The Syntactic Side of Languages

Natural Languages

- stream of phonemes via lexical analysis → stream of words via parsing → sentences

Artificial Languages

- stream of characters via lexical analysis → stream of tokens via parsing → abstract syntax

What is a token?
Variable names, numerals, operators (e.g., +, /, etc.), key-words, …

Lexical structure is typically specified via regular expressions.

References

The following is partly based on

Basics of Compiler Design
by Torben Mogensen
http://www.diku.dk/hjemmesider/ansatte/torbenn/Basics/

Regular Expressions and Automata using Haskell
Simon Thompson
http://www.haskellcraft.com/craft3e/Reg_exps.html

Regular Expressions  (S. Kleene, mid-1950s)

Definition 1.
A regular expression has one of six forms:

- $\emptyset$ — matches no string
- $\epsilon$ — matches the empty string
- $x$ — matches the character ‘$x$’
- $(r_1|r_2)$ — matches the strings matched by $r_1$ or $r_2$
- $(r_1r_2)$ — matches the strings $w_1w_2$ where $w_1$ matches $r_1$ and $w_2$ matches $r_2$
- $(r)^*$ — matches $\epsilon$ and the strings $w_1 \ldots w_k$ where $k > 0$ and each $w_i$ matches $r$

We omit the parens in $(r_1|r_2)$, $(r_1r_2)$, and $(r)^*$ when we can.

§Both Thompson and Mogensen omit this form, and henceforth, so shall we.
($\emptyset$ is necessary for algebraic treatments of regular languages.)
Lexical Analysis

Regular Expressions: Examples

- Sheep Language = \{ ba!, baa!, baaal, \ldots \}.
  - ba*! = (b((a(a^*))!)) matches exactly the Sheep Language strings.
  - (0|1)* matches exactly the strings over 0 and 1, including \( \varepsilon \).
  - (a|(1(0|1)^*))1 matches exactly the binary representation of odd integers.

—more examples shortly—

Notation

\[ r \downarrow s \equiv_{\text{def}} \text{regular expression } r \text{ matches string } s. \]

Applying the Big-Step Rules

Class Exercise. Work out derivations for:

- (0|1)* \downarrow 0101
- (0|1)*((01)|(10)) \downarrow 0110

Big-Step Rules for RegEx Matching

\[
\begin{align*}
\text{\( \varepsilon \)-match:} & \quad \varepsilon \downarrow \varepsilon \\
\text{\( \text{Literal-match:} \) } & \quad \text{x} \downarrow \text{x} \\
\text{\( |-\text{match}_1: \) } & \quad r_1 \downarrow s \quad (r_1|r_2) \downarrow s \\
\text{\( |-\text{match}_2: \) } & \quad r_2 \downarrow s \quad (r_1|r_2) \downarrow s \\
\text{\( \text{++-match:} \) } & \quad (s = s_1|s_2) \\
\text{\( \ast\text{-match}_1: \) } & \quad r^* \downarrow \varepsilon \\
\text{\( \ast\text{-match}_2: \) } & \quad r \downarrow s_1 \quad r^* \downarrow s_2 \quad (s = s_1|s_2)
\end{align*}
\]

[Stage direction: Copy these onto the board, but leave some room.]
Matching Regular Expressions in Haskell, I

**Abstract Syntax**
```
data Reg = Epsilon
  | Literal Char
  | Or Reg Reg
  | Then Reg Reg
  | Star Reg
  deriving (Eq)
```

```
matches :: Reg -> String -> Bool
matches Epsilon st
  = (st == "")
matches (Literal ch) st
  = (st == [ch])
matches (Or r1 r2) st
  = matches r1 st || matches r2 st
```

(continued)

**Credits/Pointers**
- The code here is based on work by Simon Thompson. See: [http://www.haskellcraft.com/craft3e/Reg_exps.html](http://www.haskellcraft.com/craft3e/Reg_exps.html)

---

Matching Regular Expressions in Haskell, II

```
data Reg = Epsilon | Literal Char | Or Reg Reg | Then Reg Reg | Star Reg
  deriving (Eq)
```

```
matches (Then r1 r2) st
  = or [ matches r1 s1 && matches r2 s2 | (s1,s2) <- splits st ]
matches (Star r) st
  -- using frontSplits is
  = matches Epsilon st || -- necessary trickery*
    or [ matches r s1 && matches (Star r) s2
     | (s1,s2) <- frontSplits st ]
```

```
splits, frontSplits :: [a] -> [ ([a],[a]) ]
splits st = [ splitAt n st | n <- [0 .. length st] ]
frontSplits st = [ splitAt n st | n <- [1 .. length st] ]
```

* Our first example of avoiding a left-recursion (≈ a black hole).

