Programming Languages 1
Course Specification

Gary T. Leavens
Department of Computer Science, Iowa State University
Ames, IA, 50011-1040 USA
leavens@cs.iastate.edu

August 23, 1999

Abstract

Computer Science 541 studies modern programming languages, with an emphasis on language design and semantics (especially for object-oriented languages). This document specifies the course's general and specific objectives.

1 Introduction

The study of programming languages is primarily concerned with the following questions:

- What are good ways to program? That is, what are expressive ways of specifying computational processes?
- How can a (good) strategy for programming be effectively expressed?
- What are the costs of various ways of programming and expressing programs? And how can those costs be lowered?

Com S 541 addresses all these questions to some extent. In addressing the third question, however, we focus on the costs of programming in terms of human time and effort, not on machine efficiency (time and space costs) or on issues of compilation.

1.1 Course Description

The catalog description of the course is as follows:

> Survey of the goals and problems of language design. Formal and informal studies of a wide array of programming language features including type systems, naming, state, and control. Creative use of functional, object-oriented, declarative, concurrent and other programming paradigms. (3 credits).

Com S 541 is distinguished from Com S 342 (Principles of Programming Languages) is that Com S 342 concentrates on essential semantic concepts, studied with the use of interpreters (coded in a functional style). Com S 342 avoids mathematical formalisms, while in Com S 541, we will not shy away from them. In this version of Com S 541, we concentrate on mathematical semantics. In Com S 541 we aim to study modern functional and object-oriented languages, and assume that the graduate students are capable of dealing with the realistic versions of such languages. In Com S 541, we use mathematical tools to draw design lessons from our study of semantics, as opposed to simply understanding the features of modern languages.

Com S 541 is distinguished from Com S 641 (Semantics of Programming Languages) in that Com S 641 discusses particular formal semantic description techniques in depth, whereas a broader and less mathematically deep use of semantic description techniques is made in Com S 541; furthermore, an attempt is made in Com S 541 to show how to use the concepts behind these techniques, not so much their details, in language design.

1.2 Acknowledgements

The course described here was originally developed with the help of Kelvin Nilsen. Khrishna Kishore Dhara has been helpful in designing several offerings of this course. The 1997 and 1998 offerings were designed with the help of the students in the course.

Final exams for similar courses at other universities were provided by Uday Redy (University of Illinois), John Mitchell (Stanford), Dan Friedman and J. Michael Ashley (Indiana), and Dave Gifford and

Franklyn Turbak (MIT); these helped provide perspective on what is important for such a course. My early ideas for this course were formed as a teaching assistant in similar graduate courses led by Dave Gifford and John Guttag of MIT. Other ideas have been shaped by Barbara Liskov, Lawrence Flon, Per Brinch-Hansen, and students in previous editions of Com S 541. Thanks to Dale Miller and Bamshad Mobasher for material on λProlog. Thanks to Dave Schmidt for help in answering questions about his books.

2 Prerequisites

The formal prerequisite in the Iowa State catalog is successful completion of either Com S 342 or Com S 440 (Principles of Compiling); that is, successful completion of an undergraduate course in programming languages or compiler construction.

The skills taught in Com S 440 relevant to Com S 541 include the ability to:

- read and write context free grammars,
- explain and implement the data structures used in a compiler, and their purpose,
- estimate the run-time costs of various language features, such as parameter passing mechanisms,
- modularize, document, and manage a large and evolving piece of software.

The topics taught in Com S 342 are perhaps more directly relevant to Com S 541 than the material in Com S 440, but at many schools some of these topics are covered in a course on compiler construction. The skills of Com S 342 relevant to Com S 541 include the ability to:

- Modify interpreters to change or enhance their behavior so as to implement various features of programming languages such as: control flow, variables, recursion, scoping, syntactic sugars, arrays, parameter passing mechanisms, objects, and inheritance.
- Write programs using such features, and explain (using appropriate terminology) and answer questions about the user-visible behavior of such programs.
- Explain (using appropriate terminology) and answer questions about the data structures and algorithms used in interpreters to implement such features.
- Compare alternatives in the design and implementation of such features.
- Solve problems using the functional paradigm.

If you do not have this background, especially if you are interested in research in programming languages, you should take Com S 342 or Com S 440 (preferably both if you want to do research in this area). Mere reading of texts on these subjects is not enough.

Please speak with me if you have any doubts about how your background matches these prerequisites.

3 General Objectives

The general objectives for Com S 541 are divided into two parts: a set of essential of objectives and a set of enrichment objectives. The enrichment objectives are found on the course web page.

In all cases, we allow the use of reference material to complete the tasks implied by the objectives.

