CS 344 - Scanning

Announcements

- HW2 is due in 1 week
Last time: Regular expressions

- A character
- The empty string, ε
- 2 regular expressions concatenated
- 2 regular expressions separated by an or (written |)
- A regular expression followed by * (Kleene star - 0 or more occurrences)
Ex: Give the regular expression for \( \exists w \mid w \) begins with a 1 and ends 0 with a 0 if

\[ 1 (0|1)^* 0 \]

Ex: \( \exists w \mid w \) starts with 0 and has an odd length if

\[ 0 ((01)^* (01))^* \]
Deterministic Finite Automate (DFA)

Regular languages are precisely the 0 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
Example: Let $L$ be the language of strings of 0's and 1's such that the number of 1's is even.

- $L = \text{accept if number of 1's is even}$

Relevant regex: $(0^*10^*10^*)^*$

Strings accepted by the automaton: $001010l_1$, $1111 \not\in L$
Ex: 3 symbol alphabet: \( \{0, 1, 2\} \)

Counts modulo 3
accepts words \( w \) if \( \sum = 0 \mod 3 \)
NFAs w/ ambiguity

\[ \rightarrow 10 \]

Thm: NFAs and DFAs are equivalent.
Converting NFAs to DFAs (p. 57)
(Empty set is possible!)
Context free Grammars (\textit{BNF})

Ex:

\[
\text{Expr} \rightarrow \text{Expr} \ \text{Op} \ \text{Expr} \ \mid \ (\text{Expr}) \ \mid \text{id} \ \mid \text{number}
\]

\[
\text{Op} \rightarrow + \ \mid - \ \mid \ast \ \mid \div
\]
A derivation: derive \( \frac{1}{2} \ast x + \text{intercept} \)

\[
\text{Expr} \Rightarrow \text{Expr Op Expr} \\
\Rightarrow \text{Expr} + \text{Expr} \\
\Rightarrow \text{Expr} + \text{id(\text{intercept})} \\
\Rightarrow \text{Expr Op Expr} + \text{id(\text{intercept})} \\
\Rightarrow \text{Expr Op Expr} + \text{id(\text{intercept})} \\
\Rightarrow \text{number} \left( \frac{1}{2} \right) \ast \text{id}(x) + \text{id(\text{intercept})}
\]
Derivation tree

(expr
  /\  /
(expr op expr) + id(intercept)
  / \       /
(expr op id(x))
  / \\
\text{id(slope) \times v2}

(rightmost derivation)
Ambiguous grammars

```
expr
  `/`
  expr
  `/`
  expr
  `/`
  op
  expr
  `/`
  +
  id(intercept)

id(slope) * id(x)
```

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)
goal: avoid ambiguity from last ex

Expression grammars: Simple calculator

Expr → Term | Expr Add-op Term

Term → Factor | Term Mul-t_op Factor

Factor → id | number | -Factor | (Expr)

Add-op → + | -

Mul-t_op → * | /
Parse Tree

\[ 6 \cdot 3 + 4 \times 5 \] (avoids rightmost derivation)
Scannness: do this in code

Find the syntax (not semantics) of code.

Output tokens.

A few types:

- Ad-hoc

- Finite automata → DFA/NFA
  - nested case statements
  - table + driver

→ Simulates DFA
Ad-hoc: case based code

If current \in \{ `(`, `)`, `+`, `-`, `*` \}
  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

Advantages:
- Code is fast & compact

Disadvantages:
- 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:

- Concatenation

Or:

- $\varepsilon$
- $\varepsilon$
- $\varepsilon$

A

B

A

B

A|B

could be huge
and Kleene closure ($\varepsilon$).

d) Kleene closure
Example: decimals $d^* (d \mid d^*) d^*$

Base: \[
\begin{array}{c}
0 \\
\downarrow \hspace{2cm} \downarrow \\
0 \\
\end{array}
\]
Final product:
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
Now:

Given DFA, generate case statements to simulate it.

State = 1
repeat:
read curr_char
case state is:

A: case curr_char = d
   state = B

B:

(c) Start

A → B → C → DEF
A → B → C → DEF
A → B → C → DEF
A → B → C → DEF

state = C

state = C

state = C

state = C
Scanners-Tools

In reality, this DFA is often done automatically.

Specify the rules of the language, and the program does this for you.

Many such examples:
lex (flex), jflex, Quex, Rigel, ...
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.