CS 344 - Scanning

Announcements

- Essay due on Friday
- Next HW will be up by Friday
Last time: Regular expressions

- A character
- The empty string, $\varepsilon$
- 2 regular expressions concatenated
- 2 regular expressions separated by an "or" (written $|$)
- A regular expression followed by $*$ (Kleene star - 0 or more occurrences)

$$d^+ = (d^*)^+$$
Ex: Give the regular expression for $\exists w \mid w$ begins with a 1 and ends with a 0.

$1 (0|1)^* 0$

Ex: $\exists w \mid w$ starts with 0 and has an odd length.

$0 (0|1)(0|1)^*$
Deterministic Finite Automate (DFA)

Regular languages are precisely the things recognized by DFAs.

- A set of states
- Input alphabet
- A start state
- A set of accept states
- A transition function: given a state and an input, output a new state
Ex: String of 0's if number of 1's is even

Start state $S_1$ transitions:
- $S_1 \rightarrow S_2$ on input 0
- $S_2 \rightarrow S_1$ on input 1

Accept state $S_2$
Ex: 3 symbol alphabet: \( \{0, 1, 2\} \)

computing mod 3
accepting if sum of word is 0 mod 3
NFAs: DFAs w/ ambiguity

NFA has $n$ states
DFA has up to $2^n$ states

If read in a 1, could go to $s_1$ or $s_2$.
Converting NFAs to DFAs

\[ \exists S_1, S_2, S_3 \]

\[ \exists w \mid w \text{ contains } \text{11 as a substring} \]
Context free Grammars (ε-BNF)

Ex:

$$
expr \rightarrow \text{expr op expr} \downarrow \\
(\text{expr}) \mid \text{-expr} \\
$$

terminals → id \mid \text{number} \\
\text{variable} \mid \text{int/Float} \\

$$
\text{op} \rightarrow + \mid - \mid \ast \mid / \\
$$

terminals

$$
x = 5 \\
y = 2
$$
A derivation: derive slope * x + intercept

expr \Rightarrow expr \text{ op } expr
\Rightarrow expr + expr
\Rightarrow expr + id \text{ (intercept)}
\Rightarrow expr \text{ op } expr + id
\Rightarrow expr * expr + id
\Rightarrow expr * id(x) + id
\Rightarrow id(slope)* id + id
Derivation tree.

```
expr
  /\   /\   /
expr op expr + id(intercept)
  |     |     |
id(slope) * id(x)
```

(rightmost derivation)
Ambiguous grammars

```
expr
  /\   /
expr op expr
  /   /   +
Expr op Expression

id(slope) * id(x)

id(slope)  *
  /\   /
expr op expr
  /   /   +
Expr op Expression

id(x) + id(intercept)
```

leftmost derivation
There are infinitely many ways to make a grammar for any context. See language.

Problem in the parsing stage: which is better?

(Try to define unambiguous grammars.)
Another example (from last time)

Expression grammars: Simple calculator

\[ \text{expr} \rightarrow \text{term} | \text{expr} \text{ add-op } \text{term} \]

\[ \text{term} \rightarrow \text{factor} | \text{term} \text{ mult-op } \text{factor} \]

\[ \text{factor} \rightarrow \text{id} | \text{number} | - \text{factor} | (\text{expr}) \]

\[ \text{add-op} \rightarrow + | - \]

\[ \text{mult-op} \rightarrow \times | \div \]

\[ \text{terminals} \]
Parse Tree

expr
  /\       
expr   add_op
   /\       /
 term +  term
 /\       /\     /
 factor factor mult_op
   /\     /
 number(3) number(4) number(5)
Scannness

Find the syntax (not semantics) of code.

Output tokens.

3 types

• Ad-hoc

• Finite automaton
  • nested case statements
  • table + driver
Ad-hoc  (last time)

if current ∈ \{ “,”, “)” \} return that symbol
if current = “;”
    read next
    if it is = announce “assign”
    else announce error
if current = “/”
    read next
    if it is “*” or “/”
        read until “*” or “newline” (resp.)
    else return divide

etc.
Ad-hoc approach

Advantage: code is fast and compact

Disadvantage:
  - very ad-hoc
  - hard to debug
  - no explicit representation
DFA approach

Recall our simple calculator language.

But how to get this DFA and then how to actually model it?
Constructing a DFA

Given a regular expression, we can construct an NFA.

Simple NFA:

(Base case)
3 operations

Concatenation:

Or:
and Kleene closure ($\star$).

d) Kleene closure
Example: decimals \( d^* (d \cdot d^* \cdot d^*) d^* \)

Base: \( 0 \rightarrow 0 \)

\( d^* \): 

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \]

\( d \cdot d^* \): 

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \]

\( d \cdot d^* \cdot d^* \): 

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \]

\( d^* \cdot d^* \cdot d^* \): 

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \]

\( d^* \cdot d^* \cdot d^* \cdot d^* \): 

\[ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \]
Final product:

A | B | C
Next: Convert to DFA.
(Lots of states, but same principle as we saw earlier.)

Result:
(see p. 57-58)
Note: This DFA is a bit redundant. Not minimal. Can easily find the equivalence classes and minimize.
Process to minimize
Given DFA, generate case statements to simulate it.

Now:

\[ \text{State} = 1 \]
\[ \text{repeat:} \]
\[ \text{read curr-char} \]
\[ \text{case state is:} \]
\[ \begin{align*}
1: & \quad \text{case curr-char = d} \\
& \quad \text{state} = 2 \\
& \quad \text{case curr-char = .} \\
& \quad \text{state} = 3
\end{align*} \]
Scanner Tools

In reality, this DFA is often done automatically. Specify the rules of regular language, and the program does this for you.

Many such examples:

Lex (flex), JLex (JFlex), Quex, Ragel, ...
Next time:
Lex/Flex: C-style driver

Look for HW on regular expressions, NFA/PFA, and context-free languages.

Next programming assignment will use flex.