Group 29 - Confused

Complete design specs and syntax-directed translator design

Syntax Directed translation scheme

In designing the syntax directed translation, we had to generate the syntax with grammar, and the actions for each of the determined grammar statements.

The specifications for the parser generator are given below:
This is the Backus-Naur Form of the Context Free Grammar we have designed for our language:

PARSER

1. **INITIALISATION & RE-ASSIGNMENT STATEMENT GRAMMAR**
    a. DATA_TYPE = INT_TYPE | STRING_TYPE ..
    b. VAL = NUMBER | STRING_LITERAL | BIRF
    c. SET = SET SEPARATOR VAL | VAL
    d. ARRAY_VAL = LBLOCKPAREN RBLOCKPAREN | LBLOCKPAREN SET RBLOCKPAREN
    e. INIT_STMT = DATA_TYPE ID ASSIGN VAL EOL | ARRAY ID ASSIGN ARRAY_VAL EOL | DATA_TYPE ID ASSIGN EXPR EOL
    f. REASS_STMT = ID ASSIGN VAL EOL | ID ASSIGN ARRAY_VAL EOL | ID ASSIGN EXPR EOL

2. **JUMP STATEMENT GRAMMAR**
    a. BREAK_STMT = BREAK EOL
    b. EXIT_STMT = EXIT EOL

3. **FUNCTION STATEMENT GRAMMAR**

PARAMETERS -

a. PARAM = ID | VAL
b. PARAMS = PARAMS SEPARATOR PARAM | PARAM

I. RETURN FUNCTION

i. BIRF_WOP = GET_NAME | GET_NEXT_TETROMINO | GET_CHAR
ii. BIRF_WOP_CALL = BIRF_WOP ( )
iii. BIRF_WP = GET_BOARD | CHECK_CLEARED_LINE
iv. POP_CALL = POP (ID) | POP (ARRAY_VAL)
v. LEN_CALL = LEN (ID) | LEN (ARRAY_VAL)
vi. BIRF_WP_CALL = BIRF_WP (PARAMS) | POP_CALL | LEN_CALL
vii. BIRF = BIRF_WP_CALL | BIRF_WOP_CALL

II. VOID FUNCTION

a. BIVF_WOP = ROTATE_RIGHT | ROTATE_LEFT | MOVE_RIGHT | MOVE_LEFT | HARD_DROP | DISPLAY_BOARD | CLEAR_SCREEN
b. BIVF_WP = SOFT_DROP | CLEAR_LINE | DISPLAY | DISPLAY_TETROMINO | DISPLAY_UPCOMING_TETROMINO | ADD_SCORE | REM | PUSH
c. SET_SPEED_CALL = SET_SPEED (EASY) | SET_SPEED (MEDIUM) | SET_SPEED (HARD)
d. SET_GAME_MODE_CALL = SET_GAME_MODE (NORMAL) | SET_GAME_MODE (SPRINT)
e. REM_CALL = REM (ARRAY_VAL SEP NUMBER) | REM (ARRAY_VAL SEP ID) | REM (ID SEP NUMBER) | REM (ID SEP ID)
f. PUSH_CALL = PUSH (ARRAY_VAL SEP VAL) | PUSH (ARRAY_VAL SEP ID) | PUSH (ID SEP VAL) | PUSH (ID SEP ID)
g. BIVF_WOP_CALL = BIVF_WOP ( )
h. BIVF_WP_CALL = BIVF (PARAMS) | | SET_SPEED_CALL | SET_GAME_MODE_CALL | REM_CALL | PUSH_CALL
i. BIVF = BIVF_WOP_CALL EOL | BIVF_WP_CALL EOL

4. COMPOUND STATEMENTS-

a. ALL_STMT = INIT_STMT | REASS STMT | BREAK_STMT | EXIT_STMT | BIVF | IF_ELSE | WHILE_LOOP | TIMEOUT_LOOP
b. COMPOUND_STMT = COMPOUND_STMT ALL_STMT | ALL_STMT

