Exam 1 Review

COMP360

Topics

• Language Paradigms	chapter 1
• Theory of language	section 2.4
• BNF	section 2.1
• Scanning	section 2.2
• Parsing	section 2.3

Questions on Anything

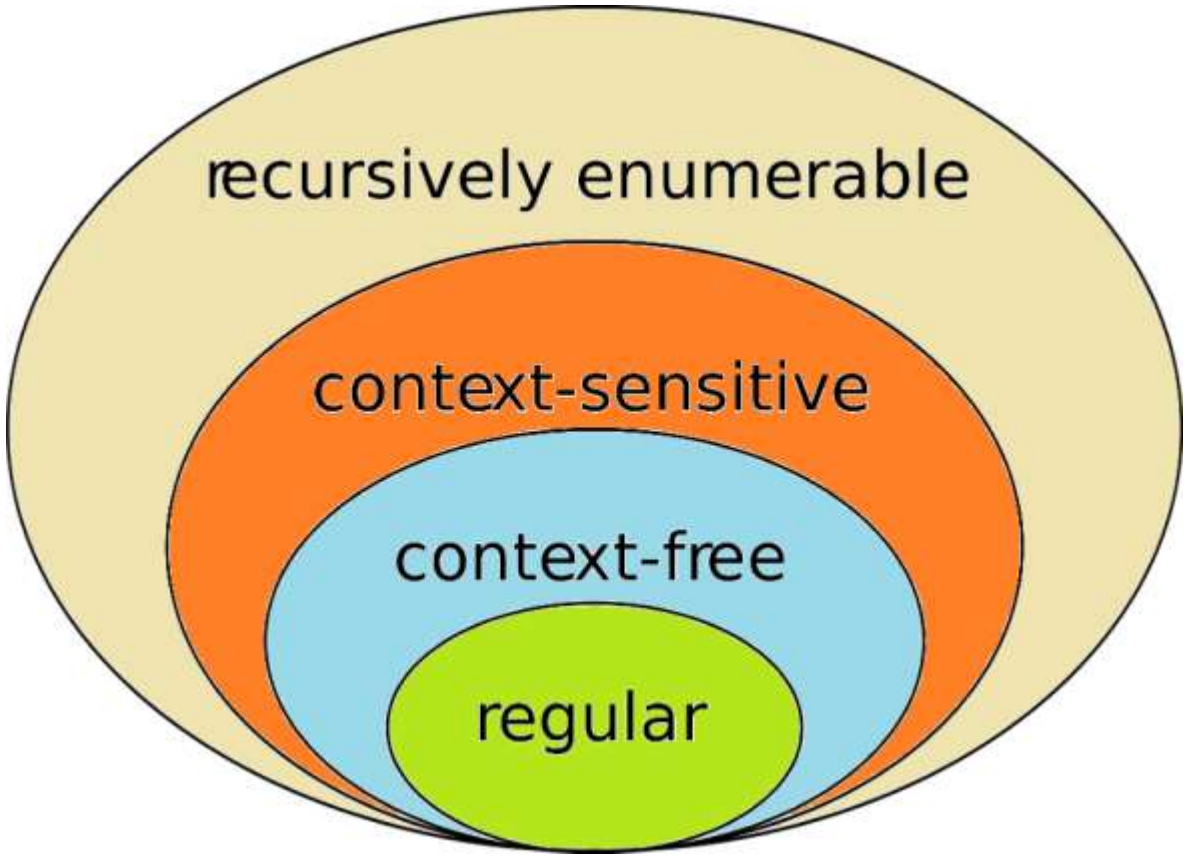
- The exam may contain questions from any of the material covered in class
- Material in the textbook that was not covered in class will not be on the exam

One Page of Notes

- You are allowed one and only one 8½ by 11 inch page of notes during this exam. You may print on both sides.
- You are not allowed to use more than 187 square inches of paper surface
- You will do better if you make your own page of notes and not copy your friend's notes

Chomsky Hierarchy

- There are four classes of languages
- Each class is a subset of another class
- Regular languages are the simplest while recursively enumerable (RE) languages are the most complex



