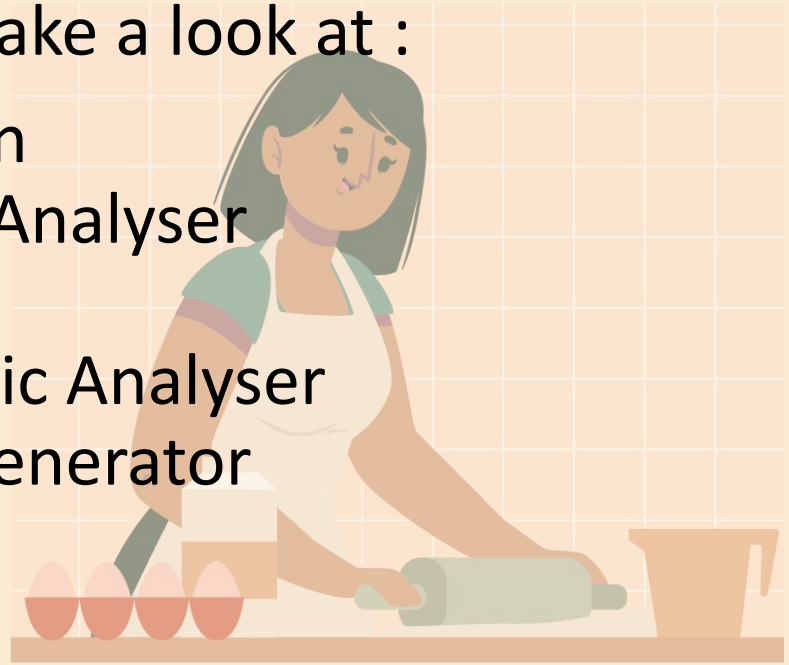
MICHELIN

A Code which Cooks

Contents

Let us take a look into the design of Michelin compiler.
In these slides, we'll take a look at :

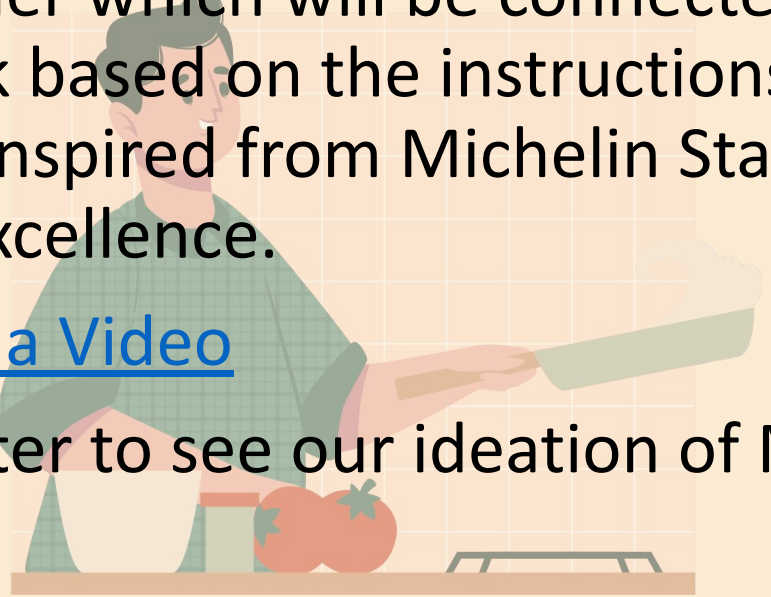
1. Making of Michelin
2. Michelin's Lexical Analyser
3. Michelin's Parser
4. Michelin's Semantic Analyser
5. Michelin's Code Generator



Making of Michelin

Why Michelin?

The idea we started off with Michelin was to build a coding language and a compiler which will be connected to some kind of a machine that will cook based on the instructions from the code. We named it Michelin as inspired from Michelin Star, the ultimate hallmark of culinary excellence.



Idea in the form of a Video

Check this video later to see our ideation of Michelin applied into real life.

Adoption from OCaml

The following are some of the reasons why we chose to go ahead with OCaml for building our Compiler:-

- A hand-coded lexer/parser was definitely possible to build, especially given that we don’t have a lot of keywords right now; however we did not pursue the idea since extending the features of the language might get very messy later on.
- It was hard to decide between Flex/Bison and OCamllex/yacc but we finally decided to go with OCaml because we wanted to try some functional programming (as a new skill).

