Preparation Questionnaire-I

1. **Which of the available paradigmatic options did you choose for your tetris-engine programming language design? (We discussed Python, C, C++, Java as options. Or any other.) Describe in short (no more than 2-3 sentences).**

   We chose an imperative programming paradigm. Imperative programming is a programming method describing sequences of operations in order to change the current state of the program. An imperative code explains how to do things (it contains logic, loops, conditions, etc.). It includes features of Procedural Programming paradigm and Object Oriented programming. Advantages of imperative programming :

- Imperative programming contains logics, loops, conditions etc. which will be utilised by the programmer in his code.
- It is also easier to debug an imperative program because you manage all the steps: tell the program what you want, and how to
  get it.

2. **Is your designed language to be procedural, object-oriented, functional, or a combination? Describe its paradigm in your own words.**

What is Procedural Programming? [Definition]

Fundamentally, the procedural code is the one that directly instructs a device on how to finish a task in logical steps. This paradigm uses a linear top-down approach and treats data and procedures as two different entities. Based on the concept of a procedure call, Procedural Programming divides the program into procedures, which are also known as routines or functions, which the programmer can define individually.

Key Features of Procedural Programming

The key features of procedural programming are: Predefined functions from libraries, Local Variable, Global Variable, Modularity and Parameter Passing.

Advantages of Procedural Programming

- The code can be reused in different parts of the program, without the need to copy it

- It allows the programmer to divide the main program into sub-programs/functions. For eg. The programmer can initialise a tetromino-with its shape, speed, colour etc and set game features like scoring system, level upgradation etc. into different functions. Increases readability of the program.

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object-oriented programming are –

- Bottom-up approach in program design
- Programs organised around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes
- Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

3. **Which are the basic units composing a program in your language? Are they procedures, are they functions, or just blocks and statements? Describe in your own language.**

   In computer programming, a procedure is an independent code module that fulfils some concrete task and is referenced within a larger body of source code. This kind of code item can also be called a function or a sub-routine. The fundamental role of a procedure is to offer a single point of reference for some small goal or task that the developer or programmer can trigger by invoking the procedure itself.

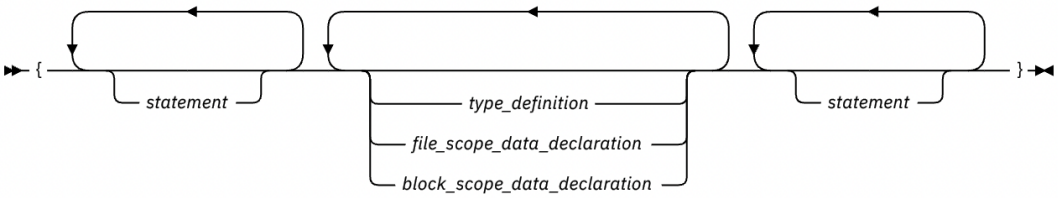
   A procedure may also be referred to as a function, subroutine, routine, method or subprogram. So something like rotateLeft would be implemented as a procedure.

   By using a procedure, a programmer can make a program do that one thing in many different ways, using different parameters and sets of data, simply by invoking the procedure with different variables attached.

We also give the ability to embed statements in blocks using '{' and '}'punctuations. This helps to resolve scope(as mentioned in the textbook) as it defines a local scope but also improves readability.

A block statement, or compound statement, lets you group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can use a block wherever a single statement is allowed.

Block statement syntax



A block defines a local scope. If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Overall structure of a program written in our language would have some preprocessing directives like imports. The remaining logic will largely be implemented either using OOP paradigms or largely as procedures and functions.

4. **Basic units of your program structure may be spread across files and other aggregates. Which component in the language processor pipeline finally brings them all together? How?**

We allow the programmer to use macros and imports and the pre-processor.

The #bring <filename> directive of our language works by directing the preprocessor to scan the specified file as input before continuing with the rest of the current file. The output code file from the preprocessor stage contains the output that has already been generated so far, followed by the output resulting from the current call to included file, followed by the output that comes from the remaining code of source file.

