Theory of Strings

Robert M. Keller
Harvey Mudd College
2 September 2003

Why are we doing this?

- Strings are **fundamental** to most formal systems (e.g. grammars).
- Strings **generalize** natural numbers (which are themselves very important) in a very natural way.
- Strings are **easy** to work with compared to set-theoretic definitions of natural numbers.

Informal Definition of Strings

- A string is just a sequence $x_1x_2....x_n$ where $n \geq 0$.
- The elements of a string are called "letters" or "symbols", but they could be members of any set.
- While infinite strings are sometimes used, for now we will be working only with finite strings.

Formal Definition of Strings

- A string over the set of letters □ is any member of the following inductively defined set, named □* :
  - The empty string, □, is in □*.
  - If a string x is in □*, and □ is in □, then □x is in □*.
  - The only elements of □* are those obtained by applying the above rules.
- By □x we mean a combination of □ with x in such a way that both can be recovered from the result. This can be informally thought of as the "followed by" operator (similar to [□ | x] in the rex language).

Notes on the Empty String

- The empty string symbol □ (upper-case lambda) is a "meta symbol" and as such is never an ordinary letter in □.
- Other symbols are often seen in place of □ :
  - □ (lower-case lambda) is used in CS 60.
  - □ (lower-case epsilon) used in some texts.

Notes on □x as an ordered-pair

- Effectively this designates an "ordered-pair" of □ with x, which could be written more verbosely as (□, x).
- It is a pair because two things are combined.
- It is ordered because (□, x) is not the same thing as (x, □). (In fact, the latter is not meaningful in the current context.)

Ordered-Pairs as Sets

- There are ways of defining the ordered pair concept from more basic, set-theoretic, operations.
- A standard one is that **every pair (x, y) abbreviates a set {{x}, {x, y}}**.
- The key thing here is the assurance of the property:

  $(x, y) = (x', y')$ implies $x = x'$ and $y = y'$

  i.e. that pairs are **decomposable** into their parts.
- This can be proved.

Proof that $(x, y) = (x', y')$ implies $x = x'$ and $y = y'$

- Assume $(x, y) = (x', y')$.
- So $\{\{x\}, \{x, y\}\} = \{\{x'\}, \{x', y'\}\}$.
- If $x = y$, then the LHS is really the same as $\{\{x\}\}$, a set of one element. So the RHS must also be a set of one element, which is to say that $\{x'\} = \{x', y'\}$. But this can be true only if $x' = y'$. We have $\{\{x\}\} = \{\{x'\}\}$. Since these are sets of one element, the elements must be equal, so $\{x\} = \{x'\}$. By the same argument, $x = x'$. Combining with $x = y$ and $x' = y'$, we see that $y = y'$.
- If $x \neq y$, then the LHS is a set of two elements, and therefore the RHS is also. But $\{\{x'\}, \{x', y'\}\}$ is a set of two elements only if $x' \neq y'$. Since $\{x, y\}$ and $\{x', y'\}$ are now known to have two elements, while $\{x\}$ and $\{x'\}$ have one, we conclude that $x = x'$. Then we are led to conclude that $y = y'$, for if not, the two sets of two elements could not be equal, so neither could the original two sets.

Something to Think About

- In our previous argument about pairing, we appealed to certain ideas about sets.
- How can we capture those ideas succinctly as axioms, so that we can articulate exactly what we are using about sets?

Notes on □ and pairing

- □ is not a pair (□ is the only such string with this property).
- □ could be equated to the empty set.
- Example: The string abc is constructed as pairs
  (a, (b, (c, □))).
- As sets, this would be:

  {{a}, {a, {{b}, {b, {{c}, {c, {}}}}}}}

- Now you can see why we prefer the string notation.

String Concatenation

- Concatenation means “chaining together”, i.e. following one string by another.
- Concatenation will be shown by juxtaposition:
  xy is y concatenated to x.
- Some texts take concatenation as a primitive, given, operation.

String Concatenation

- Concatenation is a function or binary operator on □* and is defined inductively:
  - □y = y — basis
  - (□x)y = □(xy) — induction step
- On the left-hand sides above is the thing we are defining, and on the right how we are defining it.
- This can be seen as analogous to defining *append* in rex.

