Welcome to CS 536:
Introduction to Programming Languages and Compilers!

Instructor: Beck Hasti
- hasti@cs.wisc.edu
- Office hours to be determined

TAs
- Aaryan Patel
- Daniel Smedema ← Epic TA
- Jack Stanek
- Nick Boddy

Course websites:

`canvas.wisc.edu`

`www.piazza.com/wisc/spring2025/compsci536`

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

About the course

We will study compilers (& programming languages)

We will understand how they work

We will build a full compiler

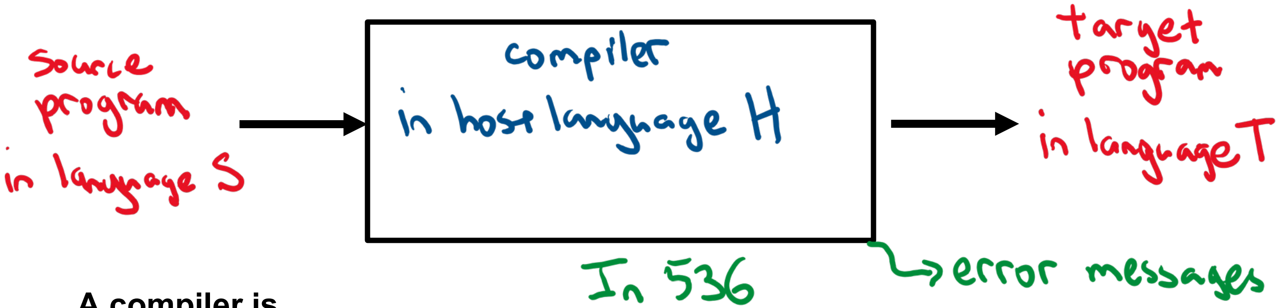
Course mechanics

Exams (60%)
- Midterm 1 (18%): Thursday, February 27, 6:30 – 8 pm
- Midterm 2 (16%): Thursday, March 20, 6:30 – 8 pm
- Final (26%): Thursday, May 8, 6:30 – 8:30 pm

Programming Assignments (40%)
- 6 programs: 5% + 7% + 7% + 7% + 7%+ 7%

Homework Assignments
- 8 short homeworks (optional, not graded)

What is a compiler?

