A. Define your own Toy language (5 points)

For your project, you will define your own toy programming language. You are free to choose and define the syntax and semantics of your language. Your language many be based on an existing language, or it may derive from a combination of multiple existing languages; however, it cannot be an exact simple subset of an existing language.

At the minimum, your language must include the following constructs:

1) Data Types

a) Primitive types covering: integer/real/string/boolean values
b) User-defined record types (e.g., struct in C, record in Pascal)
c) Constants – i.e., immutable identifiers
d) Variables – i.e., mutable identifiers
e) Arrays of the above types

2) Operators and Expressions

a) Basic arithmetic expressions with operator precedence and associativity, covering the following operators:
   - Addition / Subtraction
   - Multiplication / Division
   - Unary negation operator
   - String concatenation

b) Basic relational expressions with operator precedence and associativity, covering the following operators:
   - Equals / Greater than / less than

c) Basic logic expressions with operator precedence and associativity, covering the following operators:
   - AND / OR / NOT

3) Language Constructs

a) Assignment statement
b) Selection: decision branching; choose between 2 or more alternative paths
c) Repetition: looping; i.e., repeating a piece of code multiple times in a row
d) Function and Procedures calls
e) Comments

4) Built-in Functionality

You are free to implement the below using their own grammar constructs, or you can implement them as functions or, in the case of the command-line parameters, as specialized function arguments. If the latter, there would be no effect on your grammar.

a) Accept command-line inputs
b) Print formatted text to standard output
c) Read formatted text from standard input

Deliverable:

Your first deliverable is a valid ANTLR4 g4 grammar file, along with a valid source text file for your language that can be tokenized and parsed using the ANTLR4 tools set.

Note, you should provide examples that cover each of the language construct, operators and data types.

***YourLanguageName*.g4** – *the grammar file for your language*
Program.source – *sample source file to parse*

B. Perform Semantic Analysis and Type-Checking (15 points)

Build and run a semantic type checker that will type check your program and raise the following issues, if they occur. Note, depending on your language design, not all of the below errors are relevant, or they may be relevant only in certain cases.

`UNDECLARED_IDENTIFIER ("Undeclared identifier")`
Error occurs when trying to use an identifier (variable, function, class, etc.) that has not been defined or declared before in the program

`REDECLARED_IDENTIFIER ("Redeclared identifier")`
Error occurs when trying to declare or define an identifier (variable, function, class, etc.) that has already been declared or defined before in the program

`INVALID_VARIABLE ("Invalid variable")`
Error occurs when using an invalid or illegal name for a variable in your program

`NAME_MUST_BE_PROCEDURE ("Must be a procedure name")`
Errors occurs when trying to use a variable, object, or other non-function or non-procedure object as if it were a function.

`INVALID_TYPE ("Invalid type")`
Error occurs when trying to declare a type that is does not exist in the language.

`TYPE_MISMATCH ("Type mismatch")`
Error occurs when trying to perform an operation on two values of different types that are not compatible with each other

`INVALID_OPERATOR ("Invalid operator")`
Error occurs when using an operator that is not supported by the type of the operands.

`INCOMPATIBLE_ASSIGNMENT ("Incompatible assignment")`
Error occurs when trying to assign a value to a variable that is not compatible with the variable's type

`INCOMPATIBLE_COMPARISON ("Incompatible comparison")`
Error occurs when trying to compare values or variables of different data types that cannot be compared in a meaningful way

```
TYPE_MUST_BE_INTEGER ("Datatype must be integer")
TYPE_MUST_BE_NUMERIC ("Datatype must be integer or real")
TYPE_MUST_BE_BOOLEAN ("Datatype must be boolean")
TYPE_MUST_BE_STRING  ("Datatype must be string")
```

Error occurs when a value or variable is expected to be of a certain data type, but it is not. Typically occurs when a function requires certain input type.

```
ARGUMENT_COUNT_MISMATCH ("Invalid number of arguments")
```

Error occurs when a function or method is called with the incorrect number of arguments

```
INVALID_RETURN_TYPE ("Invalid function return type")
```

Error occurs when a function or method returns a value that is of a different data type than the type specified in its return type declaration

```
TOO_MANY_SUBSCRIPTS ("Too many subscripts")
```

Error occurs when an array or a data structure is accessed with more subscripts than it has dimensions or levels.

```
INVALID_FIELD ("Invalid field");
```

Error occurs when a program or script attempts to access a field or property of an object that does not exist or is invalid.

Note, as you implement more features in your language, you will be required to perform the relevant semantic checks.

You are allowed to work with your randomly-assigned partner on this phase of the compiler.

C. Code Generation – Generate Jasmin Code (10 points)

The next step of your project is to write the code generator to generate the Jasmin code for programs written in your language. Follow the patterns that you used in the Pascal compiler from the homework assignments to implement the code generator.
You are allowed to work with your randomly-assigned partner on this phase of the compiler.

1) You must submit a working Hangman application written in your own language. The application should function as follows:

   - Select a 5-10 letter secret word and hardcode it as a constant in your code. Use only lowercase letters.

   - Print a welcome message along with the word placeholder and prompt the user to enter a letter, as shown below:

```
$> java hangman

Welcome. Try and guess my secret word: _ _ _ _ _
Select a letter:
```

   - Accept a letter as input from the user.

