Contents

1 Pre-amble

The virtual machine for the Thymio requires bytecode to run, these are hexadecimal numbers of the type $0x2000$,$0x7001$, etc. and represent commands for the robot to execute. We can think of these values as "virtual" machine code. A disassembler also exists for these values. For example, $0x8002$ is ADD, and so on. We can think of this disassembled code as the Assembly bytecode instructions. Since the disassembled code is significantly easier to debug, this compiler generates Assembly bytecode instructions.
More importantly, this document will refer to the Assembly bytecode instructions as op-code, for the sake of brevity.

2 Intermediate code utilities

These constructs and definitions serve to be used through out the code generation and printing of the intermediate code, and will be analysed individually.

2.1 Storing the intermediate code

```
input_file   = None
code         = bytearray()
globals      = {}
locals       = {}
globals_n    = 0
locals_n     = 0
word_size    = 4

var_offset = 200
stop_flag = 0
scope_flag = globals
```

The *input_file* variable is for storing the source code we want to compile. The intermediate code is stored in the variable *code*, which is a **bytearray**. This is an artefact of the first-most implementation of this compiler, inspired by the compiler in:
https : //rosettacode.org/wiki/Compiler/code_generator
This **bytearray** requires a *word_size* to function properly, and has a value of 4. Do not change this number, everything will crash.
There are important consequences of this choice.
Firstly, the intermediate code must have the opcode encoded numerically. This may seem problematic, but is quite useful for printing the code. This encoding is done via dictionaries and definitions discussed in the next subsection. So for example, *ADD* must be encoded as 5, and so on.
Secondly, it restricts how much information we can store in a given line of the bytearray, and so we need to store extra information in memory constructs, discussed later. To visualise these limitations, consider table 1, representing some intermediary code in the bytearray:

line #	Byte0	Byte1	Byte2	Byte3	Byte4
1	*PUSH*	*5*	0	*0*	*0*
2	*PUSH*	*1*	0	*0*	*0*
3	*ADD*				
4	*JIN*	*K*	0	0	0

Table 1: Deconstruction of the bytearray for some opcode instructions.

Firstly, the numerical encoding of the opcode is 1:1, so in the visual table its sufficient just to have the opcode for visual presentation(even though in the actual bytearray it is encoded).
Secondly, opcode instructions that require an argument, such as *STORE* or *PUSH* generate 4 extra bytes, bytes1 to 4 in table 1. This reserves 4 bytes for the argument. However,

opcode that does not have an argument, like ADD, does not generate these extra bytes. Unfortunately, opcode like $jump.if.not(JIN)$ requires 2 arguments: 1 for the comparator, and 1 for the line number we wish to jump to(K). This cannot be stored in the bytearray as it will cause a crash. Therefore, the comparator needs to be stored in the memory constructs described later.
In order to access the elements of the bytearray we access them like a list, but we must take into account the extra bytes generated. For example, $code[0] = PUSH$, $code[1] = 5$ but $code[2] = 0$. So if we want to access the $PUSH$ opcode in line 2, this is found in $code[5]$. Similarly, $code[10] = ADD$, but $code[11] = JIN$. This is because the ADD opcode doesn't generate the extra 4 bytes. In order to indicate the correct byte addressing, we need to modify the line numbers.
Similarly, we would also like to keep track of the corresponding line number of the opcode instructions withing the Thymio.
Finally, since the bytes containing 0 are not of interest, we can ignore them.
This memory setup can be re-written as indicated in table 2:

opcode address	intermediate code address	opcode command	opcode argument
0	0	*PUSH*	*5*
2	5	*PUSH*	*1*
4	10	*ADD*	*N/A*
5	11	*JIN*	*K*

Table 2: Adjusted bytearray representation that will be used in this documentation

The "intermediate code address" corresponds to the addressing of the bytes in the bytearray, and "opcode address" is the line number of opcode as far as the Thymio is concerned. All subsequent representations of the intermediary code will be done like this.
The **globals/locals** dictionaries are responsible for storing the global/local variables along with their addresses. For example, consider the following source code:

```
a = 1
b = 2
```

Then we have **globals** = {'a':0, 'b':1}. Notice how the first address for the variables starts at 0. Since we can't use this in the Thymio, we have the var_offset variable to adjust the alignment of these addresses. This variable is used in the $code\_gen(x)$ and $list\_code()$ functions.
$globals\_n/locals\_n$ keeps track of the number global/local variables, so that if we need to introduce a new variable, its address will be the increment of these variables.
Finally, $stop\_flag$ is just for the compiler to determine where to emit the $STOP$ command (this will be elaborated later), and the $scope\_flag$ is used to indicate to the compiler what type of scope we are in.

2.2 Opcode encoding

As mentioned before, these are used to encode the opcode for the bytearray, along with some useful dictionaries for later.

```
LOAD, STORE, PUSH, ADD, SUB, MULT, DIV, MOD, LT, GT, LE, GE, EQ, NE, AND, OR,\
NEG, NOT, JUMP, JIN, PRTC, PRTS, PRTI, STOP, CALLSUB, RET, DC1,DC2, CALLNAT,\
    STORE_IND,LOAD_IND = range(31)

operators = {ast.Lt: LT, ast.Gt: GT, ast.LtE: LE, ast.GtE: GE, ast.Eq: EQ, \
                ast.NotEq: NE, ast.And: AND, ast.Or: OR, ast.Sub: SUB,\
                ast.Add: ADD, ast.Div: DIV, ast.Mult: MULT, ast.Mod: MOD}


comp_ops = {LT:"lt", GT:"gt", LE:"le", GE:"ge", EQ:"eq", NE:"ne"}
```

The encoding of the opcode is done via the **range()** function, so $LOAD = 0$, and so on. Among these opcodes, $PRTC$,$PRTS$, and $PRTI$ are not used at all, and are an artefact from the aforementioned compiler.
$DC1$ and $DC2$ are for the event lookup table, where $DC1$ is used for length of the table, and $DC2$ for the event functions. This distinguishement needs to be made because more than 1 argument cant be stored in the bytearray. When the opcode is printed in the *list_code*() function, this is taken into account and is replaced with *dc*.
The *operators* dictionary is used for converting the operators identified by the AST into their encoded form.
The *comp_ops* dictionary is a convenient way to convert comparative operators to strings, and is used for storing the limitations of the bytearray in the memory constructs.

2.3 Memory constructs

The *HoleCall* class is used to create memory constructs. These allow for three things: Firstly, to save information that cant fit in the bytearray due to its limitations.
Secondly, to store address holes when we don't know in advance where to direct the opcode, such as in a function call. We may know that a variable calls a function, but we don't know in advance where it is defined, so we store the missing information in one of these structures. When the compiler determines the positions of the function definition, the missing information is filled.
Thirdly, if we need access the firmware names of the Thymio variables, we store them here.

```
class HoleCall:

    def __init__(self):
        self._dict = {}#stores holes from calls/returns/etc...
        self.flag = 0#usefull conditional flag
    def store_hole(self,address,caller):#self descriptive method
        self._dict[address] = caller
    def eject_holes(self,caller):#one call can come from many holes
        hole_addresses = []
        for address,call in self._dict.items():
            if call == caller:
                hole_addresses.append(address)
        for address in hole_addresses:
            self._dict.pop(address)
        return hole_addresses
    def eject_call(self,address):#but each hole has only one call
        caller = self._dict[address]
        self._dict.pop(address)
        return caller

func_calls = HoleCall()
func_rets = HoleCall()
jump_cond = HoleCall()
event_calls = HoleCall()
thymio_var_calls = HoleCall()
thymio_nf_calls = HoleCall()
```

We need a total of 6 memory constructs. The purpose of each construct is as follows:

- *func_calls* is to store the missing addresses of the function calls.
- *func_rets* stores the missing addresses when a function has a return value.
- *jump_cond* stores the extra information needed for the *JIN* opcode.
- *event_calls* stores the missing addresses of the event functions.
- *thymio_var_calls* stores the firmware names of the Thymio variables that don't require native function calls.
- *thymio_nf_calls* stores the firmware names of the Thymio variables that require native function calls.

