Part 4 Documentation:

I. Things we changed compared to previous phases:

We made some changes to our grammar to fix some recursion problems and to solve some pairwise disjointness cases we found when trying to implement our acceptor.

Following is the updated version of our grammar:

<program> ::= <constants> <varDeclaration><fctDefinition><mouhima>
<constants> ::= tabita identifier aadad “;” <constants> | ∈
<varDeclaration> ::= <varType> <variables><varDeclaration>| ∈
<variables> ::= identifier, <variables>
<varType> ::= <Stype><structured> | <address>
<Stype> ::= aadaad | ramz
<address> ::= &<Stype>
<structured> ::= “[“<expression>”]” | “[“<expression>”]”
<fctDefinition> ::= dalla <Stype> identifier (<parameters>) bda <statements> sali
<fctDefinition>| identifier “(“<parameters>”)” bda <statements> sali <fctDefinition>| ∈
<mouhima> ::= mouhima“(“ ”)” bda <statements> sali

<parameters> ::= <parameter> <parameters> | ∈

<parameter> ::= <Stype> identifier
<statements> ::= <asgOrFnctCall><statements> | <selectionStmt><statements> |
<repetitionStmt><statements>| hbess”;” <statements>|<returnStmt> |
<varDeclaration><statements>|∈
<asgOrFnctCall>::= identifier <temp4>| <qra>| <tebaa>| “(“<expression>”)” <asgOp>
|LIT <asgOp>
<asgOp> ::= <operator> <expression>”=>”identifier | “=>” identifier
<temp4> ::= “(”<IDs>”)” <OutpStore>| <operator> <expression> “ =>” identifier | “=>”
identifier
<statementstemp> ::= <statements> | ∈
<fctCall> ::= identifier”(“<IDs>”)” <OutpStore>| <qra> | <tebaa>
< OutpStore> ::= “=>” identifier “;” | “;"
<IDs> ::= identifier<identifiertemp> | ∈
<identifiertemp> ::= ,<IDs> | ∈

```
 <returnStmt> ::= rajaa <returnedVal>;
 <returnedVal> ::= identifier | aadad |∈
 <assignmentStmt> ::= <expression> => identifier;
 <selectionStmt> ::= ila (<condition>): <statements>: <selectionTemp>
 <selectionTemp>::= ila ghalat: <statements>: | ∈
 <repetitionStmt> ::= mahed (<condition>): <statements>:
 <expression> ::= <operand> <expTemp> |"("<expression>")" <expTemp>
 <expTemp>::= <operator> <expression> | ∈
 <condition>::= (MACHI | ∈) <operand> <CompEx> <operand> <conditionTemp>
 <conditionTemp>::= (<LogEX>  <condition>) | ∈
 <operand>::= identifier | LIT
 <operator> ::= + | - | / | * | %
 <CompEx>::= < | > | <= | >= | = | !=
 <logEX> ::= WLA | O | MACHI
 <qra> :== qra(<IDs>);
 <tebaa> :== tebaa(<combination>);
<combination> ::= "$"text"$" <combination> | identifier<combination> | ∈
```

We also made a change to our lexer; we changed the tokens to become a class we named Token. This change would allow us to include the following information about the token: the corresponding lexeme, line number in the program, and token number.

II. Parser:

a. Classes:

- Node: This class represents nodes used in the parse tree. It has two methods:
  - *makeTree():* takes a value and appends it as a node to the parse tree.
  - generate_tree(): prints the entire parse tree level by level using recursion.

- Parser: This class represents the parser and includes methods each of whom represents an acceptor for our language, and each one of them constructs the corresponding sub-tree. By default, it sets the root first to None and the current_token to the beginning of the tokenList. It also has an additional 5 methods:

  - *parse():* This method is used to generate the parse tree and to print the leaves of the parse tree after generating it.

- *generateCode():* It takes as input argument the root of the tree and prints its leaves going from the left to the right.

