RegExps & DFAs

CS 536
Pre-class warm up

Write the regexp for Fortran real literals

An optional sign (‘+’ or ‘-‘)

An integer or:

1 or more digits followed by a ‘.’ followed by 0 or more digits

or: A ‘.’ followed by one or more digits

(‘+’|’-’|ε)(digit+(‘.’|ε))| (digit*’.’digit+))
Last time

Explored NFAs

for every NFA there is an equivalent DFA

epsilon edges add no expressive power

Introduce regular languages / expressions
Today

Convert regexps to DFAS

From language recognizers to tokenizers
Regexp to NFAs

Literals/epsilon correspond to simple DFAs

Operators correspond to methods of joining DFAs

$x^n$, where $n$ is even or divisible by 3
Regexp to NFA rules

Rules for operands

Literal ‘a’

\[ \varepsilon \]

\[ a \]
Regexp to NFA rules

Rules for alternation A|B

Make new start state $q'$ and new final state $f'$

Make original final states non-final

Add to $\delta$:

$q', \varepsilon \rightarrow q_A$
$q', \varepsilon \rightarrow q_B$
$F_a, \varepsilon \rightarrow f'$
$F_b, \varepsilon \rightarrow f'$
Regexp to NFA rules

Rule for catenation $A \cdot B$

Make new start state $q'$ and new final state $f'$

Make original final states non-final

Add to $\delta$:

$q', \varepsilon \rightarrow q_A$
$f_A, \varepsilon \rightarrow q_B$
$f_b, \varepsilon \rightarrow f'$
Regexp to NFA rules

Rule for iteration $A^*$

Make new start state $q'$ and new final state $f'$

Make original final states non-final

Add to $\delta$:  
$q', \varepsilon \rightarrow q_A$  
$q', \varepsilon \rightarrow f'$  
$f', \varepsilon \rightarrow q_A$
# Regexp operator precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Analogous math operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>low</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>multiplication</td>
</tr>
<tr>
<td>*</td>
<td>high</td>
<td>exponentiation</td>
</tr>
</tbody>
</table>
Tree representation of a regexp

\[(\text{letter } | \_)(\text{letter } | \_ | \text{digit})^*\]

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>_</td>
<td>low</td>
</tr>
<tr>
<td>_</td>
<td>medium</td>
</tr>
<tr>
<td>*</td>
<td>high</td>
</tr>
</tbody>
</table>
Bottom-up conversion
Bottom-up conversion
Bottom-up conversion
Bottom-up conversion
Bottom-up conversion

(\texttt{letter} \mid \texttt{‘} \texttt{‘})

letter

\texttt{ε} \rightarrow \texttt{letter} \rightarrow \texttt{ε}

\texttt{ε} \rightarrow \texttt{‘} \rightarrow \texttt{ε}

\texttt{letter}

\texttt{‘} \rightarrow \texttt{digit}

\texttt{*}
Bottom-up conversion
Bottom-up conversion

\[(\text{letter} \mid \_\_\_) \Rightarrow^* (\text{letter} \mid \_\_\_\_ \mid \text{digit})\]
Bottom-up conversion

\[
(letter | \_')
\]

\[
(letter | \_')^* \hspace{2cm} (letter | \_')^* \hspace{2cm} (letter | \_')^* \hspace{2cm} (letter | \_')^*
\]
Bottom-up conversion

\[(\text{letter} \mid \text{'_'})(\text{letter} \mid \text{'_'} \mid \text{digit}^*)\]
Regexp to DFAs

We now have an NFA

We need to go to DFA

But what’s so great about DFAs?
Table-driven DFAs

Recall that $\delta$ can be expressed as a table

This leads to a very efficient array representation

$$
\begin{array}{c|c|c}
& a & b & c \\
\hline
s_1 & s_2 & s_2 \\
\hline
s_2 & s_1 & s_2 \\
\end{array}
$$

s = start state
while (more input){
    c = read char
    s = table[s][c]
}
if s is final, accept
FSMs for tokenization

FSMs only check for language membership of a string

the scanner needs to recognize a stream of many different tokens using the longest match

the scanner needs to know what was matched

Idea: imbue states with actions that will fire when state is reached
A first cut at actions

