MICHELIN
A Code which Cooks

- Name: Sai Sidhardha Grandhi
  Roll number: CS19BTECH11050
  Roll: Project Manager
  GitHub ID: G-Sidhardha

- Name: Krishn Vishwas Kher
  Roll number: ES19BTECH11015
  Roll: Language Guru
  GitHub ID: KrishnKher

- Name: Mukkavalli Bharat Chandra
  Roll No.: ES19BTECH11016
  Role: System Architect
  GitHub ID: chandra3000

- Name: Sujeeth Reddy
  Roll No: ES19BTECH11022
  Role: System Integrator
  GitHub ID: Sujeeth13

- Name : Sree Prathyush Chinta
  Roll No. : CS19BTECH11043
  Role : Tester
  Github ID: Prathyush-1886

- Name : Praneeth Nistala
  Roll No. : CS19BTECH11054
  Role : Tester
  Github ID: Praneeth-Nistala

- Name : VEMULAPALLI ADITYA
  Roll No: CS19BTECH11025
  Role: System Integrator
  GitHub ID: VEMULAPALLI-ADITYA

Table of Contents

Michelin

Chapter-1
Introduction to Michelin

The beginning of an innovation

In recent times, technologies are growing rapidly. Time has become a valuable resource. We, humans, are exploring different ways to replace our day-to-day tasks with machines. Cooking is one such task which humans are trying to automate. In spite of having various technologies, there should be a well-written code that has a specific task.

One might think that there are hundreds and thousands of languages like c++, java, python to do coding, why do we need a new one. The problem with these languages is that the task we are doing is highly domain-specific. This will make the task of writing code in the languages you know very difficult. This might make some good recipes not worth coding. So, we are trying to solve these problems with our language.

A Better Way we think:

A simple and useful way to solve these problems is the Michelin. Some of our language specifications are

- In spite of being an object-oriented language, we are trying to make it simple.
- We are trying to make your codes run on multiple different platforms.
- We are aiming for a high-performance yet we don't want to compromise on the security as any mistake in the code might cost a lot of damage.

Our language is a portable, compiled, high-performance, simple, object-oriented programming language. In this chapter, we are trying to provide a brief look at the main design goals of our language and later examine these features in detail.

1.1 The Beginning

Our language is designed to cook dishes by writing code! In a more generic way, our language helps to program a hypothetical machine to cook. It tries to simulate the process of cooking even for someone with bare minimum knowledge of cooking and programming.

The motivation for making this language comes from the fact that cooking seems to innocuously incorporate many design features of modern OOPS languages and these seem to be very common-sense ideas to the common populace. Hence, it indeed seems feasible to design a language that uses the innate organizational qualities that people have to help cook food! It can help cook food with very precise measurements since it will be done through a machine.

Example code snippet:

```
Glass g.          \*declaring a glass g*\
Ingredient ing1("Water", 100, volume, g).
                        \*100ml Water will be placed in glass g and will  be called as ing1*\
serve g.          \*serving the glass*\

ready.            \*this refers that the code is ended*\
```

Here, we are taking a glass and adding water to it and serving it. Your cup of water is ready!

1.2 Design Goals

Our language is highly OOPS-oriented. Even difficult recipes can be coded using this language and the rest of the task is left to the machine which will try to cook the required dish. We intend it to be similar to C++, with a very specific focus on cooking and adding on different ideas that are related to it. We would like to support all the basic paradigm functionalities that OOPS supports such as inheritance, polymorphism, operator overloading, etc. since we have found many instances in cooking a dish that has a very close resemblance to many of these ideas. We have multiple other ideas as well, but we are not sure about how many of those other ones we can implement as of now, so we will keep those for later.

1.2.1 Simple, Object-Oriented, and Familiar

One of the important characteristics of the language we wanted to implement is to simplify the language as much as we can so that it does not require any intense special training to code.

We are trying to provide a clean and efficient object-based development environment. This makes it easier to code a recipe in a more organized way which eases the readability. We are planning to provide different libraries which can be used by the programming to make his task much easier.

