A report on

# Design and Development of Language Processors for a Game Programming Language for making Tetris Engines

BY

Devansh Dixit (2019A7PS0069G)

Manthan Asher (2019A7PS0144G)

Rajath V (2019A7PS0122G)

Arnav Jain (2019A7PS0158G)

Sanchet Nagarnaik (2019A7PS0001G)

Taarush Bhatia (2019A7PS0159G)

**In partial fulfillment of the compiler construction course**

## Table of Contents

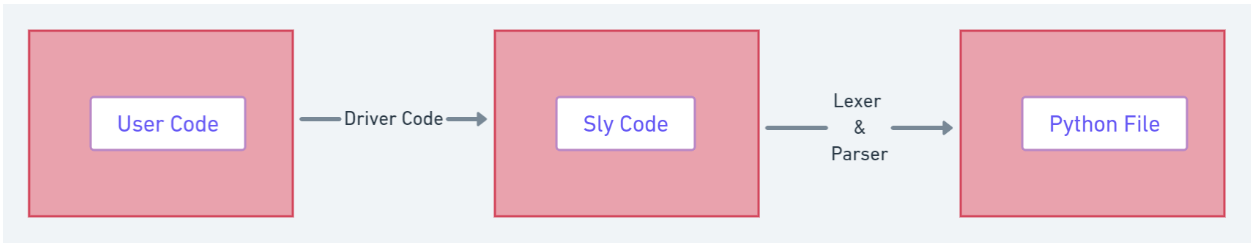
### Disclaimer:

We have just given a bare-bones structure as to how we will go about designing our game programming language. We have given the Top-Level Design Specifications and also given the Design of our Scanner. We have also done a **basic implementation** of our Lexer through sly and uploaded the code file on Gitlab as well. We have attached a screenshot of the tokenization of our code through Lexer. As we go to the later stages of our project, we may encounter new challenges as well as get hold of new ideas and hence this documentation is subject to change.

## Top Level Design Specs

### 1. Overall Program Structure

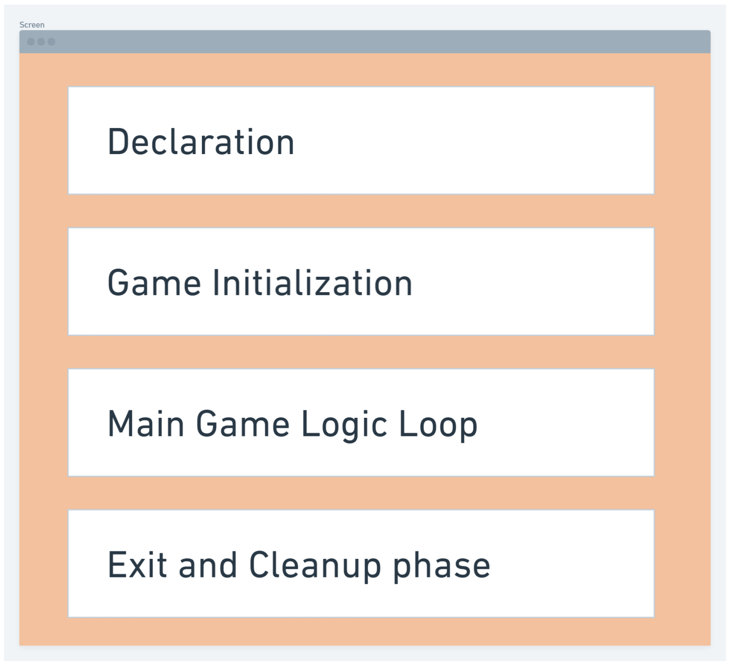
Our Tetris implementation will be based on a two-phase system, where we will primarily use the SLY library for lexical analysis and parsing, and the code that we intend to generate would be in Python, from where a Python executable will be generated that will then run Tetris:



Our tentative plan for the implementation is to have the following flow:

**Custom user code -> Python Library to scan and parse the code -> Target code in python -> Game execution.**

The code written in our language will be fed into the lexer and parser i.e the SLY library. Then the parser converts it to appropriate python code endpoints as specified in the grammar we will devise, and then that code is executed to get our game engine running.



