Regular Languages & Regular Expressions

CS 331 Programming Languages
Lecture Slides
Friday, January 17, 2025

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

Unit Overview
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

Review

Review
Formal Languages & Grammars — Formal Languages

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

Not the same as a programming language!

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.

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

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

A (**phrase-structure**) **grammar** is a list of one or more *productions*. A **production** is a rule for altering strings by substituting one substring for another.

Grammar

$S \rightarrow yS$

$S \rightarrow x$

$S \rightarrow \varepsilon$

Terminal symbols—allowed in the final string in a derivation. For now, these are lower-case letters.

Nonterminal symbols—not allowed in the final string. For now, these are upper-case letters. One nonterminal is the **start symbol**. For now: *S*.

An important application of grammars is specifying programming-language syntax.

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

Grammar

1. $S \rightarrow yS$
2. $S \rightarrow x$
3. $S \rightarrow \varepsilon$

The numbers and underlining are annotations that I find helpful. They are not actually part of the derivation.

A derivation is a list of strings.

Derivation of *yyy*

$\underline{S}$
1
$y\underline{S}$
1
$yy\underline{S}$
1
$yyy\underline{S}$
3
yyy

No "ε" appears here.

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

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more *y*'s followed by an optional *x*.

$\{\varepsilon, y, yy, yyy, \ldots, x, yx, yyx, yyyx, \ldots\}$

Review
Regular Languages — Regular Grammars & Languages

A **regular grammar** is a grammar, each of whose productions looks like one of the following.

$A \to \varepsilon$ $\qquad$ $A \to b$ $\qquad$ $A \to bC$

We allow a production using the same nonterminal twice: $A \to bA$

A **regular language** is a language that is generated by some regular grammar.

This is a regular grammar:

$$
\begin{aligned}
S &\to \varepsilon \\
S &\to t \\
S &\to xB \\
B &\to yS
\end{aligned}
$$

Therefore, the language it generates is a regular language:

$$\{\varepsilon, xy, xyxy, xyxyxy, \ldots, t, xyt, xyxyt, xyxyxyt, \ldots\}$$

Review
Regular Languages — Finite Automata [1/3]

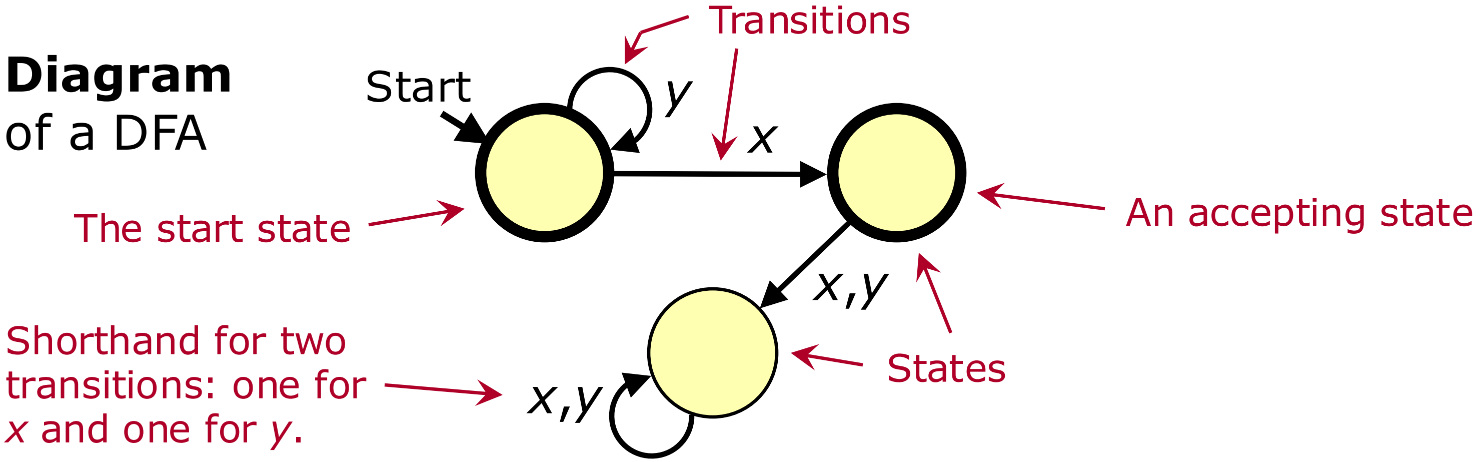
A **deterministic finite automaton** (Latin plural "**automata**"), or **DFA**, is a kind of recognizer for regular languages.

