Report for Stage 1: Tetris Game

for

COMPILER CONSTRUCTION (CS F363)

BY

Group 36

KUNAL KARIWALA -	2019A7PS0134G
RICKY PATEL -	2019A7PS0051G
AMAN GUPTA -	2019A7PS0052G
SHAURYA PURI -	2019A7PS0035G
NIKHIL MISHRA -	2019A7PS0112G

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE,
K.K. BIRLA GOA CAMPUS

Top level design specs

Overall program structure

The syntax and structure of our game programming language resembles closely to that of C++ like languages(C/C++/Java.). We aim to implement a combination of paradigms from both C++ and Python to get the best of both worlds.

The end of a logical statement/entity will be denoted by a terminator ';' which allows for a single statement to be spread across multiple lines and makes it easier for the scanner and parser to process and distinguish between logical statements/entities.

Our language predominantly follows the procedural programming paradigm, where the program is divided into small parts called functions. For example : rotateLeft is implemented as a procedure which accepts a tetrimino and rotates it counter-clockwise once.

We also aim to incorporate elements of the object oriented paradigm in our language, to gain the advantages of modularity and reusability.

Our language allows the use of imports which allow the code implementation of macros to be written elsewhere in an auxiliary file which can be imported in the source code file as required.

We aim to keep the language in "Block" like format which allows encapsulation of logical entities between '{' punctations.

Offered Primitives & Features

- **Game Difficulty level -** The programmer will give the user the choice to select different difficulty levels for the game. These levels can be named either 'easy', 'medium', or 'difficult'. If the programmer does not specify a level, then 'easy' level will be assumed by default. This allows the programmer to vary parameters of the game like fall speed, scoring, size of the playing board/grid etc. For example, the programmer can decide to make a more rewarding scoring system as the difficulty level of the game increases. The minimum number of points required to jump to the next level can also be set by the user.

- **Basic Unit Mino** - single cell. A tetrimino is made up of four minos. Each mino will have the color of its corresponding tetrimino. The value of the mino will be equal to the value of its corresponding tetrimino.
    - Position of the basic mino: the x and y coordinates which will determine the position of the tetrimino in each state and at each instant. During its "Fall" State, it changes at every instant. Once it is in its "Rest" state, it remains the same until a row in the grid is deleted and all minoes above it are moved down

- **Tetrimino -** geometric shape formed by four Minos connected along their sides. A total of seven possible Tetriminos can be made using four Minos, each represented by a unique color and identified by a unique alphabet. 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 the elements in the matrix. The shape and color of the tetrimino are programmable by the programmer to have different configurations of tetriminos

    - O-Tetrimino: a square shape; four blocks in a 2×2 square.
    - I-Tetrimino: shaped like a capital I; four blocks in a straight line.
    - T-Tetrimino: shaped like a capital T; a row of three blocks with one added above the center.
    - L-Tetrimino: shaped like a capital L; a row of three blocks with one added above the right side.
    - J-Tetrimino: shaped like a capital J; a row of three blocks with one added above the left side.
    - S-Tetrimino: shaped like a capital S; two stacked horizontal dominoes with the top one offset to the right.
    - Z-Tetrimino: shaped like a capital Z; two stacked horizontal dominoes with the top one offset to the left.

- **Parameters/Attribute for Tetriminos**
    - Color : Tetriminoes can be assigned different colors by the programmer from a total of 7 different colors. The color names are standard, like “blue”, “red”, “orange” etc. If the programmer types in a color which does not match this category, then the parser will give an error message. Applying a color applies it to all minoes that comprise a particular tetrimino. The programmable parameter is ‘Color’ of Object Mino.
    - Physical State : The tetrimino might be blocked and in a state of “Rest” when no further movement is possible or it may be falling down and in a state of “Fall”. Change of state of a tetromino changes the state of each mino.
    - Tetrimino Rotation Orientation - For the next randomized-shape tetrimino, the original Rotation Orientation can be randomly selected from possible orientation.
    - Tetromino Rotation Controls - The Tetromino Rotation Orientation of a tetrimino, which is only in its “Fall” Physical state, can be changed to other possible orientation, by either a clockwise or right_rotate OR anticlockwise or left_rotate. We attempt to make the task of the programmer easier by providing keywords - ‘left_rotate’, ‘right_rotate’ which rotate the tetrimino once on every keyword call.
    - Position of the Tetrimino - The position of each tetrimino is decided by the X and Y Position Parameters of each Mino.
    - Speed of Tetrimino: The speed of the block is the number of grid blocks that it moves in a particular time. It is decided and programmed as a parameter of each Mino, where all Mino of Tetrimino must have the same speed.