Michelin’s Lexical Analyzer

Lexing the Lexer!

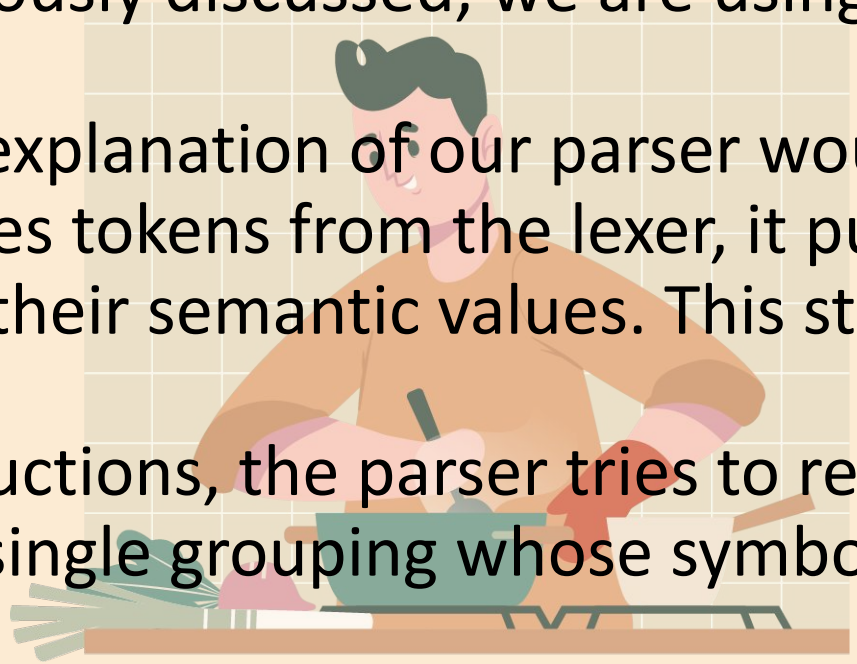
- The lexer.mll file has the information regarding the type of each token which our language recognizes.
- Tokens such as digits,characters,integer constants,decimal constants and string literals are represented using regular expressions which is a functionality provided by ocamllex.
- Then we go on about doing pattern matching of the tokens and return the corresponding token name and line number the token was found at for each valid lexeme of our language.
- The main.ml then sends the input code to the lexer and then reads the lexbuf which is a buffer provided by ocamllex for tokens.

Michelin's Parser

Background of the Parser

As we have previously discussed, we are using Ocamlyacc to make our parser.

A brief subjective explanation of our parser would be:

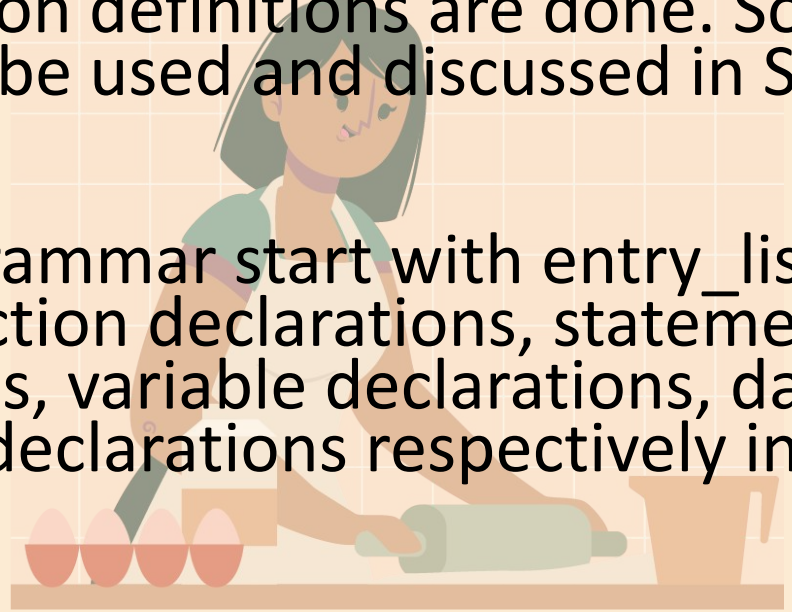


- As the parser takes tokens from the lexer, it pushes them onto a stack along with their semantic values. This stack is called as parser stack.
- By shifts and reductions, the parser tries to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.
- Hence our parser is basically a bottom-up parser.