We want to design our language in such a way that it will be familiar not only to programmers but also to someone who never coded.

1.2.2 Robust and Secure

Our language is designed for creating highly reliable code. Since the machines that use our language to function are assumed to be super expensive, we are planning to design our language such that nothing bad happens to the machine due to the code.

We are discussing different methods to make the task as safe as possible. Some of the ideas we have were discussed in the latter parts. We are not planning to include pointers in our language as it might be confusing to beginners. Since our language expects a low programming experience, we want to provide an environment that identifies the errors in the code and makes it easy for the user to correct the code.

Coming to the part of the security of the code, since the code contains a recipe that might be valuable, we also want to take care of this issue so that the user can write a code without worrying.

1.2.3 Architecture Neutral and Portable

We are not completely sure about this aspect but we are thinking like we will decide based on further readings.

1.2.4 Performance

Performance is an important aspect to discuss. Although we want to achieve high performance, we also don't want to compromise on the risk of the machines. So, we are planning to find a way which tries to balance the performance and the risk factor.

1.2.5 Compiled and Threaded

We wanted to build a procedure for our language similar to c, c++. Like any other compilers, we want to follow the traditional compile, link, and test cycles to convert our code to machine code. Other details of these are and will be discussed later.

We are planning to provide multithreading in our language as it makes the job much faster and thus are planning to include basic synchronizing tools so that there won't be any conflicts between the threads.

The libraries

We are planning to provide different libraries that are needed by the user to code.

/*

Very briefly, these libraries are:

lang.miche – This is a standard library of the language. It contains base types, declarations of classes, objects, and basic i/o

basics.miche – contains some basics inbuilt dishes like making dough, hot water, boiled milk etc.

*/

1.3 The Michelin Environment:

Writing a code using our language is not only easy to code but also easy to read. We are trying to ensure safety and security with a decent performance which helps the

programmers as well as the users. Due to the ease of using the language, a programmer only needs to focus mainly on the recipe/process compared to coding it.

Example code snippet:

```
Glass g.
Ingredient ing1("Water", 100, volume, g).
Bowl b.
b~>add(ing1). \*adding ing1 to a bowl b and heating it for 5 mins at 60 degrees*\
b~>boil(5, 60).
g~>transfer(b).          \*adding the boiled water to the glass g(from b)*\
serve g.
ready.
```

We can see how the code is pretty intuitive and easy to understand.

Chapter-2
A Programmer's Toolbox

2.1 Primitive Data Types

In the libraries of our language, we use various data types of C++ like int, double, string, etc as the ones we work with. To compare and measure different quantities in Michelin, we use classes to store a few basic data types related to the class and their scaled quantities as well.

Examples:

int a = 100.
string s = "Water".

For example, to measure a solid class, we would have multiple measurement quantities like grams, milligrams, kilograms, and so on in a single class.

2.1.1 Measurement Units

In Michelin, we use various scales and different units to suit us appropriately to measure the quantities to help us in cooking. It has metric units and also has some daily life objects we use to put them in as well.

Some of the metric units are:

- Volume
- Weight
- Temperature
- Time
- quantity
- number

These measurements are done appropriately and the data types assigned to each are based on their general usage and are typically either int or double.

Example code snippet:

glass g.
Ingredient w(Water, 100, volume, g).
plate p.
Ingredient f(fruit, 7, unit, p).

Here we can see how the 100 is read as a volume type and stored in glass. and similarly unit with the fruit.

2.1.2 Boolean Data Type

This is an 8-bit data type with YES or NO as output. We use it to get info on basic questions.

bool x = YES.

2.2 Arithmetic and Relational Operators

2.2.1 Arithmetic Operators

Operator	Description
+	Adds two operands.
−	Subtracts the second operand from the first.
*	Multiplies both operands.
/	Divides numerator by de-numerator.

%	Modulus Operator and the remainder after an integer division.
++	The increment operator increases the integer value by one.
--	The decrement operator decreases the integer value by one.

