**Content Overview**

Being able to design simple programs is a new form of literacy. Understanding how programs work will help you understand phones, web sites, banking systems, games, movies, social networks and many other computational systems you interact with. Being able to design programs will support your work as a scientist, engineer, artist and many other professions: programming will allow you to organize, store, analyze and visualize information; create animations, music, and online communities; control devices in our environment; develop computational models and simulations; and much more.

The major goal of this course is to introduce students to a systematic method for designing programs—indeed for solving hard design problems in general. Previous students have said:

*Studying computer science inspired me to think about problems differently and take a more systematic approach to them.*

*Program design is applicable not only to computer science, but numerous areas of life. It is essential in the development of problem solving skills.*

*As I began to learn about programs and how they work, I started to appreciate how "simple" programs like Microsoft Word, especially after building my own simpler version of NotePad.*

The course is designed to be interesting, accessible and useful for all —from students wanting to become expert programmers to students with only a passing interest. No prior programming experience is assumed, and very little math and science background is required. The course will also prepare you to learn more Computer Science skills and concepts in the future. The course builds on the How to Design Programs (aka Program by Design) curriculum and we are grateful to that community for their work and support. Particular thanks go to Matthias Felleisen, Kathi Fisler and Shriram Krishnamurthi for ongoing suggestions about our version of their material.

**Overall schedules/modules covered per week:**

Week 1: Modules: BSL, HTDF. Introduction and the structure of the course. The Beginning Student Language; expressions and evaluation rules; primitive operations on numbers, strings and images. The How to Design Functions (HtDF) Recipe.

Week 2: Modules: HtDD. Representing information as data. The How to Design Data (HtDD) recipe. Atomic forms of data including intervals, enumerations, and itemizations.

Week 3: Modules: HtDW, Compound. The design of simple interactive programs. The big-bang user interface framework. The How to Design Worlds (HtDW) recipe. Representing information where two or more values naturally form a whole using compound data.

Week 4: Modules: Self-Ref, Ref. Representing arbitrary sized information using lists. Decomposition of information into multiple types.

Week 5: Modules: Naturals, Helpers. Functions operating on natural numbers. A parlor trick. Rules for decomposing functions.

Week 6: Modules: BSTs, Mutual-Ref. List abbreviations. Mutual reference allows more complex arbitrary-sized data. Using binary trees to enable fast lookup of information. Using arbitrary arity trees to represent inherently hierarchical information.

Week 7: Modules: 2-One-Of, Local. Functions that consume two arguments that have 'one-of' in their types. Using local definitions to improve the structure of programs.

Week 8: Modules: Abstraction. Using abstraction to control reduce repetition and complexity in programs.

Week 9: Modules: Genrec, Search. Generative recursion, fractals and search problems.

Week 10: Final exam.

**Learning Goals**

The main learning goals for this course are:

1. *Understand a systematic design process*: this is demonstrated by being able to write programs for a reasonably complex task, where the ability to use the "one task - one function" rule can be demonstrated.
2. *Understand that programs are written both to run on computers and for people to read*: This is demonstrated by being able to to write code that is readable, well organized, documented, and tested.
3. *Understand the relation between information and data*: this is demonstrated by being able to design the data representation for a reasonably complex problem, and to describe the information encoded in the given data.
4. *Understand that the structure of the data a program operates on determines many elements of the program's structure*: this is demonstrated by being able to identify correspondences between a data definition and a program that operates on that data. Also by being able to identify how potential changes to a data definition would affect a program.
5. *Understand that one can replace repetitive code with an abstraction in a systematic way. Understand that this is at the heart of designing libraries*: this is demonstrated by being able to produce examples of code before and after abstraction: before, where one can see the repeated code, and after, where one can see the abstraction and verify that it provides the solution to the original problem, as well as several other similar problems. Students should also be able to design a program that uses existing libraries or existing code to solve a new problem.
6. *Understand that programs can be described using notations other than code, and that these models can facilitate program design*: this is demonstrated by being able to identify correspondences between non-code models of a program and the program itself and by being able to use non-code models in program design.

**Schedule**

The course consists of a number of modules. Many are 1 week long, but some are shorter. Each week we post enough modules to last a complete week. Weeks officially start on Wednesdays, but we post the modules 2 days early on Monday.

Each module consists of a number of videos, ranging in length. Most are 8-12 minutes long. Selected videos that work through the solution to larger problems can be much longer, even up to 30 minutes, but by their nature large parts of these videos are review rather than new material.

Most videos include in-video quiz questions designed to help you assess your comprehension of the material in the video. These are not graded, but we encourage you to do them all.

Videos are supported by reference material describing the design method; all that material is available from the menu at the left edge of the course website.

For each module we provide practice problems you can use to help learn the material. Solutions are provided to practice problems, but we strongly encourage you to avoid looking at the solution until you have worked the problem.

Each module has several required homework problems followed by a short quiz based on those problems. You are encouraged to do the homework problems together with another student; but you must do the quiz on your own. However, all other resources (recipe pages, help desk, etc.) you can access. Each quiz must be done by the posted deadline for that quiz. You can find that deadline on the All Quizzes page and the Announcements page.

In addition, there will be 3 peer graded projects and a final examination. The projects will involve the design of larger programs, and the peer grading rubric will be weighted heavily towards following the design recipes and the clarity of your program.