Phase 2

The second step to develop a compiler is to implement its parser. At this phase of the compiler project, you are going to make a parser which will parse the language described in the following pages which is called **Cool** with some changes. In order to accomplish this task, you will be taught to work with PGen. Note that this will be the **only** technology accepted from you for this part.

You can get the last edition of PGen from the link below:

https://github.com/Borjianamin98/PGen

Annotations Guide

Following annotations will be used in this phase description.

Annotation	Meaning
<a>	zero or one repetitions of a
a*	zero or more repetitions of a
a+	one or more repetitions of a
(a)	same as a
a | b	a or b
a b	a followed by b (concatenation)

Structure of a program

Cool programs are sets of classes. A class encapsulates the variables and procedures of a data type. A program must contain at least one class.

```
class Sort {
  int[] bubbleSort (int[] items) {
    let int i;
    let int j;
    let int n;
    n = len(items);
    for (i = 0 ; i < n- 1 ; i = i + 1)
      for (j = 0 ; j < n - i - 1 ; j = j + 1)
        if (items[j] > items[j+ 1]) then
          let int t;
          t = items[j];
          items[j] = items[j + 1];
          items[j + 1] = t;
        fi
      rof
    rof
    return items;
  }
}


class Main {
  let int[] items;
  void printArray () {
    let int i;
    print("sorted list:");
    for (i = 0 ; i < 100 ; i++)
      print (items[i]);
    rof
  }

  int main (){
    let int i;
    let int j;
    let int[] rawItems;
    rawItems = new Array(int, 100);
    for(i=0 ; i < 100 ; i = i+ 1)
      let int x;
      x = in_int();
      if (x == -1) then
        break;
      else
        rawItems[i] = x;
      fi
    rof
    let Sort s;
    items = s.bubbleSort(rawItems);
    printArray();
  }
}
```

Class

Every program in Cool is organized into classes. Classes are made of some declarations and methods that have access to variables in declarations.

Each class can be interpreted as the following form:

class className {
(declaration | method)*
}

Declarations and methods will be discussed in further sections.
"className" must be an identifier.

Types

Cool contains the following types:

Type	Description
int	32 bit integer number
real	32 bit real number
string	a sequence of characters
bool	can be true or false
self-defined types	programmer can define his own type by **class** keyword
void	

Variables (and declaration)

Variable definition in Cool can only be done at the beginning of each scope.

Every **declaration** must conform to the following syntax:

let Type or Array of Type ident (, ident)*;

Examples:

- **let int a;**
- **let real b, c;**

In Cool if a variable is a class instance or an array, access to this variable is like below syntax:

- for a field of a class instance: **variableName.fieldName**
- for an array, to access i-th element of it: **A[i]**
- variables that used as input arguments of methods does not have 'let' keyword. This will be discussed in "Methods" section.

Arrays

Arrays can be declared by following form and they can support all types mentioned in the previous section except "void".

let Type[] arrayName;

After declaring an array in the declaration part, you can create it with the instruction "**new Array (Type, n)**" with n > 0, and also, n can be expression which will be explained further.

Example:

```
let real[] arr;
arr = new Array(real, 5);
```

Some notes:

- The indices can only be integers, starting at 0.
- "arrayName" must be an identifier.
- Arrays can be passed as a parameter to a method (thus being handled as a call-of-reference scenario) or be the return statement of a function.
- Arrays support the **len(arrayName)** so that the number of elements of it can be obtained.

Strings

In Cool, we are able to declare strings, assign values to them and compare them. Regarding strings, please notice that:

- There exists a special method named **in_string()** which can take a string input from the console and assign to a string variable.
- A string as you have seen starts and ends with "
- Strings can be parameters or return values of a method.

- Strings support the **len(StringVariableName)** so that the number of elements of it can be obtained.

Methods

In Cool, all methods **can be defined only in a class** with the following syntax:

returnType methodName (Variable+, | ϵ) {

(Declaration | Statement)*

}

Examples:

```
int add(int a, int b) {
    let int result;
    result = a + b;
    return result;
}
void doNothing(){}
```

Some notes:

- ϵ means that the method can have no input arguments.
- returnType can be any of the types described in the "Types" section.
- Methods of other classed can only be called on an instance.
- “methodName” must be an identifier.
- Variable means declaring a variable like this:

  varType varName

  where **varType** can be all types described in the "Types" section **except void** and **varName** must be an identifier.

Assignment

In Cool, assignments are in the form of:

LeftValue = Expr

LeftValue can be one of the following:

- **ident** (like x which is a variable declared before)
- **ident.ident** (like a.b in which a is a class instance and b is a field declared in the class)
- **ident[Expr]** (for arrays)

Expr or Expression will be discussed later in this document.

Examples:

a = 3.14 + myInstance.number

myInstance.flag = true

a[2] = b[6]

Statement Section

in Cool, each statement block can be interpreted as the following form:

(Variable Declaration | Statement)*

Note that the Statement Section does **not** start with "{" and end with "}".

Variable Declaration was discussed in the "Variables" section. And statement will be discussed in next section.