- *createTree():* Used to create the parse tree. It calls the method program() to initialize the root and then returns it to the method parse() where it was called.

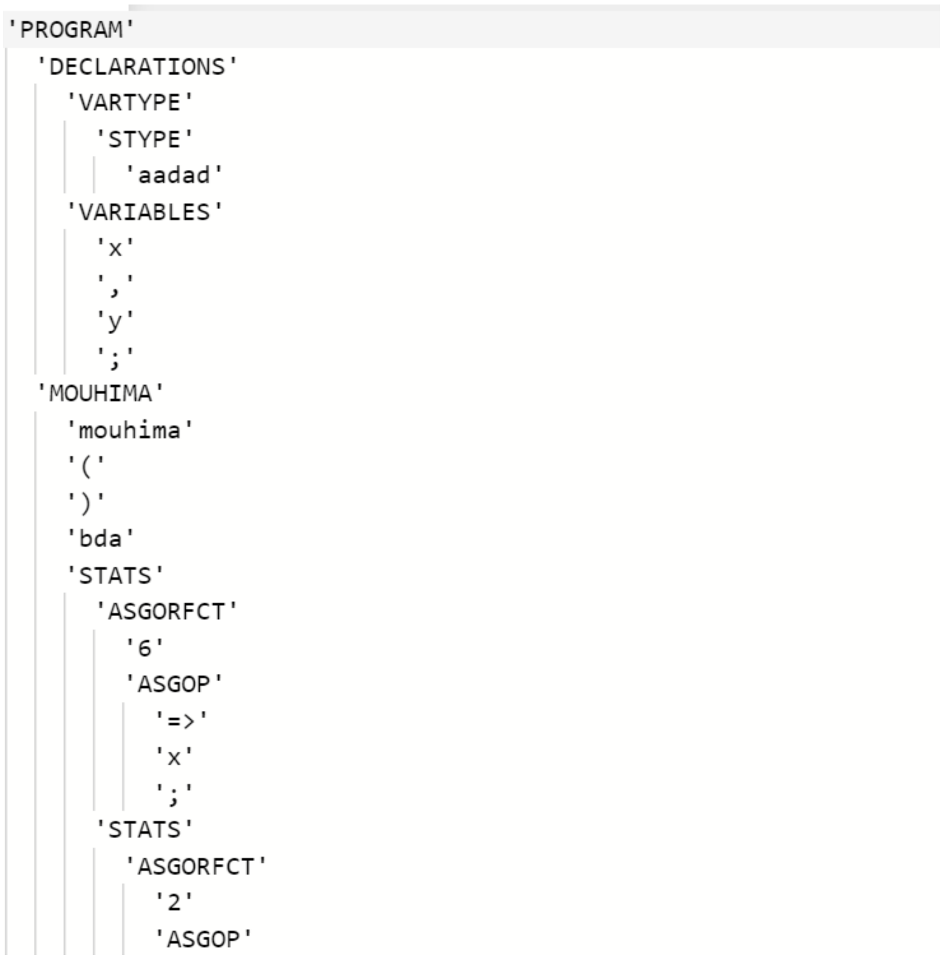
- *program():* This method is used to represent the acceptor of the program and it's used to construct its subtree. According to our language's BNF, it first calls the method constants() to check if there are any, then it calls varDeclarations() to check if there are any variable declarations, and then it calls mouhima() which marks the start of the main of our program.

- *instantiate():* This method gets called before generateAssembly() to initialize all the needed variables. It sets the current_token to the beginning of the tokenList, creates an output file and redirects the output of the program to that file.

