BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE -
PILANI, GOA CAMPUS

A

Report on

Top Level Design Specification & Scanner and Parser Design : Tetris

In partial fulfillment of the course COMPILER CONSTRUCTION

Prepared for :

Dr. Ramprasad S Joshi

INSTRUCTOR IN CHARGE

COMPILER CONSTRUCTION

Prepared By:-

ABHIJEET SWAIN	2018B4A70540G
ATISHAY JAIN	2018B5A70908G
AMAN JAIN	2018B5A70868G
DHAIRY AGRAWAL	2018B4A70827G
SHUBHANKAR RAVINDRA KATE	2018B4A70786G

Group 34

March 7,2022

TABLE OF CONTENTS

Top Level Design

Programming Language

Overall Program Structure:

Source Language: CClang

Target Language: Python

Implementation Language: Python

Compilation: Static

Interpreted Style Language: Interpreted

Reasons for Choosing Python:

- Python is a very easy-to-use language. Although C/C++ is preferred for compiler construction, due to the Visual / GUI component of tetris, we decided to use Python as our language of choice.
- Python also provides a very powerful library SLY - Sly Lex Yacc (or some variations of it like PLY) which is a straightforward implementation of lex and yacc parsing tools for Python. It is implemented entirely in Python and it provides most of the standard Lex / Yacc features including support for various grammars. It especially provides extensive error checking while being easy to use.
- In short, we have various functionalities that are being provided in Python itself, along with having a good toolset for GUI support. Plus we have a lot of online support to help us to handle errors that we may come across.

Features

Primitive:

We will try to make the language as intuitive as possible for the programmer, we will be providing the following features of Tetris using our language:

i) Grid Size: The programmer can choose the grid size of the game. The grid size can lie between (4x4) to (32x32).

ii) Game Mode: The programmer can choose which game mode he/she wants to construct. We plan to offer 2 modes:

a) *Classic Mode*: In this mode, the player plays the game for infinite time until the top row of the grid gets filled.
b) *Level Mode*: In this, the programmer can set a time limit and an incremental value. If the gamer survives for the given time, he/she can advance to the next level where the speed will increase based upon the incremental value set by the programmer.

iii) Custom Grid: The programmer can choose to either start the game with a blank grid or with an initial custom grid having some grid points occupied beforehand.

iv) Orientation: The programmer can choose the orientation of the game as horizontal or vertical.

v) Shapes: The programmer can choose the shapes to include from the list of predefined shapes(S-shape, Z-shape, T-shape, L-shape, Line-shape, MirroredL-shape, Square-shape) or he/she can make his/her own custom shapes.

vi) Horizon: The programmer can choose the number of pieces to show to the gamer which are next in the queue.

vii) Fall speed: The programmer can choose the speed at which the pieces fall in the grid. For classic mode, this speed will be constant throughout the game and for the level mode this is the speed at the first level of the game.

viii) Score Display: The programmer can select if he/she wants to display the score to the gamer or not.

ix) Configurational keys:

a) Move the piece in left direction
b) Move the piece in the right direction
c) Rotate the piece in the clockwise direction
d) Rotate the piece in the anti clockwise direction

e) Hard Drop (To increase the speed of dropping.

Programmable:

Our Language Code is majorly divided into five parts for better readability and efficient implementation:

1. BOARD
2. CONTROLS
3. PIECE
4. ACTION
5. FUNCTION

In the **BOARD** section: Two integer values are to be passed which defined the board size of our game.

In the **CONTROLS** section: Keyboard controls assigned using config_left, config_right fields.

In the **PIECES** section: The tetris pieces (including the custom pieces) which are allowed in the game should be passed.

In the **ACTION** section: The programmer can declare datatypes and call functions (which are written in the FUNCTION section).

In the **FUNCTION** section: Functions are defined here.

Syntactic Structure of our Language:

```
BOARD: 10 10   #! Board of size 10 x 10

CONTROLS:
config_left = LEFT_ARROW

PIECE: @s, @z

ACTION:
mygame()

FUNCTION:
def mygame(){
  INIT_BOARD = [[0, 1, 0, 0],
                [1, 1, 0, 1]]
  set_speed = 7
}
```

CCLang (our language) allows the use of functions, loops and conditional statements.

