Doctor Syntax was the fictitious hero of several 19th century satirical poems by William Combe (1742–1823). The publications were illustrated by Thomas Rowlandson (1756–1827).

Rowlandson’s many comic illustrations offered humorous commentary on the political and social conventions of his day. Rowlandson is perhaps best known for his animated drawings of the mis-adventures of the infamous Dr. Syntax. The various escapades of the fictional English clergyman and schoolmaster, Dr. Syntax, are chronicled in three volumes or “Tours.” The Tours of Dr. Syntax were satires on a then contemporary series of picturesque adventures to various parts of England with an emphasis on landscape illustration and poetic verse.

- The tour of Doctor Syntax in search of the picturesque
- The second tour of Doctor Syntax in search of consolation
- The third tour of Doctor Syntax in search of a wife
Overview of Syntax

- Study of Language — page 4
- Structure — page 9
- Formal Languages — page 27
- Regular Expressions — page 49
- BNF, * Context Free Grammar, parse trees — page 92
- *Attribute Grammars, *Post Systems
Semiotics. Date: 1880. A general philosophical theory of signs and symbols that deals especially with their function in both artificially constructed and natural languages. The application of linguistic methods to other than natural languages. A philosophy that emphasizes the significance of language as universal.

In his 1946 book, *Signs, Language, and Behavior*, American philosopher Charles William Morris (1901–1979) gave these definitions for the branches of the science of signs

- *pragmatics* — origin, uses and effects of signs with the behavior in which they occur;
- *semantics* — the signification of signs in all modes of signifying;
- *syntax* — combination of signs without regard for their specific significations or their relations to the behavior in which they occur.
Study of Language

- **syntax** is the form, structure of language
- **semantics** is the meaning of language
- **pragmatics** is origin, use, context of language

Distinction between semantics and syntax:
1. number versus numeral
2. character versus glyph
3. conjunction versus $\land$
Syntax in NL

'Twas brillig, and the slithy toves
   Did gyre and gimble in the wabe;
All mimsy were the borogoves,
   And the mome raths outgrabe.

Lewis Carroll, “Jabberwocky”

We know certain things—for instance, that all this happened in the past and that there was more than one tove but only one wabe. We know these things because of the grammatical structure of the English language.
Can you translate the poem “Jabberwocky” into other languages?
Can you translate the poem “Jabberwocky” into other languages?

’Twas BRKPT and the I/O queue
    Was SYMMING FASTRAND like the wind.
All idle was the CAU
    As the last run had just FINNED.

“Beware the UNIVAC, my son,
    Its FASTRAND and its high-speed drum,
And FIELDATA, and listen for
    The CTMC’s hum.”

Harry Gilbert, mid 1970s
Also, Arpawocky, RFC527, 22 June 1973, by D. L. Covill

Twas brillig, and the Protocols
    Did USER-SERVER in the wabe.
All mimsey was the FTP,
     And the RJE outgrabe,

Beware the ARPANET, my son;
    The bits that byte, the heads that scratch;
Beware the NCP, and shun
    the frumious system patch,
Structure

The arrangement of elements of an entity and their relationships to each other.

- Linear structure – like links of a chain
- Hierarchical structure – like nested mixing bowls, or boxes within boxes
- Trees – like family tree, or railroad yard

Grammars are a way of describing these and more complex structures.
Linear Structure

Links in a chain
Hierarchical Structure

Nested mixing bowls
Tree Structure

Railroad yard
Even though it is invisible, structure has great importance, even meaning. Sentence structure is crucial to disambiguate the following sentences.

1. The chicken is ready to eat.
Structure

Even though it is invisible, structure has great importance, even meaning. Sentence structure is crucial to disambiguate the following sentences.

1. The chicken is ready to eat.
   Who is eating?

2. He saw that gasoline can explode.

Even though it is invisible, structure has great importance, even meaning. Sentence structure is crucial to disambiguate the following sentences.

1. The chicken is ready to eat.  
   Who is eating?
2. He saw that gasoline can explode.  
   What is exploding?
3. Flying planes can be dangerous.
Even though it is invisible, structure has great importance, even meaning. Sentence structure is crucial to disambiguate the following sentences.

1. The chicken is ready to eat.
   Who is eating?
2. He saw that gasoline can explode.
   What is exploding?
3. Flying planes can be dangerous.
   What is dangerous?
programming languages

formal languages

expressivity

compilers

implementation, recognition

language design, description, semantics
CS Theory Classes

- **complexity**
- **computability**
- **efficiency**
- **formal languages, automata**
- **limits**
- **models**
Bad/Confusing Syntax In PL

The science of syntax is about unambiguous communication and as such is neutral. Let us at least be clear. Hard to define “good” and “bad” syntax. Only experience helps, and that takes time. The lessons of programming language design are ad hoc.
Bad/Confusing Syntax In PL

DO 10 I = 1.5
   A(I) = X + B(I)
10  CONTINUE

IF IF=THEN THEN THEN=ELSE; ELSE ELSE=IF
IF IF THEN THEN=ELSE; ELSE ELSE=0;

while ( ... ) {
   ...
   break;
   ...
}
In the study of compilers there are many fine-grained distinctions made about grammars. These distinction might point out features that humans might well avoid.

The following is not LL, but is LR:

```plaintext
stm ::= "if" expr "then" stm "else" stm
    | "if" expr "then" stm
```
Lexical Structure Versus Phrase Structure

In lexical structure the alphabet used is a set of characters (e.g., Latin1) and the goal is to categorize substring into different tokens each described by their own language.