Theoretical Computers

- Each class of languages can be recognized by a type of abstract machine
- The classes can be defined by those languages that can be recognized by the machine for that class

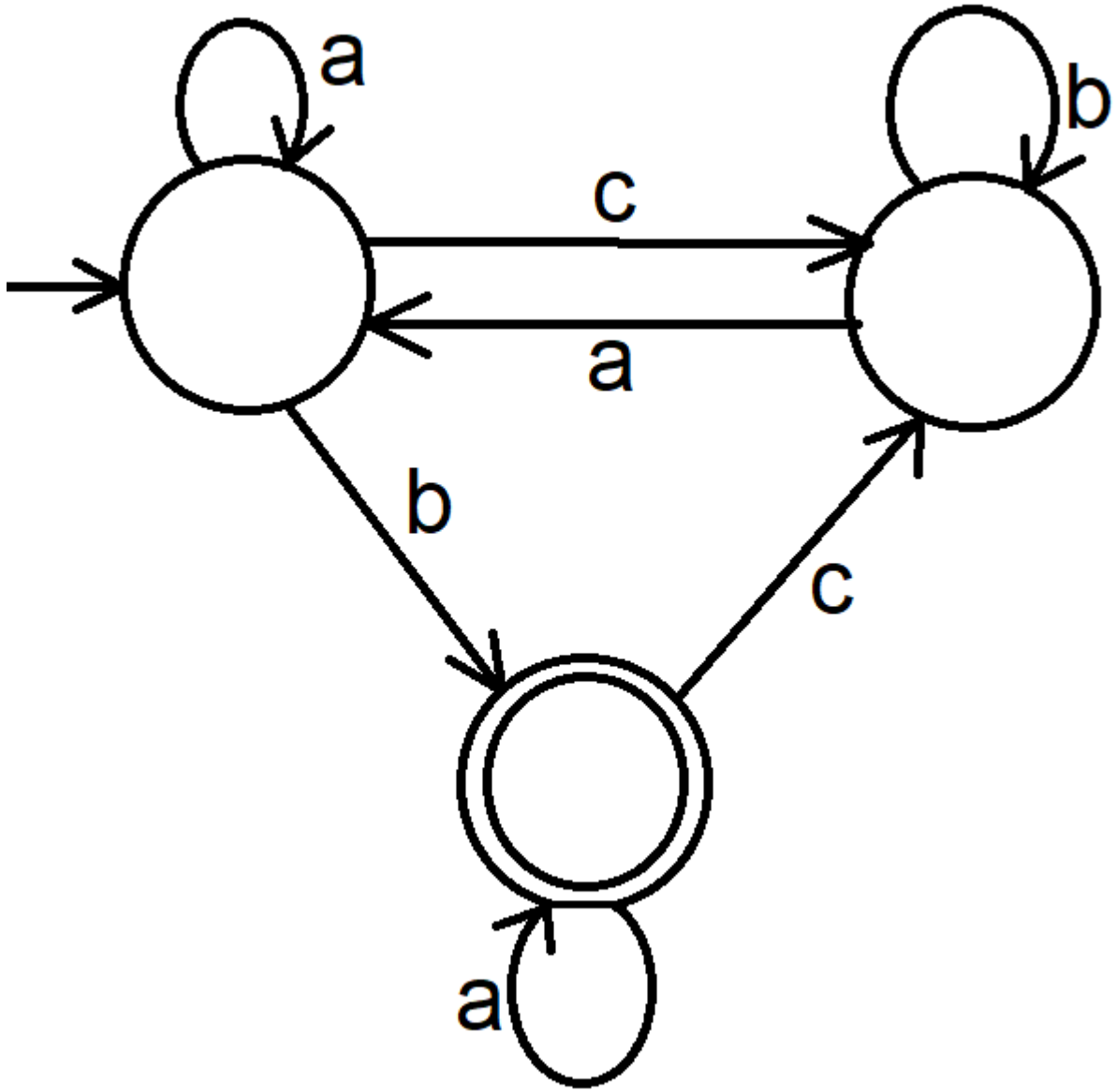
Languages and Machines

Language	Machine
Regular Expressions	Deterministic Finite State Automata (DFA)
Context Free	Push Down Automata (PDA)
Context Sensitive	Bounded Turing Machine
Recursively Enumerable	Turing Machine

