Announcements

• Because many students were on the waitlist, your two allowed late dates start with P2

• P1, Part 1 due 11 PM, Friday, 1/27/17
  • If still not yet enrolled, e-mail your solution (as a zip file) to TA Shixuan Fan (shixuan.fan@wisc.edu)

• Class resources:
  • Home page: http://pages.cs.wisc.edu/~cs536-1/
  • Piazza: https://piazza.com/class/ixjghrokh7566jb?cid=1
  • Links to both at Learn@UW: https://uwmad.courses.wisconsin.edu/d2l/home/3600601
Clarifications

- Mathematical notation for regexps: |, ·, and *
- Notation for regexps in tools and programming languages:
  . (period) match a single character, except end of line plus other notational conventions
  Often have extensions such as “beginning of line,” “groups,” “intervals,” etc.
RegExps & DFAs

CS 536
Pre-Lecture Warm Up

Write the regular expression 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

\( (‘+’|’-’|\epsilon)(\text{digit+}(‘.’|\epsilon)) \)

\| (\text{digit*’.’digit+})
Assignments

P1 assigned

Part 1 due on the 25th of January

Part 2 due on the 1st of February
Last Time

Explored NFAs

$\epsilon$-edges add no expressive power

for every NFA there is an equivalent DFA

Introduced regular languages / expressions
Today

Convert regexps to DFAS

From language recognizers to tokenizers
Regular Expression to NFA

Idea:

• Occurrences of literals or $\epsilon$ correspond to simple DFAs
• Operators (i.e., ‘|’, ‘.’, ‘*’) correspond to methods of joining FSMs

Joining FSMs:
Consider an NFA for the language $x^n$, where $n$ is even or divisible by 3
Regexp to NFA Rules

Rules for operands

Literal ‘a’

ε
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.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 $A^*$
## 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

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

Operator | Precedence
--- | ---
\* | high
. | medium
| | low
Bottom-Up Conversion
Bottom-Up Conversion
Bottom-Up Conversion
Bottom-Up Conversion

\[
\text{(letter | ' ')}
\]

\[
\begin{array}{c}
\text{letter} \\
\varepsilon \\
\varepsilon \\
\varepsilon \\
\varepsilon \\
\varepsilon
\end{array}
\]

\[
\text{digit}
\]
Bottom-Up Conversion

\[(\text{letter} \mid \text{`}\text{'}\)\]
Bottom-Up Conversion
Bottom-Up Conversion

(letter | ‘_’) * (letter|‘_’|digit)
Bottom-Up Conversion

\[(\text{letter} \mid \text{‘_’})\]

\[(\text{letter} \mid \text{‘_’} \mid \text{digit})^*\]
Bottom-Up Conversion

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

\[s = \text{start state} \]

\[\text{while (more input)}\{\]

\[c = \text{read char}\]

\[s = \text{table}[s][c]\]

\[\}\]

if $s$ is final, accept.
Using FSMs for Tokenization

Problem specification:
Given a collection of token definitions (i.e., one FSM for each kind of token), create a maximal-munch tokenizer.

FSMs only check language membership: Is string \( \omega \in L(M) \)?

The scanner needs more:

- It needs to break up the input into tokens a stream of many different tokens.
- Successively finds the next token by a “maximal munch”:
  - the longest prefix of the remaining input that corresponds to a token and returns information about what was matched.

Idea: Equip states with actions that will fire when the 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 Cut at Actions

Give our FSMs ability to put chars back

Because 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 kinds of statements

assignments: \( \text{ID} = \text{expr} \)

increments: \( \text{ID} += \text{expr} \)

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

\[
\text{ID} + \text{ID} \\
\text{ID} \land \text{ID} \\
\text{ID} < \text{ID} \\
\text{ID} \leq \text{ID}
\]

Identifiers \( \text{ID} \) follow C’s convention

<table>
<thead>
<tr>
<th>Token name</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>“=“</td>
</tr>
<tr>
<td>INC</td>
<td>“+=“</td>
</tr>
<tr>
<td>PLUS</td>
<td>“+“</td>
</tr>
<tr>
<td>EXP</td>
<td>“^”</td>
</tr>
<tr>
<td>LT</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

---

**Token name** | **Regular Expression**
--- | ---
ASSIGN | “=“
INC | “+=“
PLUS | “+“
EXP | “^“
LT | “<“
LEQ | “<=“
ID | (letter|_)(letter|digit|_)*

**State** | **Action**
--- | ---
S1 | return ASSIGN
S2 | return EXP
S3 | return INC
S4 | put back 1 char, return PLUS
S5 | Return LEQ
S6 | put back 1 char, return LT
S7 | put back 1 char, return ID
<table>
<thead>
<tr>
<th></th>
<th>=</th>
<th>+</th>
<th>^</th>
<th>&lt;</th>
<th>-</th>
<th>letter</th>
<th>digit</th>
<th>EOF</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>Ret ASSIGN</td>
<td>$S_A$</td>
<td>Ret EXP</td>
<td>$S_B$</td>
<td>$S_C$</td>
<td>$S_C$</td>
<td></td>
<td>Ret EOF</td>
<td></td>
</tr>
<tr>
<td>$S_A$</td>
<td>Ret INC, Ret PLUS</td>
<td>Back 1, Ret PLUS, Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
<td>Back 1, Ret PLUS</td>
</tr>
<tr>
<td>$S_B$</td>
<td>Ret LEQ, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
<td>Back 1, Ret LT</td>
</tr>
<tr>
<td>$S_C$</td>
<td>Back 1, Ret ID</td>
<td>Back 1, Ret ID</td>
<td>Back 1, Ret ID</td>
<td>Back 1, Ret ID</td>
<td>Back 1, Ret ID</td>
<td>$S_C$</td>
<td>$S_C$</td>
<td>Back 1, Ret ID</td>
<td>Back 1, Ret ID</td>
</tr>
</tbody>
</table>

