[PRE-FINAL] Complete design specs and syntax-directed translator design

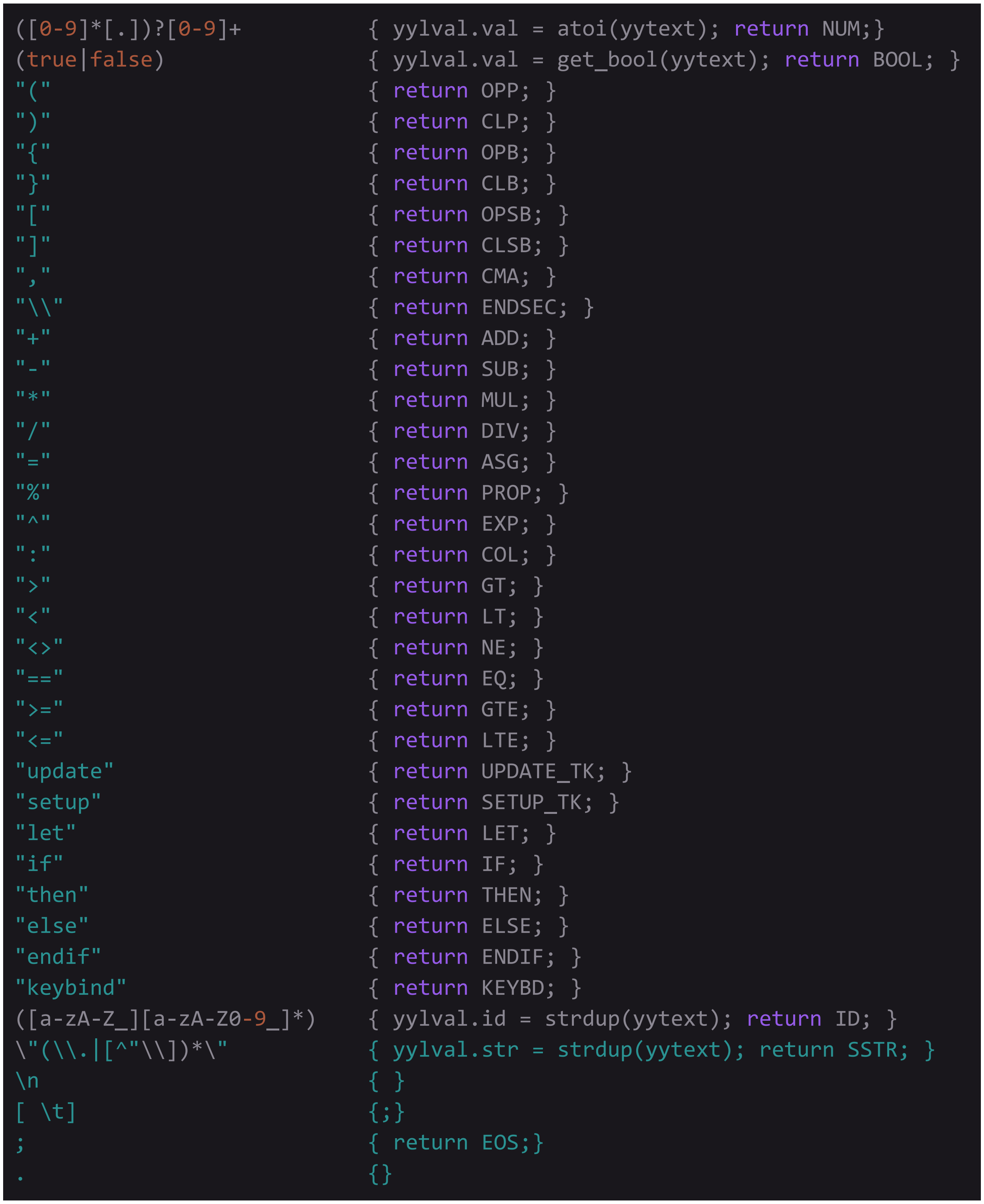
CS F363 Compiler Construction

Group 45

Asmita Limaye	2018B5A70881G
Varun Singh	2018B5A70869G
Pranav Ballaney	2018B1A70625G
Ameya Thete	2018B5A70885G

Table of Contents

Tokens for reference:



```
([0-9]*[.])?[0-9]+          { yylval.val = atoi(yytext); return NUM;}
(true|false)                { yylval.val = get_bool(yytext); return BOOL; }
"("                         { return OPP; }
")"                         { return CLP; }
"{"                         { return OPB; }
"}"                         { return CLB; }
"["                         { return OPSB; }
"]"                         { return CLSB; }
","                         { return CMA; }
"\\"                        { return ENDSEC; }
"+"                         { return ADD; }
"-"                         { return SUB; }
"*"                         { return MUL; }
"/"                         { return DIV; }
"="                         { return ASG; }
"%"                         { return PROP; }
"^"                         { return EXP; }
":"                         { return COL; }
">"                         { return GT; }
"<"                         { return LT; }
"<>"                        { return NE; }
"=="                        { return EQ; }
">="                        { return GTE; }
"<="                        { return LTE; }
"update"                    { return UPDATE_TK; }
"setup"                     { return SETUP_TK; }
"let"                       { return LET; }
"if"                        { return IF; }
"then"                      { return THEN; }
"else"                      { return ELSE; }
"endif"                     { return ENDIF; }
"keybind"                   { return KEYBD; }
([a-zA-Z_][a-zA-Z0-9_]*)    { yylval.id = strdup(yytext); return ID; }
\"(\\.|[^"\\])*\"           { yylval.str = strdup(yytext); return SSTR; }
\n                          { }
[ \t]                       {;}
;                           { return EOS;}
.                           {}
```

Syntax Directed Translation Scheme

We essentially have various rules for setup, and our setup block consists of various statements enclosed within [setup] (start_setup) and [/setup] (end_setup).
The various rules for our games fall into these categories:

1. stringrule:
   This takes the form identifier=string.
   Example: BGIMG=”blah.jpg”, to change the background image.
2. Keybind rule:
   This takes the form KEYBD string1 string2.
   Example: keybind LEFT A, to bind the A key to the action of left.
3. Generic rule:
   This takes the form identifier=numeric expression.
   Example: SINGLE_LINE_REWARD=10*level_no;
4. Array rule:
   This takes the form let identifier=array.
   Example: let block1 = [0, 0, 0, 2, 2, 2, 0, 2, 0] % 23;