Piece Data Type:

We define a special data type for pieces which are passed in the PIECE section. The instances of the piece data type must start with '@' character and store a 4x4 Array (representing the pattern corresponding to that piece).
For example:

```
@z = [[ 0, 0, 0, 0],
      [ 0, 0 , 0, 0],
      [ 1, 1, 0, 0],
      [ 0, 1, 1, 0]]
```

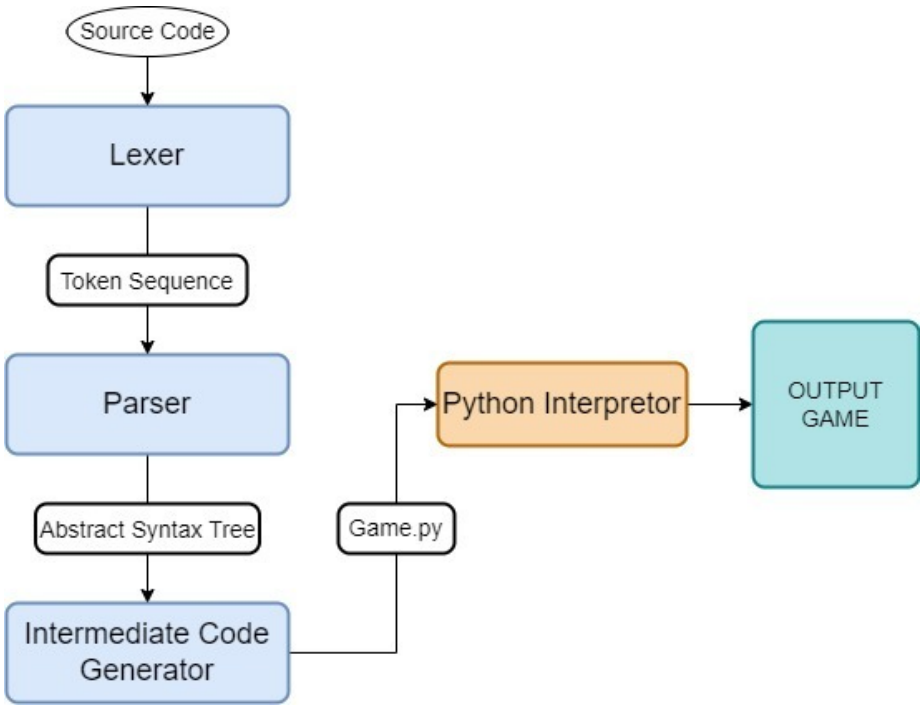
@z, @l, @j, etc., will be predefined pieces in our tetris engine. The user can create custom pieces by declaring an instance with a 4x4 2D array in the PIECES section.

Paradigm:

Imperative paradigm:-

We have chosen an imperative paradigm as it works by changing the program state through assignment statements. We want our language to perform step by step tasks by changing state. The main focus is on how to achieve the goal. The paradigm consists of several statements and after execution all the result is stored.

Pipeline Schema:



Scanner Design

Pattern-Action Pair

For our design, Python Library SLY is used. It is based on traditionally used tools lex and yacc and it implements the same parsing algorithm. SLY provides two separate classes Lexer and Parser. The Lexer class is used to split the input into a collection of tokens specified by a collection of regular expression rules.The Parser class is used to recognize language syntax that has been specified in the form of a context free grammar. The two classes are typically used together to make a parser. However, this is not a strict requirement–there is a great deal of flexibility allowed.

The token set, as seen below, is majorly divided into Keywords, Flow Control, Game Control, Identifiers, etc. The following table shows the token type assigned and the action taken for a pattern.

	Tokens	**Pattern (Regex)**	**Action**
KEYWORDS	BOARD	BOARD	return BOARD
	PIECE	PIECE	return PIECE
	CONTROLS	CONTROLS	return CONTROLS
	ACTION	ACTION	return ACTION
	FUNCTION	FUNCTION	return FUNCTION
	ON	ON	return ON
	OFF	OFF	return OFF
	DEF	def	return DEF
FLOW CONTROL	FOR	for	return FOR
	WHILE	while	return WHILE
	IF	if	return IF
	ELSE	else	return ELSE
	BREAK	break	return BREAK
	CONTINUE	continue	return CONTINUE
	RETURN	return	return RETURN
GAME CONTROL	GAME_MODE	game_mode	return GAME_MODE

