CSC 344 First Racket Problem Set Solution

Learning Abstract: This problem set is meant to go over the basics of memory management, including the stack and heap, garbage collection, and memory allocation methods. It also touches on the Rust programming language and its syntax, memory capabilities, and data types. Finally, it includes a review of a paper about Rust which goes over the ups and downs of the language.

First Task: The Runtime Stack and the Heap

Running programs require memory, and different programs require different types of ways to deal with the problem of acquiring memory. Two ways of dealing with it are the stack and the heap. Both have their benefits and downfalls but they are also both very important.

The runtime stack is an area of memory that builds upon itself every time a function is called or a variable is created. So, functions that were called first will end up at the bottom of the stack and freshly called functions will be at the top. Then, once the top function finishes running, the memory allocated for it and its variables is automatically deallocated and the function under it can continue to run. This also means that variables that are not on the top of the stack are unreachable, and once the memory space that a local variable was allocated gets deallocated, a new value can be placed in it. This often leads to mistakes involving retrieving the wrong variables.

Unlike the stack, the heap is a more permanent memory area. Programmers have to explicitly allocate and deallocate memory from the heap. This makes it easier to keep track of where variables are, however it also introduces the possibility of a memory leak. This happens when the heap memory is not freed after it is allocated, and can lead to the computer slowing down as memory builds up, and can eventually lead to harmful failures.

Source used: https://courses.engr.illinois.edu/cs225/fa2021/resources/stack-heap/

Second Task: Explicit Memory Allocation/Deallocation vs Garbage Collection

Since computers don’t have unlimited memory, we need to use processes of memory allocation and deallocation when running programs that require memory. This can be done in one of two ways, manually, or automatically. There are positives and negatives to each method, so it's important to know which kind is better suited to your needs in order to maximize your efficiency and your computer's.

The manual way of dealing with memory is through the explicit allocation and deallocation of memory space. Languages like C/C++ and Rust use this type of memory management. For example, in C the malloc function allocates memory and the free function deallocates it. The upside of this form of memory management is that it offers complete control over the lifespans of variables, but it is also its downside. Simple mistakes like forgetting to deallocate memory or freeing an object in use can lead to huge problems in your programs.

On the other hand, the automatic management of memory can be done through the use of a garbage collector. Garbage collector algorithms find objects that are no longer in use, and deallocates the memory associated with them. More languages use this type of memory management, including but not limited to Java, Python, and Haskell. This is because it allows for mostly worry free coding in terms of memory status, however the garbage collector will be continually running and so it will negatively affect the performance of your programs.

Source used:
https://www.educative.io/courses/a-quick-primer-on-garbage-collection-algorithms/jR2PP

Third Task: Rust: Basic Syntax

1. Rust has a few key differences that make it better than Haskell for certain tasks and criteria. One of the big changes is that Rust gives more control over the allocation of memory in one's program.
2. Rust distinguishes between primitive types and other more complicated types. We'll see that type names are a bit more abbreviated than in other languages.
3. The main distinction between primitives and other types is that primitives have a fixed size. This means they are always stored on the stack. Other types with variable size must go into heap memory.
4. While variables are statically typed, it is typically unnecessary to state the type of the variable. This is because Rust has type inference, like Haskell!
5. Expressions return values. Function calls are expressions. Block statements enclosed in braces are expressions.
6. Unlike Haskell, it is possible to have an if expression without an else branch.
7. Note that an expression can become a statement by adding a semicolon! The following no longer compiles! Rust thinks the block has no return value, because it only has a statement! By removing the semicolon, the code will compile!
8. Rust has simple compound types like tuples and arrays (vs. lists for Haskell). These arrays are more like static arrays in C++ though. This means they have a fixed size. One interesting effect of this is that arrays include their size in their type.
9. Arrays and tuples composed of primitive types are themselves primitive! This makes sense, because they have a fixed size.
10. Another concept relating to collections is the idea of a slice. This allows us to look at a contiguous portion of an array.