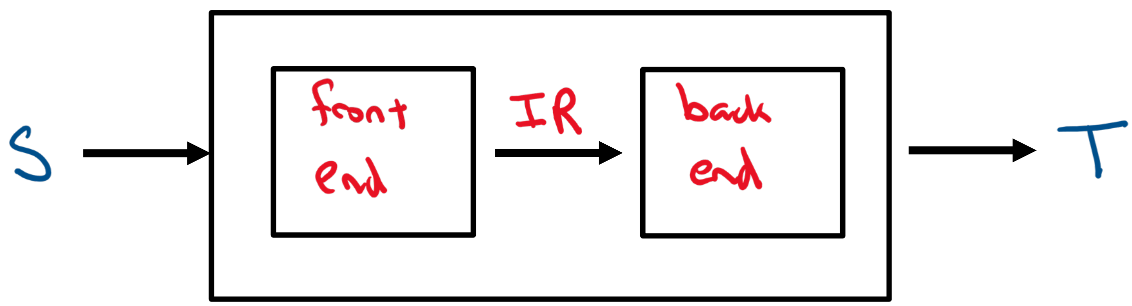


A compiler is

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

In 536
H: Java
S: bach
T: MIPS

Front end vs back end

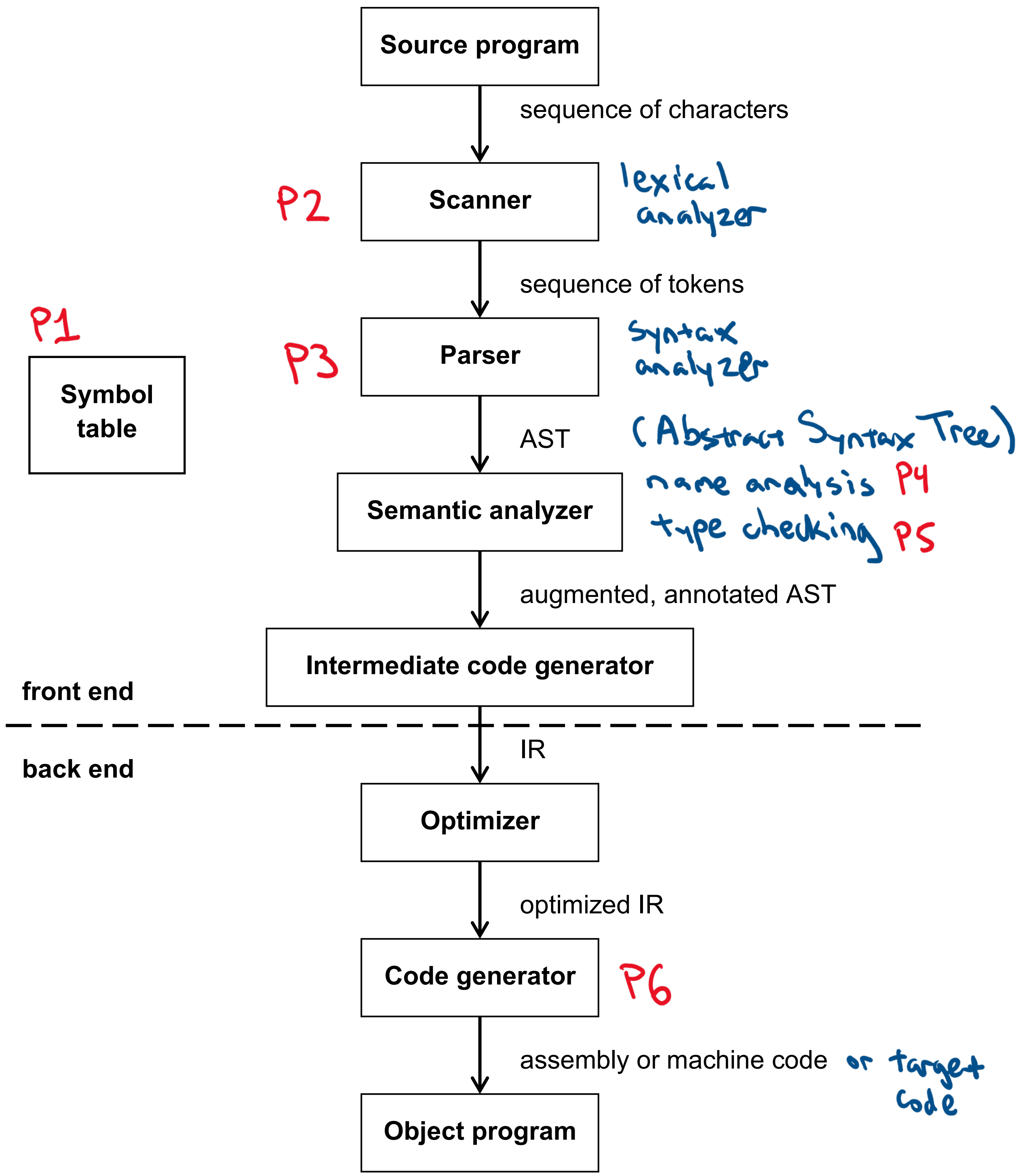


front end = understand source code S; map S to IR

IR = intermediate representation

back end = map IR to T

Overview of typical compiler



Scanner

Input: characters from source program

Output: sequence of tokens — possibly with associated info

Actions:

- group characters into lexemes (tokens)
- identify and ignore whitespace, comments, etc.

What errors can it catch?

- bad characters eg # in Java
- unterminated strings "Hello
- integer literals that are too large

Parser

Input: sequence of tokens from the scanner

Output: AST (abstract syntax tree)

Actions:

- group tokens into sentences

What errors can it catch?