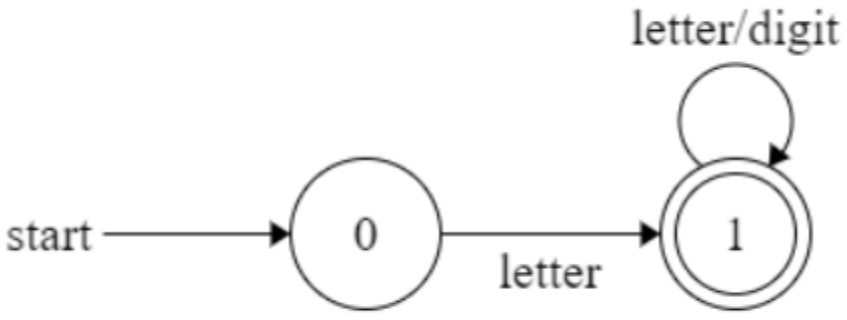
	KEY_CONTROL	config_(left | right | rot(c | ac))	return KEY_CONTROL
	SPEED_CONTROL	set_speed	return SPEED_CONTROL
	ORIENTATION	game_orientation	return ORIENTATION
	HORIZON	horizon	return HORIZON
	SCORE	score	return SCORE
	INIT_BOARD	INIT_BOARD	return INIT_BOARD
LITERALS	letter	[a-zA-Z]	return letter
	digit	[0-9]	yyval = atoi(yytext) return DIGIT
	NUMBER	digit(digit)*	yyval = atoi(yytext) return NUMBER
	STRING	"((letter)*(digit)*)*"	return STRING
	KEY_CONSTANTS	UP_ARROW | DOWN_ARROW | LEFT_ARROW | RIGHT_ARROW	return KEY_CONSTANTS
IDENTIFIER	ID	letter(letter|digit)*	return ID
	PIECE_ID	@letter(letter|digit)*	yyval = strToArr(yytext) return PIECE_ID (where yyval = Attribute Value associated with the token yytext = Lexeme matched strToArr() is a function defined in Lexer which assigns a 4x4 array corresponding to the piece)
OPERATORS	EQUAL	=	return EQUAL
	COMPARISON	<=? | >=? | ==	return COMPARISON
	PLUS	+	return PLUS
	MINUS	-	return MINUS
	DIVIDE	/	return DIVIDE
	MULTIPLY	*	return MULTIPLY
	MOD	%	return MOD

	AND	&&	return AND
	OR	||	return OR
	NOT	!	return NOT
PARENTHESIS	LEFT_PAR	(	return LEFT_PAR
	RIGHT_PAR	)	return RIGHT_PAR
	LEFT_CPAR	{	return LEFT_CPAR
	RIGHT_CPAR	}	return RIGHT_CPAR
WHITESPACE	WHITESPACE	\t' | ' '	return WHITESPACE
	NEWLINE	\n'	return NEWLINE
COMMENTS	COMMENTS	#!	return COMMENTS
PUNCTUATIONS	COMMA	,	return COMMA
	COLON	:	return COLON

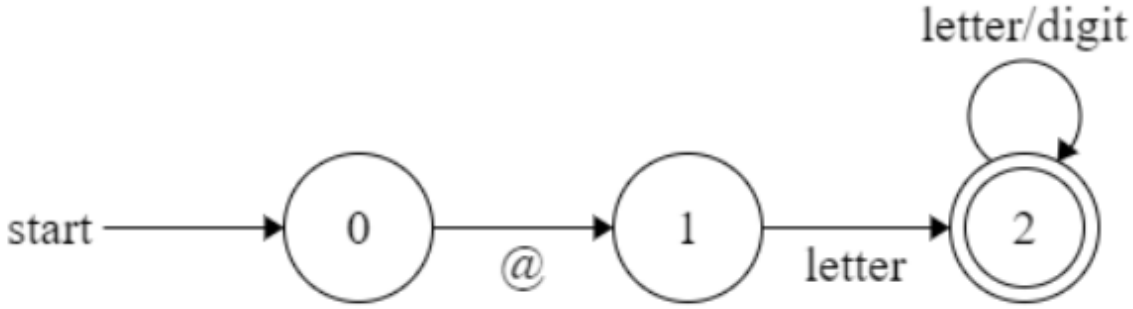
Transition Diagrams

Some of the non-trivial transition diagrams for the Lexer are given below:

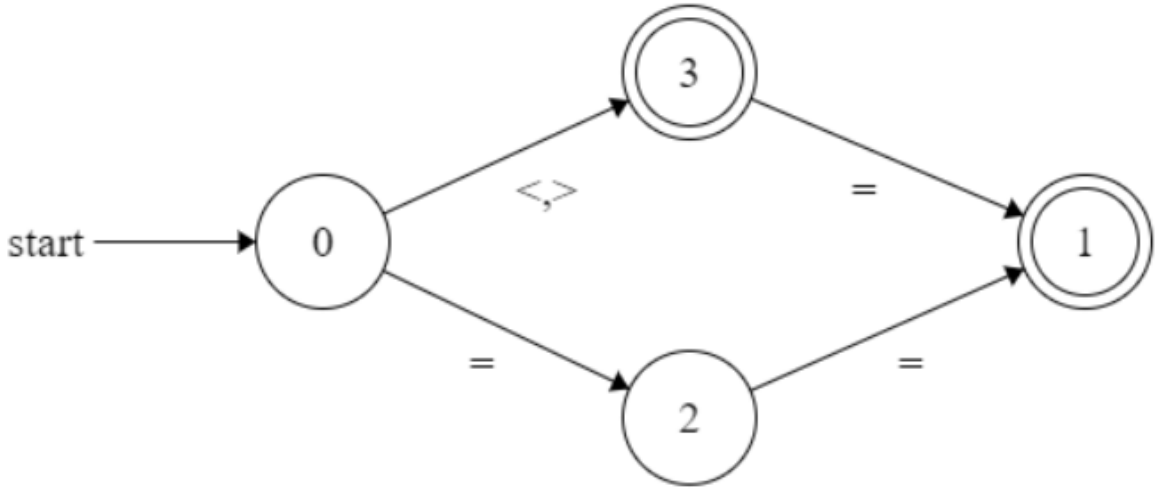
Identifier (ID):



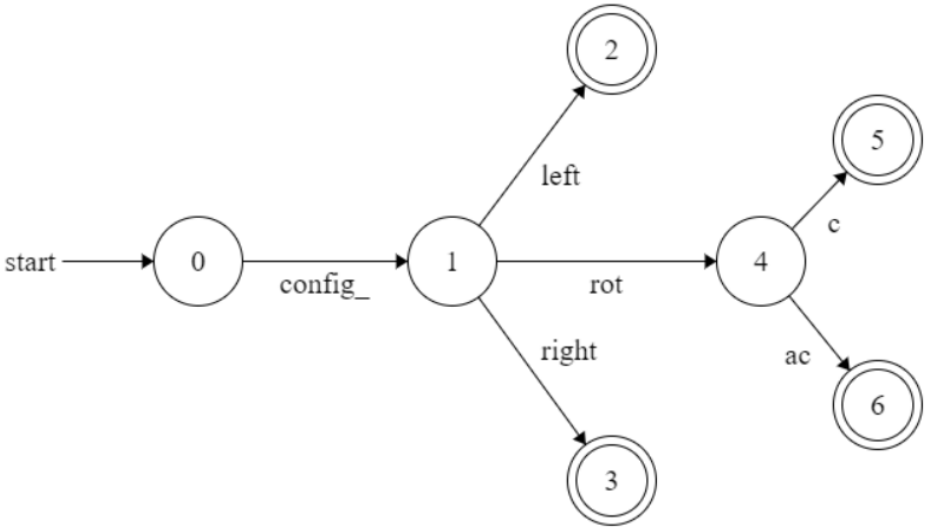
PIECE_ID:



COMPARISON:



KEY_CONTROL:



Special Lexemes and How to Handle them

When we talk about lexemes, there are a certain number of them which have a particular meaning other than the one we intend to give them. A very good example of these can be reserved keywords in the language we are using (here it is Python). Conditional statements (“if”, “elif”, “else”, “for”, “while”) need to be distinguished from their innate properties as defined by Python. These need to be especially handled as they might create problems while compiling. The SLY library documentation provides a very simple yet effective way to tackle this. We can simply take the tokens as input, and manually check through the preset list of reserved keywords. If we encounter one, we can remap that token to the reserved keyword (if and when needed).

```
from sly import Lexer

class CalcLexer(Lexer):
    tokens = { ID, IF, ELSE, WHILE }
    # String containing ignored characters (between tokens)
    ignore = ' \t'

    # Base ID rule
    ID = r'[a-zA-Z_][a-zA-Z0-9_]*'

    # Special cases
    ID['if'] = IF
    ID['else'] = ELSE
    ID['while'] = WHILE
```

The special cases will automatically be remapped to the correct token depending on the functionality we want it to have. For example, if the value of an identifier is “if” above, an IF token will be generated.

