Finite-State Machines (FSMs)

CS 536
Last time

A compiler is a
recognizer of language S (Source)
a translator from S to T (Target)
a program in language H (Host)

For example, 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 want to kill yourself
- Write programs in a nice(ish) high-level language like C; compile to binaries
Last time

front end = understand source code S
IR = intermediate representation
back end = map IR to T
Last time

Source Program
  └ Sequence of characters

  lexical analyzer (scanner)
    └ Sequence of tokens

  syntax analyzer (parser)
    └ Abstract-syntax tree (AST)

  semantic analyzer
    └ Augmented, annotated AST

  intermediate code generator
    └ Intermediate code

  optimizer
    └ Optimized intermediate code

  code generator
    └ Assembly or machine code

  object program

P1

Symbol table

P2

P3

P4, P5

P6

front end

back end
Special linkage between scanner and parser in most compilers

Conceptual organization

Source Program

lexical analyzer (scanner)

Sequence of characters

syntax analyzer (parser)

Sequence of tokens

Syntax analyzer (parser)

next token, please

lexical analyzer (scanner)

Implementation: master/slave (or "coroutine")
The scanner

Translates sequence of chars into a sequence of tokens (ignoring whitespace)

\[ a = 2 \times b + \text{abs}(-71) \]

Each time the scanner is called it should:

• find the longest prefix (lexeme) of the remaining input that corresponds to a token

• return that token
How to create a scanner?

- For every possible lexeme that can occur in source program, return corresponding token
- Inefficient
- Error-prone
Scanner generator

• Generates a scanner

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

• **Output:** scanner program

• How does a scanner generator work?
  - Finite-state machines (FSMs)
FSMs: Finite State Machines

(A.k.a. finite automata, finite-state automata, etc.)

**Input:** string (sequence of chars)

**Output:** accept / reject

i.e., input is legal in language

Language defined by an FSM is the set of strings accepted by the FSM
Example 1

Language: single line comments with //

- Nodes are states
- Edges are transitions
- Start state has an arrow (only one start state)
- Final states are double circles (one or more)
Example 1

Language: single line comments with //

1. “// this is a comment.”
2. “/ / this is not.”
3. “// \n”
4. “Not // a comment”
Example 2

Language: Integer literals with an optional + or –
(token: int-lit)

e.g., -543, +15, 0007
FSMs, formally

\[ M \equiv (Q, \Sigma, \delta, q, F) \]

- finite set of the alphabet (characters)
- start state \( q \in Q \)
- final states \( F \subseteq Q \)
- transition function \( \delta : Q \times \Sigma \rightarrow Q \)

\[ L(M) = \text{set of integer literals} \]
FSM example, formally

\[ M \equiv (Q, \Sigma, \delta, q, F) \]

\[ Q = \{ s_0, s_1 \} \]
\[ \Sigma = \{ a, b, c \} \]
\[ q = s_0 \]
\[ F = \{ s_0 \} \]
\[ \delta = s_0, a \rightarrow s_1 \]
\[ s_1, b \rightarrow s_0 \]

\[ L(M) = \{ \varepsilon, ab, abab, ababab, abababab, \ldots \} \]

What is \( L(M) \)?

anything else, machine is stuck
Coding an FSM

curr_state = start_state

done = false

while (!done)

    ch = nextChar()

    next = table[curr_state][ch]

    if (next == stuck || ch == EOF)
        done = true

    else

        curr_state = next

return final_states.contains(curr_state) &&

    next!=stuck
FSM types: DFA & NFA

Deterministic
no state has >1 outgoing edge with same label

Nondeterministic
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
NFA Example

Language: Integer literals with an optional + or – (token: int-lit)

e.g., -543, +15, 0007

A string is accepted by an NFA if there exists a sequence of transitions leading to a final state
Why NFA?

Simpler and more intuitive than DFA

Language: sequence of 0s and 1s, ending with 00
A C/C++ identifier is a sequence of one or more letters, digits, or underscores. It cannot start with a digit.
Extra Example - Part 1

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

![Diagram showing the sequence of elements in an identifier: 1. Digit, letter, \('_\); 2. \('_\), letter.]
Extra example

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

What if you wanted to add the restriction that it can't end with an underscore?
What if you wanted to add the restriction that it can't end with an underscore?
Recap

The scanner reads a stream of characters and tokenizes it (i.e., finds tokens)

Tokens are defined using regular expressions, scanners are implemented using FSMs

FSMs can be non-deterministic

Next time: understand connection between DFA and NFA, regular languages and regular expressions
Play with automata!

automatatutor.com

Loris D’Antoni