‘Top Level Design Specs And Scanner Design’

FOR

Dr. Ramprasad S. Joshi
(Department of Computer Science)

BY

Group 3

Name of the Student	ID Number
Neel Shashikant Bhandari	***2018B1A70084G***
Dev Churiwala	***2018B1A70602G***
Nilay Pradeep Rajderkar	***2018B3A70725G***
Sonakshi Gupta	***2018B1A70614G***
Taher Yunus Lilywala	***2018B1A70609G***
Aastha Bhargava	***2019A7PS0140G***

In the partial fulfillment of the requirements of
CS F363: COMPILER CONSTRUCTION

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
7 March 2022

Table of Contents

Top Level Design Specs **3**

Overall Program Structure: 3

Features: 3

Modes for Programmable Features 5

Pipeline Schema 6

Scanner Design **7**

Transition Diagrams 12

Resolving Design Issues after Deciding Lexicon: 14

Use of Memory: 14

Division of Labor between the Scanner & Parser **15**

Division and Distribution of Roles and Responsibilities Among the Team **15**

Top Level Design Specs

Overall Program Language Choice:

Python is used as the programming language to write the scanner, parser and translator. The end result is a resource for a game developer to build a game using the infrastructure provided to define its parameters.

- Being an imperative language with features for object oriented programming, Python allows for a variety of options to have modularity for the developer to easily add and remove features as well as provides tools to easily define the flow of control for the game.
- It also has a vast array of useful and diverse libraries, primarily for this project one called ‘pygame’ which has tools to construct and save game states and display ASCII objects rendered as graphical units to show the gameplay.
- We will also make use of the SLY library which is a python implementation of lex-yacc.

Features:

A variety of features are planned to be added for the developer to implement:

i) **Modular Grid Size** - Allowing the developer to change the dimensions of the Tetris board.

ii) **Difficulty Modes** - The developer has the flexibility to provide from a set of three difficulty settings: Child, Classic and Extreme. A variety of features get triggered accordingly.

iii) **Block Shapes** - All standard tetriminos. Child mode only has O and I tetriminos while Classic mode has all the standard ones (T, L, J, S and Z in addition to O and I).

iv) **Next Queue** - The programmer can choose the size and orientation of the Next Queue displaying the upcoming Tetriminos. The value for size can be chosen from a fixed range of numbers (one to six) and the orientation can either be horizontal or vertical.

v) **Hold Queue** - The Hold Queue can be enabled or disabled by default depending on the difficulty mode.

vi) **Ghost Projection** - The game programmer has the flexibility to choose the appearance of the Ghost Piece which is a projection of the current Tetrimino and indicates to the player where the Tetrimino will come to rest if it is “dropped’ from its current position. It can appear as an outline or as a translucent image. Furthermore, the programmer can choose to either provide the user with an option to toggle it on and off before any game or automatically keep it switched off in the Extreme difficulty mode.

vii) **Lock Down** - The programmer can decide what the default Lock Down setting will be. The Lock Down time interval can also be customized by the programmer.

viii) **Rotation** - The specific conditions surrounding the rotation action are programmable by the developer.

ix) **Speed Increase** - The programmer has the flexibility to choose the factors upon which the speed of the Tetriminos will depend. The speed of all incoming Tetriminos may increase with time in a linear fashion or it may depend on the number of rows cleared as the user progresses. Additionally, the speed of an individual Tetrimino will depend on specific actions as well - Soft Drop and Hard Drop. Pressing the Soft Drop button will increase the speed of only the current Tetrimino (by a factor that can again be coded by the game developer, say, 20x) for as long as the button is pressed. Choosing Hard Drop will increase the speed such that the Tetrimino immediately drops down (it won’t resume its normal speed once the key is released.)

