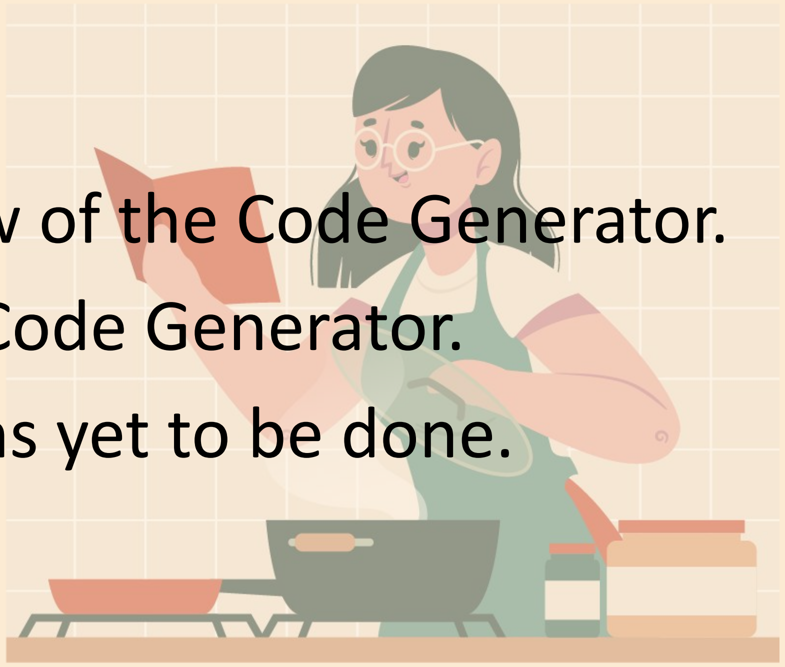
MICHELIN

A Code which Cooks

Michelin's Code Generator

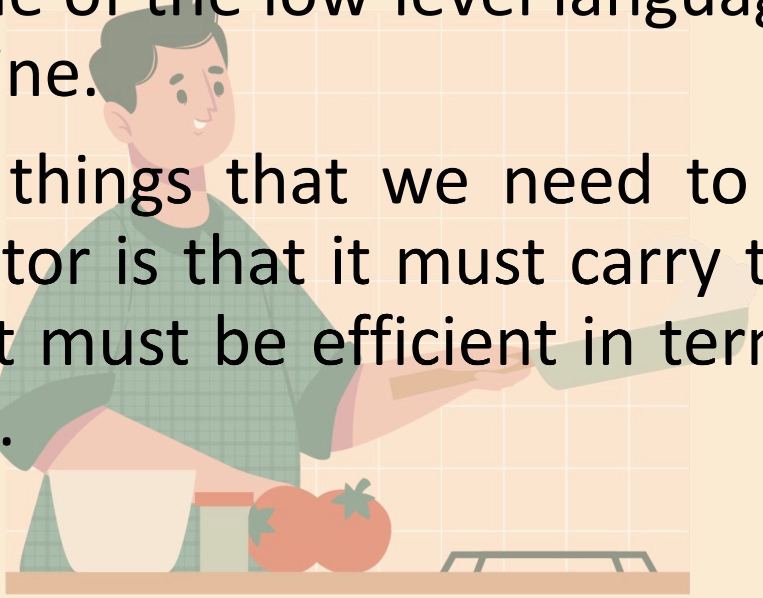
Contents

At this phase of the project, few important things to take a note of are:



1. Design Overview of the Code Generator.
2. Analysis of the Code Generator.
3. Further additions yet to be done.

Design of the Code Generator



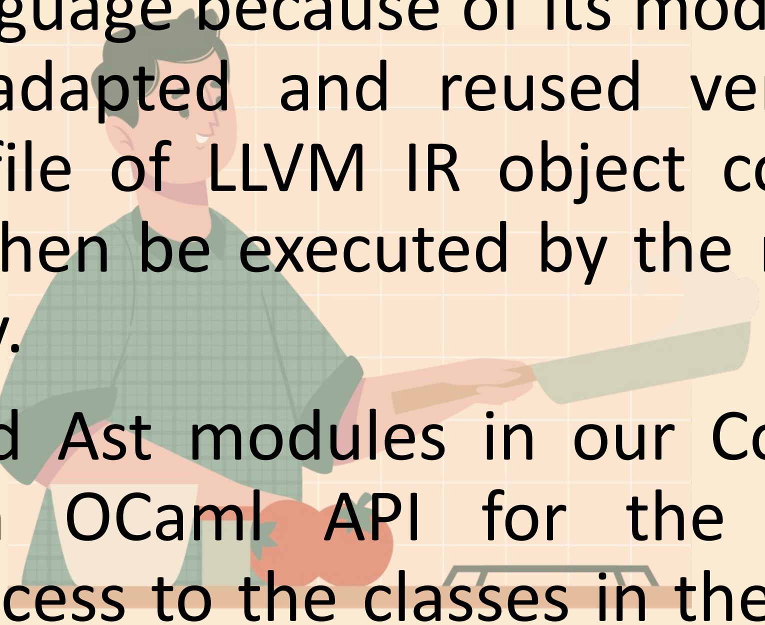
In general, the job of a Code generator is to break down the code into object code of one of the low-level languages so that be readily executed by the machine.

