COSC4315/COSC6345: Evaluating List Expressions

1 Introduction

You will create a program to evaluate simple list expressions. The input source code will be: Python. This "interpreter" program will be developed in C++.

Your program will use regular expression to recognize identifiers, numbers and strings. Your program will use a simplified context-free grammar to recognize list arithmetic expressions with '+'. In order to detect data types you will have to perform evaluation using an attribute grammar to extend the parse tree. Your program does not have to generate intermediate or object code.

2 Input

The input is one python source code file. The programs will contain the following statements: variable assignment with arithmetic expressions, list [] operators, functional if form (used in lambda expressions), function calls?,

List expressions can combine constants, simple variables (int, string) and list elements.

optional: Traditional if/else statements to control flow (The if condition will be one comparison (no and/or). Parentheses are optional, but unnecesary since the result will be the same. Function definitions with up to 3 arguments. Nested lists or lists mixing data types.

The main arithmetic operator will be the '+' operator. The list elements can be numbers or strings.

In the case of numbers + means addition, and for lists it means union. Notice '*' is not available for strings or lists.

You cannot assume the program will be syntactically correct. You can assume the input source code has no classes, no loops and no recursive functions. You can assume there will be no function calls to convert data types (casting).

3 Program and output specification

The main program should be called "listexpression.cpp".

Your program will be compiled:

```
g++ listexpression.cpp -o listexpression
```

Call syntax from the OS prompt (rename a.out!):

```
# if in path or ~/bin
listexpression file=program1.py

# default
./listexpression file=program1.py
```

4 Requirements

- Your program should be able to display the list content with a plain ”print()”. Notice that using the variable name to display results like the Python interpreter introduces complications for parsing assignent expressions and would push you to develop an interactive interpreter: discouraged.
- You should store all the identifiers in some efficient data structure with access time O(1) or O(log(n)). These include variable names. You have to create a ”binding” data structure to track data types and to store variable values; which must be clearly highlighted in your readme file.
- The assigment operator = can work for an integer, a string and a list.
- Strings will be short: up to 3 characters.
- Assume the list is strongly typed: all elements have the same data type. You can assign the data type based on element [0]
- The arithmetic expression can have up to 20 operands combining + (). [] and function calls.
- the if functional expression will have only one comparison (there are no ”and” ”or” boolean operators).
- You need to program the [] operator to access one element or ”slice” the list from the 2nd element (entry [1]). Example: l[0],l[1:].
- The input is one .py file and it is self-contained (this file will not import other py files). The output is a 2nd .out file with more comments.; this 2nd file should work exactly like the input file.
- In Python a list can mix data types, but in this homework we will take a stricter approach.
- It is acceptable to have one variable instance, overwriting the previous ocurrence. That is, you do not have to create new objects.
- You can use an existing Python parser or you can build your own.
- You should use and explore the Python interpreter to verify correctness of your program. Keep in mind Python allows much more general lists than those required in this homework.
- The program is required to detect data type conflicts.
- There will not be ”cast” or type conversion function calls since that would require to track types in functions.
- The program should not halt when encountering syntax or data type errors in the input source code.
- Optional: For each variable you can store its data type and a list of lines where it was set or changed.
- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.
- Test cases: Your program will be tested with 10 test scripts, going from easy to difficult. If your program fails an easy test script 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.

- You can assume the given python program will be clean and there will be no syntax error. 70% of the grade for each test case will be to detect errors and 30% of the grade will be for detecting warnings. Your program must print error where python shows error. You do not have to print all the warnings.
- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.
- Your program will be initially executed using an automated script. So, make sure you follow the filename and syntax format given in Section 3.