Structural Overview

The structure of Michelin's parser starts with types of how variable definitions, operator definitions, class definitions, argument definitions and function definitions are done. Some of the operators which will effectively be used and discussed in Semantic analyser are also added.

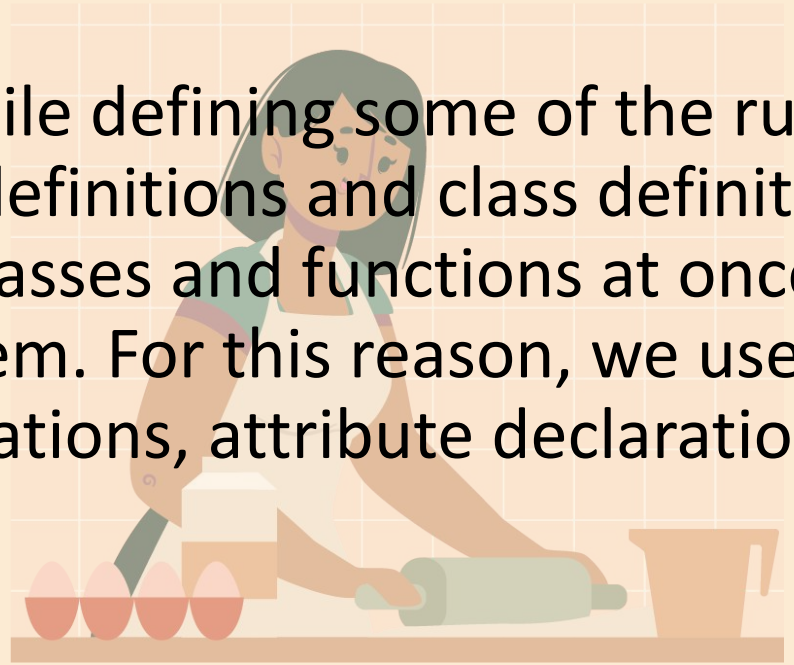
The rules of our grammar start with entry_list, then we go forward to entry, function declarations, statement declarations, argument declarations, variable declarations, data type declarations and then expression declarations respectively in the same order.



Structural Overview

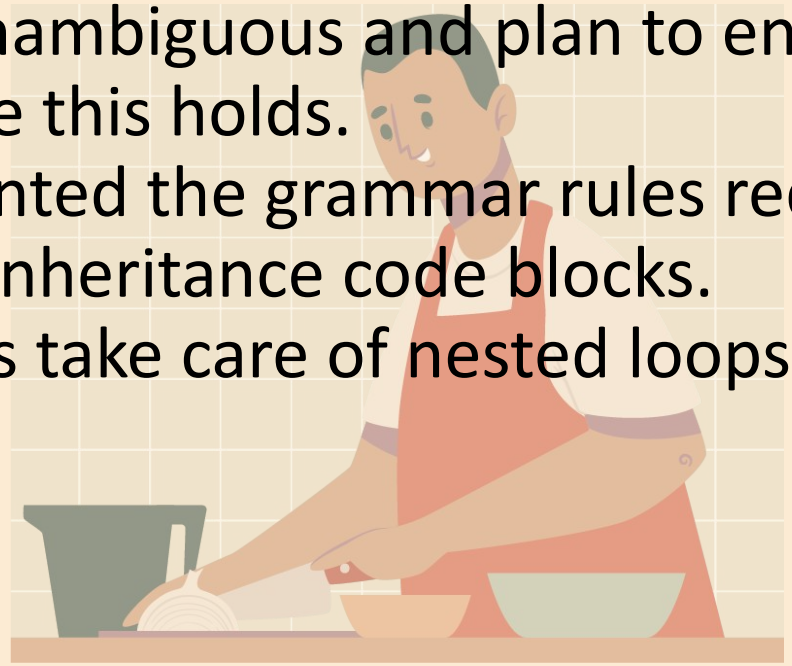
Usage as Lists

We faced an issue while defining some of the rules for our parser. In writing the function definitions and class definitions, we were not able to define both classes and functions at once with the respective tokens for each of them. For this reason, we used list structure to parse function declarations, attribute declarations and body of the code as well.



Key Points to Notice

- Our compiler is a LALR parser.
- Our grammar is unambiguous and plan to ensure that any further rules added ensure this holds.
- We have implemented the grammar rules required to parse through the class inheritance code blocks.
- Our grammar rules take care of nested loops as well.

