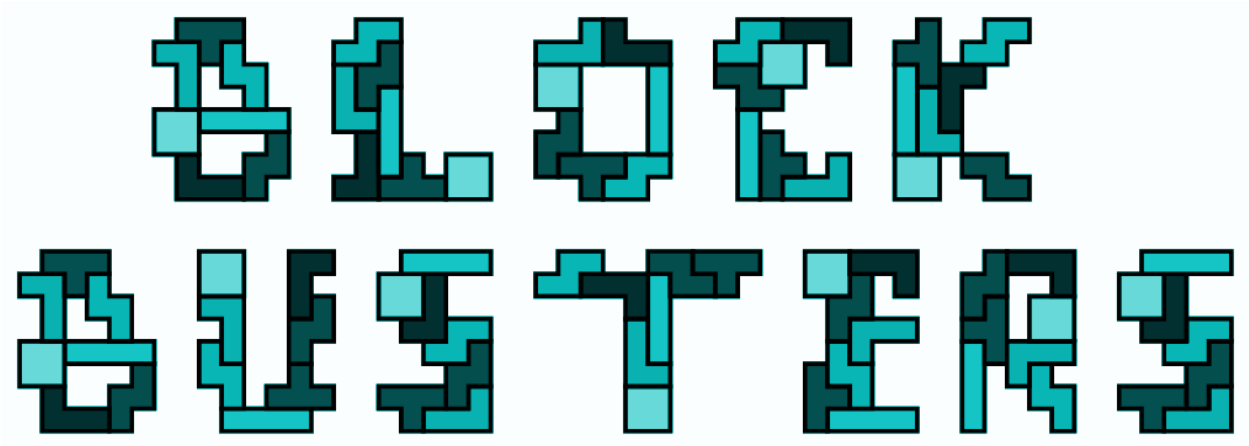
CS F363 Compiler Construction
Group-17



ID	Name
2019A7PS0076G	Hardik Shah
2019A7PS0091G	Abhineet Karn
2019A7PS0136G	Aryan Tyagi
2019A7PS0167G	Abhinav Srivastava
2019A7PS0178G	Hitarth Kothari
2019A7PS0179G	Hrishikesh Kusneniwar

Table of Contents:

1. Top - Level Design Schema:

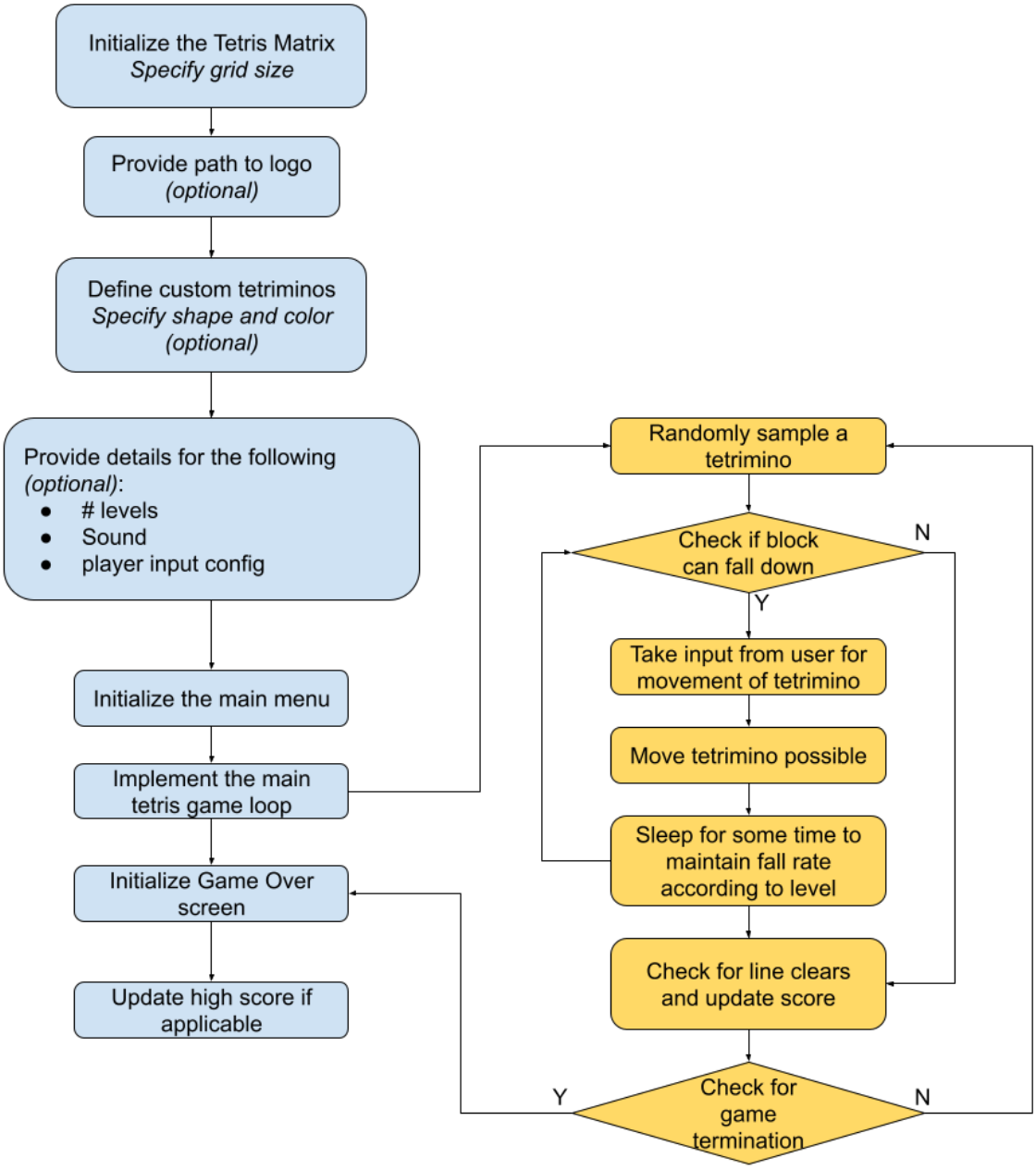
The overall program structure of the Tetris game language will be based on python. The designed language will be called Tetris Language (Tlang) and a file written by a tetris game programmer in Tlang will be saved with a *.tl* extension.