- syntax errors x = y =* 5;
- (possibly) *static semantic* errors use of undeclared variables

Semantic analyzer

Input: AST

Output: annotated AST

Actions: does more static semantic checks

- Name analysis
  - process decls & uses of identifiers
  - match uses w/ decls
  - enforces scoping rules
  - errors – multiply-declared variables, uses of undeclared variables
- Type checking
  - check types & augment AST

Intermediate code generator

Input: annotated AST – assume no syntax/static semantic errors

Output: intermediate representation (IR)

eg 3-address code
- instructions have at most 3 operands
- easy to generate from AST
  - ↳ 1 instr per AST internal node

Example

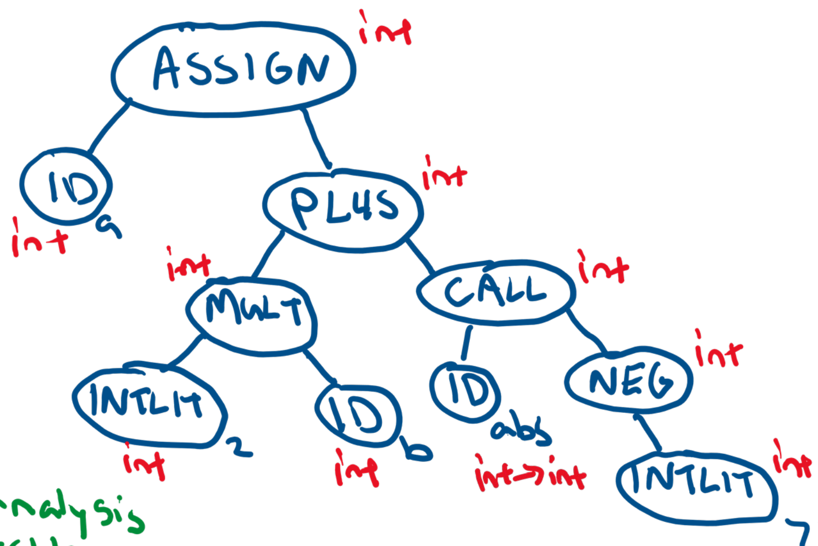
```
a = 2 * b + abs(-71);
```

Scanner produces tokens:

ID(a) ASSIGN INTLIT(2) TIMES ID(b) PLUS ID(abs) LPAREN MINUS INTLIT(71) RPAREN SEMICOLON

scanner doesn't know if unary or binary

AST (from parser)



Symbol table Name analysis gives us symbol table

ID	Kind	type
a	var	int
b	var	int
abs	fctn	int→int

3-address code

temp1 = 2 * b

temp2 = 0 - 71

move temp2 param1

call abs

move return1 temp3

temp4 = temp1 + temp3

a = temp4

} a = temp1 + temp3

Optimizer

Input: IR

Output: optimized IR

Actions: improve code

- make it run faster, make it smaller
- several passes: local and global optimization
- more time spent in compilation; less time in execution

local = look at a few instr at a time

global = look at entire fctn or whole program

Code generator

For 536 our IR is an AST

Input: IR from optimizer

Output: target code

Symbol Table

Compiler keeps track of names in

- semantic analyzer – both name analysis & type checking
- code generation – offsets into stack
- optimizer – could use to keep track of def-use info

P1 : implement symbol table

Block-structured language eg, Java, C, C++, bash

- nested visibility of names – no access outside of scope of name
- easy to tell which def of a name applies (usually nearest enclosing scope)
- lifetime of data is bound to scope of identifier that denotes it

Example: (from C)

```
int x, y;

void A() {
   double x, z;
   C(x, y, z);
}
void B(){
   C(x, y, z);
}
```

