Compiler Construction (CS F363) Project - Stage 1

Top Level Design Specifications and Scanner 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

7th March, 2022

Top Level Design Specs

1. Overall Program Structure

The syntax and structure of our Tetris Game-Language is based on (similar to) the syntax of **C-type programs**, i.e, (C, C++, Java). Reasons for this choice:

1. We wanted to 'encapsulate' the code for a particular level (see the section on 'Levels' below) inside a "Block" like structure, that makes it more easily readable. So the code for a level would be enclosed inside curly braces: {}.
2. We wanted to enforce the ending of statements with semicolons, as in C, C++, to make it easier on the parser side.
3. We found it easier to altogether ignore whitespaces including newlines '\n', tabs, '\t' and carriage returns, '\r'. If we were to implement Python-style syntax, we would have to take care of the correct indentation (i.e, uniform indentation across the program), which is slightly more complicated to implement.
4. Our comments are also C-style, enclosed within `/* ... */`, however, for simplicity, we do not provide the `//` format for comments

However, the compiler design structure and the actual "Backend" game code will be in the Python programming language. For our compiler, we will be using "lex" lexical analyzer and "yacc" parser tools for Python from the PLY library, which is basically the popular Lex-Yacc C code implemented in Python. It is functionally no different from the C-version, however the syntax is in Python, and is slightly different from the C-lex and yacc syntax.

---

2. Offered Tetris Primitives

The language enables the programmer to customize primary game features as suited, offering a wide variety of additional features as well, all of which are listed below:

Game Levels-

- The programmer is given the choice to implement multiple levels for the game. It is mandatory to include 1 level. These levels can be named as per the programmer's choice. These level names, for example "Level 10: The dragon appears", would be shown to the end-user, i.e, the game player.

- This gives the programmer the freedom to experiment with different board-sizes, different pieces, different speeds, scoring, etc across the different levels.
- For example, a programmer might want to increase (or decrease, depending on their programming style) the board size, as the levels progress. A programmer can explicitly decide to create more complicated tetrominoes for harder levels, and also increase the falling speed of the same.
- This would also come with better ‘points scored’ as the level increases. But this is, again, the programmers’ choice, as to how the points increase as the levels progress.
- The level ends when the player has scored a certain number of points, this is again programmed by the programmer, into a keyword called `passlevelscore`.

Example code:

```
level l1 {
    passlevelscore(150);
    /* Entire Code for Level*/
}
level l2 {..} /* And so on */
```

A variable sized board / playing grid -

- A grid will be used to display the falling tetris blocks and will be the main interaction platform for the user.
- The size of the board can be specified by the game programmer and may be (and we recommend it to be) different for different levels, to provide the end user with a unique experience. What we provide to the programmer is the ability to do the same. An example is given below:

Example code:

```
board = 20, 10; /* Define a board with 20 rows, 10 columns*
```

Tetrominoes -

- The tetris blocks, aka: Tetrominoes are represented by a **matrix**. In the matrix, ‘0’ means empty and ‘1’ means that the cell is filled with a 1x1 cell block. This way many different tetrominoes can be easily defined.
- The programmer can define different shapes for these pieces by defining certain elements in the matrix to be ‘1’, and empty cells to be ‘0’. The matrix and piece size is the programmer’s choice, however, our parser would throw an error if the piece is too large for the board.
- However, it would make no sense in the Tetris world if the blocks were not horizontally or vertically connected. This check will be done during parsing, and we would indicate to the programmer about the “illegal” block.

Color of Tetrominoes -

- Each block can be colored differently by the programmer. The color names are standard colors like “red”, “blue”, etc, and our parser would provide an error if a particular color cannot be found in our game, if for example, the user decides to use a rare color like “vermilion”.
- We also provide the programmer the option to use **Hex-Color codes** to specify the color. We created separate tokens for the same.

Example code:

```
color(piece2, 'red');
color(piece5, #0000FF); /* Using the hex color feature */
```

Speed of Tetrominoes -

- The speed of a block indicates how fast it moves down the board while in a falling state, i.e. number of blocks moved in a particular time.
- Each block can be programmed to fall with a different speed. The default speed of the falling block is the “Level-Speed”, as defined by the user. If the user has not defined this, the default would be ‘1’, i.e, one cell downwards every frame.

Example code:

```
speed(piece2, 2); /* This block will fall 2 cells per frame */
speed(piece5, 4); /* This block will fall 4 cells per frame */
```

Sequences of Tetrominoes -

- We provide the programmer the freedom to define the sequence or the order of falling blocks. If the programmer is creative enough, they would use this feature to, for instance, in easier levels follow blocks that fit together, but in harder levels, follow blocks that are difficult to fit together. It is completely the programmer's choice.
- In our language, sequences are defined as a list of pieces in the order they would fall in. The sequence defined by the user may be arbitrarily long. But for ease of the programmer, we also provide a repeat factor multiplier, '*', using which the programmer can repeat blocks without explicitly having to write it entirely.

Programmable randomization:

- If the programmer is not really concerned about the order of falling pieces in a sequence, but is rather concerned with 'waves' of pieces, they may simply define a sequence of pieces and randomize the sequence.
- For example, even within a particular level, the user may start off with a 'wave' of easier pieces, followed by a 'wave' of harder pieces. In the game-sequence (described in the next point), the user may also repeat this particular sequence, using the same repeat factor multiplier, '*'.
- We also provide the programmer with the option of specifying a seed value to obtain fixed results for a game session, for the purposes of testing.

Movement options -

- We decided not to play around too much with movement options, as the most standard configurations are using Arrow keys or using the keys WASD for movements. We provide this option as a moveconfig with 2 options: '**ARROW**' or '**WASD**'. We believe that adding other 'key' options makes the programmer's (and the gamer's) playing more complicated.
- The space key when pressed continuously moves the tetrominoes down much faster than their default speeds. We don't allow the programmer to change this key specification, as it makes the game-playing more complicated.
- Pressing the 'Z' key directly moves the block down to the lowest position possible.

