CS 536 Announcements for Monday, February 5, 2024

Programming Assignment 1
• symbol table files due Thursday, Feb. 8 by 11:59 pm

Homework 0
• available in schedule
• practice with DFAs, regular expressions

Homework 1
• available tomorrow
• practice with NFA→DFA translation, JLex

Last Time
• non-deterministic FSMS
• equivalence of NFAs and DFAs
• regular languages
• regular expressions

Today
• regular expressions → DFAs
• language recognition → tokenizers
• scanner generators
• JLex

Next Time
• CFGs

Recall

scanner = token to regex + regex to NFA + NFA to DFA + DFA to code

scanner generator
From regular expressions to NFAs

Overview of the process

- Conversion of literals and epsilon → simple FAs
- Conversion of operators
  - Convert operands to NFAs
  - Join NFAs

Regex to NFA rules

Rules for operands

- literal 'a'  → 0
- epsilon ε  → 0

Suppose A is a regex with NFA:

Convert so only 1 final state

Suppose A is a regex with NFA:

Rules for alternation A|B
Regex to NFA rules

Rules for catenation A.B

Rules for iteration A*

Tree representation of a regex

Consider regex: ( letter | '_' ) ( letter | '_' | digit )*
**Regex to DFA**

We now can do:

\[
\text{regex} \rightarrow \text{NFA} \xrightarrow{\epsilon} \text{NFA} \xrightarrow{w} \text{DFA}
\]

We can add one more step: **optimize DFA**

**Theorem:** For every DFA $M$, there exists a unique equivalent smallest DFA $M^*$ that recognizes the same language as $M$.

To optimize:

- remove unreachable states
- remove dead states
- merge equivalent states

But what's so great about DFAs?

**Recall:** state-transition function ($\delta$) can be expressed as a table

- very efficient array representation

\[
\begin{array}{ccc}
  a & b & c \\
  S_1 & S_1 & S_2 \\
  S_2 & S_1 & S_2 \\
\end{array}
\]

- efficient algorithm for running (any) DFA

```plaintext
s = start state
while (more input){
    c = read next char
    s = table[s][c]
}
if s is final, accept
else reject
```

What else do we need?

**FSMs** – only check for **language membership** of a string

**scanner** needs to

- recognize a stream of many different tokens using the **longest match**
- know what was matched
Table-driven DFA → tokenizer

**Idea:** augment states with actions that will be executed when state is reached

Consider: \((\text{letter})(\text{letter} | \text{digit})*\)

**Problem:** Don't get longest match

**Problem:** maybe we need this char

**Actions needed:**
- return a token
- put back a character
- report an error

Scanner Generator Example

**Language description:**
consider a language consisting of two statements

- assignment statements: \(\text{ID} = \text{expr}\)
- increment statements: \(\text{ID} += \text{expr}\)

where \(\text{expr}\) is of the form:

- \(\text{ID} + \text{ID}\)
- \(\text{ID} ^ \text{ID}\)
- \(\text{ID} < \text{ID}\)
- \(\text{ID} <= \text{ID}\)

and \(\text{ID}\) are identifiers following C/C++ rules (can contain only letters, digits, and underscores; can’t start with a digit)

**Tokens:**

<table>
<thead>
<tr>
<th>Token</th>
<th>Regular expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>“ = “</td>
</tr>
<tr>
<td>INCR</td>
<td>“ += ”</td>
</tr>
<tr>
<td>PLUS</td>
<td>“ + ”</td>
</tr>
<tr>
<td>EXP</td>
<td>“ ^ ”</td>
</tr>
<tr>
<td>LESSTHAN</td>
<td>“ &lt; ”</td>
</tr>
<tr>
<td>LEQ</td>
<td>“ &lt;= ”</td>
</tr>
<tr>
<td>ID</td>
<td>(letter</td>
</tr>
</tbody>
</table>
Combined DFA

Actions
\( S_1 \): return ASSIGN
\( S_2 \): return EXP
\( S_3 \): return INC
\( S_4 \): put 1 back, return PLUS
\( S_5 \): return LEQ
\( S_6 \): put 1 back, return LESSTHAN
\( S_7 \): put 1 back, return ID

State-transition table

<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>+</th>
<th>^</th>
<th>&lt;</th>
<th>letter</th>
<th>digit</th>
<th>EOF</th>
<th>none of these</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>ret ASSIGN</td>
<td>A</td>
<td></td>
<td>ret EXP</td>
<td>B</td>
<td>C</td>
<td>C</td>
<td>ret EOF</td>
</tr>
<tr>
<td>A</td>
<td>ret INC</td>
<td>put 1 back, ret PLUS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>ret LEQ</td>
<td>put 1 back, ret LESSTHAN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>put 1 back, ret ID</td>
<td></td>
<td></td>
<td>C</td>
<td>C</td>
<td>C</td>
<td>put 1 back, ret ID</td>
<td></td>
</tr>
</tbody>
</table>

```java
do {
    read char
    perform action / update state
    if (action was to return a token)
      start again in start state
} while not(EOF or stuck)
```
Lexical analyzer generators
(aka scanner generators)

Formally define transformation from regex to scanner

Tools written to synthesize a lexer automatically

- Lex : UNIX scanner generator, builds scanner in C
- Flex : faster version of Lex
- JLex : Java version of Lex

JLex

Declarative specification \textit{(non-procedural)}

- you don't tell JLex \textit{how} to scan / how to match tokens
- you tell JLex \textit{what} you want scanned (tokens) \& what to do when a token is matched

Input: set of regular expressions + associated actions

Output: Java source code for a scanner

Format of JLex specification

3 sections separated by `%%`

- user code section
- directives
- regular expression rules

Regular expression rules section

Format: `<regex>{code}` where `<regex>` is a regular expression for a single token

- can use macros from Directives section – surround with curly braces `{ }`
- characters represent themselves (except special characters)
- characters inside " " represent themselves (except \"")
- . matches anything

Regular expression operators: `| * + ? ( )`

Character class operators: `- ^ \`
### JLex example

// This file contains a complete JLex specification for a very small example.

// User Code section: For right now, we will not use it.

%%

DIGIT=   [0-9]
LETTER=  [a-zA-Z]
WHITESPACE=  [\040\t\n]

%state SPECIALINTSTATE

%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

%eofval{
System.out.println("All done");
return null;
%eofval}
%line
%

({LETTER}|"_")({DIGIT}|{LETTER}|"_")* {
    System.out.println(yyline+1 + ": ID " + yytext()); }

"="        { System.out.println(yyline+1 + ": ASSIGN"); }
"="        { System.out.println(yyline+1 + ": PLUS"); }
"="        { System.out.println(yyline+1 + ": EXP"); }
"="        { System.out.println(yyline+1 + ": LESSTHAN"); }
"="        { System.out.println(yyline+1 + ": INCR"); }
"="        { System.out.println(yyline+1 + ": LEQ"); }
{WHITESPACE}*  { }
.              { System.out.println(yyline+1 + ": bad char"); }

Using scanner generated by JLex in a program

// inFile is a FileReader initialized to read from the file to be scanned
Yylex scanner = new Yylex(inFile);
try {
    scanner.next_token();
} catch (IOException ex) {
    System.err.println(
        "unexpected IOException thrown by the scanner");
    System.exit(-1);
}