Name/s: ____

NIA(s): ___________________ Group: ________

Practice 2 Lexical Anlysis

In this practice we work the lexical analysis by implementing a simple scanner. There is work to be done in paper to build the regular expressions and automta to later implement them. So this assignment has special need of a well documented design report to include this and all design decisions and justifications. In line with the previous assignment, this design report is presented as the developer's manual.

1 Language definition

We want to support the following language specification, which is a very small sub-set of c requirements. For this scanner assignment we just treat the words. In the following assignment (parser), the specification may be further limited to reduce the parser complexity.

The scanner specification must consider:

- No preprocessing: We assume there are no preprocessing directives (starting with #), nor comments in the input file to the scanner. Either the input file will not have these elements, or it will use the preprocessor implemented by us to eliminate them (the ones supported by the preprocessor). In either case, the input file to the scanner cannot have any of these elements.
- Alphabet: you have to derive the alphabet of the language as a result of the specification of lexemes describe next. Clearly indicate it in the documentation (developer's manual).
- Types: The language has to support the following types: int, char, void. Note that with the combination of these types and special characters and operators, the scanner (specified next) will recognize vector, matrix and pointer types of these basic types. Although note that this is just a scanner and it does not interpret the syntax or semantics of the code.

  Examples:
  ```
  int n = 3;
  char vect[2];
  double *vect;
  int vect[]={1,2};
  ```

- Operators: The scanner has to recognize the 4 following operators: = > + *
- Especial characters: It must recognize the following 8 special characters: ( ) ; { } [ ],
- Numbers: The numbers are limited to a combination of only integers. It does not support fractional, exponential or hexadecimals (no decimals nor chars). Any combination of digits 0 to 9, including 0's at the beginning of the number.
- Keywords: if, else, while, main, return
- Non-essential characters: space, eol, tabs (and similar delimiter characters). These characters are used as delimiters and are not part of any token.
- Identifiers/variables: Any set of strings with a combination of capital and lower case chars and numbers and different of any keyword already mentioned that has a special meaning. The identifier has to start with a letter and not a number and it cannot contain any other character different than the mentioned ones. We restrict also that any identifier cannot start with a substring of another identifier or any keyword, this means, that every character at the beginning will identify what identifier is (lookahead of 1). (There are two keywords that start with i: int and if). Note that any typo may produce additional identifiers. For example: wile as a typo of while missing the h can be recognized as identifier instead of the keyword while. We do not support typos.

- Literal: Any text delimited by " ". The lexeme includes the " ". Example: "core dumped". The characters in between are not processed and we accept any character that belongs to the alphabet. We assume there are no characters that do not belong to the alphabet.
- Non-recognized: Any character not supported should generate an informative error message, generate a non-recognized token in the output and the scanner should continue with the next lexeme. If there is a consecutive set of non-recognized characters, they all should be grouped in a single non-recognized lexeme and single error message for this lexeme.

2 Tokens

A token is defined as: <lexeme, category>

Where the lexeme is the string that identifies the token, and the category is the classification of the token to be used in later phases of the compiler (as explained in theory).

You have to identify what lexemes are needed and decide what category you assign to each lexeme. This category defines the use of the token in the following compiler phases. Hence these categories may need to be adjusted in the following phase. We may later need to define additional lexemes not considered now, and we may need to define additional categories for the new lexemes or to group the lexemes differently and hence modify the category of current lexemes.

Define a flexible and easy to modify design of the token specification.

Right now consider the following categories of the lexemes:

- CAT_NUMBER: for the numbers as described in previous section.
- CAT_IDENTIFIER: for the identifiers as described in the previous section.
- CAT_KEYWORD: for all keywords of the language that are described in the previous section.
- CAT_LITERAL: for all literals as described in the previous section.
- CAT_OPERAND: for all operands as described in the previous section.
- CAT_SPECIALCHAR: for all special characters as described in the previous section.
- CAT_NONRECOGNIZED: for all lexemes that do not follow any of the specification above. These lexemes are mainly combination of characters that do not belong to the alphabet, but there are also the cases of special characters that can only appear in particular combinations in lexemes but not in other combinations. Example: the character | is part of the alphabet but it can only appear in pairs ||, so a single one is part of a non-recognized lexeme.