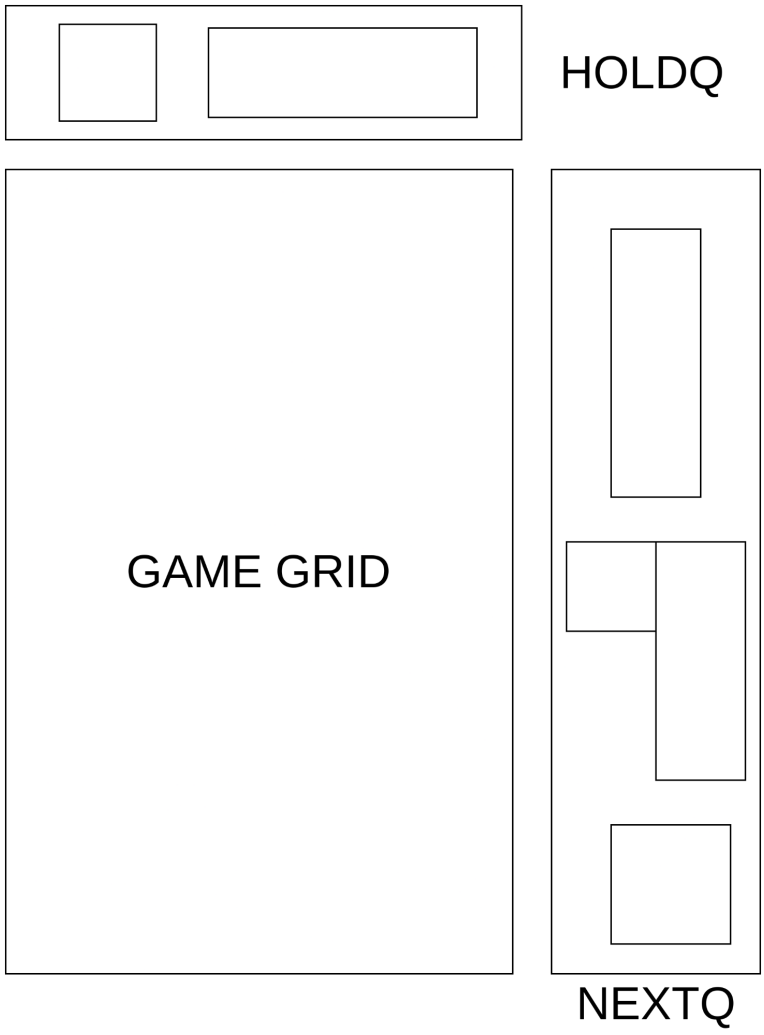
x) **Tetrimino Generation Method** - The programmer can decide if block generation will be random or from another probability distribution, depending on the difficulty mode set. More complex block shapes can be introduced more frequently in the Extreme mode.

xi) **Color Blind Mode** - The blocks can have specific patterns (instead of colors) or colorblind-friendly hues to help the user distinguish between different blocks.

xii) **Other Clearance Patterns** - The game programmer can choose specific patterns for clearance, in addition to or in place of horizontal lines cleared. They can also enable a game mode that would clear blocks if a certain number of same colored blocks are present consecutively (exact number programmable by the game developer.)

xiii) **Animation** - The animation for block movements like rotation can be disabled as per the programmer's choice.

xiv) **Sound** - Background music can be enabled or disabled.



