Compiler Construction (CS F363)
Project - Stage 2

Complete Design Specs and Syntax-Directed Translator Design

SUBMITTED BY:

Group 25

Kushal Joseph - 2019A7PS0135G
Dev Goel - 2019A7PS0236G
Rohan Jakhar - 2019A7PS0174G
Pranay Tambi - 2019A7PS0171G
Shivam Singhal - 2018A7PS0214G
Harsh Singhal - 2018B1A70347G

STUDENTS AT

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

27th April, 2022

Final Design Specs

Changes to the Design from Stage-1

Before we describe our Syntax-Directed Translation Scheme, we decided to document some of the changes we have made from our Stage-1 Submission. After discussions, we decided to alter certain aspects of our design (Game-Design, i.e, features of our language, etc). Some of them include:

- We have changed the syntax of defining levels in our language. Earlier it was:

  ```
  level level_name {
    <Code>
  }
  ```

  Now we have made it simpler of the form:

  ```
  level levelname>
      <Code>
  ```

- We have decided to reduce the overall complexity of our implementation by removing certain features which we believe were unwanted:
  - Simultaneous blocks
  - Bonus feature for block
  - Skip-block feature
- We decided to integrate all levels. This means that there will be a smooth transition in the game, and levels will automatically change when a certain number of “lines” are completed. So we have changed to “passlevelscore” keyword to “passlevellines”.
- For this submission, we have implemented “moveconfig” in our parser, i.e, but we have ignored it in our game. We simply always use arrow keys for movements.

Syntax Directed Translation Scheme

Parser Generator Specs

We are using the **Yacc** Parser generator for our Tetris Compiler. As specified, we are using Python's PLY library for the same, it is essentially a Lex-Yacc program written in python. The process of generating a parser generator is similar to the original C-style Lex Yacc, in terms of writing grammars, specifying actions, etc.

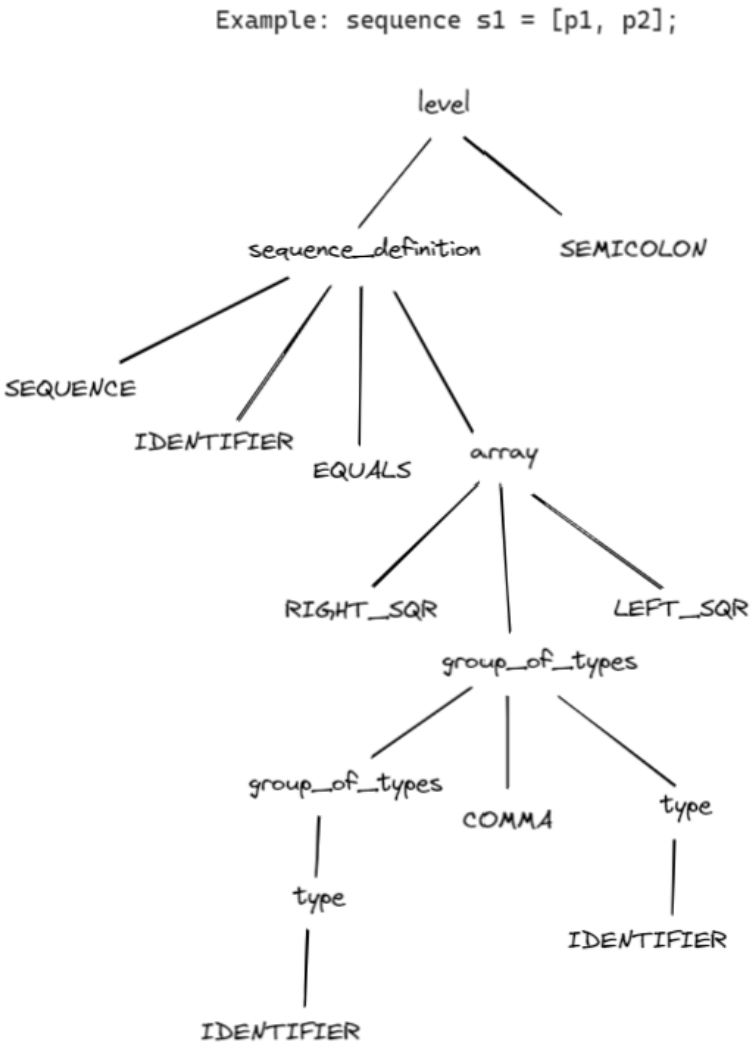
Yacc uses a parsing technique known as **LR-parsing** or shift-reduce parsing. LR parsing is a technique that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action code is triggered and the grammar symbols are replaced by the grammar symbol on the left-hand-side.

