Formal Languages & Grammars continued

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Wednesday, January 16, 2019

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

Review
Course Overview

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.

You will need to obtain access to the following programming languages (all are freely available on the web).

- 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*.

Review
Introduction to Syntax & Semantics

“**Dynamic**” means *at runtime*.

“**Static**” means *before runtime*.

Syntax is the correct *structure* of code.

Semantics is the *meaning* of code.

In programming languages like C++ and Java, which have static type checking, type errors lie in a gray area between syntax and semantics. We classify these under **static semantics**. What code does when executed involves **dynamic semantics**.

Coming Up

- How the syntax of a programming language is specified.
- How such specifications are used.
- Write a **lexer** and a **parser**; the latter checks syntactic correctness.
- Later, a brief study of semantics.

Review
Formal Languages & Grammars — Formal Languages [1/2]

A (**formal**) **language** is a *set of strings*.

Not the same as a programming language!

Alphabet: the set of characters that may appear in the strings.

For now, we write strings without quotes (for example, *abc*). We represent the empty string with a lower-case Greek epsilon (ε).

Example of a language over {0, 1}:

{ε, 01, 0101, 010101, 01010101, …}

Important examples of formal languages:

- The set of all lexemes in some category, for some programming language (e.g., the set of all legal C++ identifiers).
- The set of all syntactically correct programs, in some programming language (e.g., the set of all syntactically correct Lua programs).

Review
Formal Languages & Grammars — Formal Languages [2/2]

Two ways to describe a formal language.

- With a **generator**: something that can produce the strings in a formal language—all of them, and nothing else.
- With a **recognizer**: a way of determining whether a given string lies in the formal language.

Generally:

- Generators are easier to construct.
- Recognizers are more useful.

It is common to begin with a generator and then construct a recognizer based on it.

This construction process might be automated (but in this class it will not be).

Review
Formal Languages & Grammars — Grammars [1/2]

A (**phrase-structure**) **grammar** is a kind of language generator.

Needed

- A collection of **terminal symbols**. This is our alphabet.
- A collection of **nonterminal symbols**. These are like variables that eventually turn into something else. One nonterminal symbol is the **start symbol**.

Conventions, for now:

- Lower-case letters are terminal symbols (a b x)
- Upper-case letters are nonterminal symbols (C Q S)
- S is the start symbol.

Review
Formal Languages & Grammars — Grammars [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.

$S \rightarrow AB$

$A \rightarrow c$

$B \rightarrow Bd$

$B \rightarrow \varepsilon$

Formal Languages & Grammars continued
Grammars — Derivations [1/4]

$$\left.\begin{array}{ll} 1. & S \rightarrow AB \\ 2. & A \rightarrow c \\ 3. & B \rightarrow Bd \\ 4. & B \rightarrow \varepsilon \end{array}\right\}$$

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 result 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

Formal Languages & Grammars
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, which 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*

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

Formal Languages & Grammars
Grammars — Derivations [3/4]

Grammar

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

Derivation of *cdd*

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

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 this grammar generate?

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

$\{c, cd, cdd, cddd, cdddd, cddddd, \dots\}$

Formal Languages & Grammars
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*

$\underline{S}$

1

$x\underline{S}y$

1

$xx\underline{S}yy$

1

$xxx\underline{S}yyy$

2

$xxxyyy$

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

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

Formal Languages & Grammars
Grammars — Applications [1/2]

As the name suggests, phrase-structure grammars were first used in linguistics, as a way of formalizing 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.

Formal Languages & Grammars
Grammars — Applications [2/2]

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

- 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 programming language 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**: `{` `}` `;`

Formal Languages & Grammars
TRY IT [1/5]

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?

Formal Languages & Grammars
TRY IT [2/5]

Grammar A

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

Derivation of *xxxa*

$$
\begin{aligned}
&\underline{S}\\
\overset{1}{\Rightarrow}\ &\underline{S}a\\
\overset{2}{\Rightarrow}\ &x\underline{S}a\\
\overset{2}{\Rightarrow}\ &xx\underline{S}a\\
\overset{3}{\Rightarrow}\ &xxxa
\end{aligned}
$$

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.

Formal Languages & Grammars
TRY IT [3/5]

Grammar A

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

Derivation #1 of *xxxa*

$\underline{S}$
1
$\underline{S}a$
2
$x\underline{S}a$
2
$xx\underline{S}a$
3
$xxxa$

Derivation #2 of *xxxa*

$\underline{S}$
2
$x\underline{S}$
1
$x\underline{S}a$
2
$xx\underline{S}a$
3
$xxxa$

Note. There is more than one derivation of the string *xxxa* based on Grammar A. This is not at all unusual.

