Phase 2

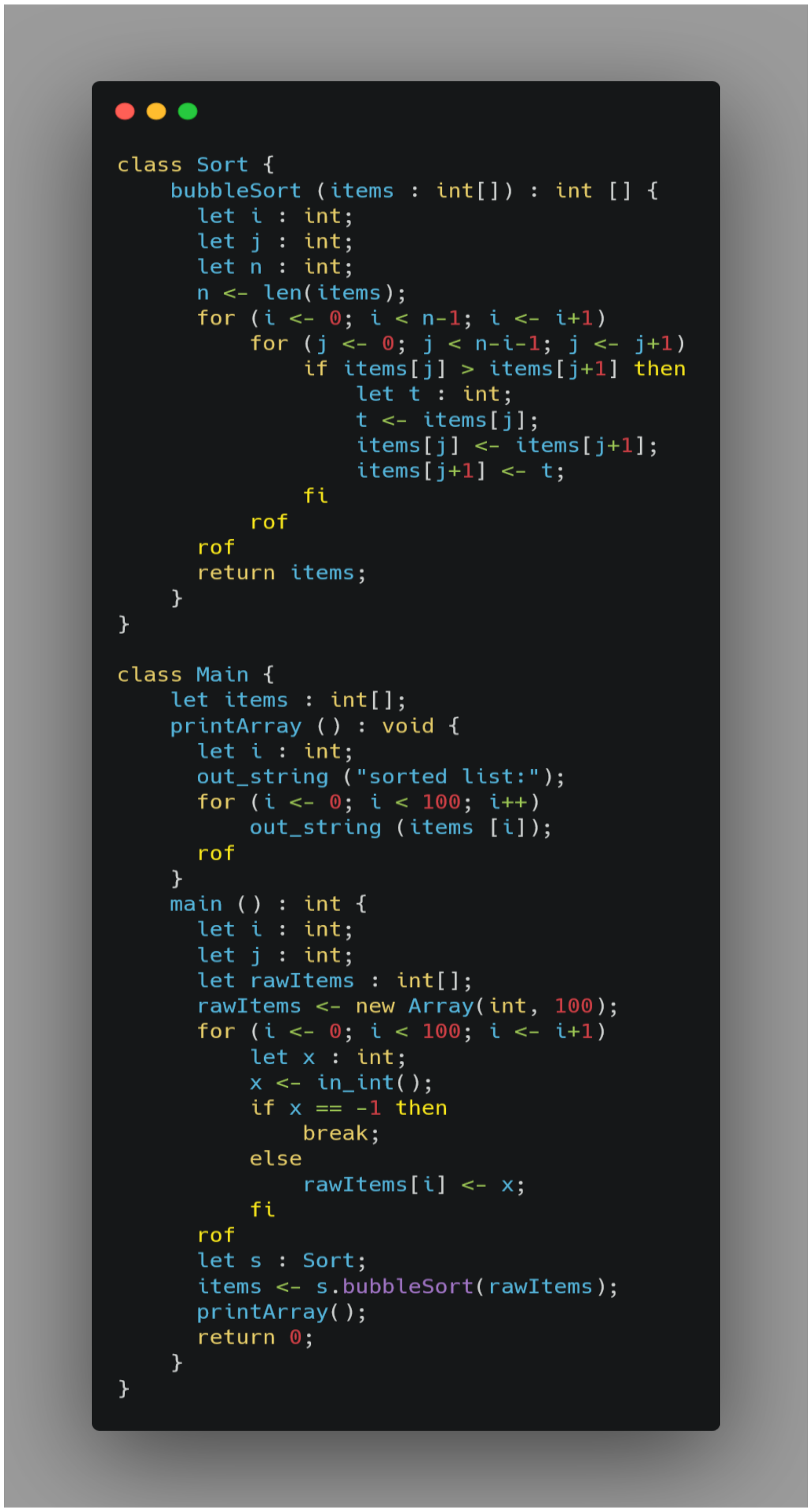
The second step to develop a compiler is to implement its parser. In this part of the project, you are going to make a parser which will parse the language described in the following pages which is called **Cool** with some changes. In order to accomplish this task, you will be taught to work with PGen.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

Structure of a program

Cool programs are sets of classes. A class encapsulates the variables and procedures of a data type.



```
class Sort {
    bubbleSort (items : int[]) : int [] {
      let i : int;
      let j : int;
      let n : int;
      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 t : int;
                  t <- items[j];
                  items[j] <- items[j+1];
                  items[j+1] <- t;
              fi
          rof
      rof
      return items;
    }
}

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

Consider that every program should have a class named **Main** which has a **main** method which will be considered as the starting point of execution of the program.

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*      arr <- new Array(real, 5);
    method*
}
(* means zero or more repetitions)
```

Declarations and methods will be discussed in further sections.

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

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

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

let ident: Type or Array of Type;

Example:

let a: int;

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]**

Arrays

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

let arrayName: Type[];

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 arr: real[];
arr <- new Array(real, 5);
```

Some notes:

- The indices can only be integers, starting at 0.
- 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(StringVarName)** 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:

methodName (Variable+, | ϵ) : returnType {

(Declaration + Statement)*

}

Some notes:

- + after Variable means one or more.
- | 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.
- Variable means declaring a variable like this:
  varName: varType
  where varType can be all types **except void**.

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, Expr)	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:

- All operators that was mentioned in “Operators and Punctuations” section in Phase 1 is considered as an Expression.

Assignment

In Cool, assignments are in the form of:

LeftValue <- Expr;

Expressions were mentioned in the previous part.

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 an instance of a class and b is a field declared in that class)
- **ident[Expr]** (for arrays)

Statement Section

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

(Variable Declaration + Statement)*

(* means zero or more repetitions)

Note that the Statement Block does not need to starts with "{" and ends with "}".

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

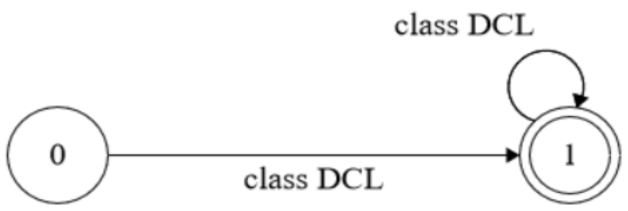
Statements

- **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;**
- **out_string(Expr);**
  This method prints Expr that must be a string.
- **out_int(Expr);**
  This method prints Expr that must be an int.
- **return <Expr>;**
- **Statement Section**

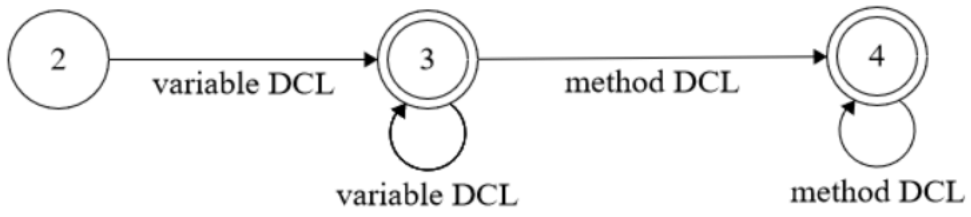
(<> means one or zero repetitions)

Part of The Parser Graph (Syntax Diagram)

Program:



class DCL:



Notes

- The due date is Farvardin 31th.
- You may need to make some changes in your scanner program so that it can pass tokens to your parser.
- What you must upload is a zip file containing a .pgs file (your diagrams), .prt file (your parser table).
- This part of the project can be done in groups of two.