CS 536 Announcements for Monday, January 29, 2024

Course websites:

`pages.cs.wisc.edu/~hasti/cs536`

`www.piazza.com/wisc/spring2024/compsci536`

Office hours

- Beck (in 5360 Comp Sci)
  - 2:00 – 3:00 pm Mondays
  - 9:00 – 10:30 am Tuesdays
  - 10:30 am – noon Fridays
- office hours for TAs are being determined

Programming Assignment 1

- test code due Sunday, Feb. 4 by 11:59 pm
- other files due Thursday, Feb. 8 by 11:59 pm

Reminders

- report exam conflicts using CS 536 Alternate Exam Request Form (link on Exam Information page)
- contact Beck within first 3 weeks of classes if
  - you participate in religious observances that may conflict with course requirements
  - you receive accommodations through the McBurney center

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

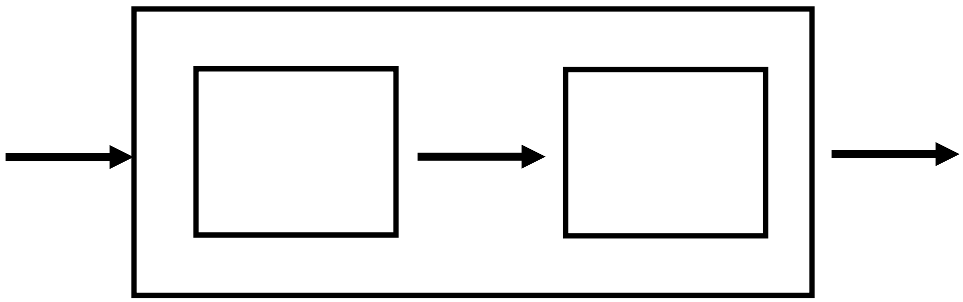
Next Time

- non-deterministic FSMs
- equivalence of NFAs and DFAs
- regular languages
- regular expressions

Recall

A compiler is

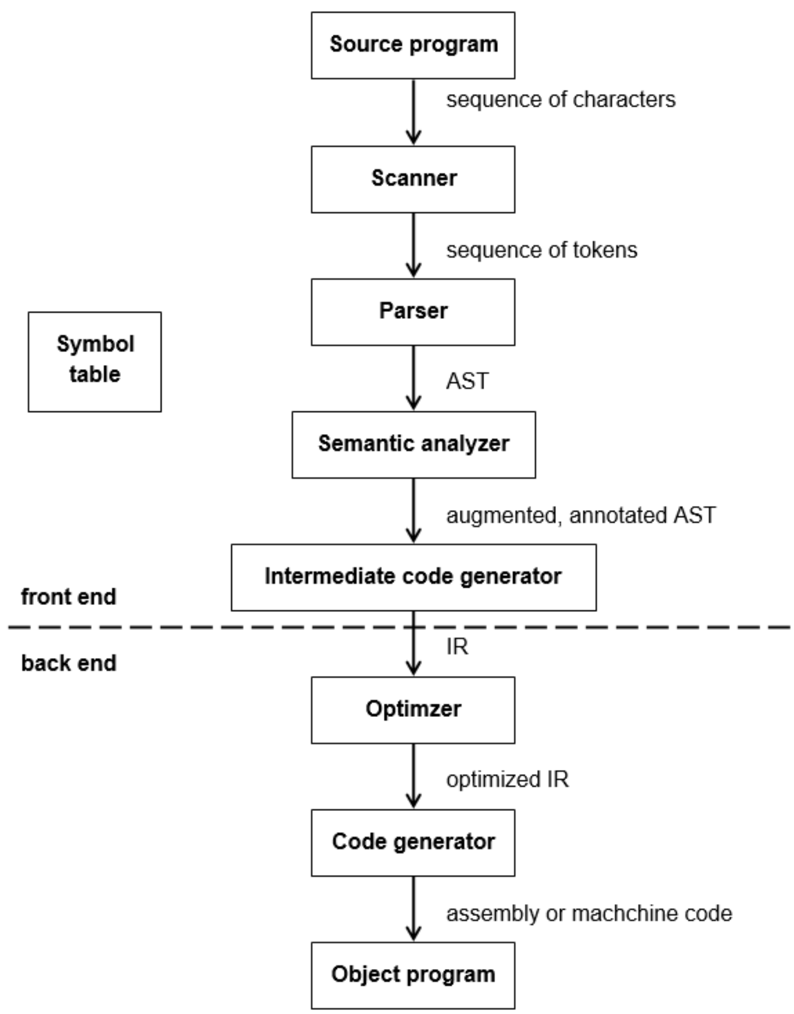
- recognizer of language S
- a translator from S to T
- a program in language H