FSA Execution

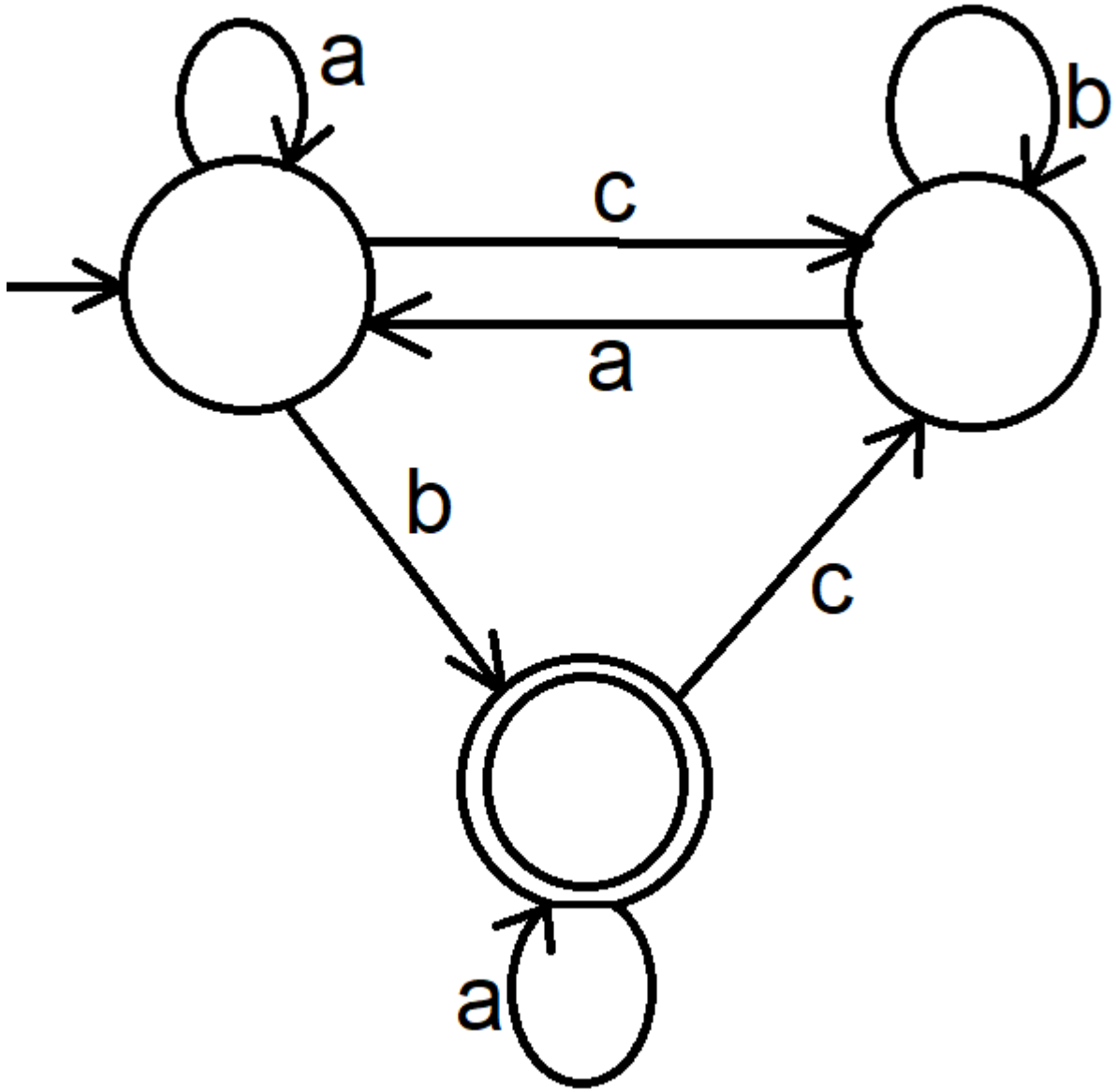
- The FSA has a “current state” which starts at the start state
- The input string is examined left to right one character at a time
- Based on the current input character, the FSA can move from the current state to a new state following an arrow labeled with the current input character
- Execution ends at the end of the input
- If the FSA cannot move, it ends in a fail state

