This “lecture” is intended as a kind of road map to Chapter 1 of the text—just the informal examples that I’ll present to motivate the ideas.

1 Expressions without Parentheses

In the last set of notes we looked at algebraic expressions and removed everything but the parentheses. We asked how we could tell whether a string of parentheses was correctly nested.

Here we’ll remove the parentheses and ask how we can determine if the resulting expression is correctly formed. That is, we’ll look at expressions like

345-2789-27+675+43-2

Let’s suppose for now that the only operation symbols that appear are + and −, and that the terms are all unsigned decimal integers.

2 A State-transition Graph for the Problem

We can draw a kind of flow chart for an algorithm that determines whether such an expression.
The incoming arrow at the left-hand node means that the algorithm starts in this ‘state’. The double circle at the right-hand node means the input is accepted if the algorithm is in this state when it finishes scanning the input string. If the algorithm is in the other state when the end of the input is reached, the input string is rejected.

Here are two pseudocode versions of the algorithm. In the first one, the state is literally a place in the program code:

```plaintext
state0:
if there is no more input,
reject and terminate;
else if the next input symbol is a digit,
go to state1;

state1:
if there is no more input
accept and terminate;
else if the next input symbol is a digit,
go to state1;
else if the next input symbol is + or -,
go to state0;
```

In the second, the variable representing the state plays an analogous role to the variable excess in the parenthesis-testing algorithm.

```plaintext
state=0;
while there is still more input
  if next input symbol is a digit
    state=1;
  else if next input symbol is + or - and (state==1)
    state=0;
  if(state=1)
    accept
  else
    reject
```

These reflect two different interpretations of what is meant by the state of a computer program, an issue that will be important to us when we try to describe formally exactly what an algorithm is.

### 3 DFA’s

But I digress. The gizmo illustrated in the diagram above, and realized in the pseudocode implementations, is called a deterministic finite automaton. “Au-
tomaton” is an old word, traditionally used to mean something along the lines of “robot”. The plural of *automaton* is *automata*. Two relatively common words of English that form their plurals in the same way are *phenomenon* and *criterion*. People who should know better habitually misuse all of these words and used the plural form for both the singular and the plural. I can get fussy about this sort of thing, but I know that I’m fighting a losing battle.

But I digress. The word “deterministic” in the definition means that when you know the present state and the next input, then the next state is completely determined.

Strictly speaking, the commonly-used definition of DFA requires that the state transition function of the automaton be completely specified: that is, for every choice of state and input symbol, there is a next state defined. The diagram above violates this condition, so we have to fix things up by adding a new state:

The set of strings accepted by a DFA is called a *regular language*.

## 4 NFA’s

Let’s suppose we wanted to an algorithm for recognizing the set of expressions in which the last operation symbol is + rather than −. We can redraw our state-transition graph to obtain:
Now there’s something really fishy going on—we’ve lost the determinism, since when we’re in state 1 and the next input symbol is + we’re allowed to guess whether or not this is the last operation symbol and change state accordingly. It’s as though you were allowed to rewrite the pseudocode as

```plaintext
state=0;
while there is still more input
    if next input symbol is a digit and (state==0) or (state==1)
        state=1;
    else if next input symbol is a digit and (state==2) or (state==3)
        state=3;
    else if next input symbol is - and (state==1)
        state=0;
    else if next symbol is + and (state==1)
        either
            state=0;
        or
            state=2;
    if(state==3)
        accept
    else
        reject
```

The either-or statement is not a standard feature in programming languages, for obvious reasons! The nondeterministic finite automaton pictured above accepts its input as long as there is some sequence of guesses that leads to an accepting state at the end of the input.

Nondeterministic algorithms don’t look like very reasonable things to be
talking about, yet the concept is surprisingly useful. In the case of finite automaton, nondeterminism is useful because (a) it’s often much easier to devise a nondeterministic “solution” for a problem; and (b) it’s always possible to eliminate nondeterminism:

**Theorem 1** The set of strings accepted by an NFA is a regular language. (That is, given an NFA, there is a DFA that accepts exactly the same set of strings.)

