This Lecture (1)

- We have seen three ways of formally describing potentially infinite languages:
  - Deterministic Finite Automata (DFA)
  - Nondeterministic Finite Automata (NFA)
  - Regular Expressions (RE)
- Because
  - a DFA is a special case of an NFA
  - any NFA can be converted into an equivalent DFA
DFAs and NFAs describe the same class of languages: the Regular languages.

This Lecture (2)

So, what class of languages do the REs describe? Smaller? Larger? Completely different?
In fact:

- Regular Expressions describe the Regular Languages
- Proof: interconversion between RE and FA
- This lecture: conversion of RE to NFA
Will start by a motivating example. Time permitting, brief look at another application: scanners. Study details in your own time if of interest.

Applications (1)

RE to NFA conversion has important practical applications.
The following is a very nice, practically oriented article you should be able to fully appreciate based on what you have learned in G52MAL thus far:

http://swtch.com/~rsc/regexp/regexp1.html

Applications (2)

Underlying message: if you’re ignorant about CS theory, your code can perform really poorly.
Example from the paper:

![Graph showing time to match (a + ε)a^n against a^n](image)

Note difference of time scale: 60 s vs. 60 µs!

Applications (3)

To quantify:
- Thompson NFA implementation a million times faster than Perl when running on a 29-character string.
- Thompson NFA handles a 100-character string in under 200 microseconds; Perl would require over $10^{15}$ years.

How old is the universe?
Current best estimate: 13.8 billion years ... or about $10^{10}$ years. $10^{15}$ years is a loooong time ...
Converting RE to NFA (2)

**Specification:**
Let $N(E)$ denote the NFA that results by applying the graphical construction to an RE $E$. Then the following equation must hold: 
$$ L(E) = L(N(E)) $$

(Note that $L$ is overloaded: the language of an RE to the left, the language of an NFA to the right.)

We proceed case by case according to the structure of the syntax of REs.

**RE to NFA, Case $\emptyset$**

Recall: $L(\emptyset) = \emptyset$

$N(\emptyset)$:

Note: $L(N(\emptyset)) = \emptyset = L(\emptyset)$; specification satisfied in this case.

States are given without names for simplicity. Suffice as construction is graphical; states to be named at the end.

**RE to NFA, Case $\epsilon$**

Recall: $L(\epsilon) = \{\epsilon\}$

$N(\epsilon)$:

Note: $L(N(\epsilon)) = \{\epsilon\} = L(\epsilon)$; specification satisfied in this case.

**RE to NFA, Case $x$ for $x \in \Sigma$**

Recall: For each $x \in \Sigma$, $L(x) = \{x\}$

$N(x)$:

Note: $L(N(x)) = \{x\} = L(x)$; specification satisfied in this case.

**RE to NFA, Case $E + F$ (1)**

Recall: $L(E + F) = L(E) \cup L(F)$

$N(E + F)$:

The NFAs $N(E)$ and $N(F)$ in parallel. The initial states of $N(E + F)$ are the union of the initial states of $N(E)$ and $N(F)$.

Thus, specification holds in this case. (This is an inductive case.)

**RE to NFA, Case $E + F$ (2)**

Note: Assuming specification holds for $E$ and $F$,

$$ L(N(E + F)) = L(N(E)) \cup L(N(F)) $$

$$ = L(E) \cup L(F) $$

$$ = L(E + F) $$

**RE to NFA, Case $EF$ (1)**

Sub-case 1: No initial state of $N(E)$ is accepting; i.e. $\epsilon \notin L(N(E))$ (Recall: $L(EF) = L(E)L(F)$)

**RE to NFA, Case $EF$ (2)**

**RE to NFA, Case $EF$ (3)**

Sub-case 2: Some initial states of $N(E)$ are accepting; i.e. $\epsilon \in L(N(E))$
Note: Assuming specification holds for $E$ and $F$, $L(N(\text{EF})) = L(N(F)) \cdot L(N(F))$

Thus, specification holds in this case. (This is an inductive case.)

Example

Systematically construct an NFA for the regular expression:

$$(a + b)^* c$$

("zero or more as or bs, followed by a single c")

Use the "graphical construction". On the white board.

**Scanning**

- The first stage of many real-world language processing tasks, such as a compiler, is to group individual characters into language-specific symbols called Lexemes or Tokens:
  - Keywords (like if, then, while)
  - Literals (like 42, 3.14, 'A', "abc")
  - Special symbols and separators (like :=, (), )
  - ... 

  - This process is called Lexical Analysis or Scanning, and is performed by a Scanner.