Adding Match Action

It can happen that a lexeme can match more than one pattern and the lexical analyzer needs to handle such cases to avoid any inconsistency. It needs to provide the subsequent handling phases with additional yet specific information about which particular lexeme it needs to choose. One example we can see is that the character ‘**a**’ and ‘**b**’ will point to the same token. The token then needs to be assigned the specific value instead of just having the token “**letter**”. Thus it will return the token “**letter**” as well as the attribute value ‘**a**’. The token name influences parsing decisions, while the attribute value influences translation of tokens after the parse. The

additional information i.e. token attribute can be assigned to the returned token using SLY library. A similar case will occur for '**0**' and '**1**' for the token "**digit**". The code snippet below will handle this case.

```
@_(r'\d+')
def NUMBER(self, t):
    t.value = int(t.value) # Convert to a numeric value
return t
```

The method **NUMBER** takes a single argument of type **Token**. The method can change the token attribute as it sees appropriate.

Overlapping Patterns

We have alluded to the two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

- Always prefer a longer prefix to a shorter prefix.
- If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program.

The first rule makes ≤ one instead of two lexemes. The second rule makes 'if' a keyword and not an ID.

Error Handling

An essential part of any language is that it can efficiently handle errors that can occur. Normally how error handling works, is that whenever it encounters a faulty character, it will stop taking input and notify the user of the problem that might have occurred. In our case, we don't want the code to just stop, we want it to handle as many errors as it can and only stop if any major unforeseen problem occurs. Some examples of errors that we might encounter:

- Encountering faulty characters
- Receiving too many characters that exceed specified lengths

The SLY library provides an *error()* method to handle exactly this. The recovery mechanism provided by SLY is comparable to Unix Yacc. The error method receives a token where the value attribute contains all remaining untokenized text. We have used *panic mode*

recovery for error handling. In *panic mode recovery*, we delete successive characters from the remaining untokenized text, until the lexical analyzer can find a well-formed token at the beginning of what input is left.

Scanner Test Plan

Input :

```
BOARD: 10 10

CONTROLS:
config_left = LEFT_ARROW

PIECE: @l

ACTION:
mygame()

FUNCTION:
def mygame(){
  set_speed = 7
}
```

Output :

TOKEN(type='BOARD', value='BOARD')

TOKEN(type='COLON', value=':')

TOKEN(type='NUMBER', value=10)

TOKEN(type='NUMBER', value=10)

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='CONTROLS', value='CONTROLS')

TOKEN(type='COLON', value=':')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='KEY_CONTROL', value='config_left')

TOKEN(type='EQUAL', value='EQUAL')

TOKEN(type='KEY_CONSTANTS', value='LEFT_ARROW')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='PIECE', value='PIECE')

TOKEN(type='COLON', value=':')

TOKEN(type='PIECE_ID', value='[[1,0,0,0],

[1,0,0,0],

[1,0,0,0],

[1,0,0,0]])

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='ACTION', value='ACTION')

TOKEN(type='COLON', value=':')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type=ID, value='mygame')

TOKEN(type='LEFT_PAR', value='(')

TOKEN(type='RIGHT_PAR', value=')')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='FUNCTION', value='FUNCTION')

TOKEN(type='COLON', value=':')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='DEF', value='def')

TOKEN(type='STRING', value='mygame')

TOKEN(type='LEFT_PAR', value='(')

TOKEN(type='RIGHT_PAR', value=')')

TOKEN(type='LEFT_CPAR', value='{')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='SPEED_CONTROL', value='speed_control')

TOKEN(type='EQUAL', value='EQUAL')

TOKEN(type='DIGIT', value='7')

TOKEN(type='NEWLINE', value='\n')

TOKEN(type='RIGHT_CPAR', value='}')

Changes made in Lexer (In Stage 2)

Although most of our lexer was working exactly to our liking, we have made some changes that we feel will better suit the language. We decided to keep all the tokens that we have defined till now, even though some have been kept just for semantic reasons. They are not used or returned but will increase the readability of the code. Plus, some minor error handling was missing which also has been included in the new and updated Lexer.

