

Regular Languages

CS F331 Programming Languages

CSCE A331 Programming Language Concepts

Lecture Slides

Friday, January 17, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

© 2017–2020 Glenn G. Chappell

Review

“**Dynamic**” means *at runtime*.

“**Static**” means *before runtime*.

Syntax is the correct *structure* of code.

Semantics is the *meaning* of code.

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.

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

Not the same as a
programming language!

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 $\{0, 1\}$:

$\{\epsilon, 01, 0101, 010101, 01010101, \dots\}$

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).

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.

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.

Here is a grammar with three productions:

$$S \rightarrow yS$$

$$S \rightarrow x$$

Two kinds of symbols:

$$S \rightarrow \varepsilon$$

- **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. Since the late 1970s, nearly all programming languages have used a grammar for their a syntax specification.

Grammar

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

The numbers and underlining are annotations that I find helpful. Strictly speaking, they are not part of the derivation.

Derivation of yyy

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

No " ε " appears here.

Using a grammar:

- Begin with the start symbol.
- Repeat:
 - Apply a production, replacing 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.

Grammar

$$S \rightarrow yS$$

$$S \rightarrow x$$

$$S \rightarrow \varepsilon$$

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

- So yyy lies in the language generated by the above grammar.

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 .

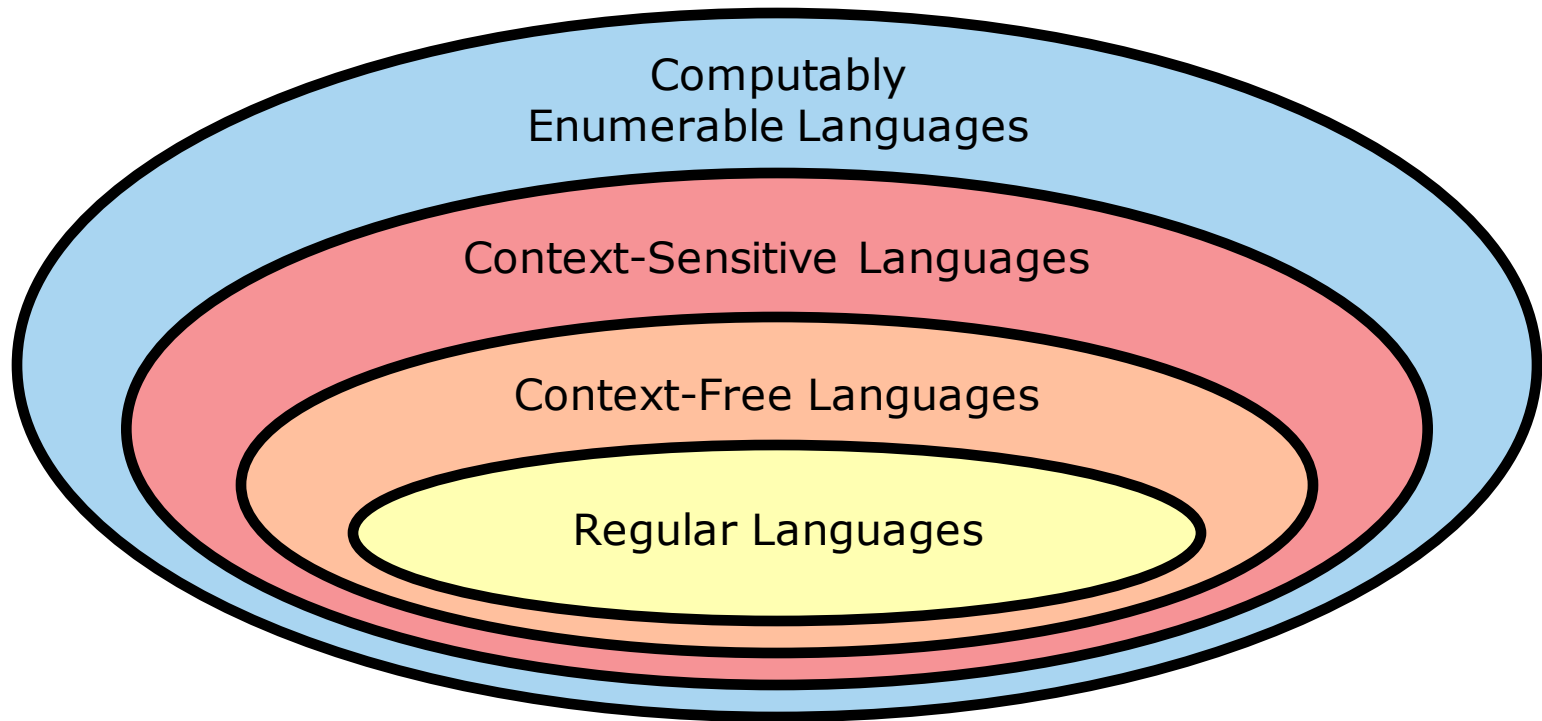
$$\{\varepsilon, y, yy, yyy, \dots, x, yx, yyx, yyyx, \dots\}$$

