Scanners
Agenda

- Regular expressions
- Finite automata
- Practical considerations
- Scanners & scanner generators
Scanners

• Recall: scanners break input stream up into a set of tokens
  • Identifiers, reserved words, literals, etc.

• What do we need to know?
  • How do we define tokens?
  • How can we recognize tokens?
  • How do we write scanners?
Regular expressions

- **Regular sets**: set of strings defined by regular expressions
  - Strings are regular sets (with one element): purdue 3.14159
  - So is the empty string: 𝜆 (sometimes use ε instead)
  - Concatenations of regular sets are regular: purdue3.14159
  - To avoid ambiguity, can use ( ) to group regexps together
  - A choice between two regular sets is regular, using |: (purdue|3.14159)
  - 0 or more of a regular set is regular, using *: (purdue)*
  - Some other notation used for convenience:
    - Use Not to accept all strings except those in a regular set
    - Use ? to make a string optional: x? equivalent to (x|𝜆)
    - Use + to mean 1 or more strings from a set: x+ equivalent to xx*
    - Use [ ] to present a range of choices: [1-3] equivalent to (1|2|3)
Examples of regular expressions

- Numbers: $D = [0-9]+$  
- Words: $L = [A-Za-z]+$  
- Literals (integers or floats): $-?D+(.D*)?$  
- Identifiers: $(\_|L)(\_|L|D)^*$  
- Comments (as in Micro): -- Not(\n)*\n  
- More complex comments (delimited by ##, can use # inside comment): ##((#|\lambda)Not(#))##
Finite automata

- Finite state machine which will only accept a string if it is in the set defined by the regular expression $(a \ b \ c^+)+$
# Building a FA from a regexp

<table>
<thead>
<tr>
<th>Expression</th>
<th>FA</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td><img src="source.png" alt="Diagram" /></td>
</tr>
<tr>
<td>λ</td>
<td><img src="source.png" alt="Diagram" /></td>
</tr>
<tr>
<td>AB</td>
<td><img src="source.png" alt="Diagram" /></td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A*</td>
<td><img src="source.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

**Mini-exercise:** how do we build an FA that accepts Not(A)?
NFAs to DFAs

• Note that if a finite automaton has a $\lambda$-transition in it, it may be non-deterministic (do we take the transition? or not?)

• More precisely, FA is non-deterministic if, from one state reading a single character could result in transition to multiple states

• How do we deal with non-deterministic finite automata (NFAs)?

• Group nodes that can be reached by the same character into a single node

• Algorithm in textbook, page 82

• Note: this can result in very large DFAs!
DFA reduction

- DFAs built from NFAs are not necessarily optimal
- May contain many more states than is necessary

\[(ab)^+ \equiv (ab)(ab)^*\]
DFA reduction

• DFAs built from NFAs are not necessarily optimal
  • May contain many more states than is necessary

\[(ab)^+ \equiv (ab)(ab)^*\]
DFA reduction

• Intuition: merge equivalent states

• Two states are equivalent if they have the same transitions to the same states

• Basic idea of optimization algorithm

• Start with two big nodes, one representing all the final states, the other representing all other states

• Successively split those nodes whose transitions lead to nodes in the original DFA that are in different nodes in the optimized DFA

• See algorithm on page 85 of textbook
Transition tables

- Table encoding states and transitions of FA
  - 1 row per state, 1 column per possible character
  - Each entry: if automaton in a particular state sees a character, what is the next state?

<table>
<thead>
<tr>
<th>State</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Diagram:
- Start state: 1
- Transition: a to 2
- State: 2 to 3
- Final state: 4
Finite automata program

- Using a transition table, it is straightforward to write a program to recognize strings in a regular language

```c
state = initial_state; //start state of FA
while (true) {
    next_char = getc();
    if (next_char == EOF) break;
    next_state = T[state][next_char];
    if (next_state == ERROR) break;
    state = next_state;
}
if (is_final_state(state))
    //recognized a valid string
else
    handle_error(next_char);
```
Alternate implementation

• Here’s how we would implement the same program “conventionally”

```c
next_char = getc();
while (next_char == 'a') {
    next_char = getc();
    if (next_char != 'b') handle_error(next_char);
    next_char = getc();
    if (next_char != 'c') handle_error(next_char);
    while (next_char == 'c') {
        next_char = getc();
        if (next_char == EOF) return; //matched token
        if (next_char == 'a') break;
        if (next_char != 'c') handle_error(next_char);
    }
}
handle_error(next_char);
```
Transducers

- Simple extension of a FA which also outputs the recognized string

- Recognized characters are output; everything else is discarded

- Annotate transitions:
  - \( T(x) \): “toss” \( x \)
  - \( x \): “save” \( x \)

- Example: DFA to recognize comments and “if” token
Example: Transducer for strings

- Recognize quoted strings
- Can use double quotation marks (""”) within string to produce a quotation mark
- \( (" (\text{Not}(")) \mid \"\")^* " \)
- Examples:
  - "ECE 573"
    - \( \rightarrow \) ECE 573
  - "Scanning is "fun""
    - \( \rightarrow \) Scanning is “fun”
Practical Considerations

Or: what do I have to worry about if I’m actually going to write a scanner?
Handling reserved words

- Keywords can be written as regular expressions. However, this leads to a big blowup in FA size.