3.1 Essential Objectives

In general terms the essential objectives for Com S 541 are that you be able to:

- Evaluate programming language features and designs based on their use in building domain-specific languages or in doing metaprogramming.
- Solve problems using the object-oriented, functional, and declarative paradigms.
- Describe the strengths and limitations of the object-oriented, functional, and logic paradigms for solving different kinds of problems (or in different application domains), especially in relation to each other.
- Explain and answer questions about specific languages that illustrate different paradigms, including questions about relevant concepts and major features.
- Design, define, and evaluate parts of programming languages or similar systems and justify your design decisions. Justifications can be by:
  - referring to known programming language concepts,
  - referring to the semantics of the features,
  - making analogies to features in specific languages that illustrate the different paradigms and their success or limitations, or

  - making some more direct argument or proof.

- Explain the tradeoffs and issues involved in the design of various language features.

- Find and critically write about research in programming languages. Evaluations of research should be based on an understanding of:

  - the goals and problems of language design,
  - the successes and limitations of previous approaches, and
  - current research directions.

Programming languages are the main technology that makes it possible for humans to instruct computers; however current programming languages are too hard to use. They take years of study to master, and there are not enough people with the necessary training. It also seems that there won't be enough people with this training anytime soon. As language designers, we should respond to this need by creating tools to create languages that help others without specialized training in programming to do their work. Hence the study of domain-specific languages and metaprogramming.

Knowing how to solve problems using the different paradigms is important for several reasons. You can find solutions to problems more surely if you have many different ways to approach problems. In the twenty-first century you will not necessarily be programming in FORTRAN or C; if you can program in a language such as Smalltalk, C++, or Ada, or other new languages you will be much in demand. As parallel programming becomes more important, the use of functional (and declarative) languages may increase. The large telephone company Ericsson uses the functional language Erlang to write all of its software. Functional programming is also a key technology for supporting domain-specific languages. See also "Why Functional Programming Matters" by John Hughes.

Even if you do not become a programmer, the ideas of the functional paradigm (function abstraction, infinite data structures, continuations, referential transparency) have important applications in all areas of computer science and in many other contexts such as mathematics and engineering. Similar comments hold true of the object-oriented paradigm. For example, the idea of data abstraction is certainly a key concept in software engineering and even in contemporary mathematics (category theory).

Understanding the strengths and weaknesses of the various paradigms is important in applying them to solve problems. Problems in the real world are not labelled with the paradigm that "should" be used to solve them, so the choice of paradigm will be important. In programming language and software engineering research, understanding the strengths and weaknesses of the existing paradigms is important for designing better ways to program.

Language design is fundamental to mathematics and science because a crucial step in solving a problem is designing an adequate notation for stating the problem (the specification) and expressing the solution. Because computers are general purpose tools, computer scientists, unlike mathematicians and traditional scientists, tend to look at widely different problems. Problems from different application domains often come without a familiar or ready-made notation; thus as computer scientists we often find it convenient to develop a special-purpose notation. These special-purpose notations, when generalized, are specification languages or programming languages. In developing and defining such a language it is helpful to draw on the results of programming language research. These results help you generate plausible designs, avoid errors, evaluate alternative designs, and precisely define the details of the design. Such justification of a design is a necessary step in debugging your design and in ultimately convincing yourself and others that your (final) design is good.

Notations that are similar to programming languages are found in every area of computer science. Besides specification languages, other similar notation systems include: user-interfaces, program libraries, formal models of computation, database query languages, operating system command languages and system call interfaces, mathematical logics, computer instruction sets, expert system shells, network protocols, and many others.

In addition, language design is challenging. Since it is one step removed from programming (you design notations that are used by programmers to write many different programs), the opportunities for good or ill are multiplied. Because of that, it is great fun!

Understanding the tradeoffs and issues involved in language design, and the semantics of various features is useful in language design itself. But it also helps one more easily understand and learn new languages and new language features. It also helps one choose a language for a programming project, and to write compilers or interpreters.

Understanding the concepts and semantics of programming languages is also necessary to make full use of them in new languages. For example, if you want to program in an object-oriented language you need to understand inheritance and message passing. A better understanding of such features, may help you to better program, reason about, and debug your programs. You will become better able to discuss advanced programming ideas with others.

Formal methods (specification and verification) are becoming increasingly important at many companies, and a deep understanding of the semantics of programming languages is also a great help in using formal methods.

Since computer science is a rapidly-changing field, it is important to be able to find and evaluate relevant papers from the research literature, even if you do not usually do research in that area. As a first step, you need to know where to find various kinds of information. It is important to distinguish peer-reviewed research literature, most of which is only found in the library, from drafts available on the web, and from trade publications.

Understanding the "state of the art" in programming language research is important for the following reasons. First, in the business world, it helps predict promising technology for programming. Knowing the goals and problems of language design helps you categorize problems that may arise; this gives you a start towards looking for existing solutions. Knowing about previous approaches to solving such problems will help you avoid mistakes and can point out fruitful approaches to solving design problems. Knowing the current research directions in language design also helps you when you are designing a non-research language, by helping you avoid features that are not well-understood. Conversely, if you are a programming languages researcher, this knowledge tells you places to spend effort.

4 Disclaimer

The details of this course are subject to change as experience dictates. You will be informed of any changes.