The *.tl* file which is the source program will be converted into a high level language code i.e. python, the target program, by a compiler. Finally, the generated target code must be executed by the programmer to run the tetris game. The target code will make use of an existing engine code which has certain parts of the tetris code implemented in python.

1.1 User primitives:

- Game Logo:
    - The programmer has an option to modify the logo of the game he wishes to build. The programmer must give the path of the logo(as .png/.jpeg file)
- Grid Size of Tetris Matrix:
    - The programmer can control the height and width of the tetris game window. A check is done during interpretation so that the height and width entered by the user satisfies the requirement 8 < gridX < gridY.
- Buffer Size of Tetris Matrix:
    - The programmer can set the size of the buffer zone.
- Custom Tetriminos:
    - The programmer can introduce a custom tetrimino in addition to the already available default 7 tetriminos by the Tlang. The custom tetrimino must be limited to a size of 4x4 matrix cells. The following must be mentioned by the programmer when introducing a new tetrimino:
        - Shape: a 4x4 matrix with 1s denoting occupied and 0s denoting unoccupied blocks by the tetrimino
        - Color: A list of three values in RGB
        - Rotation: The possible configurations of the tetrimino when rotated
- Number of Levels:
    - The number of levels that the game will offer to the player can be defined by the programmer, subject to the constraint 0 < # levels < 5. A greater level indicates greater falling speed of the tetrimino.
- Sound Effects:
    - The programmer can control whether the game will have the background music on or not during the game. The music itself can be chosen from a list of options given to the programmer.
- Player Input:
    - The programmer can configure which keyboard key, from a fixed set of keys, corresponds to the rotate and lateral movement of the tetriminos. The user can also attach additional functionality such as updating scores based on keypresses.
- Print Functionality:
    - The programmer can use a print statement for debugging.

Apart from the above mentioned features which can be used, a programmer attempting to create a Tetris game using TLang is expected to implement the main game loop using the functionalities provided by the language design. Implementations of some requisites of the Tetris Game which cannot be modified, like the game termination condition, are given to the programmer predefined in the engine file written in python. Below is a flowchart which depicts the ideal flow of code for a programmer using TLang:



The following are some of the functions defined in the language and implemented in the engine file are readily available to the programmer:

- initialize_window(*height, width, buffer_height*):
    - The tetris window will be initialized corresponding to the arguments in the function call.
    - Default (if the function is not called by the programmer): default size of tetris matrix

- set_logo(*logo_path*):
    - The logo will be set to the image whose path is passed as a string.
    - Default: original tetris logo.

- add_block(*block*):
    - The custom tetrimino defined by the user will be added to the set of tetriminos that will be considered while sampling a tetrimino. The argument should be of the *block* datatype which is given in the language definition.
    - Default: O, I, T, L, J, S and Z tetriminos

- set_levels(*levels*):
    - Set the no. of levels
    - Default: 1 level (fall speed = 1.0 sec/line)

- set_sound(*sound_path*):
    - The background will be set to a sound file whose path is passed as a string.
    - Default: no background music.

- Input keyboard config:
    - Built-in functions like on_left_button(), on_W_button() allow the programmer to add custom functionality when some input is received.