We have named identifiers which we store the value of, numerical expressions which are calculated, and strings.

```
setup → start_setup setup_block end_setup {; } //no action
start_setup → OPSB SETUP_TK CLSB {;} //no action
setup_block → proper_stmt setup_block {;}
            | proper_stmt {;}

end_setup → OPSB ENDSEC SETUP_TK CLSB {;} //no action
            | OPSB DIV SETUP_TK CLSB
            {
                Print error: "The end-setup token uses a backslash."
            }

proper_stmt →
    setup_stmt EOS {;}
    |setup_stmt
        {
            Print error: "Did you forget a semicolon?"
        }

setup_stmt → stringrule {;}
 | keybindrule {;}
 | genericrule {;}
 | arrayrule{ ;}

stringrule → identifier ASG sstring
{
    Insert string into set of specifications to be sent to game code
}
keybindrule → KEYBD identifier identifier
{
    Insert keybind rule into set of specifications to be sent to game code
}

genericrule → identifier ASG expr
{
    Insert assignment rule into set of specifications to be sent to game
code
}



arrayrule→ LET identifier ASG OPSB array CLSB PROP expr
{

    Insert block definition rule into set of specifications to be sent to
game code
}

array→ NUM CMA NUM CMA NUM CMA NUM CMA NUM CMA NUM CMA NUM CMA NUM CMA NUM
{
    Store the value of the the 9 numbers into an array called the temp
array
    temp_array.val={NUM1.val, NUM2.val,...}
}

identifier→ ID { identifier.val=ID.lexval; }

sstring→
    SSTR { sstring.val=SSTR.lexval; }


/* component may have ADD/SUB operators among factors */
expr→
  factor {expr.val=factor.val ;
| expr ADD factor {expr.val = expr.val+ factor.val;}
| expr SUB factor {expr.val = expr.val - factor.val;}
| BOOL {expr.val=BOOL.val;}
;

/* factor may have MUL/DIV operators among terms */
factor⟶ term {factor.val=term.val;}
| factor MUL term {factor.val=factor.val*term.val;}
| factor DIV term {if term is non zero factor.val=factor.val/term.val else
print error "Division by zero" ;}
;

/* term comprises of subterms which may have power operator */
term⟶ subterm {term.val=subterm.val; }
| term EXP subterm {term.val= term.val^subterm.val; }
;

/* this rule supports unary plus/minus (Associativity RIGHT-TO-LEFT) */
subterm⟶ uterm {subterm.val=uterm.val;}
| SUB uterm {subterm.val= -1* uterm.val;}
| ADD uterm {subterm.val= uterm.val;}
;

/* this rule supports brackets in the grammar */
uterm⟶ NUM {uterm.val=NUM.lexval;}
| stored_identifier {uterm.val=stored_identifier.val;}
| OPP expr CLP {uterm.val=expr.val;}}
;

stored_identifier⟶
    ID {
        Finds ID.val from the symbol table
        stored_identifier.val=ID.lexval;
    }
```

Our LR automaton is quite complicated, we had to generate one using Bison.
Here is a link to the generated graph (you can zoom and pan).

Challenges:

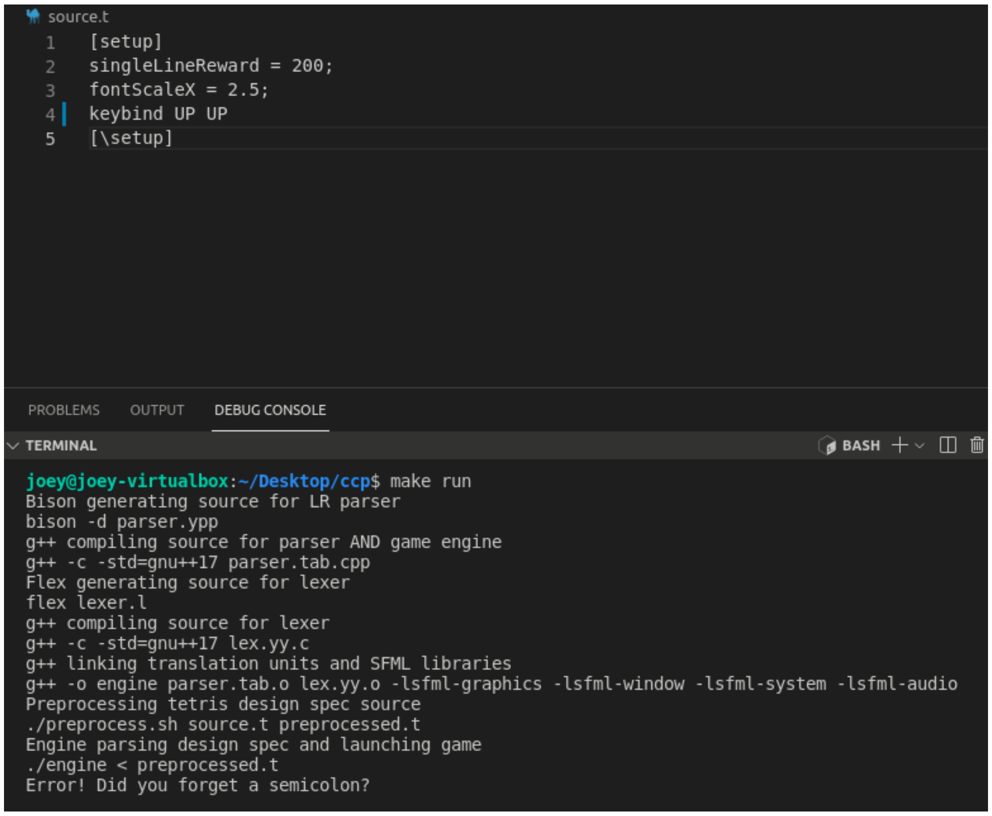
Error productions and rules:
We have two error productions.

```
end_setup → OPSB ENDSEC SETUP_TK CLSB {;} //no action
            | OPSB DIV SETUP_TK CLSB
            {
                Print error: "The end-setup token uses a backslash."
            }

proper_stmt →
    setup_stmt EOS {;}
    |setup_stmt
        {
            Print error: "Did you forget a semicolon?"
        }
```

The first one corresponds to the end of the setup block and occurs when the user writes [/setup] instead of [\setup].

The second one occurs when statements in the setup block do not end with semicolons.