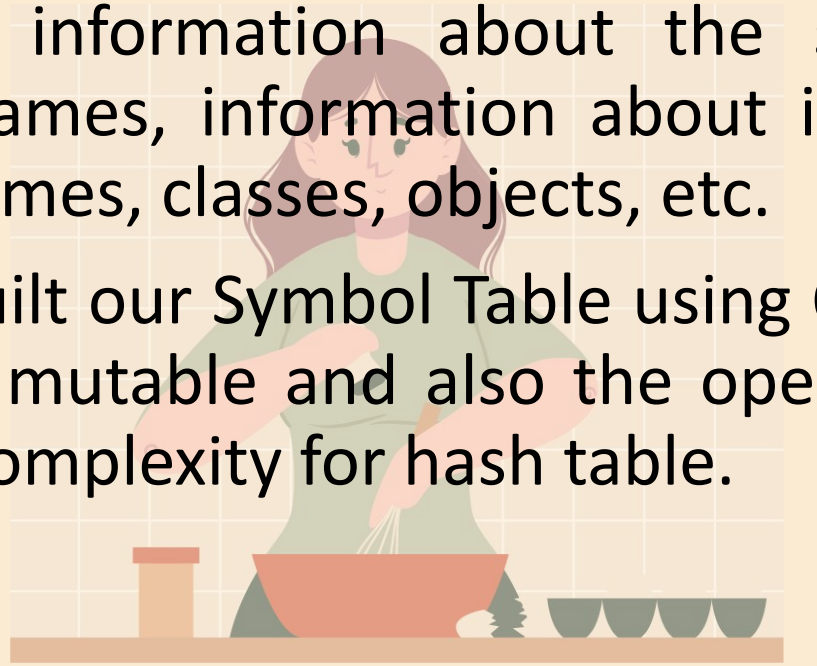


Michelin's Semantic Analyzer

Symbol Table

Symbol Table is an essential data structure in a compiler which keeps track of the information about the scope and binding information about names, information about instances of various variables, function names, classes, objects, etc.

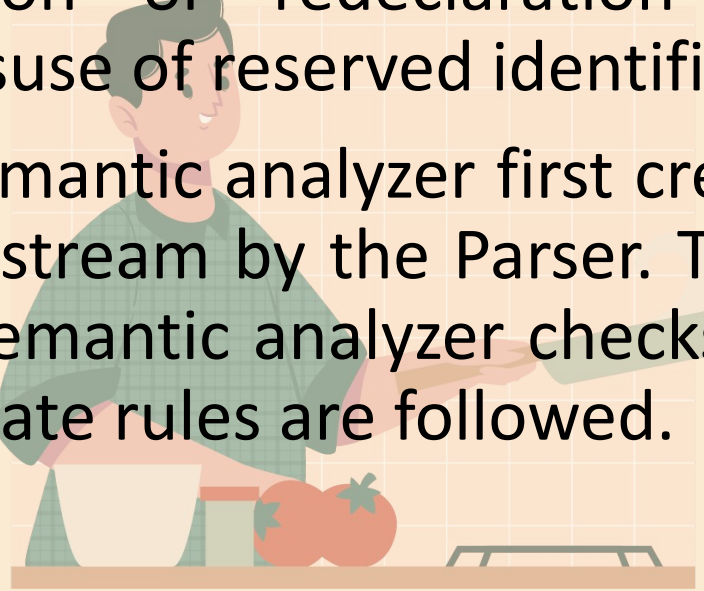
In Michelin, we built our Symbol Table using Ocaml's inbuilt hash table feature as it is mutable and also the operations such as find need constant time complexity for hash table.



Implementation of Semantic Analyzer

The job of a semantic analyzer is to check the type definition errors; no declaration or redeclaration of variables/classes/ functions; and any misuse of reserved identifiers.

In Michelin, the Semantic analyzer first creates the Symbol Table from the given token stream by the Parser. Then for every variable and it's declaration, semantic analyzer checks the symbol table and verifies if the appropriate rules are followed.