Avoid saying,
"any number of ...".

The **Chomsky Hierarchy** classifies languages according to the kinds of grammars that can generate them.

Language Category		Generator	Recognizer
Number	Name		
Type 3	Regular	Grammar in which each production has one of the following forms. <ul style="list-style-type: none"> • $A \rightarrow \varepsilon$ • $A \rightarrow b$ • $A \rightarrow bC$ 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. <ul style="list-style-type: none"> • $A \rightarrow [\text{anything}]$ 	Nondeterministic Push-Down Automaton Think: Finite Automaton + Stack (roughly).
Type 1	Context-Sensitive	<i>Don't worry about it.</i>	<i>Don't worry about it.</i>
Type 0	Computably Enumerable	Grammar (no restrictions).	Turing Machine Think: Computer Program

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



Our next topics concern the smallest two categories in the Chomsky Hierarchy.

Regular Languages

Regular Languages

Introduction

Now we look closer at the smallest of the four categories of languages in the Chomsky Hierarchy: the *regular languages*.

Regular languages have two important applications.

- In most programming languages, the set of all lexemes (words, roughly) of a particular kind forms a regular language. Thus we make use of regular languages in the early stages of compilation or interpretation, when we break up a program into lexemes—a process called **lexical analysis**, or **lexing**.
- Regular languages are heavily used in text search/replace.

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

$$A \rightarrow \varepsilon$$

$$A \rightarrow b$$

$$A \rightarrow bC$$

If you have studied this kind of grammar before, then you might have called it a *right-regular grammar*.

That is, the left-hand side of each production is a single nonterminal, while the right-hand side is one of:

- the empty string
- a single terminal, or
- a single terminal followed by a single nonterminal—*which may be the same as the left-hand side*.

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

Here is an example of a regular grammar.

$$S \rightarrow \varepsilon$$
$$S \rightarrow t$$
$$S \rightarrow xB$$
$$B \rightarrow yS$$

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more concatenated copies of xy , followed by an optional t .

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

So this language is a regular language.

Here is another grammar. This is *not* a regular grammar.

$S \rightarrow A$

$S \rightarrow At$

$A \rightarrow Axy$

$A \rightarrow \varepsilon$

Q. What language does this grammar generate?

A. The set of all strings that consist of zero or more concatenated copies of xy , followed by an optional t .

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

Q. Is this a regular language?

A. Yes! Because it is generated by a regular grammar: the one on the previous slide.

There are languages that are not regular.

For example, this grammar ...

$$S \rightarrow aSa$$
$$S \rightarrow b$$

... generates the following language.

$$\{b, aba, aabaa, aaabaaa, \dots\} = \{a^kba^k \mid k \geq 0\}$$

There is *no* regular grammar that generates this language. It is not a regular language. (I am not saying this is obvious; but it is true. Proving that a language is not regular is beyond the scope of this class.)