Example FSA



- Accepted
  - **aab**
  - **bcbbab**
  - **caba**
- Not accepted
  - **cba**
  - **bcbb**
  - **bb**

Which string is not recognized?



A. aacbab

B. abc

C. abaa

D. bcab

E. b

Push Down Automata

- A Push Down Automata (PDA) has a FSA and a stack memory
- The top of the stack, input symbol and FSA state determine what a PDA will do
- The input is examined one character at a time
- A PDA can:
  - Push a new symbol on the stack
  - Pop the top symbol from the stack
  - Change the FSA state

Push Down Automata Languages

- A Push Down Automata (PDA) can recognize context free grammars
- Most programming languages are context free grammars (CFG)
- All regular languages are a subset of context free grammars
- A PDA can “count” things

Classical Languages Recognized

Machine	Language
Deterministic Finite State Automata (DFA)	Regular expressions
Push Down Automata (PDA)	Context free grammars, $a^n b^n$
Bounded Turing Machine	$a^n b^n c^n$
Turing Machine (TM)	Anything that can be recognized

Regular Expressions

- Regular expressions are often used to specify a regular language
- A regular expression defines a series of symbols in the order they can appear in the language
- Consider a very simple language with only the letters "**a**" and "**b**"
- **aabb** is a regular expression for a string with two letters **a** followed by two **b**'s

Regular Expression Alternatives

- A regular expression can specify alternatives using the vertical bar character, |
- **`aabb | ba`** defines a regular language that accepts two **`a`**'s then two **`b`**'s or a string with just a **`b`** and an **`a`**
- You can use parenthesis to group strings
- **`(aabb | ba)a`** defines a regular language that accepts **`aabba`** or **`baa`**

Regular Expression Repetitions

- A symbol or (group) might repeat multiple times in a valid string of the language
- A "*" indicates that a symbol or (group) repeats zero, one or many times
- A "+" indicates that a symbol or (group) repeats once or many times, but at least once
- **a**$^{0..1}$ mean the symbol may appear zero or 1 time
- **(aabb | ba)**$^{+}$ **a** defines the strings

  aabbaabba, aabbbaa, baa and more

Which string is **NOT** `(aa | bb)* a+`

A. a
B. bbaabba
C. aaa
D. aabb
E. bbbbbbaaaaa

Stages of a Compiler

- Source preprocessing
- Lexical Analysis (scanning)
- Syntactic Analysis (parsing)
- Semantic Analysis
- Optimization
- Code Generation
- Link to libraries

Output of Compiler Stages

Step	Output
• Lexical Analysis (scanning)	List of tokens
• Syntactic Analysis (parsing)	Parse tree
• Semantic Analysis	Intermediate code
• Optimization	Better Intermediate code
• Code Generation	Machine language

Lexical Analysis

- Lexical Analysis or scanning reads the source code (or expanded source code)
- It removes all comments and white space
- The output of the scanner is a stream of tokens
- Tokens can be words, symbols or character strings
- A scanner can be a finite state automata (FSA)

Syntactic Analysis

- Syntactic Analysis or parsing reads the stream of tokens created by the scanner
- It checks that the language syntax is correct
- The output of the Syntactic Analyzer is a parse tree
- The parser can be implemented by a context free grammar stack machine

Simple Program

```
/*  This is an example program */
bull = (cow + cat)    *    dog;     // comment
goat = “big quote”;
```

After Lexical Scan

bull	=	(	cow	+	cat	)