There are 3 class methods that we use:
store_hole saves a number as a key and a string as its value. For example, if there is a missing address in a function call ($CALLSUB$), we store the address that needs the location of the function definition as the key, and the name of the requested function as the value.
eject_holes returns all the values for a given key. As in the previous example, if we wanted to know all the missing addresses that need the location of the function f, $eject\_holes(f)$ returns these addresses and deletes them from the internal dictionary.
eject_call does a simillar thing as *eject_holes*, but returns the key rather than the value. So if we know that at $adr1$ we need to know the function location, then the name of the function is returned with $eject\_holes(adr1)$, and then deletes it from the internal dictionary.

2.4 Bytearray functions

The following functions are primarily used to manipulate the information in the bytearray, along with determining the address offsets of any variables.

```
def int_to_bytes(val):
    return struct.pack("<i", val)

def bytes_to_int(bstr):
    return struct.unpack("<i", bstr)
```

These two functions offer a simple conversion between bytearray values and integer values, typically to read the opcode encoding in a given line in the intermediary code(*bytes_to_int*), and for putting information in it (*int_to_bytes*).

```
def emit_byte(x):
    code.append(x)

def emit_word(x):
    s = int_to_bytes(x)
    for x in s:
        code.append(x)
```

These functions are used to directly append the intermediate code with whatever we want. *emit_byte* is reserved for appending the encoded opcodes (*emit_byte*(*PUSH*) for example), and *emit_word* for any integer opcode arguments we want.

```
def emit_word_at(at,n):
    def emit_word_at(at, n):
        code[at:at+word_size] = int_to_bytes(n)

    if isinstance(at,int) and isinstance(n,int):
        emit_word_at(at, n)
    elif isinstance(at,list) and isinstance(n,list):
        for i,j in zip(at,n):
            emit_word_at(i, j)
    else:
        error(emit_word_at)



def hole():
    t = len(code)
    emit_word(0)
    return t
```

emit_word_at is used for placing integer values in arbitrary places in the intermediate code. Typically this is used to fill missing address holes. It's been defined recursively so that it can accept lists as an input, so that it can place multiple things at once
hole() is the function for generating a missing address. It does so by placing a 0 in the intermediary code, and by returning the intermediate code address, usually saved in a variable or in a memory construct.

```
def fetch_var_offset(name,mode):
    global globals_n
    global locals_n
    n = mode.get(name,None)
    if n == None:
        mode[name] = globals_n + locals_n
        n = globals_n + locals_n
        if mode == globals:
            globals_n += 1
        elif mode == locals:
            locals_n += 1
        else:
            error(fetch_var_offset)
    return n

def reset_locals():
    global locals
    locals = {}
    for i,j in enumerate(thymio_vars):
        locals[j] = i
```

$fetch\_var\_offset$ is responsible for getting the address offset of a variable, depending on the scope. For example, if **globals** = {'a':0, 'b':1}, then $fetch\_var\_offset('a', \textbf{globals}) = 0$. If however, we use this function on a new variable, like with $fetch\_var\_offset('c', \textbf{globals})$ then *globals_n* is incremented and 2 is returned.
reset_locals() is responsible for deleting the local space of variables when a function scope is exited, and pre-loading them with the Thymio variables than dont need native function calls. Unfortunately, it doesn't decrement the $locals\_n$ counter, meaning that the next local variable will occupy a uniquely new address in the Thymio memory. This is problematic since it wastes memory usage.

3 Thymio-specific dictionaries

These are primarily used to relate the name of a Thymio variables/event name/etc. with its' firmware equivalent.

```
event_dict = {"BUTTON.BACKWARD":"_ev.button.backward", \
"BUTTON.LEFT":"_ev.button.left", "BUTTON.CENTER":"_ev.button.center", \
"BUTTON.FORWARD":"_ev.button.forward", "BUTTON.RIGHT":"_ev.button.right", \
"BUTTONS":"_ev.buttons", "PROX":"_ev.prox", "PROX.COMM":"_ev.prox.comm", \
"TAP":"_ev.tap", "ACC":"_ev.acc", "MIC":"_ev.mic", \
"SOUND.FINISHED":"_ev.sound.finished", "TEMPERATURE":"_ev.temperature", \
"RC5":"_ev.rc5", "MOTOR":"_ev.motor", "TIMER0":"_ev.timer0", \
"TIMER1":"_ev.timer1", "restart":"0xffff"}#WITH THYMIO IDETNIFIERS


thymio_vars = {"thymio.motor.left.target":"motor.left.target",\
               "thymio.motor.right.target":"motor.right.target",\
               "thymio.button.center":"button.center",\
               "thymio.timer.period":"timer.period",\
               "thymio.prox.horizontal":"prox.horizontal"}

thymio_nf = {"thymio.leds.top":[3,"_nf.leds.top"],\
             "thymio.leds.bottom.left":[3,"_nf.leds.bottom.left"],\
             "thymio.leds.bottom.right":[3,"_nf.leds.bottom.right"],\
             "thymio.leds.circle":[8,"_nf.leds.circle"]}
```

event_dict is for the event functions, where the keys are the source code name, and the values are the firmware names.
thymio_vars does the same with the Thymio variables that don't call a native function.
thymio_nf is for the Thymio variables that do require a native function call. In addition there is the quantity of things we need to push onto the Thymios' stack before we can call the native function.

4 preprocessor(x,mode=0)

This functions is called in two situations with two different modes: When a node in the *code_gen*(x) function is a **list type** (mode 0), and when when latter node is the top of the AST (mode 1). Its important to note that mode 1 also runs the code in mode 0. We will analyse each each mode separately.

4.1 mode = 0

```
def preprocessor(x,mode = 0):
    main_body = x
#OPTIMIZE
    if(len(main_body) > 1):
        idx = 0
        while(idx < len(main_body)-1):
            idx += 1
            if(type(main_body[idx])!=ast.FunctionDef and\
               type(main_body[idx-1])==ast.FunctionDef):

                temp_node = main_body[idx]
                main_body[idx] = main_body[idx-1]
                main_body[idx-1] = temp_node
                idx = 0
```

This mode is responsible for reorganising the AST(x is the current **list** node) in such a way that the function definitions are pushed to the rightmost part of the tree, separating the functional code form the non-functional code. To visualise it in action, consider figure 1, with a parent node that has 3 children:

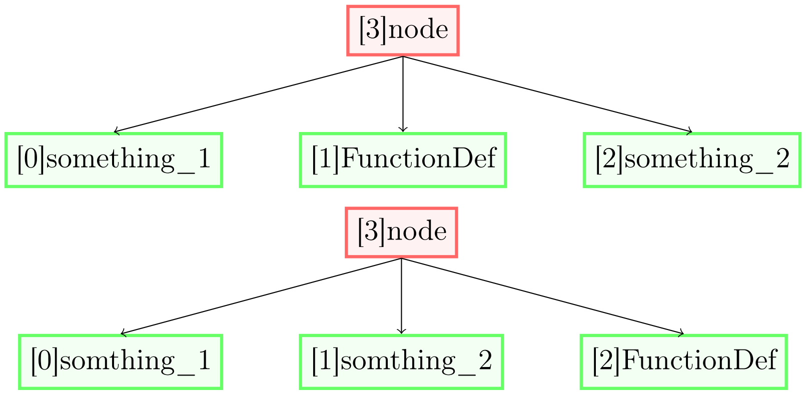

Figure 1: From top to bottom: Re-organising the AST to separate functional from non-functional code

The *#OPTIMIZE* comment is just to indicate that this is done with a naively implemented insertion sort algorithm.

