CS 536 Announcements for Thursday, January 27, 2022

Course websites:

    pages.cs.wisc.edu/~hasti/cs536/
    www.piazza.com/wisc/spring2022/comspci536

- 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
Recall

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 → sequence of characters → Scanner → sequence of tokens → Parser → AST (abstract syntax tree) → Semantic analyzer → augmented, annotated AST → Intermediate code generator → IR → Optimizer → optimized IR → Code generator → assembly or machine code → Object program

1. Symbol table
2. Scanner
3. Parser
4. Name analysis
5. Type checking
6. Back end

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

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) - finite length
- **Output**: accept / reject - is string in the language \( L \)

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

Example 1:

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

```
// stuff to end of line
int x = 5;
Not(\n)
```

Nodes are **states**

Edges are **transitions**

Start state has arrow point to it

Final states are double circles

Consider

```
// red
// green
// blue
// cyan
```

```
\n
// \n
\n
```

```
\n
```

```
```

Can have multiple
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 — follow transition
  in_ch = next char of input — consume character
  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 (Σ) = finite, non-empty set of elements called symbols
string over Σ = finite sequence of symbols from Σ
language over Σ = set of strings over Σ

finite state machine \( M = (Q, Σ, δ, q, F) \) where

- \( Q \) — set of states — finite
- \( Σ \) — alphabet — finite union of all edge labels
- \( δ \) — state transition function \( Q \times Σ \rightarrow Q \) given (state, symbol), return state
- \( q \) — start state — only 1, \( q \in Q \)
- \( F \) — set of accepting (or final) states —\( F \subseteq Q \)

L(\( M \)) = the language of FSM \( M \) = set of all strings \( M \) accepts — can be infinite

finite automata \( M \) accepts \( x = x_1x_2x_3...x_n \) iff

\( \delta(\delta(...) \delta(\delta(q, x_1), x_2), x_3) ... , x_{n-1}), x_n) \in F \)

ie, end up in final state
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

|       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | A | B | C | D | E | F | x | X | l | L |
| S0    | S1 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 | S2 |
| S1    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| S2    | S3 | S4 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 | S3 |
| S3    | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 | S4 |
| S4    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Se    | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se | Se |

To handle empty spaces, create error state Se.
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

Works provided FSM is deterministic

---

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 $\varepsilon$ (empty string)

$\varepsilon$ -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

Add restriction: can’t end in underscore