Compilers-2

Questions from the Viva
Michelin Compiler

Q. Show some error checks (Parser), especially of the inbuilt data types and classes.

Ans:

We get parser errors when we write a code that has a syntax error. For example, if we declare a variable of type plate like you declare an int, it will give an error. (The statement “Plate x.” will give a parsing error.) We have already done multiple parsing error checks in our parser; we request you to go through the demo part of this video.

Q. How will your language handle the semantic checks for cycle inheritance?

Ans:

We planned on using a DSU (Disjoint Set Union) data structure to check for cycle inheritance and for any diamond problems. Whenever class A inherits from class B we add class A to the set of Class B and now if we try to make class B inherit class A this will throw an error since both the classes are from the same set.

This DSU will also ensure that the entire class inheritance hierarchy remains a forest and no cycles are present anywhere. The main reason for using the DSU function rather than a brute force implementation is that DSU’s amortized time complexity is almost linear (except for the inverse Ackermann function term, which is almost constant for most practical input sizes).

Q. Where are the units in your language? How do you plan to implement that?

Ans:

At the moment we have not included the data of units of measurement at the moment. But we intend to do so with the items we define and the tools (a certain set of tools implemented as classes used for cooking in our language). We planned it such that every

item we take would be stored by using multiple types of units like mass and volume, and sometimes also general kitchen units like a teaspoon, a cup, and so on. Even under the standard units like mass are also taken in different units like grams, kilograms, etc., and volume will be taken as liters, milliliters, oz, etc.

Our containers (Inbuilt classes) will have certain limitations of the quantity they can store as well. So it would show errors when the code tries to go further to that limit (this is one of the things that we would handle in the dry run). The same is the case with tools like pans, ovens, etc.

But the user would also be given a choice to overwrite those limits on the quantity they can store.

Q. Where are the semantic checks for the inbuilt data types in your language? What's unique about it?

Ans:

We planned to do the semantic checks of inbuilt data types using functions. For example, when we are adding contents of glass to a plate (which shouldn't be done) our compiler is intended to show an error as the add function will be expecting 2 glasses as a parameter but it finds a plate instead. We also thought of dividing the inbuilt data types into multiple subclasses based on the physical types (for example bowl, glass will be in one group say t1, as they are measured in volume and plate, will be in other type say t2). So, in the function we can expect type t1 as a parameter to generalize functions to add from glass to glass or glass to bowl but adding from glass to plate will be an error.

However, there are some semantic checks that we have already implemented which are *specific* to our language. For example, we can have a statement such as

Vessel v = new Glass(), but we can't have Glass g = new Vessel().

The reason we had to focus on the basic semantic checks was because we would need those checks as primitives as for these more advanced checks.

And with all the checks that we have already implemented, most of the things that we wish to keep as was discussed above or in the viva are fairly straightforward to implement because we already made our symbol table for functions and variables.

The main thing we needed to add was a symbol table for classes. We did some basic checks such as a function and class not having the same name and some related checks like that.

Q. Can you elaborate on the hardware requirements (robot, multi-threading, etc.)?

Ans:

With respect to the hardware requirements, we expect it to be slightly similar to robotics where we can consider the entire set up i.e. the part that controls the pan, the part that controls the oven, etc. as a big machine and we intend to interact with these components with the help of code where for example an instruction like:

knife~>cut(ing1).

This line means that the component controlling the knife k will cut the ingredient ing1. The multithreading aspect comes into the picture when we deal with parts of a recipe that can happen parallelly such as the chopping of two separate ingredients. We had initially planned to provide a base for this implementation by providing a database that contains all the objects in the machine such as pan, knife, etc. and once an object is being used it is marked as being used and not free in the table and as such parallelism is possible only when the required object is free.

Ideal Video present in FAQ.md.

Q. What about the "recipe()" construct of your language?

Ans:

We had intended to keep recipe() analogous to C++'s lambda function. The intuition behind this was that while preparing a big dish, there are many sub-dishes that would be required as well. Now it might be unnecessary for the user to write a whole function with all the return types and other information passed to it.

It is enough for the function to be local to the specific function within which the bigger dish is being cooked (main() basically).

We wanted to write functions for cooking food items that could especially be used again and again (it's more like an algorithm) as we had already mentioned in the viva. These functions can be stored in a package (as we discussed in the viva). Note that the functions like add, boil, etc. have got to do more with the hardware so we will be storing them in slightly different files which we call libraries (instead of packages).

But again, as far as recipe() is concerned, it is only for local usage within a function. We didn't implement it because there were other semantic checks that we had to take care of before we could handle that, and those checks were more important for other components of our language.