- **Board Design**
    - The matrix/grid is the area where game play occurs. The size of the grid can be specified by the programmer using the keyword getBoard. We allow the programmer to configure the number of rows and columns visible on the board.

- **Game Statistics**
    - Scoring : The programmer defines the score as an array, eg. {5,10,20,40} where the ith index corresponds to the number of points obtained on single, double, triple and tetris line clears. This when multiplied by the level of the game contributes to the score of the player.

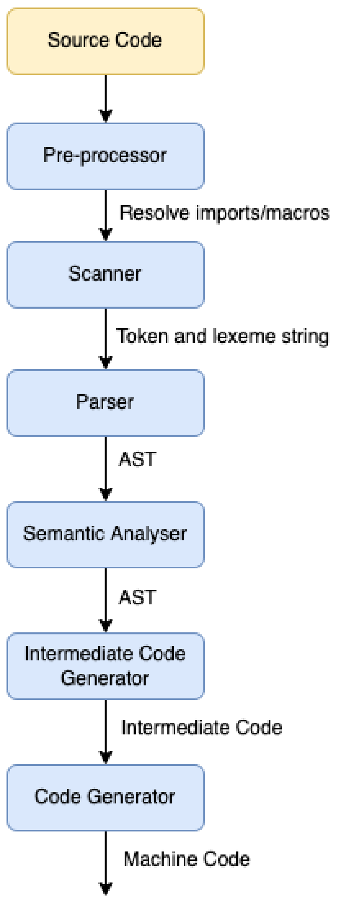
- **Movement** : We allow the programmer to choose from two different settings, either 'ASDW' configuration or 'arrow configuration'. The movement can only be left or right for a falling tetromino.

Various modes for the programmable features

Our language will allow imports, macros and will provide the programmer with the ability to implement for and while loops. It will also have the ability to break the loop and continue the loop. For the conditional cases, we aim to implement both switch and if-else for the programmer. We also allow the user to define functions and use them to implement certain features of the game.

The ability to import and use macros is because we want to be able to use pygame and macros will help with the parsing.

Pipeline schema



In the pre-processing stage, the import and macro statements fetch the packages to generate the complete, concatenated input file for the scanner.

Scanner Design

The main task of the lexical analyzer, also called lexer or Scanner is to read the input characters of the source program, group and identify them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis.

What is a token? A token is a pair consisting of a token class/name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The Scanner processes the source code file, reads the characters and tries to identify and match it with a pattern. A pattern is a description of the form that the lexemes, present in source code, of a token may take. In our language, the token classes/name explicitly exist for token including keywords like ‘if’, ‘else’, ‘while’ etc, tokens classes for operators like ‘+’, ‘*’, ‘-’, etc, token classes for identifier, constants, number, string literals, punctuation symbols and token classes for some reserved keywords that the programmer uses to call specific macros.

We use PLY(Python Lex-Yacc). PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. We generate our scanner using this PLY that is fed with our patterns and the pattern-action pair that we desire. On recognising a pattern in the source code, the scanner takes certain action based on the pattern-action pair that we have defined. In our implementation, each action ends with a return statement that returns the token class for that particular pattern, to which the lexeme matched.

Understanding PLY, Pattern-Action pair with sample example

PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. lex.py provides an external interface in the form of a token() function that returns the next valid token on the input stream. lex.py is used to tokenize an input string. For example, suppose you're writing a programming language and a user supplied the following input string:

```
x = 3 + 42 * (s - t)
```

A tokenizer splits the string into individual tokens

```
'x','=', '3', '+', '42', '*', '(', 's', '-', 't', ')'
```

Tokens are usually given names to indicate what they are. For example:

```
'ID', 'EQUALS', 'NUMBER', 'PLUS', 'NUMBER', 'TIMES',  'LPAREN', 'ID'  'MINUS', 'ID',
'RPAREN'
```

More specifically, the input is broken into pairs of token types and values. For example:

('ID','x'), ('EQUALS','='), ('NUMBER','3'), ('PLUS','+'), ('NUMBER',42), ('TIMES','*'), ('LPAREN','('), ('ID','s'), ('MINUS','-'), ('ID','t'), ('RPAREN',')')

The identification of tokens is typically done by writing a series of regular expression rules.