We can anticipate already, that the syntax of the different keywords, operands and special characters follow different syntactic rules and hence some more category differentiation will be needed for the parser assignment. So right now we are more interested on how the categories are implemented than how many are implement.

You can apply modifications of these set of categories if anyway you provide a general implementation to support several in a configurable way. Justifying the modifications explaining why it is good for your design. A justification of considering that the process is the same and repetitive is a good justification, very specially for smaller teams.

2.1 The token list

Internally, the scanner has to create a list of tokens including all tokens identified as specified above and in the order they appear in the input (and are being identified by the scanner).

Eventually, this list of tokens will be passed to a parser (to be worked in the next assignment) as the next step of the compiler.

To know what the scanner is doing, the list of tokens should be provided in a file as the output of the scanner phase as indicated in the following output section.

3 Output

The file has to provide a written version of the token list so we can verify that it keeps the correct token list in memory.
The scanner must create a new file adding the suffix scn to the extension, as follows:

<filename>.<ext>scn

Since we do not allow preprocessing, the .h files are not allowed (or already processed by the preprocessor). Hence the only extension of the input file <ext> is .c and the output file extension is cscn. However, the program should create the output extension as the input extension concatenated with the scn suffix.

Example: executing the scanner to the file example_app.c should produce an output file named example_app.cscn. It changes the extension as the content of this file now has a different format as it contains sequences of tokens instead of a c language.

The format of the tokens in the output file has to be configurable with preprocessing configuration. We want to add information in the output format while we are debugging, and this extra information should not be put in a release version.

The compiler (in particular the parser which uses this output file) does not

The tokens in the output (text) file may be written in different form depending on a OUTFORMAT preprocessor variable that can take at least two forms (you can define more options if you want):

- RELEASE: The release format is defined for the compiler use of the output file and it must put the sequence of tokens as is expected to be used by the parser. The format has to be agreeded as we can use anyone's parser in the following phase. Our format has to organize the tokens in the same lines as the lines of code in the input c file. Therefore, the input and output files should have the same number of (information) lines but with different information:
    - Each line of code (any sequence of characters) in the input translates to one line of tokens in the output. The tokens appear as is in sequence each one separated by one space character.
    - The token file should not have empty lines. So the empty lines of the input to separate the code are eliminated in the output file.

  Example:
  Input c file:
  ```
  if(x > 3)
  	printf("true");
  else
  	printf("false");
  ```

  output tokens file:
  ```
  <if, CAT_KEYWORD> <(, CAT_SPECIALCHAR> <x, CAT_IDENTIFIER> <), CAT_SPECIALCHAR> <3, CAT_NUMBER>
  <printf, CAT_IDENTIFIER> <(, CAT_SPECIALCHAR> <"true", CAT_LITERAL> <;, CAT_SPECIALCHAR>
  <else, CAT_KEYWORD>
  <printf, CAT_IDENTIFIER> …
  ```

- DEBUG: This option is to debug and we want to be able to follow what it contains by looking at it. So it has to be easy to see and it puts extra information to follow the process. Therefore, the format is:
    - It should follow the RELEASE format but in addition it has to provide the following.
    - There should be an empty line after every token line to distinguish the different potentially very long token lines.
    - Each token line starts with a line number, and this line number has to be the line number of the input file containing the code that correspond to this token line. This way we can easily match the line of tokens with a particular line in the input.
    - The debugging messages (see next section) should be written at the output file. So the messages are related with the tokens.
    - Any particular format you prefer to use so to follow better the process. Note that you can use anything that works for you as this format is just for your use, and it is not used for a final release version. So it does not need to be recognized by anyone else (parser). For example, you may prefer to have each token in a different line and hence a group of lines be one single input line.
    - We use this format to print additional information as described in following sections.