- main_menu():
    - The main menu window of the tetris game will be displayed. The user will select the level here and the number of levels offered will be displayed according to the number of levels set by the programmer using *set_levels*.

- sample_tetrimino():
    - A tetrimino from the set(default + programmer defined) will be sampled randomly.

- block_fall():
    - Returns true if the current tetrimino can fall further down, false if it has dropped

- user_input():
    - Checks if user has given any input on the keyboard

- move_tetrimino():
    - Moves the tetrimino according to the user input(if given) only if possible. If the movement desired by the player is not possible, it is just moved down.

- sleep():
    - A delay is introduced in order to maintain the fall rate according to the level chosen by the player.

- line_clear():
    - Clear all rows that are full and update the score.

- is_gameover():
    - Returns true if the game termination condition is satisfied. If not, returns false

- gameover_window():
    - The game over screen is displayed which displays the score and updates the high score if applicable.

1.2 Programmable features:

- Variable declaration and assignment:

  The programmer can declare different types of variables using the **“var”** keyword without having to explicitly specify the type. Type of the variable is handled by the python backend. Examples:

  Declaration

  - - - - - - - - -

```
var name1 := True;
var name2 := 42;
var name3 := "string";
var name4 := [ ];
```

  Assignment

  - - - - - - - - -

```
name := value
```

- Operations:

  The programmer can perform various arithmetic, boolean, relational and bitwise operations as mentioned in the table below.

Type of Operation	Operators Provided
Arithmetic	`*`, `/`, `%`, `+`, `-`
Boolean	`not`, `and`, `or`
Relational	`<`, `<=`, `>`, `>=`, `==`, `!=`
Bitwise	`!`, `&`, `<<`, `>>`, `^`, `|`

- Function declaration and call:

  The programmer can declare functions using the **“function”** keyword and specify the arguments enclosed by parentheses and separated by comma. Compound statements are enclosed within the **“begin”** and **“end”** keywords. Sample function declaration and call code is shown below.

  Declaration

  - - - - - - - - -

```
function fname(arg1, arg2) begin
     <stmt 1>
     <stmt 2>
         .
         .
         .
end
```

Call

```
fname(val1, val2);
```

- Conditional Statements:

  Our language provides the flexibility of using conditional statements to the programmers in the following manner using the **“if”**, **“then”**, **“else”** keywords.

  Usage

```
if <expr> then <stmt>;
if <expr> then <stmt> else <stmt>;
```

- Loop Statements:

  The programmer can make use of loops for controlling the flow of the code. Our language provides **“while”** and **“for”** loops that can be used in the following manner.

  Usage

```
while <expr> do <stmt>;
do <stmt> while <expr>;
for i := 0 to 10 do <stmt>;
```

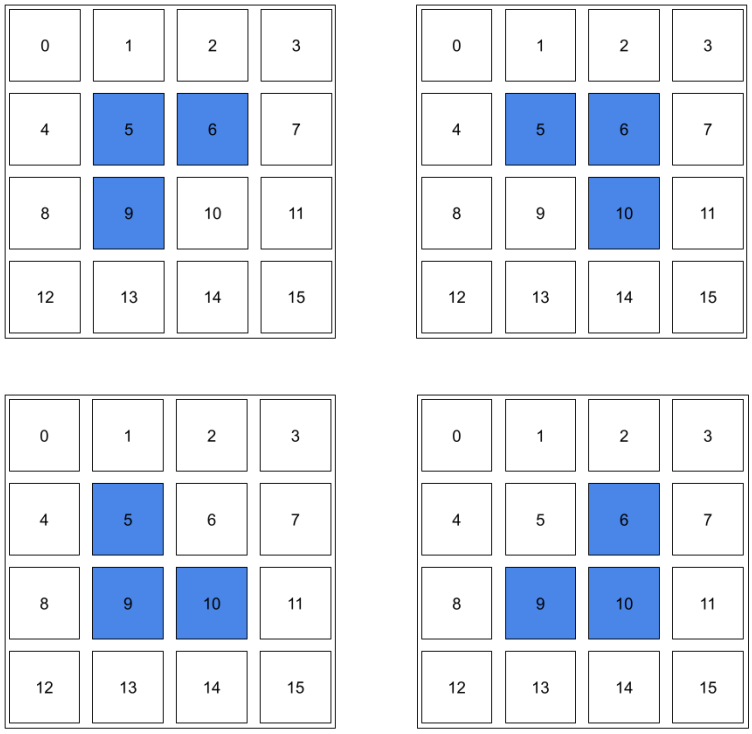
- Custom data types:

  The language provides functions for creating common game objects such as new blocks and sound. Here's a code snippet demonstrating the creation of a new red colored block.