LR parsing is commonly implemented by shifting grammar symbols onto a stack followed by looking at the stack and the next input token for patterns that match one of the grammar rules.

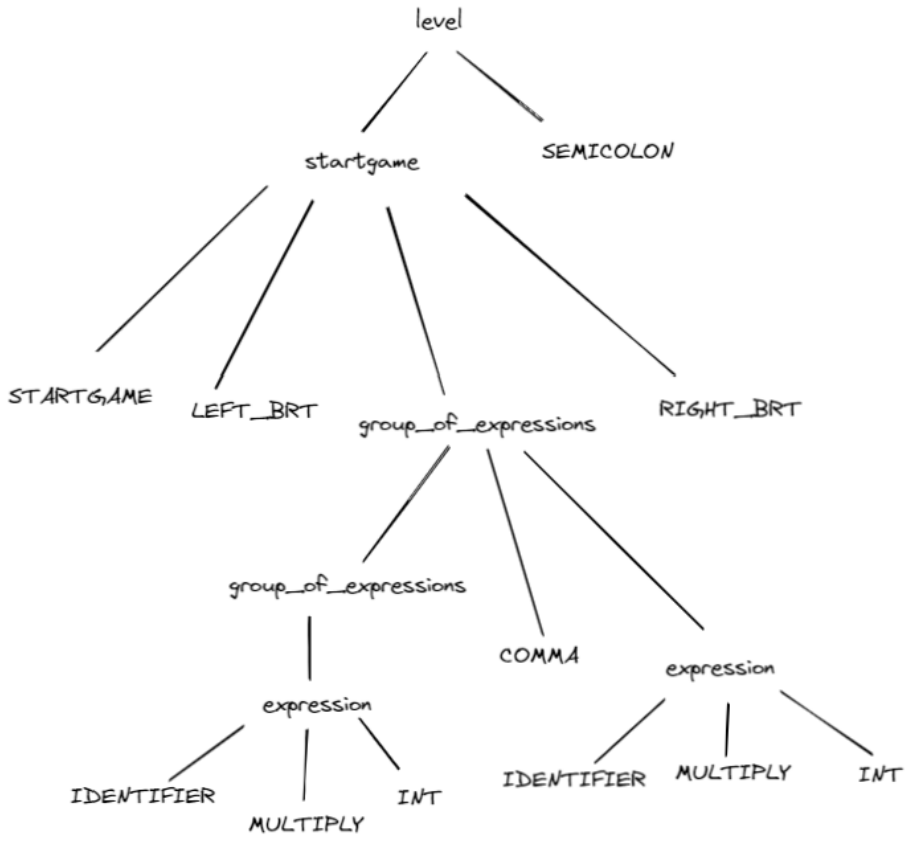
In **PLY**, we write a function for each non-terminal which is of the nomenclature p_name_of_non_terminal(p):
The **docstring** (enclosed in ‘‘‘ ’’’ python comments) must contain the **grammar rules** for that particular non-terminal. The actions may be written in Python code after.

Top Down Parser Recursion Tree

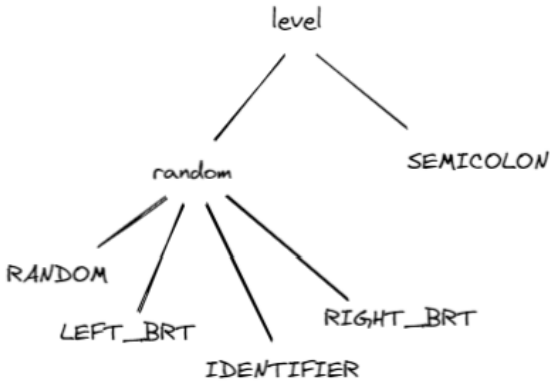
Based off the grammar rules we have written for our syntax, we present some example input strings and their top-down recursion parse trees:
We have shown several examples denoted by - **Example:** ***<input string>***