  (Previous) Example:
  output tokens file:

  ```
  1 <if, CAT_KEYWORD> <(, CAT_SPECIALCHAR> <x, CAT_IDENTIFIER> <), CAT_SPECIALCHAR> <3, CAT_NUMBER>

  3 <printf, CAT_IDENTIFIER> <(, CAT_SPECIALCHAR> <"true", CAT_LITERAL> <;, CAT_SPECIALCHAR>

  5 <else, CAT_KEYWORD>

  7 <printf, CAT_IDENTIFIER> …
  ```

- Other options: you can decide any option you feel useful and explain and document them appropriately at all documentation levels (user's manual, man pages and developer's manual).

4 Debugging

We want to have the possibility to redirect the error messages to the display (stdout) or to intermix them with the output at the output file.

We want a configuration preprocessor directive called DEBUG which can have two values: 1) ON = 1 which indicates that (all) messages are written to the output file; 2) OFF = 0 which indicates that all are written at the stdout. This means the application has to always write to a file (use always fprintf instead of printf), and the file handler to be used should be initialitzed to sdtout or the output file name when the DEBUG is configured to OFF or ON respectively.

Modifying this directive, we obtain different versions of the executable. Hence, this is a configuration "parameter" of the application, and as such it needs the corresponding documentation to inform the user (programmer) how to use it. Therefore, the user's manual has to have all necessary information to find this directive, know what exactly it does and how to modify it to get the option requested. This information has to appear also in the man page, and it can also be given to display when there is an error in how it is used so to inform the programmer of everything necessary to modify it right. In addition, the decisions taken of where it is and why, should be clearly explained in the developer's manual.

You can use this flag to have error messages only active when you are debugging. However, once you have identified possible errors you may want to keep the identification active always to detect possible implementation errors once the application is completed and not configured as debug but as a real release.

5 Error handling

We do not expect a full implementation of possible errors and combinations. But we want a general approach to error handling. The errors have to be identified by a number and use the same error message for the same error in different places of the application. However, the message can include parameters that indicate concret information of where the error happens and additional relevant data.

The same organization of errors will be used for the following assignment (a parser). So we need to think that there will be scanner specific messages, parser specific messages and potentially other categorization. So the errors have to have a unique error identifier and a step identifier. The step is the phase of the compiler that informs and identifies whether this error happens in the scanner or parser (and potentially we can leverage the ones of the previous assignment and have preprocessor errors). And we could have a smaller step than a full phase of the compiler.

It has to be designed how to implement the error list in the developer's manual. In the middle of the code we just have a link to an error identifier and pass the specific text to add to the general error message text.

The error messages should be written where the DEBUG flag indicates.

6 The core of the scanner design

The scanner needs to process the input c code to generate the output as a stream of tokens. So it needs to design the regular expressions for the language specified and design the automata to implement them.

Note that the scanner has to work character by character and hence it is forbidden in this assignment to use the string library to identify a keyword in the input. It may seem simpler, but at the end these functions hide a character-by-character identification. So they hide the operations that we would design if we handle the characters. We are interested in seeing these differences and understand the impact of our operations. Hence we will count how many operations we do as explained in the following sections.

Each keyword is a simple regular expression and hence an automaton, that each character transitions to a next state until all characters of the keyword are processed (see examples in theory slides). There is only one acceptable transition in each state which is the next character of the keyword. Any other symbol at this state stops the recognizion of this keyword. If we send all these characters to an empty state (q_∞) we can have a DFA specification for each keyword.

At this level, the special characters are also an automaton but to recognize a string of just one character. With this we have covered the keywords, operators and special characters.

The numbers, identifiers and literals have a little more general automaton design. Some of them have been covered as example in the theory slides. The restrictions of the specifications limit the design of these automata significantly. So they are still small automata, and more importantly, it simplies the implementation of the whole set of automata. (If you need further simplication make the proposal as indicated below)

There are different approaches in implementing all automata together. One is to have a list of automata that we run all of them in parallel, until one succeeds. The restrictions are such that if one succeeds we can stop and generate the token, and reinitialize all of them again to continue the parsing of the input for identifying the next token. It can happen that no automata finishes successfully and in this case we should recognize the read sub-string as a non-recognized token at the point that all automata fail.

