Project I: Construct a Lexical Scanner for VC

TS. Nguyễn Văn Vinh

Content

- Important Information
- VC Language
- Problem Definition
- Instructions

*

Important Information

- Group size: 5-6 students
- Duration: 3 weeks (28/02 - 21/03)
- Technical requirements
  - Use one of C++, Java, Python
  - The lexical scanner must be an automaton
- Submission:
  - To email: quan94fm@gmail.com
  - Must be formatted as follow
    - File zip
    - Filename: <classname>_project1_group<groupindex>.zip
    - NOTE: Group index started from 1

      Classname is defined as the school designation eg. INT2424

      Mail heading is [Compiler]<filename>
- Found occurrences of copying/cheating will be failed immediately
  - All instances of copying code will be punished. Groups and students can share their concepts and structure with each other, but not their code. The goal is to create programs with well formatted code, ease of reading, and efficient runtime.

*

Grading Barriers

- This project will equate to 20 points and will have two differently graded sections:
  - Report by pdf or doc(report) – 8 point
  - Source code - 12 point
    - Source code in your programming language
    - README (pdf, doc, txt, md)

- The source code should be properly commented and explained. For comments, it's important to mention the algorithm and sections' purpose, with the goal of showing the flow of your code throughout the function. Instruction on how to run the source code and examples are also required in the main report.
- Beside the main report, your code also need a README file that serves as running instruction and detailing your project structure. At the bare minimum, it should contain all necessary information to run your code.

*

Penalties

- For every late day after the deadline (**0h 03/11/2021**), 5% of your score will be deducted. The maximum delay is a week/7 days after the deadline. You can ask for an extension which will raise this delay period to 14 days, however, barring a very convincing case the score deduction will still applies. After these legal delay period, your project cannot be accepted under any circumstances.
- You must submit your **report, source code, and all corresponding documentation**. You can submit patches to your source code after the initial submission, however you will receive a penalty depending on the severity of the changes and the time of your patch. If your source code can't compile, we will notify you to fix it with a patch, in which case the same rules above applies.

*

VC Language

- See document on courses site!

Problem Definition

- Building a Lexical Scanner for VC language:
  - Identify all morphemes found within the VC file
  - Raise compilation errors if available

Input, output, automaton file

- Input: text file (*.vc) containing VC code
- Output: text file (*.vctok) containing the list of words of the input file, each word on a line
- Automaton data (*.dat): containing starting states, ending states, transition table (the text files are formatted as ASCII), ending states-words mapping and list of acceptable ending states

How to build a Lexical Scanner

1. Build a list of all rules; these rules are often described in normal language.
2. Draw a transitioning graph for each rule. Regex can be used in this step to help reconstructing the rule and ease the subsequent steps.
3. Combine all your rule graphs to a single graph.
4. Construct a transition table using the graph, and export to appropriate format.
5. Build your program to read the formatted table.
6. Add detection mechanism for compilation error

*

Instructions

- The main goal is to build an state automaton with the capacity to detect morphemes in VC programming language
- Use graphs to achieve this goal
- Transition graphs should be drawn independently
- To simplify running process, we can use characters as shorthand for transition

- The below is a limited version of VC used as an example. 1-4 indicate the steps to create the transition table from these rules.