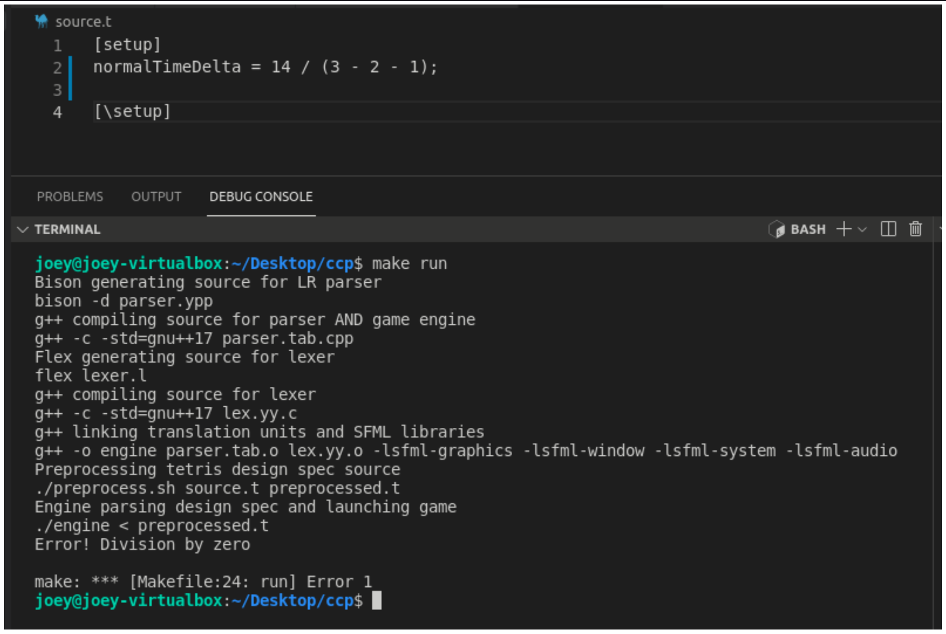
source.t

```
[setup]
singleLineReward = 200;
fontScaleX = 2.5;
keybind UP UP
[\setup]
```

```
joey@joey-virtualbox:~/Desktop/ccp$ make run
Bison generating source for LR parser
bison -d parser.ypp
g++ compiling source for parser AND game engine
g++ -c -std=gnu++17 parser.tab.cpp
Flex generating source for lexer
flex lexer.l
g++ compiling source for lexer
g++ -c -std=gnu++17 lex.yy.c
g++ linking translation units and SFML libraries
g++ -o engine parser.tab.o lex.yy.o -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
Preprocessing tetris design spec source
./preprocess.sh source.t preprocessed.t
Engine parsing design spec and launching game
./engine < preprocessed.t
Error! Did you forget a semicolon?
```

We also detect the division by zero error while resolving numerical expressions:

```
factor→ term {factor.val=term.val;}
| factor MUL term {factor.val=factor.val*term.val;}
| factor DIV term {if term is non zero factor.val=factor.val/term.val else
print error "Division by zero" ;}
```



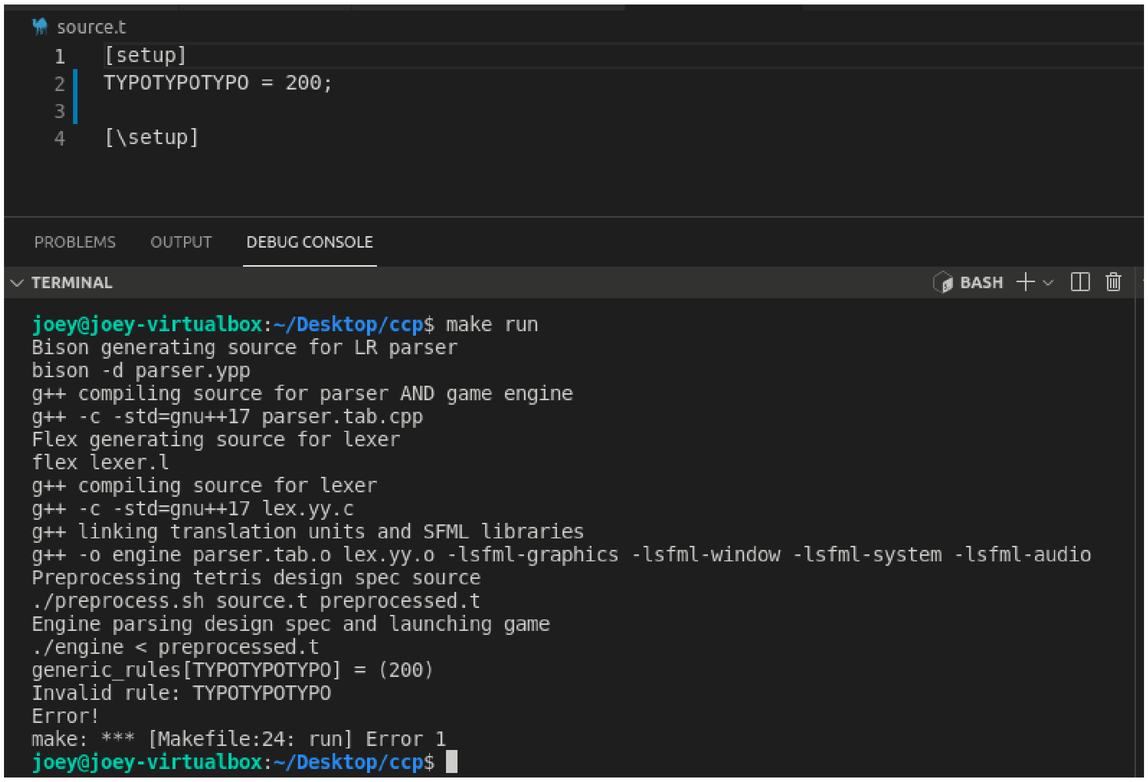
```
[setup]
normalTimeDelta = 14 / (3 - 2 - 1);

[\setup]
```

```
joey@joey-virtualbox:~/Desktop/ccp$ make run
Bison generating source for LR parser
bison -d parser.ypp
g++ compiling source for parser AND game engine
g++ -c -std=gnu++17 parser.tab.cpp
Flex generating source for lexer
flex lexer.l
g++ compiling source for lexer
g++ -c -std=gnu++17 lex.yy.c
g++ linking translation units and SFML libraries
g++ -o engine parser.tab.o lex.yy.o -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
Preprocessing tetris design spec source
./preprocess.sh source.t preprocessed.t
Engine parsing design spec and launching game
./engine < preprocessed.t
Error! Division by zero

make: *** [Makefile:24: run] Error 1
joey@joey-virtualbox:~/Desktop/ccp$
```

We detect undeclared symbols / typo'd configurable variable names.



```
[setup]
TYPOTYPOTYPO = 200;

[\setup]
```

```
joey@joey-virtualbox:~/Desktop/ccp$ make run
Bison generating source for LR parser
bison -d parser.ypp
g++ compiling source for parser AND game engine
g++ -c -std=gnu++17 parser.tab.cpp
Flex generating source for lexer
flex lexer.l
g++ compiling source for lexer
g++ -c -std=gnu++17 lex.yy.c
g++ linking translation units and SFML libraries
g++ -o engine parser.tab.o lex.yy.o -lsfml-graphics -lsfml-window -lsfml-system -lsfml-audio
Preprocessing tetris design spec source
./preprocess.sh source.t preprocessed.t
Engine parsing design spec and launching game
./engine < preprocessed.t
generic_rules[TYPOTYPOTYPO] = (200)
Invalid rule: TYPOTYPOTYPO
Error!
make: *** [Makefile:24: run] Error 1
joey@joey-virtualbox:~/Desktop/ccp$
```

Parser conflicts:
We kept calling bison with various debug flags enabled and we kept tweaking our context free grammar until the output did not report any errors. The debug flags enabled us to observe the state of the parser stack at various points in the code and thus helped us figure out exactly where we were going wrong. For example, a mistake we made was adding new rules (like the keybind rule) and forgetting to include these rules in our setup statements. So this enabled us to avoid all parser conflicts.
Challenges:
We had to write our parser and lexer in C++ instead of using C which is standardly used in lex and bison.
We also had to interface with our game code directly in our production action pairs.