## 5 Regular Expressions

Here is a very different kind of description of the set of correct expressions. Remember that when we discussed correct bracketing, we gave two rather different descriptions: One that told how to generate correct patterns, and the other that told how to test a string for correctness.

For the expressions under consideration here, we’ve already described how to test them, here is a description of how to generate them:

- A **digit** is one of the symbols 0,1,2,3,4,5,6,7,8,9.
- A **number** is a sequence of 1 or more digits.
- An **operator** is one of the symbols + or -.
- An expression is a number, followed by a sequence of 0 or more occurrences of the pattern: operator followed by a number.

This description builds sets of strings by starting with the simplest pattern—an individual symbol—and combining them using the operations ‘or’, ‘followed by’ and ‘a sequence 0 or more occurrences of’. We use symbols \( \cup \), \( \cdot \), and \( \ast \) to denote these operations. For instance, we would denote a digit by

\[
(0 \cup 1 \cup \cdots \cup 9)
\]

a sequence of 0 or more digits by

\[
(0 \cup 1 \cup \cdots \cup 9)^\ast
\]

and a sequence of one or more digits by

\[
(0 \cup 1 \cup \cdots \cup 9)(0 \cup 1 \cup \cdots \cup 9)^\ast.
\]

We denote the complete pattern by

\[
(0 \cup 1 \cup \cdots \cup 9)(0 \cup 1 \cup \cdots \cup 9)^\ast((- + -)(0 \cup 1 \cup \cdots \cup 9)(0 \cup 1 \cup \cdots \cup 9)^\ast)^\ast.
\]

Pattern specifications built using these three operations are called *regular expressions*. The regular expression denotes the set of all strings that match the specified pattern. The crucial fact about these is:
Theorem 2 A language is denoted by a regular expression if and only if it is a regular language.

The practical import is this: There is a lot of software out there that lets you search for patterns in text where you specify the pattern by a regular expression. In many cases such programs build a finite automaton from the input regular expression, and then simulate the finite automaton scanning the text. As we shall see in class, it is much easier to build an NFA from a regular expression than a DFA. The NFA can then either be converted into a DFA or simulated directly.

I will have you play around with the UNIX utility grep which allows you to search for instances of regular expressions in a file.

6 Nonregular Languages

Is the language $P$ of correct arrangements of brackets a regular language? The following intuition suggests that it is not: When we scan a string from left to right, we need to keep track of the excess of left brackets over right brackets. The set of possible values for this excess is infinite, so we cannot do this with a finite set of state values.

Here is a formal proof that $P$ is not a regular language. Suppose that it is, and suppose that it is accepted by a DFA with $n$ states. Now let us read the string $a^n$ ($n$ occurrences of $a$) starting from the initial state $q_0$ of the automaton, and look at the sequence of states we encounter:

$q_0, q_1, \ldots, q_n$.

There are $n+1$ states on this list, but only $n$ distinct states in the automaton, so two of these states must be the same: That is $q_i = q_j$ for some $i < j$. Now $q_i$ is the state we reach when we have read $i$ occurrences of $a$, so if we now proceed to read $i$ $b$'s we will wind up in an accepting state $q_{acc}$ of the automaton, since $a^i b^i \in P$. But $q_i$ is also the state we reach when we have read $j$ occurrences of $a$, and thus we wind up in an accepting state when we read $a^j b^i$. But then our DFA cannot accept the language $P$, because $a^j b^i \notin P$, a contradiction. (Another way to state this argument is that any regular language that contains all the strings $a^i b^j$ must also contain a string of the form $a^j b^i$ where $j > i$, so $P$ cannot be a regular language.)

In class we will formalize this reasoning in a general theorem called the “pumping lemma” that allows one to prove that a great many languages are non-regular.