The program structure we have decided upon has the following components associated with it:

- **Variable Declaration phase -** This section of the program is associated with initializing variables, constants and other parameters that we might require to get the game running.
- **Game initialization phase -** This part constitutes the setting up of the board for the game, initializing the different dynamic states of the same as well as setting up some features of the game including the queue for the tetrominos - (Next queue, Hold queue) and the like.
- **Main game logic loop phase-** This part is where the game logic in its entirety would be implemented. This is where we plan to implement the random tetromino generation, score calculations, game window initialization, handling how the tetrominos behave (i.e rotation, speedup, hard drop, soft drop, T spins and so on) and the overall running of the game. In this phase we show all the relevant features on the screen as well, including the score, a few of the next tetrominos in the queue, and the lines, level etc.
- **Game termination, cleanup and exit phase -** This is invoked when the game ends or if the user decides to voluntarily stop the game. We display the score once finally and then end the program.

## 2. Offered Primitives/ Features

The matrix, where the gameplay occurs, will be a 10 x 20 matrix of cells.

Choice of Tetris variant will be given before the game starts. There will be 2 choices

1) **Marathon** - This is the traditional game of tetris
2) **Sprint** - The player chooses a starting level and completes to clear a set number of lines in the shortest amount of time.

User controls while tetromino in play:

a. Game Direction Alignment
    i. **Left Arrow key -** moves the Tetrimino to the left by one cell at a time.
    ii. **Right Arrow key -** moves the Tetrimino to the right by one cell at a time.
b. Rotation (Clockwise and Anticlockwise)
    i. **Up arrow key -** Rotates clockwise
    ii. **Ctrl -** Rotates anticlockwise
c. Hard drop
    i. **Space key -** Drops the Tetrimino immediately and locks it down on the surface
d. Difficulty: based on speed
    i. **Easy(Level 1)** - The tetriminos will drop at a base speed i.e 1 second per line
    ii. **Medium(Level 2)** - The tetriminos will drop at a speed 2x the base speed
    iii. **Hard(Level 3)** - The tetriminos will drop at a speed 4x the base speed

The Information relevant to the game that will be displayed on-screen :-

a. **Next block -** It shows the Tetrimino which will appear next

b. **Statistics -** It shows all the Tetriminos which can appear in our game and their frequencies/number of times they have appeared in the game
c. **Score block -** It shows your current score.
d. **Help block -** It gives an overview of all the keys that the user can use and what they will do.

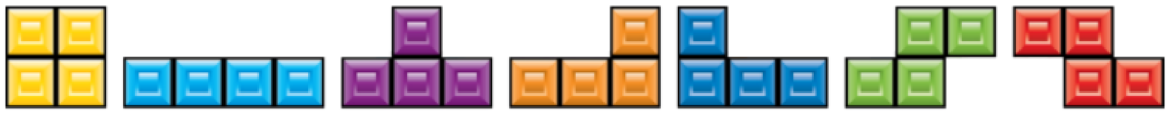
### Scoring system

a. (100 x level) points - 1 line of block is cleared
b. (300 x level) points - 2 lines of block are cleared simultaneously
c. (500 x level) points - 3 lines of block are cleared simultaneously
d. (800 x level) points - 4 lines of block are cleared simultaneously

### Game over conditions

1. **Lock Out (Game Over)**
   This Game Over Condition occurs when a whole Tetrimino Locks Down above the Skyline.
2. **Block Out (Game Over)**
   This Game Over Condition occurs when part of a newly-generated Tetrimino is blocked due to an existing Block in the Matrix.

## Tetriminos



- O-Tetrimino: yellow; a square shape; four blocks in a 2×2 square.
- I-Tetrimino: light blue; shaped like a capital I; four blocks in a straight line.
- T-Tetrimino: purple; shaped like a capital T; a row of three blocks with one added above the center.

- L-Tetrimino: orange; shaped like a capital L; a row of three blocks with one added above the right side.
- J-Tetrimino: dark blue; shaped like a capital J; a row of three blocks with one added above the left side.
- S-Tetrimino: green; shaped like a capital S; two stacked horizontal dominoes with the top one offset to the right.
- Z-Tetrimino: red; shaped like a capital Z; two stacked horizontal dominoes with the top one offset to the left.

## 3. Modes for Programmable Features

1. Built-in Functions:

   - **getBoard(rows, columns):** Returns a board with specified number of rows and columns.

   - **getName()**: Returns a String name given as input by the player to be stored as a global variable in our initialisation phase.

   - **getNextTetromino( ):** Returns the next Tetromino/Shape to be added to the queue of Tetrominoes in the game.

   - **setSpeed( level ):** Sets the speed of the incoming tetrominoes based on the level flag(EASY, MEDIUM, HARD) passed by our game initialisations.

   - **setGameMode( mode ):** sets the game mode based on a flag( NORMAL, SPRINT ) passed by our game initialisation.

   - **rotateRight()** : rotates the currently dropping tetromino 90 degrees to the right.

   - **rotateLeft()** : rotates the currently dropping tetromino 90 degrees to the left.