In phrase structure, the alphabet used is a set tokens, and one goal is to determine if the program is syntactically correct (i.e., is the phrase in the language or not). Actually the goal is more broad than that. The goal is to identify the phrase structure or derivation of syntactically correct programs.
Three Views of a Program

```ada
with Ada.Text_IO;
use Ada;

-- program to do one thing: print a message
procedure World is
-- Some text given a name for later use
   Msg: constant String := "Hello world!";
begin
   Text_IO.Put_Line (Msg);
end World;
```
Program as Stream of Characters

```ada
with Ada.Text_IO;
use Ada;

-- program to do one thing: print a message
procedure World is
  -- Some text given a name for later use
  Msg: constant String := "Hello world!";
begin
  Text_IO.Put_Line (Msg);
end World;
```

```
with -- begin 2 * x;
```
Program as Stream of Tokens

```
with Ada.Text_IO;
use Ada;

-- program to do one thing: print a message
procedure World is
  -- Some text given a name for later use
  Msg: constant String := "Hello world!";
begin
  Text_IO.Put_Line (Msg);
end World;
```

```
with --begin 2 * x;
```
Program as Tree Structure

```ada
with Ada.Text_IO;
use Ada;

-- program to do one thing: print a message
procedure World is
  -- Some text given a name for later use
  Msg: constant String := "Hello world!";
begin
  Text_IO.Put_Line (Msg);
end World;
```

```
with -- begin 2 * x;
```
Free-format as opposed to one statement per line. In earlier versions of FORTRAN a fixed lexical structure of one statement per punch card stating on column 7. COBOL and Basic also used to be fixed format. For the most part languages today permit the programmer to choose the white space (and hence the layout) of the program. Some layouts are clearly more legible and others, so the need for style guidelines. Python, Haskell, and Occam require proper indenting.

Should a language specify the layout (use of white space)?
“Hello World!” in Whitespace (S=space, T=tab, L=line feed)
“Hello World!” in Whitespace (S=space, T=tab, L=line feed)
Formal Language Versus Natural Language

Natural language is full of unresolvable differences and “jagged edges.” (This makes it rich and interesting.)
Formal Language Versus Natural Language
Formal Language

First some definitions:

- **object language**, the language being studied and **meta language**, the language used to communicate
- **symbol**
- **alphabet** is a set of symbols
- **strings** is an ordered sequence of zero, one or more symbols
- **formal language** is a set of strings

Unlike natural languages, it is unambiguous for a formal language \( L \) if \( s \in L \) or \( s \notin L \).

The key challenge is to find good (finite) representation of (infinite) sets.
Object and Meta Language

Object language versus meta-language. The object language is the language being studied, and the meta-language is the language being used to communicate. For example, in the class French 101 one may study the French language, it is the object language. To communicate to the students it may be necessary for the teacher to speak in English. English is the meta-language in this case.
What is a *symbol*?

We want to make understandable the definition of an *alphabet* as a finite set of symbols. In the study of formal languages we sometimes fix the universe of symbols (the alphabet) to be, for example, \{0, 1\} or \{a, b, c\}.

This primitive, or *a priori* (in the sense of Kant), concepts (like symbol) are awkward to define.

We begin with a simple definition and elaborate later: A symbol is a sign or a mark that can be distinguished from any other symbols. One can think of it as a letter in a word, or a digit in a number.
“Symbol” in other Languages

- Arabic: رمز (ar) m (ramz)
- Armenian: հարվարդանիշ (hy) (xorhrdaniš), սիմվոլ (hy) (simvol)
- Asturian: símbolu m
- Basque: ikur
- Bavarian: Sümbol
- Belarusian: сімвал m (símval), знак m (znak)
- Bulgarian: символ (bg) m (símvol)
- Catalan: símbol (ca) m
- Chinese:
  Mandarin: 符號 (zh), 符号 (zh) (fúhào)
Alphabet

*Alphabet.* The *alphabet* is the finite set of symbols of indivisible elements making up the formal language. These are the basic, atomic, distinct elements.

\[
\Sigma = \{a, b, c, d\}
\]
\[
\Sigma = \{0, 1\}
\]
\[
\Sigma = \{a, b, \ldots, z, 0, 1, \ldots, 9\}
\]
\[
\Sigma = \text{Latin-1}
\]
\[
\Sigma = \text{Unicode}
\]
\[
\Sigma = \{\text{if, then, (, ), ', ', ...}\}
\]

Theoreticians like to use the set \(\Sigma = \{0, 1\}\) because it is so simple. The symbols might even be composed of other sub-atomic symbols. Important character sets for programming languages to use and be written in are Latin-1, Latin-0, and Unicode, not just US-ASCII.
Unicode Supports Many Scripts

Latin: A B C D E ...

Arabic: ا ب ت ث ج ح خ ...

Hebrew: א ב ג ד ה ...

Armenian: փ խ հ վ տ ...

Cyrillic: А Б В Г Д ...

Devanagari: क ख ग घ झ च छ ...

Thai: ก ข ฃ ค ฅ ฆ โ ใ ๅ
Strings

Given an alphabet $\Sigma = \{a, b, c, d\}$. A string is an (ordered) sequence of symbols from the alphabet. E.g., $aaa$, $bac$, $d$. The sequence may contain no symbols, this empty string is given a special notation "". (Some authors use $\epsilon$, the Greek letter epsilon.)

The most important operation on strings is string concatenation which is sometimes denoted by the $\cdot$ operator. E.g.,

$$ab \cdot db = abdb, \quad da \cdot ba = daba.$$
More Strings

Given an alphabet $\Sigma = \{0, 1\}$, the following are strings: $0, \epsilon, 111, 010, 1$.

Given an alphabet $\Sigma = \{a, b, \ldots, z\}$, the following are strings: $\epsilon, x, \text{aeiou, hello, biopsy}$.

Using Latin-1 as the alphabet, the following are strings: ¡Hola!, Hola!, ¿Qué pasa?, "Hi. Bye", grösstergemeinsamer Teiler.

Using the alphabet $\Sigma$

$$\{\text{if, then, else, begin, end, (, ), ' , '}\}$$

the following are strings:

1. else )
2. if then
3. begin begin , , end end
Definition: A formal language is a set of strings over an alphabet.
Significance: A formal language expresses information unambiguously.
Example: $\{ab, bc, acc\}$
Formal Language

Formal language. A formal language is a set of strings over some alphabet.