  Usage

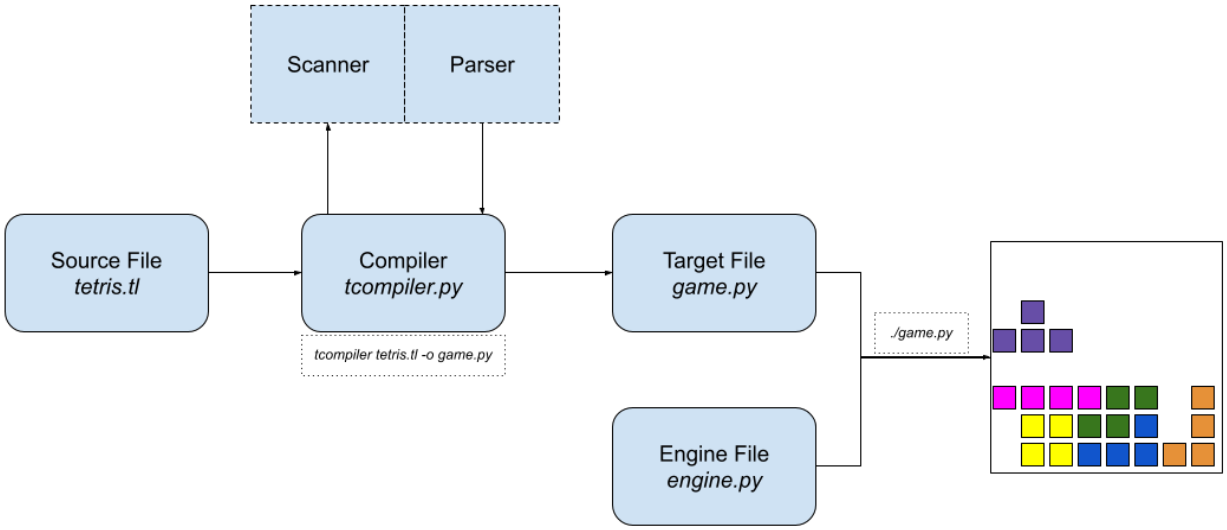
```
var newBlock := createBlock([[5,6,9],
                             [5,6,10],
                             [6,9,10],
                             [5,9,10]],
                            [255,0,0]);
```

`createBlock` is an inbuilt function in our language. It takes the configuration and color as inputs and returns a new block. The configuration parameter is a list of different orientations the block can be in. The orientation is defined as a list of indices of a 4x4 matrix (customizable!) which the block occupies.



1.3 Pipeline Schema:

The code will be written by the programmer in the TLang(*.tl)* and will be converted to the target code in python via a compiler(*tcompiler.py*) written in python. The compiler will use the SLY python library for scanning and parsing the source file and will eventually output an executable python file *game.py*. Finally, the game can be executed by running the *game.py* which will use the *engine.py* file.



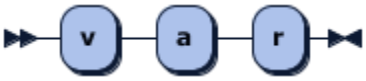
The engine file consists of an implementation of the tetris game in parts which the programmer can use. The programmer is not expected to program the basic logic of the tetris game that cannot be modified for e.g. the game termination condition.

In our project, a valid python code is generated in the parse step itself i.e. *game.py*. The action for a recognised BNF expression by the parser in the source code is to first convert the expression into valid python syntax and next to write this syntax into the executable *game.py*. Once parsing is complete, the implementations in *engine.py* are imported into the *game.py* file which can now be directly executed by a player to run the tetris game.

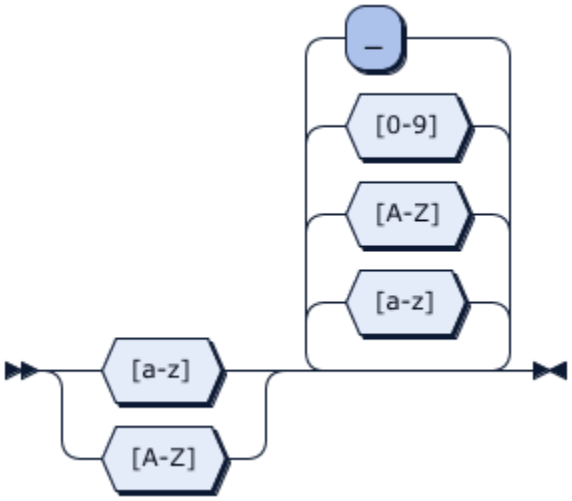
2. Scanner Design:

The overall structure of the Scanner will use the Lexer implementation from the SLY Python library. We have defined the various possible tokens and their pattern, which is a description of the form that the lexemes of a token may take. The following is a subset of the pattern-action pairs.

