integrals, Taylor's series, functions of several variables, partial derivatives, directional derivatives, chain-rule, the gradient, multiple integrals, line integrals. 3 hrs. lec., 2 hrs. rec. Prerequisite: 21-121. [Course 21-122 per CMU 1982-84 catalog]

9.1.4 Methods of Applied Math I [Math 259]

Description: Ordinary Differential Equations: first-order, second order linear, input-output analysis, Fourier series, power series methods, Laplace transform methods. Matrix algebra, eigenvalues, systems of differential equations. 3 hrs. lec. Prerequisite: 21-122. [Course 21-259 per CMU 1982-84 catalog]

9.1.5 Elements of Analysis [Math 261]

Description: Functions of several variables, chain-rule, inverse function theorem, coordinates, external problems, multiple integrals. Vector analysis: line and surface integrals, divergence and Stokes' theorems. Convergence of series and sequences, Taylor's series, Fourier series. Prerequisite: 21-259, 3 hrs. lec. [Course 21-261 per CMU 1982-84 catalog]

9.1.6 Operations Research I [Math 292]

Description: The distribution-transportation problem: row and column number solution method and sensitivity analysis; flows in networks and incidence matrices; the standard linear program; the simplex method, post-optimality and the economic lot size problem; dynamic programming and the knapsack problem; introduction to queueing. 3 hrs. lec. Prerequisite: 21-122. [Course 21-292 per CMU 1982-84 catalog]

9.1.7 Operations Research II [Math 293]

Description: Extension of linear programming, integer programming, game theory; probabilistic programming; case studies from economics, engineering and management science. Prerequisite: 21-292. 3 hrs. lec. [Course 21-293 per CMU 1982-84 catalog]

9.1.8 Combinatorial Analysis [Math 301 / CS 251]

Description: An introduction to combinatorial mathematics with an emphasis on applications in computer science. Topics covered in depth include permutations and combinations, generating functions, recurrence relations, the principle of inclusion and exclusion, and the Fibonacci and harmonic series. Topics surveyed include existence proofs, partitions, finite calculus, generating combinatorial objects, and algorithm analysis. 3 hrs. lec. Prerequisite: 21-122. [Course 21-301 per CMU 1982-84 catalog]

9.1.9 Linear Algebra [Math 341]

Description: Vector spaces, linear transformations, orthogonality and inner product spaces, projections, dual spaces, spectral theory for normal transformation, Jordan canonical form. 3 hrs. lec. Prerequisite: 21-301. [Course 21-341 as revised fall 1983]

9.1.10 Numerical Methods [Math 369 / CS 352]

Description: Algorithmic oriented course in computer problem solving. The basic principles of numerical analysis are developed and used to solve problems involving networks and graphs, non-linear equations, differential equations, and data analysis. 3 hrs. lec. Prerequisite: 21-259. [Course 21-369 per CMU 1982-84 catalog]

9.1.11 Modern Algebra [Math 473 / CS 452]

Description: Spectral theorem, Jordan canonical form, groups, integral domains, fields, polynomials, unique factorization domains, rings and ideals, coding theory. 3 hrs lec. Prerequisite: 21-341. [Course 21-473 per CMU 1982-84 catalog]

9.1.12 Applied Graph Theory [Math 484 / CS 430]

Description: Basic terminology, cycles, trees, connectivity, planarity, coloring, matching, graph algorithms, spanning trees, binary search trees. 3 hrs. rec. Prerequisite: 21-301. [Course 21-484 as revised fall 1983] *See description of* ADVANCED ALGORITHMS [430] *in Section 8.3.10.*

9.1.13 Theory of Algorithms [Math 451 / CS 451]

Description: Basic concepts — models of computation and the design of efficient algorithms, searching and sorting, integer and polynomial arithmetic, pattern-matching algorithms, NP-completeness problems, measures of computational complexity. 3 hrs rec. Prerequisite: 21-484. [Course 21-451 as revised fall 1983] *See description of* COMPLEXITY THEORY [451] *in Section 8.3.14.*

9.1.14 Numerical Mathematics I and II [Math 704 and 705]

Description: Review of linear algebra, solution of partial differential equations by finite element and finite difference methods, direct and iterative methods, adaptive grid methods. 3 hrs rec. Prerequisite: 21-369. [Course 21-704, 705 as revised fall 1983]