   - **moveRight()** : moves the currently dropping tetromino one block to the right.

- **moveLeft()** : moves the currently dropping tetromino one block to the left.

- **softDropOnLongPress( int key )**: drops the currently dropping tetromino 20 times faster on long pressing key X

- **hardDrop()** : drops the tetromino directly to its place on the board based on its current alignment.

- **checkClearedLine( line_no )**: check the line passed as line_no filled by the tetrominoes on board and return if its cleared or not.

- **clearLine( line_no )**: clear the specified line as line_no.

- **getCharacter( )**: take a character input and return corresponding flag( ARROW_RIGHT, SOFT_DROP, ARROW_LEFT, CLOCKWISE, ANTI_CLOCKWISE, HARD_DROP)

- **display(String literal)**: display the passed string on terminal

- **displayUpcomingTetromino(Tetromino t):** The given tetromino will be displayed under the upcoming tetrominoes.

- **displayTetromino( position ):** display currently dropping tetromino on specified position.

- **displayBoard()** : display board on terminal

- **addScore(String name,int score)**: adds the passed name and score to the leaderboard.

- **displayLeaderboard()**: displays the current leaderboard of the game.

- **clearScreen()**: clears terminal screen
- push(array a, val v):

- pop(array a):
- rem(array a, int index):
- len(array a):

2. Loops:

- **While(condition):** basic implementation of the generic while loop.

- **timeout( time ):** a while(true) loop will run till specified time in milliseconds has run out.

3. Conditionals:

- **IF <condition> <statement>:** We think we will try to implement the basic if statement, if successful might go on to try implementing the if else structure.

4. Built-in Types:

- **Integer type - int:** For score calculations and other local calculations.

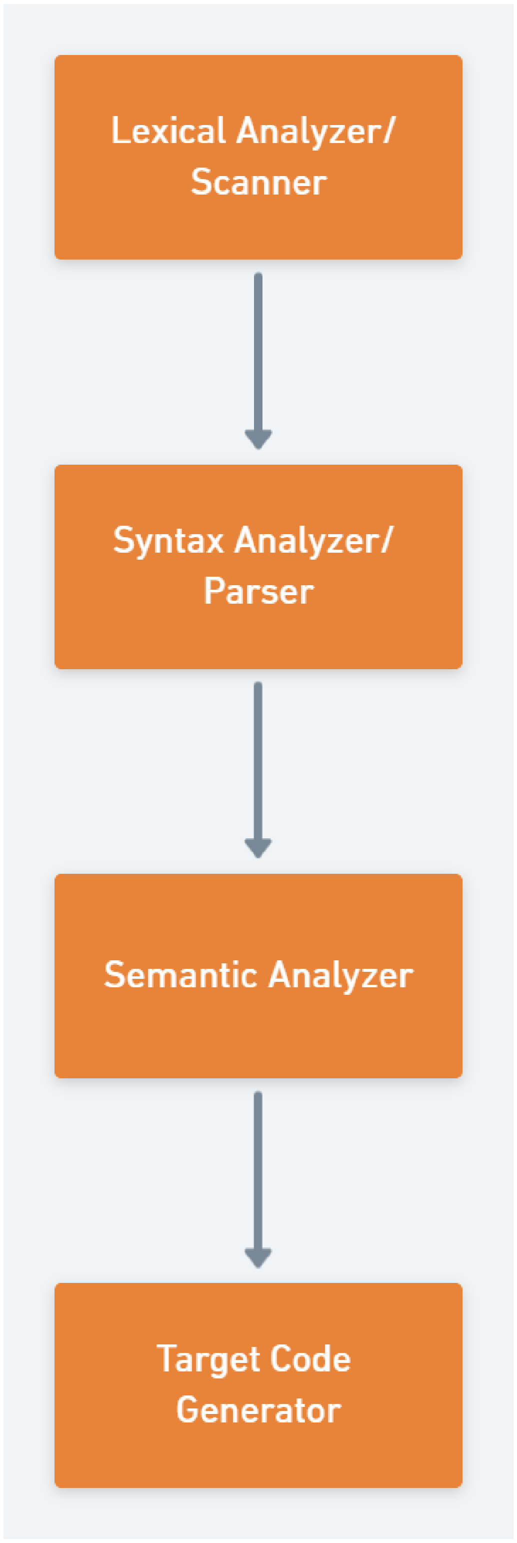
- **String type - str:** For storing certain variables to be displayed during the game.

- **Board type - board:** To store the current board in a variable.

- **Tetro type - tetro:** To store a tetromino as a variable.