- Consider writing a regular expression that accepts identifiers which cannot be `if`, `while`, `do`, `for`, etc.

- Usually better to specify reserved words as “exceptions”

- Capture them using the identifier regexp, and then decide if the token corresponds to a reserved word
Handling compiler directives

- Simple directives (e.g., include files, conditional compilation) can be parsed by the scanner
- No need to invoke full parser

```c
#include <stdio.h>

#pragma omp for
for (int i = 0; i < N; i++) {
    A[i] = B[i] + C[i]
}
```
Generating symbol table entries

- In simple languages, the scanner can build the symbol table directly
- In more complex languages, with complicated scoping rules, this needs to be handled by the parser
Lookahead

• Up until now, we have only considered matching an entire string to see if it is in a regular language

• What if we want to match multiple tokens from a file?
  • Distinguish between int a and inta
  • We need to look ahead to see if the next character belongs to the current token
  • If it does, we can continue
  • If it doesn’t, the next character becomes part of the next token
Multi-character lookahead

- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens

- Examples
  - Fortran: `DO I = 1,100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Pascal: `23.85` (literal) vs. `23..85` (range)

- 2 solutions: Backup or special “action” state
Multi-character lookahead

- Sometimes, a scanner will need to look ahead more than one character to distinguish tokens

- Examples
  - Fortran: `DO I = 1,100` (loop) vs. `DO I = 1.100` (variable assignment)
  - Pascal: `23.85` (literal) vs. `23..85` (range)

- 2 solutions: Backup or special “action” state
General approach

- Remember states (T) that can be final states
- **Buffer** the characters from then on
- If stuck in a non-final state, back up to T, restore buffered characters to stream
- Example: 12.3e+q
Why can’t we do this?

• Just build an FA which recognizes the string

\[ D^+ (\lambda | .D+) (. | ..) D^+ (\lambda | .D+) \]

and recognize the final state we are in to determine the token type?

• Note that this will recognize tokens of the form 12.3 and 12..3
Error Recovery

• What do we do if we encounter a lexical error (a character which causes us to take an undefined transition)?

• Two options
  • Delete all currently read characters, start scanning from current location
  • Delete first character read, start scanning from second character
    • This presents problems with ill-formatted strings (why?)
    • One solution: create a new regexp to accept runaway strings
Scanner Generators
Scanner generators

- Essentially, tools for converting regular expressions into finite automata
- Two well-known tools
  - **ScanGen**: a scanner generator that produces transition tables for a finite automaton driver program (as we saw earlier)
  - **Lex**: generates a scanner directly, makes use of user-written “filter” functions to output tokens
ScanGen

- User defines the input to ScanGen using a file with three sections:
  - **Options**: ScanGen settings for table optimization, etc.
  - **Character classes**: define sets of characters (e.g., digits)
  - **Token definitions**:
    - Token name \{ minor major \} = regexp
      - Can include “except” clauses to simplify regexps
      - Can “toss” parts of regexps
    - Sample ScanGen input (for Micro language): page 61 of textbook
ScanGen driver

- Driver routine provides the actual scanner, which will be called by the parser

```c
void scanner(codes *major,
            codes *minor,
            char *token_text)
```

- Reads input character stream, drives the finite automaton using the table generated by ScanGen, and returns found tokens
ScanGen produces two tables:

- **State table**: `next_state[NUM_STATES][NUM_CHARS]`
  - Encodes transition table
- **Action table**: `action[NUM_STATES][NUM_CHARS]`
  - Tells the driver when a complete token is recognized (i.e., defines accepting states), and what to do with the “lookahead” character
Actions

• Action table has 6 possible values
  • ERROR: scan error
  • MOVEAPPEND: add next character to token string and continue
  • MOVENOAPPEND: “toss” next character and continue
  • HALTAPPEND: add next character to token string and return it (final state)
  • HALTNOAPPEND: “toss” next character and return token (final state)
  • HALTREUSE: put next character back on to input and return token (final state)

• Question: Why no “MOVEREUSE” state?
• Driver program on pages 65–66 of textbook
Lex (Flex)

- Commonly used Unix scanner generator (superseded by Flex)
- Has character classes and regular expressions like ScanGen but some key differences:
  - After each token is matched, calls user-defined “filter” function, which processes identified token before returning it to parser
    - Hence, no “Toss” facility (why?)
  - No exception list
    - Instead, supports matching multiple regexps.
      - Matches longest token (i.e., doesn’t think ifa is IF ID(a))
      - In case of tie, returns earliest-defined regexp
    - To treat if as a reserved word instead of an identifier, define token IF before defining identifiers.
Lex operation

Example of Lex input on page 67 of textbook
Next Time

• We’ve covered how to tokenize an input program
• But how do we decide what the tokens actually say?
  • How do we recognize that
    
    IF ID(a) OP(<) ID(b) { ID(a) ASSIGN LIT(5) ; }

    is an if-statement?

• Next time: Parsers