Course Overview
Basic Concepts
Introduction to Formal Languages & Grammars

CS 331 Programming Languages

Lecture Slides

Monday, January 13, 2025

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Course Overview

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.
- How to write code in various programming languages.

Course Overview
Goals

Upon successful completion of CS 331, students are expected to:

- 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.

You Need

These goals will be achieved, in part, by studying five programming languages. You will need to obtain access to them.

1. Lua. *ZeroBrane Studio* is recommended.
2. Haskell. Install any recent distribution that includes *GHC*.
3. Scheme. Install *DrRacket*.
4. Prolog. Install *SWI-Prolog*.
5. ??? (the programming language you do your presentation on).

All of the first four can be downloaded free for all major operating systems.

In all cases, get the most up-to-date version that you can.

Course Overview
Topics [1/2]

Topics in this class lie on two tracks:

PL = Programming Language

1. Syntax (correct structure) & semantics (meaning) of PLs.
   - 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. PL features & categories, and specific PLs.
   - Features: execution, type systems, identifiers, values, etc.
   - Categories: dynamic languages, functional languages, concatenative languages, etc.
   - Specific PLs: Lua, Haskell, Scheme, and Prolog.

Course Overview
Topics [2/2]

The course material will be divided into eight units:

1. Formal Languages & Grammars
2. The Lua Programming Language
   - PL Feature: Compilation & Interpretation
3. Lexing & Parsing
4. The Haskell Programming Language
   - PL Feature: Type System
5. The Scheme Programming Language
   - PL Feature: Identifiers & Values
   - PL Feature: Reflection
6. Semantics & Interpretation
7. The Prolog Programming Language
   - PL Feature: Execution Model
8. Student Presentations on Programming Languages

Track 1: Syntax & Semantics of PLs.

Track 2: PL features & categories, specific PLs.

Course Overview
What You Will Do [1/2]

Work that you will submit:

- 13 online quizzes (due Sundays at 5 pm)
- 7 homework assignments
- 2 exams (Midterm & Final)

In addition, you will do an in-class presentation on a programming language near the end of the semester.

Readings will be assigned frequently. You are expected to do each reading before taking the next quiz.

Course Overview
What You Will Do [2/2]

The 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 Scheme and Prolog

Track 1: Syntax & Semantics of PLs.

Track 2: PL features & categories, specific PLs.

Assignments 2–7 will involve coding—*not* Java, C++, or Python.

After finishing Assignments 3, 4, and 6, you will have a complete interpreter, written in Lua, for a PL that I invent.

Unit Overview
Formal Languages & Grammars

Our first unit: Formal Languages & Grammars.

Topics

- Basic concepts
- Introduction to formal languages & grammars
- The Chomsky hierarchy
- Regular languages
- Regular languages & regular expressions
- Context-free languages
- Programming language syntax specification

After this we will cover The Lua Programming Language.

Basic Concepts

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

I put a term in **boldface** when I say what it means.

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

With a properly written specification, Bob will able to write a compiler without talking to Alice. Carol will be able to write a compiler without talking to Alice or Bob. The two compilers will compile the same programs. The executables produced by these compilers will do the same things.

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

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

Dynamic refers to things that happen *at runtime*.

- In C++, `new` does *dynamic allocation*.
- Python 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.

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

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.

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 what happens when code executes, involved **dynamic 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.

Later in the semester, we will look—much more briefly—at the specification of semantics.

Introduction to Formal Languages & Grammars

Introduction to Formal Languages & Grammars
Formal Languages [1/4]

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.

Or "collection" if you prefer.

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. We denote the empty string with a lower-case Greek epsilon (ε).

"*abc*"	becomes	*abc*
""	becomes	ε

Epsilon (ε) is *not* part of the alphabet. It is just a way to write "".

Formal Languages [2/4]

Here are some examples of (formal) languages.

- {*abc*, *xyz*, *q*}
- {ε, 01, 0101, 010101, 01010101, …}
  - The above set is a language over the alphabet {0, 1}.
- The set of all legal C++ identifiers.
  - That is, 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`, `static_cast`, etc.).
- The collection 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.

Formal Languages [3/4]

How do we describe a formal language in a precise way?

There are two broad categories of ways to describe formal languages: *generators* and *recognizers*.