9.1.15 Large-Scale Scientific Computing [Math 712 / CS 453]

Description: Review of scientific problems where computer modelling is important, design of algorithms, supercomputer architectures, algorithms for parallel computer structures. 3 hrs rec. Prerequisite: 21-705 or permission of instructor. [Course 21-712 as revised fall 1983]

9.2 Statistics Courses

9.2.1 Probability and Applied Statistics for Physical Science and Engineering I [Stat 211 / CS 250]

Description: This course provides an introduction to probability for students in engineering and science. The use of probability theory is illustrated with examples drawn from these fields. Topics include elementary probability theory, conditional probability and independence, random variables, distribution functions, joint and conditional distributions, law of large numbers, and central limit theorem. Students desiring a more mathematical treatment should register for 36-215. 3 hrs rec. Prerequisite: 21-122. [Course 36-211 per CMU 1982-84 catalog]

9.2.2 Probability and Statistics I [Stat 215]

Description: An introductory probability course, designed for students whose interest is the theory of probability. Generally all mathematics majors should enroll in this course in their junior year. Provides the necessary background for study of mathematical statistics and further topics in probability theory. A good working knowledge of calculus is required. Use of the theory is illustrated with examples drawn from engineering, science, and management. Topics include elementary probability theory, combinatorial analysis, conditional probability, independence, random variables and distribution functions, conditional distributions, generating functions and moment generating functions, sampling distributions, law of large numbers, and central limit theorem. 3 hr. rec. Prerequisite: 21-122. [Course 36-215 per CMU 1982-84 catalog]

9.2.3 Statistical Methods for Data Analysis I [Stat 219]

Description: This course presents basic concepts and operational methods of statistics for students in engineering, science and social science. Topics covered include reduction and summary of data, probability models and simulation, estimation, t-tests, goodness of fit tests, and multiple regression. The analysis of actual data sets is performed with Minitab, a statistical package requiring no previous computer experience. A section of this course will be offered for students with background and interests more oriented towards science, mathematics or engineering. No college-level prerequisites are necessary. 3 hrs. rec. [Course 36-219 per CMU 1982-84 catalog]

9.3 Electrical Engineering Courses

9.3.1 Linear Circuits: [EE 101 / CS 241]

Description: The objective of this course is to develop an understanding of the basic technical and mathematical skills required for the analysis of electrical systems. The concepts of charge, current, voltage, capacitance, inductance, energy and power are emphasized. Kirchoff's current and voltage laws, loop and node analyses, linear voltage current characteristics and superposition are introduced. The analytical and numerical solution of both difference and differential equations with constant coefficients and initial/boundary conditions, which arise in engineering problems is presented and used for the solution of first- and second-order differential equations which characterize R-C, R-L and R-L-C circuits. Consideration is given to the transient and sinusoidal steady-state analysis of linear circuits, including the use of phasor notation and complex algebra. 3 hrs. rec., 2 hrs. lab/comp. Corequisite: 15-104 or 15-111. [Course 18-101 per CMU 1982-84 catalog]

9.3.2 Electronic Circuits I [EE 102 / CS 242]

Description: The objective of this course is to provide the student with a solid understanding of the application of the principles learned in 18-101, and to increase the student's abilities to perform engineering analysis and synthesis. Semiconductor physics; operation of circuit devices; large and small signal models; biasing and temperature stability; diode and transistor circuits; feedback. 4 hrs. rec., 3 hrs. lab. Prerequisites: 18-101, 21-259, 33-123. [Course 18-102 per CMU 1982-84 catalog]

9.3.3 Introduction to Digital Systems [EE 133]

Description: Description of fundamental digital devices; basic switching circuit theory and design, including combinational and sequential logic circuits; finite state machines; register transfer level logic design, including modular components and their interconnection into data processing units; simple processor architecture. 2 hrs. rec., 3 hrs. comp./lab. Corequisites: 15-104 or 15-111. [Course 18-133 per CMU 1982-84 catalog] *Note: This course is very similar to CS course 240 as defined in this report.*

9.3.4 Linear Systems Analysis [EE 218]

Description: This course presents a unified analytic treatment of continuous time and discrete-time linear systems theory, and is intended to develop facility in the mathematical characterization of these systems and their performance in the time and frequency domains. Topics include convolution, Fourier series and transforms, sampling theorems, LaPlace transforms, Z-transforms, and applications of these methods to problems in control and communications. Prerequisite: 18-102. [Course 18-218 per CMU 1982-84 catalog]