5. **Have you designed the scanner? If you submit the deliverables on or before 7 March 2022, you get straight 5 to 10% credit, of the total 10% kept for this phase. Describe your design, if yes. And take the next questionnaire.**

Design of Scanner : The Scanner reads the input character of the source program, written in our own designed language, groups and identifies them into lexemes and produces as output an instance of LexToken for each lexeme in the source program. On recognising a pattern in the source code, the scanner takes certain action based on the pattern-action pair. Upon the generation of tokens, they are sent to the parser for syntax analysis.

We designed our scanner by specifying the lemexe patterns to a lexical analyser generator, PLY. PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. lex.py provides an external interface in the form of a token() function that returns the next valid token on the input stream. Yacc.py, of the Python Package PLY, calls this repeatedly to retrieve tokens and invoke grammar rules.

The output, an instance of LexToken, contains fields that are initialized to attributes of identified lexeme. This object has attributes of t.type which is the token type (as a string), t.value which is the lexeme (the actual text matched), t.lineno which is the current line number, and t.lexpos which is the position of the token relative to the beginning of the input text.

To build the scanner, the function lex.lex() of Ply Python package is used. This function uses Python reflection (or introspection) to read the regular expression rules to build the scanner. Once the scanner has been built, two methods can be used to control and use the scanner.

- lexer.input(data): Reset the lexer and store a new input string.
- lexer.token(). Return the next token. Returns a special LexToken instance on success or None if the end of the input text has been reached.

Preparation Questionnaire-II

1. **Is your lexicon finite or infinite? If finite, what is the order (several, dozens, hundreds, thousands, etc.)?**

   Lexicon refers to the complete set of meaningful units in a language. Our lexicon is infinite, since we may have a large / uncountable number of lexemes with token type identifiers. Since we do not have any upper bound on the length of the identifier, we may have infinite lexemes.

2. **Have you decided elementary operations and corresponding keywords? Is there a one-to-one correspondence between them?**

   Yes, in our implementation,the elementary operations provide a high level of abstraction to the programmer. They constitute operations like rotateLeft, rotateRight and allow the programmer to develop the game without worrying about low-level details.

   There is a one-to-one-correspondence between elementary operations and corresponding keywords. The LexToken object returned to the parser has Token Type specific to the elementary operation, to which the lexeme corresponds.

3. **In your lexicon, how many patterns are devoid of any regexp operators other than concatenation? How many of these are punctuation, and how many keywords?**

   In our lexicon, the regular expression for lexemes corresponding to reserved/special keywords, elementary operations and punctuations are devoid of any regexp operators other than concatenation.

4. **For token types that encode more than one lexeme, have you hardcoded the lexeme variety in your scanner implementation? To understand the question, compare Figures 3.13 and 3.14 in the dragon book.**

   For certain Token classes that could include more than one lexeme, the lexeme variety that classify into these classes are limited and are hardcoded to belong from only a set of possible lexemes.

   However, there are Token Classes like ‘Number’ or ‘Identifier’ that could take infinite possible lexemes that match its regular expression, thereby not hardcoded to a finite number of lexemes.

Alternate: Since we have not explicitly mentioned Token rules for lexemes like >=, <=, <, >, =, <>, etc, and rather classified as RELOP Token Class, there exists more than one lexemes that map to the same Token Class, here RELOP. We have not hardcoded the lexeme variety to particular and respective keywords. Rather, during the action for these lexeme-patterns, the Token type is updated, according to the lexeme text, by the scanner.

Extras: Refer to the example.py file below :

```
relop_attr = {'>':'GT','<':'LT','=':'EQ'}  #set of possible
lexemes

tokens = [ 'NUMBER', 'PLUS', 'MINUS', 'RELOPS'] +
list(relop_attr.values())

# Regular expression rules for simple tokens
t_PLUS     = r'\+'
t_MINUS    = r'-'

def t_RELOPS(t):
    r'[>|<|=]'
    t.type = relop_attr.get(t.value,'RELOPS')
    return t
```

We have used a rule function for defining regular expression rule. The function always takes a single argument which is an instance of LexToken. This object has attributes of type which is the token type (as a string), value which is the lexeme (the actual text matched), lineno which is the current line number, and lexpos which is the position of the token relative to the beginning of the input text. By default, type is set to the "token class" following the t_ prefix. The action function might update the token class of the lexeme. The resulting token should be returned. If no value is returned by the action function, the token is discarded and the next token is read.