Statements

- **assignment;** (discussed in the "Assignment" section. A statement can only be one assignment);
- **if (Expr) then statement <else statement> fi**
- **while (Expr) loop statement pool**
- **for (<assignment>; Expr; <assignment | Expr>) statement rof**
- **break;**
- **continue;**
- **print(Expr);**
- **return <Expr>;**
- **Statement Section**

Expressions

In Cool, expressions can appear in any of the following forms:

Expression	Example
ident	a (variable 'a' which has been declared before)
ident.ident	a.b (attribute 'b' of variable 'a' which was declared as an instance of a class)
ident [expr]	a[i + 2]
ident (Expr+, | ϵ)	f(x+1,y) (for calling a method that is belong to the current class)
ident.ident(Expr+, | ϵ)	a.f(x, y+1) (for calling a method of an instance of an another class)
in_int()	reading an integer from the input
in_string();	reading a string from the input
new Array(Type, Expression)	new Array(int, x-1)
Cast	(int) 2.5;
Constant	5 2.5 "hello" (a value of the types mentioned in the "**Variables**" section)

Obviously, any mathematical operation which involves two or more expressions as operands or unary operations (like not) are also considered as an expression:

Expression	Example	Expression	Example
Expr + Expr	a + b	**Expr <= Expr**	a <= b-2
Expr - Expr	a - 2	**Expr > Expr**	5 > b
Expr * Expr	a * b	**Expr >= Expr**	a >= 2
Expr / Expr	a / 3	**Expr == Expr**	a == b
(Expr)	(a + b)	**Expr != Expr**	a + 3 != c
Expr < Expr	a < b	**Expr || Expr**	a || b
!Expr	!a	**Expr && Expr**	a && b

Some Notes:

- Other operators that was mentioned in “Operators and Punctuations” section in Phase 1 can also be considered as operators between two Expression.

How to do this phase?

First, you should download PGen. Then, draw your parser graphs with it. After that choose “Export Full Parser” option in PGen. Before exporting full parser, it is better to check if your graphs have errors or not.

After generating full parser with PGen, you should have Lexical.java, Parser.java and CodeGenerator.java. You should connect your scanner (lexical) to your parser. Then, you should create a Main.java file and parse the input cool file. Cool files have “.cool” extension. You Main.java must have main method that writes the result of parsing your input cool file in output file that is in “outputPath” that will be given as an argument to your main.

If you choose java and syntax of input file has any errors, your main method should write “Syntax is wrong!” in the “outputPath”. If the syntax is ok and does not have any errors, “Syntax is correct!” should be written. For example, in right program of programs we mention below, condition of if statement has no parenthesis. So, right program syntax is wrong. If you choose python, then you should write a parser for the generated parse table. You can see generated java parser files and implement your python parser like it. Note that if you choose python, you should have a file named “main.py” that parse the cool input file and then writes “Syntax is correct!” for correct syntaxes and “Syntax is wrong!” for wrong syntaxes in the “outputPath”.

```
class Main {
  int main (){
    let int number;
    number = 10;
    print(number);
  }
}
```

```
Syntax is correct!
```

```
class Main {
  int main (){
    let int number;
    number = 10;
    if a == 10 then
        print(number);
    fi
  }
}
```

```
Syntax is wrong!
```

We run your files like below:

- For Java:

java <javaClassFile> --input <inputCoolFilePath> --output <outputFilePath> --table <tablePath>

for example: java Main –input ./test/test1.cool –output ./out/test1.out –table ./src/table.npt

- For Python:

python main.py --input <inputCoolFilePath> --output <outputFilePath> --table <tablePath>

for example: python main.py –input ./test/test1.cool –output ./out/test1.out –table ./src/table.npt

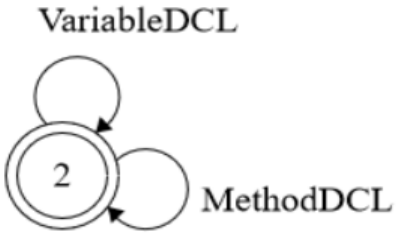
For example, “Main.java” should be like below:

```
public class Main {
    public static void main(String[] args) throws IOException {
        String inputCoolFilePath = "";
        String outputFilePath = "";
        String tablePath = "";
        if (args.length >= 6) {
            for (int i = 0; i < args.length; i++) {
                if (args[i].equals("--input")) {
                    inputCoolFilePath = args[i + 1];
                }
                if (args[i].equals("--output")) {
                    outputFilePath = args[i + 1];
                }
                if (args[i].equals("--table")) {
                    tablePath = args[i + 1];
                }
            }
        } else {
            System.out.println("Run like bellow:");
            System.out.println("java <javaClassFile> --input <inputCoolFilePath> --output
<outputFilePath> --table <tablePath>");
            return;
        }
        // inputCoolFilePath can be like this: ./test/test1.cool
        // outputFilePath can be like this: ./out/test1.txt
        // tablePath can be like this: ./src/table.npt

        // Make a new instance of your parser that reads scanner tokens
        // and then call "parse" method of your parser

        // write the result of parsing in the outputFilePath.
        // if the syntax is correct you should write "Syntax is correct!"
        // and if the syntax is wrong, you should write "Syntax is wrong!" in outputFilePath.
    }
```

Part of The Parser Graph (Syntax Diagram)

Program:

ClassDCL:



Notes

- The due date is Azar 11^{th}.
- You should connect your scanner program to parser program to pass tokens to your parser. Generated Parser.java file has a parse method. See the signature of it and then work with it.
- What you must upload is a zip file containing a ".pgs" file (your diagrams), ".prt" file (your parser table) and "Main.java" or "main.py" file that prints what we said before and other files like Parser.java.
- In the next phase, you should add semantic tokens to your graphs. At this phase, your graphs can have no semantic tokens! If you add semantic tokens in this phase, you may make less effort for next phase!
- This part of the project can be done in groups of two.