Tests

The following test cases illustrate the variety of games that can be produced using our game engine.

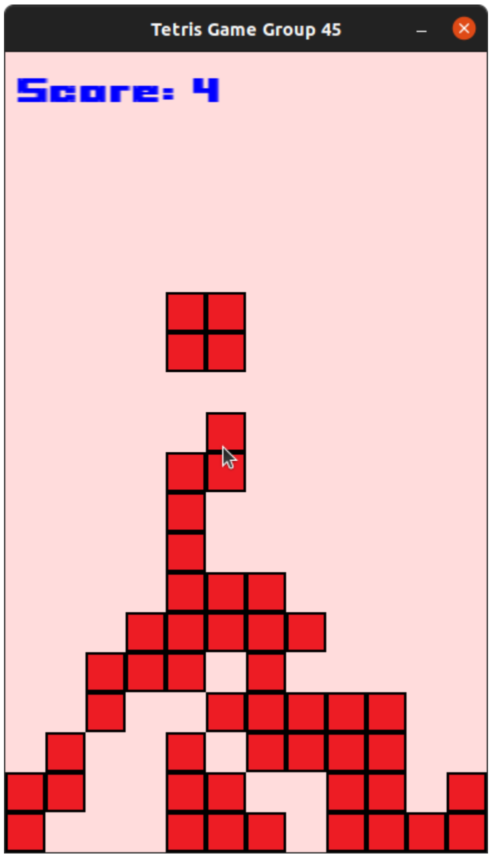
The first test case illustrates changes in the scoring system, fonts, and keybindings.

```
[setup]
gameWindowTitle = "Tetris Game Group 45";
```

```
singleLineReward = 2;
fontScaleX = 2.5;

keybind UP UP;
keybind DOWN DOWN;
keybind LEFT LEFT;
keybind RIGHT RIGHT;

[\setup]
```



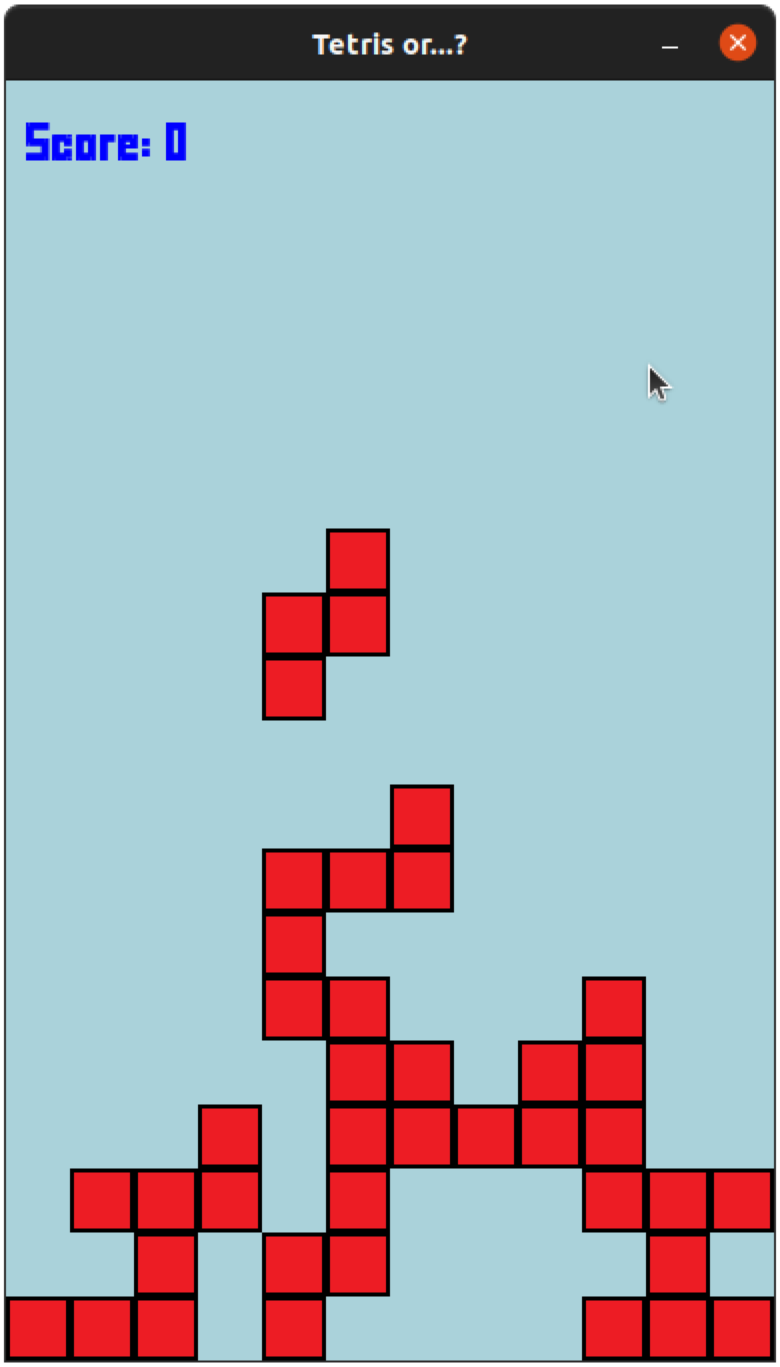
The second test case illustrates changing the background music, keybindings and background color.

```
[setup]
#this is a comment
gameWindowTitle = "Tetris or...?";
musicPath = "sounds/mario_theme.wav";

keybind UP W;
keybind DOWN S;
keybind LEFT A;
keybind RIGHT D;

backgroundColorR = 170;
backgroundColorG = 210;
backgroundColorB = 218;

[\setup]
```



The third test case illustrates mathematical expressions, changing the tile image sprite and changing the game board size.