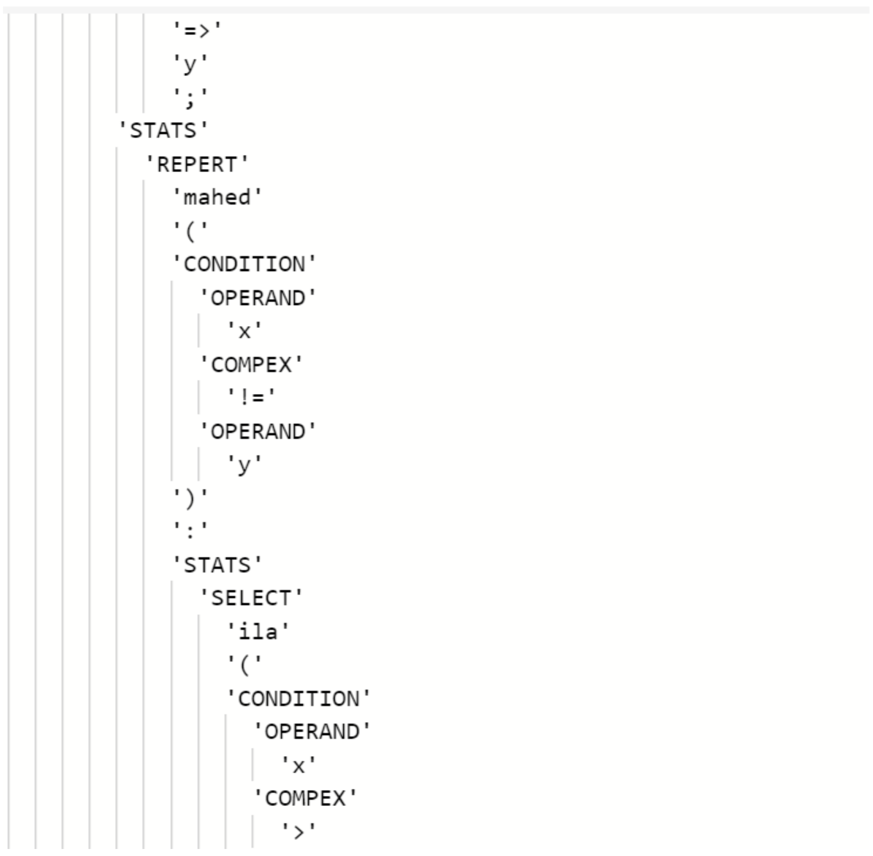
● Token: this class represents the tokens used throughout our code. The class has three attributes:
→ token_num: it represents the token number.
→ token_line: it represents the line where the token appears in the code.
→ token_name: it represents the lexeme that got matched to this token.*

● Code generator: this class is used to generate the equivalent Assembly Language to our source code.

