

TETRISJET

TETRIS PROPOSAL



Tanmay Devale : 2019A7PS0066G

Ramanathan Rajaraman : 2019A7PS0115G

Vishal Sharma : 2019A7PS0036G

Param Biyani : 2019A7PS0059G

Sarthak Chaudhary : 2019A7PS0125G

Vishal Vivek Bharambe : 2019A7PS0160G

This proposal is made as a partial fulfillment of the requirements for the Midsem assignment submission for COMPILER CONSTRUCTION CS F363, under Prof. Ramprasad S. Joshi .

TABLE OF CONTENTS

Tetris Proposal - TetrisJet	1
Top Level Design	2
Overall Program Structure	2
Offered Primitives	4
Program Features	5
Pipeline Schema	6
Scanner Design	7
Transition Diagrams	7
Scanner vs Parser	9
Work Division	9

TOP LEVEL DESIGN

OVERALL PROGRAM STRUCTURE

We are planning a C/C++ inspired language structure. The programmer will be provided with a procedural and weakly typed language. It will support multiple files via the **import** command. This will be done in the preprocessing stage. The language will follow a CSS inspired block overriding idea where the latest definition is considered. The programmer will be editing and adding variations in two ways:

- Config File:
 - Grid Dimensions (Global).
 - Block Shape and Color (Global).
 - Layout (explained in the following sections).
 - Level attributes that can override Global Defaults.

A sample config file:

```
{
  "DefaultGridHeight":10,
  "DefaultGridWidth":6,
  "highScore":0,
  "NumberOfBlocks":5,

  "blockColors":["0xFFFFFF","0xFF0000","0x00FF00","0x0000FF","0xFFFF00"],
  "blockShapes":[[0,0,0,0],[0,0,0,0],[0,0,0,0],[1,1,1,1],
                  [0,0,0,0],[0,0,0,0],[0,0,0,1],[0,1,1,1]],
                  [0,0,0,0],[0,0,0,1],[0,0,0,1],[0,0,1,1]],
                  [0,0,0,1],[0,0,0,1],[0,0,0,1],[0,0,0,1]],
                  [0,0,0,0],[0,0,0,0],[0,0,1,1],[0,0,1,1]],

  "Layout":{
    "R1":"Score",
    "R2":"NextBlock",
    "R3":"CurrentLevel",
    "R4":"HoldBlock"
  },
  "NumberOfLevels":5,
  "Levels":[
    { "velocity":10, "GridHeight":10, "GridWidth":6 },
    { "velocity":12, "GridHeight":9, "GridWidth":6 },
    { "velocity":14, "GridHeight":9, "GridWidth":6 },
    { "velocity":16, "GridHeight":9, "GridWidth":6 },
    { "velocity":18, "GridHeight":9, "GridWidth":6 }
  ]
}
```

- Code Files
 - The programmer will be able to edit event triggered functions to modify behavior
 - The functions will not be returning values and will each correspond to a specific point in the game execution loop. This was inspired by from the C# interface provided by the Unity3D game engine
 - The scope of each function will be pre-decided

```
import "path_to_file"
onCollisionEnter(Block block){
    block.colour = 'g';
}
onCollisionEnter(Block block){
    block.colour = 'b';
}
onLevelOver(){
    log("Finish");
}
```

KEYWORDS

- Data Types: block, grid, bool, int, char, float
- Control Constructs: if, else
- Loop: while, break, continue
- Modularity: import
- Macros: def, end
- Events: onCollisionEnter, onCollisionExit, onBlockSpawn
- Movement: goLeft, goRight, goUp, goDown, rotCW, rotACW

Note: The keyword list here is inexhaustive.

OFFERED PRIMITIVES

The following are the various regions whose positions can be modified by the programmer

