Semester Work

Standalone Parallel Scanner FLEX Generator Building a

Description

To be able to proceed with the development of the Part A of work you must have acquired the necessary theoretical knowledge for the description of programming *language grammars programming languages* **and how a** *parser* works *verbal analyzer* that have been taught in the theoretical part of the course.

For help with designing regular expressions use the regexpal tool or other similar that you will find in the tool Links to ______ **eClass.**

To help you check the correctness of the finite state machines you will design use the fsm tool available in the fsm tool for installing on eClass. Installation instructions and information on how to set up of the tool and information on how to set it up and how it works can be found in the 0_1-KE and FSM folder **in open eClass.**

The FLEX generator, parser using the tutorial example and the related file in the 2-FLEX **folder and from other sources on the internet. A related presentation will take place in the corresponding course of the workshop.**

After completing the assignment, you should be able to:

- **design regular expressions to describe the words of a grammar create finite automata from regular expressions**
- **write FLEX code to create a stand-alone independent speech analyzer**

The purpose of this assignment is to gain experience in describing and designing regular expressions and in particular to become familiar with the development of a lexical analyzer. You will write a speech analyzer using the FLEX tool. To develop your speech analyzer use FLEX which is freely available ______ **instructions in the 0_0-Linux & C folder. We will test the code you submit using the Linux version of FLEX, so make sure your solutions work correctly in that version before you submit them through the open eClass Assignments tool.**

To complete part A' of your assignment you should answer the following sub-questions parts in the time frames set for each:

PART A-1: Learning KE Write

The first part of this assignment is to explore the regex environment or similar, and to go through examples that you will find in the file Practice_in_KE in the envelope folder 1-Practice material.

This step is designed for your own practice without having to submit anything related to the your work.

PART A-2: Encoding Finite State Automata via FSM **Draw the CUs describing the verbal units of your working language, their corresponding finite state automata recognitions, the corresponding transition tables, and the resulting unitary automaton *(omit the reserved words for which the grammar of the language given to you says that they don't need a recognition pattern because they are already inside the symbol table (or your own equivalent structure, such as commands of the language).***

Simulate the unitary automaton in a generic Transition Table (TF) and then, with the help of the FSM meta-tool, encode the generic TF to check the correct recognition of the grammatical units.

Note: The unitary automaton must contain ONLY permissive outputs. In the single PM, leave the cells corresponding to NOT allowed transitions blank and in the FSM code them as a common transition named 'BAD'.

Be sure to post your answer by submitting a compressed ZIP file with a name beginning with the team code required and containing:

a) A Word or PDF document that will contain:

• Cover (with names of team members, team number & section) & Table of Contents • Regular expressions (partial only – not unitary) • Finite State Automata or DM (partial & unitary) • Transition tables (partial & unitary) • Check / exhaustive cases test executions / results and their comments • Comments on the above submissions (e.g. mode of operation, problems you encountered, if / how you solved them or if you did not solve them what you tried to do • Explicit reporting of deficiencies (compared to what was requested) as well as report on correctness or incorrectness running your code •
Break down the responsibilities/roles of all team members •
Be sure to consult the documentation instructions at the end of this document b) The FSM file of the single automaton as well as the individual FSM files you have created, adequately commented

Note: All of the above is recommended to be implemented as soon as possible but will be submitted together with part A-3 of the paper

PART A-3: Completing the FLEX template code **In this part it is recommended to first complete the incomplete template simple-flex-code.l from the compressed file named simple-flex-code.zip that you will find inside the folder *Assignments* of the folder 4-Material for Assignments. This file is provided as a basis for easier development of your own parser, but its use is optional as you can develop a LA from scratch as long as your implementation meets the specified specifications.**

For simple-flex-code.l you should replace the "FILL ME" inside the file with the FLEX code as well as in the corresponding token.h header file with the missing code. First make all the required replacements and fix any problems that may exist in the code. Then test the execution of your file with the lexemes given in the input.txt file, compare the execution results with those given in the output.txt file, and

try to understand the operation of the generator. For the correct implementation of the request, follow the following steps:

1. Download the file simple-flex-code.zip from JOBS in eClass. 2. Save / move the file to your home directory 3. Unzip the file by giving the command: unzip simple-flex-code.zip
4. Give the make command to the terminal in order to compile and to automatically run the parser with the input.txt file

Once you have completed the above steps, complete your speech analyzer to recognize all the speech units, separators and comments of the given language. Also be sure to insert your own comments into the code you write (inside the .l file) using the /* and */ character sequences familiar from the C language but with great care so as not to cause a problem in the C file generation process from the code FLEX.

The verbal analyst should additionally:

- ÿ **ignore** white_spaces **characters after separating them**
- ÿ **recognize comments and ignore them**
- ÿ **when encountering a newline character to increment a line counter necessary to show which line the parsed string is on**
- ÿ **when it successfully identifies and identifies a verbal unit (token), to return the source code line in which it was found, the identification name of the corresponding token and its value, e.g.** Line=1, token=INTEGER, value="5".
- ÿ **in any other case to scan all the characters of the unknown string until it finds a delimiter and displays the following message:** Line= *line number,* UNKNOWN TOKEN, value="value of *wrong string"*

  (and continue normally with parsing the next word).

Please post your completed sample code, properly commented for understanding, along with your test results in a timely manner. Your deposit will be a compressed ZIP file with a name beginning with the team code required and containing the following:

1. All parser core files (flex code, header file) adequately commented 2. Input - output accompanying files (input.txt - output.txt) that will include several recognition cases for all word units as well as invalid cases verbal units

3. A Makefile that will allow your code to automatically compile and run, such as the one provided in the template code you downloaded
4. A Word or PDF document containing:
   a. Cover (with names of team members, team number & section) & table contents
   b. Comparing the input you tested with the output you received to clearly show your parser is working properly. Test cases / test runs should be exhaustive.

c. Detailed commentary on the test results (consult the instructions documentation at the end of this document). In any case, a simple copy-paste of the results is NOT enough! d. Comments on the submitted code (e.g. how the LA works, problems you encountered, if / how you solved them or if you did not solve them, what you tried to do e. Explicit reporting of deficiencies (compared to what was requested) as well as reporting on the correct or not compiling and executing your code f. Analysis of responsibilities/roles of all team members g. Be sure to consult the documentation instructions at the end of this document

Note: The above code will form the basis for the complete development of the FLEX code of your final work and will participate accordingly in its rating.

Documentation Guidelines: Write a documentation issue with a cover, table of contents, introduction, documentation, test runs, and presentation of results properly annotated for understanding.

Note: Job
submissions with a zip file name that does not begin with a group code WILL NOT BE DONE RECEIVERS.

It is also clarified that where code or results are required:

1. No Greek will be accepted, only Greek (or only proper English). 2. Image snapshots / captures will not be accepted, but only properly formatted text (regarding line wrapping, so that it is visible and readable).

3. Test results should be annotated so they can be understood. It is noted that the tests must be exhaustive to certify its correctness and completeness code.

4. In any case, you should explicitly state in the documentation whether or not your code compiles and whether or not it produces correct/ acceptable results. Additionally, any problems or deficiencies in execution must be reported.

To better understand the assessment criteria for your work, you should carefully read the assessment rubric found in eClass.

Last update: 2024.03.08