A DFA has:

- A finite collection of **states**. One is the **start state**. Some may be **accepting states**.
- **Transitions**, each beginning at a state, ending at a state, and associated with a character in the alphabet.

Rule. For each character in the alphabet, each state has *exactly one* transition leaving it that is associated with that character.

Diagram of a DFA

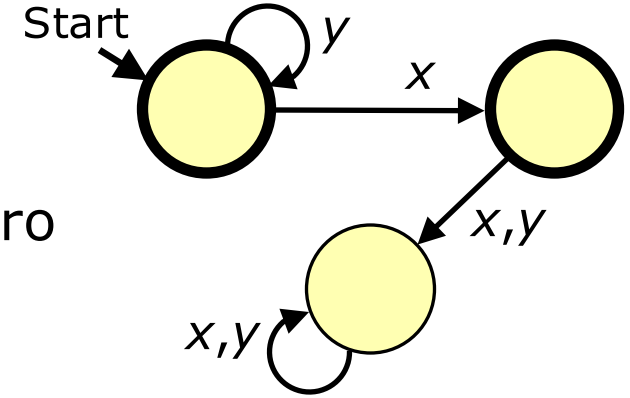


Review
Regular Languages — Finite Automata [2/3]

To use a DFA as a recognizer:

- Start in the start state; proceed in a series of steps.
- At each step, read a character from the input and follow the transition beginning at the current state and labeled with the character that was read.
- If, when we reach the end of the input, we are in an accepting state, then we **accept** the input.

The set of all inputs that are accepted is the language **recognized** by the DFA.



Q. What language is recognized by the DFA diagrammed here?

A. The set of all strings that consist of zero or more *y*'s followed by an optional *x*.

{*ε*, *y*, *yy*, *yyy*, …, *x*, *yx*, *yyx*, *yyyx*, …}

Review
Regular Languages — Finite Automata [3/3]

Fact. The languages that are recognized by DFAs are precisely the regular languages.

That is:

- For each DFA, the language it recognizes is a regular language.
- For each regular language, there is a DFA that recognizes it.

A DFA is a kind of **state machine**: it has a state, and it transitions to a new state based, in part, on its current state.

We will see the state-machine idea in code form later in the semester when we write code to do lexical analysis.

Regular Languages & Regular Expressions

Regular Languages & Regular Expressions
Regular Expressions — Introduction

We wish to define a kind of generator called a *regular expression*—or sometimes *regex*, for short. We will cover both their syntax and their semantics.

Before we do this, let us consider a kind of expression that all of us are familiar with: the *arithmetic expression*.

As a warm-up, we will describe the syntax and semantics of arithmetic expressions, using informal methods. Afterward, we will describe regular expressions in a similar way.

Regular Languages & Regular Expressions
Regular Expressions — Warm-Up: Arithmetic Expressions [1/4]

We are *not* describing regular expressions here!

An **arithmetic expression** is an expression involving numbers, identifiers, and arithmetic operators (+ – * /) as usual.

Here is an example of an arithmetic expression.

34*(3–*n*)+(5.6/*g*+3)

We describe the syntax and semantics of arithmetic expressions.

- **Syntax** refers to correct structure. Knowing the syntax of arithmetic expressions allows us to say whether some given string is a correctly written arithmetic expression, and, if it is, how it is put together.
- **Semantics** refers to meaning. Knowing the semantics of arithmetic expressions allows us to find their numerical values.

Regular Languages & Regular Expressions
Regular Expressions — Warm-Up: Arithmetic Expressions [2/4]

We can specify the syntax of arithmetic expressions by showing how to build them from small pieces.

We are *not* describing regular expressions here!

