T-603-THYD Compilers
Project: A *parser* for ON

Objective

The objective of this project is to get you starting with writing a compiler/interpreter. The focus of this part of the project is on *syntax analysis*.
You can work on the project in a group of two.

Description

We will develop an interpreter/compiler for a sizeable part of the Python programming language in the course. We will start with a much-simplified language, but as the term progresses and our knowledge expands, we will gradually add new program constructs to the language. The first version of our programming language we encounter is ***ON***.

The programming language ***ON*** supports basic statements like assignments and *if*- and *while*-statements. It also allows us to form complex expressions using Boolean, arithmetic, and comparison operators. For the most part, the constructs are legitimate Python constructs, *except* that we:

- are using curly brackets and parenthesis to enclose blocks and test conditions, respectively;
- have mandated the use of semi-colons to end simple statements (the statements can spawn more than one line);
- have introduced a new keyword ***let*** in assignment statements.

Those exceptions make tokenization and parsing easier. However, we will remove those exceptions in the following language version and use proper Python syntax.

The appendix gives the detailed syntax of the ***ON*** programming language. Study the grammar in the appendix carefully to understand the language syntax. In particular, look out for potential ambiguity in the grammar.

Your task is to write a ***top-down recursive-descent parser*** for the programming language ***ON***. A complete lexer and a parser skeleton are provided. Please finish the parser implementation. To simplify testing, you are not allowed to change the interface to the parser and the lexer.

If the input is syntactically correct, your parser should return an ***abstract-syntax tree (AST)***, otherwise report a syntax error. I recommend that you first ensure that the parser parses correctly, and only then you start thinking about creating the AST.

In Canvas, **hand-in** the Python file *onparser.py*. There are two hand-in milestones. The first one is optional, where you hand in a parser that parses correctly, and for the second, you hand in a parser that both parses correctly and creates the AST. If you choose to take advantage of submitting for the early milestone, you will get feedback on whether your parsing is correct, which you can correct (without any penalties) for your final submission.

APPENDIX: The syntax of the *ON* language *(expessed in EBNF-like style)*

program: *stmt** **EOI**

stmt: *simple_stmt* '**;**' | *compound_stmt*

simple_stmt: *assign_stmt* | *pass_stmt* | *break_stmt* | *continue_stmt*

compound_stmt: *if_stmt* | *while_stmt*

assign_stmt: '**let**' identifier '=' *expression*

pass_stmt: '**pass**'

break_stmt: '**break**'

continue_stmt: '**continue**'

if_stmt: '**if**' '**(**' *expression* '**)**' *block* ('**elif**' '**(**' *expression* '**)**' *block*)* ['**else**' *block*]

while_stmt: '**while**' '**(**' *expression* '**)**' *block*

block: *stmt* | '{' *stmt** '}'

expression: or_expression

or_expression: *and_expression* ('**or**' *and_expression*)*

and_expression: *not_expression* ('**and**' *not_expression*)*

not_expression: '**not**' *not_expression* | *comparison*

comparison: arithmetic_expr [*comparison_op arithmetic_expr*]

comparison_op: '<' | '>' | '==' | '>=' | '<=' | '**!**='

arithmetic_expr: *term* (('+'|'-') *term*)*

term: *factor* (('*' | '/' | '**%**' | '//') *factor*)*

factor: ('+'|'-') *factor* | *atom*

atom: *variable* | ***NUMBER*** | ***STRING*** | '**None**' | '**True**' | '**False**' | '**(**' *expression* '**)**'

variable: ***IDENTIFIER***

- ***NUMBER*** is token identifying integer-literals (one or more *digits*), e.g., *42*.
- ***IDENTIFIER*** is a token consisting of one or more *alphanumeric* and *understore* characters; an identifier is not allowed to start with a *digit*. (e.g., *name_10_beers*).
- ***STRING*** is a string-literal, starting and ending with a pair of either single (') or double (") quotes, e.g. *'A string'*, *"Another string"*, *'A string with " in it.'* , *"A string with 2 ' and ' in it."*). Strings are not allowed to span over lines and do not (yet) support *escaping* characters.
- An ***ON*** source file may include Python-style comments (starting with #).