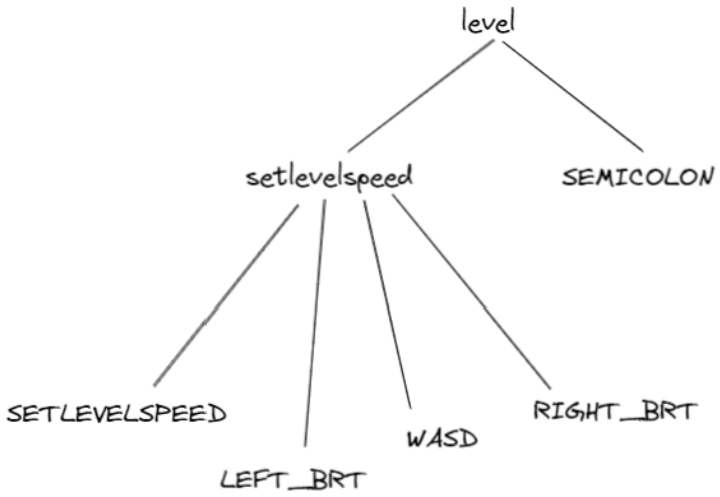
Example: startgame(s1 * 10, s2 * 3);



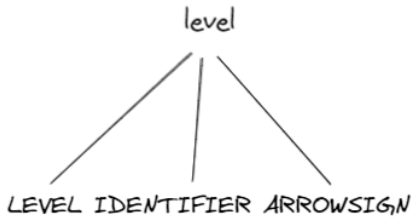
Example: random(s1);



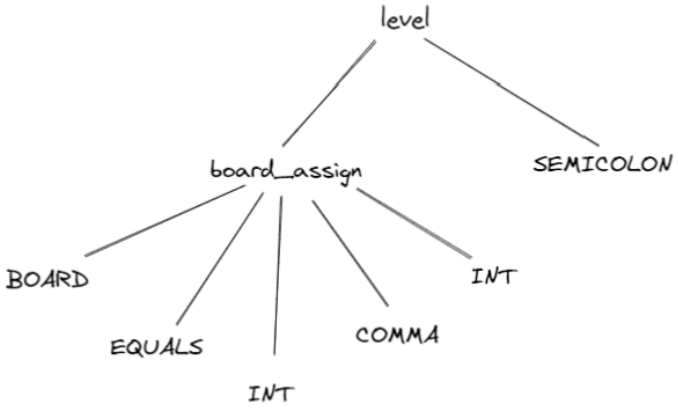
Example: setlevelspeed(1);