Example code snippet:

1.
Example for + operator
let a, b be 2 vessels
int temp = a~>get_volume() + b~>get_volume().
** Adding volumes of a, b & storing it in temp*\.*

2.
Example for * operator
let g be a glass
*Ingredient ing("water", 4*g~>get_volume(), volume, b).*
volume of water is 4 times that of volume of glass g

2.2.2 Relational Operators

Operator	Description
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.
>	Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.

<	Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.
>=	Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.
<=	Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.

Example:

bool temp = (i < j).

2.2.3 Logical Operators

Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.

Example:

bool temp = ((i < j) && (a < b)).

2.2.4 Assignment Operators

Operator	Description
=	Simple assignment operator. Assigns values from right side operands to left side operand
+=	Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.

Example:

i += g~>get_volume().

2.2.5 Miscellaneous operators

Operator	Description
&	Returns the address of a variable.
? :	Conditional Expression.
~>	Accessing class methods.

2.3 Operator Precedence Table

Category	Operator	Associativity
Postfix	() [] ++ - -	Left to right
Accessor	~>	Left to Right
Unary	+ - ! ~ ++ - - & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right

Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= &= ^= |=	Right to left
Comma	,	Left to right

2.4 Arrays

Just like any other language, Arrays in Michelin are used to store multiple data of the same type together at one place.

Multidimensional arrays are supported by Michelin.

Example:
Plate p.
Bowl b.
Ingredient x[i].
*x[0] = new Ingredient(“Apples”, 2, unit, p). *name, quantity, typeOfQuantity, storeIn**
x[1] = new Ingredient(“Grapes”, 7, unit, b).
x[2] = new Ingredient(“Berries”, 7, unit, b).

2.5 Strings

Strings exist in our language for multiple purposes. Some of them can be said as:

- To send in some specific instructions like saying to chop the onions into smaller pieces.
- For input/output requirements, etc.

Strings are also used in the language to specify the type(/state) of the ingredient stored, dishes, etc.

Example:
String s = "Wheat flour".
bowl b1.
Ingredient item(s , 250, g,b1).

2.6 Memory Cleanup

We use destructors to clean up the memory with which we can automatically free up the memory used by those particular classes.

2.7 Functions

Functions are a set of instructions bundled together to achieve a specific outcome. They are a good alternative to having repeating blocks of code in a program. Functions like boiling which do not require a specific instance of a class to call can be declared as global functions.

2.8 Function Overloading

Function overloading means using a function that operates differently for different types of parameters.

In the case of cooking for heating in the oven, the oven should work irrespective of whether materials which want to heat are solid or liquid.

Example:

```
recipe x(){
        Glass g.
        Ingredient ing[4].
        ing[0] = new Ingredient (“CoolWater”, 100, volume, g]).
        TableSpoon tb.
        ing[1] = new Ingredient (“Lemon juice”, 2*tb~>get_volume(), volume, g).
        ing[2] = new Ingredient(“Sugar”, 3*tb~>get_volume(), volume, g).
        ing[3] = new Ingredient(“Salt”, (tb~>get_volume())/2, volume, g).
        int j = 0.
        while( j < 4)  {
                g~>add(ing[ j ]).
                g~>stir(0.5).
                j++.
        }
        serve g.
}

for(i = 0; i < 5; i++) x().
ready.
```

Chapter-3
Object-Oriented Programming Welcomes Michelin

When we initially thought of the language, it quite quickly came to our attention that this language is by nature, object-oriented! Why? Well, let's dive head-first into the principles of object-oriented programming and see how the whole idea of cooking a recipe blends into it.

Here are some of the primary principles of object-oriented programming (we'll just state them briefly here since the principles themselves are quite famous and our focus here is more on how Michelin incorporates these ideals) :

3.1 Encapsulation

Encapsulation is the mechanism of hiding data implementations by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this. It is basically a way of grouping data and "scripts" that tell us how to handle the data (functions, in a programmer's dictionary!) cleverly so as to make usage of the whole platform easy for the user.