*	dog	;	goat	=	big quote	;

- The lexical scanner creates a list of all tokens in the program
- Comments, line divisions and whitespace have been removed
- Words, numbers and quote strings are single tokens instead of individual characters
- The list could be an array, a linked list or an ArrayList

Token Object

A token created by the lexical scanner should contain:

- String value; // text of the token
- int type; // type of value
  - name
  - constant
  - punctuation
- int line number, column // position in source code

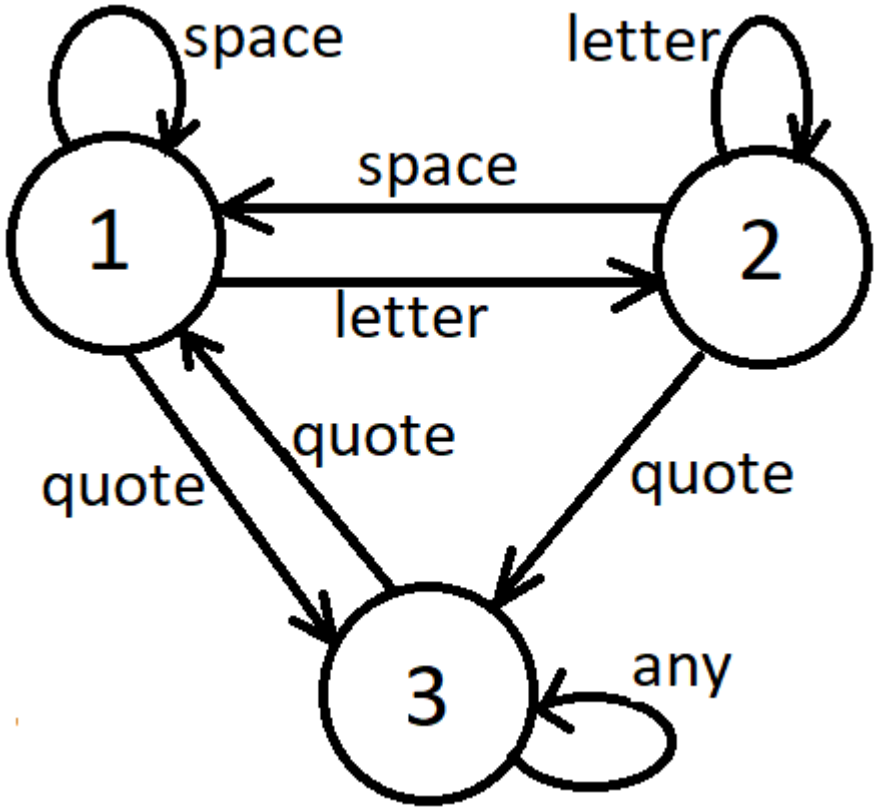
Quotes and Names

- Consider the very simple language that allows names (strings of letters) and quotes (start and end with a quote and anything inside)
- Example