Formal Languages & Grammars
TRY IT [4/5]

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 kind of a trick question.

Formal Languages & Grammars
TRY IT [5/5]

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 generate is $\{at, au, bt, bu\}$.

5. What language does Grammar C generate?
   Hint. This is kind of 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!

Formal Languages & Grammars
The Chomsky Hierarchy — Introduction

In the late 1950s, linguist Noam Chomsky described a hierarchy of categories of formal languages, defined in terms of the kinds of grammars that could generate them. Chomsky aimed to develop a framework for studying natural languages; however, his hierarchy has proved to be useful in the theory of computation.

The Chomsky Hierarchy includes four categories of languages. He called them types 3, 2, 1, and 0. More modern names are **regular**, **context-free**, **context-sensitive**, and **computably enumerable**.

For each language category, there is an associated category of grammars that can generate that kind of language. The same names are used for the grammar categories (for example, a **regular grammar** generates a **regular language**).

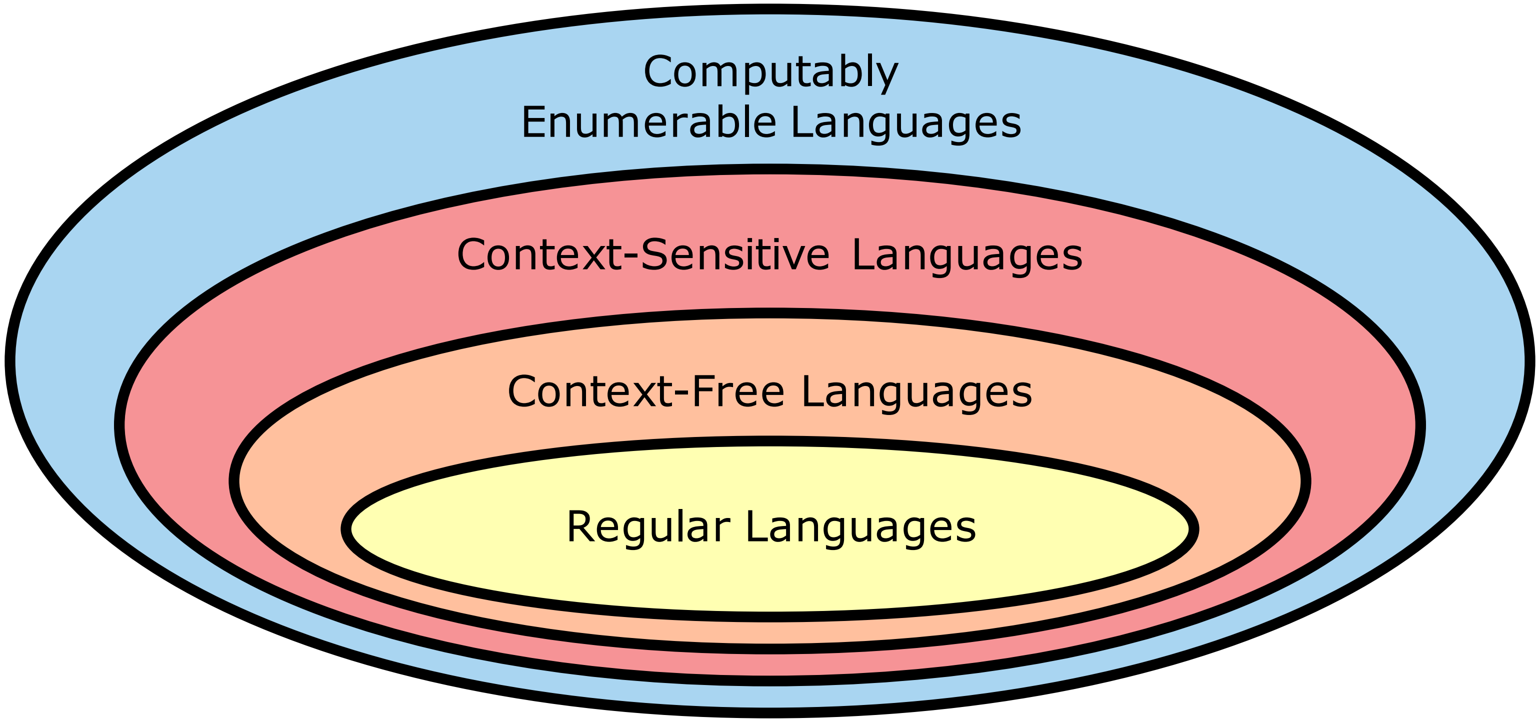
Formal Languages & Grammars
The Chomsky Hierarchy — The Hierarchy [1/2]

Here is the Chomsky Hierarchy.

