

Course Overview

Introduction to Syntax & Semantics

Formal Languages & Grammars

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Monday, January 14, 2019

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017–2019 Glenn G. Chappell

Course Overview

Description

In this class, we study programming languages with a view toward the following.

- How programming languages are specified, and how these specifications are used.
- What different kinds of programming languages are like.
- How certain features differ between various programming languages.

Course Overview

Goals

After taking this class, you should:

- Understand the concepts of syntax and semantics, and how syntax can be specified.
- Understand, and have experience implementing, basic lexical analysis, parsing, and interpretation.
- Understand the various kinds of programming languages and the primary ways in which they differ.
- Understand standard programming language features and the forms these take in different programming languages.
- Be familiar with the impact (local, global, etc.) that choice of programming language has on programmers and users.
- Have a basic programming proficiency in multiple significantly different programming languages.

Course Overview

Programming Languages to Study

In order to achieve these goals, you will study 6 programming languages:

1. Lua
2. Haskell
3. Forth
4. Scheme
5. Prolog
6. Whatever PL you do your project on



PL = Programming Language

Course Overview

You Need

You will need to obtain access to the following programming languages.

- Lua. Version 5.1 or later.
- Haskell. Install *The Haskell Platform*.
- Forth. Get the GNU version: *Gforth*.
- Scheme. Install *DrRacket*.
- Prolog. Get the GNU version: *gprolog*.

All of these are available for all major operating systems, and can be downloaded free.

In all cases, recent, up-to-date versions are preferred.

Topics in this class lie on two tracks:

1. Syntax (structure) & semantics (meaning) of programming languages.
 - We look at how syntax is specified, and how such a specification might make its way into a compiler.
 - We study the processes of lexical analysis, parsing, and execution.
 - You will write code to do the above.
2. Programming-language features & categories, and specific programming languages.
 - Features: execution, type systems, identifiers, values, etc.
 - Categories: dynamic languages, functional languages, concatenative languages, etc.
 - Specific PLs: Lua, Haskell, Forth, Scheme, Prolog, and one more.

Course Overview

Topics [2/2]

The following topics will be covered:

- **Formal Languages & Grammars.**
- PL Feature. Execution I: compilers, interpreters.
- PL #1. Lua.
- **Lexing & Parsing.**
- PL Feature. Type Systems.
- PL #2. Haskell.
- PL Feature. Variables & values.
- PL #3. Forth.
- **Semantics & Interpretation.**
- PL Feature. Reflection.
- PL #4. Scheme.
- PL Feature. Execution II: execution models, flow of control.
- PL #5. Prolog.

Red: Track 1

Blue: Track 2

Course Overview

What You Will Do

There will be 2 exams (Midterm & Final), occasional online quizzes (announced in advance), and 8 assignments.

Assignments will cover the following topics.

1. Formal Languages (math-y problems)
2. Coding in Lua
3. Writing a Lexer
4. Writing a Parser
5. Coding in Haskell
6. Writing an Interpreter
7. Coding in Forth, Scheme, and Prolog
8. Project

Red: Track 1

Blue: Track 2

Assignments 2–8 will involve coding. None will be in Java or C++. After finishing Assignments 3, 4, and 6, you will have a complete interpreter, written in Lua, for a PL that I invent.

Introduction to Syntax & Semantics

PL Specification

During the next few class meetings, and again after the Midterm, we will be looking at how programming languages are specified.

Consider. Alice invents a programming language and writes a description of it—a **specification**. Now Bob and Carol want to write compilers for this language.

If the specification is good enough, then Bob should be able to write a compiler without talking to Alice. Carol should be able to write a compiler without talking to Alice or Bob. The two compilers should compile the same programs. The executables produced by these compilers should do the same things.

How does Alice write such a specification? How do Bob and Carol use it?

Before we begin answering these questions, we look at some useful terminology ...

Introduction to Syntax & Semantics

Dynamic & Static

“Dynamic” refers to things that happen *at runtime*.

- In C++, `new` does *dynamic allocation*.
- Lua has *dynamic type checking*: a type error is not flagged until the code containing it is executed.
- In Windows, “DLL” stands for “*dynamic-link library*”. Code in a `.dll` file is linked with application code, as necessary, at runtime.
- ANSI Forth has *dynamic scope*: a *word* is accessible any time after its definition, until another word with the same name is defined.

“Static” refers to things that happen *before runtime*.

- In C++, global variables are *statically allocated*.
- Java has *static type checking*: type errors are flagged by the compiler. Code containing them cannot be executed.
- A C++ program is typically *statically linked* (mostly).
- Haskell has *static scope*: whether an identifier is accessible at a particular point in a program is determined by the compiler.