This mechanism is required by our language to ensure that a user doesn't try and make changes to certain attributes declared in classes.

For example, a certain ingredient class will have a state attribute to which we may not want the user to have access, or another case would be when we may not want the users to change the dimensions of a container object. This mechanism ensures that such events don't occur.

Consider the following example:

* *User code* *\

Vessel v.
v.dimensions = 1. * *Error.* *\

This statement will return an error as dimensions is a private attribute of the vessel and the user cannot make changes to it.

3.2 Abstraction

Abstraction refers to the idea of describing the intent of a certain class or interface at a high level without delving into the low-level details of its implementation. Many times, all we need to know from a class is what it does and which subroutines or methods do we need to use in order for it to lend some desired functionality.

In the context of cooking, we don't need a lot of different types of objects. We came up with 3 high-level classes that describe most of the functionalities that we need. These classes are: "Container", "Tools", and "Ingredients." The Container class, for example, represents objects such as pan, vessel, cauldron, cooker, etc. The Tools class represents objects such as ladle, spoon, knife, peeler, stirrer, etc. The Ingredients class is slightly more complex to explain but for now, in the context of abstraction, we just mention that some of its child classes are "Potato", "Milk", etc.

Ok, so where is the abstraction in all this?

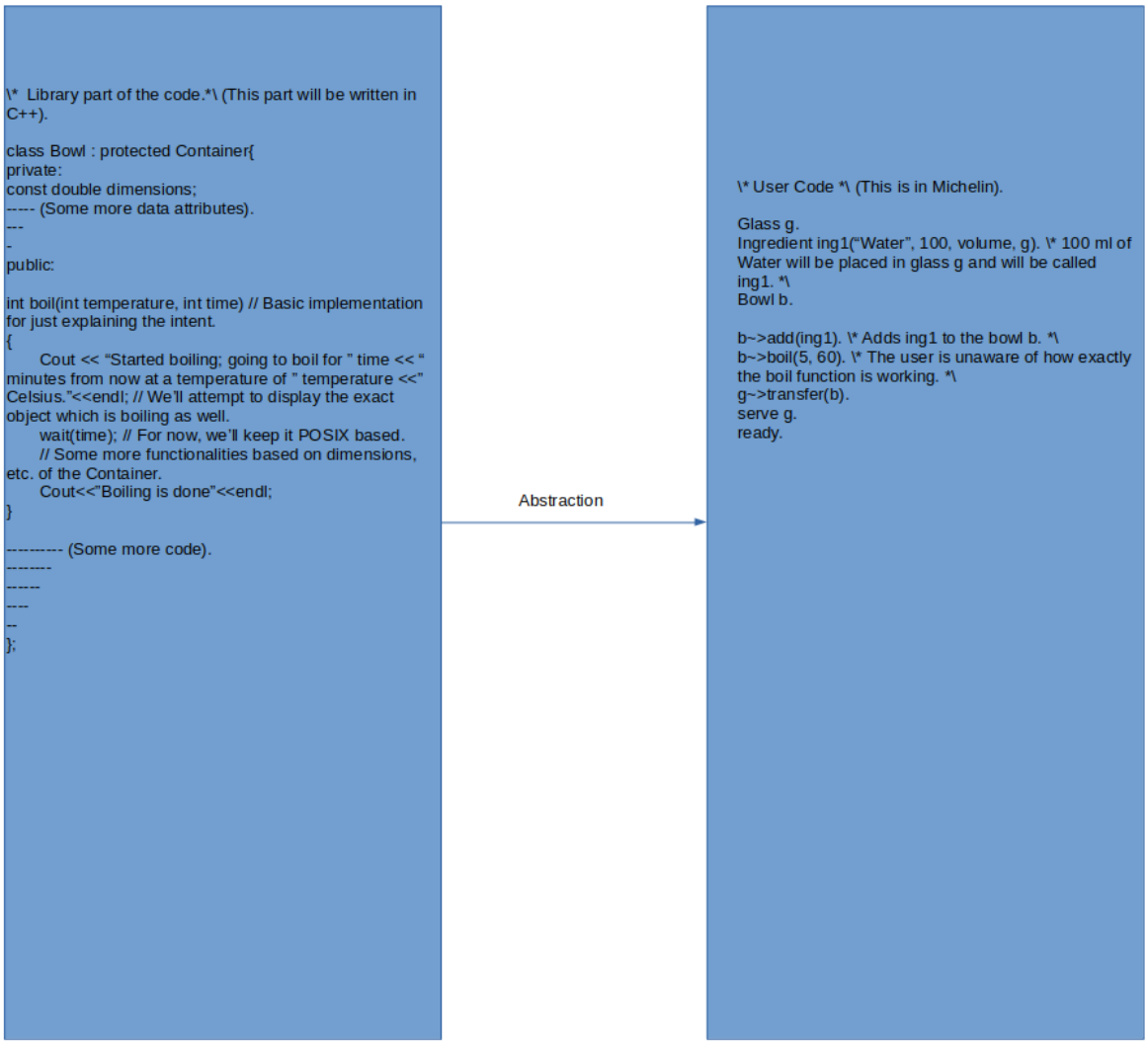
Well, the abstraction comes from the fact that all that has been mentioned above is what the classes represent and we haven't really revealed how these classes aka blueprints are actually internally storing these ideas.