Example code:

```
moveconfig(ARROW); /* Now, movements will happen using Arrow Keys */
```

Our plan with the parser and the game engine at the backend is that all these 'key assignments' will be visible to the Player before beginning the game.
In order for the player not to get confused between different versions of the game, we decided to stick with the above options, i.e, only 2 movement configs, and only 2 other special moves, which have fixed keyboard assignments.

Scoring -

- The programmer defines the scores as an array ([10, 20, 30], for example), where array[i] corresponds to the score received for clearing i consecutive rows of blocks at once. If the player clears more rows than those for which a reward is specified, they receive the same points as the last mentioned score in the array
- The scoring rule above gets multiplied if a 'bonus piece' clears any rows (mentioned below)

Example code:

```
scoring = [100, 200, 300, 500];
/* 100 points for clearing 1 row, 200 for 2 rows consecutively, and
so on. */
```

Additional Features added -

- We have introduced a '**Bonus**' option, which applies to blocks. If a block is made a 'Bonus' block, (this is defined by a block, and an Integer, the score multiplier), and it Clears a row, then we multiply the score received with the score multiplier defined for the block.
  `bonus(piece2, +20);`
- The programmer can allow the Player to '**skip**' certain pieces, at the cost of losing some points. We realized that it can be beneficial to skip pieces at times, if the skip-cost is moderate, in order not to 'mess up' the board with the piece. Pressing 'X' destroys the block (if skip-option is provided), and decrements the score of the user. Of course, if the player wants to score higher, they must try to reduce the number of blocks skipped.
  `skipblock(toughPiece12, -15);`
- **Simultaneous Blocks:** The programmer has been provided the option to create a 'piece' that essentially defines that 2 (previously defined) pieces would fall together (simultaneously).

An example is provided below: (The variable declaration details are described in the next section)

Example code:

```
piece simul_t_and_u = simultaneous(t_shaped_block, u_shaped_block);
sequence t_shape_sequence = [block1, block2, simul_t_and_u * 4];
```

---

3. Modes for Programmable Features -

As a part of our design, we have decided to abstract low-level implementation features away from the programmer, giving the programmer the full functionality and freedom to play around with the actual "Game" development
In other words, to make work easier for the programmer, all they need to do is to code up their Tetris game using the features provided, without having to worry about the low-level implementation.

For this we provide certain usual programmable features as seen in most programming languages:

Variables -

- The variables that the programmer needs to define may be the Blocks, and the Sequences.
- As in General purpose programming, we define variables for our ease, to use them more efficiently later on in our code. The programmer may have to use their defined blocks as parts of their sequence, and may have to use their defined sequences to specify the game-sequence order in the level.
- The blocks are defined very similar to how we have variable declarations in C/C++, they may be named appropriately by the user. As mentioned earlier, the blocks are defined by Matrices, with '0' meaning an empty cell, '1' meaning a filled cell in the given matrix.

Example code:

```
block t_shape3x3 = [[0, 1, 0],
                    [0, 1, 0],
                    [1, 1, 1]];
sequence t_shape_sequence = [t_shape3x3 * 20];  /* Using * as repeat factor */
```

Looping -

- The programmer need not worry about constructing any loops in a program of our language.
- The game loop is specified, as mentioned earlier, using ‘sequences’, which in turn are defined as a list of piece definitions.
- By using our startgame token, and providing a list of sequences (defined earlier), and also their repeat factors, i.e, how many times each sequence may be repeated, the programmer implicitly defines a game loop and the order in which the tetrominoes fall.
- Of course, as mentioned earlier, the programmer can also randomize the sequences.

Example code:

```
/* p1, p2, ... p10 are arbitrary pieces defined earlier */
sequence firstWaveEasy = [p2, p4, p1 * 2, p3];
sequence secondWave = [p6, p2, p7, p3, p4];
sequence thirdWaveHard = [p10 * 10, p9 * 5, p8 * 3];
random(secondWave);  /* Randomizes the secondWave sequence*/
random(thirdWaveHard);
startgame(firstWaveEasy * 10, secondWave * 3, thirdWaveHard * 7);
```

After the sequences have finished, (in our above example, this would happen fast. But the programmer may be creative and define a larger, more complex sequence of pieces), we will continue the game loop with pieces from the **last sequence** mentioned in startgame, Unless the user randomizes startgame also, by providing the random token without any parameters inside the parentheses.

---

4. Pipeline Schema

The steps followed in the pipeline for our compiler design are listed down below:

1. The programmer writes a program in our Tetris language, and names it, say, “game.ttr”.
2. In our present submission, the programmer opens a terminal and calls:

   ```
   python3 tetris_lexer.py
   ```

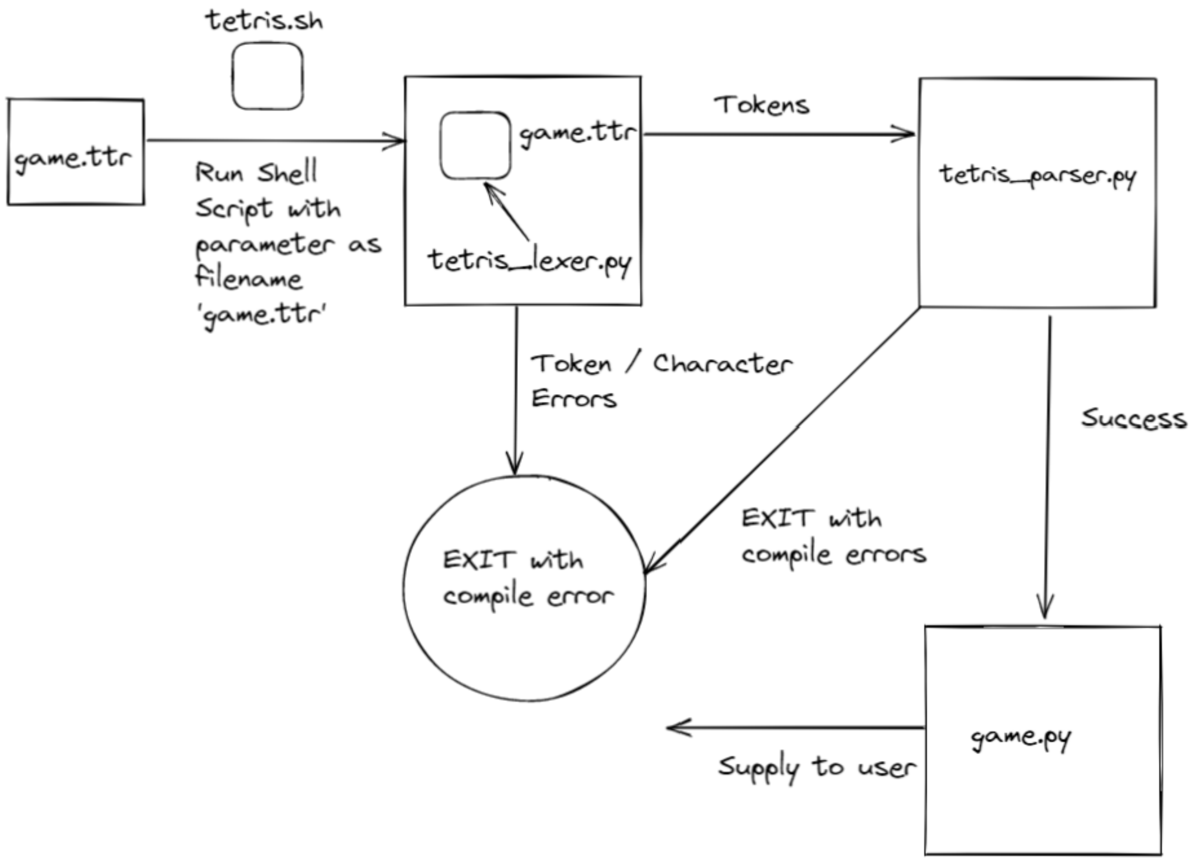
   For now, the filename should be passed via Python input in the terminal (the user will be requested to input the name of the file

For our later submissions (as shown in the following pipeline figure), this would be automated in a script like so:

```
tetris game.ttr
```

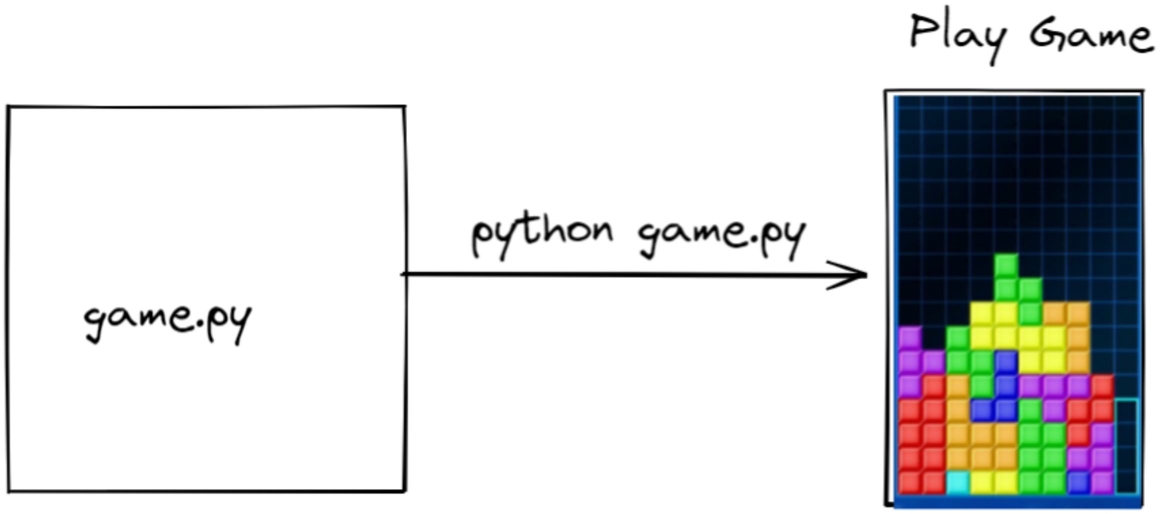
Where, “tetris” would be the name of our executable script, which essentially just runs our Python-Lexer.

3. Our lexer (tetris_lexer.py) lexically analyses “game.ttr”, and exits if there are any token/character errors, i.e, if it found a sequence that could not be matched with any defined tokens.
4. If no errors were found, it passes the token stream to our parser, tetris_parser.py.



5. Tetris_parser.py uses the token stream to create a **runnable python script**, “game.py”, (or appropriately named by the programmer), which can be passed to end-users to play directly, using “python game.py”.

6. The above command would create a GUI, and the user would be able to play the Tetris game as programmed by the programmer, (in the backend, we plan to use the PyGame Python Library).



Scanner Design

We have designed our scanner using LEX, but the Python version of Lex, using a library called PLY. As mentioned in the introduction, this is only syntactically different from the C version, not functionally different. The goal is to produce a Token-Stream from the given input file.

Our game program file's extension is .ttr. So one would have to run "python tetris_lexer.py", where tetris_lexer.py is our scanner, written as a Python Script. For the purpose of this submission, the filename must be provided as an input to the script (the program halts and accepts this as input), but for later submissions, we shall make this slightly more automated using a shell script.