   - If the letter appears in the word, output as below:

```
$> java hangman

Welcome. Try and guess my secret word: _ _ _ _ _
Select a letter: a
The letter 'a' appears 2 time(s): a _ _ _ a
Select a letter:
```

   - If the letter does not appear in the word, the user gets a strike and you output as below:

```
Select a letter: b
The letter 'b' does not appear. You have 1 strike(s): a _ _ _ a
Select a letter:
```

   - If the user guesses a letter that was already guessed (rightly or wrongly), output as below:

```
Select a letter: b
The letter 'b' does not appear. You have 1 strike(s): a _ _ _ a
Select a letter: b
'b' was already guessed: a _ _ _ a
Select a letter:
```

- The game ends if the user gets three strikes. Output as below and exit the program. The guess percentage is calculated as (number of correct guesses) / (total guesses).

```
Select a letter: b
The letter 'b' does not appear. You have 3 strike(s)
Your guess percentage was: 25.00%
You lose. Goodbye!
$>
```

- If the user guesses all the letters in the word, they win and the game ends. Output as below and exit the program:

```
Select a letter: i
The letter 'i' appears 1 time(s): a k i _ a
Select a letter:
The letter 'v' appears 1 time(s): a k i v a
Your guess percentage was 66.67%
You won with 6 guesses. Goodbye!
$>
```

D. Additional Language Constructs and Optimizations (4 points each, Max 20 points)

Extend your grammar and compiler to implement additional language features and enhance it by implementing optimizations. Below is a list of features and optimizations from which you can choose. If you want to implement one that is not listed here, suggest it to me and, if I approve, you may implement it.

You must implement at least two optimizations and two language features.

Language Features

1. ***"C-style" static variables*** – variables that are declared within a function or a block of code using the **static** keyword. These variables are only accessible in the scope in which they are declared, yet remain in memory for the entire duration of the program and retain their values across function calls.

2. ***Nested subroutines*** – allow the definition of a function or subroutine inside of another function or subroutine that has access to the variables and parameters of the enclosing function.

3. ***Pointer types*** – allow declarations of pointers, or references to your supported types. Ensure proper type checking and provide the ability to dereference the pointer. You should also have the ability to assign one pointer to another.

4. ***Iterators*** – provide the ability to create objects or data structures that allow you to traverse through a collection of elements or data items one at a time.

5. ***Immutable functions*** – allow for explicitly declaring functions as immutable and enforce immutability during static analysis, where possible and at runtime, if necessary.

6. ***High-order functions*** – functions that take other functions as arguments or return functions as values, or both.

7. ***Generic<T> functions*** – provide the ability to declare and use generic functions that can work with multiple types of data.

8. Another language feature of your choice, and which I approve.

Optimizations

1. ***Function inlining*** – for small functions of less than 3 statements, replace the function call by inserting the actual code of the function directly into the calling code.

2. ***Short-circuit evaluation*** – only evaluate the second argument of a logical operator if the first argument does not determine the overall value of the expression.

3. ***Constant folding*** – simplify expressions that involve two constants by evaluating the expressions at compile time and replacing the expressions with the results of the of the computation.

4. ***Common Subexpression Elimination –*** identify subexpressions that occur more than once in a program and eliminate the redundant computations by replacing them with a single computation.

5. ***Dead code elimination*** – detect and remove code that can never be executed.

6. ***Loop-invariant code motion –*** move code outside of a loop if it does not depend on the loop variables.

For optimizations, your compiler should accept an optional **`-o`** *command line argument, which, when specified, will instruct your compiler to run the optimizations. If the argument is not provided, then no optimizations will be run.*

Submission Instructions

Check in your working code to the YU GitHub repository.

- Your code for the project should be rooted in the **`$MYGIT/Compilers/assignments/SemesterProject`** directory.
- Your main compiler Python file, which I will be running, should be in that root directory and should be called **`compiler3645.py`**
- Your program source text files should be placed in a subdirectory called **`programs`** and should have the **`.pgm`** extension.
- I intend to compile your programs by executing the following from within your **`SemesterProject`** root directory. Please make sure your setup is correct and works for you; if it doesn't, it won't work for me either:
    - **`python3 compiler3645.py programs/`*`xxx`*`.pgm`**

1. Submission for Semantic Analysis and Type-Checking (B)

- For each of the errors listed in section B, provide a one- or two-line program that demonstrates code that fails the semantic check.
- The output message to the command line from the compiling the program must contain the text of the error message listed in the assignment instructions. *Please check your spelling*.
- If a given error is not relevant to your language, have your program simply print a message to the output explaining why it is not relevant.
- Place all your program files for this stage in the **`SemesterProject/programs/semantics`** folder and ensure that they all have the **`.pgm`** extension.
- I intend to compile your programs by executing the following from within your **`SemesterProject`** root directory:
    - **`python3 compiler3645.py programs/semantics/`*`xxx`*`.pgm`**

2. Submission for Code Gemneration (C)

You must submit code that covers the minimum language requirements from the first stage of the project. If you are able to do that within your hangman program, then you are done. If not, then you must submit an additional program or programs of your choice to cover any language construct that you missed.

- The Hangman program should be named **`hangman.pgm`** and placed in the **`programs`** folder.
- You are free to name other programs as you like, but they must have the **`.pgm`** extension
- I intend to compile your programs into Jasmin by executing the following from within your **`SemesterProject`** root directory:
    - **`python3 compiler3645.py programs/hangman.pgm`**
- This is expected to produce a file named **`hangman.j`** in the *current working directory*, which, when compiled with the Jasmin compiler, will produce the **`hangman.class`** class file.
- The same will hold true for the other programs that you may need to submit.

3. Submission for enhancements and optimizations (D)

a) For language feature enhancements, you will submit additional programs that will run in the same way that I ran your previous programs. These programs should demonstrate the functionality and/or semantics you are implementing.

    - Place all your program files for this stage in the **`SemesterProject/programs/features`** folder and ensure that they all have the **`.pgm`** extension.

b) For optimizations, you should not need to submit any additional programs. I intend to compile your programs both with and without the **`-o`** command line argument and will compare the Jasmin code that is produced.