Given an alphabet $\Sigma = \{0, 1\}$, the following are formal languages:

\[
\begin{align*}
L_0 &= \emptyset \\
L_1 &= \{1\} \\
L_2 &= \{1, 0101\} \\
L_3 &= \{0, 01, 00\} \\
L_4 &= \{1, 01, 11, 001, 011, 101, 111, \ldots\} \\
L_5 &= \{\epsilon, 1, 11, 111, 1111, \ldots\} \\
L_6 &= \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\} = \Sigma^*
\end{align*}
\]

The collection of legal programs in a programming language is also an example of a formal language.
Chomsky Hierarchy

The mechanisms for describing formal languages can be categorized in a hierarchy according to their power. This hierarchy is known as the Chomsky hierarchy after the MIT linguist Noam Chomsky.
Chomsky Hierarchy

<table>
<thead>
<tr>
<th>name</th>
<th>machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 recursively enumerable</td>
<td>Turing machine</td>
</tr>
<tr>
<td>1 context-sensitive</td>
<td>linear-bounded automata</td>
</tr>
<tr>
<td>2 context-free</td>
<td>pushdown automata</td>
</tr>
<tr>
<td>3 regular</td>
<td>finite automata</td>
</tr>
</tbody>
</table>

- Regular expressions are used to describe tokens in programming languages
- BNF, CFG used to describe syntax of programming languages
- [Context-sensitive languages play no role in programming languages]
- attribute grammars, Post systems used in semantics
$U$ is the set of all formal languages.
Each dot is a formal language. $U$ is the set of formal languages: $U = \mathcal{P}(\Sigma^*)$ where $\Sigma$ is the alphabet.
\[ U \]

- Recursive enumerable languages
- Context-sensitive languages
- Context-free languages
- Regular languages
Description Mechanisms

1. Regular expressions (Appel, page 19, but not in Webber or Mitchell; briefly mentioned in Sebesta)
2. BNF (Chapters 2 and three of Webber)
“The programming language Ada expresses computation.”
“The programming language Ada is notation for computation.”
“A program in Java denotes a computation or algorithm.”
“Regular expressions are notation for formal languages.”
“A regular expression denotes a formal language.”
“BNF is notation for formal languages.”
“A BNF definition denotes a formal language.”
Now consider: a programming language (like Ada) is a formal language.
We can try to describe Ada using regular expressions or BNF.
Regular Expressions (Literature)


Sams Teach Yourself Regular Expressions in 10 Minutes, 1st edition. by Ben Forta

Introducing Regular Expressions: Unraveling Regular Expressions, Step-by-Step

Regular Expressions: Simplicity and Power in Code by Douglas McLain Berdeaux

WHENEVER I LEARN A NEW SKILL, I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM, WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

IT'S HOPELESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

xkcd—a webcomic of romance, sarcasm, math, and language by Randall Munroe
Some people, when confronted with a problem think “I know, I’ll use regular expressions.” Now they have two problems.

From a post by Jamie Zawinski to Usenet newsgroup alt.religion.emacs in 1997.
Regular Expressions (Syntax)

- What *are* they? (syntax)
- What do they *mean*? (semantics)

- Can a language denoted by a regular expression be effectively recognized? (Implementation)
- Can tokens in a programming language be effectively recognized? (Scanning)
Regular Expressions (Syntax)

1. Empty. $\emptyset$
2. Atom. Any single symbol of $a \in \Sigma$ is a regular expression.
3. Alternation. If $r_1$ is a regular expression and $r_2$ is a regular expression, then $(r_1 + r_2)$ is a regular expression.
4. Concatenation. If $r_1$ and $r_2$ are regular expressions, then $(r_1 \cdot r_2)$ is a regular expression.
5. Closure. If $r$ is a regular expression, then $(r)^*$ is a regular expression.
Regular Expressions (Syntax)

We omit parentheses following well-known rules:

- The outermost pair of parentheses is distracting.
- Alternation and concatenation can be considered left associative. (Semantically they are associative operators, so it does not make much difference.)
- Closure binds more tightly than concatenation which binds more tightly than alternation. (Similar to exponentiation, multiplication, and addition in traditional arithmetic expressions.)

Also, we sometimes omit the concatenation operator \( \cdot \) using mere juxtaposition to indicate concatenation.
Regular Expressions (Haskell)

```haskell
data Regex a = Empty | Sym a | Star (Regex a) 
  Alt (Regex a) (Regex a) | Concat (Regex a) (Regex a) 
  deriving (Eq, Ord)

Alt (Sym 'a', Concat (Sym 'b', Sym 'c')) -- a+(bc)
Concat (Alt (Sym 'a', Sym 'b'), Sym 'c') -- (a+b)
```
Regular Expressions (Examples)

\[ a \cdot b \]
\[ a + b \]
\[ a \cdot (b + c) \]
\[ a + b \cdot c \]
\[ a^* \]
\[ a + (b \cdot c)^* \]
\[ (a + (b \cdot c))^* \]

(using the alphabet \( \Sigma = \{ a, b, c \} \))
Regular Expressions (Examples)

(Omitting the “centered dot” and using the | for alternation.)

\[ab\]
\[a \mid b\]
\[a(b \mid c)\]
\[a \mid bc\]
\[a^*\]
\[a \mid (bc)^*\]

(using the alphabet \( \Sigma = \{a, b, c\} \))
Regular Expressions (Examples)

01
101
1 + 0
1(0 + 1)
0 + 10
0*
1 + (01)*
(0 + (10)*)*

(using the alphabet \( \Sigma = \{0, 1\} \) and omitting the \( \cdot \) symbol)
That is what regular expressions look like (syntax). What do regular expressions mean (semantics)? Each regular expression denotes a formal language (a set of strings).

There are an infinite number of regular expressions. (Each one is finitely constructed.) Just like programs in programming languages. We must find a way to define the meaning or denotation of each regular expression.

