CS 315 Theory of Automata
Final Project
Deadline: 17th May 2020 11:55 pm

One of the most common thoughts that any programmer has is "Why does this language have to be written like this?". You might question why C++ uses semicolons when a full stop is more intuitive. Perhaps you want a language where you can use Urdu phrases, for example `karo a < 10 jabtak ...` to emulate a do-while loop. Or you might want to create an esoteric language to compete with whitespace (Figure. 1 shows how hello world is written in whitespace). Maybe, you want to attain better understanding of how an interpreter operates. The solution to all these problems is simple: **Make your own programming language.**

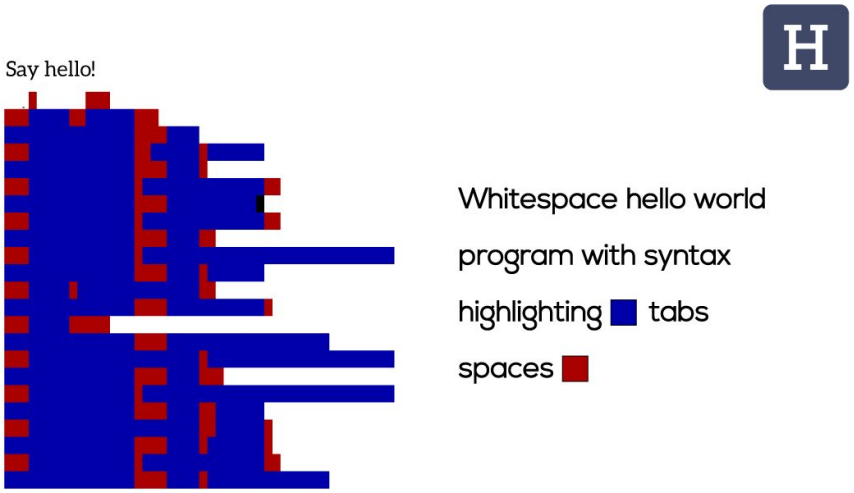
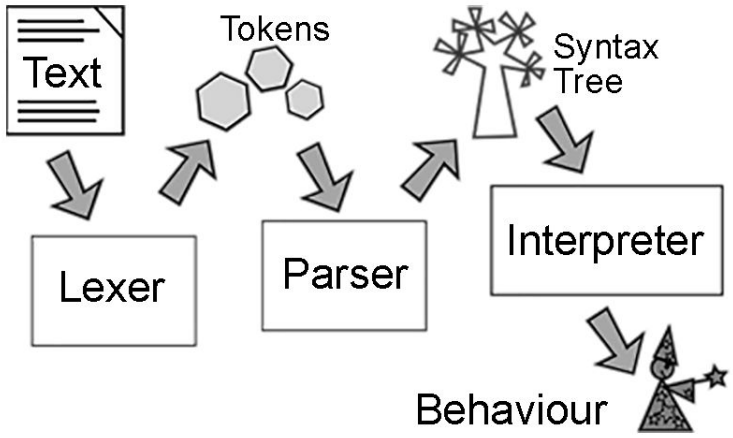


Fig. 1. Source

That is exactly what we are going to do in this project. We will give you the opportunity to design your own language which is filled with features that you want and has the syntax that you want. You want a language whose syntax is a combination of various languages? That's up to you. You want to come up with a brand new syntax? That's up to you. That language is called YAPL (Yet Another Programming Language) and this project affords each one of you the opportunity to design your own YAPL. After finishing this project, you should be able to use your language in other CS courses ;)

Now you might be wondering "How do I design a YAPL?". The answer to this is three different components: **lexer**, **parser** and **interpreter**. First of all, you need to make a lexer. A lexer basically takes a sequence of characters (ex. Your high level code) as input and outputs a sequence of tokens. Next up is the parser. The parser takes these tokens and outputs a parse tree

where each branch is filled with your expressions or statements. Finally, you have the interpreter which goes through each branch and evaluates it.



Tool Introduction:

PLY consists of two separate modules; **lex.py** and **yacc.py**, both of which are found in the Python package called ply. (Use pip to install it!)

The lex.py module is used to break input text into a collection of tokens specified by a collection of regular expression rules.

yacc.py is used to recognize language syntax that has been specified in the form of a context free grammar. yacc.py provides most of the features you expect including extensive error checking, grammar validation, support for empty productions, error tokens, and ambiguity resolution via precedence rules.

The two tools are meant to work together. Specifically, 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 calls this repeatedly to retrieve tokens and invoke grammar rules.