---

Stage Directions:
- Replace the case for Star with
  ```
  matches (Star r) st
    = matches Epsilon st ||
      or [ matches r s1 && matches (Star r) s2
       | (s1,s2) <- splits st ]
  ```
  and
  ```
  (Star (Epsilon ‘Or‘ (Literal 'a'))) "aa"
  ```
  try

Jim Royer (CIS-352)
Regular Expressions and the Languages They Name

Definition 2.
Suppose \( r \) is a regular expression and \( A \) and \( B \) are sets of strings.

- \( L(r) \) = the set of strings matched by \( r \).
- \( A \cdot B = \{ w_1 w_2 | w_1 \in A, w_2 \in B \} \).
- \( A^0 = \{ \epsilon \} \), \( A^1 = A \), \( A^2 = A \cdot A \), \( A^3 = A \cdot A \cdot A \), …

Thus:

\[
L(\epsilon) = \{ \epsilon \} \\
L(x) = \{ x \} \\
L(r_1 r_2) = L(r_1) \cup L(r_2) \\
L(r_1 r_2) = L(r_1) \cdot L(r_2) \\
L(r^*) = \{ \epsilon \} \cup L(r) \cdot L(r^*) = \bigcup_{i \geq 0} L(r)^i
\]

Short Cuts (Mogensen, §2.1.1)

- We can write \( (0|1|2|3|4|5|6|7|8|9) \) as \([0123456789]\) or \([0–9]\).
- \( r^+ = r r^* \), i.e.,
  
  \[
  r^* \equiv 0 \text{ more matches of } r \\
  r^+ \equiv 1 \text{ more matches of } r
  \]
- \( r? = r|\epsilon \equiv 0 \text{ or } 1 \text{ matches of } r \).

Examples

\[
[a–zA–Z] = \text{all alphabetic characters} \\
(0|[1–9]|0–9*)) = \text{all natural number constants} \\
[a–zA–Z][a–zA–Z0–9]* \equiv \text{C variable names} \\
"([a–zA–Z0–9]|\[a–zA–Z0–9]\)"* \equiv \text{C string constants}
\]

Regular Expressions with Their Work Boots On

- grep, egrep, fgrep — print lines matching a pattern
  See http://en.wikipedia.org/wiki/Grep

- Also see tr, sed, …
  (The original Unix developers knew their automata theory cold.)

- See http://perldoc.perl.org/perlre.html.
  (Folks in bioinformatics know their pattern matching cold.)


Non-deterministic Finite Automata

A Non-deterministic Finite Automaton (abbreviated NFA) consists of:

- A finite set of states, \( S \).
- A finite set of moves (labeled edges between states)
  (Moves are labeled by either \( \epsilon \) or \( c \in \Sigma = \text{the input alphabet} \))
- A start state (in \( S \)).
- A set of terminal or final states (a subset of \( S \)).

Example 3.

\[
S = \{ 0, 1, 2, 3 \}, \text{ start state } = 0, \text{ final sets } = \{ 3 \} \\
\text{moves } = \{ 0 \overset{a}{\rightarrow} 0, 0 \overset{b}{\rightarrow} 0, 0 \overset{a}{\rightarrow} 1, 1 \overset{b}{\rightarrow} 2, 2 \overset{b}{\rightarrow} 3 \}
\]
The Data.Set Module

To implement NFA’s we need a module for representing sets. We use:

```
Data.Set

empty :: Set a
fromList :: Ord a => [a] -> Set a
intersection :: Ord a => Set a -> Set a -> Set a
Data.Set.map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
singleton :: a -> Set a
size :: Set a -> Int
toList :: Set a -> [a]
union :: Ord a => Set a -> Set a -> Set a

etc.
```

Another Example NFA

```
machN = NFA
  (S.fromList [0..5])
  (S.fromList [Move 0 'a' 1,
               Move 1 'b' 2,
               Move 0 'a' 3,
               Move 3 'b' 4,
               Emove 3 4,
               Move 4 'b' 5])
  0
  (S.fromList [2,5])
```

Note the two sorts of nondeterminism this machine exhibits.