1. Pattern Action pairs:

The following description lists the tokens defined in our language, along with their return value, i.e, what is the exact value of the token sent to the parser. In most cases, this value is simply the token name itself:

Regular Expression (Pattern)	**Action**	**Meaning/Use**
`[a-zA-Z_][a-zA-Z_0-9]*` *Note*: This is the same regex as 'identifier', however, we first check whether the lexeme matched exists in our list of reserved tokens. If yes, we return the Reserved Word Token. If no, we claim it is an Identifier	If we find the lexeme in our reserved dictionary, return the token corresponding to that reserved word. Ex: 'board' returns a token, "BOARD" Otherwise, return "IDENTIFIER"	Reserved Keywords like 'board', 'piece', etc
`[a-zA-Z_][a-zA-Z_0-9]*`	identifies "IDENTIFIER", returns its name	Variables for pieces, sequences
`\/\*.*\*\/` (*Note*: '\' escape char used)	Identifies it to be a 'comment', Ignores it.	Comments will not be sent to the parser
`[ \t\r\n]+`	Ignored	Whitespace, including

		next lines, carriage returns, tabs
`[+-]?[0-9]+`	Identifies “INT”, returns the integer corresponding to it.	Integer values, including lexemes like “-50”, “+20”, “2”, etc
`\'[a-z]+\'`	identifies “COLORNAME”, returns the string corresponding to the color	Piece Color-name is the only place in our language where a string is written within single quotes, so this regex avoids any clashes.
`\#[A-Fa-f0-9]{6}|[A-Fa-f0-9]{3}`	Identifies “HEXCOLOR”, returns the hex color mentioned by the programmer	The programmer also has the option to specify a Hex-color value code for piece colors

The following regex’s are **Tentative**, (see the section on Division of Labor), and we may remove them altogether in a later submission (replacing them with individual tokens for each punctuation/integer in the matrix/array)

Regular Expression (Pattern)	**Action**	**Meaning/Use**
`\[\[[^\;]*\]\]`	Identifies Matrix expression, Returns a Parsed Matrix.	A simple Matrix Regex, used for Piece declarations
`\[[^\;\[]*\]`	Identifies Array, Returns the Parsed array	A simple List/Array regex, used for sequence declarations

The keywords are mentioned below, they are put into a dictionary storing “pattern”:”TOKEN_NAME”, for example, for the keyword “piece”, we would add the following dictionary entry: (“piece”, “PIECE”);

The patterns are mentioned below, along with their meanings in our language. The return token is, as mentioned above, simply the fully capitalized version of the pattern.

`level` (used to define a level)	`setlevelspeed` (used to set the default speed for a level)	`board` (used to define the board)
`piece` (used to define a piece)	`speed` (used to set the speed of a piece to some value)	`piececolor` (used to set the color of some piece to a color-name or hex-value)
`skipblock` (used to make a block skippable, please see our additional features section)	`bonus` (used to set a 'bonus' option for a piece)	`sequence` (to define a sequence)
`random` (to add randomness to a sequence or the entire game)	`startgame` (to setup the game with the sequence of sequences provided)	`scoring` (to define the row-clearing scoring rules for the level)
`passlevelscore` (set the score required to pass the current level)	`moveconfig` (set the movement config, i.e, wasd or Arrow Keys). Additionally, we define tokens: "wasd", "ARROW", which will be fed to the parser to identify the movement scheme	`simultaneous` (used when the programmer wishes to drop 2 blocks simultaneously)

2. Punctuations:

We define the usual punctuations as Tokens, with their token names CAPITALIZED. The punctuations we use are:
Equal to (`=`), Parentheses `()`, Curly braces `{}`, Semicolons `;`, commas(`,`), repeat factor (`*`).

3. Regular Definitions

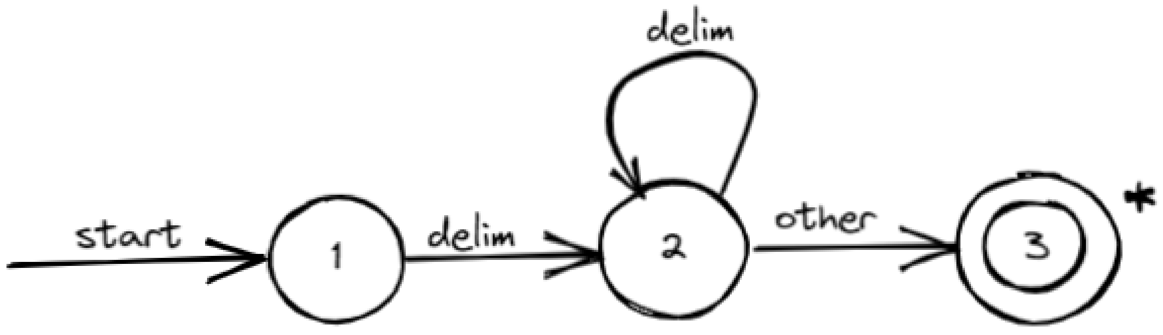
delim - [\t\r\n]
letter - [a-zA-Z]
digit - [0-9]
hex_letter - [A-Fa-f0-9]
comment_letter - character set ~ {*}
comment_letter2 - character set ~ {*, /}

---

4. Transition Diagrams

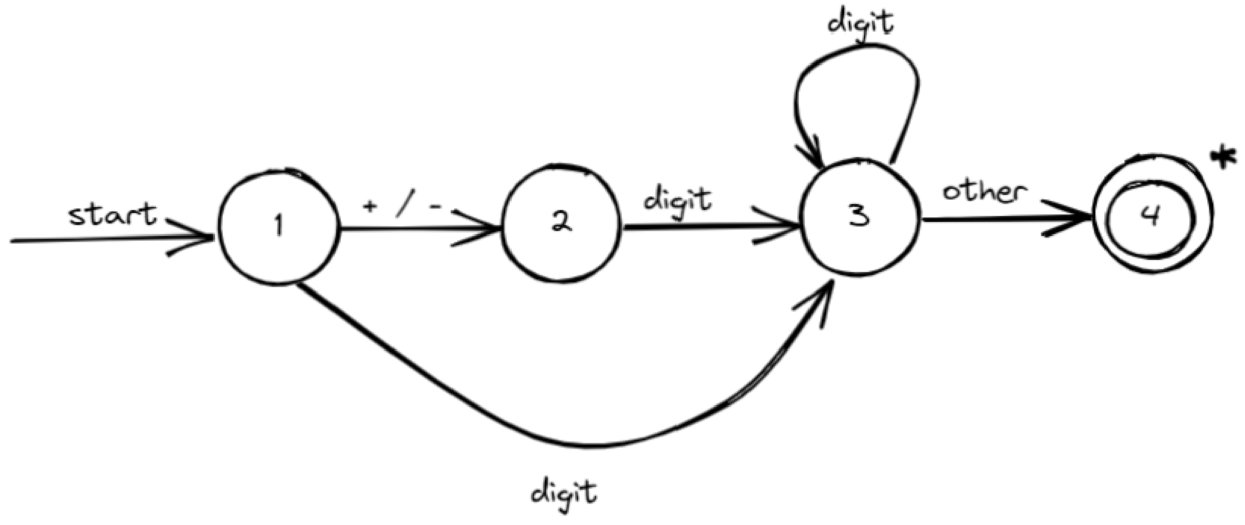
Transition diagrams for the tokens that will be generated through lexical analysis are listed below.