For example in the Containers class, we will be having a "boil" function common across all subclasses of the Containers class, because we believe that most cooking containers can be used to boil a particular ingredient.

Now when the user writes code and uses the “boil” function, the user is unaware of how exactly it has been implemented for the particular instance of the Container class that he declared (say vessel)!

However, the overall blueprint self-explained by the class is sufficient for the user to use the desired functionality.

Consider the following example:

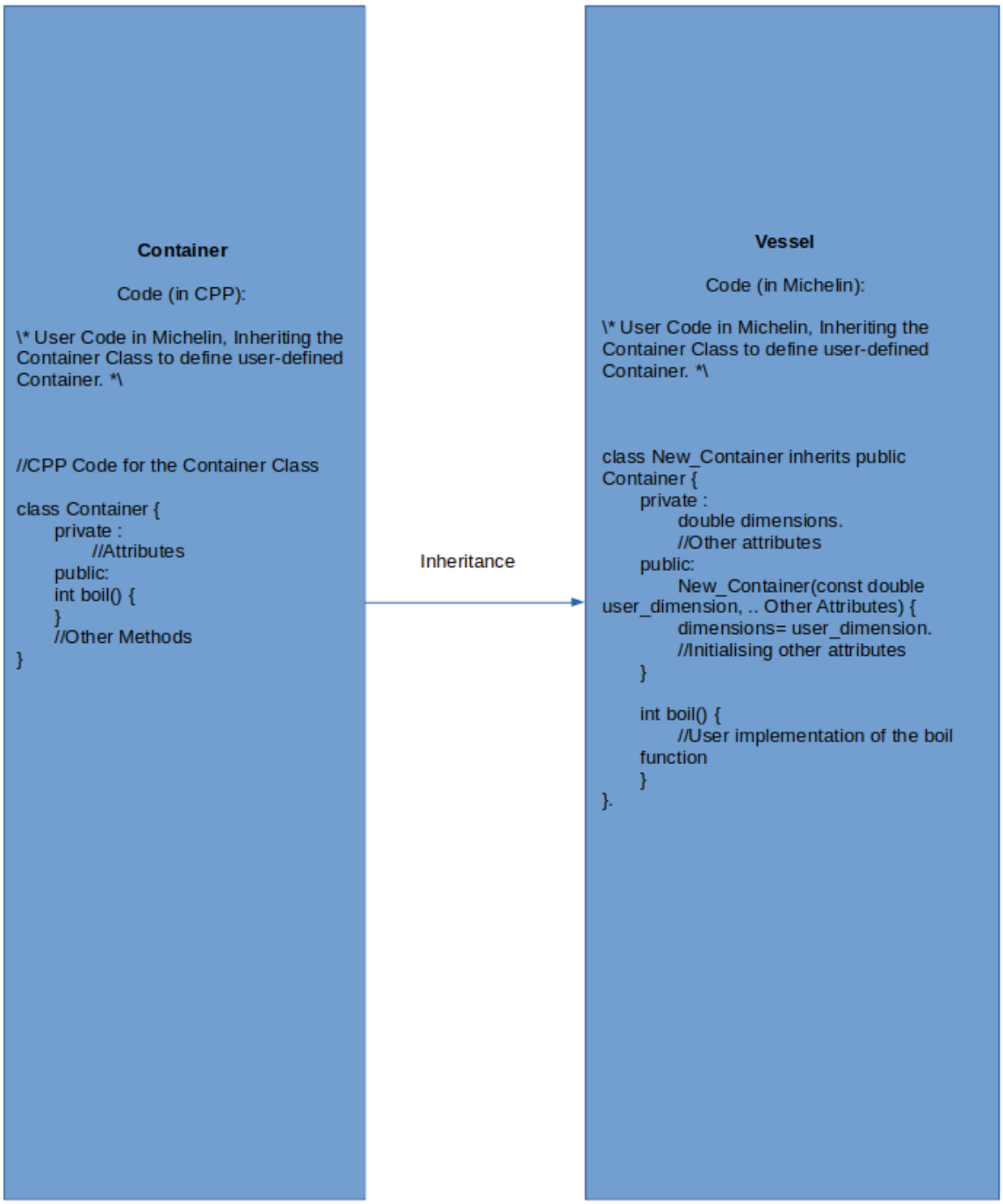


3.3 Inheritance

Inheritances express an “is-a'' relationship between two classes. We can reuse the code of existing superclasses in the derived classes using inheritance. It also helps us better understand how one class depends on the functionality of another class, which is usually called the parent class.

Extending from the previous description, we see that inheritance is crucial for the way we conceive of the cooking process in Michelin! In fact, if we take note of the Containers class, we see that within the Containers class, we have some state-based subclasses. One example of a state-based class could be based on the Vessel subclass present as a child class of the Containers class. We have SmallVessel and LargeVessel as subclasses of the Vessel class. Now both SmallVessel and LargeVessel will inherit the boil function present in the Vessel class, and might make a few modifications to the boil class present in the Vessel class in relation to the size (e.g. constants related to max_temperature that the vessel can be boiled given that the Small/Large Vessel object has initially been filled to x% of the vessel capacity).