Fourth Task: Rust: Memory Management

1. We declare variables within a certain scope, like a for-loop or a function definition. When that block of code ends, the variable is out of scope. We can no longer access it.
2. Another important thing to understand about primitive types is that we can copy them. Since they have a fixed size, and live on the stack, copying should be inexpensive.
3. What's cool is that once our string does go out of scope, Rust handles cleaning up the heap memory for it! We don't need to call delete as we would in C++. We define memory cleanup for an object by declaring the drop function.
4. Deep copies are often much more expensive than the programmer intends. So a performance-oriented language like Rust avoids using deep copying by default.
5. Memory can only have one owner. This is the main idea to get familiar with.
6. In general, passing variables to a function gives up ownership.
7. Like in C++, we can pass a variable by reference. We use the ampersand operator (&) for this. It allows another function to "borrow" ownership, rather than "taking" ownership. When it's done, the original reference will still be valid.
8. You can only have a single mutable reference to a variable at a time! Otherwise your code won't compile! This helps prevent a large category of bugs!
9. As a final note, if you want to do a true deep copy of an object, you should use the clone function.
10. Slices are either primitive data, stored on the stack, or they refer to another object. This means they do not have ownership and thus do not deallocate memory when they go out of scope.

Fifth Task: Rust: Data Types

1. When we initialize a user, we should use braces and name the fields. We access individual fields using the . operator. If we declare a struct instance to be mutable, we can also change the value of its fields if we want!
2. Rust also has the notion of a "tuple struct". These are like structs except they do not name their fields.
3. Rust also has the idea of a "unit struct". This is a type that has no data attached to it.
4. The last main way we can create a data type is with an "enum". In Haskell, we typically use this term to refer to a type that has many constructors with no arguments. But in Rust, an enum is the general term for a type with many constructors, no matter how much data each has.
5. Instead, Rust uses the match operator to allow us to sort through these. Each match must be exhaustive, though you can use _ as a wildcard, as in Haskell.
6. But unlike Haskell, Rust allows us to attach implementations to structs and enums. These definitions can contain instance methods and other functions. They act like class definitions from C++ or Python.
7. As in Python, any "instance" method has a parameter self. In Rust, this reference can be mutable or immutable. (In C++ it's called this, but it's an implicit parameter of instance methods). We call these methods using the same syntax as C++, with the . operator.
8. We can also create "associated functions" for our structs and enums. These are functions that don't take self as a parameter. They are like static functions in C++, or any function we would write for a type in Haskell.
9. For the final topic of this article, we'll discuss traits. These are like type classes in Haskell, or interfaces in other languages. They allow us to define a set of functions. Types can provide an implementation for those functions.
10. Also as in Haskell, we can derive certain traits with one line! The Debug trait works like Show.

Sixth Task: Paper Review: Secure PL Adoption and Rust

Review of “Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study”.
https://obj.umiacs.umd.edu/securitypapers/Rust_as_a_Case_Study.pdf

This paper noted that Rust is a language made with the intention of decreasing issues associated with memory management that are present in other languages such as C/C++. So, a study was conducted in order to determine if it successfully does this, and if it does, are there any drawbacks because of it. I will go over the most notable positive and negative responses of the study.

The majority of the participants in the study reported that Rust does in fact offer increased safety, some even stating that it offers 100% safety. Another reported benefit of Rust is the increased performance that it provides, which can partly be attributed to its lack of garbage collection. Participants also noted that once their Rust code compiled, they had to spend significantly less time debugging it than they did with other languages.

While Rust does deliver what it says it does, it is also very difficult to learn. 41% of participants said that it took them between a week and a month to become comfortable using Rust, and a quarter of participants said that it took them between one and six months. Even when participants became familiar with Rust, they still had a difficult time getting their code to compile, but this may balance out considering the decreased debugging time mentioned earlier.