WHITESPACE Token:



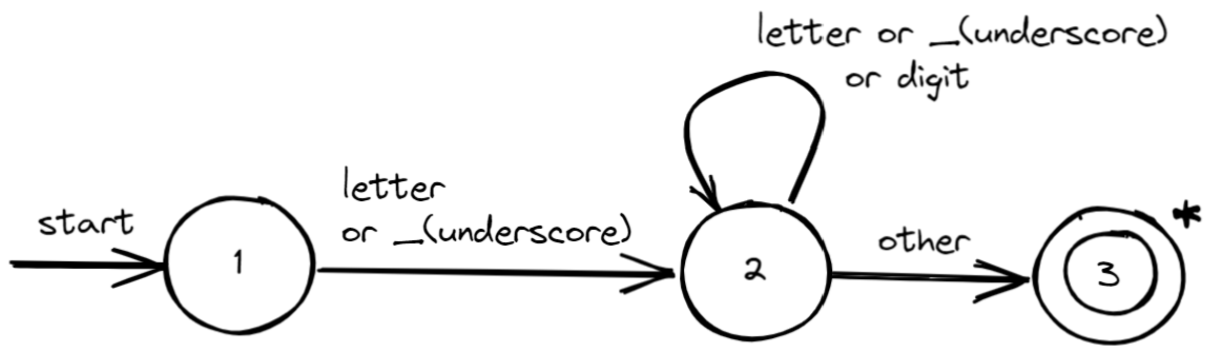
To detect whitespaces, we look for one or more ‘whitespace’ characters, which are represented by the regular definition **‘delim**’. Typically, these characters are blanks (‘ ‘), tab(‘\t’), carriage return(‘\r’) or newline(‘\n’). The transition diagram detects a block of consecutive ‘whitespace’ characters followed by a non-whitespace character.

INT Token:



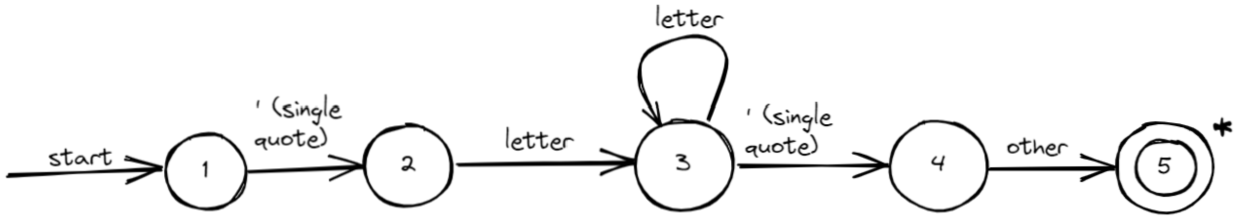
An ‘INT’ token is defined by a series of digits, preceded by a ‘+’ or ‘-’ symbol (optional). In case of the absence of ‘+’ or ‘-’, we assume the number to be positive. This combination followed by a non-INT token forms our transition diagram.
Examples - +50, -20, 80.

IDENTIFIER Token:



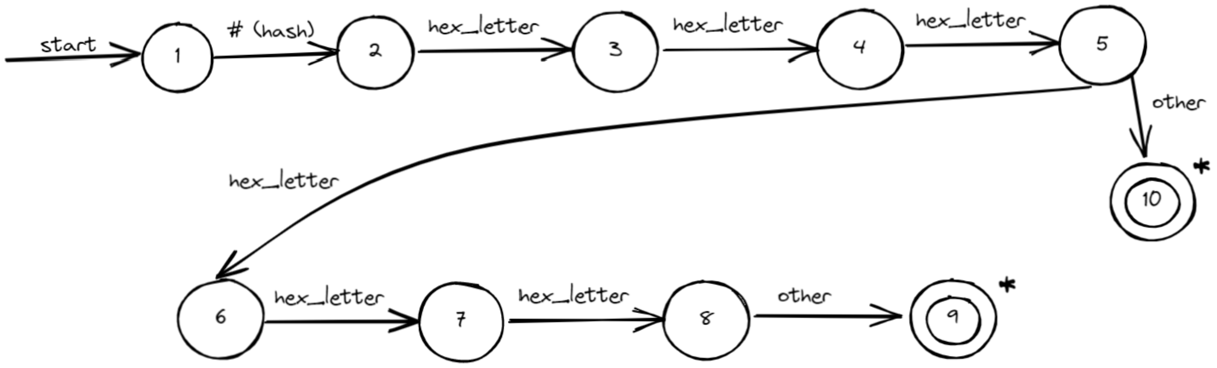
An ‘IDENTIFIER’ token is defined by a sequence starting with a letter or underscore(_). This is followed by letters or underscores(_) or digits, as an identifier doesn’t start with a digit in our language.
Examples - _id, num1, abc_123

COLORNAME Token:



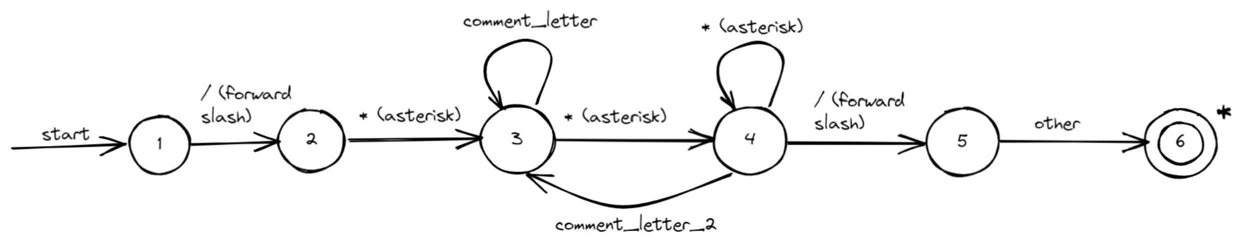
The ‘COLORNAME’ token consists of a sequence of letters defining the color of our piece enclosed in single quotes. We start by picking a single quote, followed by one or more letters, and then a single quote again. We retract our input on detecting any other character.
Examples - ‘red’, ‘yellow’

HEXCOLOR Token:



The ‘HEXCOLOR’ token represents hex color codes for the pieces. This starts off with a hash(#) symbol, followed by a 3 or 6 letter sequence of a hex_letter, which we have defined in our regular definitions.

COMMENTS Token (Identified, but Ignored):

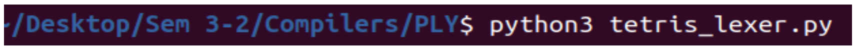


The **'COMMENT'** token is similar to comments used in certain high level languages we use everyday (like C, C++, etc.). Starting with the sequence "`/*`" and ending with "`*/`", it has a sequence of all possible letters in our character set in between.

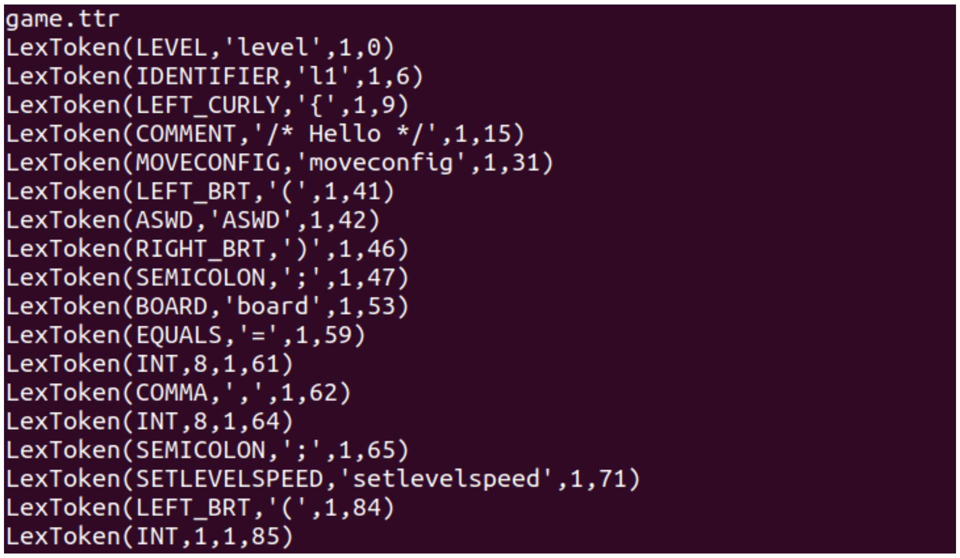
5. Test Cases:

Below, we test and see how the output of our lexer looks like:

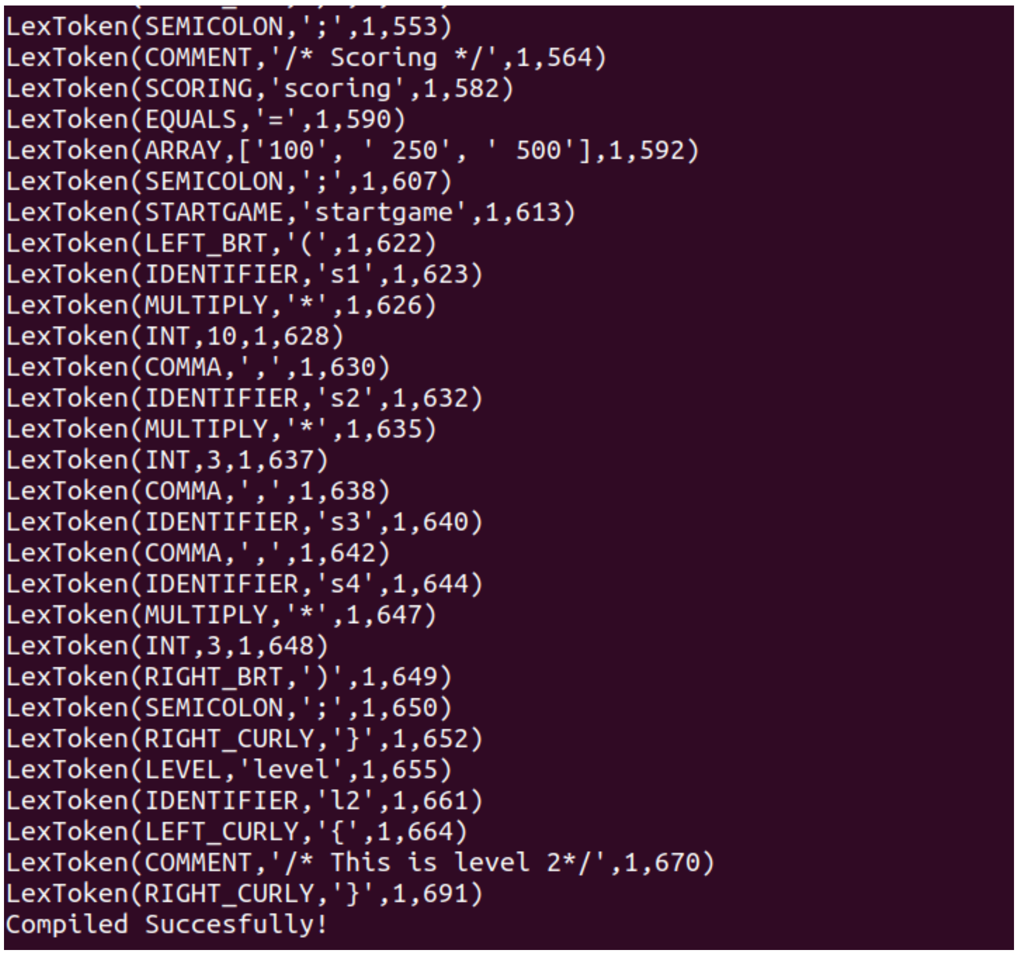
i) Successful Compilation:



```
~/Desktop/Sem 3-2/Compilers/PLY$ python3 tetris_lexer.py
```