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 if a given string lies in the language. Given a string in the language, a recognizer says, "yes"; given a string that is not in the language, it does not.

Formal Languages [4/4]

An important question, when we are dealing with a formal language: Given a string, does it lie in the language? (Every compiler must be able to answer this question—right?)

To answer this question, we need 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.

Programs like *Yacc*, *Bison*, and *ANTLR* input a kind of generator called a *grammar*, and output code (in C, perhaps) for a recognizer.

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

Grammars — Definitions [1/2]

A **phrase-structure grammar** (usually 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**.

For now, 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*

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.

$S \rightarrow AB$

$A \rightarrow c$

$B \rightarrow Bd$

$B \rightarrow \varepsilon$

You might read the right arrow as "becomes", "goes to", "turns into", or "is replaced by".

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

Grammars — Derivations [1/4]

1. $S \rightarrow AB$
2. $A \rightarrow c$
3. $B \rightarrow Bd$
4. $B \rightarrow \varepsilon$

The same grammar. Productions are numbered, to make it easy to refer to them.

Here is what we do with a grammar.

- Begin with the start symbol.
- Repeatedly apply productions. To apply a production, replace the left-hand side of the production (which must be a contiguous collection of symbols in the current string) with the right-hand side.
- We can stop only when there are no more nonterminals.

The resulting list of strings is a **derivation** of the final string.

- To the right is a derivation of *cdd* based on the above grammar.

S
AB
ABd
ABdd
Add
cdd

Grammars — Derivations [2/4]

Below are the same grammar and derivation. I have annotated the derivation to show what is happening.

- The number indicates which production is being used.
- The underlined symbols show the substring being replaced. This is the left-hand side of the production being used.

Grammar

1. $S \rightarrow AB$
2. $A \rightarrow c$
3. $B \rightarrow Bd$
4. $B \rightarrow \varepsilon$

Note the use of production 4. No "ε" appears in the derivation.

Derivation of *cdd*

$\underline{S}$

$\overset{1}{\Rightarrow}\ A\underline{B}$

$\overset{3}{\Rightarrow}\ A\underline{B}d$

$\overset{3}{\Rightarrow}\ A\underline{B}dd$

$\overset{4}{\Rightarrow}\ \underline{A}dd$

$\overset{2}{\Rightarrow}\ cdd$

Grammars — Derivations [3/4]

Grammar

1. $S \rightarrow AB$
2. $A \rightarrow c$
3. $B \rightarrow Bd$
4. $B \rightarrow \varepsilon$

Derivation of *cdd*

$\underline{S}$
$\xrightarrow{1}$ $A\underline{B}$
$\xrightarrow{3}$ $A\underline{B}d$
$\xrightarrow{3}$ $A\underline{B}dd$
$\xrightarrow{4}$ $\underline{A}dd$
$\xrightarrow{2}$ cdd

Recall: a grammar is a kind of generator.

The language **generated** by a grammar consists of all strings for which there is a derivation.

So "*cdd*" lies in the language generated by the above grammar.

Q. What language does the above grammar generate?

A. All strings consisting of a single *c* followed by zero or more *d*'s.

{*c*, *cd*, *cdd*, *cddd*, *cdddd*, *cddddd*, …}

Grammars — Derivations [4/4]

Here is another example, involving a different grammar.

Grammar

1. $S \to xSy$
2. $S \to \varepsilon$

Q. What language does this grammar generate?

A. All strings consisting of zero or more *x*'s followed by *the same number* of *y*'s.

$$\{\varepsilon, xy, xxyy, xxxyyy, xxxxyyyy, \ldots\}$$

Derivation of *xxxyyy*

$\underline{S}$

$\overset{1}{\to}$ $x\underline{S}y$

$\overset{1}{\to}$ $xx\underline{S}yy$

$\overset{1}{\to}$ $xxx\underline{S}yyy$

$\overset{2}{\to}$ $xxxyyy$

Avoid saying "any number of …". Say "zero or more" or "one or more".

Here is another way to describe this language: $\{ x^k y^k \mid k \geq 0 \}$.

Grammars — Applications [1/2]

As the name suggests, phrase-structure grammars were first used in linguistics. They were proposed as a tool for specifying the grammar of a natural language (examples of natural languages: English, French, Arabic).