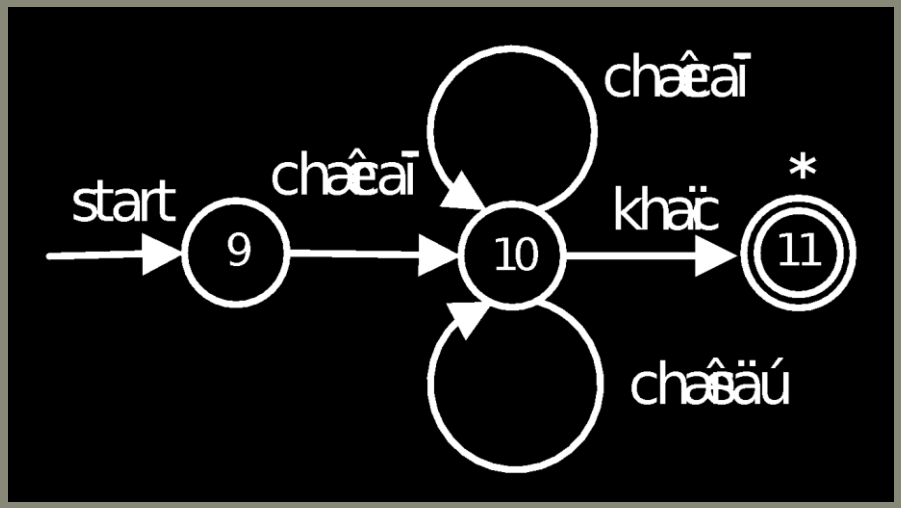
*

1. List of all morpheme

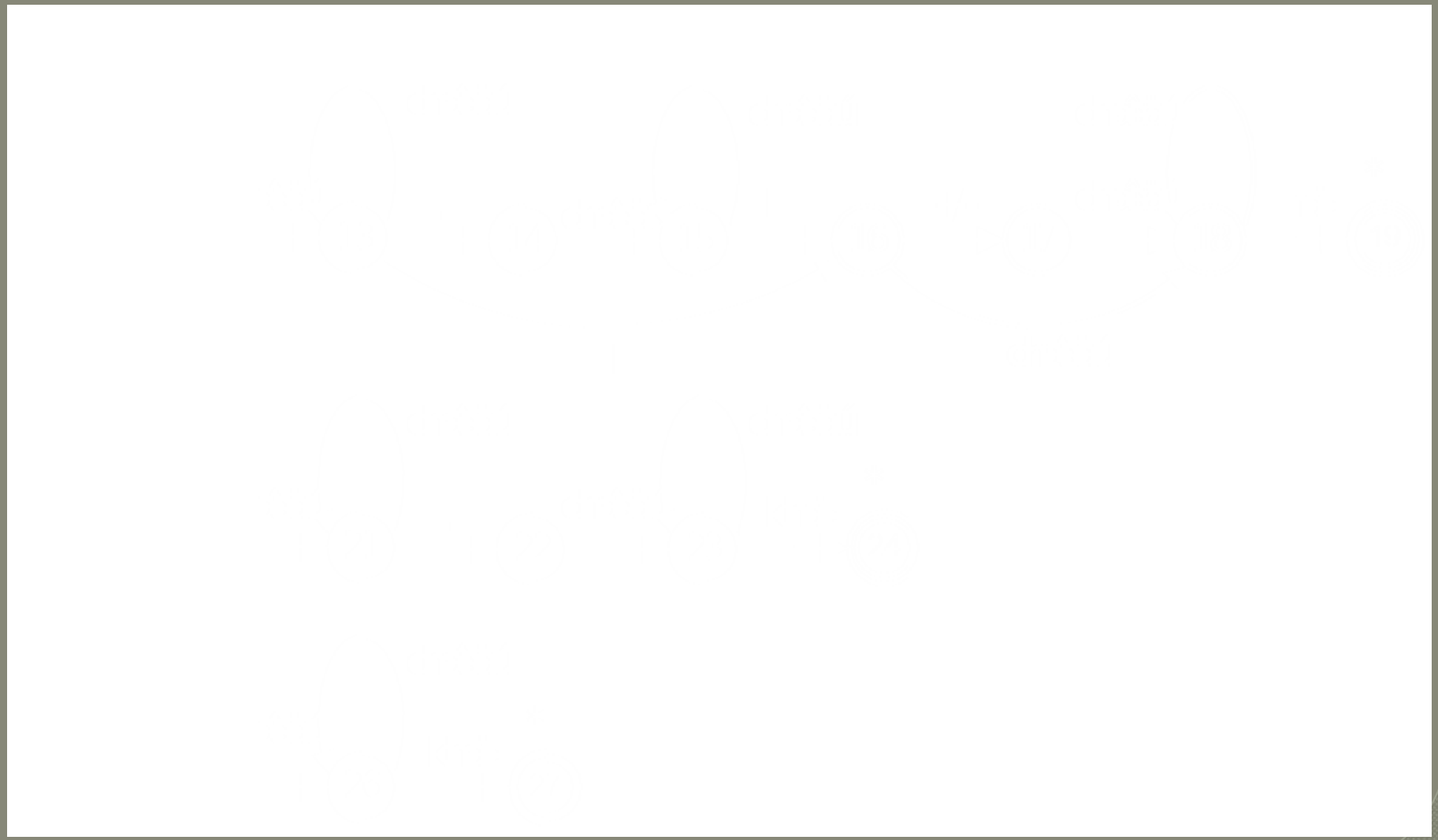
- Consider this subset VC with these morphemes:

name → character (character | number)*
relation → < | <= | = | <> | > | >=
e_notation_float → number+ (. number+) ? (E (+ | -) ? number+) ?
float → number+ . number+
int → number+

2. Creating transition graph for morphene **name**



2. Creating transition graph for morphenes: **e_notation_float, float, int**



3. Combine all morphenes graph to main graph

4. Building transition table

- Table
  - 10 row, each row equivalent to a state
  - 8 column: Each character fed into the program is categorized as character, number, E, ., <, =, >,+/- (8 different character types in total).
- Question: Why do we only have 10 rows instead of 20 (same as the main graph's number of nodes)?

Transition Table10x8

State	*Input category*							
	<	=	>	Character	Number	E	.	+/-
0	1	5	6	10	13			
1	4	2	3	4	4	4	4	4
6	8	7	8	8	8	8	8	8
10	11	11	11	10	10	11	11	11
13	27	27	27	27	13	16	14	27
14					15			
15	24	24	24	24	15	16	24	24
16					18			17
17					18			
18	19	19	19	19	18	19	19	19

Notes

- In this simplified project, we don't construct regex analysis modules, converting regex to NFA (Thompson algorithm), converting NFA to DFA, etc.
- Why?
  - Automaton is already build manually outside
  - Lessen the amount of code needed

Good luck!