First we list the pieces.

- A *numeric literal* is an arithmetic expression: 26.5.
- An *identifier* (think “variable”) is an arithmetic expression: x.

Next we list the ways to build new arithmetic expressions out of existing ones. If A and B are arithmetic expressions, then so are all of the following.

- $-A$
- $A*B$
- A/B
- $A+B$
- $A-B$

Regular Expressions — Warm-Up: Arithmetic Expressions [3/4]

We are *not* describing regular expressions here!

The list, again:

- *–A*
- *A***B*
- *A*/*B*
- *A*+*B*
- *A*–*B*

The above goes from highest precedence (unary "–") to lowest (binary "–"). Unary minus is right-associative, while all four binary operators are left-associative.

Left-associative means, for example, that 1–2–3 is the same as (1–2)–3, not 1–(2–3). **Right-associative** is the reverse.

If we want to override these precedence & associativity rules, then we can use parentheses for grouping. If particular, if *A* is an arithmetic expression, then so is the following.

- (*A*)

Regular Languages & Regular Expressions
Regular Expressions — Warm-Up: Arithmetic Expressions [4/4]

We are *not* describing regular expressions here!

We have defined the syntax of arithmetic expressions. Using the rules covered, we can look at some text and determine whether the text is actually an arithmetic expression. We can also figure out the structure of the expression: how it is put together.

However, we have *not* explained how to find the value of an arithmetic expression. The rules covered so far do not tell us what such an expression means: its semantics.

We can specify the semantics of arithmetic expressions based on our description of the syntax.

- The value of a numeric literal is its numeric value.
- The value of an identifier is the value of the variable it names.
- The value of –*A* is –1 times the value of *A*.
- The value of *A***B* is the product of the value of *A* and the value of *B*.
- Etc.

Regular Languages & Regular Expressions
Regular Expressions — Syntax [1/2]

Now we specify the syntax of **regular expressions** (or **regexes**). As we did with arithmetic expressions, we do this by showing how to build them from small pieces.

First we list the pieces.

- A *single character* is a regular expression: *a*.
- The *empty string* is a regular expression: ε.

Next we list the ways to build new regular expressions out of existing ones. If *A* and *B* are regular expressions, then so are all of the following.

- *A**
- *AB*
- *A*|*B*

Regular Languages & Regular Expressions
Regular Expressions — Syntax [2/2]

The list, again:

- $A*$
- AB
- $A|B$

The above goes from high to low precedence. All are left-associative.

Parentheses can be used for grouping, to override precedence & associativity. In particular, if A is a regular expression, then so is the following.

- (A)

For example, here is a regular expression: $(a|x)*cb$

Regular Languages & Regular Expressions
Regular Expressions — Semantics [1/2]

We can now determine whether a given string is a regular expression, and, if it is, find its structure. Next we discuss what regular expressions mean: their semantics.

Regular expressions are a kind of language generator. A regular expression is said to **match** certain strings. The language generated by the regular expression consists of all strings that it matches.

Once again, we can describe the semantics based on our description of the syntax.

Here are the rules for what the pieces match.

- A single character matches itself, and nothing else.
- The empty string matches itself, and nothing else.

Regular Languages & Regular Expressions
Regular Expressions — Semantics [2/2]

Now suppose that *A* and *B* are regular expressions.

- *A** matches the concatenation of zero of more strings, each of which is matched by *A*.
  - Note that *A** matches the empty string, no matter what *A* is.
- *AB* matches the concatenation of any string matched by *A* and any string matched by *B*.
- *A*|*B* matches all strings matched by *A* and also all strings matched by *B*.
- (*A*) matches the same strings that are matched by *A*.

The asterisk (*), used as above, is called the **Kleene Star**, after Stephen Kleene, a 20th century mathematician who worked in mathematical logic. “Kleene” is, somewhat mysteriously, pronounced KLAY-nee.

Regular Languages & Regular Expressions
Regular Expressions — Language Generated [1/3]

Again, the language generated by a regular expression consists of all strings that it matches.

Fact. The languages that are generated by regular expressions are precisely the regular languages.