Accepting and rejecting strings

- What is the accepting path of `abb` through `M`?
- What other paths are possible?
- What are the accepting paths of `ab` through `N`?
- What happens with `N` and `aa`?
A small-step semantics for an NFA

Notation
For $M = (\text{States}, \text{Moves}, \text{start}, \text{Final})$:

- $M \vdash s \xrightarrow{a} s' \equiv_{\text{def}} (s, a, s') \in \text{Moves}$.
- $M \vdash s \xrightarrow{\varepsilon} s' \equiv_{\text{def}} (s, \varepsilon, s') \in \text{Moves}$.

$$M \vdash s \xrightarrow{a} s' \Rightarrow (s, a, s') \in \text{Moves}$$

$$M \vdash s \xrightarrow{\varepsilon} s' \Rightarrow (s, \varepsilon, s') \in \text{Moves}$$

[Stage direction: Copy these onto the board.]

Applying the Small-Step Rules, 1

$M = (\{0, 1, 2, 3\}, \{0 \xrightarrow{b} 1, 1 \xrightarrow{a} 2, 2 \xrightarrow{a} 2, 2 \xrightarrow{\varepsilon} 3\}, 0, \{3\})$

An accepting path for $baaa!$:

$$0 \xrightarrow{b} 1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{\varepsilon} 3$$

Applying the Small-Step Rules, Class Exercise

$M = (\{0, 1, 2\}, \{0 \xrightarrow{a} 1, 1 \xrightarrow{a} 0, 0 \xrightarrow{b} 2, 2 \xrightarrow{b} 0\}, 0, \{0\})$

What are accepting paths for $aabbbaa$ and $aabaa$?
NFAs implemented in Haskell

-- (trans nfa str)
-- = the set of states reachable in nfa by following str

trans :: Ord a => Nfa a -> String -> Set a

See http://www.cis.syr.edu/courses/cis352/code/RegExp/ImplementNfa.hs

Example: An NFA for \((ab|ba)^*\)

Example: An NFA for \((ab|ba)^*\)

Handling \(\epsilon\)-Closures

setlimit :: Eq a => (Set a -> Set a) -> Set a -> Set a
setlimit f s = let next = f s
    in if s==next
    then s
    else setlimit f next

Above we assume \(f\) is monotone, i.e., \(f(A) \supseteq A\) for all \(A\).

closure :: Ord a => Nfa a -> Set a -> Set a
closure (NFA states moves start term) s = setlimit add s
where
  add stateset = S.union stateset (S.fromList accessible)
where
  accessible = [ s | x <- S.toList stateset ,
    (Emove y s) <- S.toList moves ,
    y==x ]

Example: An NFA for \((ab|ba)^*\)

Taking one step

onemove :: Ord a => Nfa a -> Char -> Set a -> Set a
onemove (NFA states moves start term) c x
= S.fromList [ s | t <- S.toList x ,
    Move z d s <- S.toList moves ,
    z==t , c==d ]

= \{ s : t \in x and (t,c,s) \in moves \}

onetrans :: Ord a => Nfa a -> Char -> Set a -> Set a
onetrans mach c x = closure mach (onemove mach c x)
Taking many steps

trans :: Ord a => Nfa a -> String -> Set a
trans mach str = foldl step startset str
  where
    step set ch = onetrans mach ch set
    startset = closure mach S.singleton (startstate mach)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl step s (c1:c2:...:ck:[]) = (... ((s ‘step’ c1) ‘step’ c2) ‘step’ ... ‘step’ ck)

Example: The NFA for \((ab|ba)^*\)

The translation in Haskell

From: BuildNfa.hs

build :: Reg -> Nfa Int

build Epsilon = NFA (S.fromList [0..1])
  S.singleton (Emove 0 1)
  0
  S.singleton 1
build (Literal c) = NFA (S.fromList [0..1])
  S.singleton (Move 0 c 1)
  0
  S.singleton 1
build (Or r1 r2) = m_or (build r1) (build r2)
build (Then r1 r2) = m_then (build r1) (build r2)
build (Star r) = m_star (build r)

\(M(r)\) = an NFA for accepting \(L(r)\).

\(\rightarrow 0 \epsilon \rightarrow 9\)

Figure: \(M(\epsilon)\)

\(\rightarrow 0 \times \rightarrow 9\)

Figure: \(M(r_1|r_2)\)

\(\rightarrow 0 \epsilon \rightarrow 9\)

Figure: \(M(x)\)

\(\rightarrow 0 \epsilon \rightarrow 9\)

Figure: \(M(r_1r_2)\)

\(\rightarrow 0 \epsilon \rightarrow 9\)

Figure: \(M(r^*)\)

m_or, m_then, and m_star are a bit ugly — because of all the state renumbering.
Theory Break: Regular Languages

Definition 4.
The regular languages are the languages described by regular expressions ($L(r) : r$ is a reg. exp. $\}$).

