A Mini-Interpreter Evaluating Lambda, Arithmetic and List Expressions

1 Introduction

You will develop a mini-interpreter of a modern programming language. The input source code will be: Python. You will create a program to evaluate list expressions, with array-based lists (i.e. the default list mechanism). There will also be simple arithmetic expressions.

Since Python and JavaScript are without doubt two dominating languages today, input files will have list manipulation code that will be as similar as possible between both languages, other than minor syntax changes or different function names. In other words, the knowledge you have on JavaScript or some other dynamic language does help you solve this homework.

This "interpreter" program will be developed in C++ (but pure C is also acceptable). Notice some other language like Java, JavaScript, Ruby and Python itself are not acceptable since their libraries do 90% of the work and you learn nothing.

Theory in practice: Your program will use regular expressions to recognize identifiers (var/function names), numbers and strings. Your program will implement a simplified context-free grammar to recognize arithmetic expressions with '+' (which may involve lists or integers). In order to detect and check data types you will have to perform evaluation using an attribute grammar to extend the parse tree. Your program has to evaluate the code, line by line, exactly like the interpreter, but your program does not have to generate intermediate or object code.

2 Input: source code file

The input is one Python source code file, passed as argument on the OS command line, working in the same manner as the interpreter.

REQUIRED: The input program will contain the following statements (details in last section):

- One statement per line. Therefore, semicolon is not necessary
- variable assignment,

  where variable can be integer, string or list. Data types are out of scope: bool, real, time, classes.
- lambda expressions:

  one or two arguments, functional if form (also called ternary operator). No nested lambda expressions, no recursion.
- Lists:

  elements can be numbers or strings.

  List expressions, with [] operator to access elements and + to add elements.

  List expressions, which can combine specific values (ints, string), simple variables (int, string) and list elements (one element, or slice).
- Arithmetic expressions with lists, integers or strings:

  The main arithmetic operator will be the '+' operator.

  Optional operators: * or - are optional.

- if/else:

  to control flow (The if condition will have only one comparison; without and/or/not). Parentheses are optional, but unnecessary since the result will be the same. if/else can be nested up to 3 levels, either way.

  Nesting: yes.

- Casting=no

  You can assume there will be no function calls to convert data types (casting).

OPTIONAL: You can develop additional features. In order to do this you have to inform us at least 2 weeks before the deadline exactly which features you are programming. Extra credit will vary between 10 and 30 points, which can also be applied towards HW1, but not the midterm. Optional (choose 3 at least):

- General arithmetic expressions: combining + - * / and (), with nesting. In the case of numbers + means addition, and for lists it means union. Important warning: '*' is not possible for strings and lists.
- General boolean expressions: *and, or, not* and parentheses.
- Function definitions with up to 3 arguments, including the function call.
- Nested lambda expressions.
- while() loop, including nested loop combined with if's up to 3 levels. elif required as well. *for* discouraged as it is OO and therefore more complex. Multiple statements per line separated by semicolon.
- Nested lists and lists mixing data types.
- Linear recursion: Recursive functions, including evaluation.
- casting, dynamic type conversion

OUT: Totally out of scope:

- Non-linear recursion, mutual recursion, detecting potential infinite recursion
- Higher order functions like currying, map() and reduce()
- Iterators
- Classes, objects, garbage collection
- Exceptions
- Concurrency and parallelism
- GUI

3 Program and output specification

The main program should be called "mini_python.cpp". The compilation should be calling g++ by default as explained below. If you decide to go for a larger project, using advanced C++ and C libraries you can use 'make', but make it clear in your readme file. One way or the other, your program should be easy to compile without errors and preferably without warnings (-Wall).

Your program will be compiled:

```
g++ mini_python.cpp -o ~/bin/mini_python
```

Your program will behave exactly like the python interpreter (e.g. "python3"). Call syntax from the OS prompt (rename a.out!):

```
# if in path or ~/bin
mini_python test1.py

# default
./mini_python test1.py
```

4 Requirements

- One statement per line. Therefore, semicolon is not necessary.
- In general, the input python program will be clean and there will be no syntax error. However, there may be minor syntax mistakes (missing parenthesis), extra statements (e.g. recursive functions), unhandled data types (e.g. an object). In such case your program should halt and display an error message (short).
- The variable assigment operator = can work for an integer, a string and a list.
- Identifiers will be at most 10 characters. Also, strings will be short: up to 10 characters.
- Data type inference and checking: required.

  Python: a string is a list.
- You should store all the identifiers (variables) 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.
- You should store the list in arrays or linked lists. If you use arrays you should have some functionality to resize the array (say linearly or geometrically). You can assume lists will have no more than 1000 elements, but you should still aim to minimize RAM usage, especially for lists.
- Your program should be able to display the variable content with a plain ”print(varname)”. Notice that using the variable name alone, to display results like the Python interpreter introduces complications for parsing assignent expressions and would push you to develop an interactive interpreter: this is discouraged as it messes some theory principles.
- Optional: You are encouraged to develop recursive C++ functions to manipulate the list and to evaluate arithmetic expressions. You can use lambda expressions and functional libraries in C++.
- Assume the list is strongly typed: all elements have the same data type. You can assign the data type of list l based on element l[0]. In Python a list can mix data types, but in this homework we will take a stricter approach to simplify development.
- Arithmetic expression:

  required: expression can have up to 20 operands combining + () [] and calling lambda functions.

  optional: * - / and non-lambda function definitions and calls.
- lambda calculus:

  if there are two arguments there will be one arithmetic operator. the if functional expression will have only one comparison (there are no ”and” ”or” boolean operators).
- List access operators.

  You need to program the [] operator to access one element or a ”slice” the list from the ith element (entry [1]). Examples: l[0] or l[1:].
- The input is one .py file and it is self-contained (this source code file will not import other py files).
- It is acceptable to have one variable instance, overwriting the previous ocurrence. That is, you do not have to create new objects to avoid mutation.
- You can use an existing Python parser or you can build your own. Tools like lex/flex, yacc/bison have way more features than needed to solve this homework, but you can use them.

  C++ libraries like regex are also great, but more general than needed for this homework.
- You should use the Python interpreter to verify correctness of your program. You should interactively test your program, comparing output with the interpreter. Keep in mind Python allows much more general lists than those required in this homework.

- The program is required to detect data type conflicts and print a warning or error.
- 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. Ths information can be displayed in a ”varhistory.log” file to debug python code.
- Your program should write error messages to a log file ”error.log” (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information.
- Test cases: Your program will be tested with 10 test files, going from easy to difficult. In general, each test case is 10 points off if your program produces incorrect results.
- Source code: it will be checked for indentation, clarity, redundancy, efficiency and generality. It is expected you have comments at the top giving and overview and for the most complex parts.
- Your program will be executed using an automated script. Manual grading will be done only for regrading. Therefore, make sure you follow the TAs instructions. Failure to follow instructions will result in failed test cases.