---

5. **For token types that encode too many lexemes, how do you handle the lexeme variety in your scanner implementation? This situation is like figure 3.14 in the dragon book.**

Token Classes/Types like [number, identifier}, which can have more than one lexeme match their pattern, are mentioned in a Token Class/Type list.

For special lexemes, like ‘if’, ‘else’, that also match the pattern for identifier, however are used for other purposes in our program language, we maintain a predefined list of pairs of these special lexemes and their respective intended Token Classes. We add this list of other Token Classes to the Token Class/Type list which we had defined earlier.

To handle these reserved keywords / special lexemes like ‘if’, ‘else’, we have written a single rule (Rule as Function) to first match it to an identifier and then, within this function, we do a special name lookup of this particular lexeme in the predefined list of special lexemes / reserved keyword list. If the lexeme is found in the predefined list, we update the token class/type. This approach greatly reduces the number of regular expression rules that we would have explicitly written otherwise, and is likely to make things a little faster.

We can have multiple other predefined lists and a similar approach can be used to handle punctuation lexemes, relation operator lexemes, operator lexemes, whitespace lexemes, function call lexemes specific to our game, etc.

Let’s say we have the list of pairs reserved words, that we encounter in our language, and their intended Token Class/type defined as follows:

```
reserved = {
    'if' : 'IF',
    'then' : 'THEN',
    'else' : 'ELSE',
    ...
}
// function to lookup for lexeme
def t_ID(t):
     r'[a-zA-Z_][a-zA-Z_0-9]*'
     t.type = reserved.get(t.value,'ID')    # Check for reserved
words
```

```
    return t
```

Extra Question: Why should we avoid writing individual rules for reserved words / special lexemes?

1) *It enables more efficient scanning and identification of token class, as it will have to check with less rules*
2) *Many times, these additional rules will be triggered for identifiers that include those words as a prefix.*

   Example :

   t_FOR   = r'for'

   t_PRINT = r'print'

   These rules will be triggered for 'forget' and 'printed', instead of t_IDENTIFIER that must have been triggered.

6. **Do you have an overlap between token types? That means, are there lexemes matching more than one pattern? If so, how is the ambiguity resolved, and which token types are having precedence?**

- Yes. A lexeme could overlap between two different types of token. For eg. The 'else' keyword, if we go by the ply implementation, occurs in the list of 'reserved' words and also matches the pattern for an identifier.

<u>Precedence of regular expression rules:</u>

The regular expression rules can be defined in two ways in PLY:

1. Defining regular expression rule as a function
2. Defining regular expression rule as pair of token class and string(represents the regex)

When building the master regular expression, rules are added in the following order and in-priority as follows:

- All tokens defined by functions are added in the same order as they appear in the lexer file
- Tokens defined by strings are added next by sorting them in order of decreasing regular expression length (longer expressions are added first)

In general practice, whenever there is a possibility of ambiguity and we want a particular token to have higher priority, we define the regular expression rule using function.

7. **If you change the order in which pattern-action pairs are given in your scanner implementation, will it change its behaviour? Will it change the lexicon definitions? Will it require major changes in the interface between the scanner and the parser? Why?**

As per the different ways of writing regular expressions for token class, and their rules for precedence, any change in the order in which token regex pairs are declared in our implementation may change its behaviour.

For regular expression rules of token type defined as strings, it does not change the behaviour since these rules are added by sorting them in order of decreasing regular expression length (longer expressions are added first).

For regular expression rule of token type defined as functions, changing the order in which the pattern-action pair are written in lexer file, changes the behaviour since they are added in the same order as they appear in the lexer file

For example:

```
def t_A(t):
     r’[ONE]’
     return t
def t_B(t):
     r’[ONE][ONE]*’
     return t
```

For lexeme “ONE” found in the source program, depending on the order of t_A and t_B, the behaviour changes.