- **Array type - array:** To store an array of the upcoming tetrominoes and other implementations for which we might need it.
    - Indexing with '[]'
    - length property
    - rem(index i) function to remove element at given index

4. **Pipeline Schema**



**Lexical analyser / Scanner:** It reads the characters from the source program and groups them into lexemes which correspond to a token. Tokens are defined by regular expressions which are understood by the lexical analyzer. It also removes lexical errors, comments, and white space, then sending the tokenized data to the parser. We’ve planned to tokenize using the lexer available as a part of SLY.

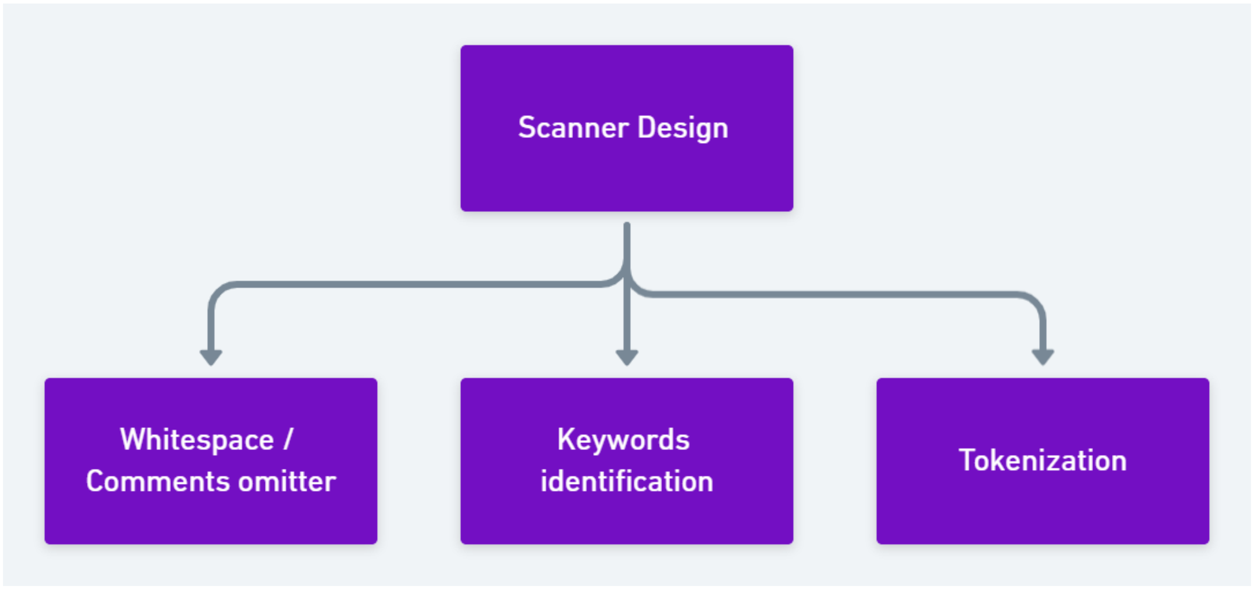
**Syntax analyser / Parser:** A parser takes input in the form of a sequence of tokens from the scanner, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming. A parser also checks all data provided to ensure it is sufficient to build a data structure in the form of a parse tree or an abstract syntax tree.

**Semantic analyser :** Semantic Analysis makes sure that declarations and statements of a program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both the syntax tree of the previous phase and symbol table are used to check the consistency of the given code. Type checking is an important part of semantic analysis where the compiler makes sure that each operator has matching operands.

**Target code generation :** This is the final stage of compilation. Each line in optimized code may map to one or more lines in machine (or) assembly code, hence there is a 1:N mapping associated with them. In our case, the target code is going to be in Python, and that will be the game engine that finally runs our tetris implementation.

## Scanner Design

Scanner breaks the character stream of the input programming file into tokens. If the scanner can not find a valid token for the current string, it will throw an error with the line number

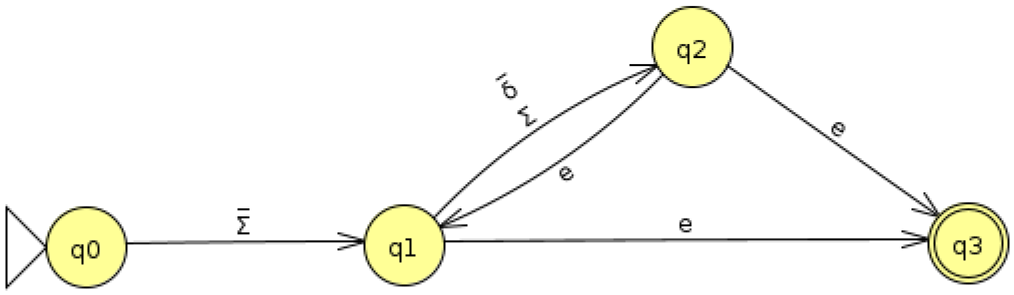


### Transition diagrams for some tokens:

We show the transition diagrams using Non Deterministic Finite Automaton (NFAs).