5. CONDITIONAL GRAMMAR-

IF_STMT = IF (EXPR) LCURLYPAREN COMPOUND_STMT RCURLYPAREN

ELSE_STMT = ELSE LCURLYPAREN COMPOUND_STMT RCURLYPAREN | ELSE IF_STMT ELSE_STMT | ε

IF_ELSE = IF_STMT ELSE_STMT

6. LOOPING GRAMMAR -

WHILE_LOOP = WHILE (EXPR) LCURLYPAREN COMPOUND_STMT RCURLYPAREN

7. EXPRESSIONS

a. OPERAND = ID | NUMBER
b. U_OP2 = '-'
c. U_OP1 = NOT
d. B_OP1 = OR
e. B_OP2 = AND
f. B_OP3 = GT | LT | GTE | LTE | NE | EE
g. B_OP4 = '+' | '-'
h. B_OP5 = '*' | '/' | '%'
i. FACTOR = OPERAND | '(' EXPR ')'
j. FACTOR_WS = FACTOR | U_OP2 FACTOR
k. TERM = TERM B_OP5 FACTOR_WS | FACTOR_WS
l. SUM = SUM B_OP4 TERM | TERM
m. REL = REL B_OP3 SUM | SUM
n. NOTEXP = U_OP1 REL | REL
o. ANDEXP = ANDEXP B_OP2 NOTEXP | NOTEXP
p. OREXP = OREXP B_OP1 ANDEXP | ANDEXP
q. EXPR = OREXP

Note:

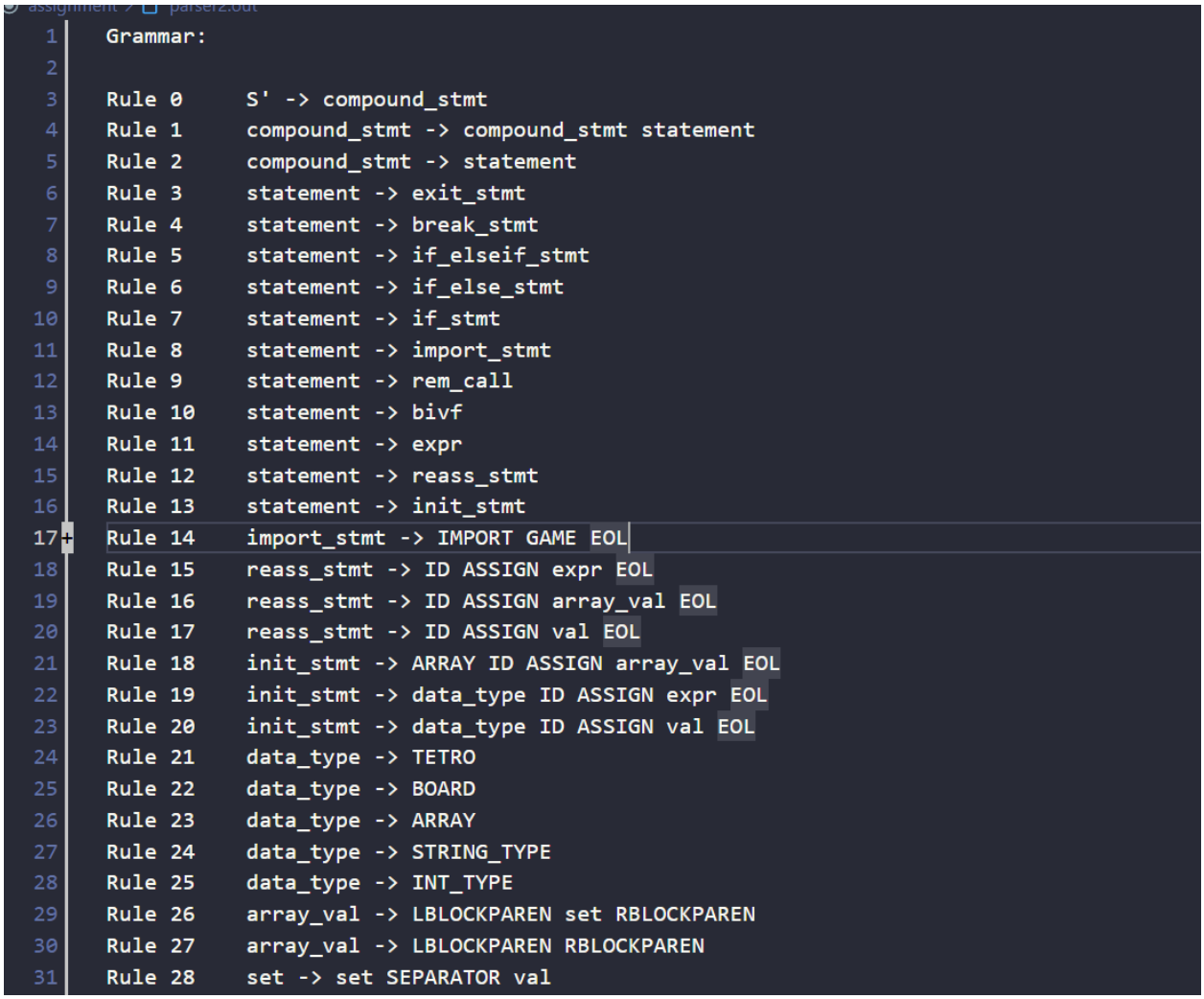
1. ε refers to the empty production rule.
2. Looping statement -> A while loop with a compound statement as the condition in it.
3. Expressions are given with precedence for operators defined in SLY according to their documentation specifics.
4.