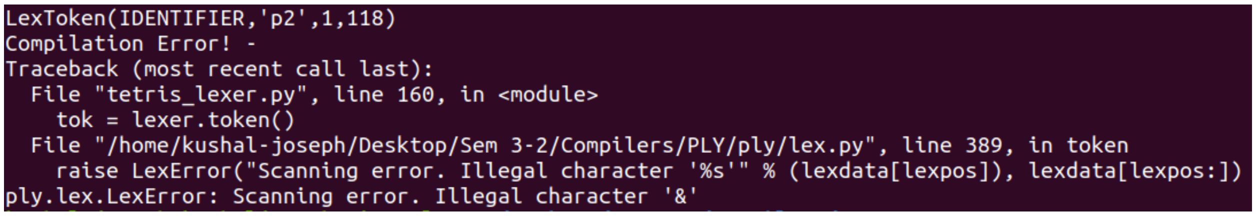


```
game.ttr
LexToken(LEVEL,'level',1,0)
LexToken(IDENTIFIER,'l1',1,6)
LexToken(LEFT_CURLY,'{',1,9)
LexToken(COMMENT,'/* Hello */',1,15)
LexToken(MOVECONFIG,'moveconfig',1,31)
LexToken(LEFT_BRT,'(',1,41)
LexToken(ASWD,'ASWD',1,42)
LexToken(RIGHT_BRT,')',1,46)
LexToken(SEMICOLON,';',1,47)
LexToken(BOARD,'board',1,53)
LexToken(EQUALS,'=',1,59)
LexToken(INT,8,1,61)
LexToken(COMMA,',',1,62)
LexToken(INT,8,1,64)
LexToken(SEMICOLON,';',1,65)
LexToken(SETLEVELSPEED,'setlevelspeed',1,71)
LexToken(LEFT_BRT,'(',1,84)
LexToken(INT,1,1,85)
```



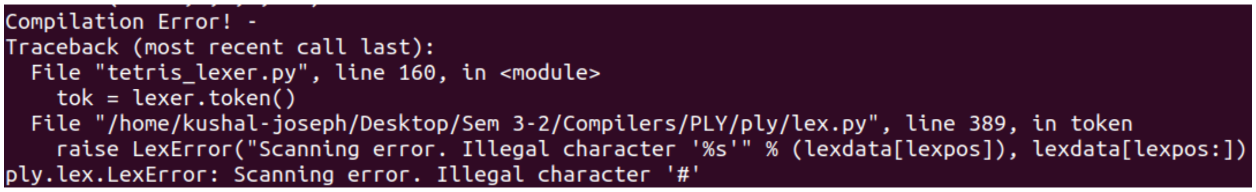
```
LexToken(SEMICOLON,';',1,553)
LexToken(COMMENT,'/* Scoring */',1,564)
LexToken(SCORING,'scoring',1,582)
LexToken(EQUALS,'=',1,590)
LexToken(ARRAY,['100', ' 250', ' 500'],1,592)
LexToken(SEMICOLON,';',1,607)
LexToken(STARTGAME,'startgame',1,613)
LexToken(LEFT_BRT,'(',1,622)
LexToken(IDENTIFIER,'s1',1,623)
LexToken(MULTIPLY,'*',1,626)
LexToken(INT,10,1,628)
LexToken(COMMA,',',1,630)
LexToken(IDENTIFIER,'s2',1,632)
LexToken(MULTIPLY,'*',1,635)
LexToken(INT,3,1,637)
LexToken(COMMA,',',1,638)
LexToken(IDENTIFIER,'s3',1,640)
LexToken(COMMA,',',1,642)
LexToken(IDENTIFIER,'s4',1,644)
LexToken(MULTIPLY,'*',1,647)
LexToken(INT,3,1,648)
LexToken(RIGHT_BRT,')',1,649)
LexToken(SEMICOLON,';',1,650)
LexToken(RIGHT_CURLY,'}',1,652)
LexToken(LEVEL,'level',1,655)
LexToken(IDENTIFIER,'l2',1,661)
LexToken(LEFT_CURLY,'{',1,664)
LexToken(COMMENT,'/* This is level 2*/',1,670)
LexToken(RIGHT_CURLY,'}',1,691)
Compiled Succesfully!
```

ii) In the case of a Token/Character error:
For instance: Adding an unknown character ‘&’ randomly, our lexer identifies the illegal character and informs the programmer about it:



```
LexToken(IDENTIFIER,'p2',1,118)
Compilation Error! -
Traceback (most recent call last):
  File "tetris_lexer.py", line 160, in <module>
    tok = lexer.token()
  File "/home/kushal-joseph/Desktop/Sem 3-2/Compilers/PLY/ply/lex.py", line 389, in token
    raise LexError("Scanning error. Illegal character '%s'" % (lexdata[lexpos]), lexdata[lexpos:])
ply.lex.LexError: Scanning error. Illegal character '&'
```