Block implementation of the interface

Modes for Programmable Features

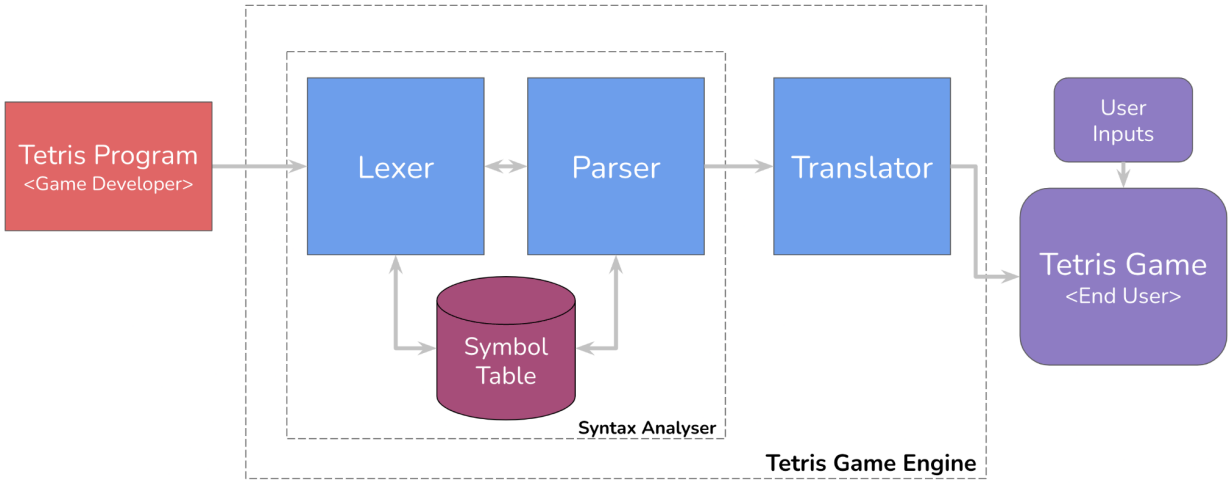
1. **Bool**
    a. true: ‘True’
    b. false: ‘False’
2. **String**
    a. string: ‘“’ STRING ‘“’
3. **Identifier**
    a. iden: [a-zA-Z][a-zA-Z]*
4. **Numbers**
    a. num: [0-9]+
5. **Block**
    a. block: INDENT anyCode DEDENT
6. **If - else - elif**
    a. ifStatement: simpleIf | ifElse | Elseif
    b. simpleIf: ‘if’ expression ‘:’ block
    c. ifElse: ‘if’ expression ‘:’ block
    ‘else’ ‘:’ block
    d. Elseif: ‘if’ expression ‘:’ block
    (‘elif’ expression ‘:’ block)*
    ‘else’ ‘:’ block
7. **For - loop**
    a. for: ‘for’ id ‘in’ expression ‘:’ block

Modular Approach for Programming

For most of the part of the program development, we will try to maintain a modular approach and use most game world terminals as objects.

- We define both the game board and tetromino pieces as objects.
- By changing certain attributes for the class itself, for example, speed for the game board or speed for the tetromino block itself, we try to inherit these properties into the objects of these particular classes as well.
- By keeping data states and attributes for individual objects, we try to maintain the data hiding properties and use encapsulation for each of the modules created.

Pipeline Schema



Pipeline Schema

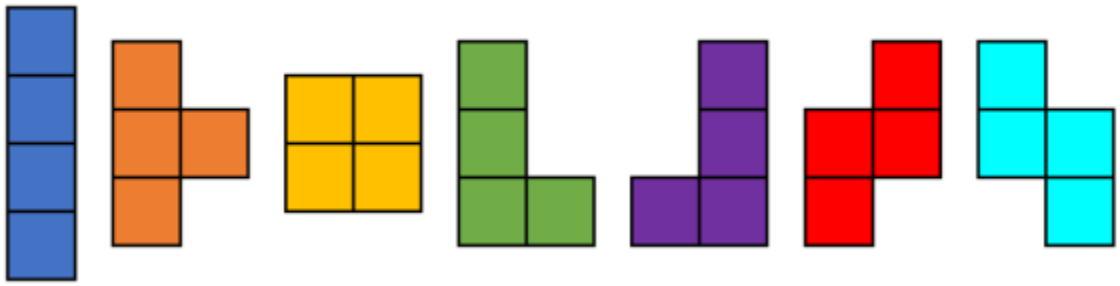
- **Lexer** - The scanning process helps us to identify familiar tokens in our tetris world design. Using the parser, we try to organize the familiar entities and disallow certain features that would not otherwise exist in the game. For example, we would not allow the blocks running outside the game window by continuous right or continuous left movements.