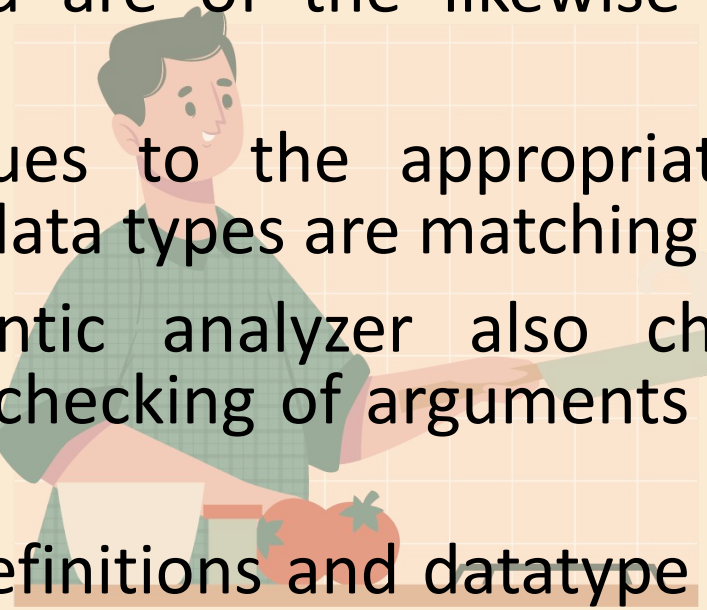
Implementation of Semantic Analyzer

Then the Semantic analyzer checks if all the arithmetic operations performed are of the likewise variable types and is allowed.

The assigned values to the appropriate variables are then checked to see if the data types are matching or not.

Then the Semantic analyzer also checks out for proper declaration and type checking of arguments for both functions and classes

The appropriate definitions and datatype matching of arrays are also checked.

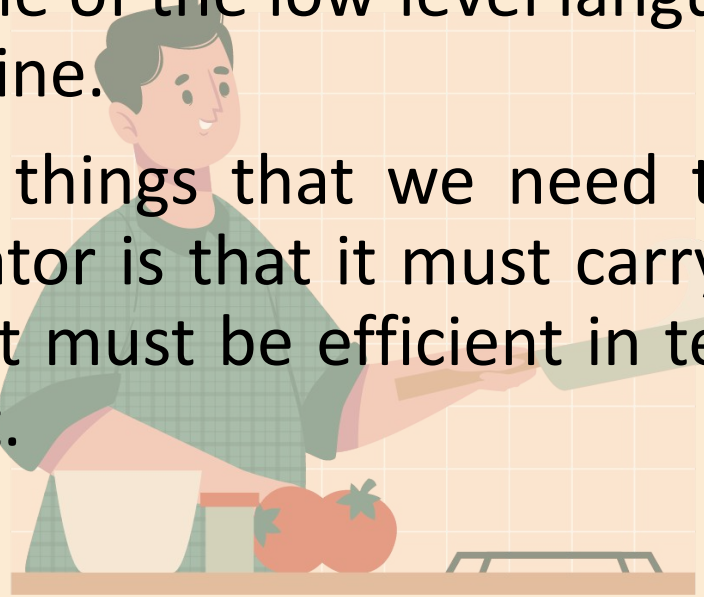


Michelin's Code Generator

Design of the Code Generator

In general, the job of a Code generator is to break down the code into object code of one of the low-level languages so that be readily executed by the machine.

Few of the major things that we need to be concerned while making a code generator is that it must carry the exact meaning of the source code and it must be efficient in terms of CPU usage and memory management.



Design of the Code Generator

For the process of Code Generation in Michelin, we chose LLVM IR as our low-level language because of its modular design allows its functionality to be adapted and reused very easily. Our code generator creates a file of LLVM IR object code of the Michelin Language which can then be executed by the machine to generate the output accordingly.

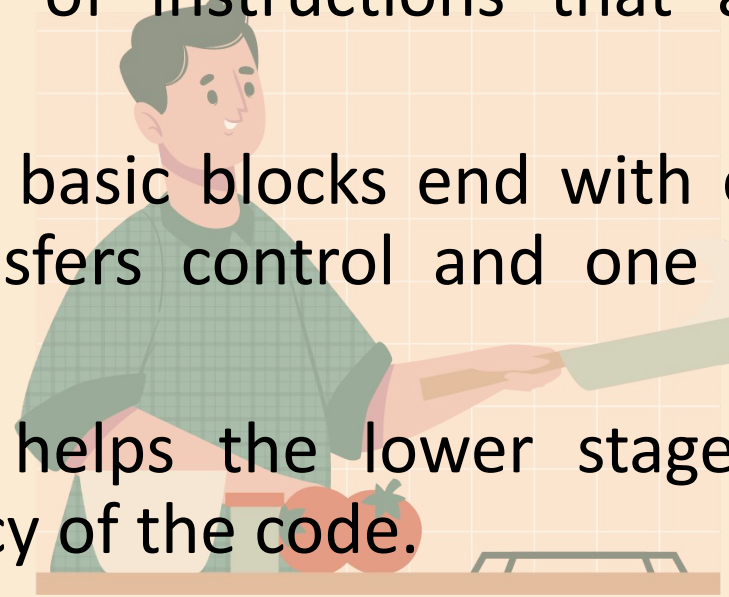
We used Llvm and Ast modules in our Code Generator. Llvm module provides an OCaml API for the LLVM intermediate representation and access to the classes in the VMCore library. Ast module describes all of the syntactic constructs of OCaml.