So, we define a (recursive) function from regular expressions (as pieces of syntax) to sets of strings (languages).
1. Empty. The language with no strings.
2. Atom. The language with one string of length one—$a$ itself.
   Note that the notation $a$ is ambiguous. It stands for both the symbol and the language.
3. Alternation. $(r_1 + r_2)$ is the union of two languages.
4. Concatenation. $(r_1 \cdot r_2)$ is the set all strings beginning in one language and then followed by a string in the second.
5. Closure. $(r)^*$ is zero, one, or more strings.
Regular Expressions (Examples)

\[
\begin{align*}
a \cdot b &= \{ab\} \\
a + b &= \{a, b\} \\
a \cdot (b + c) &= \{ab, ac\} \\
a + (b \cdot c) &= \{a, bc\} \\
a^* &= \{\epsilon, a, aa, aaa, \ldots\} \\
a + (b \cdot c)^* &= \{\epsilon, a, bc, bcbc, \ldots\} \\
(a + (b \cdot c))^* &= \{\epsilon, a, bc, aa, abc, bcbc, \ldots\}
\end{align*}
\]

(using the alphabet \(\Sigma = \{a, b, c\}\))
Other Definitions

Some authors replace one of the five cases of the definition:

1. Empty. $\emptyset$ denoting the language with no strings with another case

1. Epsilon. $\epsilon$ denoting the set consisting of the single string ""

Do you see the difference?

Notice the $\epsilon$ is superfluous as $\{\epsilon\}$ is represented by $\emptyset^*$. Can you prove this?
Regular Expressions (Formal Semantics)

A regular expression denotes a formal language (set of strings) over an alphabet $A$ by means of a function $D$. The function $D$ takes a regular expression and associates with it a particular formal language. The function is defined recursively over the five cases of the inductive definition of the set of regular expressions.

1. Empty.

$$D[\emptyset] = \{\}$$

2. Atom. For each $a \in \Sigma$,

$$D[a] = \{a\}$$

3. Alternation.

$$D[(r_1 + r_2)] = D[r_1] \cup D[r_2]$$

\[ D[(r_1 \cdot r_2)] = \{ x \cdot y \mid x \in D[r_1], y \in D[r_2] \} \]

where \( x \cdot y \) is string concatenation.

5. Closure.

\[ D[(r)^*] = \bigcup_i (D[r])^i \]

where \( S^i \) is defined recursively as follows:

\[
\begin{align*}
S^0 &= \{ \epsilon \} \\
S^{i+1} &= \{ x \cdot y \mid x \in S, y \in S^i \}
\end{align*}
\]
Using the Definitions

\[ D[(a \cdot b)] = \{ x \cdot y \mid x \in D[a], y \in D[b] \} \]
\[ = \{ x \cdot y \mid x \in \{ a \}, y \in \{ b \} \} \]
\[ = \{ a \cdot b \} = \{ ab \} \]

\[ D[(a + (a \cdot b))] = D[a] \cup D[(a \cdot b)] \]
\[ = \{ a \} \cup \{ ab \} \]
\[ = \{ a, ab \} \]

\[ D[(a + (b \cdot c))^*] = \bigcup_{i} (D[(a + (b \cdot c))])^i \]
\[ = \{ \epsilon \} \cup \{ a, bc \} \cup D[(a + (b \cdot c))]^2 \cup \ldots \]
\[ = \{ \epsilon, a, bc \} \cup \{ aa, abc, bca, bcbc \} \cup \ldots \]
Is the recursively defined function $D$ well-defined? Yes. When are such recursively defined functions well-defined? Over free-generated sets.
Consider the following inductive definition of a subset $M$ of $Nat$, the natural numbers, using integer multiplication $\cdot$:

- $1 \in M$,
- if $n \in M$, then $9 \cdot n \in M$,
- if $n \in M$, then $23 \cdot n \in M$.

Suppose we define the function $g : M \rightarrow Nat$ inductively by:

- $g(1) = 1$,
- $g(9 \cdot n) = 9$,
- $g(23 \cdot n) = 23$.

Prove that $0=1!$
More Regular Expressions

To make regular expressions more convenient, regular expressions are almost always extended with new notation. Here are some additional meta-symbols commonly seen (perhaps with different syntax). Regular expressions with these new meta-symbols can be defined in terms of the original definitions. Let $r$ be a regular expression over the alphabet $\Sigma$.

1. Optional. $r? = (r + \emptyset^*)$

2. One or more. (This is a second and conflicting use of the meta-character +.) $r^+ = (r \cdot r^*)$

3. Any. $\mathbf{.} = (a + b + \ldots + y + z)$ where $\Sigma = a, b, \ldots, y, z$.

4. Range. $[a - z] = (a + b + \ldots + y + z)$. (Assumes that $\Sigma$ is ordered.)

5. Range complement. $[\neg c - x] = (a + b + y + z)$. (Assumes that $\Sigma$ is ordered.)
Where are regular expressions used?
The program `grep` is a command-line utility for searching text originally written for Unix. The `grep` command searches text files for lines matching a given regular expression and prints matching lines to the program's standard output.
Suppose the file preamble.txt has the following lines:

We the People of the United States, in Order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish this Constitution for the United States of America.

> grep and preamble.txt
common defence, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our Posterity, do ordain and establish

> grep -i people preamble.txt
We the People of the United States, in Order to form a more perf
of Liberty to ourselves and our Posterity, do ordain and establish

(But does not match “domestic.”)

common defence, promote the general Welfare, and secure the Blessings
Practical Regular Expressions

grep options regexp filename

Options:

- E extended regular expression
- i ignore case
- x match whole line
- w match word in line

Syntax of regular expressions for grep

| | or | $ | end of line |
| | . | any | ? | optional |
| [] | set | {n,m} | n through m times |
| [^ ] | set compliment | () | grouping |
grep

grep -E -w -i '[a-f]{3,4}' /usr/dict/words

beef
dead
dead
dead
among others.
grep

grep -E -w -i '[a-f]{3,4}' /usr/dict/words

beef
dead
deaf
fade
feed

among others.
grep

grep -E '.{5,}' /usr/dict/words
grep

grep -E '.{5,}' /usr/dict/words

Aarhus
Aaron
Ababa
aback
abacus
...
zombie
zoology
Zoroastrian
zucchini
Zurich
zygote
grep

grep -i -E '[aeiou]{4,}' /usr/dict/words
grep

grep -i -E '[aeiou]{4,}' /usr/dict/words

aqueous
Hawaiian
IEEE
obsequious
onomatopoeia
pharmacopoeia
prosopopoeia
queue
Sequoia
grep -i -E '(q[^u]|q$)' /usr/dict/words
grep

grep -i -E '(q[^u]|q$)' /usr/dict/words

CEQ
Colloq
IQ
Iraq
q
Qatar
QED
q’s
seq
grep

Is there a regular expression that matches those words whose letters appear in alphabetical order?

grep -x -E /usr/dict/words
Is there a regular expression that matches those words whose letters appear in alphabetical order?