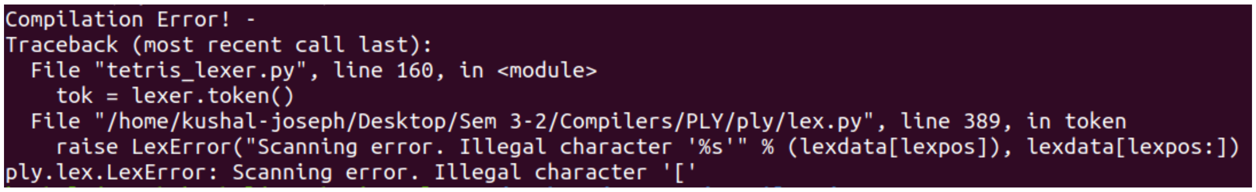
iii) Programmer doesn't specify Hex color properly:



```
Compilation Error! -
Traceback (most recent call last):
  File "tetris_lexer.py", line 160, in <module>
    tok = lexer.token()
  File "/home/kushal-joseph/Desktop/Sem 3-2/Compilers/PLY/ply/lex.py", line 389, in token
    raise LexError("Scanning error. Illegal character '%s'" % (lexdata[lexpos]), lexdata[lexpos:])
ply.lex.LexError: Scanning error. Illegal character '#'
```

Although ‘#’ is not an ‘illegal’ character, it is used to specify Hex-Colors, but here, since the hex-color provided was ‘#1Fa1’, which is not valid, it identifies the same.

iv) Programmer forgets the final ‘]’ in the matrix definition:



```
Compilation Error! -
Traceback (most recent call last):
  File "tetris_lexer.py", line 160, in <module>
    tok = lexer.token()
  File "/home/kushal-joseph/Desktop/Sem 3-2/Compilers/PLY/ply/lex.py", line 389, in token
    raise LexError("Scanning error. Illegal character '%s'" % (lexdata[lexpos]), lexdata[lexpos:])
ply.lex.LexError: Scanning error. Illegal character '['
```

In this case, the beginning of the matrix, i.e, ‘[‘ doesn’t match either with Matrix or Array token. Hence, it throws a compilation error.

Division of Labor between Scanner and Parser

Lexeme Overlap Situations:

We have defined our lexicon so as to minimize the clashes between the tokens described by patterns, and so that any clashes could be resolved in the scanner itself.
One of the situations where we did find clashes was in the token definition for reserved words. In our language, “**reserved words**”, “**color-names**” and “**identifiers**”, i.e, variables have a similar pattern. They are composed of letters, digits and underscores, with the condition that they cannot begin with a digit. (except for color-names, which cannot have numbers or underscores)

To resolve this problem:

1. We realized that keywords and identifiers would never be enclosed within quotes ‘’. However, since the colorname is passed Only using the piececolor command, it makes sense that it could be enclosed within quotes, ‘’. This is a clear distinction between color-names and the identifiers. We used this fact to specify a unique regex for the color-names, as mentioned in the Pattern-Action pairs section. Furthermore, since color-names can’t have digits, we included that conditon in its regex.
2. However, keywords and identifiers have exactly the same regex as well. For this purpose, we decided to use a Python Dictionary to store pattern action pairs for all the defined keywords. For example, an entry in the dictionary would be (“setlevelspeed”, “SETLEVELSPEED”);
3. When we identify the common regex (see Pattern-Action pairs), we first **check if the pattern is present in our keyword-dictionary.** For example, if we match a pattern “setlevelspeed”, we identify that it indeed is in our dictionary, so it must be a keyword.
4. If it is a keyword, we return its token value, as stored in the dictionary, here, “SETLEVELSPEED”.

This way the clashes between Keywords, Identifiers and color-names were handled in our Scanner itself, and the Parser directly receives the correct token

Parsing Matrices and Arrays:

We use Matrices (2D arrays) to define our Pieces, as described in an earlier section. One choice that we had to take was whether to:

1. **Only Identify the matrix in the scanner itself**, i.e, when a matrix pattern is matched, for example "[[1, 0, 1], [1, 1, 1]]", then we send this entire string as a token to the parser.
2. **Identify and Parse the matrix in the scanner itself**, i.e, convert it from a string to a List of Lists, and send this to the parser. This would make things much easier on the parser side. The parsing can be done using simple split() commands in python (refer to our tetris_lexer.py file), and the parsing done is **correct**, i.e, it always properly parses a Correct input matrix. Hence, we would return:
   [[1, 0, 1], [1, 1, 1]] as a Parsed-2D array, not a string
   Note: If we were to choose Option 1, this same procedure would be done in the parser, this being the only difference between Options 1 and 2.
3. **Do not Identify or Parse the matrix in the lexer**, instead return each punctuation/Integer as a separate token. In this case, the above pattern would actually return the following stream:
   '[',   '[',   '1',   ',',   '0',   '1',   ']',   ',',   '[', and so on.

For now, **we have kept all these options open**. Depending on the feasibility of implementing the Parser, we will pick one of the above options. In our submission, we have implemented **both Options 2, and 3**. For our final submission, we would pick whichever suits best with our Parser and end-result.

The same points apply to arrays as well. We use programmer defined arrays in our language to specify sequences of pieces, and also specify the scoring structure.

Meta-Data to be retained:

Almost all the code written in our language corresponds to structures that are immutable during execution. For example, pieces, board size, sequences, once defined, are immutable during runtime.
The code provided by the programmer in our language is used as meta-data to describe a Tetris game, as designed and developed by the programmer. The game, GUI, play-environment, animations, etc, will be generated by our parser, hence this data is retained throughout the execution of the program.

Passing Data between Lexer and Parser:
We use Lex-Yacc (the Python implementation, called PLY), hence, the exchange of tokens is done by Lex-Yacc itself. We have the option (because of the Python implementation) to either keep both lexer (Lex) and parser (Yacc) in the same file, or we can separate the files, i.e, tetris_lexer.py, tetris_parser.py. We have decided to go with the second option, i.e, separate files for Lexer and Scanner

---

Distribution of Roles among the Team:

1. **Tetris Game Specification, Features:** All members of the team
2. **Lexer Design and Code using PLY library:** Kushal, Rohan
3. **Automata/Transition Diagrams design**: Dev, Harsh, Pranay
4. **Documentation of Language, Report**: Dev, Kushal
5. **Tetris Game Research**: Shivam, Pranay, Harsh