If we think about this list of automata running together is like implementing a big NFA that connects all DFA's that we have designed with ε-transitions. So, a second approach would be to consider this implementation as one NFA, and create the transition table of this NFA with multiple transitions at a time. One third approach would be to transform this NFA as the corresponding DFA that it would eliminate all ε-transitions but it will have more states. There may be more approaches to this design. You can decide your approach and explain your design in the developer's manual (and in the user's manual if it has impact in the use of the application).

7 Number of Operations

We want to understand how many operations the implementation takes so we can discuss different approaches with specific measures of the impact. Any implementation is correct, even it is inefficient. But we are interested in measuring it.

So we want to measure how many comparisons need to be done to identify each token. We request to do this analysis in paper an include a section in the developer's manual to explain it.

We also want the program to compute the exact number of operations it takes. We will do this at the preprocessor level, so it can be eliminated in a RELEASE code but included in a DEBUG code. Note that counting means incrementing a variable any time that the action happens. So if we do it for every comparison we could be doubling the number of operations we have in the normal operation of the code. So if this code remains in a release version it slows it down significantly. So it is important to implement it at the preprocessor level.

So we request to implement at the preprocessing level a constant COUNTOP that if it is ON=1 it must count and print information on the numbers of operations as requested next. And it has to eliminate all these operations when it is OFF=0. This means that this c code has to be included in a directive #if to eliminate when COUNTOP = OFF.

Where you put this configuration constant and how to use it should be explained in the user's manual and man page, and appear in an error message if it can create an idenfiuable error message.

We want to print the number of comparisons it takes to identify each token. So there should be a COUNTOP message for every token. And it should print at the end the total number of comparison operations it has taken in the whole input scan.

These messages should be sent where the DEBUG flag indicates (the same as the error messages).

8 Interaction with next phase (parser)

The output of the scanner is the input of the parser that we will (partially) work in the next assignment. However, we want to make this submission independent from the next one. So the specification has been to produce an output file with the tokens as specified above, in addition of the token list in memory. But we want later to have the ability to continue the execution of the scanner and directly call the parser without having to load the file of tokens.

So it is important to have the main program of this assignment to consider that it calls now the scanner only but later it will call the parser too. So we will want the following parser assignment to have the two options: 1) to continue running the scanner in the same execution with all data structure in memory, and 2) start from scratch and load the tokens file and do the parser execution as independent program.

So now you should have into account this, and do the program design to have this "hook" prepared for the following assignment.

9 Length of the Assignment

We want to have a feeling in this assignment of the implementation of a scanner, and understand why they are implemented with automata. The description is restricted to limit the scope. But some things need to be general to impose to think a general approach. The objective is to impose the general design and not to include tedious repetitive work. But some are unavoidable. That's why you are a large team to handle some of this. However, if the work can be further simplied, but keep the essence of the learning process you can propose the simplification. Discuss your simplification with a well justify reasoning with the professor to get approval for the simplification. If approval this same reasoning should be clearly explained and justified in the developer's manual (and in the user's manual if there is any implication on how to use the application).

Likewise, if the work seems too big to handle, discuss it with the professor to check whether you are in the right direction or not.

There are also some extra learning objectives of how to do and why to use the preprocessor, how to do debugging, error handling and algorithm design in the overall context proposed in the assignment.

10 Submission instructions

You have to work and submit the practice with the practice team of 6 people. See aula global to choose your team. Put in the submission the team identifier (GA, GB, …), but also put the names to confirm that all have contributed in the submission. Just put the names of the members of the team who contributed in the corresponding submission. Inform in advance to the professor, so we can address any issue, and she is informed of any particular time constraints with the submissions of particular students. Otherwise, it will be considered copy, and all involved (who did and who did not) will be considered responsible.

This assignment has 1 pre-submissions and the final submission.

Presubmission 1: The objective of this submission is the design of the solution and the team work planning distribution.

