CS 530 Section 01 Fall 2022

Assignment 3 – Parser

Part A

Write a set of grammar rules in the Extended Backus-Naur Form (EBNF) (e.g. Figure 5.15) for describing the syntax of expressions according to the following specifications.

1. **Categories of tokens** are:
   (a) Logical operators (Binary, infix) used for connecting two terms
      - **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) used for expressing a range of integer values
      - Only use the short **en-dash** - symbol from the ASCII standard.

   (c) Relational operators (Unary, prefix) used for expressing a range of integer values
      - <
      - >
      - <=
      - >=
      - ==
      - !=
      - **not**
      - As part of the syntax rules, use a non-terminal (e.g. < **relop** >) to specify the token pattern for the relational operators. Refer to examples in Chap. 5.1.2.

   (d) Integer
      - **int** is used to represent any integer value

2. The **expression syntax patterns** are specified as follows:
   (a) An **expression** is in the form of **term** *{***op term***}* where term is followed by 0 or many terms connected by an operator. The **op** can be one of the **logical operators** (specified above). Note curly brackets mean 0 or many times of repeating *op term*.

   (b) A **term** is specified as one of the following forms:
      - **int** - **int**
        - **int** is a token that represents an integer (specified above)
        - **dash** operator (en-dash) between two numbers is for representing an integer range

Example: 1 - 100

- Please note:
  - ONLY **positive** integers are considered here. + would NOT be a valid symbol.

- **relop int**
  - **Relational** operator **relop** represents an integer range in relation to the integer. Example: > 10
- **An expression in 2 (a) enclosed by a pair of parentheses**
  - This is important to build the expression recursively.
  - Refer to 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 should have precedence over the **logical operators**. Note that operators within the parenthesis would have higher precedence than the ones outside the parenthesis.

You can come up with any number of EBNF grammar rules describing the expression syntax as long as your rules sufficiently and accurately reflect what is specified in the above specifications.

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))

Examples of **invalid** expressions:

- >
- 2 - - 4
- - 7
- 7 -
- = 6
- (!= 5) and
- 2 - 4 *and* >< 300
- >= 5) xnor < 10

Part B (60 points)

Use **Python3** to implement a simple **recursive descent parser** for **validating expressions** based on the **grammars** designated from **Part A**.

The essence of the recursive descent parsing is to have ONE parsing procedure for each specified grammar rule (or non-terminal) starting from the outermost grammar rule, and parsing procedures can be invoked recursively based on the circular references between grammar rules (e.g., Figure 5.15 rules 10a, 11a, 12).

- **Hints:**
  - Please refer to the **recursive descent parsing algorithm in Figure 5.17**
- Recursive descent parsing uses a top-down approach, starting from the top of the parse tree, then drilling into lower hierarchical levels.
- The procedures work together to handle all combinations of the grammar rules, and they automatically handle the nested compositions of terms with multi-level priority brackets.

Important:

For this assignment, although you do NOT need to strictly follow ONE parsing procedure per grammar rule; **at least TWO parsing** procedures should be implemented: **one for the expression rule and one for the term rule.** Otherwise, **20% penalty** on the overall grade will be applied.

A skeleton code (**rdParser.py**) has been provided on Canvas. You **must NOT change the file name rdParser.py**; otherwise, auto-grading will fail automatically. The skeleton code has already **implemented the lexical part** of the compiler, which turns the expression into a list of tokens. You will then perform the **syntactic analysis** to determine if the **structure** of these tokens is **valid** based on the Part A grammar rules.

Additional hints in comments are provided along with the skeleton code.

Do **NOT** change the skeleton code other than adding the recursive descent parsing logic for each grammar rule and using the parsing logic in the **validate(self)** for validating the input expression. You would also want to some **testing** code, see **Appendix** at the end.

Programming Languages and Environment

- Programming Languages: You must use Python 3 for this assignment. It is available on Edoras. Note that you should use the **python3** command to ensure that you are using Python 3 for compiling and debugging your code. Using **python** would execute your code with Python 2.

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 with the command line. Note that **pdb** has a higher learning curve, though it is much more powerful.