Language Category		**Generator**	**Recognizer**
Number	**Name**		
Type 3	Regular	Grammar in which each production has one of the following forms. • $A \to \varepsilon$ • $A \to b$ • $A \to bC$ Another kind of generator: **regular expressions** (covered later).	Deterministic Finite Automaton Think: Program that uses a small, fixed amount of memory.
Type 2	Context-Free	Grammar in which the left-hand side of each production consists of a single nonterminal. • $A \to$ [anything]	Nondeterministic Push-Down Automaton Think: Finite Automaton + Stack (roughly).
Type 1	Context-Sensitive	*Don't worry about it.*	*Don't worry about it.*
Type 0	Computably Enumerable	Grammar (no restrictions).	Turing Machine Think: Computer Program

Formal Languages & Grammars
The Chomsky Hierarchy — The Hierarchy [2/2]

Each category of languages in the Chomsky Hierarchy is contained in the next. So every regular language is context-free, etc.



Next we look briefly at each category in the Chomsky Hierarchy, how it is defined, and why we care about it.

Formal Languages & Grammars
The Chomsky Hierarchy — Why We Care [1/5]

A **regular language** is one that can be generated by a grammar in which each production has one of the following forms.

- $A \to \varepsilon$
- $A \to b$
- $A \to bC$

Alternative generator: **regular expression** (covered later).

A regular language can be recognized by a **deterministic finite automaton**.

- Think: a program using only a small, fixed amount of memory.

Regular languages generally describe lexeme categories.

- The set of all legal C++ identifiers is a regular language.

Thus, these languages encompass the level of computation required for **lexical analysis**: breaking a program into lexemes.

Regular languages are also involved in text **search/replace**.

Formal Languages & Grammars
The Chomsky Hierarchy — Why We Care [2/5]

A **context-free language** is one that can be generated by a grammar in which the left-hand each production consists of a single nonterminal.

- $A \rightarrow$ [anything]

A context-free language can be recognized by a **nondeterministic push-down automaton**.

- Roughly: a finite automaton plus a memory that acts as a stack.

Context-free languages generally describe programming-language syntactic correctness.

- The set of all syntactically correct Lua programs (for example) is a context-free language.

Thus, these languages encompass the level of computation required for **parsing**: determining whether a program is syntactically correct, and, if so, how it is structured.

Formal Languages & Grammars
The Chomsky Hierarchy — Why We Care [3/5]

As for **context-sensitive languages**: we generally do *not* care.

- I would call this category a mistake—an idea that Chomsky thought would be fruitful, but turned out not to be.
- I mention this category only for historical interest. You do not need to know anything about context-sensitive languages.

In case anyone is interested: The kind of grammar that describes a context-sensitive language allows restricting the expansion of a nonterminal to a particular context. For example, such a grammar might include the following production.

$$xAy \rightarrow xBcy$$

So *A* can be expanded to *Bc*, as long as it lies between *x* and *y*.

And the recognizer for a context-sensitive language is called a *linear bounded automaton*. Google it, if you wish. Or don't.

Formal Languages & Grammars
The Chomsky Hierarchy — Why We Care [4/5]

A **computably enumerable language** is one that can be described by a grammar. We place no restrictions on the productions in the grammar.

The recognizer for a computably enumerable language is a **Turing machine**—a formalization of a computer program.

We care about computably enumerable languages because they encompass the things that computer programs can do.

- If a language is computably enumerable, then some computer program is a recognizer for it.
- Otherwise, *no such program exists*.

Computably enumerable languages are important, but we will not discuss them any further in this class.

Note. This kind of language is also called a **recursively enumerable language**. This terminology comes from a branch of mathematics called *recursive function theory*.

Formal Languages & Grammars
The Chomsky Hierarchy — Why We Care [5/5]

Summary

- A lexeme category (e.g., C++ identifiers) generally forms a **regular language**. Recognition of a regular language is thus the level of computation required for lexical analysis—and text search/replace.
- In most programming languages, the set of all syntactically correct programs forms a **context-free language**. Recognition of context-free languages is thus the level of computation required for parsing.
- **Context-sensitive languages** are mostly a historical curiosity.
- Recognition of **computably enumerable** languages encompasses the tasks that computer programs are capable of. These languages are important in the theory of computation.

Our next topic is *Regular Languages*. We will cover ideas to be used in lexical analysis.

After that, we study *Context-Free Languages*, covering ideas to be used in parsing.