- **Parser** - Organize familiar entities, blocks outside game window not allowed. Decide balance between language and parser. Blocks outside grid not be allowed by the language

- **Translator** - Generate a 2D world which updates stages in real time. Applying laws of physics, so that block stacking looks proper.

Scanner Design

- **Terminals:** In the tetris world, we consider tetrominos as an essential element. We want the tetromino pieces to be named terminals. In this design choice, we consider each tetromino as a separate terminal and hence we have defined 7 different terminals.
- Below are tokens for each of them:
    - I_TET
    - O_TET
    - T_TET
    - L_TET
    - J_TET
    - S_TET
    - Z_TET



Pattern	**Action**
IF	If statement
ELSE	Else statement
TRUE	True boolean
FALSE	False boolean

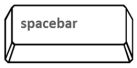
Pattern	**Action**
FOR	For loop
ELIF	Else if statement
IN	In Keyword
RANGE	Range function
PASS	Pass keyword
BOARD	For the entire board
BUFFER	Time and space buffer
ANIMATE	Toggle animation on/off
TUTORIAL	Tutorial Mode
CHILD	Easy mode for Children
CLASSIC	Classic Mode
COLORBLIND	Colorblind Mode
SPEED	Control speed of falling blocks

Pattern	**Action**
SOUND	In-game sound control
NEXTQ	Upcoming blocks
MODE	Chooses from defined modes
EXTREME	Extreme mode
GHOST	Projection of the block
CONFIG	Used to keybind controls
HOLDQ	Blocks already in queue

- We have a second set of token types, the delimiters that give structure to the program.
    - ~tab space~
    - “
    - ”
    - :
    - (
    -)
    - [
    -]
    - {
    - }

- The third type are the variable names, the literal constants.
    - CLOCKWISE
    - ANTICLOCKWISE
    - LEFT
    - RIGHT
    - SOFTDROP
    - HARDDROP