4.2 mode = 1

```
if(mode):
    global globals_n
    emit_byte(DC1)
    tab_len_hole = hole()
    emit_byte(DC2)
    start_hole = hole()
    for i in main_body:
        if type(i)==ast.FunctionDef and i.decorator_list:
            emit_byte(DC2)
            event_hole = hole()
            event_name = i.decorator_list[0].args[0].attr
            event_calls.store_hole(event_hole, event_name)
    emit_word_at(start_hole,len(code)-start_hole)
    event_calls.store_hole(len(code)-start_hole, "restart")
    emit_word_at(tab_len_hole,len(code))#-tab_len_hole)

    #loading the thymio variables
    for i,j in enumerate(thymio_vars):
        globals[j] = i
        locals[j] = i
        globals_n += 1
```

This mode is responsible for constructing the event lookup table, which is the first thing put into the intermediate code. This mode is called when the current node is of **type ast.Module**, and it's at this height in the AST where the event function handlers are defined, specifically within the children of the **body** node, which is below the **ast.Module** node. First we generate the code responsible for signaling the length of the event table, and the line where the executable code begins. The latter two have yet to be determined until the table is completed, so two address holes are created that are filled at the end of the process(*tab_len_hole*,*start_hole*).
Then, we iteratively go through the **main_body** node looking functions that have a decorator defined alongside it. Since in this implementation decorators exist only to signal event functions(and are not used in the typical Python fashion), no further checks are necessary. For a given event function, we don't know where in the code it will be defined, so we create an address hole(*event_hole*) and store it in the *event_calls* construct along with the event function name. We do this ,so that later in the **ast.FunctionDef** code generator, these missing locations can be filled. This information is also needed in the *list_code*() function.
Once all the event functions have been identified, the length of the event table is known, and we know where the executable code will begin. Therefore, we can fill the *tab_len_hole* and *start_hole*. Note that we also need to store the name of the *restart* event independently from the event functions since it will always be on the Thymio regardless of the source code. The net effect of $mode = 1$ on the intermediate code can be seen in table 3:

Opcode address	intermediate code address	opcode operation	opcode arguments
0	*0*	DC1	*length*
1	*5*	DC2	*adr_ start*
3	*10*	DC2	*0*
...	...	...	...
i	*adr_ start*	...	...

→

	event_calls
self._ dict	*5:"restart"*
	10:"event_ name"
	...

← executable starts

Table 3: The output process on the intermediate code from executing *mode* = 1.

Finally, the last **For** loop is responsible for pre-loading the **globals/locals** lists with the Thymio variables that don't require a native function call.

5 code_gen(x)

In this part, the intermediate code is generated and stored inside the **code** byte array. When the AST is generated, all of its nodes are assigned a certain **type**, and we generate code accordingly. For example, a given node x might be of **type ast.Num**, in which case the **ast.Num** code generator is invoked.

```
def code_gen(x):
    global stop_flag
    global scope_flag
```

The function itself will be called recursively as we traverse deeper into the AST, therefore we only need one argument, x. We also need access to the **scope_flag** to signal if the variables are local(to be erased when we exit the scope) or not. We also need the **stop_flag** to tell the compiler when the non-functional code has finished being generated.

5.1 ast.Pass

The **Pass** command generates no intermediate code, as is intended in Python.

```
    if type(x) == ast.Pass:
        pass
```

5.2 ast.Module

```
    elif type(x) == ast.Module:
        preprocessor(x.body,mode = 1)
        code_gen(x.body)
```

This is the uppermost node in the AST, therefore we only care to traverse below it. Importantly, we invoke the **preprocessor** function in a special mode which serves two purposes:

1. Separate function definitions from the non-functional code(the typical use of the **preprocessor**).
2. Initialises the event lookup table(special mode).

Point 1. serves only to re-arrange the AST and so not intermediate code is generated. Point 2. does generate the latter.
After the preprocessor finishes, we traverse into the next layer of the tree; The **body** node, which is of type **list**. We can visualise this traversal in figure 2

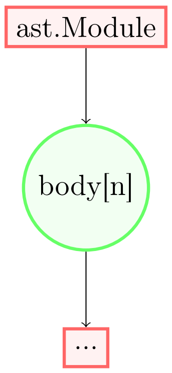

Figure 2: We begin at the ast.Module node, and point the copmiler to enter body[n]

Note that the body can have **n** sub-nodes, with indexes ranging from $[0, n-1]$

5.3 list

```
elif type(x) == list:
    preprocessor(x)
    for i in range(len(x)):
        code_gen(x[i])
```

Typically **body** nodes are of type **list**. The idea is to first separate functional from non-functional code, and then run through it index by index in a linear fashion. No intermediate code is generated, and this serves just to re-arrange and direct the code_gen function deeper into the AST.
The **preprocessor** is necessary because a **body** node is part of an **ast.FunctionDef** node. This means that a function definition could contain another function definition (nested functions). This is to avoid data erasures in the **locals** list. Suppose that **locals**= $\{'a' : 220, 'b' : 221\}$ and we have the following source code:

```
def f(...):
    a = ...
    b = ...
    def g(...):
        ...
    b = a
```

Without the **preprocessor**, the moment the compiler enters the function definition of **g(...)**, the **locals** list will not be reset. The variables a, b might be changed since there may be variables a, b local to **g(...)** that are different from those in **f(...)**. The command $b = a$ will cause the compiler to risk assigning meaningless values to these variables.
By using the **preprocessor** function, function definitions are the last thing to be generated in any scope of source code, avoiding data collisions in the local scope of the function it is currently in.
In the absence of any functions, or if the **preprocessor** finishes, we iterate through each index of the **list** to generate the corresponding opcode in the range $[0, len(x) - 1]$. We can visualise this in figure 3

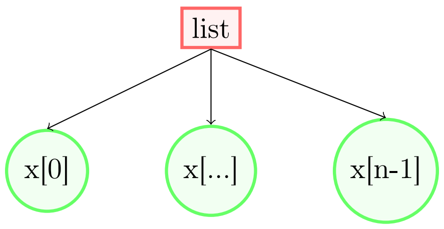

Figure 3: Iteratively entering through the n indices of x from 0 to $n - 1 (= len(x) - 1)$

5.4 ast.Assign

```
elif type(x) == ast.Assign:

    code_gen(x.value)
    if(type(x.value) == ast.Call):
        emit_byte(LOAD)
        loadadr = hole()
        f = x.value.func.id
        func_rets.store_hole(loadadr, f)
    code_gen(x.targets[0])  #assume 1 target
```

There are two types of assignments as far as the compiler is concerned: those which call functions, and those that do not. Let us consider the latter case: Arithmetic operations, list assignments, etc... will result in some values being pushed onto the Thymios stack. Since we want to assign the result to a variable, the output of the expression must first be calculated before we can assign it to the variable, therefore *x.value* is the first opcode to be generated, followed by *x.targets*, the latter responsible for assigning the result to a variable.
Note that the *x.targets* is a **list**, since Python allows for multiple variable assignment. In this implementation, we assume 1 target for assignment. Therefore, the following is acceptable:

```
a = 5*6+23+...
```

but the following will cause the compiler to crash:

```
a,b,c,... = 1,2,3,...
```

Now suppose the source code is calling a function with the form:

```
a = f(...)
```

As usual the *x.value* will generate the appropriate opcode for calling the function, however at the end we need to load the **return** value in order to assign it to our variable. This is where the **if(...)** statement is important. We can see the consequence of these operations in table 4:

Opcode address	intermediate code address	opcode operation	opcode arguments
...	...	...	...
i	*loadadr*	LOAD	*0*
i+1	*loadadr+5*	...	...

