Practice 2 Lexical Analysis

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 automata to later implement them.

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.
- 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, return
- Non-essential characters: space, eol, tabs (and similar delimiter characters). These characters are used as delimiters.
- 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. These are the only exception to be handle of having same prefix.). Note that any typo may produce different words. For example: wile as a typo of while missing the h can be recognized as identifier instead of the keyword while. But note they can share a prefix. 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 lexeme (character or word) 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_OPERATOR: for all operators 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. One example can be an identifier that has a prefix, and hence it cannot be supported. Any lexeme that cannot be supported should be included in this category. Document well what options you consider in this case.

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.

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 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 in the design slides.

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 initializted 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 ($q_\varnothing$) 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 finish 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 impact measuresof. Any implementation is correct, even it is inefficient. But we are interested in measuring it.

So we want to measure how many operations are needed 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 with a COUNTCONFIG flag, so it can be eliminated in a RELEASE format 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 down significantly the program. So it is important to implement it at the preprocessor level.

So we request to implement at the preprocessing level the following constants:

1. COUNTCOMP: This counter should count the number of comparisons being made. If an instruction "if" (or any other) do 2 comparisons, it should count 2.
2. COUNTIO: This counter should count the input/output instructions, incrementing the counter by the number of characters read or write (to file or display, as these operations as much slower than moves in memory, and relates to the number of passes/reads you do) for each individual instruction.
3. COUNTGEN: This general counter should count each individual instruction of the rest of instruction types.
4. COUTNOUT: This flag indicates where to send the print messages. If COUNTOUT is OUT=1, the messages should be sent to the output file. When the COUNTFILE is DBGCOUNT=0, it should be created a new file just for the counting messages named in a string set in a configuration constant named COUNTFILE. The name of the file has to be <filename>.<ext>dbgcnt.

The informative messages should include:

1. The line number processed of the input file that the updates of the counter corresponds to.
2. The function name where this update is done.
3. The amount incremented
4. The partial counting inside this function for each COUNT flag
5. The total counting for each COUNT flag.

You should put the counter along the code so that it keeps incrementing how many instructions are being added as it passes to the different parts of the code. And as indicated, we want to know the amount added per function (partial value of the counter) to get a feeling of how many operations are needed per token type.

Where (in what file of your code) 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 idenfiable error message.

Reminder: The counting applies only when the format is DEBUG, but not in a RELEASE format.

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 Submission instructions

The same as Practice 1, except when you create the file submission, put P2 instead of P1.

10 Evaluation Criteria

The same as practice 1.

11 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.