LR Automaton

The SLY docs say that:

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.

SLY has a built-in functionality to generate a *parser.out* file which is automatically generated with parsing tables, states, and shift reduce configurations, conflicts and how they are handled when an example code snippet is passed into the parser. We are attaching this file in the gitlab submission as well.
This has all the grammar along with the parsing states with output for each of them like so:



```
Grammar:

Rule 0     S' -> compound_stmt
Rule 1     compound_stmt -> compound_stmt statement
Rule 2     compound_stmt -> statement
Rule 3     statement -> exit_stmt
Rule 4     statement -> break_stmt
Rule 5     statement -> if_elseif_stmt
Rule 6     statement -> if_else_stmt
Rule 7     statement -> if_stmt
Rule 8     statement -> import_stmt
Rule 9     statement -> rem_call
Rule 10    statement -> bivf
Rule 11    statement -> expr
Rule 12    statement -> reass_stmt
Rule 13    statement -> init_stmt
Rule 14    import_stmt -> IMPORT GAME EOL
Rule 15    reass_stmt -> ID ASSIGN expr EOL
Rule 16    reass_stmt -> ID ASSIGN array_val EOL
Rule 17    reass_stmt -> ID ASSIGN val EOL
Rule 18    init_stmt -> ARRAY ID ASSIGN array_val EOL
Rule 19    init_stmt -> data_type ID ASSIGN expr EOL
Rule 20    init_stmt -> data_type ID ASSIGN val EOL
Rule 21    data_type -> TETRO
Rule 22    data_type -> BOARD
Rule 23    data_type -> ARRAY
Rule 24    data_type -> STRING_TYPE
Rule 25    data_type -> INT_TYPE
Rule 26    array_val -> LBLOCKPAREN set RBLOCKPAREN
Rule 27    array_val -> LBLOCKPAREN RBLOCKPAREN
Rule 28    set -> set SEPARATOR val
```

Challenges

In designing the grammar and its corresponding syntax directed translation scheme, we encountered a few challenges regarding error handling and symbol table mapping.

- In implementing grammar statements like:
    - `ID ASSIGN val EOL`
    - `ID ASSIGN array_val EOL,`
      We had to ensure that the types of the identifier and the type of the val in the first case and the type of the identifier and the type of array_val must match.
      For this, we used a map for var_type and var_val i.e for variable types and variable values.
- Implementing symbol table in the form of var_type_map and var_val_maps:
    - This is used to ensure type safety in the language for statements.
    - To map token values and also determine their types, we used a python dictionary to store the type and their values as a hashed map.
    - This is then used to check for potential conflicts in terms of variable name matching, in which case an error would be thrown by the compiler with the line at which the conflict generated a syntax error.

      ```
      int a = 5;   // works fine
      str a = “sameNameError”; -> This will generate an error because the variable name a is already defined with the type int, and cannot be used again.
      ```
    - In a similar fashion we used a var_type_map to determine potential type mismatch errors in grammar statements, for instance, if an integer were to be initialised with a string literal.

      ```
      int a = “wrongType”; -> Leads to an error thrown (Type mismatch error);
      ```

- Another form of error we had to handle was the case when there was no variable initialised, but was used in an assignment statement. In this case, we throw another error by using the var_val_map again.

An example of a code snippet that illustrates this error handling:

```
@_('ID ASSIGN val EOL')
def reass_stmt(self, p):
    if self.var_val_map.get(p.ID) == None:
        self.error(self.VARIABLE_NOT_FOUND_ERROR, p.lineno)
        return self.VARIABLE_NOT_FOUND_ERROR
    else:
        if self.var_type_map[p.ID] == p.val['type']:
            self.var_val_map[p.ID] = p.val['value']
            return f"{p.ID} {p.ASSIGN} {p.val['value']}\n"
        else:
            self.error(self.VARIABLE_MISMATCH_ERROR, p.lineno)
            return self.VARIABLE_MISMATCH_ERROR
```

(In this case, the grammar rule is used to reassign an identifier a value. If the variable is not in the map, we need to throw a variable not found error. Otherwise we check for type mismatch errors if any)

Test Cases

In generating test cases, we need to check whether the appropriate grammar rules are being used when a statement is passed and its tokens generated.