Design of the Code Generator

Our Code Generator generates code of IR as basic blocks that consist of sequence of instructions that are in general always executed together.

Each of the basic basic blocks end with exactly one terminator instruction that transfers control and one enter instruction that begins the execution.

The overall CFG helps the lower stages of the compiler to optimise the efficiency of the code.



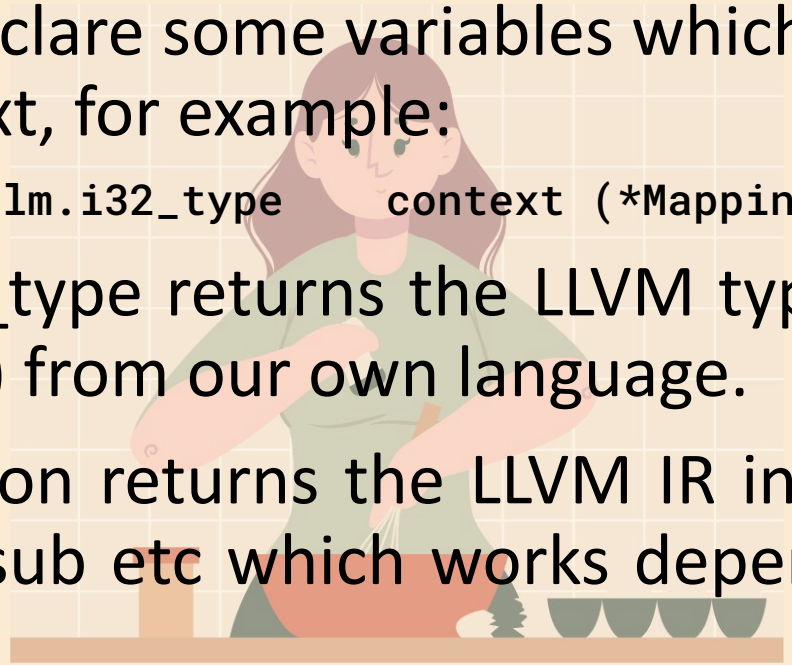
Code Generator Analysis

In Michelin's Code Generator, at first we import the llvm and ast modules. Then we declare some variables which help in inferring the types from the context, for example:

```
let int64_t     = lm.i32_type    context (*Mapping ints;*)
```

The variable llvm_type returns the LLVM type for a datatype (as defined in our Ast.ml) from our own language.

The builder function returns the LLVM IR instructions for binary operations like add, sub etc which works depending on the type of expressions.



Code Generator Analysis

The recursive function output_list takes the ast from parser and calls output_entry on each entry. In output_entry, we match the entry and if it is a function, we call function t1 which takes body of the function as input.

In function t1, we take statement list as input and iterates on it and calls function t2 for each statement. In function t2 the statement is matched with expression and calls the builder function or is matched with locals(ie declaration) and calls the t3 function.

