CS 536 Announcements for Thursday, January 27, 2022

Course websites:

pages.cs.wisc.edu/~hasti/cs536/
www.piazza.com/wisc/spring2022/compsci536

- waitlisted folks: feel free to add yourself to Piazza

Programming Assignment 1
- released tomorrow (Friday, Jan. 28)
- test code due Friday, Feb. 4 by 11:59 pm
- other files due Tuesday, Feb. 8 by 11:59 pm

Last Time
- intro to CS 536
- compiler overview

Today
- start scanning
- finite state machines
  - formalizing finite state machines
  - coding finite state machines
  - deterministic vs non-deterministic FSMs
A compiler is
- recognizer of language $S$
- a translator from $S$ to $T$
- a program in language $H$

**Example:** for gcc, $S$ is C, $T$ is x86, $H$ is C

**Why do we need a compiler?**
- processors can execute only binaries (machine-code/assembly programs)
- writing assembly programs will make you lose your mind
- allows you to write programs in nice(ish) high-level languages like C; compile to binaries

**front end** = understand source code $S$; map $S$ to $IR$

**IR** = intermediate representation

**back end** = map $IR$ to $T$
Overview of typical compiler

Source program

Scanner (lexical analyzer)
sequence of tokens (one at a time)

Parser (syntax analyzer)
AST (abstract syntax tree)
name analysis P4

Semantic analyzer
augmented, annotated AST

Intermediate code generator

Optimizer
optimized IR

Code generator
assembly or machine code or target code

Object program

Symbol table

front end

back end
Special linkage between scanner and parser (in most compilers)

**Conceptual organization**

**Scanning**

Scanner translates sequence of chars into sequence of tokens

Each time scanner is called it should:
- find longest sequence of chars corresponding to a token
- return that token

**Scanner generator**

- **Inputs:**
  - one regular expression for each token
  - one regular expression for each item to ignore (comments, whitespace, etc.)
- **Output:** scanner program

To understand how a scanner generator works, we need to understand FSMs
Finite-state machines
(aka finite automata, finite-state automata)

- **Inputs**: string (sequence of characters)
- **Output**: accept / reject

Language defined by an FSM = the set of strings accepted by the FSM

---

**Example 1:**

**Language**: single-line comments starting with //  (in Java / C++)

Nodes are states
Edges are transitions
Start state has arrow point to it
Final states are double circles
How a finite state machine works

curr_state = start_state
let in_ch= current input character
repeat
  if there is edge out of curr_state with
  label in_ch into next_state
  curr_state = next_state
  in_ch = next char of input
  otherwise
  stuck // error condition
until stuck or input string is consumed
if entire string is consumed and
  curr_state is a final state
accept string
otherwise
reject string

Formalizing finite-state machines

alphabet ($\Sigma$) = finite, non-empty set of elements called symbols
string over $\Sigma$ = finite sequence of symbols from $\Sigma$
language over $\Sigma$ = set of strings over $\Sigma$
finite state machine $M = (Q, \Sigma, \delta, q, F)$ where
  $Q$ = set of states
  $\Sigma$ = alphabet
  $\delta$ = state transition function $Q \times \Sigma \rightarrow Q$
  $q$ = start state
  $F$ = set of accepting (or final) states
$L(M)$ = the language of FSM $M$ = set of all strings $M$ accepts

finite automata $M$ accepts $x = x_1x_2x_3...x_n$ iff
Example 2: hexadecimal integer literals in Java

Hexadecimal integer literals in Java:
- must start 0x or 0X
- followed by at least one hexadecimal digit (hexdigit)
  - hexdigit = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E, F
- optionally can add long specifier (l or L) at end

State transition table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1 – 9</th>
<th>a – f</th>
<th>A – F</th>
<th>x</th>
<th>X</th>
<th>l</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Se</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coding a state transition table

curr_state = start_state
done = false

while (!done)
    ch = nextChar()
    next = transition[curr_state][ch]
    if (next == error || ch == EOF)
        done = true
    else
        curr_state = next

return final_states.contains(curr_state) && next != error

Deterministic vs non-deterministic FSMs

deterministic

- no state has >1 outgoing edge with same label

non-deterministic

- states may have multiple outgoing edges with same label
- edges may be labelled with special symbol ε (empty string)

ε-transitions can happen without reading input
Example 2 (revisited): hexadecimal integer literals in Java

Example 3: FSM to recognize keywords for, if, int

Example 4: identifiers in C/C++

A C/C++ identifier
- is a sequence of one or more letters, digits, underscores
- cannot start with a digit