That is:

- For each regular expression, the language it generates is a regular language.
- For each regular language, there is regular expression that generates it.

Regular Languages & Regular Expressions
Regular Expressions — Language Generated [2/3]

Consider the regular expression mentioned previously:

(*a*|*x*)**cb*

What language does this regular expression generate?

Each of the expressions "*a*" and "*x*" matches itself.

The expression "*a*|*x*" matches two strings: "*a*" and "*x*".

So the expression "(*a*|*x*)*" matches any string consisting of nothing but *a*'s and *x*'s. For example, it matches "*aaaxaxaaaxxx*". It also matches the empty string.

We conclude that the expression "(*a*|*x*)**cb*" matches zero or more *a*'s and/or *x*'s, followed by *c*, followed by *b*. For example, it matches *cb*, *acb*, *xcb*, *aacb*, *axcb*, *xacb*, *xxcb*, *aaacb*, *aaxcb*, etc.

Regular Languages & Regular Expressions
Regular Expressions — Language Generated [3/3]

Watch out for precedence! In particular, the Kleene star is a high-precedence operator.

For example, as we have said, this regular expression

$$(a|x)*$$

matches any string consisting of nothing but *a*’s and *x*’s.

On the other hand, the following two regular expressions

$$a|x*$$
$$a|(x*)$$

(which are essentially the same) match the string “*a*”, along with any string consisting of zero or more *x*’s: *a*, *ε*, *x*, *xx*, *xxx*, etc.

Regular Languages & Regular Expressions
Regular Expressions — TRY IT #1 (Exercise)

Exercise

1. What language does the following regular expression generate?

$$(xy)^*\underbrace{(\,|\,t)}$$

What comes before the vertical bar? The empty string. You can think of this as “$(\varepsilon|t)$”, although we usually would not write it that way.

Regular Languages & Regular Expressions
Regular Expressions — TRY IT #1 (Answer)

Answer

1. What language does the following regular expression generate?

   $(xy)^*(|t)$

The language containing all strings that consist of zero or more repetitions of "*xy*" followed by an optional "*t*":

$\{\varepsilon, xy, xyxy, xyxyxy, \ldots, t, xyt, xyxyt, xyxyxyt, \ldots\}$

Regular Languages & Regular Expressions
Regular Expressions — TRY IT #2 (Exercises)

Consider the language containing all strings consisting of zero or more *x*’s, followed by either *y* or *z*. That is,

{ *y*, *xy*, *xxy*, *xxxy*, *xxxxy*, …, *z*, *xz*, *xxz*, *xxxz*, *xxxxz*, … }

This is a regular language.

Exercises

2. Write a regular expression that generates the above language.
3. Draw the diagram of a DFA that recognizes this language.

Regular Languages & Regular Expressions
Regular Expressions — TRY IT #2 (Answers)

Consider the language containing all strings consisting of zero or more *x*'s, followed by either *y* or *z*. That is,

{ *y*, *xy*, *xxy*, *xxxy*, *xxxxy*, …, *z*, *xz*, *xxz*, *xxxz*, *xxxxz*, … }

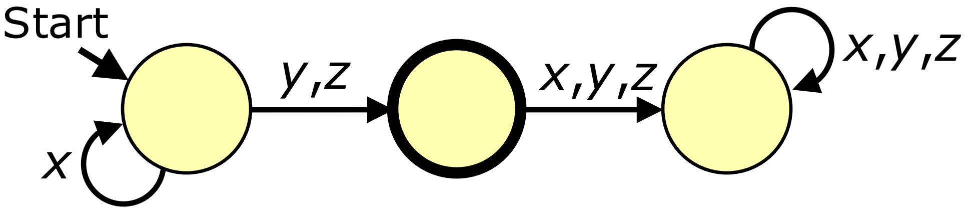
This is a regular language.

Answers

2. Write a regular expression that generates the above language.

*x**(*y*|*z*) OR *x***y*|*x***z*

3. Draw the diagram of a DFA that recognizes this language.



Other answers are possible.

Regular Languages & Regular Expressions
Regular Expressions in Practice [1/7]

Regular-expression libraries are available in many programming languages. They are used by various command-line tools and advanced search options in some applications. We look at the syntax typically used.