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.

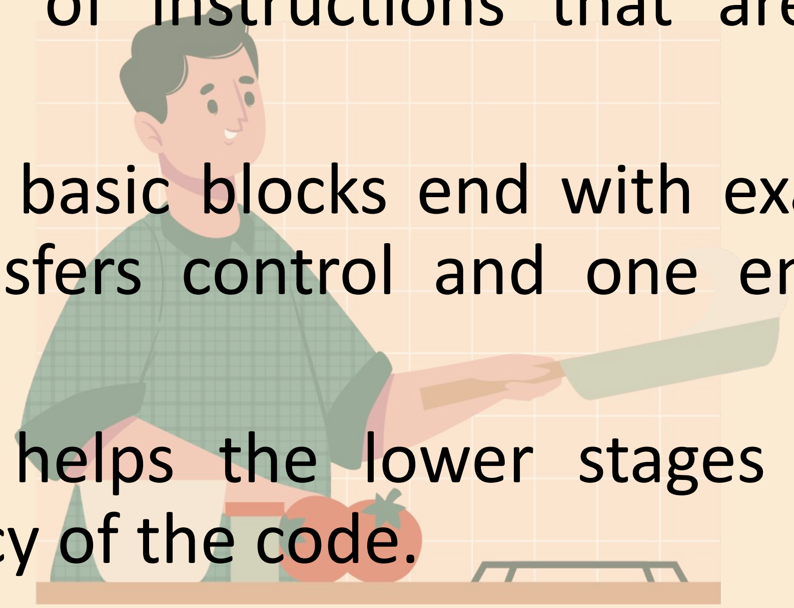
Design of the Code Generator

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

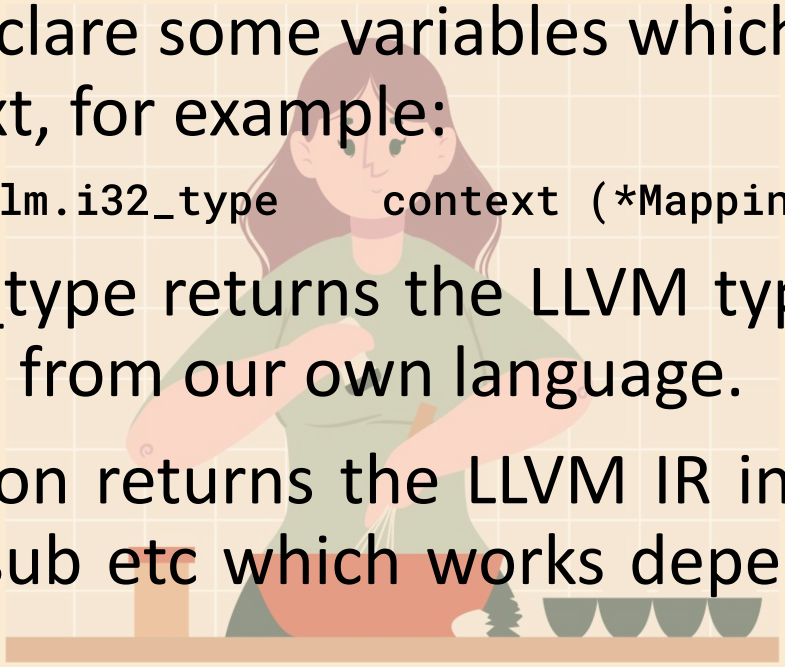


Our Code Generator generates code of IR as basic blocks that consist of sequence of instructions that are in general always executed together.

