Kristy Choi, eyc2120
Raymond Xu, rx2125
Benjamin Low, bkl2115
Dennis Wei, dw2654
Kevin Lin, kl2806

Pseudo: Algorithm Design and Implementation Language

Introduction

We will implement a high-level pseudocode-style programming language for designing and implementing algorithms. Traditional high-level programming languages such as Java and C++ are often unnecessarily verbose for algorithmic definition. Our language will use Pythonic syntax, type inference, and keywords to maximize human readability and ease of algorithmic prototyping.

Our language will be perfect for rapidly prototyping algorithms, verifying the behavior of algorithms on given inputs, and for educational purposes, as the syntax is inspired by the easily-readable pseudocode from CLRS, the classic textbook for algorithmic analysis. Our language will follow an imperative programming paradigm, as the pseudocode from CLRS explicitly avoids objects and member functions. Our language will have automatic memory management and compile to LLVM.

Features

Type inference

One key feature of our language is type-inference. The compiler will automatically infer the types of variables at compile-time, eliminating the need to explicitly declare the types of function parameters and local variables. As such, programmers can simply define a function with parameters `(A, i, j)`, and as conventional pseudocode suggests, these will be correctly labelled as a list and two integers. We will accomplish this using the Hindley-Milner type inference algorithm in combination with a standardized notation for common data structures.

Keywords

Powerful keywords are another key element of our language. Common helper functions such as "swap" and "union" will act as keywords for convenience and reduce the cognitive burden of having to implement helper functions later. As a starting point, we will implement keywords that are most frequently used in the algorithms provided by CLRS. In addition, we will implement an "assert" as a keyword that takes a boolean value that returns error if false, allowing programmers to test and verify their assumptions about their programs

Example syntax:

```
A = [5, 6, 8, 7, 9]
swap 2, 3 in A
```

Automatic variable initialization

Variables do not have to be explicitly initialized, or even explicitly defined. During compile-time, we will be identify all variables and their corresponding types. These variables will be automatically initialized to predictable default values. For example, one can define a for loop that goes from 0 to 10 without explicitly declaring the index variable and its type.

Graph algorithms/library

Our language will support implementation of common graph algorithms such as BFS and DFS, as well as sorting algorithms. An example of a default implementation of sorting in `Pseudo` would be the Quicksort algorithm. At compile time, the compiler will infer what properties graphs and nodes have (such as augmented values) and allocate memory accordingly.

Syntax

The syntax of our language resembles a fusion of Python and pseudocode, allowing programmers to express and experiment with algorithmic ideas concisely. We will use Python-style indentation to denote control flow, without use of braces.

Variables are typically named using lowercase letters exclusively (i.e. `i, j, k, sum, count`). Data structures typically are named a single capital letter (i.e. `A, B, S, T`). Functions must be named in all capital letters (`MERGE, BFS,` etc.) and include the `def` keyword prefacing the name. With the exception of functions, these naming patterns are recommended conventions in the language - they are encouraged, but by no means enforced.

Data Types

Our language will support basic types as well as types that come up frequently in common algorithms. Primitive types include int, float, boolean and strings. More advanced data types include Nodes, Edges, Lists, Sets, Maps, and Graphs.

Lists serve as multi-purpose data structures. A list functions as an array, list, stack, and queue, and supports common idiomatic operations for each of them, such as `append()`, `push()`, and `pop()`.

Example Program

```
def BFS-CONNECTIVITY(G, s, t)
    for each u in s.adj
```

```
          Q.push(u)
     while Q is not empty
          v = Q.pop()
          if v is t
               return true
          visited.append(v)
          for each u in v.adj
               if u not in visited
                    Q.push(u)
     return false

def MAIN()
     a.adj = [b, c]
     b.adj = [a, d]
     c.adj = [a, b]
     d.adj = [b]
     G = [a, b, c, d]
     print BFS-CONNECTIVITY(G, a, d)
```

```
$ pseudo bfs.sd
$ ./bfs
>> true
```

Team Roles

Kristy Choi - Manager
Raymond Xu - Language Guru
Benjamin Low - System Architect
Dennis Wei - System Architect
Kevin Lin - Tester