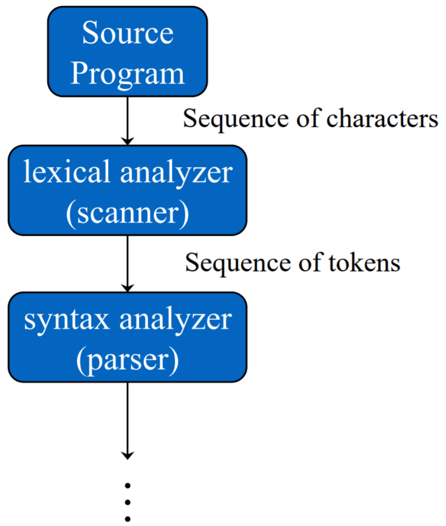
front end = understand source code S; map S to IR
IR = intermediate representation
back end = map IR to T

Why do we need a compiler?

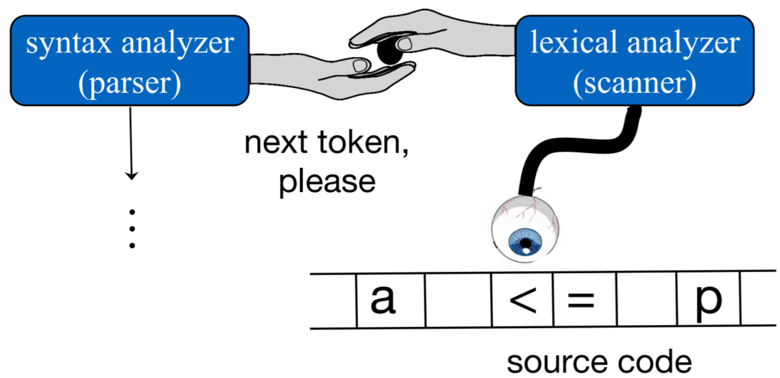
- processors can execute only binaries (machine-code/assembly programs)
- writing assembly programs will make you lose your mind
- allows you to write programs in nice(ish) high-level languages like C; compile to binaries



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 (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 $M = (Q, \Sigma, \delta, q, F)$ where

- Q = set of states
- Σ = alphabet
- δ = 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** $x = x_1 x_2 x_3 \ldots x_n$ iff

$$\delta(\delta(\delta(\ldots\ \delta(\delta(\delta(s_0, x_1), x_2), x_3), \ldots\ x_{n-2}), x_{n-1}), x_n)$$

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

Q =

Σ =

δ =

q =

F =

State transition table

	0	1 – 9	a – f	A – F	x	X	l	L
s_0								
s_1								
s_2								
s_3								
s_4								
s_e								

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
```

Example 3: identifiers in C/C++

A C/C++ identifier

- is a sequence of one or more letters, digits, underscores
- cannot start with a digit

Deterministic vs non-deterministic FSMs

deterministic

- no state has >1 outgoing edge with same label
- edges can only be labelled with elements of Σ

non-deterministic

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

$\boldsymbol{\varepsilon}$ -transitions can happen without reading input

Example 2 (revisited): hexadecimal integer literals in Java

Example 4: FSM to recognize keywords `for`, `if`, `int`

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 (deterministic) FSMs
- FSMs can be non-deterministic