There is also another kind of behavior that is closely related to, if not identical to inheritance that we wish to add into our language. This is in relation to the idea that sometimes one may want to use the output of a particular sub recipe in another larger recipe (somewhat analogous to C++’s lambda functions)



The above example shows Inheritance in Michelin.

3.4 Polymorphism

In Greek, poly refers to “many/multiple” while morph refers to “shape/form.” As can be inferred from the etymological break up of the word, polymorphism refers to the idea of a single entity externally having the same form but performing different actions. Some common ways polymorphism is demonstrated is through the ideas of runtime and compile-time polymorphisms.

A simple example of this in our language would be in the following scenario.

Let’s suppose that we want to add multiple kinds of ingredients to a single pan, so for this example, we could take milk and sugar. Now milk and sugar are fundamentally different entities in regards to the state itself (milk is a liquid ingredient while sugar is a solid ingredient).

But we provide the ability to store both of these “different” entities in a single array of type “Ingredient.” This is an example of polymorphism in the context of storing data. This is just the tip of the iceberg in terms of its functionality.

Here’s an example code snippet demonstrating the use of overriding a particular function.

Example:

```
\* User Code in Michelin. *\
class Custom_container inherits public Container{
        Double dimensions.
        override int boil(int time){
                //my own implementation of the boil function
        }
}
```

Here what we are doing is creating our own custom class and inheriting a predefined class and overriding the boil method to allow the user to make his/her own implementation of the boil method. This acts as another example of abstraction that the user herself/himself can use.