`grep -x -E /usr/dict/words`

grep

Is there a regular expression that matches those words whose letters appear in alphabetical order?

grep -x -E /usr/dict/words

grep -x -E
/usr/dict/words

almost
begin
below
biopsy
dirty
empty
first
forty
ghost
glory

(Some of the longer words.)
Can we modify the previous example to allow double letters?

```bash
grep -x -E 'a*b*c*d*e*f*g*h*i*j*k*l*m*n*o*p*q*r*s*t*u*v*w*x*y*z*' /usr/dict/words
```

accent
access
almost
biopsy
chilly
choosy
effort
floppy
glossy
knotty

(Some of the longer words.)
grep -E -e '(y.*){3,}' /usr/dict/wordsA
grep

grep -E -e '(y.*){3,}' /usr/dict/wordsA

The words with at least three y's.

polytypy
psychophysiology
synonymy
syzygy  straight-line alignment of three celestial bodies
Back references

grep -E -e '(.())\1\1' /usr/dict/words
Back references

```
grep -E -e '(.\1\1)' /usr/dict/words
AAA
AAAS
IEEE
iii
viii
```
Perl supports the usual regular expressions:

\  Quote the next metacharacter
^  Match the beginning of the line
.  Match any character (except newline)
$  Match the end of the line (or before newline at the end)
|  Alternation
(..) Grouping
[..] Character class

*  Match 0 or more times
+  Match 1 or more times
?  Match 1 or 0 times
{n} Match exactly n times
{n,} Match at least n times
{n,m} Match at least n but not more than m times
PERL supports “reluctant” and “possessive” quantifiers in addition to the “greedy” quantifiers. PERL has additional boundary matchers, Unicode character class operators, and look-ahead and look-behind matchers. Some of these require exponential back-tracking.

\p{property} Match a character with Unicode property
\P{property} Match a character without Unicode property
(?=pattern) A zero-width positive look-ahead assertion
(?!pattern) A zero-width negative look-ahead assertion
(?<=pattern) A zero-width positive look-behind assertion
(?<!pattern) A zero-width negative look-behind assertion

For more information see the documentation at www.perldoc.com.
Java supports regular expression in the class `java.util.regex.Pattern`. It has character class operations for union and intersection.

```
[...]            Character class
[^...]          Complement
[...[...]]      Union
[...&/&[...]]   Intersection
[...&/&[^...]]  Substraction
```
// Greedy quantifiers
String match = find("A.*c", "AbcAbc");  // AbcAbc
match = find("A.+", "AbcAbc");  // AbcAbc

// Nongreedy quantifiers
match = find("A.*?c", "AbcAbc");  // Abc
match = find("A.+?", "AbcAbc");  // Ab

// Find first substring in input that matches pattern.
static String find (String pat, CharSequence input) {
    final Pattern pattern = Pattern.compile(pat);
    final Matcher matcher = pattern.matcher(input);
    if (matcher.find()) {
        return matcher.group();
    }
    return null;
}
Using regular expressions to match comments often runs into two problems:

- The “every character” pattern often does not match the characters indicating a newline.

  /* The first line, 
   but not the second line. */

- The greedy “star” will match more than what was intended.

  x = 1; /* Both comments */
  x = 2;
  /* at once! */ x = 3;

```java
final static String commentMultilineString = "/\*.*?\*/
final static Pattern multiLineCommentPattern = Pattern.compile (commentMultilineString , Pattern.DOTALL);
```
Regular expressions are used in the definition of programming languages to describe the basic lexical units–tokens.

1. **NUMBER**: \([0 - 9]+\)
2. **IDENTIFIER**: \([a - zA - Z][a - zA - Z0 - 9]^*\)
3. a keyword: *begin*
4. a hex number: \(0x[0 - 9A - F]+\)
5. **REAL**: \([0 - 9] +'.'[0 - 9] + (E[0 - 9]+)?\)

Metacharacters: \([\] +' −()]?\)
Tools

- regular expressions
- flex/lex
- lexical analyzer
- input stream
- token stream
Given a program

```c
float match0 (char *s) /* find a zero */
{ if (!strncmp (s, "0.0", 3))
    return 0.;
}
```

the lexical analyzer will return the stream

`FLOAT ID(MATCH0) LPAREN CHAR STAR ID(s) RPAREN LBRACE BANG ID(STRNCMP) LPAREN ID(s) COMMA STRING(0.0) COMMA NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF`
Lexical Tokens

Scanning or tokenization described in words:

An identifier is a sequence of letters and digits; the first character must be a letter. The underscore \_ counts as a letter. Upper- and lowercase letters are different. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters that could possibly constitute a token. Blanks, tabs, newlines, and comments are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.
Regular expressions

1. How do you recognize if a string matches a regular expression?
2. How do you tokenize a character string?

These questions are left to a compiler construction class.
Expressiveness of RE

We like RE because they describe strings. RE are great: easily understandable, concise, recognizers are easy to make, . . . But,
Expressiveness of RE

We like RE because they describe strings. RE are great: easily understandable, concise, recognizers are easy to make, ... But, The major shortcoming of regular expressions, as far as the description of programming languages is concerned, is that bracketing is not expressible. Bracketing is indispensable. For example, arithmetic expressions require opening and closing parentheses. Sequences of statements are often bracketed by begin/end pairs.

