COSC4315/COSC6345: A Mini-Interpreter Programmed in C++ Language: Python

1 Introduction

You will develop a mini-interpreter of an interpreted programming language. The "system" programming language is: C++ (or pure C). The input source code will be: Python. You will create a program to evaluate list expressions, with array-based lists (i.e. the most common list mechanism). There will also be simple arithmetic expressions.

Since Python and JavaScript are 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 Python or JavaScript does help you solve this homework.

This "interpreter" program will be developed in C++ (but pure C is also acceptable). Notice that developing your interpreter in an interpreted language like JavaScript, and Python itself is not acceptable since their libraries do 90% of the work and you learn nothing. However, other compiled languages like Java or Kotlin can be considered (talk to instructor).

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 needs to behave as close as possible as the Python interpreter, producing the same results.

Parser: The parser can be simplified by adding begin/end braces { } and parentheses () to the Python source code and creating another file (extension .pyb as seen in class, remember this change makes Python much closer to C syntax). if you already have the strings that make up the first 1-2 lexical units in each Python line , say 'if' 'def' 'identifier' '#' then you have the "handle" to match one grammar rule. As explained in class, the key parsing aspects are handing recursively nested statements/calls and arithmetic and boolean expressions with infix notation (boolean expressions are simpler case arithmetic). you can do a bottom-up or top-down parser, depending on your programming expertise and preference. Remember that parsers are PDAs.

Evaluator: you can implement your own operational semantics generating intermediate code (sequence, basic if, basic infix to posfix) or perform recursive top down evaluation (similar to the textbook recursive example). It is acceptable to perform several passes on the source code to add begin/end symbols, and to detect syntax or data type errors. Execution should stop at the first error, like the Python interpreter.

2 Input: source code file

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

3 REQUIRED

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

- One statement per line. Therefore, semicolon is optional (not necessary).
- variable assignment
- Data types: integer or list. String: optional (read section below).

  Data types out of scope: bool, string, real, time, classes.
- Lists:

  List expressions, with [] operator to access elements and + to add elements. In general, you can expect to get the head of the list and the slice ":" operator to get the tail of the list (l[0], l[1:]).

List arithmetic expressions, which can combine specific values simple values and list elements (one element, or slice).

List length e.g. "len(l)".

- Arithmetic expressions.

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

  Out of scope, optional arithmetic operators: * or - (optional).

- Data type inference and checking: required.

  Python: data type can dynamically change with a new assignment.

  Execution should stop at the first error.

- if/else:

  Recursive nesting: yes.

  to control flow (The if condition will have only one comparison; without and/or/not). The else part is optional, like any programming language. Parentheses are optional, but they can help building the parser. The if/else statement can be recursively nested up to 1 level, either way. In other words, an outer if/else can have two if/else recursively nested in any combination, resulting in 2 levels.

- Comments, starting with # (column 1 by default)

- loops: no (while optional).

- Functions: definition and evaluation (call). Two arguments, simple data types or list.

  Functions can call functions (f(a) can call g(b)). A parameter in a function can be a function call (e.g. f(g(a))).

  We will assume a list passed as parameter is not modified (i.e. works as pass by value).

  The return statement at the end is optional. That is, the function can represent a subroutine in Fortran, void function in C++ or procedure in Pascal.

- Casting=no; coercion=no

  You can assume there will be no function calls to convert data types (casting). Since there are no real numbers coercion is not necessary.

- You cannot assume the program will be syntactically correct. That is, you have to detect errors (showing line number enough, no need to be very specific).

- You cannot assume all expressions have correct data types. You have to detect data type errors in operands.

4 OPTIONAL

You can develop additional language features. In order to do this you have to inform the TAs at least 10 days before the deadline exactly which features you are programming. Extra credit will vary between 10 and 30 points towards this homework to reach 100.

Choose 3 features at least:

- Recursive functions OR recursive lambda expressions. This item includes definition and evaluation.

- Nested statements up to 3 total levels (i.e. 1 outer + 2 inner). It is encouraged this is done with at least if/while combinations (not if/else alone).