The start symbol could represent a *sentence*.

Various other nonterminals might represent things like *subject*, *predicate*, or *prepositional phrase*.

The terminal symbols would be the words of the natural language.

Grammars — Applications [2/2]

In computing, an important application of phrase-structure grammars is specifying PL syntax.

- The language generated is the set of syntactically correct programs.
- The start symbol represents a *program*.
- Other nonterminals might represent things like *statement*, *for-loop*, or *class definition*.
- Terminal symbols are typically the *lexemes*—words, roughly—of the programming language.

Since the late 1970s, virtually every PL has had its syntax specified using a grammar.

We discuss lexemes in more detail later in the semester. For now, here are some examples of lexemes in C++.

- **Keywords**: `for` `class` `const` `return`
- **Identifiers**: `mergeSort17` `ARRAY_SIZE` `x`
- **Literals**: `"Hello"` `-42` `3.47e-12f`
- **Operators**: `+=` `<<` `!` `::`
- **Punctuation**: `{` `}` `;`

TRY IT #1 (Exercises)

Grammar A

1. $S \rightarrow Sa$
2. $S \rightarrow xS$
3. $S \rightarrow x$

Exercises

1. Based on Grammar A, write a derivation for *xxxa*.
2. Is there a derivation based on Grammar A for the string *aaa*?
3. What language does Grammar A generate?

TRY IT #1 (Answers)

Grammar A

1. $S \rightarrow Sa$
2. $S \rightarrow xS$
3. $S \rightarrow x$

Derivation of *xxxa*

$\underline{S}$

$\xrightarrow{1}$ $\underline{S}a$

$\xrightarrow{2}$ $x\underline{S}a$

$\xrightarrow{2}$ $xx\underline{S}a$

$\xrightarrow{3}$ $xxxa$

Answers

1. Based on Grammar A, write a derivation for *xxxa*.

See above, on the right.

2. Is there a derivation based on Grammar A for the string *aaa*?

No, every string in the language generated begins with *x*.

3. What language does Grammar A generate?

The language generated is the set of all strings consisting of one or more *x*'s followed by zero or more *a*'s.

TRY IT #1 (Note)

Grammar A

1. $S \rightarrow Sa$
2. $S \rightarrow xS$
3. $S \rightarrow x$

Derivation #1

$\underline{S}$

$\overset{1}{\Rightarrow}\ \underline{S}a$

$\overset{2}{\Rightarrow}\ x\underline{S}a$

$\overset{2}{\Rightarrow}\ xx\underline{S}a$

$\overset{3}{\Rightarrow}\ xxxa$

Derivation #2

$\underline{S}$

$\overset{2}{\Rightarrow}\ x\underline{S}$

$\overset{1}{\Rightarrow}\ x\underline{S}a$

$\overset{2}{\Rightarrow}\ xx\underline{S}a$

$\overset{3}{\Rightarrow}\ xxxa$

Derivation #3

$\underline{S}$

$\overset{2}{\Rightarrow}\ x\underline{S}$

$\overset{2}{\Rightarrow}\ xx\underline{S}$

$\overset{1}{\Rightarrow}\ xx\underline{S}a$

$\overset{3}{\Rightarrow}\ xxxa$

There is more than one derivation of the string *xxxa* based on Grammar A. This is typical. It is not a problem.

Grammar B

$S \rightarrow XY$

$X \rightarrow a$

$X \rightarrow b$

$Y \rightarrow t$

$Y \rightarrow u$

Grammar C

$S \rightarrow A$

$A \rightarrow xA$

$A \rightarrow AA$

Exercises

4. What language does Grammar B generate?
5. What language does Grammar C generate?
   Hint. This is almost-but-not-quite a trick question.

TRY IT #2 (Answers)

Grammar B

$S \rightarrow XY$

$X \rightarrow a$

$X \rightarrow b$

$Y \rightarrow t$

$Y \rightarrow u$

Grammar C

$S \rightarrow A$

$A \rightarrow xA$

$A \rightarrow AA$

Answers

4. What language does Grammar B generate?

The language generated is {*at*, *au*, *bt*, *bu*}.

5. What language does Grammar C generate?
   Hint. This is almost-but-not-quite a trick question.

The language generated contains no strings: {}.

The language containing no strings is not the same as the language containing only the empty string!