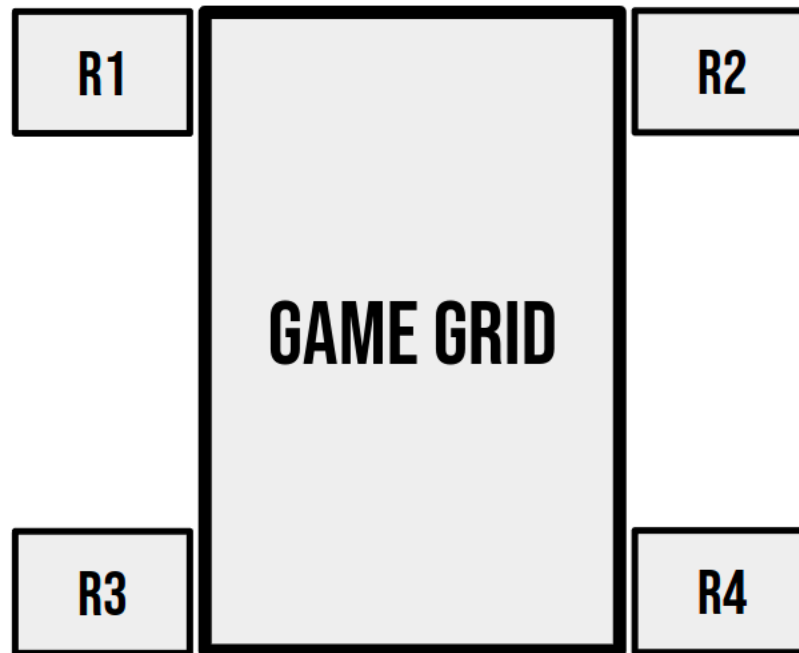


Fig: A sample layout for the GUI

Possibilities for R1 to R4:

- Current and High-Score
- Level Information
- Next Piece
- Hold Piece

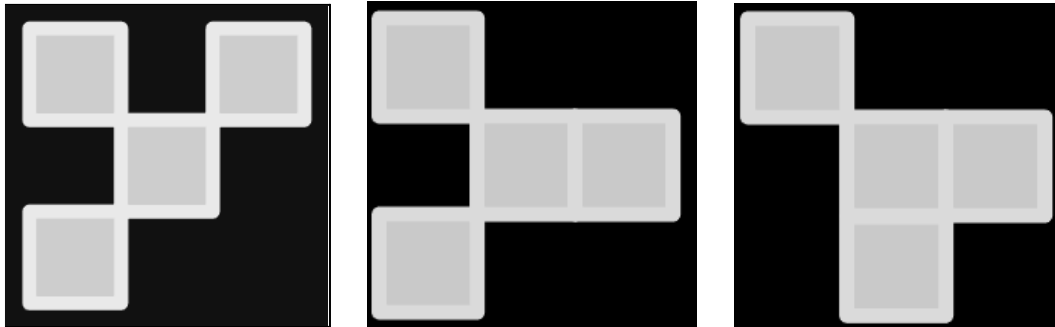
The programmer would be able to modify the layout of the game screen using various attributes such as position of the different regions (game grid, R1, R2, R3, R4), the structure and color of the blocks and level attributes such as grid size and velocity of the falling blocks. Apart from this the programmer can also write the logic for what happens based on certain pre-defined in-game events for eg., `onCollisionEnter`, `onLevelOver`. The following are a few examples (inexhaustive) of custom game features that the programmer can implement:

- If the game designer wants to have some blocks initially positioned on the game grid at the beginning of a level, they can make use of the `onLevelInit` event and define the positions and shapes of the blocks to be placed thus giving programmers a freedom of level design in terms of structure of the initial game grid.
- If the game designer can set different behaviors for different collision types (with walls, with pre-initialized blocks, with blocks on the playing field, etc) then they can do so using the `onCollision` event and modifying the behavior of the colliding entities.
- If the game designer wishes to have a different behavior to update the score when a row

clears, they can do so using the onRowClear event.

The structure of the blocks can be defined by changing the block attributes exposed to the programmer to set the position of 4 blocks anywhere on a 4x4 grid and specify their color and if the block should have any special functionality(eg. power ups , etc).

Sample custom sized pieces are shown below:



PROGRAM FEATURES

We plan on allowing support for the **while** loops, **if-else** conditionals, **break-continue**, **for** loop control and macros as a preprocessor directive. We justify these choices with the following:

- The main game loop will be handled without the programmer's intervention and hence we do not believe they will require powerful loop structure. For any form of repeated calculations, the provided **while** loop will suffice
- The use of **functional macros** was decided upon as it would reduce scoping issues and would make the parsing and translation more streamlined