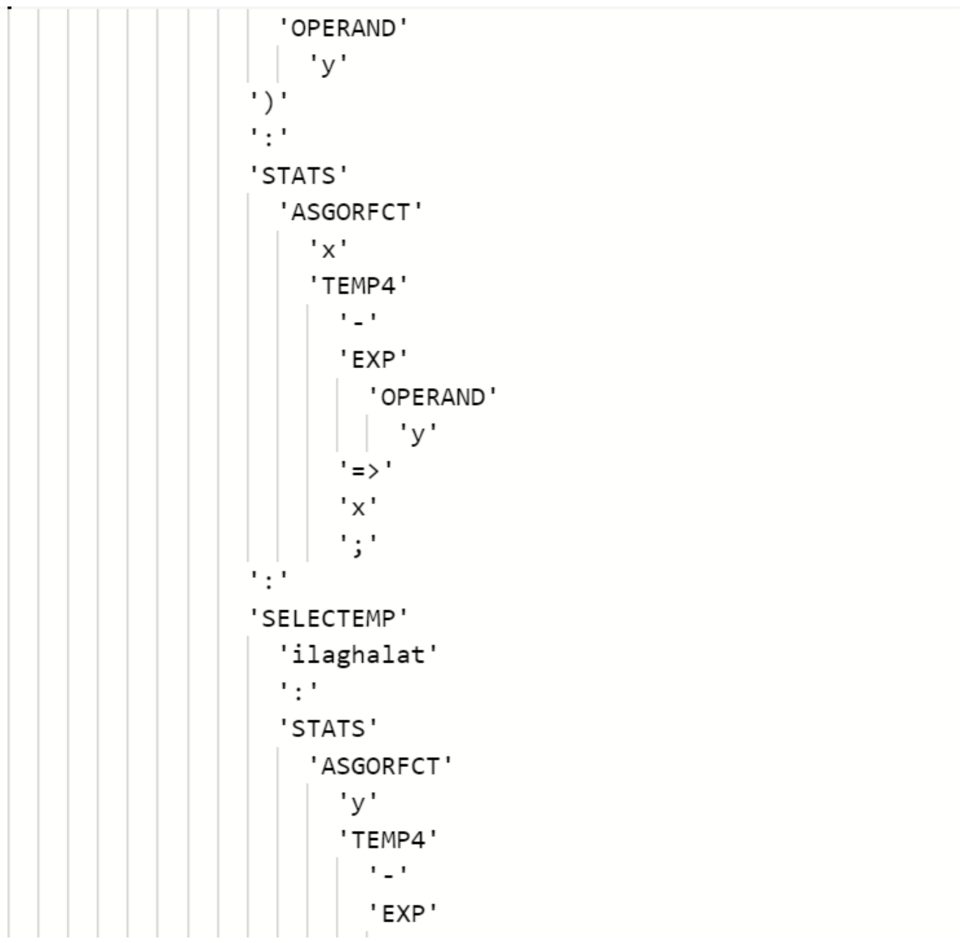
b. Parse Tree:



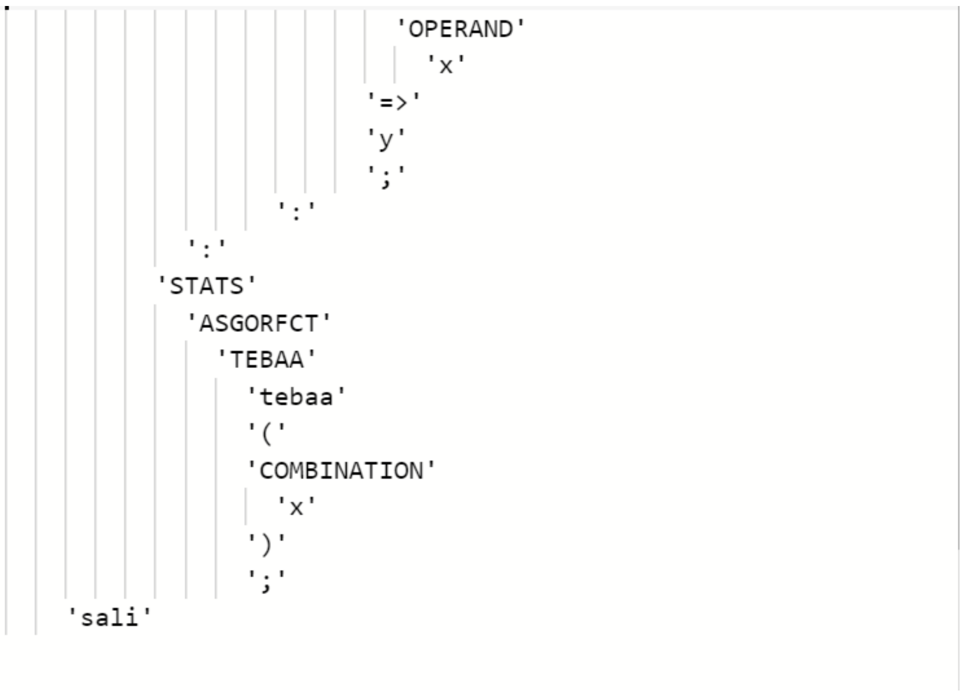
```
'PROGRAM'
  'DECLARATIONS'
    'VARTYPE'
      'STYPE'
        'aadad'
    'VARIABLES'
      'x'
      ','
      'y'
      ';'
  'MOUHIMA'
    'mouhima'
    '('
    ')'
    'bda'
    'STATS'
      'ASGORFCT'
        '6'
        'ASGOP'
          '=>'
          'x'
          ';'
      'STATS'
        'ASGORFCT'
          '2'
          'ASGOP'
```



```
            '=>'
            'y'
            ';'
        'STATS'
          'REPERT'
            'mahed'
            '('
            'CONDITION'
              'OPERAND'
                'x'
              'COMPEX'
                '!='
              'OPERAND'
                'y'
            ')'
            ':'
            'STATS'
              'SELECT'
                'ila'
                '('
                'CONDITION'
                  'OPERAND'
                    'x'
                  'COMPEX'
                    '>'
```



```
                  'OPERAND'
                    'y'
                ')'
                ':'
                'STATS'
                  'ASGORFCT'
                    'x'
                    'TEMP4'
                      '-'
                      'EXP'
                        'OPERAND'
                          'y'
                      '=>'
                      'x'
                      ';'
                ':'
                'SELECTEMP'
                  'ilaghalat'
                  ':'
                  'STATS'
                    'ASGORFCT'
                      'y'
                      'TEMP4'
                        '-'
                        'EXP'
```



```
                          'OPERAND'
                            'x'
                        '=>'
                        'y'
                        ';'
                  ':'
            ':'
          'STATS'
            'ASGORFCT'
              'TEBAA'
                'tebaa'
                '('
                'COMBINATION'
                  'x'
                ')'
                ';'
    'sali'
```