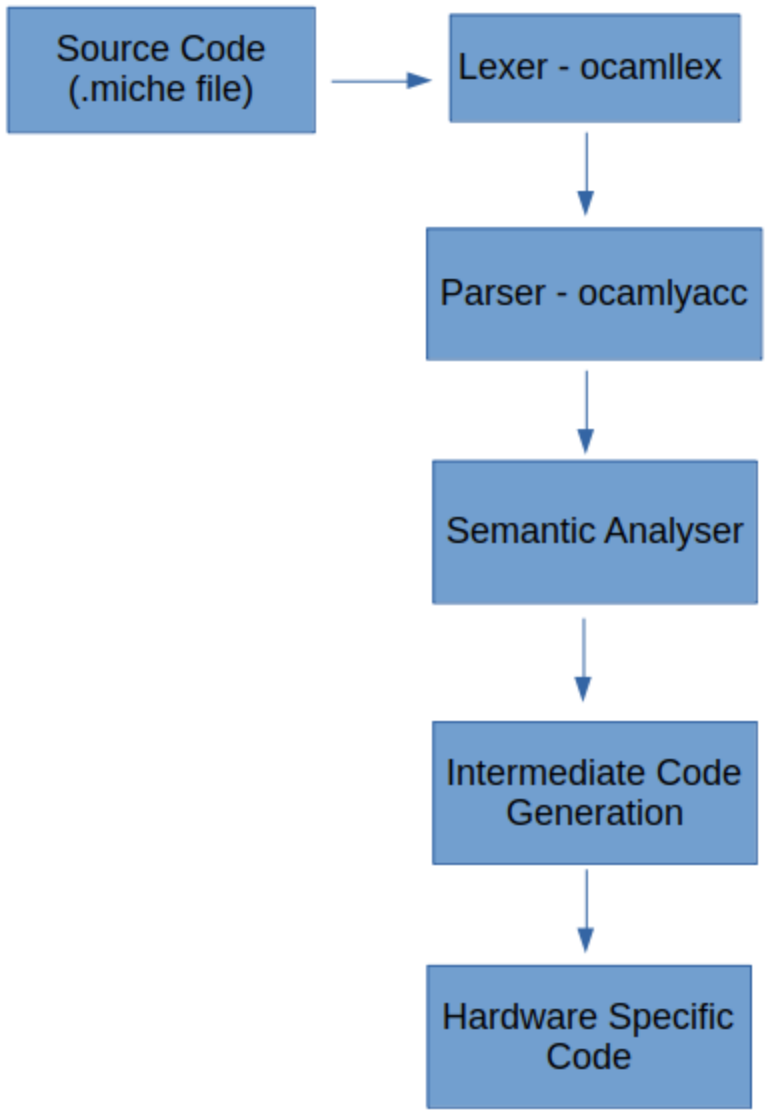
3.5 Other Details

1. All the classes defined in Michelin Language will be redefined in CPP internally.
2. By default, all the functions in class Container and class Tools are virtual.
3. The user cannot define a function as virtual for her/his classes.

Now from the above-given examples, it can clearly be seen why we choose to design our language using the OOP paradigm. To summarize in a nutshell everything in a kitchen can be visualized to be an object with certain properties and our goal is to make use of these properties to make a certain desired delicious output.

Chapter-4
Compiler Architecture of Michelin

The Compiler Architecture of the Michelin language is as follows:



4.1 Lexer

The Lexer takes in the ".miche" (The extension of Michelin Language - .miche) program and converts it to tokens. The Lexer generates tokens using regular expressions and pattern rules. The Lexer discards all the white spaces and the comments present in the program. The stream of tokens generated are passed to the Parser. The Lexer of Michelin has been implemented using Ocamllex.

4.2 Parser

The Parser takes in the stream of tokens generated by the Lexer. The parser generates the syntax tree using the Context Free Grammar. The parser detects any syntax errors present in the program. The implemented parser is a "Look Ahead LR" parser. The Parser for Michelin has been implemented using Ocamlyacc.