1. The status worksheet filled with the work distribution assigned and current status description. You can use the provided template or build your own format of worksheet that works for your team to describe the progress.
2. The design document describing how you have devided the overall program in different modules and why and how these modules are defined and how they interact between them. It is important to have a diagram or drawing to explain these components and interactions. Go as much into detail as possible with the design before starting the implementation defining data structures, functions and parameters that the functions exchange. Define your code file distribution and assign independent responsibilities to each team member, assigning file ownerships for the updates. This initial template should be empty but it should run so each member can work in its own files and advance the implementation. Agree on the data structures for each concept

appearing in the problem. This document at this level can be in bullet form and hence in presentation format for each of description and elaboration. (Leave the formal text description for a future submission when the solution is most advanced.)

3. Code: provide an initial code template that has the skeleton of your design distributed in files so that each team member has an ownership of files to advance in parallel to others. This version has to have little code but it has to compile and run so each one can compile and run his/her code to make progress.

Final submission:
The final submission must include:

1. The c code (.c and .h files) with a good program structure and modules, and well documented as indicated above. Also include a make file to compile it.
2. At least one sample input with the corresponding output file. These tests should be clearly explained and relate with the exact file names inside the report.
3. A report file including the design specification with all documentation requested in this handout. Compared to previous submission, this reports should include the user's manual, and developer's manual that previously this information was requested separately. Build a good report that contains all sections required to explain in detail the design issues, and the programming issues of the implementation. Give appropriate explanations and justifications of your design decisions.
4. The final version of the status worksheet giving a clear view of the status of the code submitted and filling the self-evaluation part as final self-evaluation of your submission.
5. A presentation (in slides format) of the submission (see the separate submission task and deadline for this in aula global)

Create a folder called p2_<Gn> where n is your group identity. Put the code files and documentation files in the folder (organize it in subfolders as you feel appropriate and add a README file to explain what is included where inside the folder and comment it in the documentation). Do not include the materials and files provided with the handout. If you create additional test files include them. Include as many files as you feel relevant. Rename the documentation files (but not the code files) to add the group identity, prefix _<Gn >. Put all code files in a sub-folder code.

Compress the p2_<Gn> folder (with all sub-folders) and submit in aula global the zip/rar file called p2_<Gn>.zip/rar. Only one student of the team must submit in aula global, as all will see the submission as the practice task is configured as a group task.

The submission must be done in aula global before the deadline indicated there. **Make sure you put the name/group inside the file and in the name of the files of all files (reports, code[1], input samples...)** so that the authors can be known when reading the documents and also when listing the files in a directory where all submissions can be collected.

11 Evaluation Criteria

The practice is evaluated in 3 levels following the attach evaluation sheet. Make sure you read carefully the evaluation so that you cover everything as requested in the submission.

There are 3 levels: non-competent, competent and excellent/outstanding. The non-competent does not reach the minim level required. The competent has a level to pass, and it means the team has this part of practice with a 5. An outstanding means the submission (or an individual criteria) is of very high quality and gives extra grade to this practice submission.

[1] Put the authors inside each code file as a documentation header on top of each file. Do not change the name of the code files with the group Identity, but do put it in all other files (documentation).

Note that a good design, at your experience level, requires a quite finished implementation. It may not require a fully functioning and debugged implementation (for each functionality), but a quite throrough implementation. So while a non-working implementation may get a competent level, to get so it has to have a design very specific that covers everything needed in the implementation so that the design has a high probability of success (finally working) as is. So a competent may not work but it has to have solved or addressed all important details in a correct way, which is the same as having a thorough design (if the design covers it) with an unfinished implementation. So a report with a very good design and an incomplete implementation can have a competent.

12 Copy and Plagiarism

In all submitted material put the team identity but also list the set of the team members who have participated in the submission. Do not put the name of students who has not done any work. If any student has a problem to participate in a particular submission, let me know and we can discuss how to handle the situation, and how to get this student the corresponding work.

Remember that all work must be yours. You have to explain things in your own words, and the code has to be written by you. You cannot use copy code from other sources (books, other solutions, or any other), and if you use ideas or material you have to put it in your own words or programming.

Add a bibliography section in your report and do the correct citations in the sections where you explain the ideas you have used from sources. It is a good practice to use sources and material, so explain it if you use them well. But do not copy from these sources.

Remember that you can discuss issues with other students and teams, but you cannot exchange code or text and used them as is. You can exchange ideas, but once you understand them, the team has to do your own version.