SE 3XB3 Lab 3 Group 30 Report

Wenyu Yin (yinw7), 400296711
Xiang Zhang (zhanx326), 400306856

F_1: Global Variables And First Visits

simple.py in Pep/9

```
                BR      program
x:              .block  2
program:        LDWA    2, i
                ADDA    3, i
                STWA    x, d
                DECO    x, d
                LDBA    '\n', i
                STBA    charOut, d
                .END
```

add_sub.py in Pep/9

```
                BR      program
value:          .block  2
_UNIV:          .EQUATE 42
variable:       .block  2
result:         .block  2
program:        DECI    value, d
                LDWA    value, d
                ADDA    UNIV, i
                STWA    result, d
                LDWA    3, i
                STWA    variable, d
                LDWA    result, d
                SUBA    variable, d
                SUBA    1, i
                STWA    result, d
                DECO    result, d
                .END
```

Global Variable

A global variable is a variable whose state is shared among all methods, and may be modified or used by those methods. For example, in *add_sub.py*, the variable result keeps being modified throughout the program, and it is added or subtracted multiple times; therefore, it would be appropriate to call it a global variable.

If one of the variables is picked from an RBS program, it should be decided as a global one if it appears in most of the functions, and all methods share only one copy of it (i.e., static).

NOP1 Instructions[1]

NOP stands for no operation. The translator inserts a NOP instruction at the beginning of each branch. We suppose the primary reason for this is to cause a small amount of time delay so that all memory allocation done above has finished their work, which is required for correct computation of the main process. Without delay, the main process may execute prior to the initialization (memory allocation) and fails to execute. It can also serve as a placeholder that could come into use in the later development process since

[1]:
https://en.wikipedia.org/wiki/NOP_(code)#:~:text=A%20NOP%20is%20most%20commonly,in%20program%20development%20(or%20to

we may want to insert some code before the current program (e.g., pre-computation before "*value = int(input())*").

Specifically, in this lab, NOP1 is used as a placeholder when the program transits into a new state, but we are unsure about the first instruction in that state. If the first instruction is determined (e.g., Pep/9 translation of an if-statement always starts with a load instruction), we can simply call the *__access_memeory* method and specify the label parameter. On the other hand, NOP1 helps us in cases where the first instruction can be anything, making it possible for the algorithm to traverse the ast and recursively generate appropriate outputs.

Global Variable Visitor[2]

- The global variable visitor is responsible for extracting information about the global variables defined in the RBS, specifically the name of global variables. It has a *visit_Assign*() function that stores the variable name specified by targets in the AST to the class's instance variable "*results*".
- Rationale
    - With this visitor, we are able to initialize all global variables that will be used in the main process. Later on, either in the top-level program or functional calls, we can refer to these global variables (store/load) without any program, thanks to the initialization of this visitor. In other words, this visitor lays the foundation for subsequent programs to use and modify global variables.
- Limitation
    - Cannot extract local variables (even though this class is not responsible for local variables, we will need some class responsible for translating local variables later on.)

Top-Level Program Visitor

- Top-level program visitor extracts and stores all top-level instructions information (label and instruction) in a list based on the AST. This is achieved by calling different functions that are created to handle each corresponding operation in the AST and translating it into the correct format.
- Rationale
    - This Visitor allows us to get and store all generated instructions and reformat them without altering the existing content in AST (node) class, enhancing the open/closed principle. And because of creating a new class, it groups the same behaviour methods acted as single responsibility. At the same time, it is easier to accumulate all instructions.
- Limitation
    - Does not support the interpretation of function calls.
    - Does not support the translation of conditional statements.

Static Memory Allocation Generator

- The static memory allocation generator is responsible for storing information about global variables and generating pep/9 assembly code.
- Rationale

[2]: https://refactoring.guru/design-patterns/visitor

        - This generator enhances the signal responsibility principle by separating the feature of generating (printing) assembly codes from visiting AST nodes, ensuring each class takes care of its individual responsibility.
    - Limitation
        - It only supports variables that have a size of 2 bytes.
        - It does not allow the definition of constant variables (i.e., *.EQUATE* instruction).