PIPELINE SCHEMA

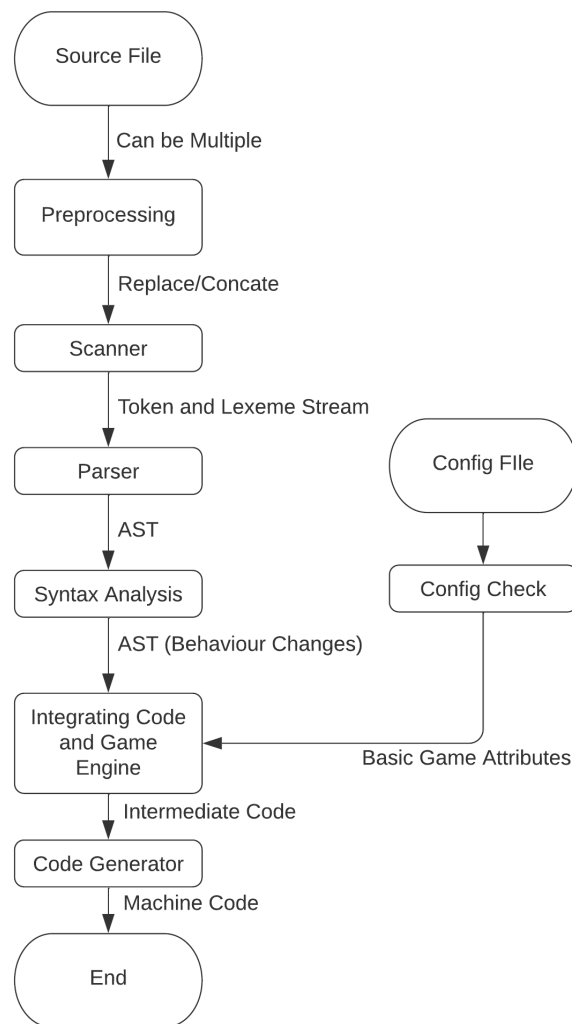


Fig: General Pipeline from Game Programme to Machine Language.

PREPROCESSING

At this stage the **import** and **macro** statements are replaced/injected to generate the complete input file for the scanner. Our plan is to concat all import files then following the CSS idea of keeping the latest definitions in a top-down priority.