$\rightarrow$

	func_rets
self._dict	*laodadr:f*

Table 4: Address of **return** value not yet determined, *loadadr* is stored in *func_rets* structure.

Since the compiler does not know in advance where to load the **return** value from, it simply puts 0 as an address(temporarily) and stores the address of the LOAD instruction in the *func_rets* structure, along with the name of the function being called(f). The process of filling this temporary address with the true value is described in **ast.Return**.
There is a limitation to this setup, since the compiler will generate false code with the following command:

```
a = 3 + f(...)
```

The function call is embedded in a **ast.BinOp** code generator, so the **if(...)** instruction will never be run, so no *LOAD* instruction will be generated.
The programmer is therefore forced to work around this by writing:

```
a = f(...)
a = a + 3
```

5.5 ast.Attribute

```
    elif type(x) == ast.Attribute:
        if type(x.value) == ast.Attribute:
            x.value.attr = x.value.attr+"."+x.attr
        elif type(x.value) == ast.Name:
            x.value.id = x.value.id+"."+x.attr
        else:
            error("Unknown name type")
        if type(x.ctx) == ast.Load:
            x.value.ctx = ast.Load()
        elif type(x.ctx) == ast.Store:
            x.value.ctx = ast.Store()
        else:
            error("Uknown context")
        code_gen(x.value)
```

This generator is used exclusively for the Thymio constructs in the source code, such as **thymio.motor.left.target**,**thymio.leds.top**,etc...
The compiler would rather treat these variables as individual names, so this generator pushes attributes to the bottom of the tree until the lowermost level is the name of the Thymio variable with its corresponding context. Consider we have the source code:

```
thymio.leds.top = ...
```

Note how we wish to store something in this Thymio variable, this will invoke the **Store** context(ctx) in the AST.To visualise how the attributes are pushed down the tree, observe figure 4:

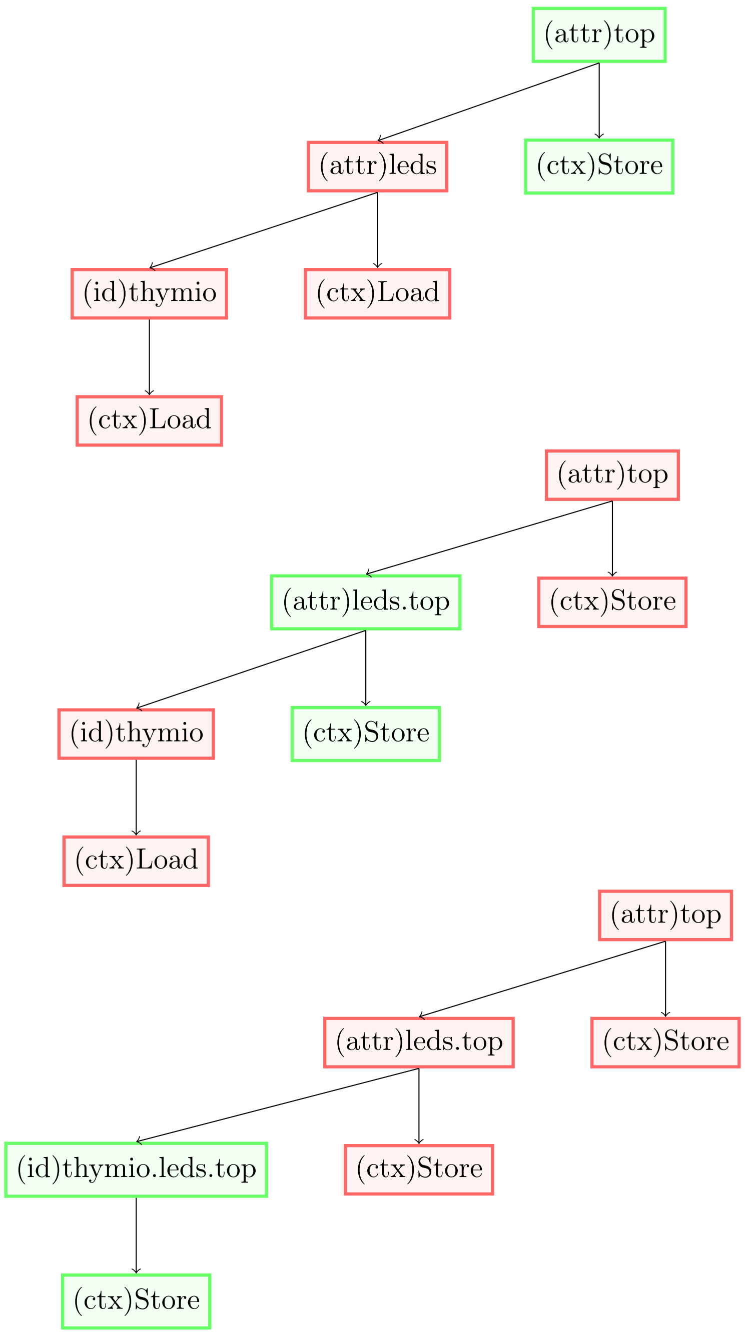

Figure 4: Top to bottom: consecutive steps of pushing the attributes down the AST.

Once the context and the attributes have been pushed to the bottom, the name of the variable is finally stored at the lowermost level. When we invoke the $code\_gen(x.value)$ command, we can continue with generating intermediate code for the previous hypothetical python source code.

5.6 ast.Num

```
    elif type(x) == ast.Num:
        emit_byte(PUSH)
        emit_word(x.n)
```

A very simple generator, usually invoked when we manipulate integers(more generally, non-variables), such as in the source code:

```
a = 1
b = 2
...
```

The **PUSH** command itself pushes some 16 bit value onto the stack of the Thymio, and is manipulated according to wherever this generator called in the AST. The *emit_word* function is called to place the 16 bit value in the intermediate code. Since it pushes a 16 bit value, we need two lines of opcode: one for the **PUSH** command, and one for the 16 bit value. Note that there also exists the **PUSH.S** command which is for 12 bit numbers, however this has not been implemented. This version stores the number and the command in 1 line of opcode.
By adding an extra condition in this code generator that tests the size of $x.n$, we can make this more efficient.
The result of it can be seen in 5:

Opcode address	intermediate code address	opcode operation	opcode arguments
...	...	...	...
i	*adr*	PUSH	$x.n$
i+2	*adr+5*	...	...

Table 5: Pushing a number onto the Thymio stack. The **PUSH** command requires 2 lines of opcode.