- Commonly, white space and comments are understood as token separators.
- An additional task of the scanner is often to discard white space and comments as they usually serve no purpose after the scanning.
- Regular expressions is the most commonly used formalism for describing the Lexical Syntax of a language; i.e. the syntax of the tokens, white space, and comments.
- In essence, a scanner is thus a finite automaton.
There are many famous so-called scanner
generators; e.g., Lex, Flex: given regular
expressions describing the lexical syntax,
they produce a scanner for the language.

We will study a hand-written scanner in
Haskell for a simple language called TXL (for
"Trivial eXpression Language") to give a concrete
example and some practical experience.

When studying the code, try to understand
how the code actually implements a DFA.

Study the following if you're interested; not in exam.

The given regular expression accurately describes
the lexical syntax of TXL, and is thus fine for
checking if a string (word) belongs to TXL or not.
However, for the purpose of breaking a string
into tokens, it is not quite precise enough as
there are ambiguities:

• Id and Keyword overlaps: is let an identifier
  or a keyword?
• Choice between long token or short
tokens: is abc123 an identifier, an identifier
  abc and an integer literal 123, or maybe even
  three identifiers followed by three literals?

The given regular expression accurately describes
the lexical syntax of TXL, and is thus fine for
checking if a string (word) belongs to TXL or not.
However, for the purpose of breaking a string
into tokens, it is not quite precise enough as
there are ambiguities:

• Id and Keyword overlaps: is let an identifier
  or a keyword?
• Choice between long token or short
tokens: is abc123 an identifier, an identifier
  abc and an integer literal 123, or maybe even
  three identifiers followed by three literals?

\[
\begin{align*}
\text{Lexical Syntax TXL (1)} \\
\text{\(' a'\) etc. R.E. for ind. char.; \text{Space} \text{ etc. are \"macros\".} \\
\text{Space} & = ' ' + '\n' \\
\text{Graphic} & = '+' + '-' + '*' + '/' \\
& \quad + '(' + ')' + '=' \\
\text{Digit} & = '0' + ... + '9' \\
\text{Alpha} & = 'a' + ... + 'z' \\
\text{AlphaNum} & = \text{Alpha} + \text{Digit} \\
\text{LitInt} & = \text{Digit} \text{ Digit}^* \\
\text{Id} & = \text{Alpha (Alpha + Digit)}^* \\
\text{Keyword} & = 'l' 'e' 't' + 'i' 'n' \end{align*}
\]

Finally, a regular expression for the entire
language:

\[
txl = (\text{Graphic} + \text{LitInt} + \text{Id} + \text{Keyword} + \text{Space})^*
\]

Such issues are commonly resolved by adopting
certain conventions:

• Keywords takes precedence over
  identifiers; i.e., a token is an identifier only if it
  is not a keyword.
  (Thus, let is a keyword, not an identifier.)
• "Maximal Munch Rule": tokens should be
  as long as possible; i.e., prefer grouping as a
  single long token over a sequence of shorter
  ones.
  (Thus, abc123 is a single token, an identifier.)

\[
\begin{align*}
\text{TXL Scanner (1)} \\
\text{type Id = String} \\
data Token = T_Int Int \\
\mid T_Id Id \\
\mid T_Plus \\
\mid T_Minus \\
\mid T_Times \\
\mid T_Divide \\
\mid T_LeftPar \\
\mid T_RightPar \\
\mid T.Equal \\
\mid T_Let \\
\mid T_In
\end{align*}
\]

\[
\begin{align*}
\text{TXL Scanner (2)} \\
y : : [Char] \rightarrow [Token] \\
\quad \text{-- End of input} \\
y [] = [] \\
\quad \text{-- Drop white space and new lines} \\
y (' ' : cs) = y cs \\
y ('\n' : cs) = y cs
\end{align*}
\]

\[
\begin{align*}
\text{TXL Scanner (3)} \\
y ('+' : cs) = T_Plus : y cs \\
y ('-' : cs) = T_Minus : y cs \\
y ('\cdot' : cs) = T_Times : y cs \\
y ('/' : cs) = T_Divide : y cs \\
y ('(' : cs) = T_LeftPar : y cs \\
y (')' : cs) = T_RightPar : y cs \\
y ('=' : cs) = T.Equal : y cs
\end{align*}
\]

\[
\begin{align*}
\text{TXL Scanner (4)} \\
y (c : cs) \\
\quad \text{-- Lex simple tokens} \\
y ('c' : cs) = T_Id Id : y cs \\
y ('c' : cs) = T_Minus : y cs \\
y ('c' : cs) = T_Times : y cs \\
y ('c' : cs) = T_Divide : y cs \\
y ('c' : cs) = T.LeftPar : y cs \\
y ('c' : cs) = T.RightPar : y cs \\
y (c : cs) = T.Equal : y cs
\end{align*}
\]