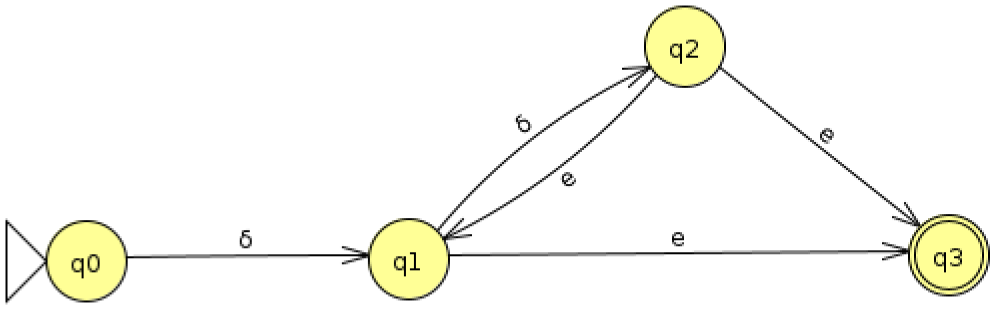
**Σ = [a-zA-Z]**
**δ = {0-9}**
**e = ε (epsilon)**

### Identifiers -



This is the NFA for the identifiers as we have defined for our language. Identifiers have a regular expression that takes in alphanumeric characters.

**Numbers -**

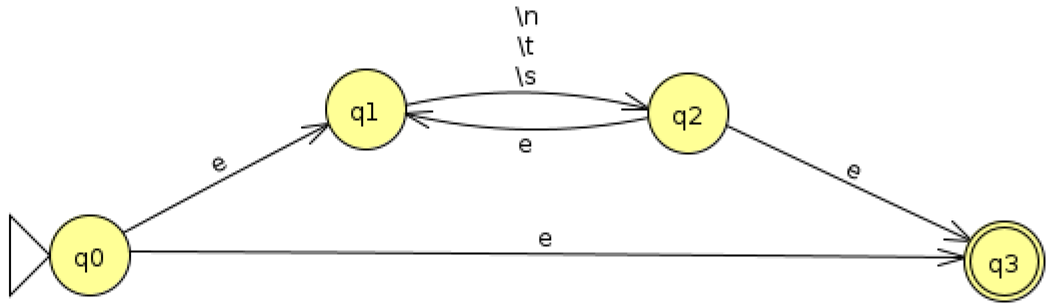


In a similar fashion we devise the NFAs for numbers.

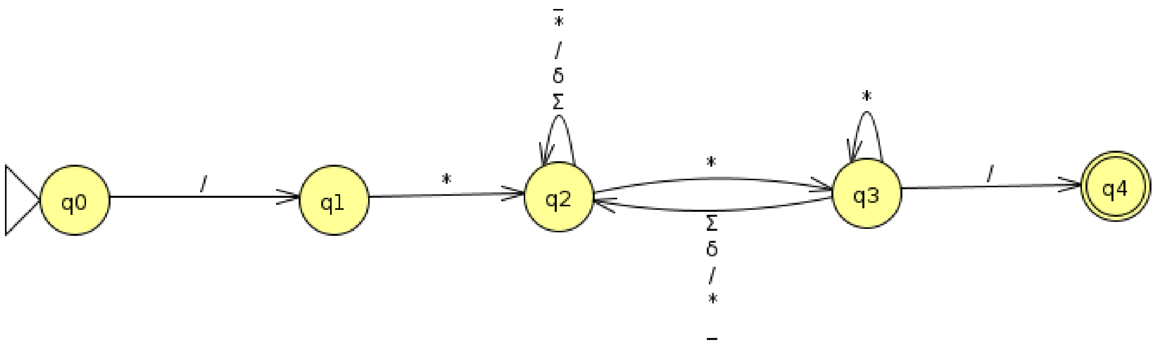
**e - Epsilon**

And **δ = {0-9}**

## Whitespace -



**Comments -**



## Pattern-Action pairs:

We now define the pattern action pairs, where we map the tokens to actions associated with them:

| Pattern | Action |
|---|---|
| //Types | |
| ‘int’ | return INT_TYPE |
| ‘str’ | return STR_TYPE |
| ‘tetro’ | return TETRO |
| ‘array’ | return ARRAY |
| ‘board’ | return BOARD |