The template of PLY file consists of following in given order

1. Token List - All lexers must provide a list tokens that defines all of the possible token names that can be produced by the lexer. This list is always required and is used to perform a variety of validation checks.
2. Specification of tokens (Pattern and Pattern-Action Pairs) - Each token is specified by writing a regular expression rule compatible with Python's re module. Each of these rules are defined by making declarations with a special prefix t_ to indicate that it defines a token. The name following the t_ must exactly match one of the names supplied in tokens. If some kind of action needs to be performed, a token rule can be specified as a function.

Example: tokenizer for a simple expression evaluator for numbers and operators

```
import ply.lex as lex
 # List of token names.   This is always required
 tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN',
'RPAREN', )
 # Regular expression rules for simple tokens
 t_PLUS    = r'\+'
 t_MINUS   = r'-'
 t_TIMES   = r'\*'
 t_DIVIDE  = r'/'
 t_LPAREN  = r'\('
 t_RPAREN  = r'\)'
 t_ignore  = ' \t'

 # Regular expression rules with some / that require action code
 def t_NUMBER(t):
     r'\d+'
     t.value = int(t.value)
     return t
 # Build the lexer
 lexer = lex.lex()
```

To handle reserved words, you should write a single rule to match an identifier and do a special name lookup in a function like this:

```
reserved = {
   'if' : 'IF',
   'then' : 'THEN',
   'else' : 'ELSE',
   'while' : 'WHILE',
}

tokens = ['LPAREN','RPAREN',...,'ID'] + list(reserved.values())
```

Transition Diagrams for Scanner :

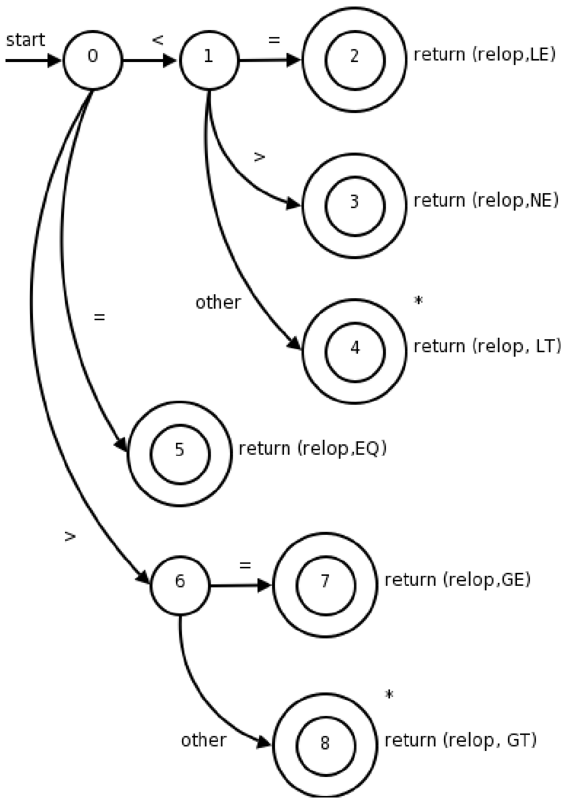
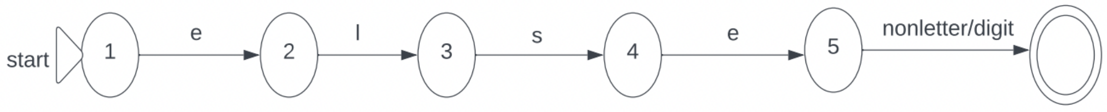
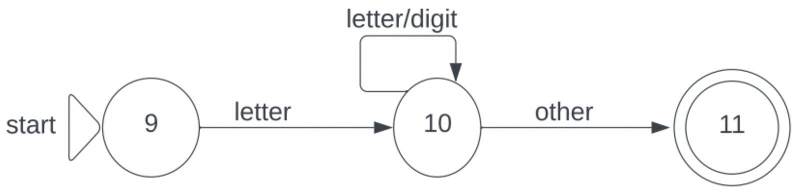