In A: x → double, y → int, z → double

In B: x → int, y → int, z → undeclared

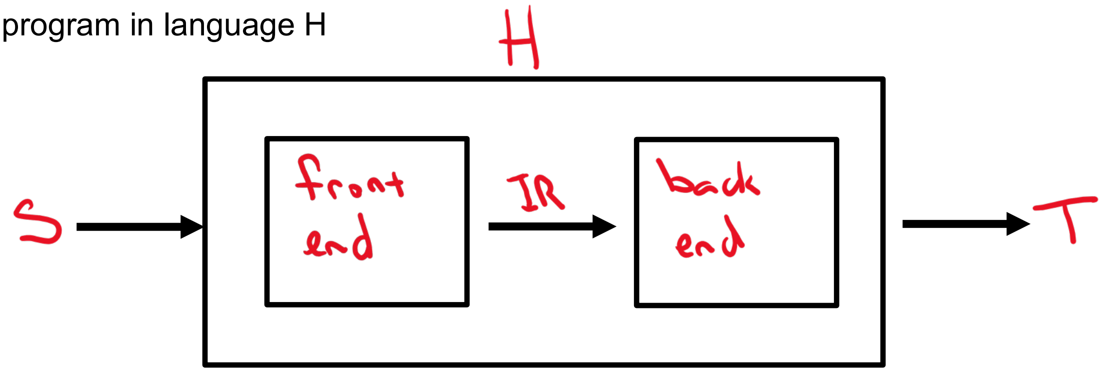
block structure ⇒ need nesting of sym tables

⇒ list of hashtables

Recall

A compiler is

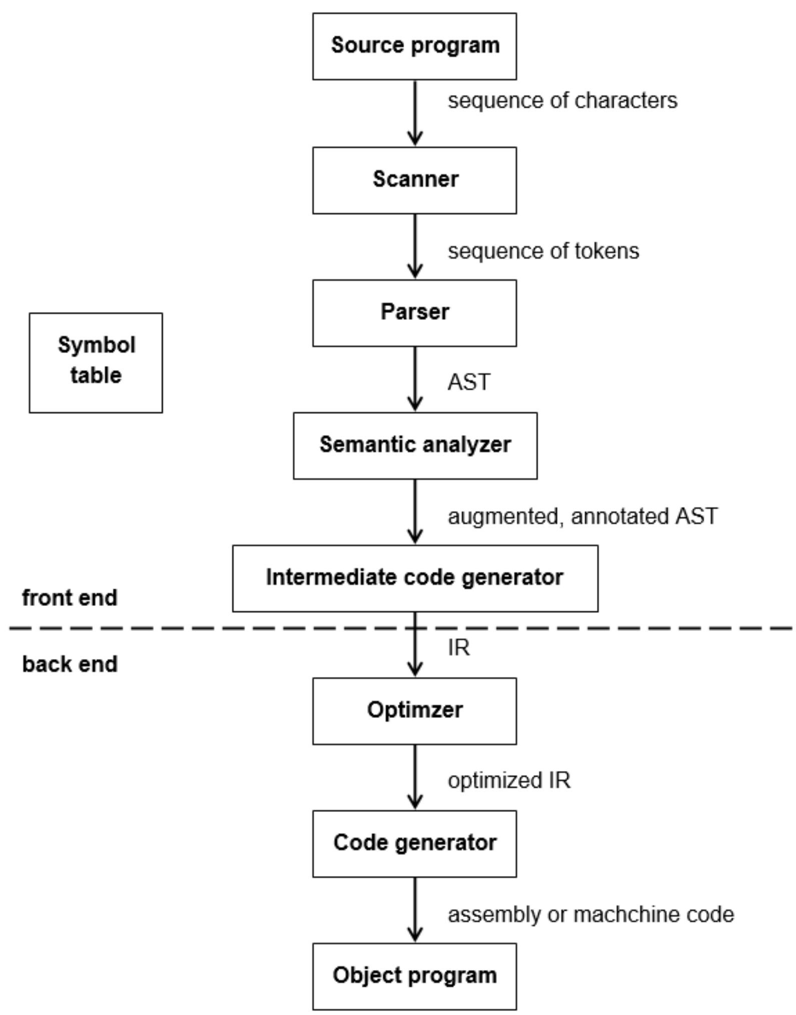
- recognizer of language S
- a translator from S to T
- a program in language H