| | |
|---|---|
| | |
| //Loops and conditionals | |
| ‘while’ | return WHILE |
| ‘if’ | return IF |
| ‘else’ | return ELSE |
| ‘timeout’ | return TIMEOUT |
| | |
| //Hardness and types of the game as tokens | |
| ‘EASY’ | return EASY |
| ‘MEDIUM’ | return MEDIUM |
| ‘HARD’ | return HARD |
| ‘NORMAL’ | return NORMAL |
| ‘SPRINT’ | return SPRINT |
| | |
| //Keybindings and rotations | |
| ‘ARROW_LEFT’ | return ARROW_LEFT |
| ‘ARROW_RIGHT’ | return ARROW_RIGHT |
| ‘CLOCKWISE’ | return CLOCKWISE |
| ‘ANTI_CLOCKWISE’ | return ANTI_CLOCKWISE |
| ‘HARD_DROP’ | return HARD_DROP_FLAG |
| ‘SOFT_DROP’ | return SOFT_DROP_FLAG |
| | |
| //Misc (literals, parentheses..) | |
| regex = "[0-9][0-9]*' | return NUMBER |

| | |
|---|---|
| regex = '\"(\\.\|[^"\\])*\"' | return STRING_LITERAL |
| regex = ‘[,]’ | return SEPARATOR |
| regex = ‘\[‘ | return LBLOCKPAREN |
| regex = ‘\]’ | return RBLOCKPAREN |
| regex = '\{' | return LCURLYPAREN |
| regex = ‘\}’ | return RCURLYPAREN |
| regex = ‘;’ | return EOL |
| ‘=’ | return ASSIGN |

In addition to this, we have tokens for built-in functions that we have defined earlier as well:

‘getBoard' = GET_BOARD
'getName' = GET_NAME
'getNextTetromino' = GET_NEXT_TETROMINO
‘setSpeed' = SET_SPEED
'setGameMode' = SET_MODE
'rotateRight' = ROTATE_RIGHT
'rotateLeft' = ROTATE_LEFT
'moveRight' = MOVE_RIGHT
'moveLeft' = MOVE_LEFT
‘hardDrop' = HARD_DROP
'checkClearedLine' = CHECKED_CLEARED_LINE
'clearLine' = CLEAR_LINE
‘getCharacter' = GET_CHAR
'display' = DISPLAY
'displayUpcomingTetromino' = DISPLAY_NEXT_TETRO
'displayTetromino' = DISPLAY_TETRO
'displayBoard' = DISPLAY_BOARD
'addScore' = ADD_SCORE
'clearScreen' = CLEAR_SCREEN
‘break’ = BREAK

‘push’  = PUSH

‘rem’  = REM
‘pop’ = POP
‘len’ = LEN
‘exit’ = EXIT
‘and’ = AND
‘advance’ = ADVANCE
‘playHW’ = PLAY_HW
‘setGameDifficulty’ = SET_GAME_DIFFCULTY

‘not’ = NOT
‘Or’ = OR
‘import’ = IMPORT
‘game’ = GAME

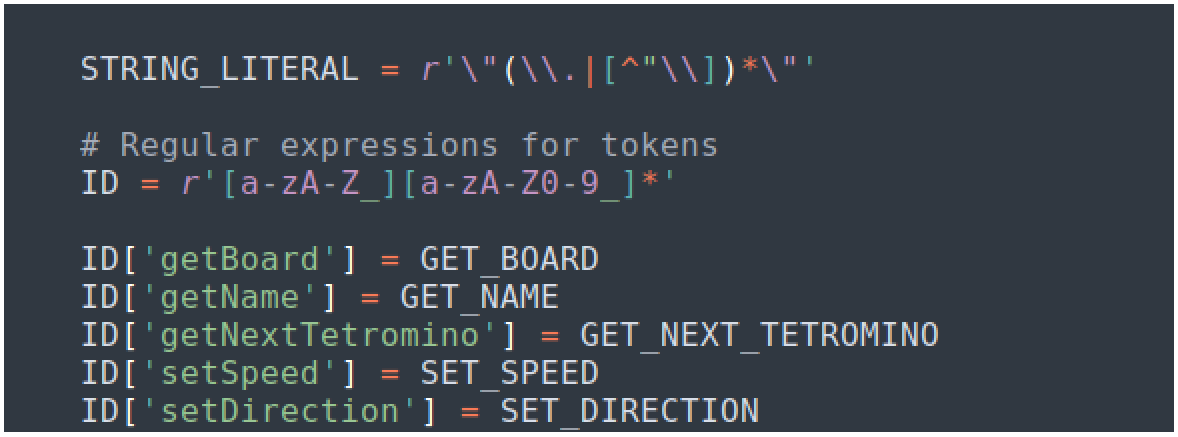
**Relational Operators**
‘GT’ = r’>’
‘LT’ = r’<’
‘GTE’ = r'>='
‘LTE’ = r'<='
EE = r'=='
NE = r'!='

