CS 530 Assignment 2 – Parser

Fall 2021 Section 01

Points: 80

You must work on your own for this assignment

Due Date: 11:00 PM on Tuesday, 30th November 2021

Late Submission Due Date: 11:00 AM on Thursday, 2nd December 2021

Please ask for help early for the assignment before the due date. **Do NOT expect help for your assignment like half day before the due time**.

Task

Part A

You will be writing the grammar rules in Extended Backus-Naur Form (EBNF) (e.g. Figure 5.15) for a simple recursive descent parser that can validate and evaluate expressions with the following specification:

1. Assume the parser is given a set of tokens passed from a lexical analyzer.
   The categories of the tokens are as follows:

   (a) Logical Operators (Binary, infix)

   - **and** is the AND operator
   - **or** is the OR operator
   - **nand** is the NAND operator
   - **xor** is the XOR operator
   - **xnor** is the XNOR operator

   (b) Dash Operator (Binary, infix)

   - - is the DASH symbol
   - The dash symbol we will be testing for is the em-dash symbol, which is the dash defined symbol through ASCII.

   (c) Relational Operators (Unary, prefix)

   - <
   - >
   - <=
   - >=
   - =
   - ! =
   - **not**

   (d) Integer Token

   - **int** is used to represent any integer value

You will need to use a non-terminal (e.g. < **relop** >) to specify the pattern for the relational operator tokens in EBNF form. You can then use this non-terminal wherever you need to specify a relational operator in EBNF form. You can refer to examples in 5.1.2.

You can use logical, dash and integer tokens directly in your EBNF form.

2. The expression (language construct) pattern is specified as follows:

   (a) **term** *{***op term***}* : can be just a term or any length as long as pairs of op and term are added.

   (b) A **parenthesis** pair may be used to group any **term** *{***op term***}* combination. For example, both of the following combinations are valid: • term op (term op term) op term

   - term op term op (term op term)

   An **op** can be one of the logical operators (specified above).

   (c) A **term** is specified as any of the following forms:

   - **int** - **int**
     - **int** is a token that represents an integer (specified above)
     - The use of **dash** operator indicates a range between two integers. **–** Example: 1 - 100
   - **relop int**
     - The use of **relational** operator indicates a range in relation to the integer. **–** Example: > 10
   - An expression specified in **2(a)** and **2(b)**.
     - This is important to build the expression recursively.
     - Refer Rules 10a, 11a and 12 in Figure 5.15.
     - Example: (1 - 100 or (> 100 and < 150)) and ! = 100

   Operator Precedence: The **dash** and **relop** operators have precedence over the **logical operator**. Note that operators within parenthesis would always have higher precedence.

You can have any number of rules as long as your rules sufficiently and accurately reflect what is specified in the specifications.

The following are examples of valid expressions based on the expression patterns specified above:

- 7 - 17
- > 90
- (1 - 100 and not 50) or > 200
- (7 - 17) or > 90
- > 50 or = 20
- 1 - 100 and ! = 50
- (5 - 100) and (not 50) or (>= 130 or (2 - 4))

The following are examples of invalid expressions based on the expression patterns specified above:

- >
- 2 - - 4
- - 7
- 7 -
- (! = 5) and
- 2 – 4 *and* >< 300
- >= 5) xnor < 10

Part B

Based on the grammar generated in Part A, you will be implementing the recursive descent parser in Python3.

A skeleton code (**rdParser.py**) has been provided on Canvas. The skeleton code has already implemented the lexical part of the compiler i.e. the input you will be working with has already been tokenized. You will be writing the code for the syntactic analysis part of the compiler i.e. parse the tokens to determine if the input expression is valid or not based on the above grammar rules. Additional information in comments is provided along with the skeleton code.

Please pay attention to the # IMPORTANT comment, certain part of the skeleton MUST NOT be changed, or the autograding tests will FAIL.

Programming Languages and Environment

- Programming Languages: You must use Python 3.6 for this assignment. This is available on edoras. Note that you should use **python3** on edoras to ensure that you are using Python 3. Note that using **python** on edoras would execute your code with Python 2.
- File Name: You must name your file **rdParser.py** for it to run with the autograder.
- Source Code Commenting: Your program should have appropriate comments at appropriate places. Refer to the best programming practices in the Syllabus on Canvas. Points will be deducted if your code does not have good comments.
- Plagiarism Check: Points will be deducted if there is plagiarism found in your code. An automated program structure comparison algorithm will be used to detect software plagiarism. An affidavit of academic honesty stating the program is written by you will need to be submitted along with your submission of the code.
  Additional details are provided in the Turning In section.

Turning In

Submit the following program artifacts through **Gradescope**. Make sure that all files mentioned below contains your name and Red ID!

- Text Artifacts
  - PDF/txt file which contains the EBNF grammar generated in Task A. It is recommended that you type it up, however, a scanned copy will be allowed as far as the handwriting is legible.
  - PDF/txt file which contains the Learning Outcomes for Assignment 2.
- Program Artifacts
  - Source code file: You MUST and ONLY submit the **rdParser.py** that implements Task B.
  - Academic Honesty Affidavit. Please type the affidavit in the comments at the beginning of the source code and type your name and Red ID as signature.

    Use the single-programmer affidavit template available on Canvas.

Late Submission Policy

Refer to the Syllabus for the Late Submission Policy.

Appendix - Testing Python Code

There are multiple ways to test and debug your code. A few of them are listed below:

- You can directly run **python3** on your terminal and test your code in interactive mode.

```
$ python3

>>> _
```

- You can write your code on edoras using text editors (vim/nano) and use the **python3** interpreter to test your code.
- You can use an IDE to write, test and debug your code. Popular choices include VSCode and PyCharm.

To test your function, you can instantiate a parser object from your **recDescent** class, then call **validate()** on the object and print out the return value.

```
r = recDescent('5 - 100') print(r.validate())
# your validate would first call lex()
# then call the top level parsing procedure and go from there
```

To debug your code, you can set a breakpoint in your IDE to check your program state and execute it line by line. Alternatively, you can use **pdb** (The Python Debugger) if you are working on command line. Note that **pdb** has a higher learning curve, though it is much more powerful.