Fig. Transition diagram for relational operator



Hypothetical transition diagram for the keyword else



Transition diagram for keywords and identifiers

For overlapping patterns, where a particular lexeme might match to 2 or more patterns, certain rules are defined. Rules :-

- the scanner should scan the longest possible prefix
- incase the longest possible prefix is the same,

For example, for lexeme ‘for’, the scanner must match it with a pattern for keyword ‘FOR’ rather than an identifier, both of which correctly match the lexeme ‘for’.

For special lexemes like whitespace, tab space, etc, the scanner reads them, and rather than returning the token class to the parser, lexical analysis is restarted on the character that follows these special lexemes.

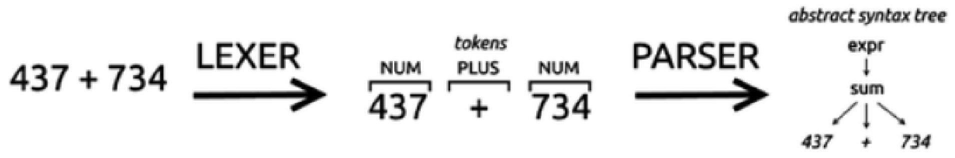
Lexemes that do not match with any patterns, are thrown for error. During error handling, the line of code where the lexeme was found is returned along with a possible error message.

Token Class	**Informal description**	**Sample lexemes**
KEYWORD	predefined, reserved words used in programming that have special meanings to the compiler	if, else, while, for, do, break, int, etc.
CONST	Includes constant values, can be integer, decimal numbers, character, etc.	12, 13.5, 18, ’c’, “string”, etc.
SPL_SYMBOL	Includes punctuations, brackets, etc.	{ , } , ( ) , [,] , ; , etc,
OPERATOR	Includes arithmetic, relational, logical,assignment operators, etc.	=, <, <=, ==, !=, etc.
MOVEMENT	Includes the inputs which user gives for moving the tetrimino while playing the game	moveLeft, moveRight, rotateCW, rotateACW, etc.
WHITESPACE	Includes tabs, blank spaces, blank lines, carriage returns	“ ”, \n, \t, etc.
IDENTIFIER	Letters followed by letters and digits	score, total, value, etc.

Fig. Example of Lexeme and Token Class

Division of Labor between Scanner and Parser

To make the task of the programmer easier, our program design utilizes a large number of tokens - level, score, speed, rotation to name a few. This reduces effort on the part of the programmer, and increases the work done by the scanner and parser. The scanner generates a symbolic table which has the required <lexeme,token class> pairs, which is then passed on to the parser. As an example of a division of tasks between our scanner and parser, both variable name and function name are tokenized by our scanner as 'identifier', but the distinction between the two only occurs in the parsing stage.



When our scanner discovers a lexeme constituting a token, it enters that lexeme into the symbol table. The parser, on receiving that token interacts with the scanner by the 'getNextToken' command, which causes the lexical analyzer to read characters from its input until it can identity the next lexeme and produce the next token for the it, which it return to the parser.

Our parser will concern itself majorly with assignment and arithmetic statements to parse the function blocks and standard function statements.

Parser organizes the code in a tree-like structure (called the abstract syntax tree), so that the code can easily be manipulated in machine language. As an example of a division of tasks between our scanner and parser, both variable name and function name are tokenized by our scanner as 'identifier', but the distinction between the two only occurs in the parsing stage.

Division and Distribution of Roles and Responsibilities Among the Team (Tentative)

We plan to approach each stage of the pipeline collectively. However, the design of major components has been divided as follows :

Scanner : Ricky Patel, Aman Gupta

Parser : Shaurya Puri, Nikhil Mishra

Semantic Analysis : Ricky Patel, Kunal Ashish Kariwala

Code Generation : Shaurya Puri, Aman Gupta

Optimization and Translation* : Nikhil Mishra, Kunal Ashish Kariwala

* (may be subject to change in the future)