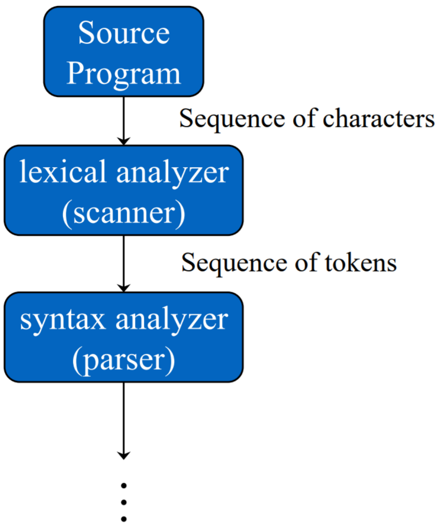
front end = understand source code S; map S to IR
IR = intermediate representation
back end = map IR to T

Why do we need a compiler?

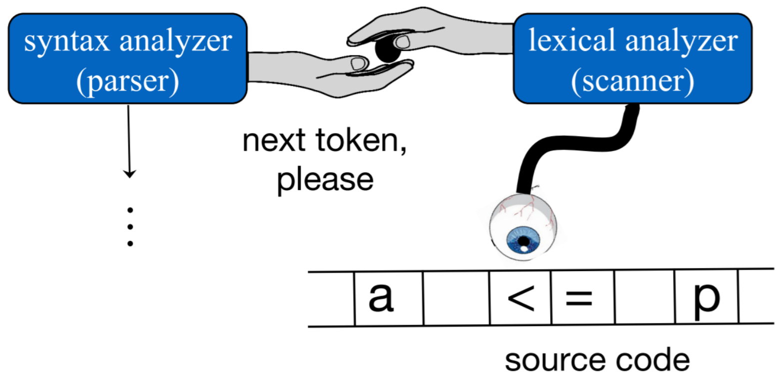
- processors can execute only binaries (machine-code/assembly programs)
- writing assembly programs will make you lose your mind
- allows you to write programs in nice(ish) high-level languages like C; compile to binaries



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

FA Finite-state machines FSM
(aka finite automata, finite-state automata)

- **Inputs**: string (sequence of characters) – finite length
- **Output**: accept / reject – is string in language L

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

Compiler recognizes legal programs in source lang S
FSM recognizes legal strings in some lang L

Example 1:

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

// stuff to end of line

shorthand for many transitions (1 for every char that is not \n)

Nodes are states

Edges are transitions

Start state has arrow point to it – only 1

Final states are double circles – can have multiple

Consider

//red\n ✓

//\n ✓

//cyan\nteal X stuck

//green X

//blue EOF X

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

    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 $Q \times \Sigma \to Q$ 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** $x = x_1 x_2 x_3 \ldots x_n$ iff

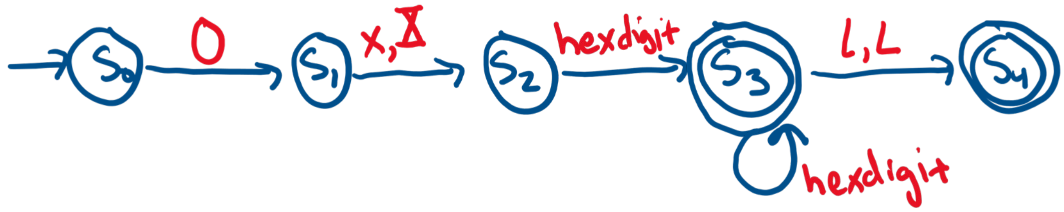
$$\delta(\delta(\delta(\ldots\ \delta(\delta(\delta(s_0, x_1), x_2), x_3), \ldots\ x_{n-2}), x_{n-1}), x_n) \in F$$

end in final state

Example 2: hexadecimal integer literals in Java

Hexadecimal integer literals in Java:

- must start 0x or 0X ⟵ number 0 (not letter capital-O)
- 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 = { s_0, s_1, s_2, s_3, s_4 }