5.7 ast.If

5.7.1 CHANGES HERE

```
    elif type(x) == ast.If:      #ONLY FOR CONDITIONALS
######### NEW CODE ENCLOSED BELOW #######################
        if type(x.test) != ast.Compare:
            code_gen(x.test)
            emit_byte(PUSH)
            emit_word(0)
            emit_byte(GT)
        else:
            code_gen(x.test)          # expr
######### NEW CODE ENCLOSED ABOVE #######################
        cond_op = code[-1]
        del code[-1]
        emit_byte(JIN)              # if false, jump
        p1 = hole()                 # make room for jump dest
        jump_cond.store_hole(p1, cond_op)
        code_gen(x.body)            # if true statements
        if (len(x.orelse) != 0):
            emit_byte(JUMP)         # jump over else statements
            p2 = hole()
        emit_word_at(p1, len(code) - p1)
        if (len(x.orelse) != 0):
            code_gen(x.orelse)      # else statements
            emit_word_at(p2, len(code) - p2)
```

This generator is responsible for producing **If** statements, ~~although there are some restrictions. This originates from the code generated in *x.test*, where the opcode for a comparison is generated. This means that all **if** statements must have a comparator. So the following source codes will cause the compiler to crash:~~ both of the below variants are valid.

```
if(True):
    ...
if(1+1):
    ...

if(True != False):
    ...
if(1+1 > 0):
    ...
```

~~Note that the latter code is still valid under normal Python syntax, it's just a bit more demanding of the programmer.~~
Once the *x.test* code is generated, the limitation of the **code** byte array is apparent: Since we must use the $jump.if.not(JIN)$ opcode, we would need to store 3 things on the same line of intermediary code (The JIN command, the comparator, and the jump address). Since this isn't possible, we store the comparator(*cond_op*) in a separate construct, *jump_cond*.
Before we can tell the **JIN** command where to go if the comparison is false, we save the missing address in $p1$ and then generate the **body** of the **if** statement via *x.body*. Similarly,

we test the existence of an **else** statement and create the corresponding missing address $p2$ for the **JUMP** command.
It's at this point that we know where to go if the **JIN** command come out as false, so we use the *emit_word_at* function to fill the missing address at $p1$.
Note that the relative address is stored in the intermediary code($len(code) - p1$) rather than the absolute address($len(code)$). This is usefull since it lets us express negative jumps in code, as is done in the **ast.While** code generator.
Finally, we generate the **body** of the **else** statement(if it exists) and fill the missing address $p2$ with the appropriate location. We can visualise the overall effect on the intermediary code in table 6:

Opcode address	intermediate code address	opcode operation	opcode arguments			
...	...	...	...			jump_cond
i	*adr1*	JIN	*adr2+5-adr1*	$\rightarrow$	*self._dict*	*adr1:cond_op*
...	...	...	...			
j	*adr2*	JUMP	*adr3-adr2*			
j+1	*adr2+5*	...	...			
k	*adr3*	...	...	$\leftarrow$	outside else statement	

Table 6: Resultant effect of the **If** code generator on the intermediary code.

5.8 ast.Compare

```
elif type(x) == ast.Compare:
    code_gen(x.left)
    code_gen(x.comparators[0])
    emit_byte(operators[type(x.ops[0])])
```

This produces the intermediate code necessary for the comparative operations $>, <, ==, !=, <=, >=$. Due to how the Thymio processes these statements, the things to be compared must first be put onto its stack. Therefore, $x.left$ generates the code for the left part of the comparator, and $x.comparators[0]$ generates the right part. Note that the reason the latter is a **list** is due to some more exotic expressions that Python also allows for. For simplicity, comparators must be used in a "reduced" fashion, so the following types of statements are allowed:

```
a+2*b > c - 2
```

but the following will cause a crash:

```
[1,2] > [5,6,7]
```

Finally, the comparator itself is generated with the last line. The code **type(x.ops[0])** gives us the ASTs' intepretation of the comparator, which we then pass through the **operators** dictionary to determine what the intermediate code equivalent is.
Since we can have statements such the following in Python:

```
a = 5>2
```

we use *emit_byte* so that the resulting logical answer can be stored into whatever variable we want. To visualise the result of the above Python comparison(without the assignment), consider table 7:

Opcode address	intermediate code address	opcode operation	opcode arguments	
...	....	...	...	
i	*adr1*	PUSH	*5*	
i+2	*adr1+5*	PUSH	2	
i+4	*adr1+10*	GT	*N/A*	
i+5	*adr1+11*	...	...	← outside comparison

Table 7: Resultant intermediate code from the comparison $5 > 2$

5.9 ast.While

5.9.1 CHANGES HERE

```
    elif type(x) == ast.While:          #ONLY FOR CONDITIONALS
        p1 = len(code)
######### NEW CODE ENCLOSED BELOW #######################
        if type(x.test) != ast.Compare:
            code_gen(x.test)
            emit_byte(PUSH)
            emit_word(0)
            emit_byte(GT)
        else:
            code_gen(x.test)            # expr
######### NEW CODE ENCLOSED ABOVE #######################
        cond_op = code[-1]
        del code[-1]
        emit_byte(JIN)
        p2 = hole()
        jump_cond.store_hole(p2, cond_op)
        code_gen(x.body)
        emit_byte(JUMP)             # jump back to the top
        emit_word(p1 - len(code))
        emit_word_at(p2, len(code) - p2)
```

The **While** statement works with similar reasoning to the **If** statement, the key difference being the negative relative jump thanks to the variable $p1$, which is the address at the start of the comparator test. That way, we conduct a the **While** test after each iteration of the loop. ~~Similarly to the **If** statement, there must be a comparator for the code to work. So the following source code will crash:~~ Both of the below variants are valid:

```
While(True):
    ...
While(a):
    ...

While(True != False):
    ...
While(a > 0):
    ...
```

Another difference is the use of the *emit_word* function which allows for the negative jump. Since we already know $p1$, we can immediately place the relative address after the **JUMP** command(we don't have to specify a location earlier in the code).
We can visualise the effect on the intermediary code in table 8:

Opcode address	intermediate code address	opcode operation	opcode arguments	
l	*adr0*	(comparator test begin)	...	
i	*adr1*	JIN	*adr2+5-adr1*	→
...	...	...	...	
j	*adr2*	JUMP	*adr0-adr2*	
j+1	*adr2+5*	...	...	← outside while statement

	jump_cond
self._dict	*adr1:cond_op*

Table 8: Resultant effect of the **While** code generator on the intermediary code.

5.10 ast.Call

```
elif type(x) == ast.Call:        #NO ARGUMENTS
    emit_byte(CALLSUB)
    subcalladr = hole()
    f = x.func.id
    func_calls.store_hole(subcalladr, f)
```

This code is responsible for when a function is called. At time of writing, functions cannot be called with arguments. Since we do not know the address of the beginning of the function in advance, we store the missing address in the *func_calls* construct, along with its name. We can visualise this in table 9:

Opcode address	intermediate code address	opcode operation	opcode arguments
...	....	...	...
i	*subcalladr*	CALLSUB	*0*

→

	func_calls
self._dict	*subcalladr:f*

Table 9: Placing the **CALLSUB** command in the intermediate code, and storing the missing address in the *func_calls* construct.

