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 that will parse the language described in the following pages, which is called **Cool** with some changes. In order to accomplish this task, you can use PGen or Antler.

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 {
    func bubbleSort(int[] items) int[]{
        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]){
                    let int t;
                    t = items[j];
                    items[j] = items[j + 1];
                    items[j + 1] = t;
                }
            }
        }
        return items;
    }
}
class Main {
    let int[] items;
    func printArray() void{
        let int i;
        print("Sorted list: ");
        for (i in range(0,100)){
            print(items[i]);
        }

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

Class

Every program in Cool is organized into classes. Classes are made of 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)

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 the 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 are used as input arguments of methods does not have the 'let' keyword. This will be discussed in the "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 Type[n]**" with n > 0, and also, **n** can be an expression that will be explained further.

Example:

```
let real[] arr;

arr = new 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.

Range

The second way of creating an array is using the keyWord of "**range",** it creates a sequence of numbers, starting from 0 by default, and increments by 1 (by default), stops before a specified number.

range(start, stop, step)

start and step are optional and have a default value.

Example:

```
let int[] arr, arr2, arr3;

arr = range(5) //[0, 1, 2, 3, 4]

arr2 = range(2, 5) //[2, 3, 4]

arr3 = range(0, 10, 2) //[0, 2, 4, 6, 8]
```

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 **inputStr()** 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:

func methodName (Variable+, | ϵ) returnType {

(Declaration | Statement)*

}

Examples:

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

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 classes 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)* }

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

Statements

- **assignment;** (discussed in the "Assignment" section. A statement can only be one assignment);
- **if (Expr) statement <else statement>**
- **while (Expr) statement**
- **for (<assignment>; Expr; <assignment | Expr>) statement**
- **for (variable in Range) statement**
- **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)
inputInt()	reading an integer from the input
inputStr();	reading a string from the input
new type[expression]	new 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) is also considered 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 were mentioned in the " Operators and Punctuations " section in Phase 1 can also be considered as operators between two Expressions.

How to do this phase?

For this phase like the first phase, you can use Pgen with Java or Antler with Java or python.

The difference is that in PGen you should draw parser graphs but Antler generates parser from grammars.

If you choose PGen, download PGen. Then, draw your parser graphs with it. After that choose the “Export Full Parser” option in PGen. Before exporting the 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.

In both languages,your program should write the result of parsing your input cool file in the output file, If syntax has any errors, your output should be “Syntax is wrong!” and If the syntax is correct and doesn't have any errors, “Syntax is correct!” is the output. For example, in the right program that we mentioned below, the condition of if statement has no parenthesis. So, the right program syntax is wrong, but the left one has no problem.

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

```
Syntax is correct!
```

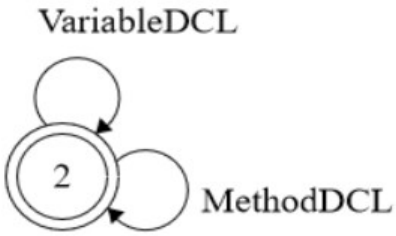
```
class Main{
    func main(){
      let int number;
      number = 10;
      if a == 10 {
        print(number);
      }
    }
}
```

```
Syntax is wrong!
```

Part of The Parser Graph (Syntax Diagram) In PGen

Program:

ClassDCL:



Part of Grammar in Antler

grammar Program;
Prog : ClassDCL (ClassDCL)*
ClassDCL : (VariableDCL | MethodDCL)*

Notes

- The due date is ordibehesht 23th.
- 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) or ".g4" (your grammers), ".prt" file (your parser table) and all the code including the main file that prints the required output.
- 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 the next phase!
- This part of the project can be done in groups of two.