- **Controls:**



Spacebar - Hard drop

Right Arrow - Move right

Left Arrow - Move left

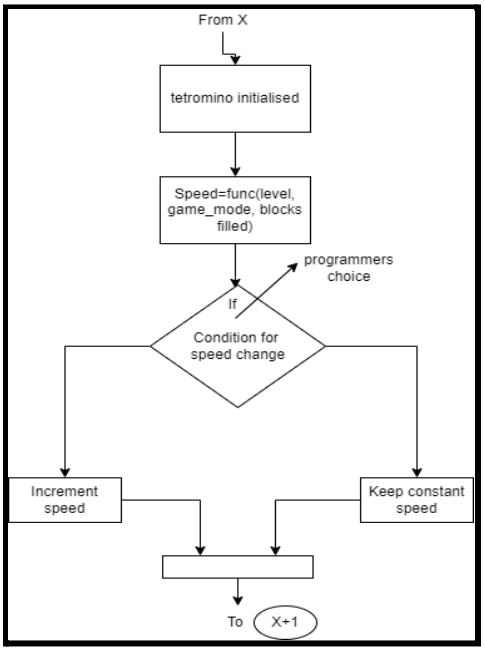
Up Arrow/ X key - Rotate Clockwise

Down Arrow - Soft Drop

Z key - Anticlockwise Rotate

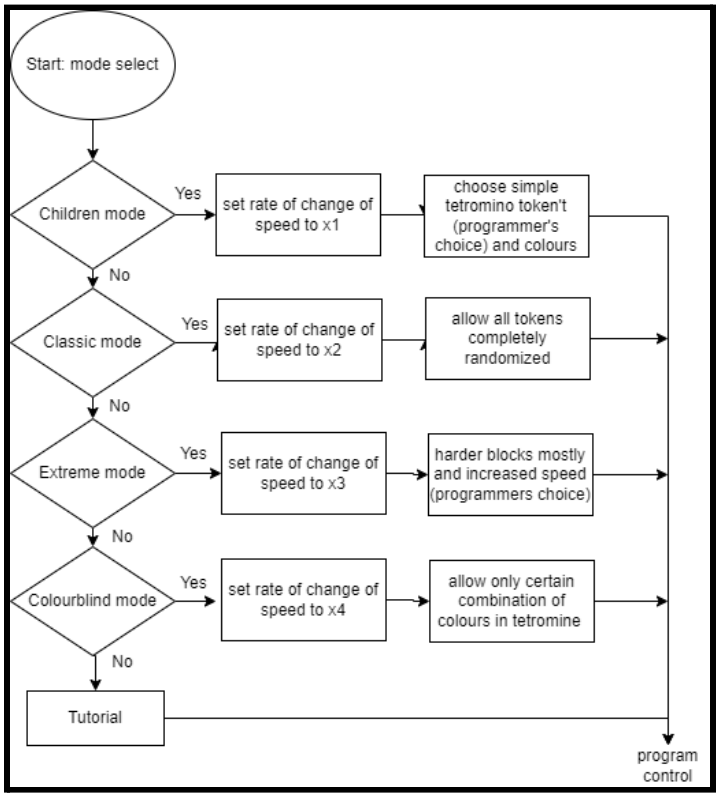
P key - Pause game

- **Program flow control:**
    - We allow the game developer to let the behavior of the engine be dynamically controlled.
    - To allow for a feature to change the speed of the fall after a certain N number of blocks have fallen down on the ground, we provide a conditional flow in the program by the use of **if-else statements**.



Conditional flow for ‘if-else’ statement

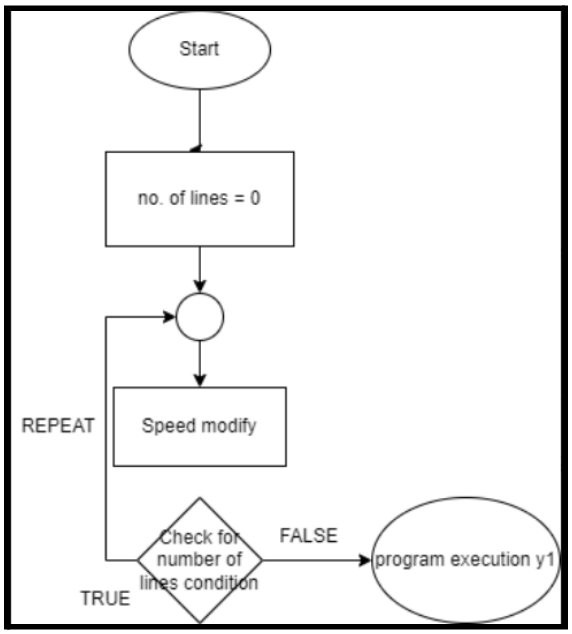
- To allow for different rates of speed change for different levels we use a **if-elif-else statement** to allow for this flow. We provide 4 different modes -
  - Classic mode - For randomized tetromino generation.
  - Extreme mode - With more difficult tetromino shapes and faster speeds
  - Child mode - With easier tetromino shapes and slower speeds.
  - Colorblind mode - With more distinctive colorful blocks.
  - Tutorial - A quick game play tutorial



Conditional flow for ‘if-elif-else’ statements

- We can either make up the gameplay to challenge or assist the player. The number of lines from bottom that cannot be cleared might decide the speed of the game. Hence, there is a dynamic clearing or a fixed clearing option. To allow for speed changes in