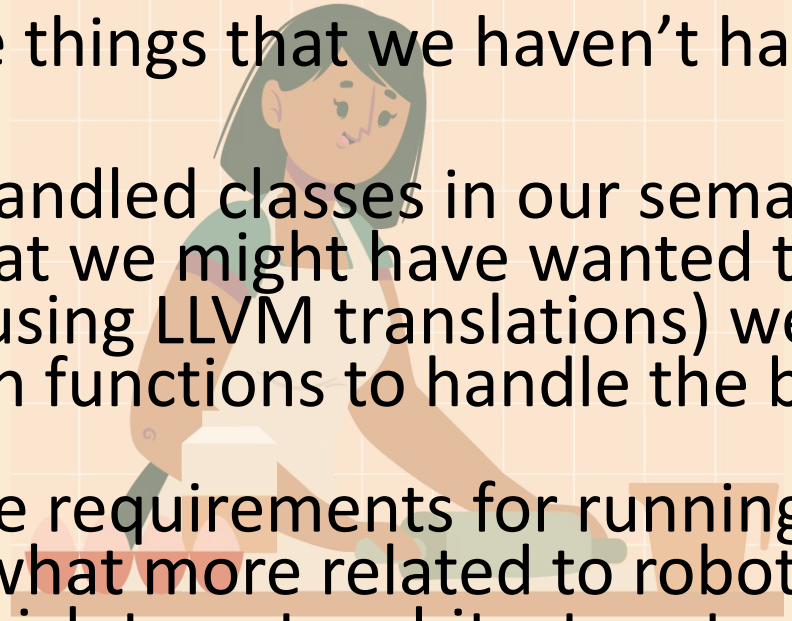
In function t3 we match the decl statement with attribute declaration the we call the builder function.

Further Additions Yet to be Done

We are yet to add few of the applications and features for the code generation phase.

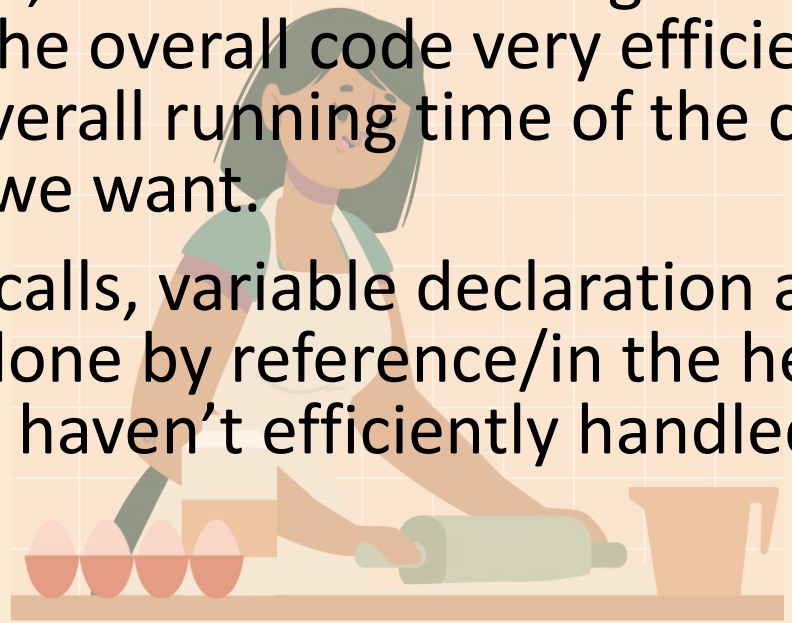
In particular, some things that we haven't handled as of now include:

1. We haven't fully handled classes in our semantic analysis yet, so some functions that we might have wanted to write for it in the code generation (using LLVM translations) were left out.
2. We haven't written functions to handle the basic blocks and branches.
3. Since the hardware requirements for running our language to full efficacy are somewhat more related to robotic hardware, we weren't sure of which target architecture to use as well.



Further Additions Yet to be Done

4. We have not made of LLVM's special passes for loop unrolling, tiling, etc. as well, which is something we would have liked to do, as it makes the overall code very efficient and also decreases the overall running time of the code, which is something that we want.
5. All our function calls, variable declaration are by default intended to be done by reference/in the heap, but in order to facilitate this we haven't efficiently handled pointer references yet.



Team Details

Sai Sidhardha Grandhi
CS19BTECH11050
Project Manager
GitHub ID: G-Sidhardha

Krishn Vishwas Kher
ES19BTECH11015
Language Guru
GitHub ID: KrishnKher

Mukkavalli Bharat Chandra
ES19BTECH11016
System Architect
GitHub ID: chandra3000

Sujeeth Reddy
ES19BTECH11022
System Integrator
GitHub ID: Sujeeth13

Sree Prathyush Chinta
CS19BTECH11043
Tester
GitHub ID: Prathyush-1886

Vemulapalli Aditya
CS19BTECH11025
System Integrator
GitHub ID: VEMULAPALLI-ADITYA

Praneeth Nistala
CS19BTECH11054
Tester
Github ID: Praneeth-Nistala

Thank You