5.11 ast.FunctionDef

```
elif type(x)== ast.FunctionDef:        #NO ARGUMENTS
    if(stop_flag == 0): # This is called only once,
        emit_byte(STOP) # after the source code(without the functions)
        stop_flag = 1   # has been generated.

    call_holes = func_calls.eject_holes(x.name)
    emit_word_at(call_holes,[len(code)-i for i in call_holes])

    decorator_flag = 0

    if x.decorator_list:
        decorator_flag = 1
        deco_id = x.decorator_list[0].args[0].attr
        event_holes = event_calls.eject_holes(deco_id)
        emit_word_at(event_holes,[len(code)-i for i in event_holes])
        for i in event_holes:
            event_calls.store_hole(len(code)-i, deco_id)

    func_rets.flag = x.name
    scope_flag = locals
    code_gen(x.body)
    if decorator_flag:
        emit_byte(STOP)
    else:
        emit_byte(RET)
    reset_locals()
```

This code generates the intermediate code for functions. At time of writing, functions cannot be called with arguments. The first thing which is done is to set the **stop_flag**

to 1, signaling to the compiler that it does not have to insert the **STOP** command at the end of the code. We then place the the **STOP** command only once, just before generating any functional code. In the absence of functions in the source code, the **stop_flag** will remain 0, so the compiler will place **STOP** in the intermediate code only at the final step of compilation.
After this preparatory step, all the missing addresses that call this function must be filled. Using the *eject_holes* method of the *func_calls* construct, all of the places in the intermediate code that call this function have their temporary addresses re-written with the true address using the *emit_word_at* function. For example, the **ast.Call** generator places a 0 as the temporary address, and is then replaced with $len(code) - i$ (relative address).
There is a limitation created by this, as the following code will cause the compiler to crash:

```
def f(...):
    ...
def g(...):
    a = f(...)
    ...
```

Since the addresses are resolved in the function definition of **f(...)**, any calls to this function after this point will not be resolved.
The code in the **If(...)** statement does a very similar thing, just for the event handler functions. Recall that the very first intermediate code that is generated is the event lookup table by the **preprocessor**. So if this function is an event handler, the lookup table needs to have its temporary addresses re-written with this true (but relative) address. The **list_code()** printer function needs to know these addresses along with the event function names, so they are re-saved in the *event_calls* construct (since the *eject_holes* method removes them).
In case the function has a **return** value, the public field *flag* of the *func_rets* construct stores the name of this function.
Finally, the **scope_flag** is set to **locals**, indicating to the compiler we are entering a deeper scope. This is followed by generating the body of the function with *code_gen(x.body)*.
When this completes, either **STOP** is placed in the intermediate code to tell the Thymio that the event handler has terminated, or **RET** to indicate the end of a function.
Lastly, the *reset_locals*() function eliminates all local variables and we exit this scope. To visualise the change in intermediate code, consider table 10:

Opcode address	intermediate code address	opcode operation	opcode arguments
i-1	*f_adr-1*	STOP	*N/A*
i	*f_adr*	...	...
...	...	...	...
j	*f_end*	RET	*N/A*

$\rightarrow$

intermediate code address	opcode operation	opcode arguments
adr1	CALLSUB	*f_adr-adr1*
adr2	CALLSUB	*f_adr-adr2*
...	...	...

Table 10: Generating the **body** of the function(left) and replacing the temporary addresses in the intermediate code(right)

5.12 ast.Return

```
elif type(x) == ast.Return:
    code_gen(x.value)
    scope_flag = globals
    n = fetch_var_offset('0_return_dummy',scope_flag)

    ret_holes = func_rets.eject_holes(func_rets.flag)
    emit_word_at(ret_holes,[n for i in ret_holes])

    emit_byte(STORE)
    emit_word(n)
    emit_byte(RET)
    reset_locals()
```

This code generator handles the return values and addresses. The value we wish to return is first generated and we change the **scope_flag** to **globals**. At this point, the location where return addresses will be loaded from is created thanks to the variable n. By creating the dummy variable 0_*return_dummy* in the **globals list**, this address is created dynamically, rather than reserving a fixed address. Once created, all subsequent return values will be stored to this address.
Since a **return** command is found within a function, we replace the temporary addresses created by **ast.Assign** with n, according to which function is called it (stored in the $flag$ field). This step is redundant due to earlier versions of this code, since all return values will be loaded from the same address, regardless of the function.
Finally, we create the intermediate code to store the return value at address n, and similar to **ast.FunctionDef**, we exit the function and reset the **locals** list.
We can visualise this proccess in table 11:

Opcode address	intermediate code address	opcode operation	opcode arguments
...	...	...	...
i	*adr_ret*	STORE	*n*
i+1	*adr_ret+5*	RET	*N/A*

$\rightarrow$

intermediate code address	opcode operation	opcode arguments
...	CALLSUB	...
adr1	LOAD	*n*

Table 11: Generating the return(left) and replacing the temporary **LOAD** addresses in the intermediate code(right)

Unfortunately, this set up will cause the following source code to crash:

```
def f(...):
    ...
    return
    ...
    return None
    ...
    return [1,2,3]
```

The compiler is always expecting to return something, so an empty return will cause a crash. Similarly, since the return address is fixed once determined, lists cannot be returned since any neighbouring data will be gibberish.

5.13 ast.Expr

```
elif type(x) == ast.Expr:
    code_gen(x.value)
```

This generator is invoked since some nodes contain it when we go deeper into the AST. We are interested in just going deeper, so we call $code_gen(x.value)$

5.14 ast.List

```
elif type(x) == ast.List:
    for i,j in enumerate(reversed(x.elts)):
        code_gen(j)
        if i < len(x.elts)-1:
            n = fetch_var_offset(len(code), scope_flag)
            emit_byte(STORE)
            emit_word(n)
```

This code is necessary for creating *arrays* (not Python *lists*). They are generated backwards with the last element of the array being placed first in the intermediate code. So when the first element is placed last, in the case of an assignment, it can immediately be assigned to its variable. To visualise this, consider the following code:

```
a = [1,2]
```

It will generate the intermediate code seen in table 12:

Opcode address	intermediate code address	opcode operation	opcode arguments		
...	...	...	...		
i	*adr1*	PUSH	*2*		
i+2	*adr1+5*	STORE	*n*		
i+3	*adr1+10*	PUSH	1		
i+5	*adr1+15*	STORE	*n+1*	←	address of 'a' is n+1

Table 12: Constructing an *array*(not a *list*) in the intermediate code

5.15 ast.Name

5.15.1 CHANGES HERE

```
    elif type(x) == ast.Name:
        n = 0
        if type(x.ctx) == ast.Load:
            emit_byte(LOAD)
############### ENCLOSED CODE BELOW REDUNDANT ############
            if x.id in globals and x.id not in locals:
                n = fetch_var_offset(x.id,globals)
            else:
############### ENCLOSED CODE ABOVE REDUNDANT ###########
                n = fetch_var_offset(x.id,scope_flag) #ALIGN IF DELETE ABOVE
        elif type(x.ctx) == ast.Store:
            n = fetch_var_offset(x.id,scope_flag)
            emit_byte(STORE)
        else:
            error("Unknown name type")
        if x.id in thymio_vars:
            var_hole = hole()
            thymio_var_calls.store_hole(var_hole, thymio_vars[x.id])
        else:
            emit_word(n)
        if x.id in thymio_nf:
            global var_offset
            for i in range(thymio_nf[x.id][0]):
                emit_byte(PUSH)
                emit_word(n-i+var_offset)

            emit_byte(CALLNAT)
            nf_hole = hole()
            thymio_nf_calls.store_hole(nf_hole, thymio_nf[x.id][1])
```

The part of the code generator is responsible for dealing with affectations or changes made to variables. It it quite dense and verbose, mainly due to the Thymio variables that have to be treated differently from generic Python variables.
For a generic Python variable, we first determine its context (if we want to load it to the Thymio stack, or store something from the latter into it), and put the corresponding opcode command in the intermediary code. Then we determine its address with the *fetch_var_offset* function, and similarly place this value in the intermediary code with *emit_word*.
Suppose however we want to access an array from within a function:

```
myarr = [...]

def f(...):
    ...
    a = myarr[...]
    ...
```

The enclosed code is redundant due to code updates. Feel free to delete.~~This is allowed under Python syntax,~~

```
if x.id in globals and x.id not in locals:
    n = fetch_var_offset(x.id,globals)
else:
    n = fetch_var_offset(x.id,scope_flag)
```

~~Note that this approach is not robust and should be re-written differently.~~
If we want to manipulate a Thymio variable that does **not** require a native function, we add the second **If** statement. The reason we store it as a missing address is because in the *list_code* function, we need to know the name of the Thymio variable in order to invoke the firmware name. This too is not robust and should be re-written.
Finally, if a Thymio variable requires a native call, the final **If** statement deals with this. This part of the code is by far the least robust, since it also tells the compiler the intermediate code for executing only the Thymio LEDs. This is extremely restrictive and will not function with anything else that requires a native function call.