- The COMMENT token is ignored and not returned. We felt returning a token for it was counterintuitive. A comment should be informative to the programmer and does not provide any viable information for the Lexer.

- We have also included an error message for encountering an Illegal token.

Apart from these, we have made very minor changes here and there in structuring the Lexer for better readability.

Parser Design

Introduction

In general a Parser uses a Context Free Grammar (CFG) to validate the input string and produce output for the next phase of the compiler. This is done by systematically breaking down the possible cases of the input into relevant “baskets” or expressions. Output could either be a parse tree or an abstract syntax tree. Making these expressions is only half the job done, now to interleave semantic analysis with the syntax analysis phase of the compiler, we use Syntax Directed Translation.

Syntax Directed Translation (or SDT in short) 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 lexical values of nodes, constants & attributes associated with the non-terminals in their definitions. 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.

Sly is an ***LALR parser generator***. LALR Parser is lookahead LR parser. It is the most powerful parser which can handle large classes of grammar. It is intermediate in power between SLR and CLR parser. It is the compaction of CLR Parser, and hence tables obtained in this will be smaller than CLR Parsing Table. The size of CLR parsing table is quite large as compared to other parsing table. LALR reduces the size of this table. LALR works similar to CLR. The only difference is, it combines the similar states of CLR parsing table into one single state.

Syntax Directed Translation Scheme

Grammar:

Rule 0 S’ -> S
Rule 1 S -> Section NEWLINE S’
| <empty>

Rule 2 Section -> FUNCTION COLON EXPR5 NEWLINE
| ACTION COLON EXPR4 NEWLINE
| PIECE COLON EXPR3
| CONTROLS COLON EXPR2
| BOARD COLON EXPR1

Rule 3 EXPR1 -> NUMBER NUMBER NEWLINE

Rule 4 EXPR2 -> GAME_CONTROLS EQUAL CONSTANT NEWLINE

| GAME_CONTROLS EQUAL CONSTANT NEWLINE EXPR2 NEWLINE

Rule 5 GAME_CONTROLS -> INIT_BOARD
| SCORE
| HORIZON
| ORIENTATION
| SPEED_CONTROL
| KEY_CONTROL
| GAME_MODE

Rule 6 CONSTANT -> KEY_CONTROL
| KEY_CONSTANTS
| ID
| STRING
| NUMBER

Rule 7 EXPR3 -> PIECE_ID EQUAL STRING
| PIECE_ID EQUAL STRING COMMA EXPR3
| PIECE_ID NEWLINE
| PIECE_ID COMMA EXPR3

Rule 8 EXPR4 -> <empty>

Rule 9 EXPR5 -> PROCEDURE NEWLINE EXPR5
| <empty>

Rule 10 PROCEDURE -> DEF ID LEFT_PAR F_ARGS RIGHT_PAR LEFT_CPAR NEWLINE
FBODY NEWLINE RIGHT_CPAR

Rule 11 F_ARGS -> EXPR2

Rule 12 FBODY -> RETURN
| ID EQUAL BEXPR

Rule 13 FBODY -> ID EQUAL AEXPR
| BREAK
| CONTINUE
| EXPR2 NEWLINE FBODY
Rule 14 AEXPR -> term
| AEXPR -> AEXPR MINUS term
| AEXPR PLUS term

Rule 15 term -> factor
| term COMPARISON factor
| term MOD factor
| term DIVIDE factor

```
                    | term MULTIPLY factor

Rule 16    factor -> LEFT_PAR AEXPR RIGHT_PAR
                   | NUMBER

Rule 17    BEXPR -> OFF
                  | ON
                  | ID BOOL_OP ID
                  | NOT ID

Rule 18    BOOL_OP -> OR
                     | AND
```

Transition Diagram: Refer to grammar.txt (The diagram was too intricate as there were a total of 103 states with loads of connections associated with them. So we have generated the corresponding table to refer to the LR(0) automata.)

Challenges faced:

The grammar was particularly difficult as we had a lot of tokens defined in the first place. It took time to make accurate and semantically sound.

- Although finding SDT actions corresponding to each production rule was not difficult, finding out different ways to generate the intermediate code based on the rule was exciting and challenging at the same time.