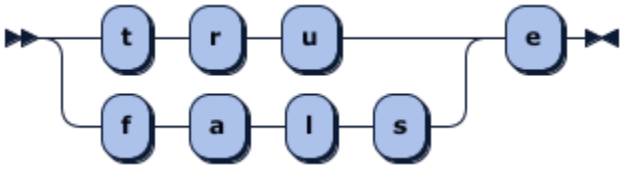
```
"var"                       { return VAR;       }
```



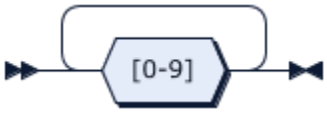

```
[a-zA-Z][a-zA-Z0-9_]*       { return IDENT;     }
```



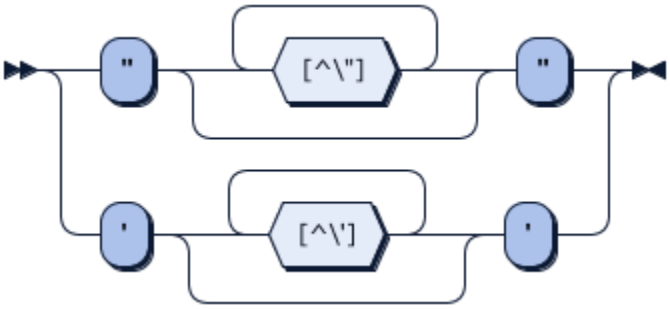

```
"true"|"false"   { return BOOL;      }
```




```
[0-9]+                      { return NUM;       }
```




```
\"[^\"]*\" | \'[^\']*\'              { return STRING;    }
```



*more diagrams for the pattern-action pairs can be found in the lex-diagram folder.

2.1 Lexer implementation using Python library SLY (Sly Lex Yacc) :

Our scanner utilizes the predefined lexer class of the `SLY` library. Tlang scanner inherits from the SLY scanner. We store the names of all the tokens in the set `tokens`. Then we define the regular expression, to map suitable lexemes for each of these tokens. Tokens are specified by writing a regular expression rule compatible with the `re` module. In the next step, we invoke the `tokenize()` method which is a generator function that produces a stream of `Token` instances. `Token` contains type name and value as attributes. A special token `ignore` is used to skip over whitespaces and tabs. A literal character is a single character that is returned “as is” when encountered by the lexer. We are not using any literals in our lexer. We are also storing the line number by using the `ignore_newline()` method and the column number by using the index attribute for a token for the purpose of debugging. Due to the large amount of tokens and lexemes that they map on to, there exists some overlapping between the lexemes and the tokens, which can be solved using the special cases method provided by the SLY library.

A good way to visualize these discrepancies between the patterns of lexemes, is to look over some examples:

Consider a tokenizer, that maps the lexemes following the regex : `[a-zA-Z][a-zA-Z0-9_]*` to the token `IDENT` and the literal `true` to token `BOOL`.

`var true123 := true;`	`type='VAR', value='var'`
	`type='IDENT', value='true123'`
	`type='ASSIGN', value=':='`
	`type='IDENT', value='true'`
	`type='EOL', value=';'`

The above example illustrates how an incorrectly designed lexer may scan the lexemes and generate the tokens. An overlap between `IDENT` and `BOOL` tokens occurs here, where the literal `true` in the input expression does not map to `BOOL` but rather to `IDENT`.

`var true123 := true;`	`type='VAR', value='var'`
	`type='BOOL', value='true'`
	`type='NUM', value='123'`
	`type='ASSIGN', value=':='`
	`type='IDENT', value='true'`
	`type='EOL', value=';'`

One might think the logical solution to the above error is to rearrange the order of declaration of tokens, and define the `BOOL` token before the `IDENT` token. However, after implementing the solution, we are faced again with a rather intriguing problem. Since the literal `true` is mapped

to `BOOL` before `IDENT`, while defining the variable name as `true123` the scanner considers the `true` portion of the variable name as the `BOOL` token and the rest as `NUM`, when it should actually be the case where the entire variable name (`true123` in this case) should map to the token `IDENT`. This is where the versatile error handling capabilities of SLY are best utilized.