9.3.5 Electronic Circuits II [EE 221 / CS 340]

Description: Continuation of analog circuit analysis: feedback amplifiers; frequency response; stability; operational amplifiers; op-amp characterstics; op-amp circuits; waveform generators; oscillators; tuned circuits, power amplifiers; amplifier classification; harmonic distortion. 3 hrs. rec., 3 hrs. lab. Prerequisite: 18-102. [Course 18-221 per CMU 1982-84 catalog]

9.3.6 Analysis and Design of Digital Circuits [EE 222 / CS 341]

Description: This course introduces some advanced topics in the design and analysis of digital circuits. Topics to be discussed include the analysis of RTL, DTL, TTL, and ECL gates plus MIS components such as an ALU and lookahead carry adder with emphasis on performance limitations (noise margins, propagation delay, fan-in, fan-out, etc.); analysis of noise, cross-talk and reflections in IC interconnections; non-linear circuit analysis techniques including Newton-Raphson, Euler integration and Predictor-Corrector Methods; semiconductor processing for simple bipolar and metal-oxide devices along with the models developed in the course. 2 hrs. rec. 2 hrs. comp., 3 hrs. lab. Prerequisite 18-221. [Course 18-222 per CMU 1982-84 catalog]

9.3.7 Introduction to Solid State Electronics [EE 236]

Description: This course will introduce students to semiconductor solid state devices. The course will first cover the essential physics of semiconductor device operation, including the concepts of energy bands, the Fermi distribution function, transport of current by electrons and holes, tunneling, effective mass, etc. Following this, the operation of p-n junctions, Schottky barrier diodes, bipolar transistors, junction field effect transistors (JFET), and metal-oxide-semiconductor field effect transistors (MOSFET) will be discussed along with their use in integrated circuits. The course is intended for students with no prior experience or knowledge of semiconductors. Sophomores and higher level students who have completed Physics III, Electricity and Magnetism, are all well qualified to take this course. This course will provide a solid background for students desiring to take 18-331, Semiconductor Devices and Applications. 3 hr. recitation. Prerequisites: 33-123 or permission of instructor. [Course 18-236 per CMU 1982-84 catalog]

9.3.8 Introduction to Computer Architecture [EE 247 / CS 440]

An outline for this course is given in Section 8.3.10.

9.3.9 Fundamentals of Control [EE 301]

Description: An introduction to the fundamental principles and main ideas of classical feedback control and its application. Emphasis is on problem formulation and the analysis and synthesis of servo-mechanisms using frequency domain techniques. Topics include analytical; graphical, analog techniques for treating automatic control systems; analysis of performance, stability criteria, realizability, and speed of response; compensation methods in the frequency domain, root-locus design, and pole-zero synthesis techniques; the use of analog computers in control systems; systems with delay and computer control systems; state-space description of linear systems; and non-linearities in control systems. 3 hrs. rec. 2 hrs. comp. Prerequisite: 18-213. [Course 18-301 per CMU 1982-84 catalog]

9.4 Psychology Courses

9.4.1 Psychology of Learning and Problem Solving [Psy 113]

Description: A course aimed at increasing students' learning and problem-solving skills through understanding and applying topics in cognitive psychology. Topics covered will include representing problems searching for solutions, making decisions, learning and creativity. Emphasis will be placed on the acquisition of skills which can be transferred to the student's own area of interest. [Course 85-113 per CMU 1982-84 catalog.]

9.4.2 Information Processing Psychology and Artificial Intelligence [Psy 213]

Description: Analysis of computer programs for producing intelligent behavior and their relationship to human information processing. The course focuses on perceptual information processing, memory systems, problem-solving and language processing. Students will write programs to simulate aspects of human information processing. rerequisites: Ability to program in some computer language. [Course 85-213 per CMU 1982-84 catalog.]

9.4.3 Human Factors [Psy 363]

Description: The purpose of the course is to acquaint students with a rapidly expanding area of psychology, investigating the effects of human factors on cognitive and behavioral functioning. Central to the area is the notion that physical and social environments should be planned and constructed in a way that maximizes the fit between those environments and the psychological characteristics of the people that will inhabit them. In general, the course will focus on the use of machines as aides to human functioning. Included will be a discussion of the role that computers can play in information processing and human problem solving. Prerequisites: any 100- or 200-level psychology course. [Course 85-363 per CMU 1982-84 catalog.]