III. Static Semantics:

a. Type checking:

Type checking was done inside the parser class where we check for the appropriate types inside each acceptor depending on its case.

- **Types we checked:**
  - We check the types of the parameters of the function when a function is called, and that is done inside the IDs() acceptor.
  - We check for the type of the output of a function when it is called, and that is done inside OutpStore() acceptor.
  - We check for the types of assignment , and that is done inside the temp4() acceptor.
  - We check for the types of the sides of a condition, which is done inside operand() acceptor

- **Global variables:**

  - funct_parameters: function parameters is a global list where we keep track of all functions defined before the main. Whenever we meet a new function definition, we append the word ‘dalla’ to the list, followed by the function name. Each function name is followed by the word ‘output’ in case it returns something, which is followed by the type of the output. After that, types of the function parameters are inserted in the list in their specific order.
  - current_ID: a global variable that stores the name of the current ID that is recently read.
  - current_function: a global variable that keeps track of the function that is being called, so that we check its parameter types by comparing them to the information from the funct_parameters list.
  - outp: since the output type is the first one to be inserted after a function in the funct_parameters list, and it is the last one to be checked in a function call, we use the variable ‘outp’ to store the output type of the function being called so that we don’t need to scan over funct_parameters again.
  - qraortebaa: a global variable to distinguish between general functions and (qra or Tebaa) functions, so that we don’t go through definition checking and type checking in the case of these two special functions.
  - cond_type: a global variable where we keep track of the type of the left hand side of the condition to check it later with the other side of the condition (to make sure we are not comparing a string to an integer for example).

b. Definition Checking:

For definition checking we simply look for the identifier in the symbol table whenever we meet an ID, so if it is not in the symbol table an error is displayed in the screen with the details of the error and line where it occurs.

IV. AL code generation:

The Assembly Language code generation is done through the function generateAssembly(). This method is used to generate the equivalent assembly code for codes written in our language. It uses the token stream to do so.

