Report for Stage 2: Tetris Game

for

COMPILER CONSTRUCTION (CS F363)

BY

Group 36

KUNAL KARIWALA -	2019A7PS0134G
RICKY PATEL -	2019A7PS0051G
AMAN GUPTA -	2019A7PS0052G
SHAURYA PURI -	2019A7PS0035G
NIKHIL MISHRA -	2019A7PS0112G

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE,
K.K. BIRLA GOA CAMPUS

Syntax Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax-directed definitions. A syntax-directed translation scheme (SDT) is a context-free grammar with program fragments embedded within production bodies. The program fragments are called semantic actions and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammar symbols, then we quote them.

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.

Translation Scheme

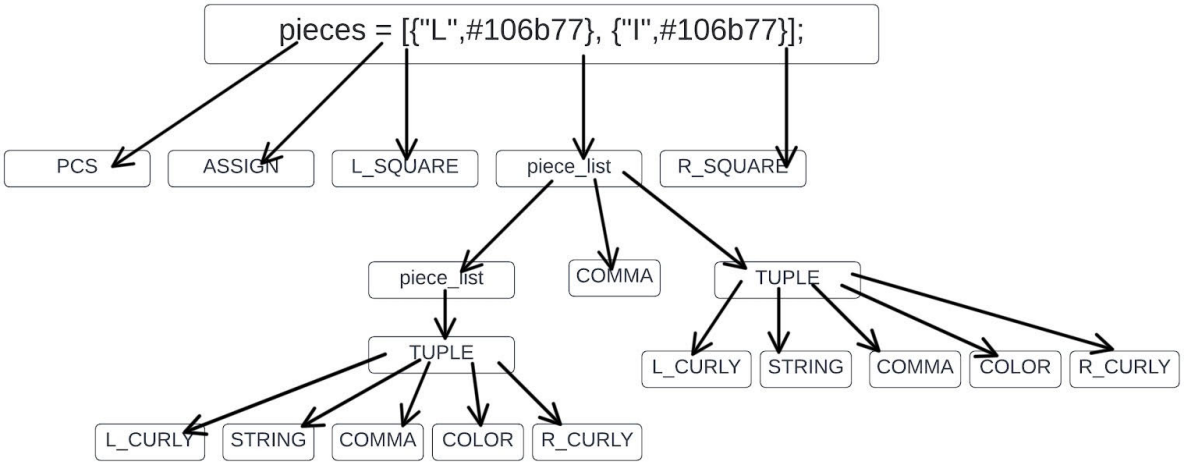
Production	Semantic Rule
expr2 -> expr2 PLUS term	{exp2.VAL := exp2.VAL + exp2.VAL }
expr2 -> expr2 MINUS term	{exp2.VAL := exp2.VAL - exp2.VAL }
expr2 -> expr2 TIMES term	{exp2.VAL := exp2.VAL * exp2.VAL }
expr2 -> expr2 DIVIDE term	{exp2.VAL := exp2.VAL / exp2.VAL }
expr2 -> term	{expr2.VAL := term.VAL }
term -> term MULTIPLY factor	{term.VAL := term.VAL * term.VAL }
term -> term DIVIDES factor	{term.VAL := term.VAL / term.VAL }
term -> factor	{term.VAL := factor.VAL }
factor -> NUMBER	{ factor.VAL:= LEXVAL}
factor -> (expr2)	{factor.VAL := expr2.VAL }
num_var -> SPEED	{num_var.VAL:= LEXVAL}
bool_var -> FLAG	{bool_var.VAL:= LEXVAL}

Example:

Source Code Snippet :

```
pieces = [{"L",#106b77}, {"I",#106b77}];
```

Breakdown of above Expression :



CFG Grammar for above Example:

pieces_set : PCS ASSIGN L_SQUARE piece_list R_SQUARE

piece_list : piece_list COMMA tuple | tuple

tuple : L_CURLY STR COMMA COLOR R_CURLY

Challenges

Parser conflicts

The grammar specification of our language may be ambiguous for certain strings in our source program file. For example, if you are parsing the string "3 * 4 + 5", there is no way to tell how the operators are supposed to be grouped. For example, does the expression mean "(3 * 4) + 5" or is it "3 * (4+5)"? When an ambiguous grammar is given, a **"shift/reduce conflict"** or **"reduce/reduce conflict"** occurs.

A **shift/reduce conflict** is caused when the parser generator can't decide whether or not to reduce a rule or shift a symbol on the parsing stack. By default, all shift/reduce conflicts are resolved in favor of shifting. Although this strategy works in many cases (for example, the case of "if-then" versus "if-then-else"), it is not enough for arithmetic expressions. To resolve ambiguity, especially in expression grammars, SLY allows individual tokens to be assigned a precedence level and associativity. This is done by adding a variable *precedence* to the parser class. When shift/reduce conflicts are encountered, the parser generator resolves the conflict by looking at the precedence rules and associativity specifiers.