goat "Cat chased the dog!" frog

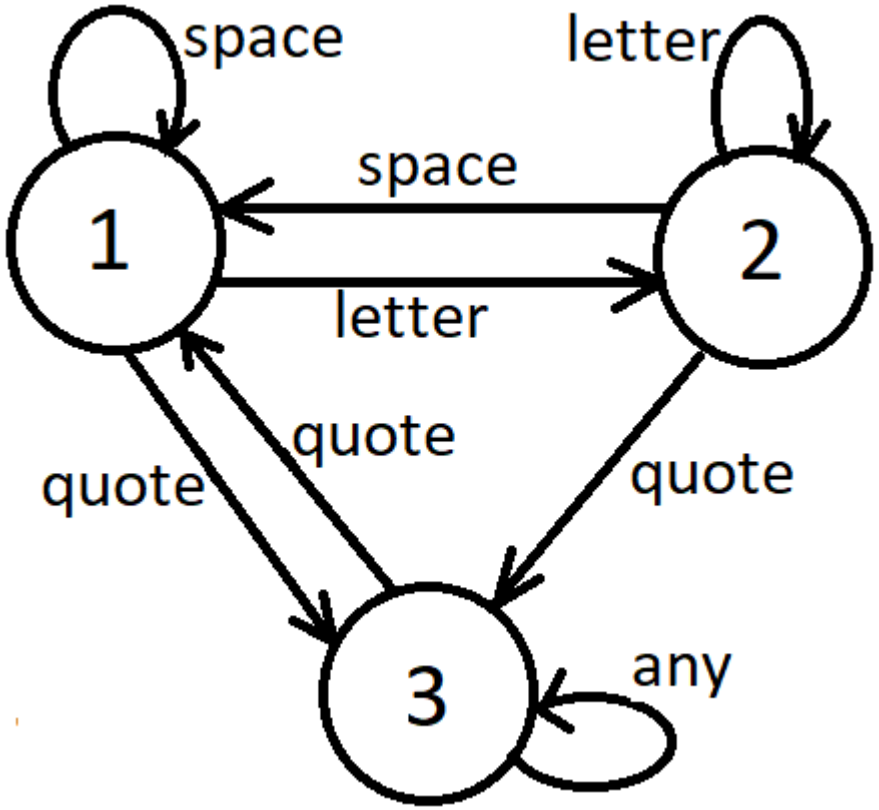
Quote and Name FSA

	1	2	3
Letter	2	2	3
Space	1	1	3
Quote	3	3	1
Other	0	0	3



Quote and Name Mealy Machine

	1	2	3
Letter	2/1	2/2	3/2
Space	1/0	1/0	3/2
Quote	3/3	3/3	1/0
Other	0/0	0/0	3/2



Actions

0 = do nothing

1 = create token with input character

2 = add input to token

3 = create empty token

Convert Input Character to Index

```
if (Character.isLetter(symbol)) {
    inx = 1;
} else if (Character.isDigit(symbol)) {
    inx = 2;
} else if (symbol == ' ' || symbol == '\t') {
    inx = 3;
} else if (symbol == '#') {
    inx = 4;
} else if ("=|{};".indexOf(symbol) != -1) {
    inx = 8;
} else {
    inx = 9;
}
```

FSA Program

```
state = 1
while not end of file
        symbol = next input character
        symIndx = number based on type of character
        action = actionTable[symIndx, state ]
        state   = stateTable[symIndx, state ]
        if state = 0 then error
        switch ( action )
                case 1: new Token(symbol); put on list
                case 2 : append symbol to last token
                case 3: new Token(); put on list
```

If you are in a comment state and you input a letter, what action should you take?

A. do nothing
B. create token with input character
C. add input to token
D. create empty token
E. None of the above

BNF Fundamentals

- Terminal symbols are tokens or symbols in the language. They come from the lexical scanner
- Nonterminal symbols are “variables” that represent patterns of terminals and nonterminal symbols
- A BNF consists of a set of **productions** or **rules**
- A language description is called a **grammar**

BNF Structure

- Nonterminal symbols are sometimes enclosed in brackets to differentiate them from terminal symbols
- I will use **bold font** for terminal symbols and no brackets
- Productions are of the form:

  nonterminal → nonterminals or terminals

- Multiple definitions are separated by | meaning OR

  whatever → this | that

Possible Snowflake BNF

1. SF → parmstmt code ret
2. parmstmt → **parm** varlist **;**
3. varlist → **var**
   | **var** varlist
4. varclist → **varconst**
   | **varconst** varclist
5. ret → **return var ;**
6. pattern → **varconst**
   | **varconst | pattern**

Possible Snowflake BNF (cont)

7. code → line
   | line code
8. line → assign
   | loop
9. assign → **var** **=** varclist **;**
   | **var** pattern **=** varclist **;**
10. loop → **while** **varconst** pattern **{** code **}**

How would you modify the Snowflake BNF to add an IF statement?