Rules and Guidelines:

- Implement a **lexer**, **parser**, and **interpreter** of your YAPL in PLY library (Use ply.lex for tokenizing and ply.yacc for parsing) and execute the test cases provided to you.
- The project is divided up into three parts (The divisions are explained under Task Breakdown). Each student has to implement all constructs in the compulsory (**Easy**) part. Each student will be allotted a single construct in the **Medium** and **Hard** parts and they have to attempt only the construct (in Medium and Hard part) that is assigned to them.
- Find the Medium and Hard programming constructs allotted to you using this link.
- Your language should be able to handle any code which uses the functionalities specified in the given tasks, since you will be marked on new test cases after submission.
- You have been provided test cases at the end of this document. These test cases (with some changes) will be used to evaluate your project. Therefore, you are supposed to submit .txt files for each test case (for the constructs that you have been allotted) as explained in the submission format at the end of the document.
- You should implement maximum error handling that a language provides.
- Timely uploading to LMS is YOUR RESPONSIBILITY. **Do not wait for the last moment**. We will only accept LMS submitted assignments and directly run the test cases on our machine.
- Plagiarism will be detected using multiple tools and forwarded directly to the DC.

Tips and Tricks:

Disclaimer: This is all up to you! These tips are completely optional and just for guidance.

- It will be easier if you use a stricter syntax, such as semicolons for the end of a statement and braces to limit the scope of blocks instead of indentation.
- Save error handling for the end, focus on the core functionality first!
- Refer to the test cases for clearing any confusions regarding the task descriptions.

Tasks Breakdown: (Total Marks: 20)

Compulsory (Easy): (Marks: 6)

Variables (2):

a. Declaration, assignment, access
b. Static-typing (types restricted to: int, double, char, string, bool)
c. Initialisation and declaration of a variable with the name of a pre existing variable should generate an error.

Expressions (3):

a. Numerical Operators: (+ , - , / , * , ^, % , ++, --)
b. Logical Operators (<, >, <=,>=, !=, ==, NOT, AND, OR)
c. Nested parentheses
d. Type (e.g. for String + Int) and division by 0 errors

Standard Output (1)

a. Single object is printed with a line break.
b. Multiple objects separated by a delimiter (e.g. comma as in Python) are printed with spaces in between and a line break at the end.

Medium: (Marks: 6)

Attempt only the one allotted to you:

If-elseif-else statements:

a. Can be nested
b. Can be an if statement, or just an if-elseif-elseif-...else or if-elseif or an if-else

Do-While Loops:

a. Can be nested

For Loops:

a. Can be nested

Hard: (Marks: 8)

Attempt only the one allotted to you:

List:

a. Declaration, assignment, access
b. Access outside the list-size should also give an error (e.g. Index Out of Bounds)
c. list.pop(index of item to remove) // list.pop(0) removes the head of the list and returns it
d. list.push(value) // appends the value at the end of the list
e. list.index(index) // returns the value at that index

f. list.slice(start, end) // end index excluded e.g. [6, 3, 1, 5, 2].slice(1, 4) returns [3, 1, 5]

Structs:

a. Define a struct
b. Create an object with the defined struct type
c. Access the variables/attributes declared inside the struct and assign values
d. AttributeError when a non-existing attribute is accessed

Required Test Cases with Outputs:

All the given test cases with outputs have been provided in a pseudo-code-like syntax for your ease of understanding. Therefore, you do not need to abide by or stay limited to any of the given keywords or tokens in general as long as it is understandable, but just keep in mind the features of every construct shown. Your outputs should exactly match the ones provided.

Standard Output:

```
PRINT(“STANDARD OUTPUT”)
PRINT(5.4, TRUE, 2)
```

```
STANDARD OUTPUT
5.4 TRUE 2
```

Variables:

```
STRING a = "start"
INT b = 1
DOUBLE c = 2.5
BOOL d = FALSE
INT e = 0
STRING f = "end"
PRINT(a)
PRINT(b,c)
PRINT(d,e)
PRINT(f)
INT e = 5
```

```
start
1 2.5
FALSE 0
end
RedeclarationError
```

Expressions:

```
STRING mystring = ("theory" + " of ") + "automata"
INT a = 1
INT b = 4
b++
PRINT(mystring)
DOUBLE c = (1.5+0.5+1)
DOUBLE determinant = b ^ 2 - 4 * a * c
DOUBLE quadratic_root1 = (-b + determinant^(1/2)) / ( 2.0*a )
PRINT(quadratic_root1)

BOOL d = FALSE
PRINT(NOT TRUE == (NOT (NOT d)) AND (TRUE != 0))
PRINT(4 + "A")
PRINT("Hi")
```

```
theory of automata
-1
TRUE
TypeError
```