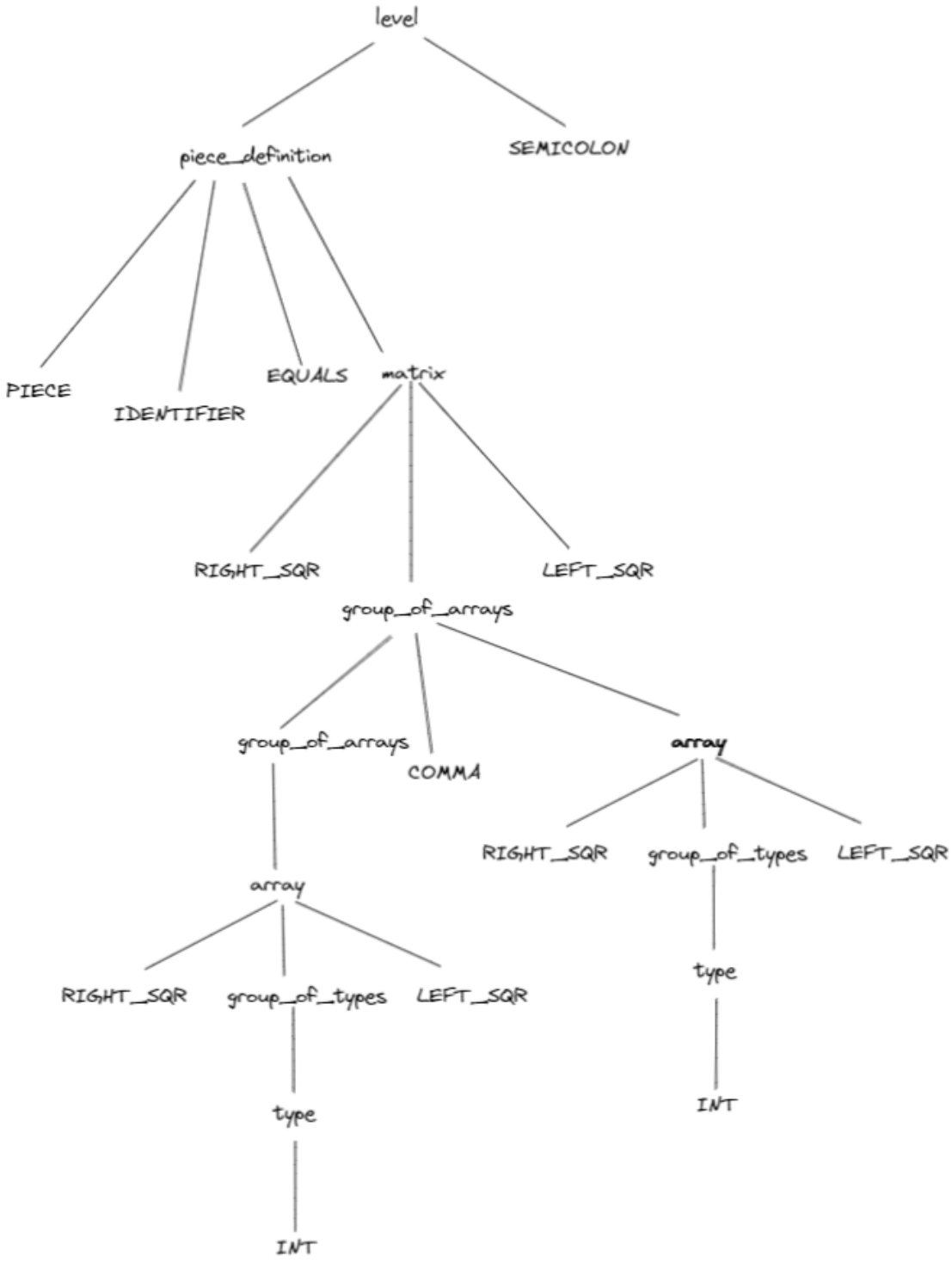
Example: level l1>



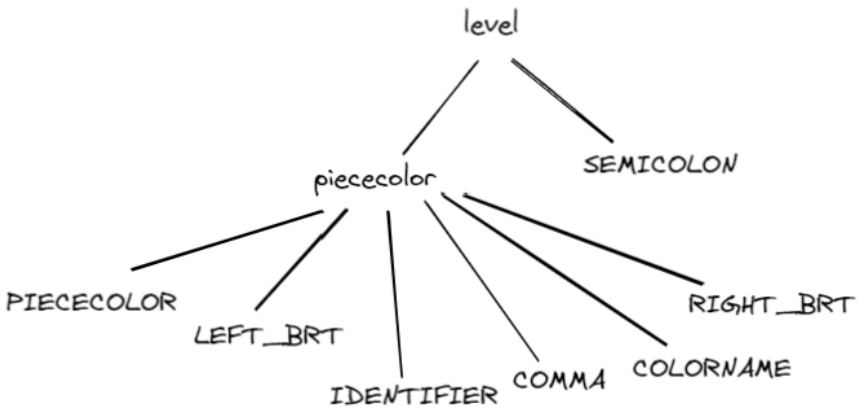
Example: board = 8, 8;