SLY allows us to create special cases, or rather remapping of tokens, where we can map the literals like `true` and `false` to token `BOOL`, such that while parsing, the lexemes get matched to the new token type, and not to the ones defined earlier on. An example of how one can code this, using the library in python, is shown below :

```
IDENT = r'[a-zA-Z][a-zA-Z0-9_]*'
...
IDENT['true'] = BOOL
IDENT['false'] = BOOL
```

Introducing changes to codes like this, finally gives us the required output :

`var true123 := true;`	`type='VAR', value='var'`
	`type='IDENT', value='true123'`
	`type='ASSIGN', value=':='`
	`type='BOOL', value='true'`
	`type='EOL', value=';'`

A demonstration of running the scanner on the terminal along with the output :
*check scanner.py for more details

```
45      IDENT['var'] = VAR
46      IDENT['true'] = BOOL
47      IDENT['false'] = BOOL
48      IDENT['if'] = IF
49      IDENT['not'] = NOT
50      IDENT['and'] = AND
51      IDENT['or'] = OR
```

```
PROBLEMS 59  OUTPUT  TERMINAL  GITLENS  COMMENTS  DEBUG CONSOLE

~/Des/S/CC/CS_F363-Compiler-Construction/A/scanner ❯ python3.8 scanner.py
var true123 := true;
type='VAR', value='var'
type='IDENT', value='true123'
type='ASSIGN', value=':='
type='BOOL', value='true'
type='EOL', value=';'
```

For error handling, SLY ensures that if a bad character is encountered while lexing, tokenizing will stop. However, we can add an error() method to handle lexing errors that occur when illegal characters are detected. The error method receives a `Token` where the value attribute contains all remaining untokenized text.

2.2 Scanner test plan and test cases:

Since our scanner runs through the command line, we can test it using a simple bash script. We can write various instructions in the Tetris language organized into files in the input folder. We then provide the expected output in a different folder following the same naming convention. We then loop over all input files and store the compiled output in the output folder. Then we can use the diff command to check whether the generated code matches the expected output.

The command to compile all input files:

```
cd input
find . -type f | xargs -I {} -n 1 bash -c '../tcompiler {} > ../output/{}'
```

The command to check the resultant output with expected output:

```
cd ../output
find . -type f | xargs -I {} -n 1 bash -c '
 if diff {} ../expected/{} > /dev/null; then
   printf "TEST PASSED: {}\n"
 else
   printf "TEST FAILED: {}\n"
 fi
'
```

*see test folder for details on gitlab

Sample test cases:

Input	Expected Output
`score := score + 1;`	`score=score+1`
`var name := "Test";#`	`name="Test"`

Combining the two commands into a single bash script, gives the following result on the given test cases:

```
./runner.sh
TEST PASSED: variable_assignment
TEST PASSED: variable_declaration
```

3. Division of Labour between the Scanner and the Parser:

Suppose we are parsing the following string `x := 42`. The first step of parsing is to break the text into tokens where each token has a type and value. The Scanner is used to do this. It generates a stream of tokens containing the type and value of the token. The output of running the Scanner on the given string might be `[ ('IDENT', 'x'), ('ASSIGN', ':='), ('NUM', '42') ]`. The Scanner can also use RegEx pattern matching to assign a type to a group of characters. This is useful when we want to parse some text whose exact value we might not know. The identifier `x` and the number `42` were matched using th patterns `[a-zA-Z][a-zA-Z0-9_]*` and `[0-9]+` respectively. The scanner is also responsible for ignoring whitespaces and comments.

Once we have a stream of tokens ready, the Parser can be used to assign structure to tokens. For example, if the Parser sees the tokens `IDENT`, `ASSIGN` and the next token is of type `NUM`, it can deduce the input to be a declaration statement. The way a parser achieves this is using a stack. It stores all the tokens it has seen so far in a stack. The token on the top determines what happens next. If the next token is a part of a rule in the grammar using the other tokens on the stack, the stack is reduced using the provided action, else the token is shifted onto the stack. If no further match can be found the parser throws an error. The stack must be empty when the parser has finished parsing all the tokens. The grammar of the language is usually defined in BNF notation. A successful parse usually results in an Abstract Syntax Tree. This AST is a machine readable form of the input tokens. What kind of AST is generated is completely controlled by the action described in the pattern-action pairs.

Appropriate tree traversal algorithms can be used to traverse the tree. Linters are an example of a tool which finds common bugs like using a variable before it is defined. Interpreters read the AST, and execute instructions based on the current node of the tree. ASTs can also be compiled to other languages. In our project, we use appropriate actions to generate a valid python code in the parse step itself.

To summarize:

- The scanner handles low-level character-by-character analysis.
- The parser embeds a scanner and handles higher level language analysis.

The scanner extracts the tokens but knows nothing about the grammar. The parser handles the grammar, but knows nothing about the original string. This keeps the code modular and simple.