The set of strings

\[ \{a^n \cdot b^n\} = \{ \epsilon, ab, aabb, aaabbb, \ldots \} \]

is not regular!
Confex Free Grammars
Backus-Naur Form
John Backus proposed a metalanguage of “metalinguistic formulas” to describe syntax of the new programming language ALGOL. Apparently, Backus was familiar with Chomsky’s work.

Peter Naur called Backus’s notation *Backus normal form*. Soon afterward, Donald Knuth argued that BNF should rather be read as Backus-Naur form, as it is “not a normal form in the conventional sense” (CACM, 1964), unlike, for instance, Chomsky normal form.
In his book *Computing: A Human Activity* (1992), Naur rejected the formalist view of programming as a branch of mathematics. He did not like being associated with the Backus-Naur Form (a notation technique for context-free grammars attributed to him by Donald Knuth), and said he would prefer it to be called the Backus Normal Form.

Naur disliked the term “computer science,” suggesting it instead be called “datalogy” or “data science.” The term “dataology” has been adopted in Denmark and in Sweden as *datalogi*, while “data science” is used to refer to data analysis (as in statistics and databases).
BNF (Backus-Naur Form)

Conventions (e.g., tokens in bold font) and meta-symbols:

::= “is defined as”
| “or”
[] “optional” (not necessary, EBNF)
{} “zero or more” (not necessary, EBNF)

BNF is a collection of rules or productions. Each rule has a LHS and a RHS. The LHS is a syntactic category (a part of the language), called a nonterminal and the RHS is a sequence of nonterminals and tokens (called terminals).

\[
LHS_1 ::= RHS_1 \\
LHS_2 ::= RHS_2 \\
\vdots \\
LHS_n ::= RHS_n
\]
Examples and Derivations

Grammar 1 and derivation: “revolutionary ideas”
Example 3.1, page 129. Program with statement lists
Example 3.2, page 131. Simple assignment statements
Sebesta, seventh edition
Example Grammar

```
sentence ::= subject predicate .
subject ::= article { adjective } noun
predicate ::= verb [ adverb ]
article ::= A | The
adjective ::= green | colorless | big | new | revolutionary
noun ::= caterpillar | idea
verb ::= sleeps | appears | crawls
adverb ::= furiously | infrequently
```
**Derivation**

_Sentential form._ A *sentential form* is a string of terminal and nonterminal symbols.

_Derivation._ A *derivation* of a sentential form is created from a nonterminal by repeatedly replacing the LHS nonterminal of some rule with its RHS. Every BNF definition is a description of a formal language (a set of string), the language of all sentences derivable from a syntactic category.

We derive two sentences used as example by Noam Chomsky—one sentence is meaningful, the other makes no sense.
Example Derivation

sentence ⇒
1  subject predicate . ⇒
2  article adjective adjective noun predicate . ⇒
3  A adjective adjective noun predicate . ⇒
4  A new adjective noun predicate . ⇒
5  A new revolutionary noun predicate . ⇒
6  A new revolutionary idea predicate . ⇒
7  A new revolutionary idea verb adverb . ⇒
8  A new revolutionary idea appears adverb . ⇒
9  A new revolutionary idea appears infrequently .
Example Derivation

sentence ⇒
1  subject predicate . ⇒
2  article adjective adjective noun predicate . ⇒
3  A adjective adjective noun predicate . ⇒
4  A colorless adjective noun predicate . ⇒
5  A colorless green noun predicate . ⇒
6  A colorless green idea predicate . ⇒
7  A colorless green idea verb adverb . ⇒
8  A colorless green idea sleeps adverb . ⇒
9  A colorless green idea sleeps furiously .
Syntax versus Semantics

Two sentences (with the same structure, syntax) derived from the grammar.

A new revolutionary idea appears infrequently.
A colorless green idea sleeps furiously.

One sentence is meaningful, the other is meaningless. Clearly, the meaning of the words (semantics) is important.
Examples and Derivations

* Grammar 1 and derivation: “revolutionary ideas”
Example 3.1, page 112. Program with statement lists
Example 3.2, page 113. Simple assignment statements
Sebesta, fifth edition
Alternate Styles of BNF

“traditional” style:
⟨traditional style⟩ ::= key
Nonterminals in angle brackets, terminals in bold font.

CFG or “mathematical” style:
A → Ba
Nonterminals in uppercase letters, terminal in lower case letters.

“verbatim” style:
verbatim_style ::= "key"
ISO 14977 is much like verbatim style.

Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?” CACM, volume 20, number 11, November 1977, page 822.
In addition to linear forms of BNF there are also syntax graphs, also known as syntax diagrams. See Chapter 2, page 21 of Webber Figure 3.6, page 122 of Sebesta, fifth edition (not in sixth edition). See Hyper GOS syntax diagram generator.
if_statement ::= "if" condition "then" sequence_of_statements
{ "elsif" condition "then" sequence_of_statements }
[ "else" sequence_of_statements ] "end" "if" ";"
A grammar is a 4-tuple \( \langle T, N, P, S \rangle \)

- \( T \) is the set of terminal symbols,
- \( N \) set of nonterminal symbols, \( T \cap N = \emptyset \),
- \( S \), a nonterminal, is the start symbol,
- \( P \) are the productions of the grammar.

A production has the form \( \alpha \to \beta \) where \( \alpha \) and \( \beta \) are strings of terminals and nonterminals (\( \alpha \) can’t be the empty string). We are primarily interested in context-free grammars in which \( \alpha \) is a single nonterminal.
Example 1

Let $G$ be the grammar defined by the following productions.

$S \rightarrow A \ B$

$A \rightarrow \epsilon$

$A \rightarrow A \ a$

$B \rightarrow \epsilon$

$B \rightarrow B \ b$

(We infer easily that $T$, the set of terminals, is $\{a, b\}$; $N$, the set of nonterminals, is $\{A, B\}$; and $S$ is the start symbol.)

$L_G = \{a^i \ b^j\}$
Example 2

Let $G$ be the grammar defined by the following productions.

$$S \rightarrow \epsilon$$

$$S \rightarrow a \ S \ b$$

(We infer easily that $T$, the set of terminals, is \{a, b\}; $N$, the set of nonterminals, is \{S\}; and $S$ is the start symbol.)

$$L_G = \{a^i \ b^i\}$$
What’s a Grammar?