**Literals mapped to themselves**
literals = {'(', ')', '+', '-', '/', '*', '%'}

**Total number of:**
- Distinct Patterns = 77
- Dictinct Token Types = 76
- Tokens types that are lexemes themselves = 7

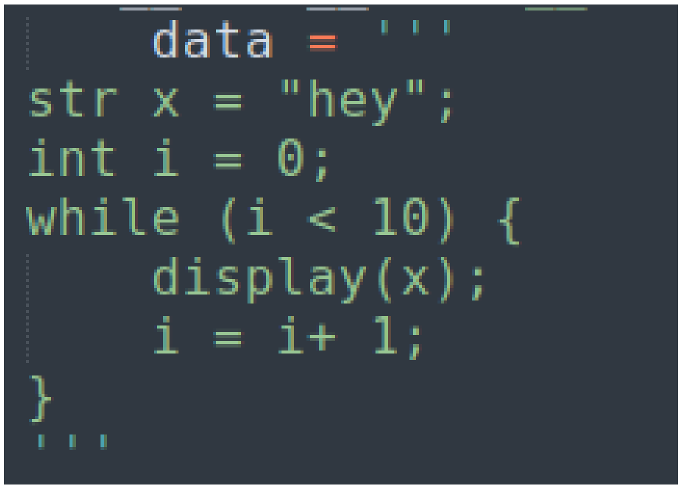
---

We have identified keywords in our language by first ‘tokenising it as an identifier then remapping it as the particular keyword. A snapshot of our code for implementing this in SLY with Python is shown below:

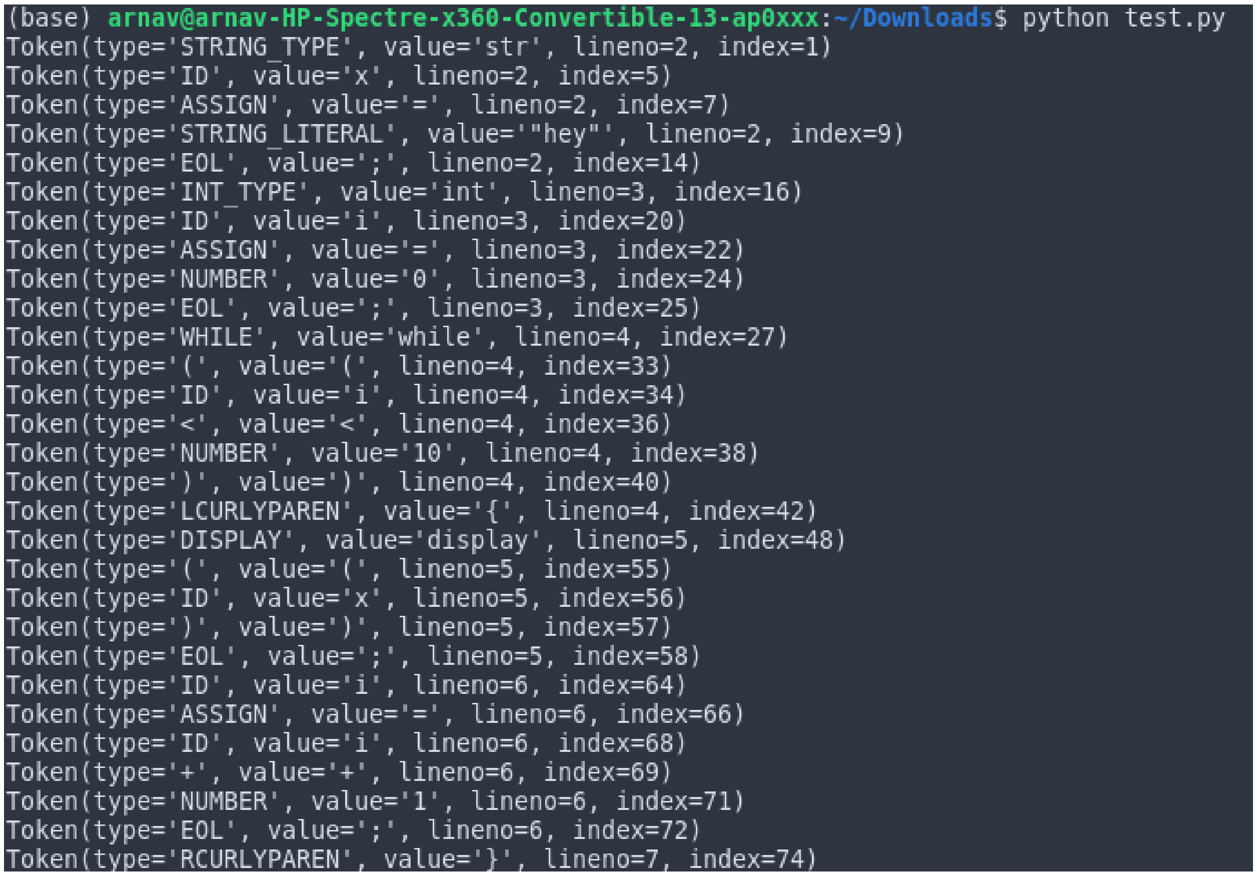


```
STRING_LITERAL = r'\"(\\.|[^"\\])*\"'

# Regular expressions for tokens
ID = r'[a-zA-Z_][a-zA-Z0-9_]*'

ID['getBoard'] = GET_BOARD
ID['getName'] = GET_NAME
ID['getNextTetromino'] = GET_NEXT_TETROMINO
ID['setSpeed'] = SET_SPEED
ID['setDirection'] = SET_DIRECTION
```