SCANNER DESIGN

TRANSITION DIAGRAMS

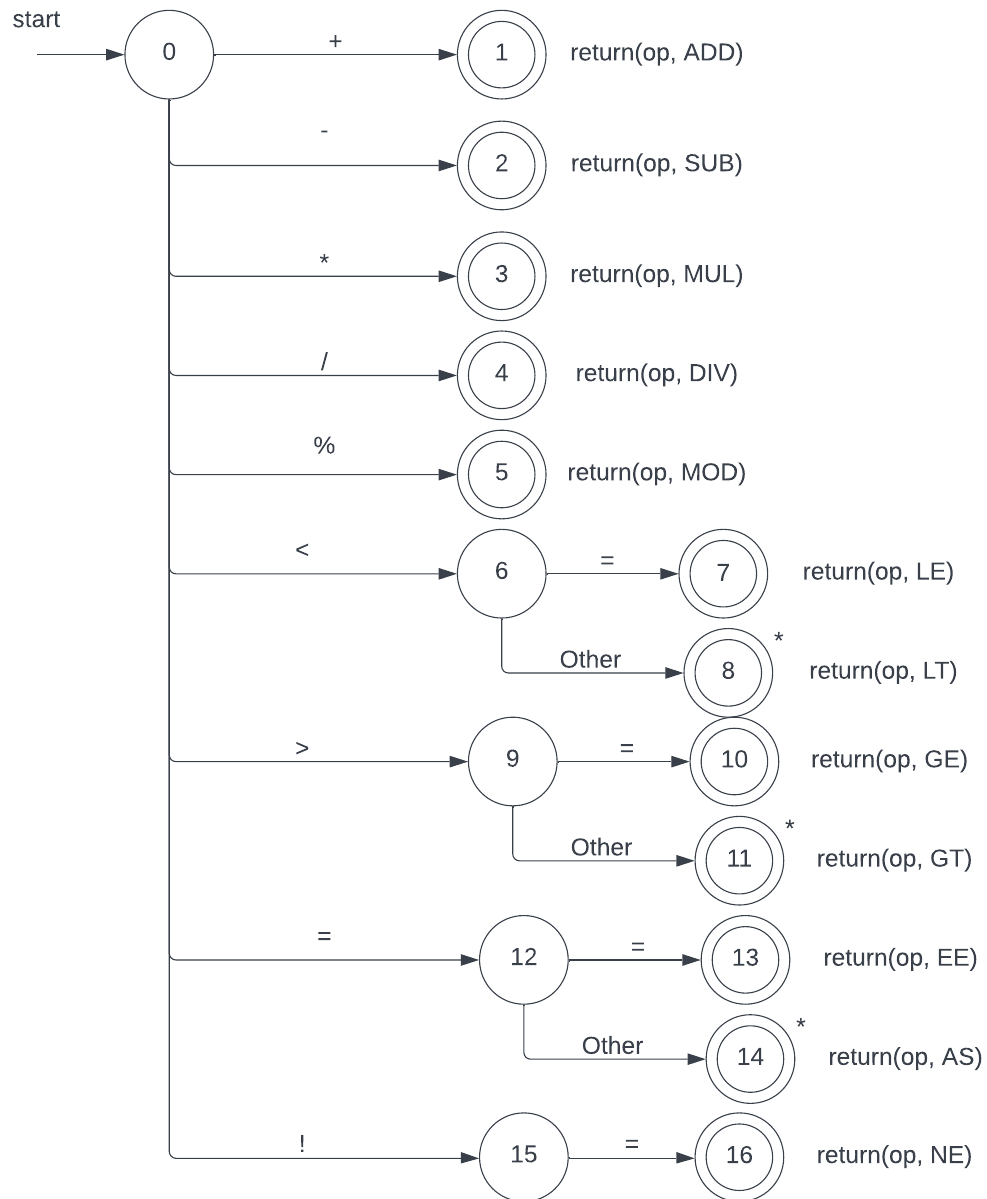


Fig: Transition Diagram for Operators

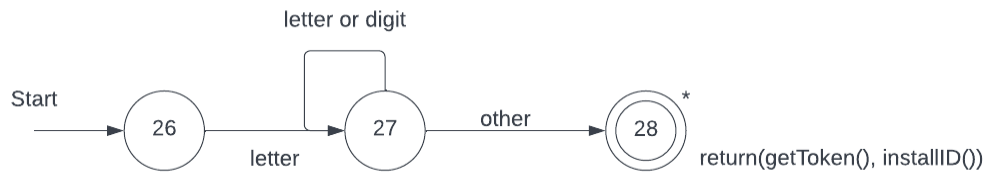


Fig: Transition Diagram for Identifiers

To handle keywords we have planned to install them in the symbols initially. The symbol table indicates that strings are not ordinary and it tell which token they represent. The function *getToken* examines the symbol table entry for a lexeme found and returns whatever token name the symbol table says the lexeme represents either id or keyword as indicated in the symbol table.

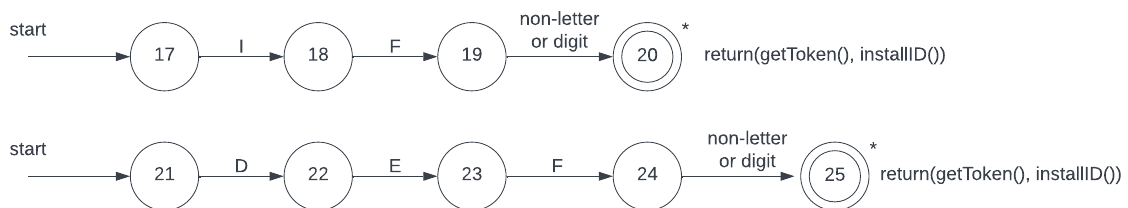


Fig: Transition Diagram for Keywords (if, def).

The handling of keywords (reserved word tokens) must be prioritized before that of identifiers.

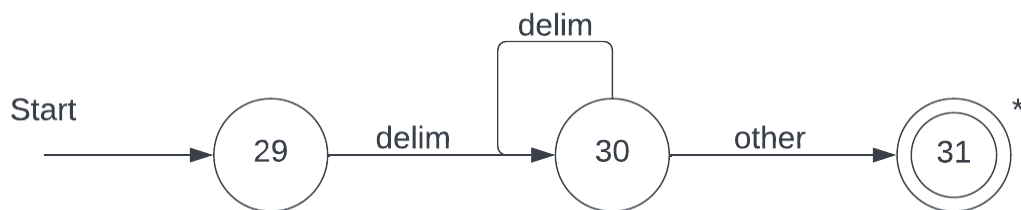


Fig: Transition Diagram for Whitespace.

*: In State 31, we discovered a string of whitespace followed by a non whitespace character. We return the parser but retract the input to start at the non whitespace. After whitespace, we must resume the lexical analysis procedure.

SCANNER VS PARSER

Our Design relies on a large number of independent tokens for all events and movement commands. This in turn means that our pipeline will be scanner heavy. We made this decision as it gives us the most streamlined procedure after the parser and helps us immensely in the modular intermediate code generation we are planning. Our parser will work mainly on assignment and arithmetic statements while exploiting the high token count to parse the function blocks and standard function statements.

WORK DIVISION

Ramanathan Rajaraman :

- Game Physics
 - Collision
 - Rotations
- Event Systems

Vishal Sharma :

- Layout Design
- Score Handling
- Event Systems

Tanmay Devale :

- <ADD>

Vishal Vivek Bharambe :

- <ADD>

Param Biyani :

- Special Collision Effects
- GUI and Assets

Sarthak Chaudhary :

- Game Physics
 - Grid Analysis
- Event Systems