Entry Point Generator

- The entry point generator stores instructions that the program visitor translated from RBS program to the pep/9 assembly syntax, and it is also responsible for printing them out.
- Rationale
    - Similar to the static memory allocation generator, this generator does nothing but output the assembly code for the top-level program. Therefore, it also enhanced the single responsibility principle by isolating printing from traversing the AST.
- Limitation
    - This generator only prints out instructions that were generated by TopLevelProgram.py; hence, its limitations follow from the Top-Level Program Visitor.

F_2: Allocation, Constants, And Symbols

Improvement Analysis

- Memory allocation
    - AST represents the variable with a known integer value using an attribute 'value' to store an ast.Constant class value. Otherwise, the variable is using global memory allocation. Therefore, when the value attribute in *Assign()* clause is an instance of ast.Constant, the variable will be stored under the key '*val*' in the dictionary as a tuple of its variable name and the value (extracted by *node.value.value*).
    - In the *StaticMemoryAllocation* generator, the allocation will use *.WORD n* and the value n can be found in the tuple.
- Constants
    - In an RBS program, constants are represented as a word that consists of capital letters and starts with a '_.' Therefore, each time we encounter an assign statement whose target has an id with the aforementioned characteristics, we extract its value using the value attributes in the node's value clause (i.e., *node.value.value*).
        - Note this case is handled before normal variable assignments because the value in its value clause is also an instance of ast.Constant class.
    - If a variable is stored under the '*const*' key in the result dictionary, it will be recognized by the *StaticMemoryAllocation* generator, and its name will be attached to its value using the *.EQUATE* instruction.
    - We modify the *visit_Assign()* function in *TopLevelProgram* to raise ValueError when encountering a situation in which the code reassigns value to a constant variable. Additionally, for the *_access_memory()* function, we perform an additional check on the node's name. If so, the variable is accessed in i-mode rather than d-mode.

- Symbol table
    - We created a *SymbolTable* class that is responsible for storing all variable names and their translated version. The *convert* method within the class assigns an appropriate variable name to each variable in the RBS problem such that those new names are guaranteed to follow the naming rules in pep/9. Specifically, for each variable, we check if the length of its name is greater than 8 characters. If so, the first 8 characters of the variable's name are taken, and the last characters are modified until the new name is unique in the names dictionary. Otherwise, the original name of the variable is kept.
    - The *convert* method is called in the global variable visitor such that all variables have a translated name when they are referred to in the top-level program. Additionally, the *get_name()* method is called within *_access_memory()*, *visit_assign()*, and *visit_Name()*, such that the variable names are also translated properly in the top-level program.
    - The *SymbolTable* class is also responsible for recording the names of constants and normal variables since their initial values have already been declared in the memory allocation. The names of these special variables are stored in a set, which comes with multiple designated methods that help to add or remove items from it. In *StaticMemoryAllocation.py*, we call the add method to add constants and normal variables to the set. The names in the set will be examined when an assignment statement is analyzed, and if the name is in the set, we will not bother performing the redundant immediate loading and storing operations. In the meantime, the skipped names will be removed from the set so that later assignments can proceed as normal.

Fibonacci or Factorial numbers

- Handling overflows
    - When we implement recursive functions in "real" programming languages, overflows could occur since computers typically use a stack to keep track of recursive calls. An unexpectedly significant amount of memory could be occupied by the stack. There are several ways that we can handle overflows in programming languages.
    1. Some programming languages, like Python, throw a StackOverflow error when an overflow occurs[3]. Hence, the intuitive way to handle this problem is to catch the error such that the program does not crash due to the overflow.
    2. We could also terminate the program before the overflow error occurs, such as using a try-except statement to print out the overflow problems.
    3. However, the above two ways result in program termination or unexpected results. Another way to handle overflows is to make functions tail-recursive[4]. That is, we prefer to accumulate results from previous recursive calls and pass them to later calls, then continuously make recursive calls and summarize their results at the end. This way, we can keep the stack at a constant size while obtaining the expected results.