Consider the language of Pascal identifiers

Accounting for longest matches

BAD: not longest match

BAD: maybe we needed that character
A second take at actions

Give our FSMs ability to put chars back

Since we’re allowing our FSM to peek at characters past the end of a valid token, it’s also convenient to add an EOF symbol
Our first scanner

Consider a language with two statements

assignments: \( ID = expr \)

increments: \( ID += expr \)

where \( expr \) is of the form

\( ID + ID \)
\( ID ^ ID \)
\( ID < ID \)
\( ID <= ID \)

Identifiers \( ID \) follow C conventions

<table>
<thead>
<tr>
<th>Token name</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>INC</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>LT</td>
<td>&quot;&lt;&quot;</td>
</tr>
<tr>
<td>LEQ</td>
<td>&quot;&lt;=&quot;</td>
</tr>
<tr>
<td>ID</td>
<td>(letter</td>
</tr>
</tbody>
</table>
Combined DFA

```
<table>
<thead>
<tr>
<th>Token name</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>&quot;=&quot;</td>
</tr>
<tr>
<td>INC</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>LT</td>
<td>&quot;&lt;&quot;</td>
</tr>
<tr>
<td>LEQ</td>
<td>&quot;&lt;=&quot;</td>
</tr>
<tr>
<td>ID</td>
<td>(letter</td>
</tr>
</tbody>
</table>
```

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>return ASSIGN</td>
</tr>
<tr>
<td>S2</td>
<td>return EXP</td>
</tr>
<tr>
<td>S3</td>
<td>return INC</td>
</tr>
<tr>
<td>S4</td>
<td>put back 1 char, return PLUS</td>
</tr>
<tr>
<td>S5</td>
<td>Return LEQ</td>
</tr>
<tr>
<td>S6</td>
<td>put back 1 char, return LT</td>
</tr>
<tr>
<td>S7</td>
<td>put back 1 char, return ID</td>
</tr>
<tr>
<td>=</td>
<td>+</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>$S_0$</td>
<td>Ret</td>
</tr>
<tr>
<td>$S_A$</td>
<td>Ret</td>
</tr>
<tr>
<td>$S_B$</td>
<td>Ret</td>
</tr>
<tr>
<td>$S_C$</td>
<td>Back 1, Ret ID</td>
</tr>
</tbody>
</table>

Diagram:

- $s_0$: letter, digit
- $s_1$: "="
- $s_2$: "+
- $s_3$: "=
- $s_4$: "=
- $s_5$: "=
- $s_6$: not "="
- $s_7$: not ("_", letter, digit)
- $s_C$: letter,
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

The transformation from regexp to scanner is formally defined

Can write tools to synthesize a lexer automatically

Lex: unix scanner generator

Flex: fast lex

JLex: Java version of Lex
JLex

Declarative specification

tell it what you want scanned, it will figure out the rest

**Input**: set of regexps + associated actions

xyz.jlex file

**Output**: Java source code for a scanner

xyz.jlex.java  source code of scanner
jlex format

3 sections separated by %%

user code section

directives

regular expressions + actions
//User Code Section (uninterpreted java code)

%

//Directives Section

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

%state SPECIALINTSTATE — State declaration

//Configure for use with java CUP (Parser generator)
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol

//End of file behavior
%eofval{
System.out.println("All done");
return null;
%eofval}

//Turn on line counting
%line

%

//Regular Expression rules
Rules section

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

- can use macros from the directive sections in regex, surround with curly braces

Conventions

- chars represent themselves (except special characters)
- chars inside "" represent themselves (except \)

Regexp operators

- | * + ? () .

Character class operators

- - range
- ^ not
- \ escape
`="
{ System.out.println(yyline + 1 + "": ASSIGN"); }
`="
{ System.out.println(yyline + 1 + "": PLUS"); }
`="
{ System.out.println(yyline + 1 + "": EXP"); }
`="
{ System.out.println(yyline + 1 + "": LT"); }
`="
{ System.out.println(yyline + 1 + "": INC"); }
`="
{ System.out.println(yyline + 1 + "": LEQ"); }

{WHITESPACE} { }

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