Lexical Analysis

Finite Automata

(Part 2 of 2)

Kinder, Gentler Nation

- In our post drop-deadline world ...
- ... things get easier.
- While we’re here: reading quiz.

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \rightarrow \text{input state}$

Finite Automata

- Transition $s_1 \rightarrow^a s_2$
- Is read
  In state $s_1$ on input “$a$” go to state $s_2$
- If end of input (or no transition possible)
  - If in accepting state $\Rightarrow$ accept
  - Otherwise $\Rightarrow$ reject

Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition $a$
A Simple Example

- A finite automaton that accepts only “1”

- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet $\Sigma = \{0, 1\}$

- Check that “1110” is accepted but “110…” is not

And Another Example

- Alphabet $\Sigma = \{0, 1\}$
- What language does this recognize?
And Another Example

- Alphabet still $\Sigma = \{ 0, 1 \}$

- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state

---

Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

- Machine can move from state A to state B without reading input

---

Deterministic and Nondeterministic Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No $\varepsilon$-moves

- **Nondeterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves

- Finite automata have finite memory
  - Need only to encode the current state
Execution of Finite Automata

• A DFA can take only one path through the state graph
  - Completely determined by input

• NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

• An NFA can get into multiple states

• Input:
  0 1 1 0 1 0 1

• Rule: NFA accepts if it can get in a final state

NFA vs. DFA (1)

• NFAs and DFAs recognize the same set of languages (regular languages)
  - They have the same expressive power

• DFAs are easier to implement
  - There are no choices to consider
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA

```
0 -> 1 -> 0
1 -> 0
```

DFA

```
0 -> 1 -> 0
1 -> 0
```

- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

- High-level sketch

Regular expressions

<table>
<thead>
<tr>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Specification</td>
<td>Table-driven Implementation of DFA</td>
</tr>
</tbody>
</table>

Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A
```
A
```
- For ε
```
ε
```
- For input a
```
a
```
Regular Expressions to NFA (2)

- For \( AB \)
  ![Diagram](image1)

- For \( A | B \)
  ![Diagram](image2)

Regular Expressions to NFA (3)

- For \( A^* \)
  ![Diagram](image3)

Example of RegExp -> NFA conversion

- Consider the regular expression \( (1 | 0)^*1 \)
- The NFA is
  ![Diagram](image4)
NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA
  - a non-empty *subset of states* of the NFA
- Start state
  - the set of NFA states reachable through $\varepsilon$-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from the states in $S$ after seeing the input $a$
    - considering $\varepsilon$-moves as well

NFA $\rightarrow$ DFA Example
NFA → DFA: Remark

- An NFA may be in many states at any time
- How many different states?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - \(2^N - 1\) = finitely many

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition \(S_i \rightarrow S_k\) define \(T[i, a] = k\)
- DFA “execution”
  - If in state \(S_i\) and input \(a\), read \(T[i, a] = k\) and skip to state \(S_k\)
  - Very efficient

Table Implementation of a DFA

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Implementation (Cont.)

- NFA → DFA conversion is at the heart of tools such as flex or ocamllex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

PA1: Lexical Analysis

- Correctness is job #1.
  - And job #2 and #3!
- Tips on building large systems:
  - Keep it simple
  - Design systems that can be tested
  - Don’t optimize prematurely
  - It is easier to modify a working system than to get a system working

Lexical Analyzer Generator

- Tools like lex and flex and ocamllex will build lexers for you!
- You will use this for PA1
- I’ll explain ocamllex; others are similar
  - See PA1 documentation
Ocamllex “lexer.mll” file

```ml
let re_name = re,
rule normal_tokens = parse
  re1 { token1 }
| re2 { token2 }
and special_tokens = parse
| re_n { token_n }
```

Example “lexer.mll”

```ml
let digit = ['0' - '9']
rule initial = parse
  '/' { Tok_Divide }
| digit digit* { let token_string = Lexing.lexeme lexbuf in
              let token_val = int_of_string token_string in
              Tok_Integer(token_val) }
| _ { Printf.printf "Error!
"; exit 1 }
```

Adding Winged Comments

```ml
let digit = ['0' - '9']
rule initial = parse
  "//" { eol_comment }
| '/' { Tok_Divide }
| digit digit* { let token_string = Lexing.lexeme lexbuf in
              let token_val = int_of_string token_string in
              Tok_Integer(token_val) }
| _ { Printf.printf "Error!
"; exit 1 }
```

and eol_comment = parse
  "\n" { initial lexbuf }
| _ { eol_comment lexbuf }
Using Lexical Analyzer Generators

```
$ ocamllex lexer.mll
45 states, 1083 transitions, table size 4602 bytes

(* your main.ml file ... *)
let file_input = open_in "file.cl" in
let lexbuf = Lexing.from_channel file_input in
let token = Lexer.initial lexbuf in
match token with
| Tok_Divide -> printf "Divide Token!\n"
| Tok_Integer(x) -> printf "Integer Token = %d\n" x
```

How Big Is PA1?

- The reference “lexer.mll” file is 88 lines
  - Perhaps another 20 lines to keep track of input line numbers
  - Perhaps another 20 lines to open the file and get a list of tokens
  - Then 65 lines to serialize the output
  - I’m sure it’s possible to be smaller!

- Conclusion:
  - This isn’t a code slog, it’s about careful forethought and precision.

Homework

- Friday: PA1 due
- Next Tuesday: Chapters 2.3 - 2.3.2
  - Optional Wikipedia article