CSC 403, Assignment 4

Due on 2 December at 11:59 pm

With apologies for people also taking compilers this term, this assignment will give you the opportunity to build a relatively small parser. Figure 1 on the last page shows the grammar of a simple yet pretty realistic programming language. This assignment asks you to construct a parser for this language, as follows:

1. We make the lexical analysis as simple as possible. For this purpose you are allowed to assume that all the lexical units in the language are separated by blank or newline characters from each other. This way you can tokenize the input at blanks without a need for regular expressions and finite automata.

   The end product of your simplified lexical analysis is a function `gettoken()` which will return the next lexical token in the input each time it is called. This function will need to return a structure rather than a single symbol. This structure will contain the type of the token and any other pertinent information you may need (you decide what is to be included).

   The possible token types are shown in the grammar from Figure 1 as capitalized words, parentheses, and operators. Note that the actual lexemes for the tokens IF, ELSE, WHILE, and BREAK are all in lower case. `BASIC` types are `bool`, `int`, `char`, and `double`.

2. You will note that the grammar as given in Figure 1 is *not* suitable for recursive descent parsing. Modify the grammar so that it can be the basis of a recursive descent parser. Include the modified grammar with your submission.

3. Define a suitable structure for a node in the parse tree. At the same time define suitable function(s) or operator(s) for printing a parse tree using a text-only representation. While I provide no particular instructions on how to print a tree, note that the tree will be inspected manually and so the printout must be as easy to read by humans as possible.

4. Implement a recursive descent parser for the language defined in Figure 1.

5. Implement a main function that takes a program from the standard input, parses it, and prints the resulting parse tree to the standard output.

Submission guidelines

The solution must be given as a Python 3 program and must not use external modules (only the standard library is required). The whole program must reside in a single file, or multiple files within a single folder. All the input must be read from the standard input stream (or a text file through an argument) and all the output must be produced to the standard output stream (e.g. print). No GUI or menu elements are allowed. Provide your submission by email.

Include as comments in the source code file your answer to Question 2, evidence of testing of the parser, and any other information about your program you deem necessary and/or interesting. Note that evidence of testing is compulsory.

Marking

Marking will be based exclusively on automated tests, except for a cursory code inspection to make sure that you have implemented a recursive descent parser, to establish ownership of the code, and to check out your documentation (including but not limited to your answer to Question 2). Clean code, good programming practice, etc. are therefore *not* part of the grading scheme. It is however crucial for your program to build and run using the following command line:

python my_parser.py file

where *file* contains the code to parse. If my tests fail for any reason including but not limited to build failure then zero marks will be given.

```
 ⟨program⟩  ::=  ⟨block⟩
   ⟨block⟩  ::=  { ⟨decls⟩ ⟨stmts⟩ }
   ⟨decls⟩  ::=  ⟨decls⟩ ⟨decl⟩   |  ε
    ⟨decl⟩  ::=  ⟨type⟩ ID ;
    ⟨type⟩  ::=  ⟨type⟩ [ NUM ]   |  BASIC
   ⟨stmts⟩  ::=  ⟨stmts⟩ ⟨stmt⟩   |  ε
    ⟨stmt⟩  ::=  ⟨loc⟩ = ⟨bool⟩ ;
             |   IF ( ⟨bool⟩ ) ⟨stmt⟩
             |   IF ( ⟨bool⟩ ) ⟨stmt⟩ ELSE ⟨stmt⟩
             |   WHILE ( ⟨bool⟩ ) ⟨stmt⟩
             |   ⟨block⟩
     ⟨loc⟩  ::=  ⟨loc⟩ [ ⟨bool⟩ ]   |  ID
    ⟨bool⟩  ::=  ⟨bool⟩ || ⟨join⟩   |  ⟨join⟩
    ⟨join⟩  ::=  ⟨join⟩ && ⟨equality⟩   |  ⟨equality⟩
⟨equality⟩  ::=  ⟨equality⟩ == ⟨rel⟩
             |   ⟨equality⟩ != ⟨rel⟩
             |   ⟨rel⟩
     ⟨rel⟩  ::=  ⟨expr⟩ < ⟨expr⟩
             |   ⟨expr⟩ <= ⟨expr⟩
             |   ⟨expr⟩ >= ⟨expr⟩
             |   ⟨expr⟩ > ⟨expr⟩
             |   ⟨expr⟩
    ⟨expr⟩  ::=  ⟨expr⟩ + ⟨term⟩
             |   ⟨expr⟩ - ⟨term⟩
             |   ⟨term⟩
    ⟨term⟩  ::=  ⟨term⟩ * ⟨unary⟩
             |   ⟨term⟩ / ⟨unary⟩
             |   ⟨unary⟩
   ⟨unary⟩  ::=  ! ⟨unary⟩
             |   - ⟨unary⟩
             |   ⟨factor⟩
  ⟨factor⟩  ::=  ( ⟨bool⟩ )
             |   ⟨loc⟩
             |   NUM
             |   REAL
             |   TRUE
             |   FALSE
```

Figure 1: A grammar for a simple programming language.