![Diagram](image_url)
### tokenize_input

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>^</th>
<th>&lt;</th>
<th>–</th>
<th>letter</th>
<th>digit</th>
<th>EOF</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>ASSIGN</td>
<td>$S_A$</td>
<td>EXP</td>
<td>$S_B$</td>
<td>$S_C$</td>
<td>$S_C$</td>
<td>Ret EOF</td>
<td></td>
</tr>
<tr>
<td>$S_A$</td>
<td>INC</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
<td>Back 1, PLUS</td>
</tr>
<tr>
<td>$S_B$</td>
<td>LEQ</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
<td>Back 1, LT</td>
</tr>
<tr>
<td>$S_C$</td>
<td>Back 1, ID</td>
<td>Back 1, ID</td>
<td>Back 1, ID</td>
<td>Back 1, ID</td>
<td>$S_C$</td>
<td>$S_C$</td>
<td>Back 1, ID</td>
<td>Back 1, ID</td>
</tr>
</tbody>
</table>

```plaintext
tokenize_input() {
    s = start state
    do {
        c = read char
        s = table[s][c]
        perform action
        if (action was to Ret a token) {
            s = start state
        }
    } (while not EOF or stuck);
}
```

### get_next_token

```plaintext
get_next_token() {
    s = start state
    do {
        c = read char
        s = table[s][c]
        perform action
    } (while not EOF or stuck);
}
```

A Ret action causes `tokenize_input` to emit the token name.

A Ret action causes `get_next_token` to actually return.
Lexical-Analyzer Generators (a.k.a. scanner generators)

The transformation from a set of regexps – one for each kind of token – to the scanner is precisely defined:

- Convert the set of regexps to a DFA with actions
- Scanner = the DFA + get_next_token()

Can write tools to synthesize a scanner automatically:

Lex: original scanner generator available on Unix
Flex: fast lex
JLex: Java version of Lex
Drawbacks of the Algorithm

- What about overlap in tokens?
  - = vs. ==
  - + vs. + =
  - Keyword “for” vs. identifier “formula”

The scanner must know how to resolve the ambiguity.

- These examples above all handled correctly (why?)

- OK for the kind of overlap that arises in the syntax for typical programming languages. However, in general, there can be a problem. Consider

  Token 1: abc
  Token 2: (abc)*d
  Input: abcabcabc
A Better Algorithm

• In general, there can be a problem. Consider
  Token1: abc  Token2: (abc)*d  Input string: abcabcabcabc

• Goal: abcabcabcabc should be tokenized as Token1 Token1 Token1

• Idea

  • Use one DFA for each kind of token (e.g., M₁ for abc and M₂ for (abc)*d)
  • Start running all DFAs simultaneously on the remaining input
  • A DFA drops out when it enters a stuck state (i.e., has no available transition on the next input character)
  • Update most_recent_accepted_position and most_recent_accepted_token whenever any machine enters a final state. (Break ties by assigning some order to the DFAs.)
  • When the last DFA drops out, return most_recent_accepted_token (or FAIL, if most_recent_accepted_token never set). For finding the next token, the remaining input starts at most_recent_accepted_position

Using the most-recent accepted position ⇒ the longest token is identified ⇒ maximal munch
Example

- Token1: abc  Token2: (abc)*d  Input string: abcabcabc

- Consume first abc
  - Machines for both Token1 (M1) and Token2 (M2) are still in play
  - M1 is in its accepting state

- On next “a,” M1 drops out; M2 in in play, but is not in an accepting state

- After next “bcabc,” M2 drops out, but never entered its accepting state

- Token1 returned; remaining input (= abcabc) handled similarly

- Result: abcabcabc is tokenized as Token1 Token1 Token1

- Drawback: Cost = $O(n^2)$

Another Variant, Using One DFA

Token definitions: $R_1, R_2, \ldots, R_k$

Given: DFA $M$ such that $L(M) = L(R_1|R_2| \ldots |R_k)$

```
Tokenize(M: DFA, input: string)
let \( \langle Q, \Sigma, \delta, q_0, F \rangle = M \) in
begin
  i = 0;
  forever {
    q = q_0; mrap = -1; mrat = -1;
    while (i < length(input)) {
      q = \delta(q, input[i]);
      i = i + 1;
      if (q \in F) { mrap = i; mrat = tokenFor(q); }
    }
    if (mrap == -1) return 'Failure'
    i = mrap;
    print(mrat);
    print(mrat);
    if (i \geq length(input)) return 'Success'
  }
End<
Declarative specification

tell it what you want scanned, it will generate the res

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

xyz.jlex file

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

xyz.jlex.java source code of scanner

An action in a JLex specification is simpler than the action of the generated DFA: in particular, one does not specify putting back characters
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"); }