4.3 Semantic Analyser

The Semantic Analyser uses the syntax tree generated by the parser to check the source code for semantic consistency with the language definition. An important purpose of semantic analyser is type-checking. If there are no semantic errors, the semantically checked syntax tree is used for intermediate code generation.

4.4 Intermediate Code Generation

The intermediate code that is generated is not architecture-neutral. Our main aim is to generate a code that is understood by particular hardware (like in Arduino programming as an example). In this case, the hardware is the particular device (hypothetical machine) that will actually be cooking the dishes as per our programming commands.

Moreover, there are some commands that we plan to translate into database queries. An example of such a statement is where we want to fetch a certain amount of rice

for a particular dish that we wish to make from the food inventory (connected to the device). We will try to make the language in such a way that the user just has to write a very simple statement for fetching that but internally it gets translated into a query from the map in which we will be storing the ingredients.

However, we are not sure how exactly we can construct this device right now. So we thought of two options to come up with an implementation for the purpose of testing it.

One plan was to replace the "hardware related" functions such as boil(), etc. into Python code where we show for example a graphical image of a pot being boiled; this in no way was intended to be a part of the compiler. It was just an idea we came up with to test the validity of our language. More specifically, we were planning of transferring the output of the AST into a Python code (via a translator or something of that sort) which will actually generate the necessary images such as stoves, ingredients, etc. on its console (we were planning of doing this since Python has a very convenient GUI library for animations -- manim). This does seem somewhat hard to implement and we seek input on its viability and ease of implementation.

The other idea which is much simpler is to demonstrate that we just show a log file for each code in regards to in which object what is boiling (mentioned this as an example) and between which containers was there a transfer operation, etc.

We seek input on which of these 2 methods is better to proceed with (again, this is solely for the sake of testing).

4.5 Hardware Specific Code

The final code that is generated is compatible with a hypothetical machine that can cook. As mentioned above, this stage doesn't have much use for us currently, since we can't design the hardware for the hypothetical machine that we have proposed right now. **But we are not very sure if we can somehow include some hardware-related aspect practically in the current proposal of the language and seek input on this**.

Chapter-5
Security In Michelin

5.1 Memory Layout

Michelin does not make use of pointers like in C and C++. The Michelin compiled code references memory via symbolic handles that are resolved to real memory addresses at run time.

5.2 Limitation checks

Since our language is used to instruct a machine to prepare a delicious dish, it is our duty as language designers to run a check on whether or not the given recipe input is feasible for the hardware system or not.

So we have implemented a security check level that runs a simulation of the given recipe and checks whether or not it can run within the limitations of the hardware system. If these checks weren't done then although logically it may seem correct, it may cause wastage of ingredients.

For example, if a vessel object of a particular volume is created and is used to store milk. If the user adds excess milk into the vessel it will overflow and lead to lots of wastage. So before the code is executed we throw an error and ask for this to be fixed. Similarly, we don't only check for semantic errors, we do device-dependent checks as well like ensuring the user doesn't try cooking on stoves that aren't present on the device or accessing a mode not present on the device.

Chapter-6
Multithreading in Michelin

Multithreading is a highly compatible feature with our language which includes various processes that can happen parallelly thereby increasing the efficiency of the code/process.

We plan to explicitly program the thread support to help the user to parallelize the various processes while ensuring that they can be implemented easily.

This process of parallelization is highly dependent on the users and we plan to make the library thread-safe.

For example, multiple stoves can run at the same time, accessing the ingredients and other resources parallelly, multiple chefs using the oven or refrigerator simultaneously, etc.

Possible Ideas for implementing thread support:

1. Creating a library from scratch that can interact with the OS.
2. Trying to involve the C or C++ library of threads into our language.

Due to the time constraints present on the project and due to the extra knowledge required which involves the communication between the language/compiler with the operating system to implement parallel programming (multithreading), we have decided to not add this feature to our language as it may end up taking too much time away from the designing process for other important features.