Hint. Take
some time to
make sure you
know these!

Introduction to Syntax & Semantics

Syntax & Semantics

Expression:
something that
has a value.



Syntax is the correct *structure* of code.

- The string "a + b" is a syntactically correct C++ expression.
- The string "a b +" is not a syntactically correct C++ expression (but it is a syntactically correct Forth expression).

Syntactically: adverb
form of the word "syntax".



Semantics is the *meaning* of code.

- In C++, the semantics of "a + b" is roughly as follows: function `operator+` is called, with `a` and `b` passed as its arguments. The return value of this function becomes the value of the expression.

There is a gray area between syntax and semantics.

- In C++, "`3+string("abc")`" will probably cause a type error. Is this a problem with syntax or semantics?
- The standard answer: we classify such issues under **static semantics**. The above "a + b" example, which concerned when happens when code executes, involved **dynamic semantics**.

Introduction to Syntax & Semantics

Where We Are Headed

Next we look at how syntax is specified.

- People write compilers based only on the written specification of a programming language. So syntax must be specified very precisely.

In a few weeks, we will discuss how syntax specifications are *used*. Over two homework assignments, you will write code to do **parsing**: determining whether code is syntactically correct, and, if so, what its structure is.

During the second half of the semester, we will look—much more briefly—at the specification of semantics.

A **string** is a finite sequence of zero or more characters. A **formal language** (or just **language**) is a set of strings.

A formal language is not the same as a programming language. This unfortunate terminology is, alas, very standard.

The characters in these strings lie in some **alphabet**. We talk about a language **over** an alphabet.

When we study formal languages as abstract objects, we often write strings without quote marks. That makes it tricky to represent the empty string, so we denote the empty string with a lower-case Greek epsilon (ϵ).

"abc"	becomes	abc
""	becomes	ϵ

Formal Languages & Grammars

Formal Languages [2/4]

Here are some examples of (formal) languages.

- $\{abc, xyz, q\}$
- $\{\epsilon, 01, 0101, 010101, 01010101, \dots\}$
 - The above set is a language over the alphabet $\{0, 1\}$.
- The set of all legal C++ identifiers.
 - These are all strings that contain only letters, digits, and underscores ("_"), begin with a letter or underscore, and are not one of the C++ reserved words (`for`, `class`, `if`, `const`, `private`, `virtual`, `delete`, `friend`, `throw`, `this`, etc.).
- The set of all syntactically correct Lua programs.
 - We do not normally think of a whole program as a string. But it is.

The last two examples above illustrate why we are talking about formal languages in a class on programming languages.

How do we describe a language?

We can use a *generator* or a *recognizer*.

A **generator** is something that can produce the strings in a language—all of them, and nothing else.

A **recognizer** is a way of determining whether a given string lies in the language.

An important question, when we are dealing with a formal language: Given a string, does it lie in the language?

To answer this question, we need a recognizer.

- Every compiler has to contain a recognizer.

But it is usually easier to construct a generator.

A common technique: Write a generator, and then have a program use it to produce a recognizer automatically.

- Example: the program *Yacc*, which inputs a kind of generator called a **grammar**, and outputs C code for a recognizer.

Over the next few days, we will have a lot more to say about generators and recognizers.

Formal Languages & Grammars

Grammars — Definitions [1/2]

A **phrase-structure grammar** (often just **grammar**) is one kind of language generator.

To write a grammar, we need a collection of **terminal symbols**.
This is our alphabet.

We also need a collection of **nonterminal symbols**. These are like variables that eventually turn into something else. One nonterminal symbol is the **start symbol**.

Our conventions, for now, will be that lower-case letters are terminal symbols, upper-case letters are nonterminal symbols, and “S” is the start symbol.

Some terminal symbols: a b x

Some nonterminal symbols: A Q S

Start symbol



Formal Languages & Grammars

Grammars — Definitions [2/2]

A **grammar** is a list of one or more *productions*. A **production** is a rule for altering strings by substituting one substring for another. The strings are made of terminal and nonterminal symbols.


Here is a grammar with four productions.

$$S \rightarrow AB$$

$$A \rightarrow c$$

$$B \rightarrow Bd$$

$$B \rightarrow \varepsilon$$



Epsilon (ε) is neither terminal nor nonterminal. It is not a symbol at all. Rather, it represents a string containing no symbols.

Formal Languages & Grammars

TO BE CONTINUED ...

Formal Languages & Grammars will be continued next time.