Theorem 5.
The regular languages $\subseteq$ the languages accepted by NFAs.

Proof: We need to show the reg.-exp. $\rightarrow$ NFA translation is correct — which is a not-too-hard structural induction.

Theorem 6.
The regular languages $\supseteq$ the languages accepted by NFAs.

Proof: There turns out to be an NFA $\rightarrow$ reg.-exp. translation (which we’ll skip here).

Deterministic Finite Automata

Definition 7.
A deterministic finite automata (abbreviated DFA) is a NFA that
- contains no $\epsilon$-moves, and
- has at most one arrow labelled with a particular symbol leaving any given state.

So in a DFA there is at most one possible move in any situation.
The DFAs also characterize the regular languages.

NFAs and DFAs
- They both accept exactly the regular languages.
- You can translate NFAs to equivalent DFAs, but
- you may pay a price in size blow up.

Extra Topics

Example NFA $\rightarrow$ DFA Translation, I

$A = e$-closure($\{0\} = \{0, 1, 2, 4\}$.

$B = e$-closure($\{s : s' \xrightarrow{a} s, s' \in A\} = \{1, 2, 3, 4, 6, 7\}$. \hspace{1cm} (A $\xrightarrow{a} B$)

$C = e$-closure($\{s : s' \xrightarrow{b} s, s' \in A\} = \{1, 2, 4, 5, 6, 7\}$. \hspace{1cm} (A $\xrightarrow{b} C$)

$D = e$-closure($\{s : s' \xrightarrow{a} s, s' \in B\} = \{1, 2, 4, 5, 6, 7, 8\}$. \hspace{1cm} (B $\xrightarrow{b} D$)

$C = e$-closure($\{s : s' \xrightarrow{b} s, s' \in B\} = \{1, 2, 4, 5, 6, 7\}$. \hspace{1cm} (B $\xrightarrow{b} C$)

Similarly, $C \xrightarrow{b} D$, $C \xrightarrow{b} C$, $D \xrightarrow{a} D$, $D \xrightarrow{b} C$. 

Jim Royer (CIS 352)  Lexical Analysis  February 13, 2018  29 / 40
The NFA to DFA algorithm in Haskell

```haskell
make_deterministic :: Nfa Int -> Nfa Int
make_deterministic = number . make_deter

number :: Nfa (Set Int) -> Nfa Int
number (NFA states moves start finish) = NFA states' moves' start' finish'
where ...

make_deter :: Nfa Int -> Nfa (Set Int)
make_deter mach = deterministic mach (alphabet mach)
```

Switch to NfaToDfa.hs.

Minimizing DFAs, 1

**Definition 8.** Suppose \( s \) and \( s' \) are states in a DFA \( M \).
- \( s \) and \( s' \) are **distinguished** by \( x \) when 
  \( M \) started in \( s \) run on \( x \) accepts \( \iff \) \( M \) started in \( s \) run on \( x \) rejects
- \( s \) and \( s' \) are **indistinguishable** when no string \( x \) distinguishes them.
  
  So, we can treat merge \( s \) and \( s' \) safely into a single state.

- \( \epsilon \) distinguishes \( D \) and each of \( A, B, C \)
- \( a \) distinguishes \( A \) and each of \( B \) and \( C \).
- \( B \) and \( C \) turn out to be indistinguishable.
- The result of merging \( B \) and \( C \) is:

Minimizing DFAs, 2

See Tom Henzinger’s notes on the Myhill-Nerode Theorem

(Much handier than the Pumping Lemma for regular languages)
In building a compiler or interpreter, you want to specify the lexical part of the language (e.g., token) by regular definitions (hopped-up regular expressions). E.g.:

\[
IF = \text{if} \\
ID = [a-zA-Z][a-zA-Z0-9]^* \\
NUM = [\+-][0-9]^* \\
FLOAT = \text{a nasty mess}
\]

Then you translate the entire collection of these to an NFA. E.g.:

By convention, you take the longest match of a string.

Then you translate the NFA to a DFA with which you scan through the input and spit out tokens with lightning speed.

See §2.9 of Mogensen for details.