A grammar is a description of a language. Every grammar denotes a formal language (set of strings). A string of nonterminals $\omega$ is in the language, if it is derivable from the start symbol.

We say $\gamma \alpha \delta \Rightarrow \gamma \beta \delta$ (derives in one step) whenever $\alpha \rightarrow \beta$ is a production. We use $\alpha \Rightarrow^* \beta$ as the reflexive and transitive closure of the $\Rightarrow$ relation. A string $\omega$ of terminals is in the language defined by grammar $G$, if $S \Rightarrow^* \omega$. So, $L(G) = \{ \omega \in T^* \mid S \Rightarrow^* \omega \}$. In this case the string $\omega$ is called a sentence of $G$. If $S \Rightarrow^* \alpha$ where $\alpha$ may contain nonterminals, then we say $\alpha$ is a sentential form of the grammar $G$. 
**Chomsky Hierarchy**

<table>
<thead>
<tr>
<th>name</th>
<th>grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>unrestricted</td>
<td></td>
</tr>
<tr>
<td>context-sensitive</td>
<td>$</td>
</tr>
<tr>
<td>context-free</td>
<td>$\alpha \in \mathbb{N}$</td>
</tr>
<tr>
<td>regular</td>
<td>$A \rightarrow \omega B$, $A \rightarrow \omega$</td>
</tr>
</tbody>
</table>
Context-free grammar is the same as BNF.

\[
\begin{align*}
S & ::= AB \\
A & ::= \\
A & ::= Aa \quad \text{OR} \quad S ::= AB \\
B & ::= \\
B & ::= Bb
\end{align*}
\]
A formal definition of a parse tree for a context-free grammar is possible, but it is not illuminating. But the trees in the figures are suggestive. Each production used in the derivation of a string appears as a subtree in the diagram. The left-hand-side nonterminal appears as a node, and all the grammar symbols in the right-hand side of the production appear as children of this node.
An example from Sebesta, 7th, Example 3.2, page 131. An grammar for simple assignment statement

\[
\begin{align*}
assign &::= id := expr \\
expr &::= expr + expr \\
 &\quad |\quad expr \ast expr \\
 &\quad |\quad ( expr ) \\
 &\quad |\quad id \\
id &::= A | B | C
\end{align*}
\]
stmt ::= if-stmt | while-stmt | begin-stmt | asgn-stmt
if-stmt ::= if bool-exp then stmt else stmt
while-stmt ::= while bool-exp do stmt
begin-stmt ::= begin stmt-list end
asgn-stmt ::= variable := arith-exp
stmt-list ::= stmt | stmt ; stmt
bool-exp ::= arith-exp compre arith-exp
compare-op ::= < | <= | > | >= | =
arith-exp ::= variable | const | arith-exp op arith-exp
arith-op ::= + | * | − | /
variable ::= a | b | c | x | y | z
constant ::= 0 | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9
stmt ::= if-stmt | while-stmt | begin-stmt | asgn-stmt

if-stmt ::= "if" bool-expr "then" stmt "else" stmt;

while-stmt ::= "while" bool-expr "do" stmt;

begin-stmt ::= "begin" stmt-list "end";

asgn-stmt ::= variable "=" arith-expr;

stmt-list ::= stmt | <stmt> ";" <stmt-list>;

bool-expr ::= arith-expr compare-op arith-expr;

compare-op ::= "<" | "<=" | ">" | ">=" | ">" | ">=" |

arith-expr ::= variable | constant | arith-expr arith-op arith-expr;

arith-op ::= "+" | "-" | "*" | "/";

variable ::= "a" | "b" | "c" | "x" | "x" | "z";
Example Derivation

\[ \text{assign } \Rightarrow \]
Example Derivation

\[ assign \Rightarrow \]
\[ id := expr \Rightarrow \]
Example Derivation

\[
\text{assign } \Rightarrow \\
id := \text{expr } \Rightarrow \\
A := \text{expr } \Rightarrow
\]
Example Derivation

\[
\text{assign } \Rightarrow \\
\quad id := expr \Rightarrow \\
\quad A := expr \Rightarrow \\
\quad A := id \ast expr \Rightarrow
\]
Example Derivation

\[\text{assign} \Rightarrow\]
\[id := expr \Rightarrow\]
\[A := expr \Rightarrow\]
\[A := id \ast expr \Rightarrow\]
\[A := B \ast expr \Rightarrow\]
Example Derivation

\[\text{assign} \Rightarrow\]
\[\text{id} := \text{expr} \Rightarrow\]
\[\text{A} := \text{expr} \Rightarrow\]
\[\text{A} := \text{id} \ast \text{expr} \Rightarrow\]
\[\text{A} := \text{B} \ast \text{expr} \Rightarrow\]
\[\text{A} := \text{B} \ast (\text{expr}) \Rightarrow\]
Example Derivation

assign \Rightarrow

\begin{align*}
  id & := expr \Rightarrow \\
  A & := expr \Rightarrow \\
  A & := id \ast expr \Rightarrow \\
  A & := B \ast expr \Rightarrow \\
  A & := B \ast ( expr ) \Rightarrow \\
  A & := B \ast ( id + expr ) \Rightarrow
\end{align*}
Example Derivation

\[\text{assign} \Rightarrow\]
\[id := \text{expr} \Rightarrow\]
\[A := \text{expr} \Rightarrow\]
\[A := \text{id} \ast \text{expr} \Rightarrow\]
\[A := \text{B} \ast \text{expr} \Rightarrow\]
\[A := \text{B} \ast (\text{expr}) \Rightarrow\]
\[A := \text{B} \ast (\text{id} + \text{expr}) \Rightarrow\]
\[A := \text{B} \ast (\text{A} + \text{expr}) \Rightarrow\]
Example Derivation

\[ \text{assign} \Rightarrow \]
\[ id := \text{expr} \Rightarrow \]
\[ A := \text{expr} \Rightarrow \]
\[ A := id \times \text{expr} \Rightarrow \]
\[ A := B \times \text{expr} \Rightarrow \]
\[ A := B \times ( \text{expr} ) \Rightarrow \]
\[ A := B \times ( id + \text{expr} ) \Rightarrow \]
\[ A := B \times ( A + \text{expr} ) \Rightarrow \]
\[ A := B \times ( A + id ) \Rightarrow \]
Example Derivation

\[assign \Rightarrow\]
\[id := expr \Rightarrow\]
\[A := expr \Rightarrow\]
\[A := id \ast expr \Rightarrow\]
\[A := B \ast expr \Rightarrow\]
\[A := B \ast ( expr ) \Rightarrow\]
\[A := B \ast ( id + expr ) \Rightarrow\]
\[A := B \ast ( A + expr ) \Rightarrow\]
\[A := B \ast ( A + id ) \Rightarrow\]
\[A := B \ast ( A + C )\]
Parse Tree