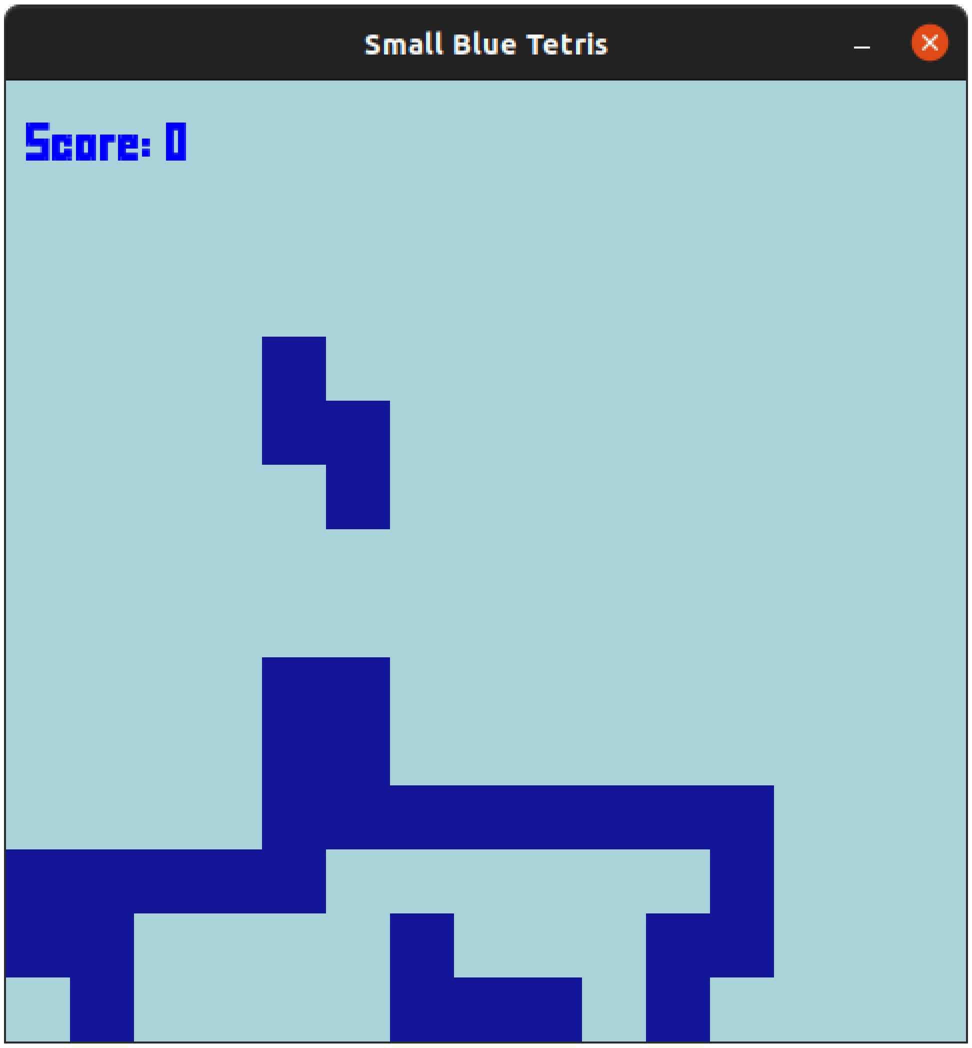
```
[setup]

gameWindowTitle = "Small Blue Tetris";
tileSpriteImgPath = "images/tetris_tile_blue.png";
```

```
numTilesX = 15;
numTilesY = numTilesX * 1.5;

backgroundColorR = 170;
backgroundColorG = 210;
backgroundColorB = 218;

[\setup]
```



We allow the designer to:

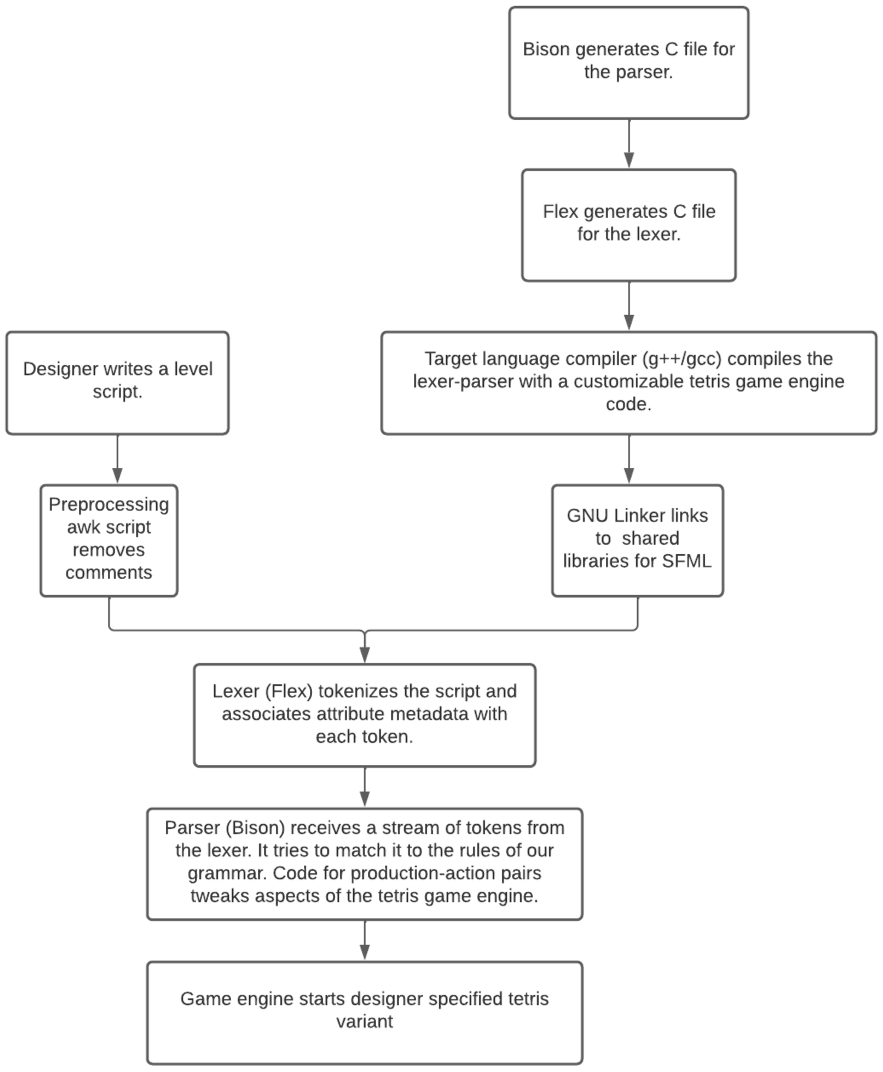
- customize rewards for clearing 1, 2, 3, or 4 lines with a piece
- customize window size / title / framerate
- customize tiling (default 10 x 12 Tetris board)
- customize controls for movement, rotation, and soft drop
- customize fonts for display text
- customize sound effects and background music
- customize fall speed, soft drop speed

Complete end-to-end Tetris game engine programming toolchain

Overview

As can be surmised from the **flowchart** and **makefile**: we compile our lexer, parser, and game engine into a single executable program. This program is a lexer-parser whose pattern-action and production-action pairs can directly interface with game engine code. It takes a preprocessed 'tetris design script' as input and launches the specified variant of the game.