5.16 ast.Subscript

5.16.1 CHANGES HERE

```
    elif type(x) == ast.Subscript:

        if type(x.value) == ast.Attribute:
            if type(x.ctx) == ast.Load:
                x.value.ctx = ast.Load()
            elif type(x.ctx) == ast.Store:
                x.value.ctx = ast.Store()
            code_gen(x.value)
            thymio_var_calls._dict[len(code)-4] += '+'+str(x.slice.value.n)

        else:
##################### NEW CODE ENCLOSED BELOW ##############################
            if x.value.id in globals and x.value.id not in locals:
                n = fetch_var_offset(x.value.id,globals)
            else:
                n = fetch_var_offset(x.value.id,scope_flag)
##################### NEW CODE ENCLOSED ABOVE ##############################
            if type(x.slice) == ast.Index:
                n = n - x.slice.value.n

            if type(x.ctx) == ast.Load:
                emit_byte(LOAD)
            elif type(x.ctx) == ast.Store:
                emit_byte(STORE)
            else:
                error("Unknown name type")
            emit_word(n)
```

This generator deals with accessing the elements of arrays. Again, this is verbose due to the Thymio variables.
For a generic array, the code in the **else** statement is affecting the intermediate code. Since arrays are stored in reverse, the address of the element we wish to change will be that of

the first element minus index. Note however that only simple access to arrays is possible, so $a[2]$ can be accessed, but $a[2i + 1]$ is not supported. It does not support expressions as an index.Since arrays can be accesed in a local scope, the new snclosed code allows for that, rather than needing to use the **global** keyword.
If we want to affect a Thymio variable, first we have to deal with the attributes(from the naming in the source code). Next we have to perform a very ugly command due to the fact that the Thymio accesses its array elements in the forward direction(and not in reverse like in my arrays). The command:

```
thymio_var_calls._dict[len(code)-4] += '+'+str(x.slice.value.n)
```

accesses the firmware name(string) of the Thymio array, and redefines it as itself plus the index of the element we want to access. We do this because the *list_code*() function would like to print this value immediately. This works, but it also breaks encapsulation, since we are accessing the private member of the *thymio_var_calls* construct. Please consider re-writing this.

5.17 type(None)

```
    elif type(x) == type(None):
            pass
```

This generator is incomplete, and serves a double role: To implement the **None** command of Python, and to offer a solution to an empty return value, so that the following code can compile:

```
def f(...):
    ...
    return None
    ...
```

5.18 ast.Global

```
    elif type(x) == ast.Global:
        glob_id=x.names[0]
        n = fetch_var_offset(glob_id, globals)
        locals[glob_id] = n
```

This is responsible for allowing the use of global variables in a local scope. It simply looks for the variable we want in the **globals list**, and then appends the **locals list** with its name and address.

5.19 ast.NameConstant

```
    elif type(x) == ast.NameConstant: #ONLY FOR SETTING TRUE/FALSE VARIABLES
        emit_byte(PUSH)
        if x.value == True:
            emit_word(1)
        elif x.value == False:
            emit_word(0)
        else:
            error("Unknown␣logical␣state")
```

This generates **True**/**False** logical types. For simplicity, True is the same as the integer 1, and False is 0.

5.20 ast.UnaryOp

5.20.1 CHANGES HERE

```
    elif type(x) == ast.UnaryOp:  #WE ASSUME ONLY LOGICAL 'NOT' EXISTS
################ NEW CODE BELOW ###########################
        if type(x.op) == ast.Not:
            code_gen(x.operand)
            emit_byte(PUSH)
            emit_word(0)
            emit_byte(EQ)
        if type(x.op) == ast.USub:
            code_gen(x.operand)
            emit_byte(PUSH)
            emit_word(-1)
            emit_byte(MULT)
############## NEW CODE ABOVE ############################
```

There are many type of unary operations, however only the **not** operator has been implemented.The Python AST considers negative numbers as Unary Operations on positive numbers, this needed to be accounted for in the enclosed code.

5.21 ast.BoolOp

```
elif type(x) == ast.BoolOp: #ONLY 'AND', 'OR' LOGICAL STATEMENTS
    code_gen(x.values[0])
    for i in range(len(x.values)-1):
        emit_byte(PUSH)
        emit_word(0)
        emit_byte(JIN)
        h1 = hole()
        if type(x.op) == ast.And:
            jump_cond.store_hole(h1,EQ)
        elif type(x.op) == ast.Or:
            jump_cond.store_hole(h1,NE)
        else:
            error("Unknown binary operation")
        code_gen(x.values[i])
        emit_byte(JUMP)
        h2 = hole()
        emit_word_at(h1, len(code) - h1)
        code_gen(x.values[i+1])
        emit_word_at(h2, len(code) - h2)
```

This generates sequence of of **and**/**or** logical statements of arbitrary length, and in whatever permutation. Specifically, it implements the "short-circuit" form of these statements, meaning that if we have the following statements, they will immediately be assigned the result in the comments:

```
a = True and b #a = b
c = False and b #c = False

d = True or b #d = True
e = False or b #e = b
```

5.22 ast.BinOp

```
elif type(x) == ast.BinOp:
    code_gen(x.left)
    code_gen(x.right)
    emit_byte(operators[type(x.op)])
```

This is responsible for generating arithmetic operations. Since the Thymio needs the operands in a certain order on its stack, the operand itself is the last thing to be generated in the intermediary code.

6 list_code()

This portion is primarily focused on properly interpreting and printing the intermediate code into real opcode that will run on the Thymio. It does so thanks to some of the memory constructs that exist to store more information than the **bytearray** can handle in one line of intermediate code.
Most statements in this function are self descriptive in how they function, but the important components will by analyzed in deeper detail.

6.1 Preparation and address mapping

```
opcode = []
def list_code():
    global opcode
    global var_offset
    membloat = [LOAD,STORE,PUSH,CALLSUB,JUMP, JIN ,DC1,DC2,CALLNAT]
    tpcbloat = [JIN ,DC2, PUSH]
    memdict = {}
    pc = 0
    tpc = 0
    while pc < len(code):
        memdict[pc] = tpc
        op = code[pc]
        pc += 1
        tpc +=1
        if(op in membloat):
            pc += word_size
        if op in tpcbloat:
            tpc +=1

    #print(memdict)
    print("Datasize:␣%d" % (len(globals)))

    pc = 0
    tpc = 0
    while pc < len(code):
        #print("%4d " % (pc), end='')
        print("%4d␣" % (tpc), end='')
        op = code[pc]
        pc += 1
        tpc += 1
        ...
```

This part of the code seeks to set up the necessary structures for opcode printing. In particular, we create the list *opcode* = [] which will serve to store the opcode instructions as strings, should be wish to export them. We also need the **global** variable *var_offset* so that we can add it to the variable addresses. As mentioned in the first section, the *fetch_var_offset* function begins counting variable addresses at 0, which is what the **LOAD** and **STORE** instructions refer to in the intermediary code. This cannot work on

the Thymio, so with var_offset we can align these addresses with something compatible. Next, we need to map the opcode line addresses of the intermediary with that used in the Thymio. This is used mostly for debugging and checking memory alignment issues, but is necessary to take advantage of the memory constructs to work.
membloat refers to the opcode commands that increase the line counter addresses of the intermediate code by 5. This is a consequence of the **bytearray** structure where it is stored. Similarly, *tpcbloat*(true program counter bloat) increases the line counted addresses of the Thymio code by 2. *memdict* serves to store the mapping between these line counter addresses.
pc refers to the counter for the intermediate code, and *tpc* for the Thymio true opcode counter The first **While** loop runs through each line of the intermediate code and tests if the current instruction is in either in *membloat* or *tpcbloat* and increments the counters accordingly. Finally, these equivalent numbers are stored in the *memdict* dictionary.
All the printing commands seen here serve just for debugging. The second **While** loop we iterate through the intermediate code to actually print and store the opcode for the Thymio. *pc* is reset to remain aligned with the intermediate code, and *tpc* is reset so that we can print the Thymio line numbers alongside the opcode instructions. This is just for debugging. The *opcode* array does not store any of this information.
Within the second **While** loop, *op* and *pc* are important for memory alignment. *pc* aligns the line number of the intermediate code, stored in *code*. *op* is the opcode command that exists at that line. Depending on its value, we decide what to print.