Regular Languages

Finite Automata — Basics [1/4]

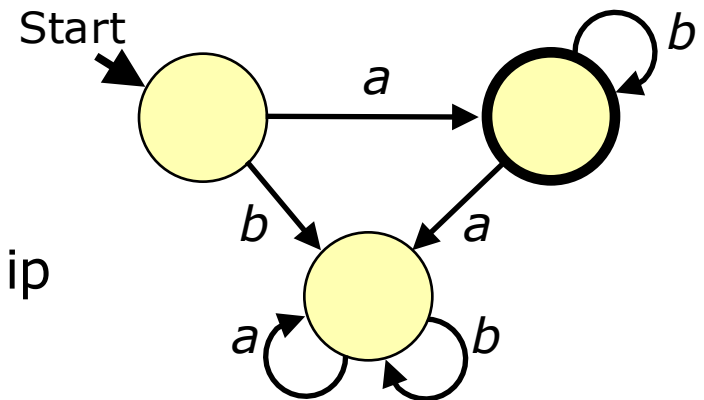
A **deterministic finite automaton** (Latin plural “**automata**”), or **DFA**, is a kind of recognizer for regular languages.

A DFA consists of a finite collection of **states**, with **transitions** between these states.

- One state is the **start state**.
- Some states may be **accepting states**.
- Each transition begins at a state, ends at a state, and is associated with a character in the alphabet—that is, some terminal symbol.
- For each character, each state has *exactly one* transition leaving it that is associated with that character.

Here is a **diagram** of a DFA with three states, one of which is accepting.

On the next slide we make the relationship between a DFA and its diagram more precise.

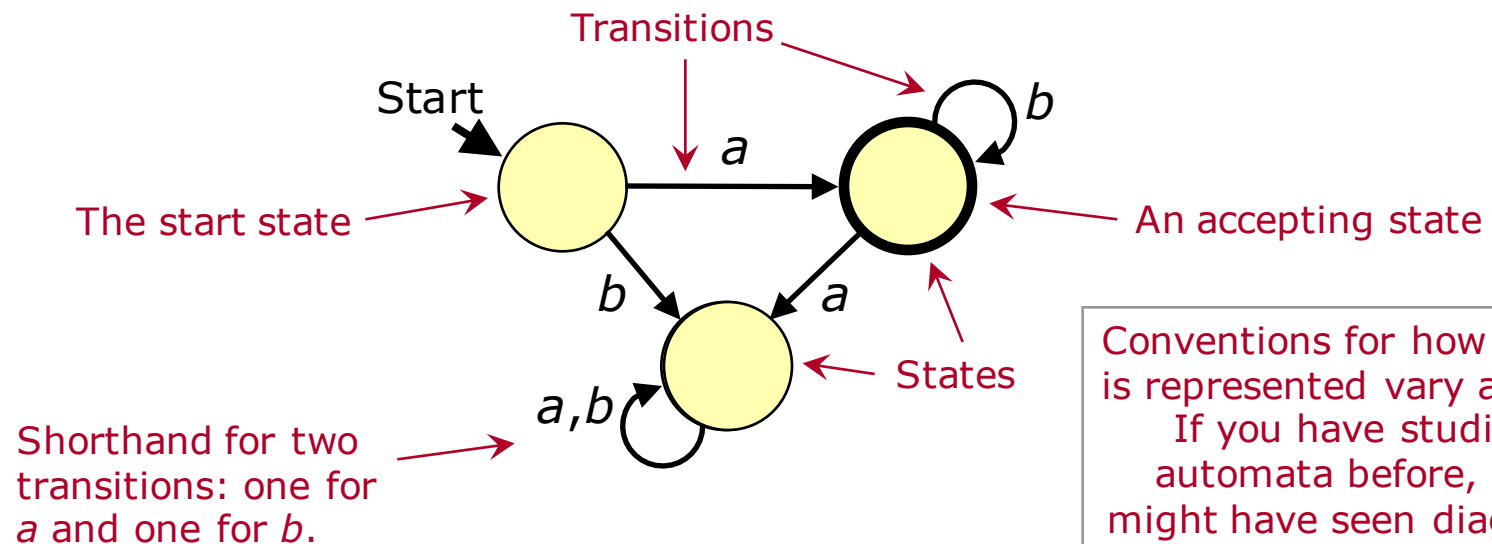


Regular Languages

Finite Automata — Basics [2/4]