Parser Conflicts:

Associativity Problem/Shift-reduce Conflict:

Usually arithmetic and boolean expressions with more than one operation in the same expression give rise to ambiguous grammar, which means that there may be more than one parse tree for the same string as the precedence and the associativity of the operations are not specified. For such an expression, not only is the grammar complex, but the parse trees are huge with lots of subtrees that are just reinterpretation steps. To overcome these challenges and conflicts, we have completely gone away with such arithmetic and boolean expressions with multiple operations i.e., our parser only accepts expressions of the form “Num/Var Op Num/Var”. The reason for such a decision is that considering all the use cases of our programming language, these expressions are hardly ever used. However, if the programmer still wants to do such calculations in their program, they can do so in multiple lines with binary operations, hence the power of the language is not affected.

Symbol Table:

Declaration Check: In our compiler design, we have used symbol tables to keep track of identifier declaration and definition. In any programming language, it is important to check if the variables or functions are declared or defined earlier before they are being used, further, if they are defined what values do they store. To achieve this, we have used two symbol tables, each for Variables and Functions.

Structure of Symbol Table:

In CCLang, **Functions** do not store any value so we have used an **unordered list** as a symbol table for functions.

Whenever a function is defined a new entry is made into the list with the name of the function. When a function is called, our compiler checks if the function entry already exists in its symbol table.

Symbol_table_func (List)

func1
func2
func3
..
..

For **Variables**, we use a **hash map** (dictionary) as a symbol table.

Whenever a variable is declared with its value (variables are always declared with a value in CCLang), a new entry is made into the hashmap with the variable name as the key and the variable value as its value. When a variable is used, our compiler checks if the variable entry exists in its symbol table and if so, then it replaces the identifier with its value in the table.

Symbol_table_var (Hash Map)

Key	Value
var1	val1
var2	val2
var3	val3
..	
..	

The **scope** of each identifier is kept global for simplicity.

Test Cases

Test Case-1 (Successful):

```
BOARD: 10 20

CONTROLS:
set_speed = 9

PIECE: @l @t

ACTION:
mygame()

FUNCTION:
def mygame() {
  x = 9
  x = x - 2
  set_speed = 7
  return
}
```

Test Case-2 (Unsuccessful):

```
BOARD: 10 20

CONTROLS:
horizon = 1

PIECE: @l @t @i

ACTION:
mygame()

FUNCTION:
def mygame() {
  x = 9
  x = x - 2
  set_speed = x
  return
}
```

In the above test case, the parser throws an error at line 15 (set_speed = x) as at this point our grammar does not allow initializing any game control using a variable.

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

As discussed in the pipeline schema earlier, our parser after parsing the tokens given by the lexer generates an intermediate code, this intermediate code imports the tetris engine and adds features, functionalities and configures the engine as programmed by the programmer. Further, to execute this intermediate code, a Makefile is generated.

Effectively, after the programmer writes the code and compiles it, two files are generated:

1) Intermediate Code: mygame.py
2) Makefile (which runs the intermediate code)

To run the program, programmer just needs to run the Makefile.

Makefile:

python mygame.py

Division of labor between Scanner & Parser

The Scanner is designed by keeping the complete scanner transition table and the parser grammar in mind. We, for now, tried to create a balance between the lexer and the parser but in cases where the work could be done by both (for example: assignment of 2D array to the corresponding pieces) we tried to give benefit to the parser, so that there is more flexibility in designing the grammar.

As the number of tokens increase, the workload of lexer increases but the grammar definition of the parser simplifies, so there's a tradeoff. Our token set is extensive yet we try to ensure that unnecessary tokens are not created to unnecessarily burden the lexer. For example, for tetris pieces instead of making different tokens for different pieces, we create a token class PIECE_ID which recognizes all the pieces. This design decision also help us to allow custom pieces which can be programmed by the programmer in our game.

Roles and Responsibilities of Team Members

All members of our group will be working together on all the components of the projects. In order to streamline our work, each member is responsible for planning and supervising the various tasks needed to be completed in the different aspects of the project.

Abhijeet Swain - Documentation, Planning, and designing the Tetris engine while interfacing it with the parser.

Aman Jain - Worked on implementation of the lexer design and debugging of the errors and issues in various components.

Atishay Jain - Coordinated the work and took major design decisions related to Scanner, Parser and their interfacing.

Dhairy Agrawal - Worked on defining main components of the grammar, implementation of the parser and its interfacing with the Tetris Engine.

Shubhankar Ravindra Kate - Documentation, Planning and Top Level Design Documentation and structuring of project.