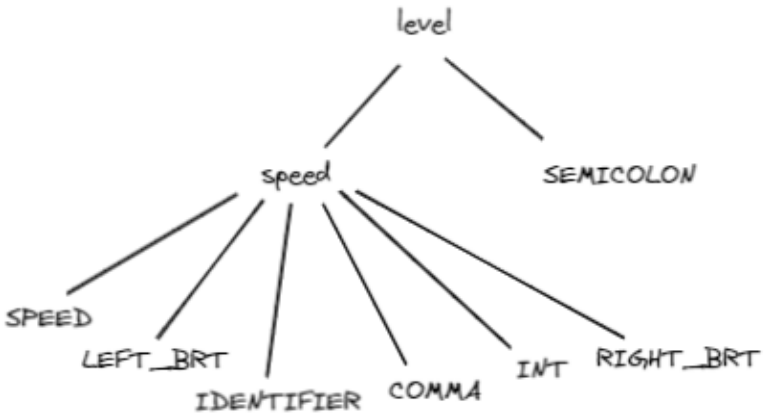
Example: piece p1 = [[1, 1, 0], [0, 1, 0]];



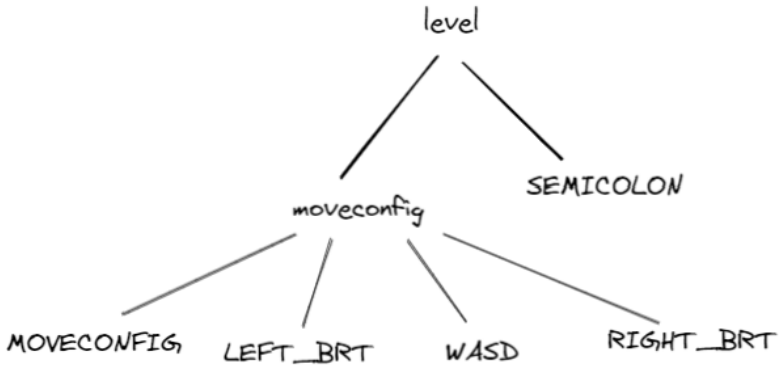
Example: piececolor(p1, 'blue');



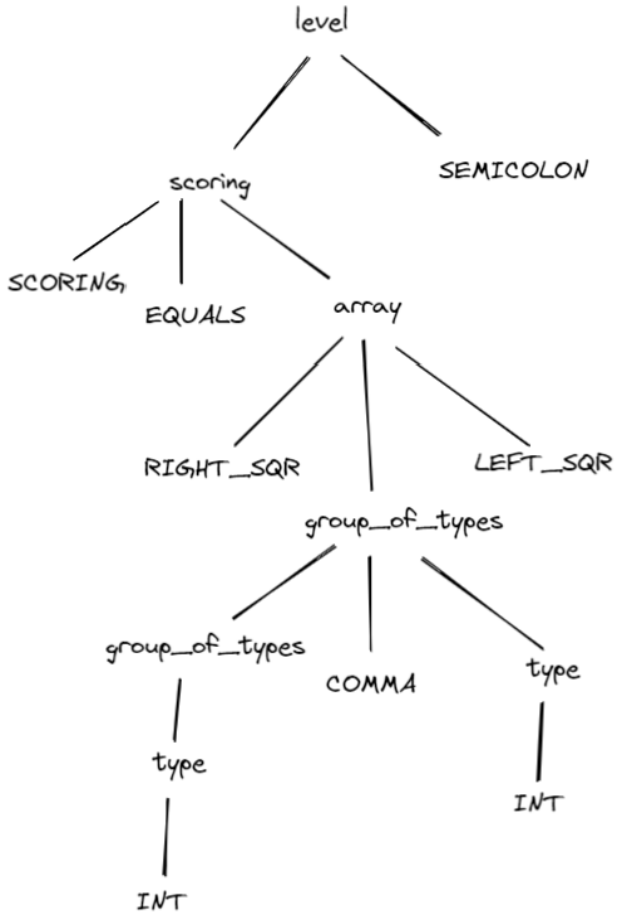
Example: speed(p4, 5);