To make a diagram of a DFA, we draw a circle (or other enclosed shape) for each state. For each transition, we draw an arrow from the state it begins at to the state it ends at, labeling the arrow with the transition's character.

We make accepting states bold, and we draw an arrow labeled "Start" to the start state.



Conventions for how a DFA is represented vary a little. If you have studied automata before, you might have seen diagrams that were slightly different.

Regular Languages

Finite Automata — Basics [3/4]

We use a DFA as a recognizer as follows.

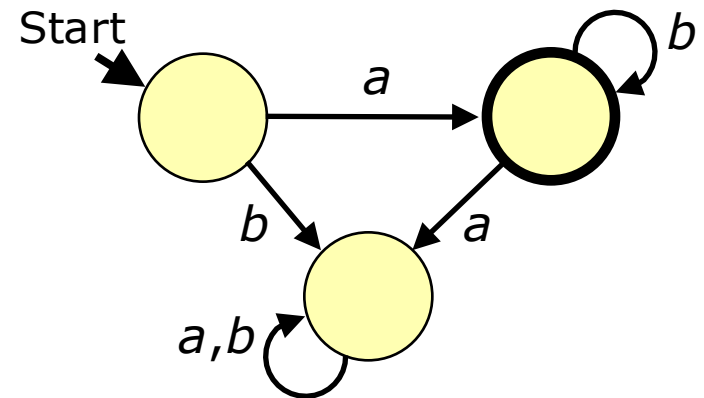
- We are always in one of the states, beginning with the start state.
- We proceed in steps. At each, we read a character from the input and follow the transition beginning at the current state and labeled with the character that was read. Where this ends is our new state.
- If, when we reach the end of the input, we are in an accepting state, then we **accept** the input; otherwise we do not accept.

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

Q. What language does this DFA recognize?

A. The set of all strings that consist of an *a* followed by zero or more *b*'s.

$\{a, ab, abb, abbb, abbbb, abbbbbb, \dots\}$



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

That is:

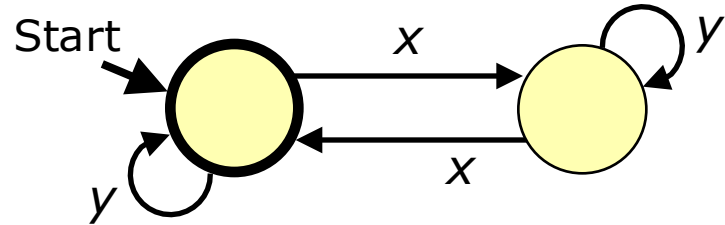
- For each DFA, the language it recognizes is a regular language.
- For each regular language, there is a DFA that recognizes it.

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

Finite Automata — Try It (Exercise)

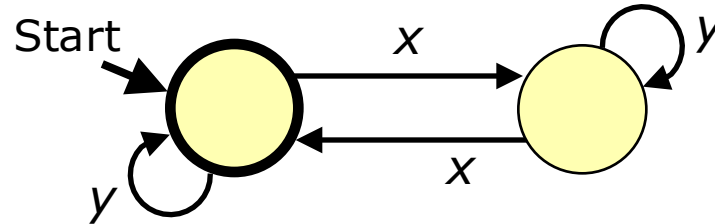


Exercise

1. What language does the above DFA recognize?

Regular Languages

Finite Automata — Try It (Answer)



Answer

1. What language does the above DFA recognize?

The language containing all strings of x and y characters that have an even number of x characters (including the empty string!).