Why define by induction?

- Definition by induction is **precise**, compared to other alternatives.
- Definition by induction enables **proof** by induction.

String Axioms

- These axioms characterize strings, analogously to the way in which the Peano axioms characterize the natural numbers.
- In the following x, y, ... are implicitly quantified over strings
- □, □′, ... are implicitly quantified over letters.

String Axioms

- SA1: (□x) (Either x = □ or (□□)(□y) x = □y)
- SA2: (□x) (□□) □x ≠ □
- SA3: (□x, x′) (□□, □′)
  □x = □′x′ implies □ = □′ and x = x′
- SA4: Let P be any predicate on □*.

basis | induction step

$$\frac{P(□), \quad (□x)(P(x)\ □\ (□□)P(□x))}{(□x)\ P(x)}$$

infer

- SA4 is a "rule of inference", allowing to prove properties by induction.

Example of Inductive Proof

- ST1: (□x) x□ = x, in other words, □ concatenated to any string is just that string.
- For proof, we identify P(x) in SA4 with x□ = x.
- SA4 says that ST1 follows if we can show two things:
  - P(□), i.e. □ □ = □. basis
  - (□x)(P(x) □ (□□)P(□x)),
    i.e. (□x)(x□ = x □ (□□) (□x)□ = □x) induction step
- How are these two statements shown?

What we don’t need to prove

- ST1’: (□x) □x = x

  This looks very similar to ST1, but we don’t need to prove it. Why?

Another Example of Inductive Proof

- ST2: (□x, y, z) x(yz) = (xy)z. In other words, concatenation is associative.
- Here we identify P(x) in SA4 with x(yz) = (xy)z.
- SA4 says that ST2 follows if we can show two things:
  - P(□), i.e. □ (yz) = (□y) z. **basis**
  - (□x)(P(x) □ (□□)P(□x)),

    i.e. (□x)(x(yz) = (xy)z □ (□□) (□x)(yz) = ((□x)y)z **induction step**
- How are these two statements shown?

Defining String Reversal

- We want to define inductively reversal (^{R} operator)
- Here's an intuitively-correct idea:
  - $□^R = □$ basis
  - $(□x)^R = x^R(□□)$ induction step
- Note: We have to use □□ rather than just □ because $x^R□$ is not defined. But since □□ is a string, we can concatenate.
- Note that we are using concatenation to define this operator. This could be considered somewhat heavy-handed.

Defining String Reversal

- Another option would be to use an auxiliary function r:
  - $x^R = r(x, □)$, where
    - $r(□, y) = y$ basis
    - $r(□x, y) = r(x, □y)$ induction step
- Here we haven't used any overt concatenation, and this makes us feel better.
- Note that this is the familiar rex definition in disguise.
- We could now try to prove that the two versions of reversal are equivalent. We'll table this.

Proving stuff about reversal (^{R} operator)

- ST3: $(xy)^R = y^R x^R$
- Try induction with P(x): $(\square y)(xy)^R = y^R x^R$
- Basis: P($\square$): $(\square y)(\square y)^R = y^R \square^R$
- Induction step: $(\square y)(xy)^R = y^R x^R$

  implies $(\square\square)(\square y)((\square x)y)^R = y^R (\square x)^R$
- LHS of = is: $((\square x)y)^R$

  $= (\square(xy))^R$

  $= (xy)^R (\square\square)$

  $= (y^R x^R) (\square\square)$
- RHS of = is: $y^R (\square x)^R$

  $= y^R (x^R(\square\square))$

  $= (y^R x^R) (\square\square)$

Justify each step

Free the Monoids!

- Algebraically, □* is known as the
  "free monoid generated by □".
  (usually serves to deter the casual reader from going any further)
- Recall that a monoid is a set together with:
  - An associative operator
  - An identity for the operator
- Identify the components that justify calling □* a monoid.

Natural Numbers

- Pick □ to be any 1-letter alphabet. (e.g. use ∅ as a letter).
- Then □* is essentially the set of natural numbers:
  - □ is like 0
  - □x is like x+1
- The string axioms are then equivalent to the Peano axioms.
- The induction rule is the usual mathematical induction principle.