Flowchart


Bison generates C file for the parser.
Flex generates C file for the lexer.
Target language compiler (g++/gcc) compiles the lexer-parser with a customizable tetris game engine code.
GNU Linker links to shared libraries for SFML
Designer writes a level script.
Preprocessing awk script removes comments
Lexer (Flex) tokenizes the script and associates attribute metadata with each token.
Parser (Bison) receives a stream of tokens from the lexer. It tries to match it to the rules of our grammar. Code for production-action pairs tweaks aspects of the tetris game engine.
Game engine starts designer specified tetris variant

Makefile

```
dependencies:
      sudo apt-get install libsfml-dev

lexer: lexer.l
         @echo "Flex generating source for lexer"
         flex lexer.l
         @echo "g++ compiling source for lexer"
         g++ -c -std=gnu++17 lex.yy.c

parser: parser.ypp game.cpp
            @echo "Bison generating source for LR parser"
            bison -d parser.ypp
            @echo "g++ compiling source for parser AND game engine"
            g++ -c -std=gnu++17 parser.tab.cpp

engine: parser lexer
        @echo "g++ linking translation units and SFML libraries"
        g++ -o engine parser.tab.o lex.yy.o -lsfml-graphics
-lsfml-window -lsfml-system -lsfml-audio

run: engine
       @echo "Preprocessing tetris design spec source"
       ./preprocess.sh source.t preprocessed.t
       @echo "Engine parsing design spec and launching game"
       ./engine < preprocessed.t


clean:
      rm -f *.o *.out lex.yy.c parser.tab.cpp parser.tab.hpp
preprocessed.t engine
```

[MIDSEM] Top-level Design Specifications and Scanner Design

CS F363 Compiler Construction

Group 45

Asmita Limaye	2018B5A70881G
Varun Singh	2018B5A70869G
Pranav Ballaney	2018B1A70625G
Ameya Thete	2018B5A70885G

Table of Contents

Top-level Design Specs

Overall Program Structure

We envision our Tetris programming language to be an imperative programming language with support for conditional control flow statements. As such, the design is heavily motivated by the requirements of the game and many of the elements of the game, such as block rendering and simulation or user interfacing, are abstracted away from the programmer to the game engine. The programmer only specifies what should occur, and not necessarily *how* it should happen — so in that sense, our language has elements of declarative programming languages.

Our structure therefore involves three major elements that work in tandem to create a final compiler: the lexer, the parser and the game engine. While many details about the parser and the game engine have yet to be ironed out, the parser and lexer are compiled together with the game engine's source code. Programmable aspects are customised as the designer's input script is parsed, for each level. There is no source-to-source translation and machine code generation is handled by the target language compiler.

We write our lexer in Flex and our parser in GNU Bison. The game engine and all underlying programming support is provided in the C programming language. Finally, to render the graphics for the game, we plan to use the CSFML (A C distribution of the Simple and Fast Multimedia Library). The programmer would generate a script in our language which would be passed to the compiler (lexer-parser-engine) pipeline, which in turn would generate an executable that can be played. Further details can be found in the pipeline schema section.

Primitives

Language primitives are the simplest possible elements available in a programming language. We envision our Tetris programming language to offer support for 7 major primitive types: a user-programmable tetromino; special reserved constants which we call engine constants; two major sections in the code called the setup section and the update section; support for tetromino arithmetic as well as arithmetic on numeric literals; user-defined variables, and finally, control-flow structures.

While we believe that this list of primitives is exhaustive, further development of the parser and the game engine might require us to reconsider some of these decisions, ultimately motivating the addition or removal of these features. The next few sections go into these primitives in detail.

Custom Tetromino

Apart from those specified in the Tetris game specifications, we allow the user to create their own tetromino blocks. As a result, our Tetris programming language incorporates support for creating tetrominos using MATLAB-style fixed-sized arrays. While granting the programmer more freedom over the creation of these blocks, this formulation also allows us to perform something called tetromino arithmetic. As an initial step, we limit ourselves to 3 x 3 tetrominos, hence our arrays are 9-element arrays. Each element in the array denotes whether the particular pixel in the 3 x 3 block is filled or not, with the value of the non-zero element specifying the colour. We allow the elements to lie in the range [0, 7], with zero indicating no fill and 1 indicating the colour violet, 2 indicating the colour indigo, 3 indicating blue, and so on, all the way until 7, which indicates red.

Consider this example: suppose we want to design a T-shaped tetromino that is coloured yellow. In the Tetris-programming language, the programmer would create the tetromino as a variable using the let keyword (more on variables follows).

```
let block1 = [
        0, 0, 0,
        5, 5, 5,
        0, 5, 0
];
```

Engine Constants

We allow the programmer to specify certain special state variables, which we call Engine constants, as these constants will later be used by the game engine to control the flow of the game. Some examples of these engine constants are block fall speed, the background music, the background image, the allowed key bindings for the game, and so on. Because there are various possible state variables, we plan to standardize the list of engine constants available to the programmer as we progress through the project. For any unspecified constants, the engine will revert to a default value. These are represented by capital letters in the code. For example, to set a key binding for left motion of a block to the key ‘A’, the programmer would write

```
keybind LEFT A;
```

“Keybind” is a reserved keyword and “LEFT” is an engine constant. In another example, to set the initial falling speed of a block to 0.5, the programmer uses the engine constant FALLING_SPEED.

```
FALLING_SPEED=0.2;
```

Code Sections

Our language requires the programmer to segment the code into two major sections: the setup section and the update section. These sections appear like HTML tags and are represented as [section]...[\section]. Within the setup section, the programmer would initalize the various engine constants and tetrominoes, and within the update section, they would write statements that have some form of temporal dependencies. All the statements in the update block would be executed repeatedly by the game engine according to the refresh rate of the game. Statements within a section must be terminated by a semicolon (;).

Our language also supports single line comments, which are preceded by a pound or hash symbol (#). This works exactly like comments in Python.

Tetromino Arithmetic

The Tetrominoes in our programming language are created by specifying MATLAB-style 9-integer arrays. Since tetrominoes are just numerical arrays, they can be added and subtracted as usual. This allows the programmer to build new tetromino from existing ones. The plus (+) operator performs a union while the minus (-) operator performs set difference. An example of such arithmetic follows.

```
let block1 = [
        0, 0, 0,
        5, 5, 5,
        0, 5, 0
];

let block2 = [
        5, 5, 5,
        0, 0, 0,
        0, 0, 0
];

let block3 = block1 + block2;
```

Block 3 would now be:

```
block3 = [
        5, 5, 5,
        5, 5, 5,
        0, 5, 0
];
```

Arithmetic

Arithmetic operators (+, -, * and /) can be used with numerical parameters such as fall speed and position with numbers. For example, this could be used to increase the fall speed by 5% at regular intervals.

```
FALLING_SPEED = FALLING_SPEED + (FALLING_SPEED * 0.05);
```

Conditionals

Our programming language supports conditional expressions according to the following syntax:

```
if [EXPR1] then [EXPR2] endif;
```