6.2 LOAD

```
if op == LOAD:
    if pc in thymio_var_calls._dict:            #PLEASE REFACTOR _DICT
        x = thymio_var_calls.eject_call(pc)
    else:
        x = bytes_to_int(code[pc:pc+word_size])[0] + var_offset
    print("load "+str(x));
    opcode.append("load " + str(x))
    pc += word_size
```

This code is for printing **LOAD** commands, but it distinguishes between Thymio variables. Since we need to load an internal Thymio address that can change between firmwares, we need the firmware name of that address.
When we are not in this situation, thing we want to load is found between $pc : pc + word\_size$ in the intermediate code. Using *bytes_to_int*, we can convert this back into an integer from a hexadecimal. Finally we increment *pc* by $word\_size (= 4)$ since opcode arguments take up this much space in the intermediate code. Therefore, when we next increment *pc* at the the start of the **While** loop, we are certain that its pointing to the next opcode instruction in the intermediate code, and not some gibberish value.
This code breaks encapsulation of the *thymio_var_calls* construct, consider re-writting this.

6.3 STORE

```
elif op == STORE:
    if pc in thymio_var_calls._dict:          #PLEASE REFACTOR _DICT
        x = thymio_var_calls.eject_call(pc)
    else:
        x = bytes_to_int(code[pc:pc+word_size])[0] + var_offset
    print("store␣"+str(x));
    opcode.append("store␣" + str(x))
    pc += word_size
```

The same reasoning as the **LOAD** printing applies here, jsut that now we are interested in storing.

6.4 PUSH,CALLSUB,CALLNAT

```
elif op == PUSH:
    x = bytes_to_int(code[pc:pc+word_size])[0]
    print("push␣%d" % (x));
    opcode.append("push␣" + str(x))
    pc += word_size
    tpc += 1
elif op == CALLSUB:
    x = bytes_to_int(code[pc:pc+word_size])[0]
    print("callsub␣%d" %(memdict[pc+x]))
    opcode.append("callsub␣" + str(memdict[pc+x]))
    pc += word_size

elif op == CALLNAT:
    x = thymio_nf_calls.eject_call(pc)
    print("callnat␣"+str(x))
    opcode.append("callnat␣" + str(x))
    pc += word_size
```

This code continues with the same reasoning as far as getting the opcode argument is concerned, its just with **CALLNAT** that we need the firmware name of the native function. Similarly, since they have an argument, we increment *pc* with *word_size*.
In **CALLSUB** we see the utility of the *memdict* mapping. Recall that when the address holes are filled, the relative addresses are stored in the opcode argument. So, x is the relative address. Therefore, $pc + x$ is the true location of the function in the intermediate code, and $memdict[pc+x]$ gives the Thymio equivalent location of the real opcode.

6.5 ADD to NOT, STOP, RET

```
elif op == ADD:
    print("add")
    opcode.append("add")
elif op == SUB:
    print("sub")
    opcode.append("sub")
elif op == MULT:
```

```
            print("mult")
            opcode.append("mult")
        elif op == DIV:
            print("div")
            opcode.append("div")
        elif op == MOD:
            print("mod")
            opcode.append("mod")
        elif op == LT:
            print("lt")
            opcode.append("lt")
        elif op == GT:
            print("gt")
            opcode.append("gt")
        elif op == LE:
            print("le")
            opcode.append("le")
        elif op == GE:
            print("ge")
            opcode.append("ge")
        elif op == EQ:
            print("eq")
            opcode.append("eq")
        elif op == NE:
            print("ne")
            opcode.append("ne")
        elif op == AND:
            print("and")
            opcode.append("and")
        elif op == OR:
            print("or")
            opcode.append("or")
        elif op == NEG:
            print("neg")
            opcode.append("neg")
        elif op == NOT:
            print("not")
            opcode.append("not")
            ...
        elif op == STOP:
            print("stop")
            opcode.append("stop")
        elif op == RET:
            print("ret")
            opcode.append("ret")
```

These are self descriptive, these commands are not followed by an opcode argument, so we do not need to increment *pc*.

6.6 JUMP,JIN

```
    elif op == JUMP:
        x = bytes_to_int(code[pc:pc+word_size])[0]
        print("jump %d" % (memdict[pc+x]));
        opcode.append("jump "+str(memdict[pc+x]))
        pc += word_size
    elif op == JIN:
        x = bytes_to_int(code[pc:pc+word_size])[0]
        cond_op = comp_ops[jump_cond.eject_call(pc)]
        print("jump if not "+cond_op+" %d" %(memdict[pc+x]));
        opcode.append("jump.if.not "+cond_op+' '+str(memdict[pc+x]))
        pc += word_size
        tpc += 1
```

Again here we see the use of *memdict* to deal with the relative jumps. The $jump.if.not(JIN)$ opcode has the comparator we wish to use stored in the *jump_cond* structure, and x is just the relative jump. the comparator is passed through the *comp_ops* dictionary so that we can get a string version of the comparator.

6.7 DC1,DC2

```
    elif op == DC1:
        x = bytes_to_int(code[pc:pc+word_size])[0]
        x = int((x/5-1)*2+1)
        print("dc %d"%(x))
        opcode.append("dc "+str(x))
        pc += word_size
    elif op == DC2:
        x = bytes_to_int(code[pc:pc+word_size])[0]
        event_id = event_dict[event_calls.eject_call(x)]
        print("dc "+event_id+", %d"%(memdict[pc+x]))
        opcode.append("dc "+event_id+", %d"%(memdict[pc+x]))
        pc += word_size
        tpc += 1
```

Finally, this code is for dealing with the event lookup table. for **DC1**, the command $x = int((x/5 - 1) * 2 + 1)$ made sense at one point, but it serves to convert the length of the table into the result of the *memdict* dictionary. This is redundant, since we could just use the dictionary. But it works, so it was never changed.
DC2 does a similar thing as **JIN** in that it uses its corresponding memory construct to eject the firmware names of the event functions(*event_id*).

6.8 opcode list and other

```
    # for bc in opcode:
    #       print(bc)
    ...
input_file = open("arithmetic_test.py", "r")
tree_arithmetic = ast.parse(input_file.read())
code_gen(tree_arithmetic)
code_finish()
list_code()
```

Finally, at the bottom of the *list_code* function, we can uncomment this block of code to print the opcode that the Thymio will actually use so it can run.
The code following *input_file* is responsible for reading the source code (stored in *arithmetic_test.py*), generating the AST(*ast.parse*), and generating the intermediate code and printing it.