Reduce/reduce conflicts are caused when there are multiple grammar rules that can be applied to a given set of symbols. This kind of conflict is almost always bad and is always resolved by picking the rule that appears first in the grammar file. Reduce/reduce conflicts are almost always caused when different sets of grammar rules somehow generate the same set of symbols. It should be noted that reduce/reduce conflicts are notoriously difficult to spot simply looking at the input grammar. When a reduce/reduce conflict occurs, SLY will try to help by printing a warning message

Error productions

Error production strategy helps in generating appropriate error messages that are encountered while parsing a given input string.

The lexical analyzer tokenizes the input string one line at a time, with '\n' serving as the delimiter. The output of the lexer is passed on to the parser, which checks for consistencies within the tokens generated against the set of production rules specified by the parser. If a token does not match against any of the rules, or if there is a syntactical inconsistency, an error message is displayed along with the corresponding line number. If no error is detected, the program continues with the next line.

```
with open('tet_conf.tads', 'r') as fileh:
        lines = file.readlines()
        line_count = 1;
        for line in lines:
            result = parser.parse(lexer.tokenize(line))
            try:
                result = parser.parse(lexer.tokenize(line))
            except:
                print("Syntax error in line #",line_count,":",line, "skipping
this line")
            line_count+=1
        # print(result)
        print(parser.data_map)
```

Test Cases

1) INPUT:

```
speed = (9+2);
```

OUTPUT:

First, the sum of 9 and 2 is computed, and the speed is assigned as that sum.

2) INPUT

```
FLOAT = 10;
```

OUTPUT:

ERROR, since FLOAT is a reserved keyword and cannot be assigned as a variable name.

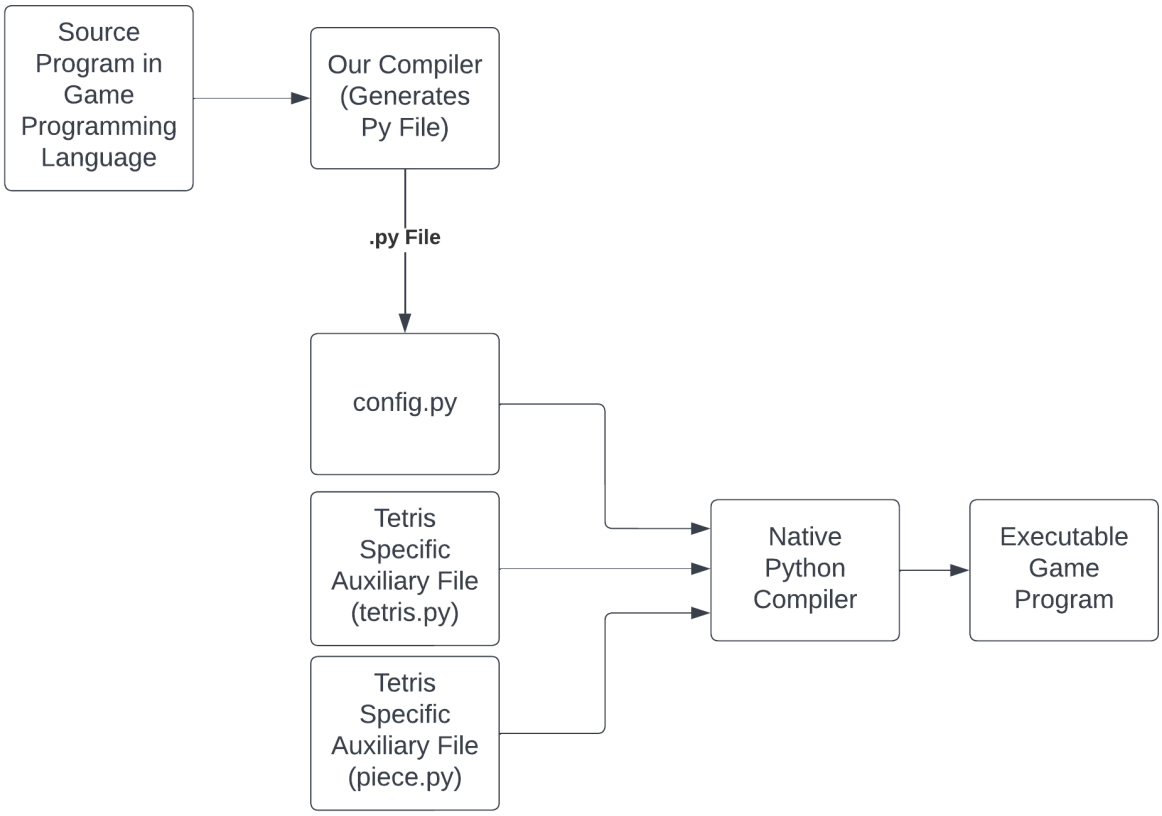
3) INPUT

```
perf = 10;
per f = perf - 2;
```

OUTPUT

ERROR , since per f is not recognized by our parser.

Complete end-to-end Tetris Game Engine Programming Toolchain



Makefile

dependencies:

    pip install sly

    pip install pygame

lexer:

    python lexer.py sample.tg

parser :

python parser.py

runGame:

python tetris.py

```
dependencies:

pip install sly

pip install pygame

lexer:

python lexer.py sample.tg

parser :

python parser.py

runGame:

    python tetris.py
```