CS 536 Announcements for Tuesday, February 8, 2022

Programming Assignment 1
- Part 2 files due Tuesday, Feb. 8 by 11:59 pm

Last Time
- regular expressions
- regular expressions → DFAs

Today
- language recognition → tokenizers
- scanner generators
- JLex

Next Time
- CFGs

Recall

\[
\text{scanner} = \text{token to regex} + \text{regex to NFA} + \text{NFA to DFA} + \text{DFA to code}
\]
Regex to DFA

We now can do:

\[
\text{regex} \rightarrow \text{NFA} \rightarrow \text{NFA, \ no \ } \varepsilon \rightarrow \text{DFA}
\]

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 \( () \) can be expressed as a table

- very efficient array representation

\[
\begin{array}{c|ccc}
& a & b & c \\
\hline
s_1 & s_2 & s_2 & s_2 \\
\hline
s_2 & s_1 & s_1 & s_2 \\
\hline
\end{array}
\]

- efficient algorithm for running (any) DFA

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

What else do we need?
Table-driven DFA $\rightarrow$ tokenizer

**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

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

Consider $(\text{letter})(\text{letter} \mid \text{digit})^*$

- $S_1 \xrightarrow{L} S_2 \xrightarrow{D, L, 0}$
- $S_1 \xrightarrow{\text{not } L \mid D}$

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

- $S_1 \xrightarrow{L} S_2 \xrightarrow{D, L, 0}$
- $S_2 \xrightarrow{\text{not } L \mid D}$

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

Also, add EOF token, EOF char in alphabet
Language description:
consider a language consisting of two statements
- assignment statements: ID = expr
- increment statements: ID += expr
where expr is of the form:
- ID + ID
- ID ^ ID
- ID < ID
- ID <= ID
and 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>&quot; = &quot;</td>
</tr>
<tr>
<td>INCR</td>
<td>&quot; += &quot;</td>
</tr>
<tr>
<td>PLUS</td>
<td>&quot; + &quot;</td>
</tr>
<tr>
<td>EXP</td>
<td>&quot; ^ &quot;</td>
</tr>
<tr>
<td>LESSTHAN</td>
<td>&quot; &lt; &quot;</td>
</tr>
<tr>
<td>LEQ</td>
<td>&quot; &lt;= &quot;</td>
</tr>
<tr>
<td>ID</td>
<td>&quot;letter</td>
</tr>
</tbody>
</table>

Combined DFA
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
(non-procedural)
- you don’t tell JLex how to scan / how to match tokens
- you tell JLex 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
// This file contains a complete JLex specification for a very small example.

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

%% directives
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

%%

regex rules

_photosynthesis_((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"); }
Regular expression rules section

Format:  \texttt{<regex>\{code\}} where \texttt{<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:  \(|\) \(\ast\) \(\oplus\) \(?\) \(\) \(\) \(\) \(\) \(\) \(\) for grouping

Character class operators:

<table>
<thead>
<tr>
<th>denoted</th>
<th>[</th>
<th>]</th>
<th>^</th>
<th>|</th>
</tr>
</thead>
<tbody>
<tr>
<td>matches 1 char</td>
<td>range</td>
<td>not</td>
<td>escape</td>
<td></td>
</tr>
</tbody>
</table>

Using scanner generated by JLex in a program

```java
// 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);
}
```