Natural Numbers

- Addition is just concatenation over □*.
- What are subtraction, multiplication, etc.?
- You’d need to define them inductively: more on this later.

Natural Numbers from Zip

- 0 is ∅ — basis
- n+1 is n □ {n} — induction step

for example

- 1 is ∅ □ {∅} which is {∅} which has 1 element
- 2 is {∅} □ {{∅}} which is {∅, {∅}} which has 2 elements
- 3 is {∅, {∅}} □ {{∅, {∅}}} which is {∅, {∅}, {∅, {∅}}} which has 3 elements, etc.
- ...

- Let □ designate the set of natural numbers.

The Length | | of a String

- |□| = 0 — basis
- | □x | = x+1 — induction step

Some Properties of Length

- L1: $|xy| = |x| + |y|$
- L2: $|x^R| = |x|$

Theory of Languages

Robert M. Keller
Harvey Mudd College
2 September 2003

Definition of the Concept "Language"

- A **language** over an alphabet Σ is any subset of Σ^*.
- Give some precise examples of languages.

Operations on Languages

- Since languages are sets, we can define their **union**, **intersection**, etc. just as with any sets, e.g.
- Let L and M be two languages.

$$L \cup M = \{x \mid x \in L \text{ or } x \in M\}$$
$$L \cap M = \{x \mid x \in L \text{ and } x \in M\}$$
$$L - M = \{x \mid x \in L \text{ and } x \notin M\}$$

Product of Languages

- Let L and M be two languages. Define

  $LM = \{xy \mid x \in L, y \in M\}$

  called the "product" or
  (loosely) the "concatention" of languages.

- Give examples.

- What if either is $\varnothing$?

Power Operator for Languages

- L be a language. Define the "n^{th} power" of L inductively:

  $L^0 = \{\epsilon\}$

  $L^{n+1} = L\,L^n$

- Examples?

Plus and Star Operators for Languages

- L be a language. Define

  $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$

- Define

  $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$

- Thus

  $L^* = \{\epsilon\} \cup L^+$

- Give examples.

L^* vs. Σ^*

- They are the same, provided that L is the set of 1-letter strings, one for each letter in Σ.

Language Identities: Devise RHS's

- $L\varnothing$ =
- L{□} =
- (LM)N =
- LL^* =
- LL^+ =
- {□}* =
- $\{$□$\}^+$ =
- $\varnothing^*$ =
- $\varnothing^+$ =
- (L □ M)N =
- (L □ M*)* =
- $(LM^*)^*$ =

Solving a Language Equation

- This will be seen to be a useful device shortly:
- The equation L = LA □ B has as a solution for L:

  $L = BA^*$

- How can we justify this?

Regular Operators and Languages

- Union, Star, and Product (Concatenation) are called the **Regular Operators** on Languages.
- A language is **regular** if it can be formed from languages consisting of only one string of one letter each using a **finite** number of regular operators.
- Note: * counts as only one operator, despite it being defined as an infinite union.
- Examples of Regular Languages?

True or False?

- Any language of exactly one element is regular.
- Any finite language is regular.
- Σ^* - L, where L is finite, is regular.
- Every language is regular. To see this, let $L = \{x_1, x_2, x_3, \ldots\}$.

  Then $L = \{x_1\} \cup \{x_2\} \cup \{x\}_3 \cup \ldots$, which is clearly regular.

Regular Expressions

- A regular expression is a **shorthand** way of representing regular languages using regular operator symbols in conjunction with the following symbols.
- Each letter □ in □ stands for the language with just one string of one letter, that letter.
- □ stands for the language {□}.
- ∅ stands for the empty language ∅.
- Example: If □ = {0, 1}, then 0 stands for the language with just one string, having one letter, 0.

Examples of Regular Expressions

- 0 □ 1
- (0 □ 1)*
- (0 □ 1)0*1*
- ((0 □ 1)0*1)*
- ((□ □ 1)0*1)*
- 0*110* □ 1*001*

Regular Expression Notation Notes

- Instead of □, some sources use infix + or | in regular expressions.
- * binds the tightest,
  then concatenation,
  then □.
- □ is not a regular operator, nor is -.