Convert from PS Tricks. Concrete Parse tree??

\[ id := id * ( id + id ) \]

\[
\begin{array}{c}
* & \text{expr} \\
\end{array}
\]

\[
\begin{array}{c}
\text{expr} := \text{assign} \\
\end{array}
\]

\[
\begin{array}{c}
id \\
\end{array}
\]
Concrete parse trees may have a lot of extra stuff and may be inconvenient to use directly.
(Webber, Section 3.8 Abstract Syntax Trees, page 38.)
Example Derivation

\[
\begin{align*}
\text{assign} \Rightarrow \\
\text{id} &:= \text{expr} \Rightarrow \\
\text{A} &:= \text{expr} \Rightarrow \\
\text{A} &:= \text{id} \ast \text{expr} \Rightarrow \\
\text{A} &:= \text{B} \ast \text{expr} \Rightarrow \\
\text{A} &:= \text{B} \ast (\text{expr}) \Rightarrow \\
\text{A} &:= \text{B} \ast (\text{id} + \text{expr}) \Rightarrow \\
\text{A} &:= \text{B} \ast (\text{A} + \text{expr}) \Rightarrow \\
\text{A} &:= \text{B} \ast (\text{A} + \text{id}) \Rightarrow \\
\text{A} &:= \text{B} \ast (\text{A} + \text{C})
\end{align*}
\]
Concrete Parse Tree

```
assign
  /   
 id  =  expr
   /   
  A    

id  *  expr
  /   
 B    

(expr)
  /   
 id  +  expr
   /   
  A    

id
  /   
C    
```
Abstract Syntax Tree

A := B * (A + C).

expr can be replaced by the kind of expression +, *.

id contains no information. Parentheses are no longer needed.
If \( A := (B \times A) + C \) was the statement, then the abstract syntax would be different and so no important information would be lost.
Dangling else

An *ambiguous grammar* is one in which some sentence has more than one parse tree. [Sebesta, 3.3.1.7 Ambiguity, page 132ff.]

The grammar of Example 3.3 [Sebesta, 7th, page 132], simple expressions, is ambiguous.

Also, this grammar of if/then/else statements is ambiguous.

Sebesta, 7th, 3.3.1.10 An Unambiguous Grammar for if–then–else, page 137; Figure 3.5, page 138.
Two Parse Trees

[Convert from PS tricks.]
Equivalent Grammar

Just because a grammar is ambiguous does not mean that all grammars for the same language are ambiguous.

\[ S \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } S \text{ else } S \mid S' \]

This grammar for the same language is not.

\[
\begin{align*}
S & \rightarrow \ MS \mid \ UMS \\
MS & \rightarrow \text{if } C \text{ then } MS \text{ else } MS \mid S' \\
UMS & \rightarrow \text{if } C \text{ then } S \mid \text{if } C \text{ then } MS \text{ else } UMS
\end{align*}
\]
Alternative Language

If the “natural” grammar is ambiguous, could it mean that the construct is confusing to programmers? Is another approach possible?

Many languages have redesigned the syntax of the if statement:

\[ S \rightarrow \text{if } C \text{ then } S \text{ endif } | \]
\[ \text{if } C \text{ then } S \text{ else } S \text{ endif } | \]
\[ S' \]

The two statements with different structure now have different syntax.

if \( C_1 \) then if \( C_2 \) then \( S_1 \) endif else \( S_2 \) endif

if \( C_1 \) then if \( C_2 \) then \( S_1 \) else \( S_2 \) endif endif
Tools

BNF (LALR)

bison/yacc

token stream → parser → parse tree
with begin 2 * x;

with  comment  identifier  literal  begin
Recursive Descent Parsers

Grammars in the right form permit a parser of simple recursive functions. Each nonterminal turns into a recursive function forming a set of mutually recursive functions. Each function does a case analysis on the input to determine which production/RHS to follow. Terminals in the RHS are matched (consumed) and nonterminals turned into recursive calls.

Java Applet demo: http://cswebsrv.cs.binghamton.edu/~zdu/parsdemo/recframe.html
http://www.uni-paderborn.de/fachbereich/AG/agkastens/compiler/parsdemo/recframe.html
See example in Figure 2.10 in Scott.
Limitations of BNF/CFG

\[
\langle \text{block} \rangle ::= \langle \text{block identifier} \rangle : \begin{align*}
\text{begin} & \langle \text{statement} \rangle \{ \langle \text{statement} \rangle \} \\
\text{end} & \langle \text{block identifier} \rangle ;
\end{align*}
\]

\[
\langle \text{call} \rangle ::= \langle \text{identifier} \rangle (\{ \langle \text{expression} \rangle \})
\]
Beyond BNF/CFG

Formalisms with greater expressive power

- unrestricted grammars
- Post systems
- attribute grammars

Such formalisms are used in describing semantics — a later topic. Notice the pragmatic definition of syntax (at or below context-free) and semantics (beyond context-free).

The set of strings

$$\{a^n \cdot b^n \cdot c^n\} = \{\epsilon, abc, aabbc, aaabbbccc, \ldots\}$$

is not context free!
programming languages

formal languages

compilers

expressivity

implementation, recognition

language design,
description, semantics

programmed languages