- Consider modifying line 8 and adding another production

8. line → assign
| loop

Possible Solution

- Consider modifying line 8 and adding another production

8. line → assign
| loop
| ifstmt

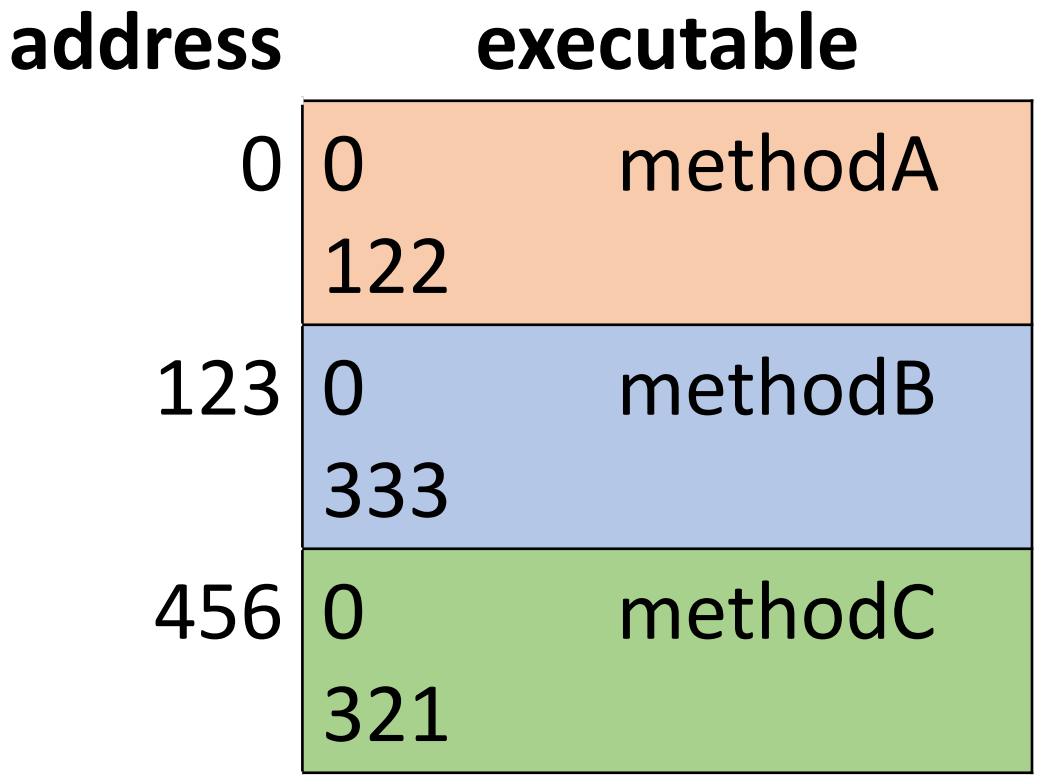
13. ifstmt → **if** **varconst** pattern **{** code **}**

Address Evolution

Level	Address Type
Source code	Name
Object files	Relative to start of segment
Executable file	Relative to start of executable
During execution	Machine address

Combining Object Files

- The instruction segments of the object files are concatenated into one segment
- This establishes the location and address of the object files in the executable



Adjusting Address in the Code

- After the address of each object file is established in the executable, the addresses in the instructions are adjusted by adding the start address of the appropriate object file
- The address of external names are inserted into the instructions

Compiler Output

```
int           cat  = 47,
extern int dog;                  // dog in another file
int  myProg(int stuff) {
    int cow = stuff * cat + dog;
```

Addr	Machine Language	Assembler
001e	8b 45 08	mov eax, stuff[ebp]
0021	0f af 05 00 00 00 00 + data seg	imul eax, cat
0028	03 05 00 00 00 00 + dog	add eax, dog
002e	89 45 f8	mov cow[ebp], eax

Two Pass Assembler

- The first pass puts the labels and their addresses in the symbol table
  - Only looks at labels and mnemonics (to get length)

- The second pass displays the machine language
  - Gets numeric values from the symbol table
  - Puts all the numbers into the format for that instruction

Likely Exam Questions

- Write FSA for a scanner
- Write a BNF
- Write a recursive descent parser