It is common—but not universal!—for slashes to be used as delimiters for regular expressions (for example, `/a*(b|c)/`). These are not part of the regular expression itself, just as beginning and ending quotes are not part of the content of a string (`"abc"`).

Here and later in the semester we will refer to characters used as delimiters using the terminology shown.

Parentheses	(	)
Brackets	[	]
Braces	{	}
Angle brackets	<	>

Regular Languages & Regular Expressions
Regular Expressions in Practice [2/7]

Regular-expression libraries typically accept something like the syntax we have described, except that "ε" is replaced by an actual empty string. In addition, a number of shortcuts are commonly used.

First, "`.`" matches any single character, except possibly the end-of-line character.

Second, brackets with a list of characters between them will match any one of the characters in the list. The following two regular expressions match the same strings.

```
/[qwerty]/
/(q|w|e|r|t|y)/
```

Regular Languages & Regular Expressions
Regular Expressions in Practice [3/7]

With the bracket syntax, “-” specifies a range of consecutive characters. The following expressions match the same strings.

```
/[0-9]/
/[0123456789]/
/(0|1|2|3|4|5|6|7|8|9)/
```

So the following will match any single ASCII letter.

```
/[A-Za-z]/
```

Placing “^” just after the opening bracket means that all characters *not* in the list are matched. So this regular expression

```
/[^A-Za-z]/
```

matches any single character that is *not* an ASCII letter.

Regular Languages & Regular Expressions
Regular Expressions in Practice [4/7]

Third, "+" means one-or-more, in the same way that "*" means zero-or-more. So the following two expressions match the same strings.

```
/(abc)+/
/abc(abc)*/
```

Fourth, "?" means zero-or-one. So the following two expressions match the same strings.

```
/x(abc)?/
/x|xabc/
```

Regular Languages & Regular Expressions
Regular Expressions in Practice [5/7]

Last, the various special characters above are treated as ordinary characters when preceded by a backslash(`\`); this is called **escaping**.

The rules for backslash escaping vary from one regular-expression library to another. See your library’s documentation.

For example, “`.`” matches any character, while “`\.`” matches only “`.`”.

To match a single backslash, use an escaped backslash: “`\\`”.

To match a slash, use an escaped slash: “`\/`”.

The extras we have mentioned so far are all just shortcuts. They make regular expressions more convenient, but they do not allow for the generation of any new languages.

In answers to assignments, quizzes, and exams in this class, I will allow any of the shortcuts we have mentioned so far.

Regular Languages & Regular Expressions
Regular Expressions in Practice [6/7]

In regex libraries, it is common for matching functions to determine whether a regex matches *some substring* of a given string. To match the whole string, one can typically use "`^`", which matches the beginning of a string, and "`$`", which matches the end. For example:

```
/^ab*c$/
```

A program that applies a regex to a file will typically try to match each line, in turn.

TO DO

- Try out some practical regexes.

See `regex.py`, `regex2cpp`.

Regular Languages & Regular Expressions
Regular Expressions in Practice [7/7]

Many programming languages & libraries include facilities that make their "regular expressions"—so called—decidedly non-regular. That is, they allow for the generation of languages that are not regular.

One way to do this is to allow a requirement that two sections of a string are the same. For example, the following regex, used in Perl or Python, matches strings `b`, `aba`, `aabaa`, `aaabaaa`, etc.

```
/(a*)b\1/
```

The language generated by this expression is the same language given earlier as an example of a language that is not regular. For the purposes of this class, we do *not* consider the above to be a regular expression.

Regular Languages & Regular Expressions Wrap-Up

Regular languages form the smallest of the four classes of languages in the Chomsky hierarchy. These languages, and related ideas, are used in lexical analysis (lexing), and in text search/replace.

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

$A \to \varepsilon$ $\qquad$ $A \to b$ $\qquad$ $A \to bC$

Regular languages are precisely those languages that are recognized by some **DFA**.

Regular languages are the languages that can be generated by a **regular expression**—in the strict sense of the term.