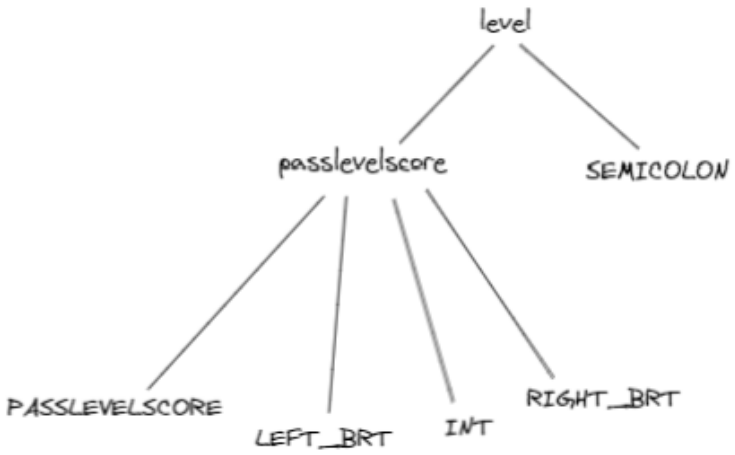
Example: moveconfig(WASD);



Example: scoring = [100, 250];



Example: passlevelscore(150);



Challenges

Writing a grammar for Matrices and Arrays:

An array is a list of comma separated numbers enclosed within square brackets. For this purpose, we separately defined a grammar for “group of numbers”, and then defined our arrays to be `[ group of numbers ]`, where [and] are terminal tokens

Similarly, matrices (used for piece definitions) are groups of comma-separated arrays enclosed within square brackets. The (simplified) grammar for matrices looks like this: `[ group of arrays ]`, where we had to write a separate grammar for group of arrays.

To see all the grammars written in our implementation, please see the “tetris_compiler.py” file on our GitLab submission page.

Defining the Starting symbol:

We had a slight challenge with deciding which non-terminal to use as the starting symbol. We had to choose between “level” and “tetris_compiler”, which is now commented out (Please see our parser code). The main issue arose due to our code

being segregated into “levels”, and the fact that the parameters could be different in all the different levels. We found it difficult to write a working grammar for our prior level declaration (see point 1 in changes, at the beginning of this doc). Finally we decided to go with “level” as our starting non-terminal, and also as mentioned earlier, to change our syntax for levels.

Grammar for our “Startgame List”:

Our start game list in our language defines the order and multiplicity of the sequences defined and how they will flow in the gameplay. The syntax is of the form:

```
startgame(s1 * 20, s2 * 5, s3 * 15);
```

Where s1, s2, s3 are predefined sequences of blocks, and the multiplicity defines how many times the sequence will fall.

Here, similar to arrays, we have a “group” of “expressions”. And also, the last expression must not be followed by a comma, but every other one must. So the appropriate grammar would be:

```
group_of_expressions : group_of_expressions COMMA expression
                     | expression
startgame : STARTGAME LEFT_BRT group_of_expressions RIGHT_BRT
```

Where the words capitalized are terminal tokens.

Test Cases

We tested for multiple test cases on our parser design by providing different kinds of statements which our lexer-parser combination can perform. These tests are performed on statements which are successfully accepted in our language as well as those on which our lexer fails. Some of the test cases we ran are listed below with a brief description of the errors generated by our parser:

Our compiler detects basic syntax errors:

1. Missing a semicolon:

```
piece p4 = [[1, 1], [0, 1], [1, 1], [1, 1]]
piece p5 = [[1]]
piece p6 = [[1, 1, 0, 1, 1], [0, 1, 1, 1, 0], [0, 0, 1, 0, 0], [0, 0,
1, 0, 0]];
```

   As we can see, the above 2 lines do not end with semicolon, and the error is correctly pointed out:

```
(base) C:\Users\kusha\OneDrive\Desktop\Sem 3-2\Compilers\Project>python tetris_compiler.py
<tetris.Level object at 0x000002ADD2EF3190>
level
('board', 22, 20)
WASD
('moveconfig', 'WASD')
('setlevelspeed', 20)
None
None
('piece', 'p1', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 1, 0], [1, 0, 1]])
('piece', 'p2', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 0, 0], [1, 0, 0]])
('piece', 'p3', [[0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 1, 1]])
Syntax error in input! Exiting!
```

2. Mistake in Array/Matrix Syntax:

```
piece p1 = [[1, 1, 1] [1, 0, 1], [1, 1, 1], [1, 1, 0], [1, 0, 1]];
```

   In the above code, we are **missing a comma** between 2 arrays in a matrix. We correctly identify the error:
   This is the first piece defined, hence the other pieces are not printed

```
(base) C:\Users\kusha\OneDrive\Desktop\Sem 3-2\Compilers\Project>python tetris_compiler.py
<tetris.Level object at 0x000001AC25523190>
level
('board', 22, 20)
WASD
('moveconfig', 'WASD')
('setlevelspeed', 20)
None
None
Syntax error in input! Exiting!
```