For example,

```
if SOFT_DROP_ON == true then SOFT_DROP_REWARD_MULTIPLIER = 1 endif;
```

Variables

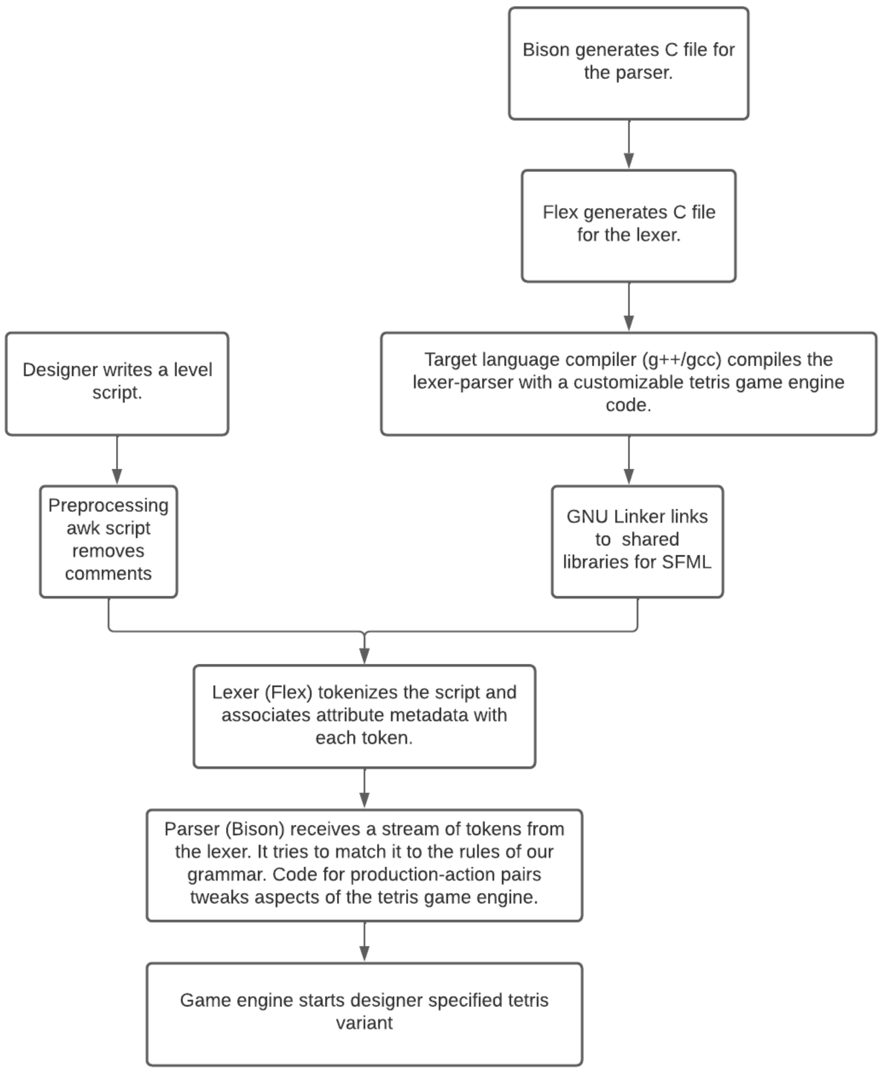
Our programming language supports variable definitions according to the following syntax:

```
let level_no = 1;
```

We provide support for integers, floating point numbers, booleans and strings. Our language is strongly and statically-typed with in-built type inference.

Pipeline Schema

The parser and lexer are compiled with the game engine's code. Programmable aspects are customised as the designer's input script is parsed, for each level. There is no source-to-source translation and machine code generation is handled by the target language compiler (gcc). The following flowchart specified the envisioned pipeline for the compiler.



Bison generates C file for the parser.
Flex generates C file for the lexer.
Designer writes a level script.
Target language compiler (g++/gcc) compiles the lexer-parser with a customizable tetris game engine code.
Preprocessing awk script removes comments
GNU Linker links to shared libraries for SFML
Lexer (Flex) tokenizes the script and associates attribute metadata with each token.
Parser (Bison) receives a stream of tokens from the lexer. It tries to match it to the rules of our grammar. Code for production-action pairs tweaks aspects of the tetris game engine.
Game engine starts designer specified tetris variant

Scanner Design

Pattern-action pairs in the lexer can directly interface with the Tetris engine code (that is why it is written in C/C++ and compiled with the parser and lexer). The lexer, especially for numeric literals, generates the same token for multiple possible lexemes. To tackle this, we use the union data structure supported by Flex to allow the incorporation of an attribute to disambiguate the various lexemes. In our case, we support integer, float, boolean and string data types; as a result, the union would contain three fields which store the appropriate object and the attribute field in the returned token would then contain the union which can later be accessed directly by the parser.

We have a finite lexicon for the language and have around 30 tokens that can be generated on parsing a source code. As a tokenization demo, we have a bare-bones implementation of our parser on the GitLab repository with instructions on how to run a tokenization. While specific patterns are more or less final at this stage, we might make some optimizations as we begin to develop the parser. Actions corresponding to a matched lexeme can only be specified once we have a clear understanding of what the parser would look like, and hence we defer that discussion to a slightly later date.

(Tentative) List of tokens

Token	**Attributes**	**Description**
NUM	Floating point number	For floating point numbers
BOOL	Boolean	For boolean values
OPP	‘(‘	Opening parenthesis
CLP	‘)’	Closing parenthesis
OPB	‘{’	Opening curly brace
CLB	‘}’	Closing curly brace
OPSB	‘[‘	Opening square bracket
CLSB	‘]’	Closing square bracket
CMA	‘,’	Comma
ENDSEC	‘\’	Denotes the end of a section
ADD	‘+’	Addition operator
SUB	‘-’	Subtraction operator
MUL	‘*’	Multiplication operator
DIV	‘/’	Division operator
ASG	‘=’	Assignment operator

EXP	‘^’	Exponent operator
COL	‘:’	Colon symbol
GT	‘>’	Greater-than operator
LT	‘<’	Less-than operator
NE	‘<>’	Inequality operator
EQ	‘==’	Equality operator
GTE	‘>=’	Greater than or equal to
LTE	‘<=’	Less than or equal to
UPDATE_TK	update	Update token. Denotes beginning of the update section.
SETUP_TK	setup	Setup token. Denotes beginning of the setup section.
LET	let	Preceeds variable assignments
IF	if	A part of conditional statements
THEN	then	A part of conditional statements
ELSE	else	A part of conditional statements
ENDIF	endif	A part of conditional statements
KEYBD	keybind	Preceeds a key binding
ID	Character string	For names of variables
STR	Character string	For string literals used for display messages
EOS	‘;’	End of statement

An example of a designer’s script written in our language and tokenization demo are included in the appendix.