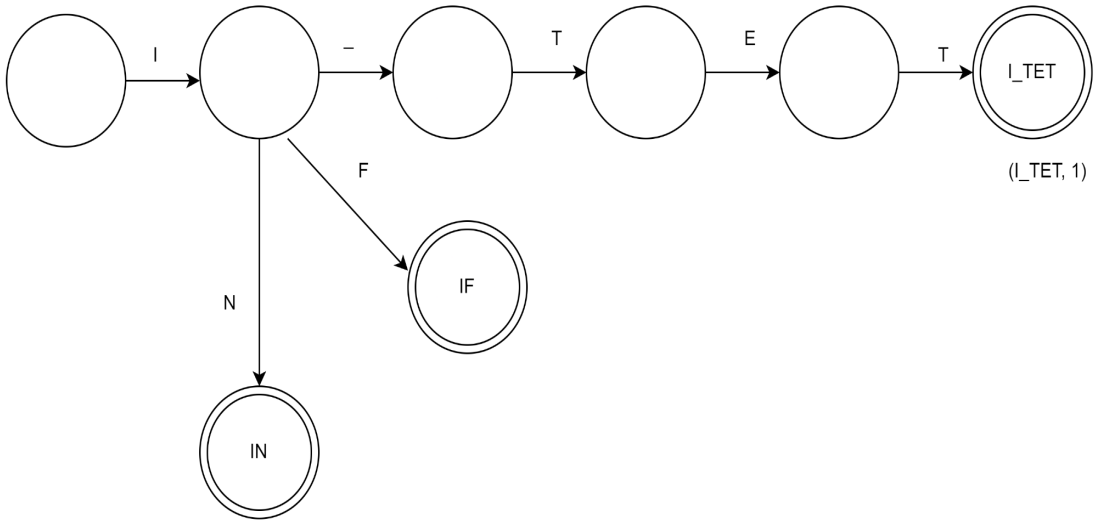
such events, which require condition checking at each iteration, we have to provide for a loop structure. Hence a **for block** is necessary and is provided in the control flow.