Regular Expressions as Patterns

- Any language can be equated to a “pattern”, namely the pattern that matches all strings in the language.
- Examples:
  - 0* is the pattern that matches strings containing only 0’s
  - 0*10* is the pattern that matches strings in {0, 1}* containing exactly one 1.
  - 0*100* is the pattern that ...
  - 0(0 □ 1)*1
  - ((0 □ 1) (0 □ 1))*
  - (0 □ □) (10)* (1 □ □)
- Note: To qualify as a pattern, the language of the expression must be that of **exactly** the set of strings matching the pattern, not a subset or superset.

Regular Expressions as Patterns

- Give regular expressions for the following patterns over {0, 1}:
  - Strings in which each 1 is followed by a 0.
  - Strings in which no 1 is followed by a 0.
  - Strings in which every 1 is preceded by and followed by a 0.
  - Strings in which the number of 1's is divisible by 3.
  - Strings in which there is no run of 3 consecutive 1's.

Application: Searchers

- Do

  man egrep

  on a UNIX system.
- How do such search algorithms work?

Finite-State Automata

Robert M. Keller
Harvey Mudd College
2 September 2003

Automata

- Colloquially, an **automaton** (plural "automata") is an autonomous device (such as a robot or wind-up toy).
- In CS, the term has a more specific meaning: that of an abstract **mathematical machine** that can perform a specific function.

Uses of Automata

- There are many uses, one of which is to specify algorithms for accepting languages.
- An automaton **accepts a language** if it can tell, for any given input string, whether or not the string is in the language.

Example: Compilers, etc.

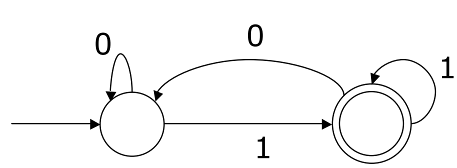
- Every compiler contains an automaton, that tells whether or not the input string is well-formed, i.e. is in the language that it compiles.
- Every pattern search program is effectively an automaton for recognizing patterns.

Finite-State Automata (FSA or DFA, they are the same)

- An automaton is finite-state if its behavior is representable by transitions between a states in a finite set, some of which are designated accepting and others not.
- Each automaton has a designated start state.

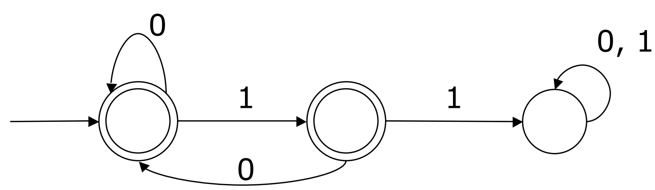
Examples of FSA

- An FSA capable of accepting exactly the strings ending with 1.



Examples of FSA

- An FSA capable of accepting exactly the strings containing no two consecutive 1's.



Thing to Check

- For each combination of a state and a symbol, there should be exactly one arrow leaving the state with that symbol.

- This is the "deterministic" ("D") in DFA.

- If this property does not hold, better fix it; your automaton might be wrong.

Application

- One way to implement a search is to construct, perhaps on the fly, an automaton that accepts the corresponding language, then simulate the automaton on the given input.

Two Ways to Define Specific Languages

- Give an FSA that accepts the language.
- Give a regular expression for the language.

Remarkable Fact

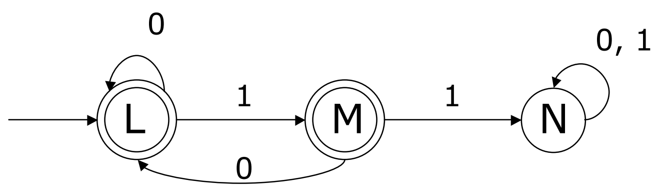
- The preceding two ways are equivalent.
- Equivalent here means that the two methods define the same family of languages.

Application of this Theory

- Sometimes it's easier to give an automaton for a language.
- Sometimes it's easier to give a regular expression.
- It would be nice to be able to go from one to the other more-or-less freely.