Division of Labour between the Scanner and the Parser

Currently, the division of labour between the parser and lexer is highly asymmetrical. All the lexer does is generate and pass a stream of tokens to the parser — it does not disambiguate the lexeme from the token in cases where a single token might match multiple lexemes, for example numeric literals.

The parser receives the stream of tokens from the lexer and tries to match it to the rules of our grammar. The code for the production-action pairs in the parser will do the heavy-lifting of customising aspects of a tetris game; different snippets are invoked when different rules are matched.

The designer’s input script dictates how the programmable aspects of the tetris engine are compiled during the parse phase.

Additional Information

In our pipeline, we also contain a preprocessing phase which would mostly involve removing comments, whitespaces and empty lines from the source code. Currently, we have a shell script that does the preprocessing for us (the script can be found on the GitLab repository, inside the lexer directory), but we plan to migrate it to an awk script later as the current script is awkwardly written. Moreover, we want to solve the problem of array declarations spanning multiple lines which need to be handled by the preprocessor such that they can be tokenized easily by the lexer.

Division and Distribution of Roles and Responsibilities Among the Team

Ameya Thete	Lexer, Parser, Testing
Asmita Limaye	Lexer, Game engine, Documentation
Pranav Ballaney	Parser, Testing, Documentation
Varun Singh	Parser, Game engine

Appendix

Lexer code

```
%{
#include <stdbool.h>
extern int yyerror(char* s);
%}
%%
([0-9]*[.])?[0-9]+          { printf("NUM %s\n", yytext);}
(true|false)                { printf("BOOL %s\n", yytext);}
"("                         { printf("OPP\n");}
")"                         { printf("CLP\n");}
"{"                         { printf("OPB\n");}
"}"                         { printf("CLB\n");}
"["                         { printf("OPSB\n");}
"]"                         { printf("CLSB\n");}
","                         { printf("CMA\n");}
"\\"                        { printf("ENDSEC\n");}
"+"                         { printf("ADD\n");}
"-"                         { printf("SUB\n");}
"*"                         { printf("MUL\n");}
"/"                         { printf("DIV\n");}
"="                         { printf("ASG\n");}
"^"                         { printf("EXP\n");}
":"                         { printf("COL\n");}
">"                         { printf("GT\n", yytext);}
"<"                         { printf("LT\n", yytext);}
"<>"                        { printf("NE\n", yytext);}
"=="                        { printf("EQ\n", yytext);}
">="                        { printf("GTE\n", yytext);}
"<="                        { printf("LTE\n", yytext);}
"update"                    { printf("UPDATE_TK\n");}
"setup"                     { printf("SETUP_TK\n");}
"let"                       { printf("LET\n");}
"if"                        { printf("IF\n");}
"then"                      { printf("THEN\n");}
"else"                      { printf("ELSE\n");}
"endif"                     { printf("ENDIF\n");}
"keybind"                   { printf("KEYBD\n");}
([a-zA-Z_][a-zA-Z0-9_]*)    { printf("ID %s\n", yytext);}
\"(\\.|[^"\\])*\"           { printf("STR %s\n", yytext);}
\n                          { }
[ \t]                       {;}
;                           { printf("EOS\n", yytext);}
.                           { printf("Unknown symbol\n", yytext);}
%%
```

Sample script

```
# Group G45
# Date: 2021-03-06
# This sample is a tentative illustration of a program written in the
Tetris programming
# language. The program consists of two major section: a setup section,
which sets up the program
# state and environment; and an update section, which lists statements
that are executed every time the
# game is refreshed. Ideally, the setup section consists of declarative
statements creating the various
# tetrominos, assigns key bindings for the level/program, as well as
initialization of state variables we
# call "engine constants". The update section consists of imperative
statements that alter the state of the
# program by modifying these constants, either via conditional control
flow or plain statements. Within the
# setup block, the programmer is able to create variables to add
functionality not already provided by these
# engine constants.

[setup]

# Level number
let level_no = 1;

# Key binding for user input
keybind LEFT A;
keybind RIGHT D;
keybind CLOCKWISE P;
keybind ANTICLOCKWISE O;
keybind SOFT_DROP S;
keybind HARD_DROP W;

# Customise rewards
SINGLE_LINE_REWARD=10*level_no;
DOUBLE_LINE_REWARD=20*level_no;
TRIPLE_LINE_REWARD=30*level_no;
TETRIS_LINE_REWARD=40*level_no;

# Boolean values to enable/disable soft and hard drop
SOFT_DROP_ON=true;
HARD_DROP_ON=true;

if SOFT_DROP_ON == true then SOFT_DROP_REWARD_MULTIPLIER = 1 endif; #
1*n, where n is the number of lines the tetromino is soft dropped
if HARD_DROP_ON == true then HARD_DROP_REWARD_MULTIPLIER = 2 endif; #
```

```
2*n, where n is the number of lines the tetromino is hard dropped

# End the game when 15 lines have been cleared, game doesn’t end if a
number of lines
# have been cleared if this variable hasn’t been set
LINES_TO_CLEAR=15;
# End game when a set time in seconds is over, the game has no timer if
this variable hasn’t been set.
TIMER=50;

# Restrict program to 3x3 tetrominos, can extend later.
# V I B G Y O R
# 1 2 3 4 5 6 7
let block1 = [
        0, 0, 0,
        2, 2, 2,
        0, 2, 0
];

let block2 = [
        0, 1, 0,
        1, 1, 1,
        1, 1, 1
];

let block3 = block1 * 3;
let block4 = block2 + block3;

NUM_TILES_X = 5;
NUM_TILES_Y = 20;

# Sequence of falling tetrominos follows a distribution based on these
values
BAG = {block1:23, block2:26, block3:36, block4:72};
# Set positions of the blocks to follow a distribution proportional to
these values for each block
POS = {1,3,4,1,1,2,2,1,1,5,6,7,3,4,5,6};

# Set a relative path to fetch background music and image
BGMUS="blah.mp3";
BGIMG="blah.jpg";

# The initial falling speed for the level
FALLING_SPEED=0.2;

[\setup]
```

```
# everything in the update block gets executed every frame
[update]

TIMER = TIMER - 1;
FALLING_SPEED = FALLING_SPEED + TIMER/10;
if SCORE == 10 then FALLING_SPEED = 100 endif;

[\update]
```

Tokenization demo

```
paav@paav-virtualbox:~/Desktop/cc$ make run
chmod 777 preprocess.sh
./preprocess.sh sample.t preproc.t
flex lexer.l
gcc lex.yy.c -lfl
Autorunning with make.

OPSB
SETUP_TK
CLSB
EOL
LET
ID level_no
ASG
NUM 1
EOS
EOL
KEYBD
ID LEFT
ID A
EOS
EOL
KEYBD
ID RIGHT
ID D
EOS
EOL
KEYBD
ID CLOCKWISE
ID P
EOS
EOL
...
(output truncated)
```