9.4.4 Cognitive Processes and Problem Solving [Psy 411]

Description: Psychological processes in thinking and problem solving; relation of language to thinking; relation of perception and imagery to problem solving; semantics and internal representations; development of information processing capacity. Methods for studying thinking empirically; constructing and testing computer simulation models of adult's and children's thinking. Prerequisite: consent of the instructor. [Course 85-411 per CMU 1982-84 catalog]

9.4.5 Thinking [Psy 417]

Description: The course is intended as an extension of Psychology 411. It will review research on higher-level mental processes and the implications of this research. Possible topics include knowledge representation, pattern recognition, symbolic knowledge, schematic knowledge, memory for facts, skill acquisition, problem-solving, reasoning, language comprehension, language generation, and language acquisition. The factual content will mainly come from assigned readings and class discussions. Also, students will be required to perform a series of projects simulating various cognitive processes. Grade will be based on these simulation assignments and a final take-home. Prerequisite: Instructor's permission. [Course 85-417 per CMU 1982-84 catalog]

9.5 Engineering and Public Policy Courses

9.5.1 Law and Technology [EPP 321]

Description: The interaction of law and technology is considered in several areas: the environment, safety and health, product liability and patents and trade secrets. The public policy which emerges as law in these areas arises from forums such as public hearings or courts of law. The focus of the course is twofold: (1) understanding present law in these areas, and (2) using the data from prior public hearings in at least two of these areas to evaluate critically the nature and validity of the technological input used in reaching the public policy decision. Prerequisite: 19-319 or 70-361. [Course 19-321 per CMU 1982-84 catalog]

9.5.2 Telecommunications Policy Analysis [EPP 402]

Description: This course reviews the physical principles and capabilities of modern telecommunications systems and surveys state-of-the art technology. Economic, cultural, political, and health-related impacts of telecommunications are discussed. The concept of the electromagnetic spectrum as a scarce but nondepletable resource and questions of economic efficiency and distributional equity will be considered as bases for national and international regulation. Cost-risk benefit determination and allocation will be studied using case studies (e.g.,telephone rate design, direct broadcast satellite licensing, ELF submarine communications alternatives). Prerequisites: 73-100, junior standing in CIT. [Course 19-402/18-402 per CMU 1982-84 catalog]

9.5.3 Policy Issues in Computing [EPP 380 / CS 380]

As computers and automation become more pervasive, it becomes the responsibility of those who understand this technology to be aware of its effects on society and to be able to interpret it to both laymen and policy makers. This course is intended for students with expertise in computer science, and it will address the effects of specific computer technologies such as networks, very large databases, and robot automation. Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212] plus any 300-level computer science course.

9.6 Engineering Courses

9.6.1 Real Time Computing in the Laboratory [CIT 252]

Description: The goal of this course is to introduce students to the use of dedicated microcomputers in laboratory situations, by covering those basics in computer organization and pertinent software concepts not taught in 15-104, 15-111. It will require laboratory work, and will draw data gathering and real-time control examples and applications from various engineering disciplines. It is primarily intended for non-Electrical Engineering majors in CIT. Prerequisite: 15-104 or 15-111. [Course 39-252 per CMU 1982-84 catalog]

9.6.2 Analysis, Synthesis and Evaluation [CIT 300]

Description: Analysis, synthesis, and evaluation in the context of realistic engineering situations. The student learns through practice to deal with problems which require the use of skills that include modeling, analyses that range from mathematical to heuristic, the use of experimental methods, inventing, making judgments of value and need, and the making of decisions and recommendations. Problems are chosen to reflect interdisciplinary nature of engineering problems. 2 hrs. rec. 2 hrs. tutorial/lab. Prerequisite: junior standing in CIT. [Course 39-300 per CMU 1982-84 catalog]

9.6.3 The History and Formulation of Research and Development Policy [CIT 401]

Description: This interdisciplinary course will study the modes of research and development over the course of the 20th century. It will examine the relationship between the institutions responsible for R&D, such as industry, government, universities and foundations, and how R&D has affected the course of technological change. The course will consider the goals of R&D policy and the factors that have gone into policy formulation. The last section of the course will deal with the future directions of R&D policy. [Course 39-401, 79-509 per CMU 1982-84 catalog]

9.6.4 Cost-Benefit Analysis [CIT 404]

Description: The course will be directed primarily to Engineering students. Approximately equal time will be devoted to theory and practical applications. Topics will include the concepts of costs and benefits, market valuation and the meaning of prices (explicit and imputed), efficiency, the distribution of wealth, effects of alternative property rights structures, externality, investment criteria, uncertainty and risk. Examples of cost-benefit analysis will be presented and techniques of estimating costs and benefits will be discussed. Finally students will be given the opportunity to improve their skills in evaluating projects and examining appropriate alternatives by means of a practical exercise. 3 hrs. rec. Prerequisite: 73-100 or 24-291 or 06-303. [Course 39-404 per CMU 1982-84 catalog.]