V. How to run the code:

a. Files used and produced:

- Token.py: contains the Token class.
- main.py: the main which runs the lexer, parser, and code generator in that order.
- ALGenerator.py: contains the code used to generate the equivalent assembly code.
- Lexer.py: contains the lexer class and its methods.
- Parser.py: contains the parser class and its methods.
- generatedCode.txt: produced by the parser, contains the content of the leaves of the parse tree, which we formatted to look like the initial code. We use this file only for checking whether the tree is well constructed and contains all elements of the code.
- output.txt: produced by the lexer, contains tokens and other relative information.
- parsetree.txt: produced by the parser, contains the entire parse tree of the program.
- ALoutput.txt: produced by the ALGenerator, contains the generated output, later used by the assembler to generate machine code.
- code.txt: contains the input code.
- machinecode.txt: produced by the assembler, contains the equivalent machine code.
- MachineCodeWithSpaces.txt: produced by the assembler, contains the machine code but is pretty printed.

b. Manual:

Our code is written in two separate languages: C and Python. To run the code, first you have to put the code in the **code.txt** file, then run main.py which runs both the lexer, the parser and the code generator, produces output.txt which is the output of the lexer and contains the tokens and information about them, and produces parsetree.txt where the contents of the parse tree are printed, as well as generatedcode which prints its leaves only. Then the code generator produces the equivalent assembly language and prints it out to the ALassembly.txt file. All of these files should be in the same location.

We can then use the generated ALoutput.txt file to run the assembler code and then the interpreter. Both the code and the text files should be in the same location.

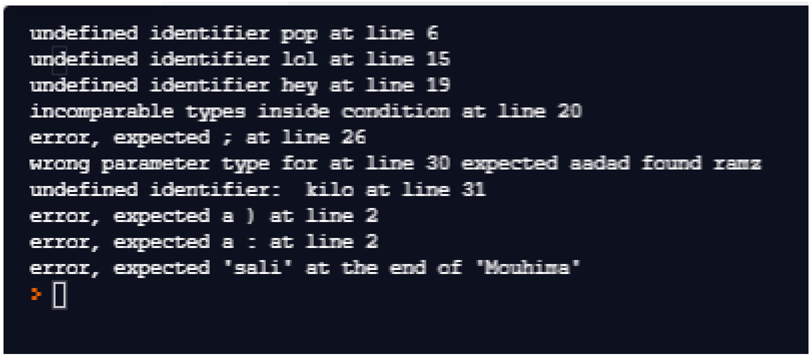
We used repl.it to run the codes that we have written, but the code can be run on any IDE provided that Python 3 is installed. The C code (assembler) can be run on any IDE provided that a GCC compiler is installed.

VI. Testing:

This code is to test any errors that we might stumble upon when we parse the program:

```
!!This code computes the area of a circle !!
tabita pi 3
dalla aadad computeArea(aadad radius)
bda
aadad result;
pi * pop * radius => result;
rejaa result;
sali
mouhima()
bda
aadad radius, flag, area;
ramz flag2;
ramz str[100];
'f' => flag2;
0 => lol;
$Please input the radius of the circle$ => str;
tebaa(str);
qra(radius);
ila (hey < 0):
mahed (flag = 'r' O flag2 = 'f'):
tebaa($Please make sure you input a positive number for the radius$);
qra(radius);
ila (radius >= 0):
1 => flag;
't' => flag2
:
:
:
```

ilaghalat:
computeArea(flag2) => area;
tebaa(kilo



```
undefined identifier pop at line 6
undefined identifier lol at line 15
undefined identifier hey at line 19
incomparable types inside condition at line 20
error, expected ; at line 26
wrong parameter type for at line 30 expected aadad found ramz
undefined identifier:  kilo at line 31
error, expected a ) at line 2
error, expected a : at line 2
error, expected 'sali' at the end of 'Mouhima'
> 
```

=> Since we are implementing an LL(1), we made use of its flexibility in error recovery, so the parser generally does not stop when it finds an error, but rather it keeps going through the code verifying the syntax to see whether there are other errors. However, the assembly code is generated only when the code is free from errors.

This code is to see the output of our project in case no errors are found:

```
aadad x, y;
mouhima()
bda
6 => x;
2 => y;
mahed (x != y):
ila (x > y):
x - y => x;
:
ilaghalat:
y - x => y;
:
:
tebaa(x);
sali
```