**Here is a screenshot of tokenization of a sample code in our through our lexer made with SLY:**



```
    data = '''
str x = "hey";
int i = 0;
while (i < 10) {
    display(x);
    i = i+ 1;
}
'''
```



```
(base) arnav@arnav-HP-Spectre-x360-Convertible-13-ap0xxx:~/Downloads$ python test.py
Token(type='STRING_TYPE', value='str', lineno=2, index=1)
Token(type='ID', value='x', lineno=2, index=5)
Token(type='ASSIGN', value='=', lineno=2, index=7)
Token(type='STRING_LITERAL', value='"hey"', lineno=2, index=9)
Token(type='EOL', value=';', lineno=2, index=14)
Token(type='INT_TYPE', value='int', lineno=3, index=16)
Token(type='ID', value='i', lineno=3, index=20)
Token(type='ASSIGN', value='=', lineno=3, index=22)
Token(type='NUMBER', value='0', lineno=3, index=24)
Token(type='EOL', value=';', lineno=3, index=25)
Token(type='WHILE', value='while', lineno=4, index=27)
Token(type='(', value='(', lineno=4, index=33)
Token(type='ID', value='i', lineno=4, index=34)
Token(type='<', value='<', lineno=4, index=36)
Token(type='NUMBER', value='10', lineno=4, index=38)
Token(type=')', value=')', lineno=4, index=40)
Token(type='LCURLYPAREN', value='{', lineno=4, index=42)
Token(type='DISPLAY', value='display', lineno=5, index=48)
Token(type='(', value='(', lineno=5, index=55)
Token(type='ID', value='x', lineno=5, index=56)
Token(type=')', value=')', lineno=5, index=57)
Token(type='EOL', value=';', lineno=5, index=58)
Token(type='ID', value='i', lineno=6, index=64)
Token(type='ASSIGN', value='=', lineno=6, index=66)
Token(type='ID', value='i', lineno=6, index=68)
Token(type='+', value='+', lineno=6, index=69)
Token(type='NUMBER', value='1', lineno=6, index=71)
Token(type='EOL', value=';', lineno=6, index=72)
Token(type='RCURLYPAREN', value='}', lineno=7, index=74)
```

## Division of Labour between Scanner and Parser

- **Error Handling:**
    - Scanner will identify the lexical errors (eg: if a literal has a start and not an end).

    - It can find the syntactical errors while deriving the meaning of our language.

- **Difference in tokenizing:**
    - There are cases where tokenization can be done in different ways.
    - This in turn changes the amount of work the parser has to do.
    - Consider for example, an initialisation of an array:

```
array arrName[size];
```

In this case, we can specify tokenization in two different ways:

```
{array, arrName[], size, ;} or {array, arrName,
[,size,], ;}
```

In such cases we decided to tokenize the square brackets as well in the beginning so that the parsing scheme becomes easier.

## Distribution of Roles among the Team members

Since compiler design and construction is a new concept for every group member and the contribution of each member in the project is crucial given the size and complexity of the problem statement, we planned on working together and, as the situation demands, division of labour can be implemented when the next plan of work is provided.