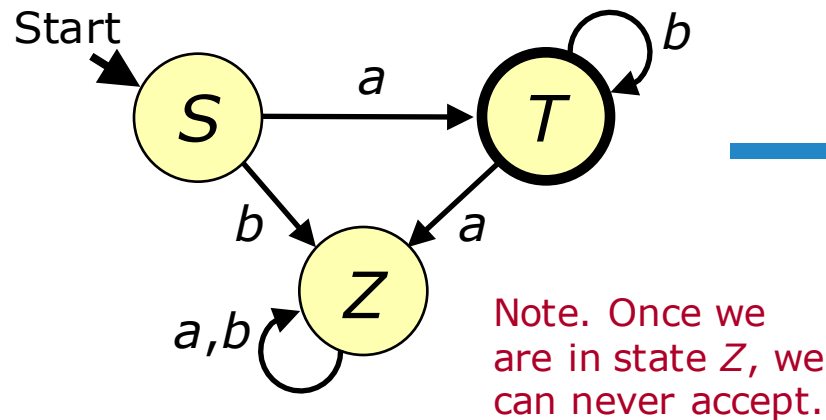
Given a DFA, we can write a regular grammar that generates the same language that the automaton recognizes.

Procedure

- Make one nonterminal symbol for each state. The nonterminal corresponding to the start state is the start symbol.
- For each transition, make a production. A transition from state A to state B labeled with character x gives the production $A \rightarrow xB$.
- For each accepting state, make a production whose right-hand side is the empty string. If state A is accepting, then write $A \rightarrow \epsilon$.

- For each state, 1 nonterminal. Start state gives start symbol.
- For each transition, 1 production. A to B by x gives $A \rightarrow xB$.
- For each accepting state, 1 production. A accepting gives $A \rightarrow \varepsilon$.

DFA with states labeled



Regular Grammar

$S \rightarrow aT$

$S \rightarrow bZ$

$T \rightarrow aZ$

$T \rightarrow bT$

$Z \rightarrow aZ$

$Z \rightarrow bZ$

$T \rightarrow \varepsilon$

Productions involving Z can never be used in a derivation, so we may omit them.

Language Recognized/Generated

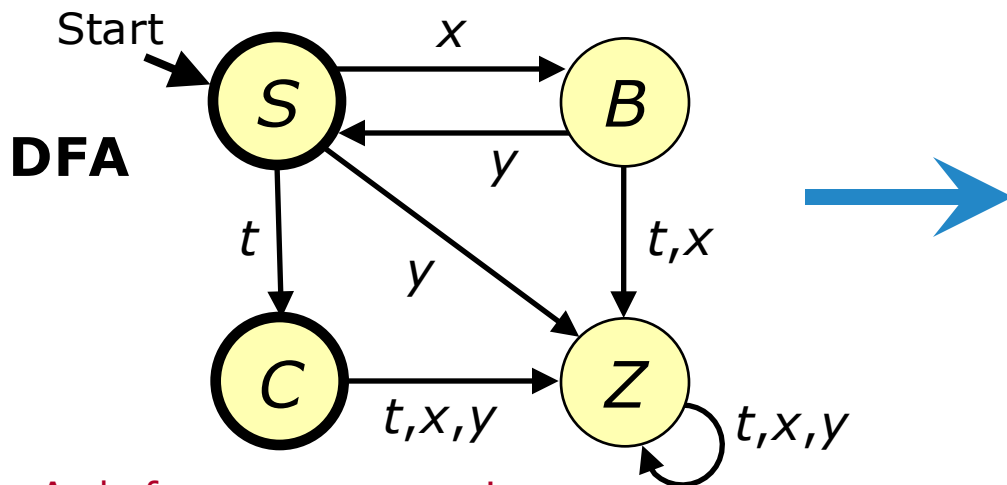
$\{a, ab, abb, abbb, abbbb, abbbbbb, \dots\}$

Regular Languages

Finite Automata — DFAs & Regular Grammars [3/4]

Another Example

Regular Grammar



As before, once we are in state Z , we can never accept.