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

2. Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

3. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

4. C. Gordon Bell, J. Craig Mudge and John E. McNamara. *Computer Engineering a DEC View of Hardware Systems Design.* Digital Press, 1978.

5. M. Ben-Ari. *Principles of Concurrent Programming.* Prentice-Hall, 1982.

6. Jon Louis Bentley. *Writing Efficient Programs.* Prentice-Hall, Inc., 1982.

7. Jon Louis Bentley. "Programming Pearls." *Communications of the ACM 26*, 8 (August 1983). Regular column.

8. Garrett Birkhoff and Thomas C. Bartee. *Modern Applied Algebra.* McGraw-Hill, 1970.

9. Barry W. Boehm. *Software Engineering Economics.* Prentice-Hall, Inc, 1981.

10. J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* American Elsevier, 1976.

11. G.S. Boolos and R.C. Jeffrey. *Computability and Logic.* Cambridge University Press, 1974.

12. Frederick P. Brooks, Jr.. *The Mythical Man-month: Essays on Software Engineering.* Addison-Wesley, 1975.

13. Michale J. Clancy and Donald E. Knuth. A Programming and Problem-Solving Seminar. Tech. Rept. Technical Report Stan-CS-77-606, Stanford University, April, 1977.

14. N.J. Cutland. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press, 1980.

15. O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. *Structured Programming.* Academic Press, 1982.

16. C.J. Date. *The System Programming Series: An Introduction to Database Systems.* Addison-Wesley, Reading, MA, 1981.

17. C.J. Date. *The System Programming Series: An Introduction to Database Systems Volume 2.* Addison-Wesley, Reading, MA, 1983.

18. E.W. Dijkstra. Co-operating Sequential Processes. In F. Genuys, Ed., *Programming Languages,* Academic Press, 1968, pp. 43-112.

19. Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Inc., 1976.

20. R.G. Dromey. *How to Solve it by Computer.* Prentice-Hall, 1982.

21. H. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

22. Lawrence Flon, Paul N. Hilfinger, Mary Shaw and Wm. A. Wulf. A Fundamental Computer Science Course that Unifies Theory and Practice. Proceedings of the SIGCSE/CSA Technical Symposium of Computer Science Education, February, 1978, pp. 255-259.

23. J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

24. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

25. Judith L. Gersting. *Mathematical Structures for Computer Science.* W.H. Freeman, 1982.

26. M.J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

27. David Gries. *Compiler Construction for Digital Computers.* Wiley, 1971.

28. David Gries. *The Science of Programming.* Springer-Verlag, 1981.

29. Ralph E. Griswold and Madge T. Griswold. *A SNOBOL4 Primer.* Prentice-Hall, Inc, 1973.

30. A.N. Habermann. *Introduction to Operating System Design.* Science Research Associates, Inc., 1976.

31. Frank Harary. *Graph Theory.* Addison-Wesley, 1969.

32. John R. Hayes. *The Complete Problem Solver.* Franklin Institute Press, 1981.

33. Peter Hibbard, Andy Hisgen, Jonathan Rosenberg, Mary Shaw, and Mark Sherman. *Studies in Ada Style.* Springer-Verlag, 1981.

34. Paul N. Hilfinger, Mary Shaw, Wm. A. Wulf and Lawrence Flon. Introducing "Theory" in the Second Programming Course. Proceedings of the Ninth SIGCSE Technical Symposium, August, 1978.

35. C.A.R. Hoare. "Communicating Sequential Processes." *Communications of the ACM 21*, 8 (August 1978), 666-677.

36. R.C. Holt, E.D. Lazowska, G.S. Graham and M.A. Scott. *Structured Concurrent Programming with Operating Systems Applications.* Addison-Wesley, 1978.

37. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory Languages and Computation.* Addison-Wesley, 1979.