Σ = { 0-9, a-f, A-F, x, X, l, L }

δ = use state transition table

q = s_0

F = { s_3, s_4 }

Example of accepted: 0x1f4d3

stuck in start: L

stuck in final state (not accepted): 0x7LL

State transition table

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

To handle empty spaces, create error state 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
```

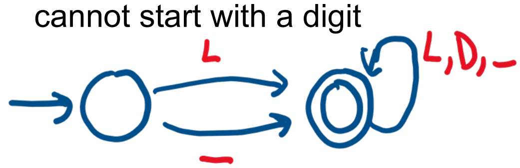
Works provided FSM is deterministic

Example 3: identifiers in C/C++

A C/C++ identifier

L D _

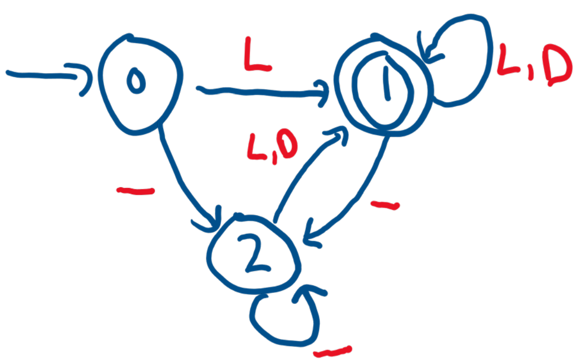
- is a sequence of one or more letters, digits, underscores
- cannot start with a digit



Legal but odd:

___ _0_

Add restriction: can't end in underscore



DFA NFA

Deterministic vs non-deterministic FSMs

deterministic

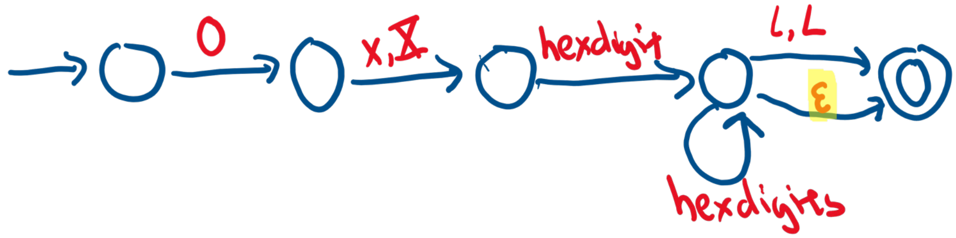
- no state has >1 outgoing edge with same label
- edges can only be labelled with elements of Σ

non-deterministic

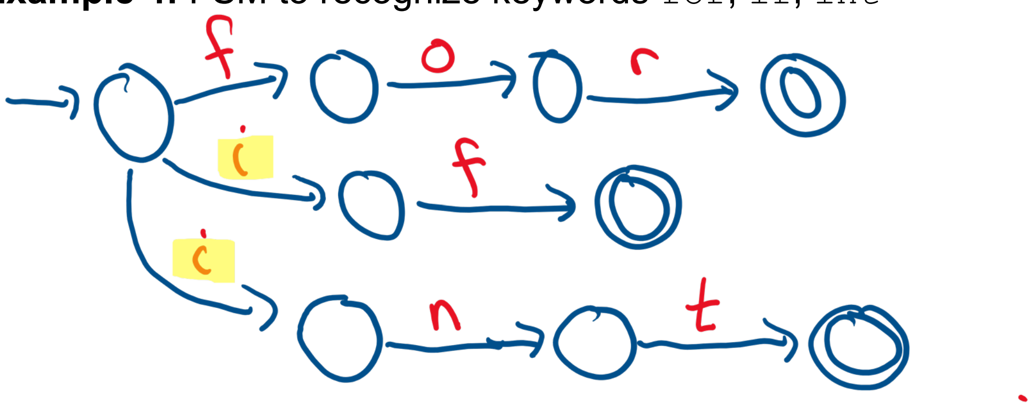
- states may have multiple outgoing edges with same label
- edges may be labelled with special symbol ε (empty string)

ε -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

Next time

- regular expressions
- understand the connections between
  - DFAs and NFAs
  - NFAs and regular expressions
- language recognition → tokenizers
- scanner generators
- JLex

Programming Assignment 1

- released tomorrow (Friday, Jan. 24)
- test code (part 1) due Sunday, Feb. 2 by 11:59 pm
- other files (part 2) due Thursday, Feb. 6 by 11:59 pm