Syntax Tests

No semicolon:

This test is used to check whether every statement ends with the EOL token. The purpose of this test is to check whether all the tokens comply with appropriate grammar rules, because the EOL specifies how the statements are differentiated among each other.
Example:
str check = “this is a statement”; correct
str check = “this is a statement” incorrect, because EOL is not specified. No rule can determine this if there is no proper EOL.

Function call syntax test:
This test is used to test if the invocation of a function via its function call has been done properly or not. It uses the function definition as a template to compare the function call statement and also the number and type of parameters being passed.

If else statement syntax test:
This testcase is used to test the “if else” conditional statement of our language since it is important to implement conditionals to enable checks to determine appropriate game engine parameters as well.
Example:

```
 if (a == 5) {
        str hello = “Hello”;
        display(hello);
}
```

This snippet is syntactically correct, assuming a was defined before.

Declaration Tests
Variable Declaration:
Checks for correct and semantically consistent variable declarations in the code.
Example: int a = 5; str b = “Hello World”; correct
int a = 5; str a = “Hello World”; incorrect

Array Declaration:

Checks for correct declaration of the enumerated data type: array
Example: array arr = [2,3,4];

Updation Tests

Updation tests essentially check for variable re-assignments. This test is used to check if mutable objects can be changed according to the grammar rules specified.

Example:

```
int a = 4;
a = 5;
```

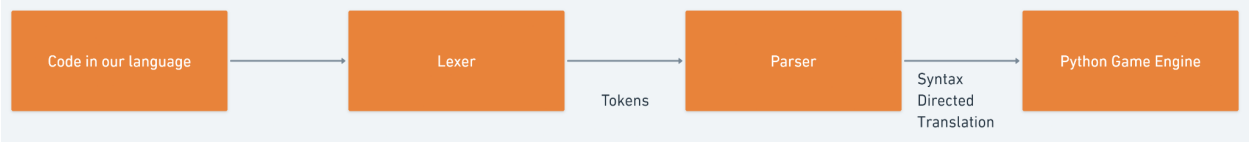
This snippet is syntactically correct and will reassign a to have a value of 5.

To run all tests, we have made a make command: `make tests`

```
Parser debugging for TetrisParser written to parser2.out
_____RUNNING ALL TESTS_____


____IN TEST: FAILING2____
Syntax Error at line 1: Variable a not declared
Syntax Error at line 1: None
None
_____END____FAILING2____


____IN TEST: IFSTATEMENT____
c = 3
d = 2
c = 4
d = 5
getName()

_____END____IFSTATEMENT____


____IN TEST: UPDATIONTEST____
e = 3
e = 5

_____END____UPDATIONTEST____


____IN TEST: ARRAYPUSH____
Syntax Error at line 1: None
None
_____END____ARRAYPUSH____


____IN TEST: FUNCTIONTEST____
t = getNextTetromino()
b = getBoard(stdscr)
hardDrop(getNextTetromino(), getBoard(stdscr))

_____END____FUNCTIONTEST____


____IN TEST: DECLARATIONTEST____
a = 4

_____END____DECLARATIONTEST____


____IN TEST: FAILING1____
Syntax Error at line 1: None
None
_____END____FAILING1____


____IN TEST: IMPORTS____
from game import *
```

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

Our design implementation involves a file which contains “our” requisite code. This code is then fed into the Lexer, which decomposes this stream of characters into token-value pairs. These pairs are fed further to the Parser as a part of this continuing pipeline. The job of the Parser is to generate the Abstract Syntax Tree of the underlying grammar. It is also in charge of using Syntax Directed Translation to convert this intermediate step into the necessary and equivalent Python code. This Python code is what is responsible for our Tetris game execution on the command line.



In our toolchain, we use SLY to implement the lexer, parser components. The game engine uses the curses module of python as an additional library component to enable keystroke inputs from the command line while the game is running.
Building our project involves the following stages:
Install requirements for the game engine -> compile and run the main.py file.

In the same manner, testing our project involves the following stages:
Install requirements for the game engine -> Compile and run the test.py file

All of these are handled by using a Makefile with run commands.