3. Spelling mistake in keyword:

```
speed(p4, 5);
sped(p1, 1);
```

The 2nd statement in the above input shows a spelling mistake in the keyword ‘speed’, which is caught by our parser and reported as an error.

```
(base) C:\Users\kusha\OneDrive\Desktop\Sem 3-2\Compilers\Project>python tetris_compiler.py
<tetris.Level object at 0x000002268DFA3190>
level
('board', 22, 20)
WASD
('moveconfig', 'WASD')
('setlevelspeed', 20)
None
None
('piece', 'p1', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 1, 0], [1, 0, 1]])
('piece', 'p2', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 0, 0], [1, 0, 0]])
('piece', 'p3', [[0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 1, 1]])
('piece', 'p4', [[1, 1], [0, 1], [1, 1], [1, 1]])
('piece', 'p5', [[1]])
('piece', 'p6', [[1, 1, 0, 1, 1], [0, 1, 1, 1, 0], [0, 0, 1, 0, 0], [0, 0, 1, 0, 0]])
None
None
('speed', 'p4', 5)
Syntax error in input! Exiting!
```

4. Missing the **quotes** in the color string declaration:

```
piececolor(p1, 'blue');
piececolor(p2, 'red');
piececolor(p3, green);
piececolor(p4, 'orange');
```

The 3rd statement in the above input does not surround the color input of green in single quotes (‘ ‘), as is required by our parser.

```
(base) C:\Users\kusha\OneDrive\Desktop\Sem 3-2\Compilers\Project>python tetris_compiler.py
<tetris.Level object at 0x00000279D2B23190>
level
('board', 22, 20)
WASD
('moveconfig', 'WASD')
('setlevelspeed', 20)
None
None
('piece', 'p1', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 1, 0], [1, 0, 1]])
('piece', 'p2', [[1, 1, 1], [1, 0, 1], [1, 1, 1], [1, 0, 0], [1, 0, 0]])
('piece', 'p3', [[0, 0, 1], [0, 0, 1], [0, 0, 1], [0, 0, 1], [1, 1, 1]])
('piece', 'p4', [[1, 1], [0, 1], [1, 1], [1, 1]])
('piece', 'p5', [[1]])
('piece', 'p6', [[1, 1, 0, 1, 1], [0, 1, 1, 1, 0], [0, 0, 1, 0, 0], [0, 0, 1, 0, 0]])
None
None
('speed', 'p4', 5)
('speed', 'p1', 1)
('piececolor', 'p1', "'blue'")
('piececolor', 'p2', "'red'")
Syntax error in input! Exiting!
```

5. Feeding **wrong data type in array**:

```
sequence s2 = [p3, p4, p6, p5];
sequence s1 = [p1, 22p, p2, p3];
```

Sequences expect “identifier” array elements. In the above code, “22p” will not be recognized as an IDENTIFIER since it begins with an integer.

```
('piececolor', 'p2', "'red'")
('piececolor', 'p3', "'green'")
('piececolor', 'p4', "'orange'")
('piececolor', 'p5', "'red'")
('piececolor', 'p6', "'orange'")
None
('sequence', 's2', ['p3', 'p4', 'p6', 'p5'])
Syntax error in input! Exiting!
```

Hence, our compiler catches all common syntax errors based out of our language syntax
Semantic errors will be caught later inside the config/game file. For example, if the color is not available, if the piece is too big, or other variable related errors.