38. Elaine Kant. "A Semester Course in Software Engineering." *Software Engineering Notes 6*, 4 (August 1981), 52-76.

39. B.W. Kernighan and P.J. Plauger. *Software Tools in Pascal.* Addison-Wesley, 1981.

40. Donald E. Knuth. *The Art of Computer Programming.* Volume 1: *Fundamental Algorithms.* Addison-Wesley, 1973.

41. Donald E. Knuth. *The Art of Computer Programming.* Volume 3: *Sorting and Searching.* Addison-Wesley, 1973.

42. Donald E. Knuth and Allan A. Miller. A Programming and Problem-Solving Seminar. Tech. Rept. Technical Report Stan-CS-81-863, Stanford University, June, 1981.

43. Donald E. Knuth. *The Art of Computer Programming.* Volume 2: *Seminumerical Algorithms.* Addison-Wesley, 1981.

44. Imre Lakatos. *Proofs and Refutations: The Logic of Mathematical Discovery.* Cambridge University Press, 1976.

45. Butler W. Lampson. Hints for Computer System Design. Proceedings of Symposium on Operating System Principles, Association for Computing Machinery, 1983, to appear.

46. Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, 1976.

47. Henry Ledgard and Michael Marcotty. *The Programming Languge Landscape.* Science Research Associates, 1981.

48. H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

49. C.L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill, 1968.

50. C.L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill, 1977.

51. Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* Holt, Rinehart, Winston, 1969.

52. Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

53. Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

54. Marvin Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, 1967.

55. Glenford J. Myers. *Software Reliability Principles and Practices.* Wiley Interscience, 1976.

56. Glenford J. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, 1978.

57. William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

58. J.E. Nicholls. *The Structure and Design of Programming Languages.* Addison-Wesley, 1975.

59. Sandra Pakin. *APL\360 Reference Manual, Second Edition.* Science Research Associates, Inc., 1972.

60. Christos H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

61. Frank W. Paul, Donald L. Feucht, B.R. Teare, Jr., Charles P. Neuman and David Tuma. Analysis, Synthesis and Evaluation -- Adventures in Professional Engineering Problem Solving. Proceedings of the Fifth Annual Frontiers in Education Conference, IEEE and the Amer. Soc. for Engr. Ed., October, 1975, pp. 244-251.

62. George Polya. *Mathematical Discovery.* John Wiley and Sons, 1962.

63. George Polya. *How to Solve It.* Princeton University Press, 1973.

64. Terrence W. Pratt. *Programming Languages: Design and Implementation (second edition).* Prentice-Hall, Inc., 1984.

65. Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, 1977.

66. Elaine Rich. *Artificial Intelligence.* McGraw-Hill, 1983.

67. H. Rogers. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

68. Moshe F. Rubinstein. *Patterns of Problem Solving.* Prentice-Hall, Inc., 1975.

69. Roger C. Schank and Christopher K. Riesbeck. *Inside Computer Understanding.* Lawrence Erlbaum Associates, 1981.

70. Robert Sedgewick. *Algorithms.* Addison-Wesley, 1983.

71. Martin Shooman. *Software Engineering.* McGraw-Hill, 1983.

72. Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1982.

73. D.F. Stanat and D.F. McAlister. *Discrete Mathematics in Computer Science.* Prentice-Hall, Inc., 1977.

74. H.S. Stone. *Discrete Mathematical Structures and Their Applications.* Science Research Associates, Inc., 1973.

75. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach To Programming Language Theory.* MIT Press, 1977.

76. Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Engelwood Cliffs, NJ, 1981.

77. R.D. Tennent. *Principles of Programming Languages.* Prentice-Hall, 1981.

78. J.P. Tremblay and R.P. Manohar. *Discrete Mathematical Structures With Applications to Computer Science.* McGraw-Hill, 1975.

79. Jeffrey D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

80. Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

81. Chris Van Wyk and Donald E. Knuth. A Programming and Problem-Solving Seminar. Tech. Rept. Technical Report Stan-CS-79-707, Stanford University, January, 1979.

82. D. vanDalen. *Logic and Structure.* Springer-Verlag, 1980.

83. Richard L. Wexelblat, editor. *History of Programming Languages.* Academic Press, 1981.

84. Wayne A. Wickelgren. *How to Solve Problems.* W.H. Freeman and Company, 1974.

85. Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP.* Addison-Wesley, Reading, Mass, 1981.

86. Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

87. William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler.* American Elsevier Publishing Co., 1975.

88. William A. Wulf, Mary Shaw, Paul N. Hilfinger, and Lawrence Flon. *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

89. Edward Yourdon and Larry L. Constantine. *Structured Design Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1979.

90. Marvin V. Zelkowitz, and Alan C. Shaw, and John D. Gannon. *Principles of Software Engineering and Design.* Prentice-Hall, 1979.