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



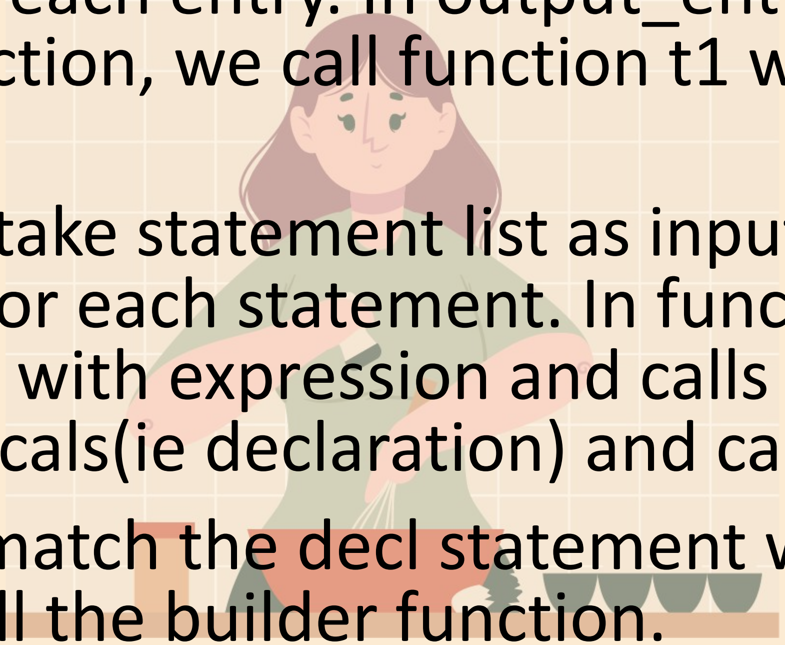
In Michelin's Code Generator, at first we import the llvm and ast modules. Then we declare some variables which help in inferring the types from the context, for example:

```
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

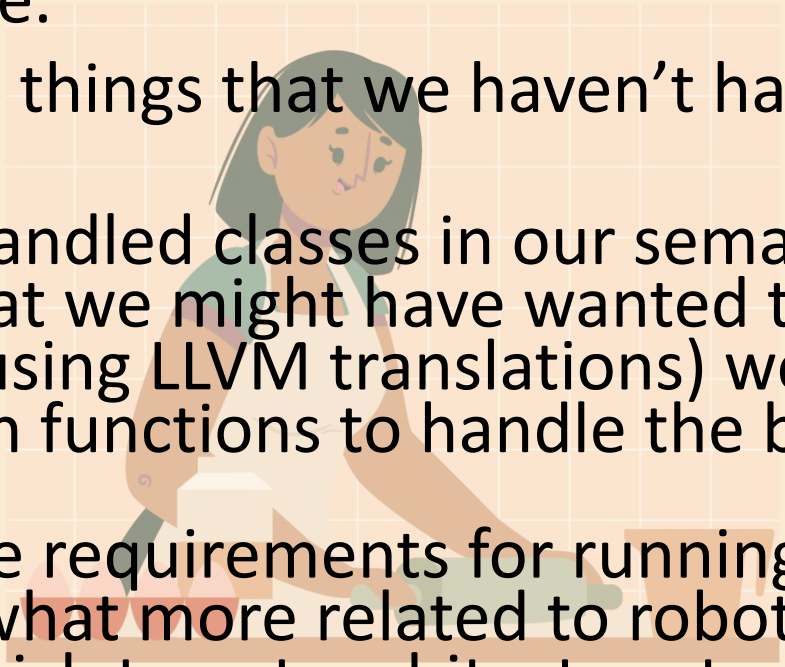


The recursive function output_list takes the ast from parser and calls output_entry on each entry. In output_entry, we match the entry and if it is a function, we call function t1 which takes body of the function as input.

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

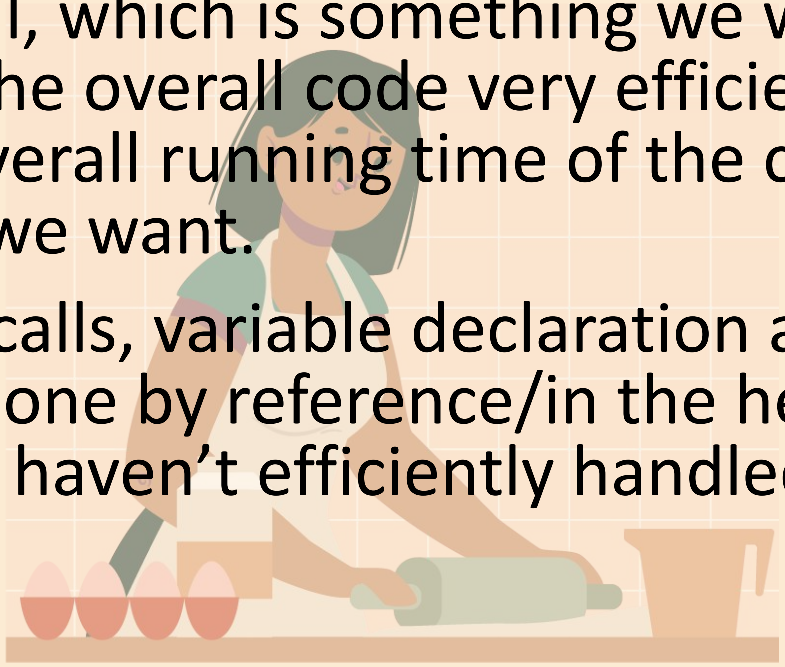


We are yet to add few of the applications and features for the code generation phase.

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