- Data types: strings, in addition to integers.

- General arithmetic expressions: combining + - * / and (), with nesting.

  Managing operator overloading: In the case of numbers + means addition, for strings it means concatenation and for lists it means union.

  Important warning: '*' is not possible for strings and lists in most languages (Python is a rare exception).

- Function definitions with up to 2 arguments with more data types. Data types: integers, strings, lists. Evaluation: function call with parameters passed by value or reference (lists).

- while() loop, including nested while loops+if/else, all of them recursively nested up to 3 levels (plus the outer level 0). *for* discouraged as it is less general.
- General boolean expressions: *and, or, not* and parentheses.
- Nested lists (list of lists) and mixed Lists, mixing data types.

Totally out of scope:

- Arbitrary number of levels of nested statements
- Totally separate stack and heap management
- Non-linear recursion, mutually recursive functions, detecting potential infinite recursion
- All list operators and functions
- Higher order functions like currying, map() and reduce()
- Casting, dynamic type conversion
- OO features: Iterators, classes, objects, garbage collection
- Exceptions
- Concurrency and parallelism
- GUI

5 Program input and output specification

The main program should be called "minipython.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++ minipython.cpp -o 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
minipython test1.py

# default
./minipython test1.py
```

6 Programming Details and Test Cases

- Identifiers will be at most 10 characters.
- print() can have up to 2 arguments (e.g. to display a variable name and its value). Default call: string+variable.
- In most test cases (not always), the input python program will be clean and there will be no syntax/type error. However, in 2-3 test cases there can be minor syntax mistakes (missing parenthesis), extra statements (e.g. recursive functions), unhandled data types (e.g. an object). In such incorrect case your program should halt and print() an error message (short, fitting in one line).
- You should store all the identifiers (variables) in some efficient data structure with access time O(1) or O(log(n)); avoid $O(n)$ access. These include variable names. You have to create a "binding" C++ data structure/class to track data types and to store variable values; which must be highlighted in your README file.
- You can store the list in arrays or linked lists. C++ STL can handle array resizing. If you use C arrays you can assume some limits assuming the source code file is no more than 100 lines. You can assume lists will have no more than 100 elements, but you should still aim to minimize RAM usage, especially for lists.

- Scanning, Parsing and evaluation of source code: it is acceptable to make multiple passes, to detect identifiers, check data types, generate an intermediate translated code representation and evaluate.

  It is acceptable to use flex/bison. It i acceptable to modify an existing Python parser, provided it works according to this specification. However, it is unacceptable to submit an existing parser/evaluator that does more than asked in this homework. That is, you need to modify it and remove unnecessary features.

- Testing: your interpreter should behave exactly like the Python interpreter and short messages a similar format. Short error messages are sufficient. The only exception is showing a variable with its name (use print()). A prompt is optional, but will not be used for automated testing.

- 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++. Indicate in your README file in which C++ classes you use recursive functions.

- 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.

- 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). The main code will be at the bottom of the py file (not interleaved wih function definitions).

- 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. A plain if/while pair is enough to get identifiers and integers, as seen in class.

- You should use the standard Python interpreter (Windows or Linux) 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 an error (short message OK).

- 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.
- Submission: in 2 phases.

  Phase 1: your program must pass simple test cases (10 test cases, integers/lists, simple expressions, if/else, short programs) and the grade will be unofficial (just pass/fail). If your program fails or you cannot submit a working program you have to talk to the instructor.

  Phase 2: your program must handle reasonable test cases (10 test cases, longer, more realistic, more complicated). You cannot submit in Phase 2 if your program did not pass Phase 1. The TAs will grade your program and you will have 3 days to fix it with -20 points: early resubmission encouraged for minor fixes.
- Final recommendations: start immediately. Split the programming work: one person testing will be insufficient (e.g. data type management, parsing, evaluation). It is very difficult to develop this homework in 2 weeks. It is impossible to do it in 1 week or less. Due date will not be changed.