UCSD CSE131 F19 – PA5

November 7, 2019

Due Date: 11pm Wednesday, November 13 **Open to Collaboration**

Classroom: FILL Github: FILL

You will implement Egg-Eater, a language with heap-allocated data.

Requirements

This assignment is much more open-ended than previous ones. You will implement a compiler for a language with heap allocation.

This requires having a mostly-working compiler from past PAs, but you don't have to have a fully-functional multi-argument calling convention working in order to complete it, and you're free to use any code from lecture to start from.

Required Features

Your compiler must support:

- Some mechanism for constructing heap-allocated data that groups together an arbitrary number of values that can have different types. You might choose named values (like dictionaries from class) or something strictly positional (like tuples or arrays). These can have their size specified implicitly by syntax (like tuples) or programmatically by an expression (like arrays with a size initialization position).
- An expression that can look up values from within heap-allocated data. This could be by name or positionally.
- If a heap-allocated value is the result of a program or printed by `print`, all of its contents should be printed in some format that makes it clear which values are part of the same heap data. For example, in the output all the values associated with a particular location may be enclosed in parentheses.
- Some kind of equality – your choice of reference or structural equality.

The following features are **explicitly optional**, though you may enjoy implementing some of them.

- Type checking
- Updating elements of heap-allocated values
- Structural equality (it's much more implementation work than physical equality)
- Detecting when out-of-memory occurs. Your language should be able to allocate at least a few tens of thousands of words, but doesn't need to detect or recover from filling up memory.

Required Tests

You **must** write the following programs to test your compiler. You can pick any details you want like function names and base case behavior, they just must be recognizable as the requested data structures and algorithms, and they must be in files with the given names.

- `input/simple_examples.boa` – A program with a number of simple examples of constructing and accessing heap-allocated data in your language.
- `input/error1.boa` – A program with a well-formed, type-checking, or runtime error related to heap-allocated values.
- `input/error2.boa` – A second program with a different well-formed, type-checking, or runtime error related to heap-allocated values.
- `input/error3.boa` – A third program with a different well-formed, type-checking, or runtime error related to heap-allocated values.
- `input/points.boa` – A program with a function that takes an x and a y coordinate and produces a structure with those values, and a function that takes two points and returns a new point with their x and y coordinates added together, along with several tests that print example output from calling these functions.
- `input/bst.boa` – A program that illustrates how your language enables the creation of binary search trees, and implements functions to add an element and check if an element is in the tree. Include several tests that print example output from calling these functions.
- `input/list.boa` – A program that illustrates how your language enables the creation of linked lists, and four functions: adding an element at the beginning, adding an element at the end, getting an element at an index, and creating a linked list of numbers that starts at 0 and goes up to the index of an input n.

Handin and Report

You will submit your implementation to `pa5`, and to `pa5-written`, a PDF containing the following:

- (10%) The concrete grammar of your language, pointing out and describing the new concrete syntax beyond Diamondback. Graded on clarity and completeness (it's clear what's new, everything new is there) and if it's accurately reflected by your `parse` implementation.
- (5%) The definition of your language's AST, highlighting new expressions and definitions beyond Diamondback. Graded on clarity and completeness, and if the abstract syntax is an accurate representation of the concrete syntax.
- (10%) A diagram of how heap-allocated values are arranged on the heap, including any extra words like the size of an allocated value or other metadata. Graded on clarity and completeness, and if it matches the implementation of heap allocation in the compiler.
- (55%) The required tests above (in addition to appearing in the code you submit, they should be in the PDF). These will be partially graded on your explanation and provided code, and partially on if your compiler implements them according to your expectations.
  - For each of the files error1-3, show the error message and explain in which phase your compiler and/or runtime catches the error. (5% each)
  - For the others, include the actual output on your compiler, the output you'd like them to have (if there's any difference) and any notes on interesting features of that output. (10% each)
- (5%) A description of the thing you think is most interesting or exciting about your design and implementation in 2-3 sentences.
- (5%) A description of a feature you'd like to add to your language next, with an outline of how you'd add it in 2-3 sentences.
- (5%) Pick two other programming languages you know that support heap-allocated data, and describe why your language's design is more like one than the other.
- (5%) A list of the resources you used to complete the assignment, including message board posts, online resources (including resources outside the course readings like Stack Overflow or blog posts with design ideas), and students or course staff discussions you had in-person. Please do collaborate **and give credit to your collaborators**.