Language Recognized/Generated

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

$S \rightarrow tC$

$S \rightarrow xB$

$S \rightarrow yZ$

$B \rightarrow tZ$

$B \rightarrow xZ$

$B \rightarrow yS$

$C \rightarrow tZ$

$C \rightarrow xZ$

$C \rightarrow yZ$

$Z \rightarrow tZ$

$Z \rightarrow xZ$

$Z \rightarrow yZ$

$S \rightarrow \varepsilon$

$C \rightarrow \varepsilon$

Once again, productions involving Z can never be used in a derivation, so we may omit them.

We can also do the transformation the other way: begin with a regular grammar, and produce a DFA that recognizes the same language the grammar generates. However, the process is more complex, and we will not cover it.

You may stop reading this slide here.

For those with some knowledge of automata:

- The complexity in the Regular Grammar \rightarrow DFA process is due to the rule that, for each character, each state in a DFA has *exactly one* transition leaving it that is associated with that character.
- Transforming a regular grammar into an automaton in the obvious way can violate that rule, giving an NFA, not a DFA (“N” for “nondeterministic”).
- An NFA can be transformed into a DFA, but the result may have a huge number of states. An n -state NFA gives a DFA with 2^n states.
- Typically, there are unnecessary states, which can be eliminated.

Regular Languages

Regular Expressions — Introduction

We wish to define a kind of generator called a *regular expression*. 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 practice, we will describe the syntax and semantics of arithmetic expressions, using informal methods. Afterward, we will describe regular expressions in a similar way.

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

We are *not* describing regular expressions here!

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. For arithmetic expressions, this would specify an expression's numerical value.

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
- $A+B$
- $A-B$

The list, again:

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

We are *not*
describing regular
expressions here!

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 $1-(2-3)$.

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

- (A)

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.

We are *not* describing regular expressions here!

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 — Syntax [1/2]

Now we specify the syntax of **regular expressions**. 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 — 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 — Semantics [1/2]

We can now determine whether a given string is a regular expression, and find its structure, if so. 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 matching the pieces.

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

Regular Languages

Regular Expressions — Semantics [2/2]

Now suppose that A and B are regular expressions.

- A^* matches the concatenation of zero or 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.

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.

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.

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: ε , x , xx , xxx , etc.

Exercise

1. What language does the following regular expression generate?

$$(xy)^*(\varepsilon|t)$$

Answer

1. What language does the following regular expression generate?

$$(xy)^*(\varepsilon|t)$$

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

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

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

$$\{ y, xy, xxy, xxxy, xxxxy, \dots, z, xz, xxz, xxxz, xxxxz, \dots \}$$

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.

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

$$\{ y, xy, xxy, xxxy, xxxxy, \dots, z, xz, xxz, xxxz, xxxxz, \dots \}$$

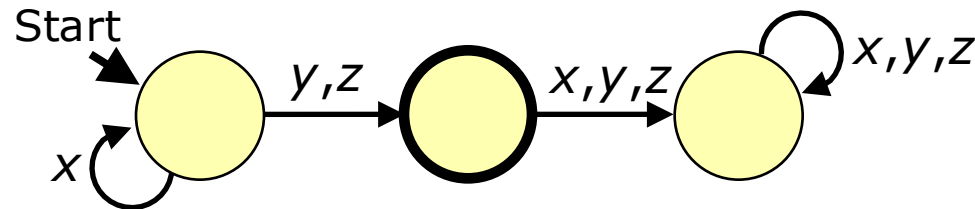
This is a regular language.

Answers

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

$$x^*(y|z)$$

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



Regular Languages TO BE CONTINUED ...

Regular Languages will be continued next time.