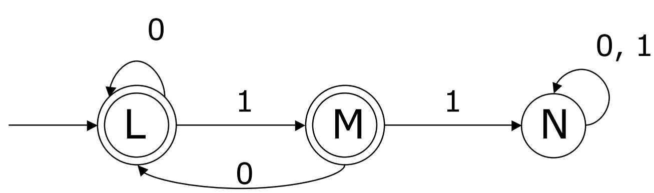
Regular Expression from FSA

- Label the States



- Identify each state with the set of paths from the start state to it. This set is a language.
- The language accepted by the FSA is the **union** of the paths to each of the accepting states, in this case L ∪ M.

Deriving Closed Forms



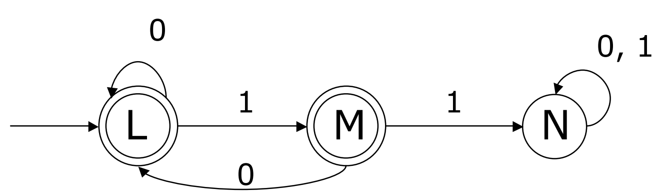
- View the acceptor as a set of **regular-expression equations**:
  - L = L0 ∪ M0 ∪ ε
  - M = L1
  - N = M1 ∪ N(0 ∪ 1)
- The ε is on the RHS of the starting state only.
- We want to solve for L and M, and take the union of the solutions.

Solving RE Equations

- **Solve** for L and M:
  - L = L0 □ M0 □ □
  - M = L1
  - N = M1 □ N(0 □1)
- **Substitution** Operation:
  - A LHS variable can be replaced with its RHS, so replacing M in the L equation:
  - L = L0 □ L10 □ □, or more simply
  - L = L(0 □ 10) □ □
- **Elimination** Operation:
  - An equation of the form L = LA □ B has the solution L = BA*, so:
  - L = □(0 □ 10)*, or more simply L = (0 □ 10)*
- Substitution again:
  - M = L1
  - M = (0 □ 10)*1

Conclusion

- The language accepted by the FSA below is
  - L □ M
  - which is (0 □ 10)* □ (0 □ 10)*1
  - or more simply
  - (0 □ 10)*(□ □ 1)



FSA □ RE Algorithm

- Express the FSA as a set of RE equations
  - Each state is a variable.
  - Each variable is equated to a union of expressions showing how to get to that state in one step from other states.
  - The start state has □ on the RHS as well.
- Solve the RE equations for the variables:
  - The variables, along with their equations, are solved for one at a time.
  - Choose a variable for elimination.
  - Expression that variable in terms of the remaining variables only, using the * operator (L = LA □ B has the solution L = BA*).
  - Substitute the solution for all occurrences of the variable in the remaining equations.
  - Repeat the above steps until no variables remain.
- Work backward, substituting the solutions found for other variables, until each variable is expressed in closed form.

Another Example

- **Solve**:
  - L = L1 □ M0 □ N0 □ □
  - M = L0 □ M1 □ N1
  - N = L1 □ M1 □ N0

- Note that these equations don't really correspond to a DFA, but it doesn't matter.
- Eliminate N, using N = (L1 □ M1)0*
  - L = L1 □ M0 □ (L1 □ M1)0*0 □ □
  - M = L0 □ M1 □ (L1 □ M1)0*1
- Regroup:
  - L = L(1 □ 10*0) □ M(0 □ 10*0) □ □
  - M = L(0 □ 10*1) □ M(1 □ 10*1)

Solution, continued

- Solving:
  - L = L(1 □ 10*0) □ M(0 □ 10*0) □ □
  - M = L(0 □ 10*1) □ M(1 □ 10*1)
- Eliminate M using M = L(0 □ 10*1) (1 □ 10*1), giving:
  - L = L(1 □ 10*0) □ L(0 □ 10*1) (1 □ 10*1)(0 □ 10*0) □ □
- Regrouping:
  - L = L((1 □ 10*0) □ (0 □ 10*1) (1 □ 10*1)(0 □ 10*0)) □ □
- Solving:
  - L = ((1 □ 10*0) □ (0 □ 10*1) (1 □ 10*1) (0 □ 10*0))*
- Working backward:
  - M = ((1 □ 10*0) □ (0 □ 10*1) (1 □ 10*1) (0 □ 10*0))*
    (0 □ 10*1) (1 □ 10*1)
  - N = (L1 □ M1)0* = ...

Summary so far

- The language accepted by an FSA is a regular language.
- We haven't shown that the converse is true.