Complete End-to-End Tetris Game Engine Programming Toolchain

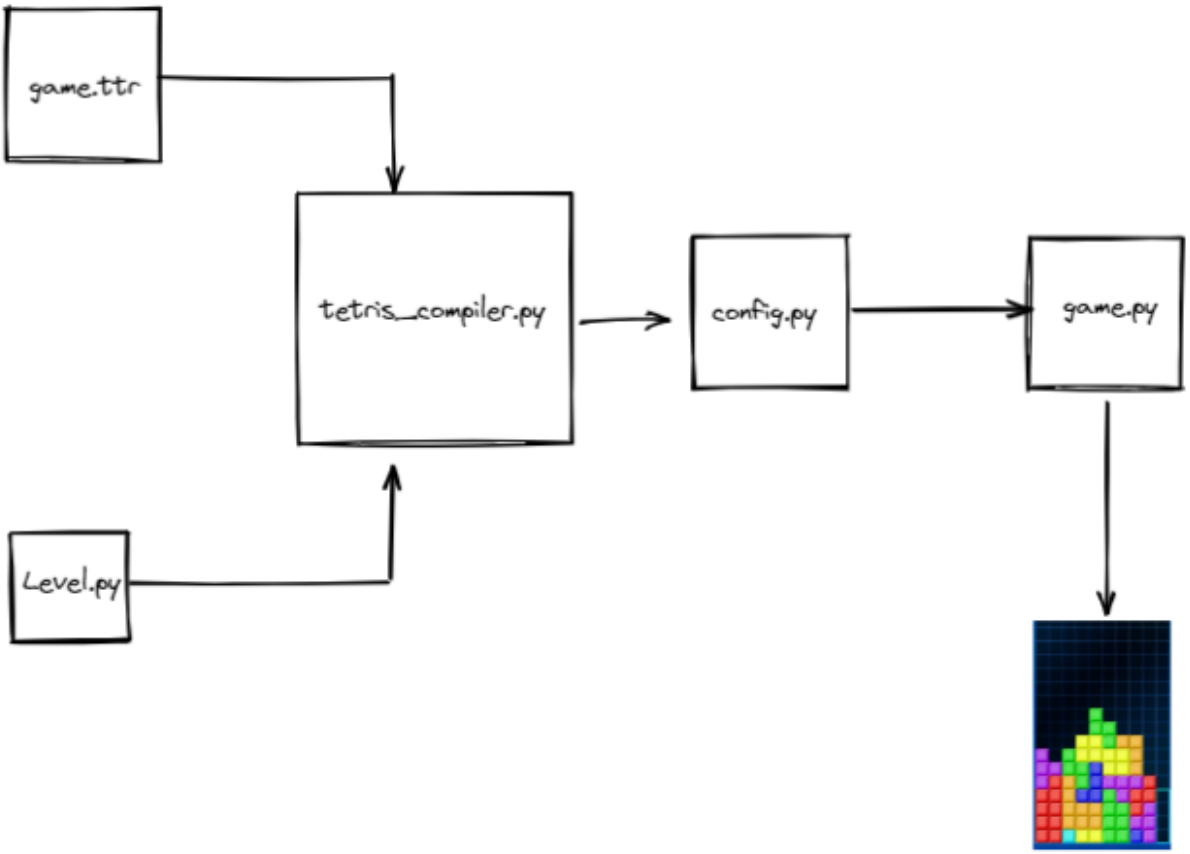
We aimed our Tetris Language for a general audience, hence we made the End-to-End tetris game engine toolchain fairly straightforward and simple.
Since we are using Python, and PLY, we harness Python imports to directly use code from a file on another. So we are able to directly connect our compiler (parser) output with our game, hence, the user can: Compile and Run with a single command

The files used in the backend are:

- **tetris_compiler.py**: This is the main compiler file, which includes Both the Lexer and the Parser. In PLY, we get the flexibility of writing both the lexer and the parser in the same python file.
- **game.py**: Here, we implemented our backend Tetris game
- **config.py**: An intermediate file storing some global variables and configuration parameters for levels and the game in general
- **Level.py**: A simple class implementation of “Level” class, with the attributes for a level

The user only has to run “tetris_compiler.py”, which would use the other 3 files and directly start the Tetris game.

A diagrammatic representation of the same is shown:



Please note that all files are already present and the diagram doesn't indicate that tetris_compiler *creates* config.py, it just uses the functions in config.py, which in turn uses game.py to create the game window directly.

Tetris Game Screenshots

Final Game view with an attached screenshot:
To run our code, please use/modify our uploaded game_final.ttr file, and simply run the tetris_compiler.py file using python.

If you wish to change the game program's name, make sure to change the input file name in **tetris_compiler.py** at line number **362**