4. Syntax Directed Translation Scheme:

Syntax Directed Translation Overview

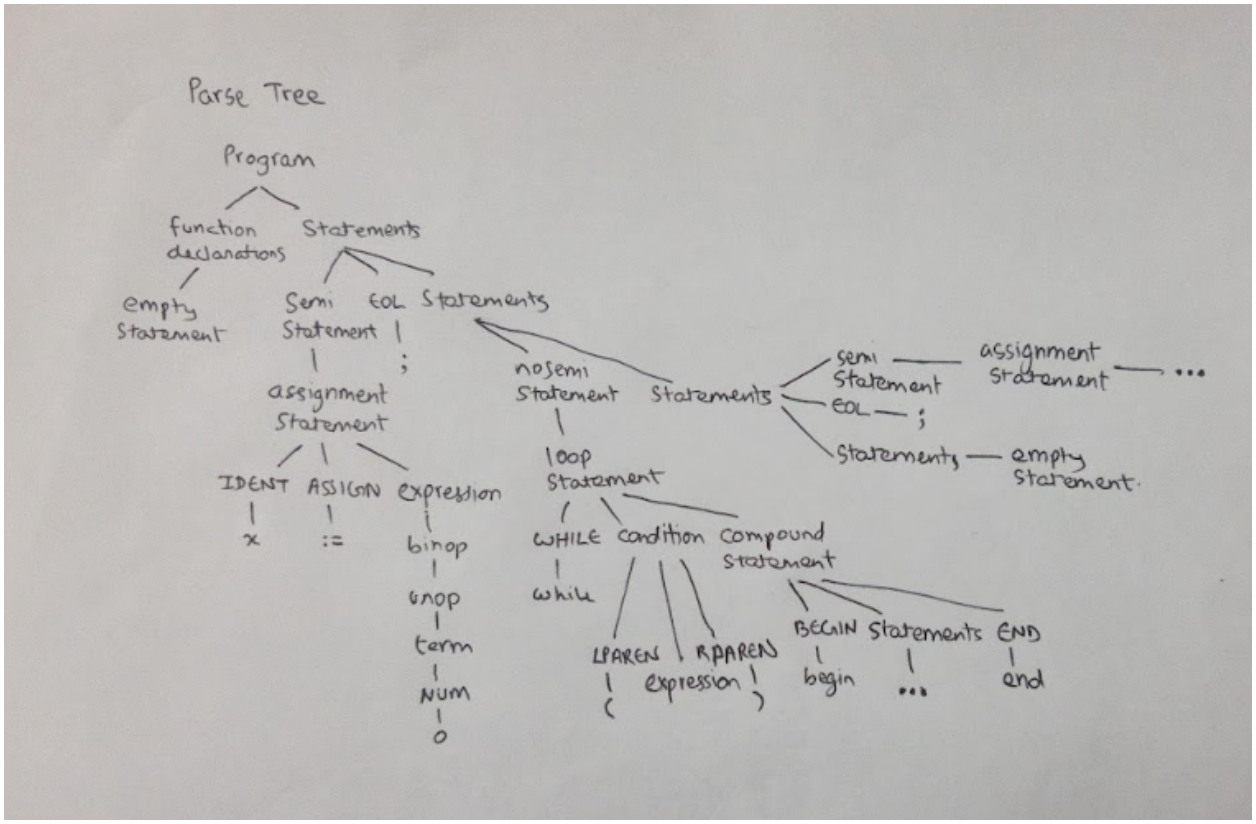
Token Stream - - - grammar rules - - > Parse Tree - - - translation rules - - > Abstract Syntax Tree

SLY uses a parsing technique known as LR-parsing or shift-reduce parsing. LR parsing is a bottom up technique that tries to recognize the right-hand-side of various grammar rules. Whenever a valid right-hand-side is found in the input, the appropriate action method is triggered and the grammar symbols on the right hand side are replaced by the grammar symbol on the left-hand-side. The underlying implementation is built around a large finite-state machine that is encoded in a collection of tables.

A syntax-directed translation is used to define the translation of a sequence of tokens to some other value, based on a CFG for the input. It is defined by associating a translation rule with each grammar rule. A translation rule defines the translation of the left-hand-side nonterminal as a function of the right-hand-side nonterminals' translations, and the values of the right-hand-side terminals.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

```
var x := 0;
while (x < 0) begin
    x := x + 1;
end
y := x * 2;
```



Syntax Directed Translation has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in the form of attributes attached to the nodes. Syntax-directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated with the non-terminals in their definitions.

A parser generator is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a parser, which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a parse tree, which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar. We'll see how a parse tree actually looks in the next section.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an abstract syntax tree (AST).

Safe from bugs. A grammar is a declarative specification for strings and streams, which can be implemented automatically by a parser generator. These specifications are often simpler, more direct, and less likely to be buggy than parsing code written by hand.

Easy to understand. A grammar captures the shape of a sequence in a form that is compact and easier to understand than hand-written parsing code.

Ready for change. A grammar can be easily edited, then run through a parser generator to regenerate the parsing code.