If-elseif-else:

```
BOOL isRaining = FALSE
BOOL isSnowing = TRUE
BOOL temp = 0
IF (isRaining == TRUE)
{
	IF(temp > 45) {
		PRINT("Wear lightweight raincoat")
	}
	ELSEIF(temp == 45) {
		PRINT("Wear lightweight raincoat")
	}
	ELSE {
		PRINT("Wear fleece and raincoat")
	}
}
ELSE IF (isSnowing != FALSE)
{
	IF(temp > 20) {
		PRINT("Wear soft shell jacket")
	}
	ELSEIF (temp >= 0) {
		PRINT("Wear down jacket")
	}
	ELSE {
		PRINT("Wear base layers and down jacket")
	}
}
ELSE {
	print("It is hard to come up with interesting examples")
}
```

```
Wear down jacket
```

Do-while Loops:

```
INT i=0
DO {
      INT j=0
      DO
      {
            INT k=0
            DO
            {
                  PRINT(“(”, i, “,”, j, “,”, k, “)”)
                  k++
            } WHILE (k<2)
            j++
      } WHILE (j<2)
      i++

} WHILE (i<2)
```

```
( 0 , 0 , 0 )
( 0 , 0 , 1 )
( 0 , 1 , 0 )
( 0 , 1 , 1 )
( 1 , 0 , 0 )
( 1 , 0 , 1 )
( 1 , 1 , 0 )
( 1 , 1 , 1 )
```

For Loops:

```
FOR (INT i = 0; i < 2; i++)
{
      FOR (INT j = 0; j <= 1; j = j + 1)
      {
            FOR (INT k = 0; k <= 1; k++)
            {
                  PRINT(“(”, i, “,”, j, “,”, k, “)”)
            }
      }
}
```

```
( 0 , 0 , 0 )
( 0 , 0 , 1 )
( 0 , 1 , 0 )
( 0 , 1 , 1 )
( 1 , 0 , 0 )
( 1 , 0 , 1 )
( 1 , 1 , 0 )
( 1 , 1 , 1 )
```

Lists:

```
LIST A = [1,2,3,4,5,6]
LIST B = []
PRINT(B)
PRINT(A.slice(0, 2))
B.push(“Get Ready!”)
B.push(“Done!”)
PRINT(B.index(1))
STRING C = “Well ” + B.pop(1)
PRINT(C)
PRINT(A[7])
```

```
[]
[1,2]
Done!
Well Done!
IndexOutOfBoundsError
```

Structs:

```
STRUCT BookStruct {
   CHAR title
   CHAR author
   CHAR subject
   CHAR book_id
};

BookStruct Book1
BookStruct Book2

Book1.title = "The Theory of Computation"
Book1.author = "Michael Sipser"
Book1.subject = "Theory of Computation"
Book1.book_id = 45315
Book2.title = "Race Against the Machine"
Book2.author = "Andrew McAfee"
Book2.subject = "Digital Technology"
Book2.book_id = 92315

PRINT(Book1.title)
PRINT(Book1.author)
PRINT(Book1.subject)
PRINT(Book1.book_id)
PRINT(“”)
PRINT(Book2.title)
PRINT(Book2.author)
PRINT(Book2.subject)
PRINT(Book2.book_id)
PRINT(Book2.publisher)
```

```
The Theory of Computation
Michael Sipser
Theory of Computation
45315

Race Against the Machine
Andrew McAfee
Digital Technology
92315
AttributeError
```

Submission Format:

- Create a folder inside your main directory (where your .py files are located) called **"test_cases"** and insert "**.txt**" files following the names below for the given test cases of the constructs allotted to you.

```
standard_output.txt
variables.txt
expressions.txt
if_else.txt
do_while_loops.txt
for_loops.txt
lists.txt
structs.txt
```

- Your lexer, parser and interpreter files **must** stay as separate .py files as shown below. For example, a typical file hierarchy for a specific student would be:

```
test_cases
      standard_output.txt
      variables.txt
      expressions.txt
      if_else.txt
      lists.txt
lexer.py
parser.py
interpreter.py
[any extra project files as required]
```

- Finally, your **interpreter.py** file requires the user to add the 'txt' file's name to be read from the test_case folder as an additional command line parameter and executes that test case by reading that file. Example prompt:

```
>> python3 interpreter.py standard_output.txt
Welcome to the MyYAPL Interpreter!
Output:
STANDARD OUTPUT
5.4 TRUE 2
```

- Zip the python files and the test_case folder together shown above with the naming convention **RollNumber_Project.zip** and upload on LMS before the given deadline.