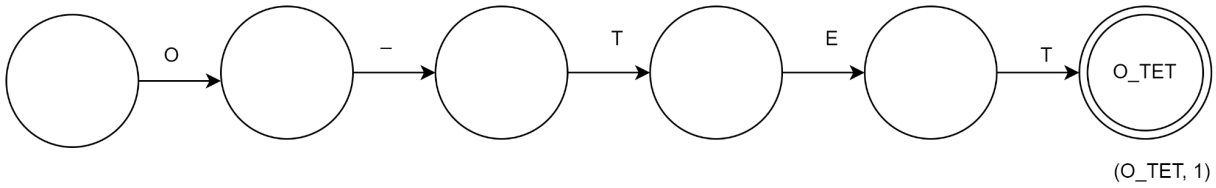


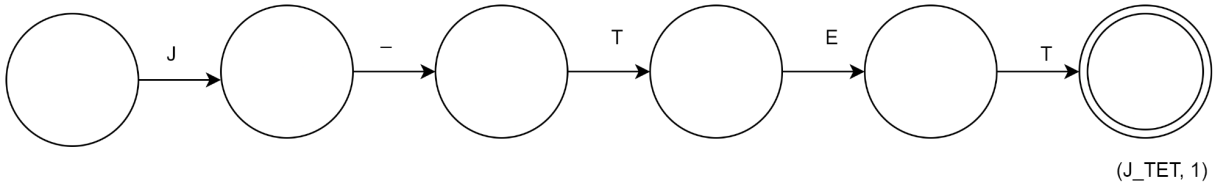
Program Flow for ‘for’ statement

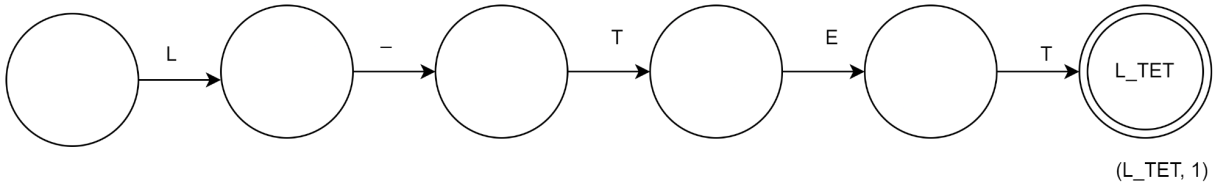
Transition Diagrams

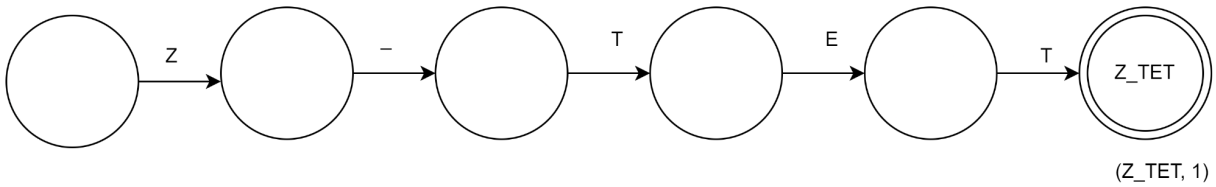
A few transition diagrams related to our lexer:



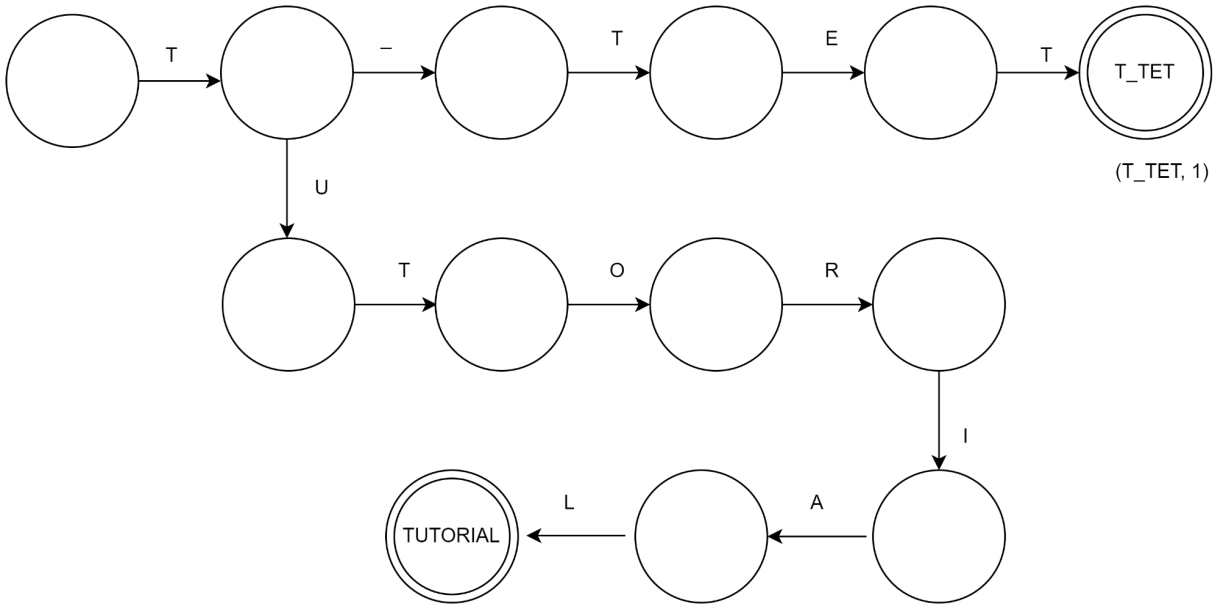




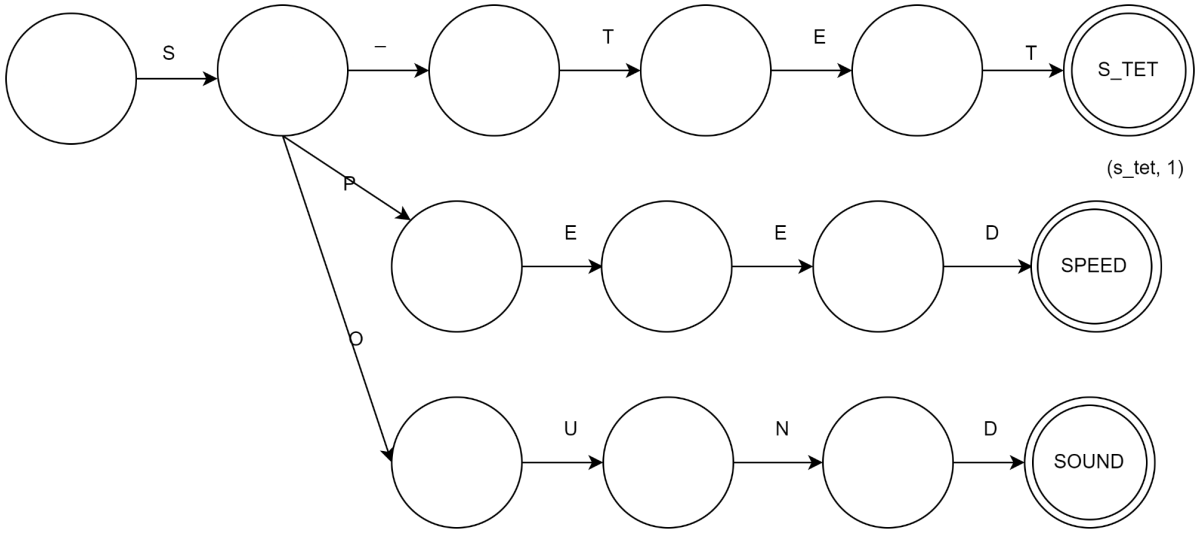




(Z_TET, 1)



(T_TET, 1)



(s_tet, 1)

Resolving Design Issues after Deciding Lexicon:

- After deciding the lexicon as we did above, we have to look at the keywords. We make sure that the keywords are like mnemonics, which are distinctive and make the programming process easy for the game designer. Ex. ROTATE for a rotation of the tetromino.
- We make sure that the keywords don't overlap too much, making it easy for the scanner to separate keywords without much effort.

Use of Memory:

- Since we have considered 7 different terminal token types, we try to reduce memory consumption, since the symbol table will now have fewer entries compared to the case when we have 1 token type and 7 values exist.
- This will also help reduce other complications, such as same names of global and local variables in the memory.
- After rotations, each token type will be converted to a token on another token type. This keeps the Finite State Machines smaller for each case, since final states are reached earlier. (Because each tetromino is a separate token type now). A smaller DFA somewhat reduces the load on the memory as well.

Division of Labor between the Scanner & Parser

- Since we have 7 terminal token types for each tetromino, we minimize the load on the parser. This is because since we have a separate value for each type instead of a token type with 7 values, we have significantly reduced the book keeping in the symbol table. Hence the parser refers to the symbol table less frequently, leading to time and memory saving in this case.
- By keeping the token types for the tetromino separate, we define separate lexicons for each block. By this distinction, the error in types or any inconsistencies will be caught by the parser without much ease.
- We aim to keep the lexicon relatively flexible. We define the speed of the falling blocks as a float, while most other token values will be integers.We allow the game designer to pass any value. In the interpretation/parsing part a proper check will be done by the parser to avoid incorrect values. If speed is a floating point variable when other types are integers, and this is passed ahead to the parser, the parser can spot errors just based on the type of the tokens.
- When a block of tetromino falls on the ground and forms on a row, it now becomes part of the row. Hence each block has various dynamic changes, where it is no more a block object but an element of the row object. Hence, we are consistently, in every such iteration, changing token types during a running game environment. By keeping the token types separate, the symbol table does not have to go through as many changes as it would have gone through if we had a single token type having 7 different values for the tetrominos.
- This would minimize the interaction between the parser and the symbol table, hence taking the load off the parser.

Division and Distribution of Roles and Responsibilities Among the Team

We will be dividing the responsibilities between the parser and translator for the next part of the project.

While we plan on keeping the roles dynamic in nature,
Neel, Dev and Nilay will be responsible for the parser while Taher, Sonakshi and Aastha will be working on the translator. However, because of the considerable overlap, there is a possibility that the roles change in the future.