5. Explanation of challenges:

1. **Parser Conflicts**
   The first iteration of the grammar that was incorporated in the parser generator had a lot of shift/reduce and reduce/reduce conflicts. To debug the issue, we looked into the debug file generated by the parser generator and figured out that the following snippet of grammar was causing the conflicts.

   After analyzing the snippet, we realized that the conflicts were occurring because there were multiple ways of reaching an empty statement from the toplevelStatements which caused an ambiguity in the parser. To overcome this issue, we had to incorporate a change in the grammar that the function declarations are constrained to be defined at the top of the code to push the empty statements to the bottom of the grammar. For this we had to change the grammar to the following:

```
program                ::=  functionDeclarations statements

functionDeclarations   ::=  functionDeclaration functionDeclarations
                       |    emptyStatement
```

2. **EBNF to BNF**
   We had initially written the entire grammar in the extended BNF form. Later, while going through the SLY documentation we found that it only supports the BNF form of grammar. So, we had to manually convert the entire grammar from EBNF to BNF. Shown below is a snippet of the grammar that had to be changed (EBNF in the left half and BNF in the right)

EBNF Notation	**BNF Notation**
`params ::= IDENT (COMMA IDENT)*`	`params ::= IDENT COMMA params` `| IDENT`

3. **Variable redeclaration**
   Our current grammar does not incorporate scoping of variables. So, if a variable that was already declared globally, is redeclared inside a conditional or a loop, the value of the global variable is overwritten which is an undesirable outcome. We are working on resolving this challenge as the work on the parser progresses further.

6. Test Cases:

All .tl codes can be found in the test cases folder here, and their corresponding outputs, here.

- **Case 1**
  This test case was created solely for the purposes of checking the behavior of comments when placed along mathematical operations with similar operator design.

  Comments in tlang (our tetris language) are always single lined and start with “//”. A direct conflict can be imagined with the division operator, which uses “/” as operand.

```
var x := 2/2 - 1; // this is a comment
```

  Expected outcome for such a situation should be such that, the code up to the comment should be successfully converted to its intermediate counterpart, and from there on, where the comment is defined, the line should be ignored.

  Actual outcomes do indeed successfully replicate this, and the parser encounters no errors.

- **Case 2**

  This case focuses more on the semantic soundness of our language. Developer is provided with a wide array of functionalities, one of which is the declaration of any number of variables.

  However, as it is with all good things, they often come with a drawback. Users can re-declare their own variables with a different value, which may result in some weird results in case of loops. For example, consider this :

```
var run := True;
while(run)
begin
    var run := False;
    ...
end
...
```

Where the entire loop depends on the value of var ‘run’. The expected outcome should be one, where the code runs with no issues, i.e. the behavior of the loop is normal, and it truncates normally, as expected. This is because the var ‘run’, when re-declared, has a scope inside the loop itself, and thus has no effect on the loop.

However, as it happens, our parser generated intermediate code, somehow isn’t able to figure out the scope separation of variables, and the while loop is exited out immediately! This abnormal behavior is unexpected, and the team is working to resolve this.

- **Case 3**

Third case involves user defined function declaration, where multiple functions are involved and need clever parsing to distinguish between them.

The test code has the developer define a user defined function, which uses a library function, already defined in the game engine, and the library function in turn utilizes another library function as its argument.

```
function new_func(arg)
begin
    if(arg != 0)
        begin
        set_window_caption("Score : " + str(arg));
    end
end
```

As is apparent in the code above, there are in total 3 different functions being called here. Expected behavior is for the parser to correctly differentiate between library and user defined functions, whilst maintaining the correct order of calls, when generating the intermediate code.

And once again, the parser is able to correctly identify and arrange all the necessary functions in the resulting intermediate code.

7. A Complete end-to-end tetris game engine programming toolchain:

- If your scheme involves post-processing (like, if you generate a C-program that needs to be bundled with some standard preamble and your tetris-specific libraries and finally compiled+linked by a native C compiler further to get a binary executable) give a makefile and/or shell script to carry out all the steps seamlessly.
- The code generated by the parser is concatenated with header code, which imports the game engine and makes the code seamless.

8. Design changes from Stage 1:

Primarily there are 2 major changes from the design proposed in stage 1:

- The syntax for declaring a custom tetrimino block has been changed.
- The functionality of giving the user flexibility to decide his/her own logo for Tetris

8. Division and Distribution of Roles and Responsibilities Among the Team:

- Abhineet Karn , Abhinav Srivastava :
    - Working with the SLY tool to connect the engine and the TLang code written by the programmer
- Aryan Tyagi , Hrishikesh Govindrao Kusneniwar :
    - Working on the scanner and parser, and the *tcompiler.py* file and testing the scanner
- Hardik Nilesh Shah , Hitarth Daxeshbhai Kothari :
    - Working on the engine file i.e. implementations of parts of the tetris game in python