Previously:

Regular languages and the models that love them:
- Deterministic finite automata
- Nondeterministic finite automata (with ε-transitions?)
- Regular expressions!

Today:

Proving the equivalence of regular expressions and finite automata

Fun:

- Assignment 1 late deadline
  - Solutions will be posted tonight
  - Assignment 2 is out
  - And there’s now an optional LaTeX template for it.

Lesson: Think before you promise.
UNIX regular expressions

From the beginning (of time), UNIX has used regular expressions in many places, including the `grep` command.

`grep` = global (search for a) regular expression and print

Many UNIX commands use an extended RE notation, but it still expresses only the regular languages.

UNIX RE notation

`[a1a2…an]` is shorthand for `a1 ∪ a2 ∪ · · · ∪ an`.

Ranges are indicated by first-dash-last and brackets, using ASCII character order, e.g.,

`[a-z]` = any lowercase letter
`[a-zA-Z]` = any letter

Dot (.) = any character
UNIX RE notation, continued

Since characters like brackets, dashes, and dots have special meaning, if you want to match them, you need to quote with backslash (\).

Union operator is represented with a bar (|)

Includes our + shorthand for “one or more”, e.g.,

\[a-z]+ = one or more lowercase letter

Perl, Python, Emacs, …

Include additional extensions, notably character classes like \b for word boundary characters, \w for word characters, etc.

With each implementation of regular expressions, they become less standard, so what you write for one language or application won’t work in another.

Lexical analysis

The first thing a compiler does is break a program into tokens, which are substrings that together represent a unit, e.g.,

- identifiers
  - reserved words like “if”
  - meaningful single characters like “;” or “+”
- multicharacter operators like “<=”
Lexical analysis, continued

There are tools like lex or flex that let you write a regular expression for each kind of token.

E.g., in UNIX notation, identifiers are something like `[A-Za-z][A-Za-z0-9_]`

Each RE has an associated action like returning a code for the token found or adding it to a symbol table.

Equivalence of FA languages and RE languages

*Kleene’s Theorem:* Regular expressions and finite automata have equivalent descriptive power.

The languages recognized by DFAs, NFAs, and NFA-εs, and described by REs are exactly the regular languages.
Proof

We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton

Cover the six cases in the formal (recursive) definition of REs.

Base cases:

1. \( R = a \)

2. \( R = \epsilon \)

3. \( R = \emptyset \)

4. \( R = (R_1 \cup R_2) \)

The class of regular languages is closed under the union operation.

For languages represented by \( R_1 \) and \( R_2 \), take their NFAs \( N_1 \) and \( N_2 \) and combine them into one new NFA \( N \).

\( N \) must accept input if either \( N_1 \) or \( N_2 \) accepts input.

The new machine guesses non-deterministically which of the two machines accepts the input.
5. \( R = (R_1 \cdot R_2) \)

The class of regular languages is closed under the concatenation operation.

For languages represented by \( R_1 \) and \( R_2 \), take their NFA\( s \) \( N_1 \) and \( N_2 \) and combine them sequentially into one new NFA \( N \).

\[ N = N_1 \cdot N_2 \]

The new machine guesses non-deterministically where to split the input in order to have a first part accepted by \( N_1 \) and a second part accepted by \( N_2 \).

6. \( R = (R_1)^* \)

The class of regular languages is closed under the star operation.

For a language represented by \( R_1 \), modify \( N_1 \) to accept \( (R_1)^* \).

Example

\((abua)^*\)

Solution from Sipser

This conversion isn’t just theoretical!

Many tools that use regular expressions run them by converting them to finite automata that they can execute.

E.g., since the 1970s, `grep` has converted each regular expression into a finite automaton that it runs to do the search.
Proof

We will prove this set of equivalences by

☑ Showing how to construct a DFA from an NFA-ε. (Already done!)
☑ Showing how to construct an NFA-ε from a regular expression
☐ Showing how to construct a regular expression from a finite automaton

Proof

Constructing a regular expression from a DFA

DFA-to-RE by state elimination

Basic idea:

For each state s:
- Eliminate s (i.e., remove all arcs into and out of s)
- Label arcs from q to p that went through s with a regular expression representing the sequence of symbols on that path.

Procedure to remove a state

Before

After
We haven’t said that DFAs or NFAs can have regular expressions on arcs in their transition diagrams – and they can’t, really.

As an intermediate model in this construction, we’re using a generalized nondeterministic finite automaton (GNFA).

A GNFA is an NFA where the transition arrows can have a regular expression as the label.

It can read blocks of symbols from the input rather than just one symbol at a time.

We can simplify things if we ensure the following before applying the procedure for state elimination:

- There is a single final state
- There are no transitions into the initial state
- There are no transitions out of the final state

Since the procedure works on NFA-εs, this is easy to do:

General process

1. Add a new start state $q_s$ and accept state $q_f$.

   Add $\varepsilon$-transitions from $q_s$ to the original start state and from each original accept state to $q_f$. Mark the original accept states as not accepting.

2. Repeatedly remove states other than $q_s$ and $q_f$ from the NFA by “shortcutting” them until only $q_s$ and $q_f$ remain.

3. The transition from $q_s$ to $q_f$ is then a regular expression for the original NFA.

Example 1
Technically, this is an NFA and we want to convert from DFAs, but this one works out easily anyway.
We will prove this set of equivalences by

- Showing how to construct a DFA from an NFA-ε. (Already done!)
- Showing how to construct an NFA-ε from a regular expression
- Showing how to construct a regular expression from a finite automaton
The regular languages are recognized by NFA-$\varepsilon$s, NFAs, DFAs, and REs (…and GNFAs)

What about languages that aren’t regular languages?
Next time!

Acknowledgments
This lecture incorporates material from:
Nancy Ide
Keith Schwarz
Michael Sipser
Jeffrey Ullman