F3: Conditionals

gcd.py in Pep/9

```
          BR          program
```

[3]: https://www.geeksforgeeks.org/python-handling-recursion-limit/#:~:text=Due%20to%20this%2C%20the%20recursion,that%20infinite%20recursions%20are%20avoided

[4]: https://lazamar.github.io/recursion-patterns/

```
a:              .BLOCK      2
b:              .BLOCK      2

program:        DECI        a, d
                DECI        b, d
while:         LDWA        a, d
                CPWA        b, d
                BREQ        end
while_body:     LDWA        a, d
                CPWA        b, d
                BRLE        else
                LDWA        a, d
                SUBA        b, d
                STWA        a, d
                BR          while
else:           LDWA        b, d
                SUBA        a, d
                STWA        b, d
                BR          while
end:            DECO        a, d
                .END
```

Explanation of Automated Translation

- A new method in *TopLevelProgram* is defined to extract information in if, elif, and else. A new variable called *_if_id* is defined along with a helper function *_identify_if()* to keep track of the name of each if-statement.
    - *_identify_if()* is similar to *_identify()*, while the previous one fetches and increments the id for if-statements, and the latter one performs the same operation on while-loop's ids.
- It is worth mentioning that the new method is named "*visit_If*" since the visit method in Python's ast library calls the overridden method in the implemented class that has the format "*visit_ClassName*" where *ClassName* is the type of visit's argument. Therefore, "*visit_If*" ensures itself is called when an if-statement in Python is detected.
- To convert conditional statements into assembly language, at first, we load the left-hand side parameter into the accumulator and compare it with the value in "comparators" in the node using "*__access_memory*." Subsequently, we append the branching instruction that will be executed when the if's condition is not met (using the *inverter dictionary* to get corresponding branching instructions). We then recursively dive deeper into the ast using the *visit()* method provided by the ast library, translating codes within the if-clause. After codes in the if-clause have been translated, we specify their termination by a branching instruction to the termination state.
- If there is an else clause, we create a new instruction "NOP1" with the label to specify the program that is branching above (if's condition is not met). Then, the body of the else-clause will be converted using the *visit()* method, and the termination of this clause will be the same as the if-clause by branching to the termination state.
- Lastly, the instruction for the termination state of the entire conditional statement is recorded

F4: Function Calls

Translation Complexity Ranking

- From the most complicated to the least one: factorial_rec.py > fib_rec.py > factorial.py > fibonacci.py > call_param.py ≈ call_return.py > call_void.py
- Rationale
    - Despite the fact that translating different types of function calls is similar in their nature (i.e., manipulating the stack), some translations can be more or less harder to accomplish. For example, to translate recursive function calls, we need to push new local variables to the stack repeatedly and allocate space in the stack to store the return values of each recursive call. As a result, we may obtain a stack whose size is significantly larger than that of other functions' stacks, making the program more complex than others. Functions that call another function in their bodies (act as both a caller and a callee) can also be more complicated to translate since we have to perform stack allocation inside the function call rather than in the main program. Intuitively, functions containing loops and conditional statements are harder to translate than other straightforward functions.
    - According to the above rationales, the function translating complexities are ordered based on if they consist of the following elements: recursive calls, other function calls, loops, and conditional statements.

call_void.py in Pep/9

```
          BR        program
_UNIV:    .EQUATE   42
;         local variable
value:    .EQUATE   2
result:   .EQUATE   0
;         function body
my_func:  SUBSP     4, i
          DECI      value, s
          LDWA      _UNIV, i
          ADDA      value, s
          STWA      result, s
          DECO      result, s
          ADDSP     4, i
          RET
program:  CALL      my_func
          .END
```

call_param.py in Pep/9

```
          BR        program
_UNIV:    .EQUATE   42
x:        .BLOCK    2
;         local variable
result:   .EQUATE   0
;         parameter
mX:       .EQUATE   4
;         function body
my_func:  SUBSP     2, i
          LDWA      _UNIV, i
          ADDA      mX, s
          STWA      result, s
          DECO      result, s
          ADDSP     2, i
          RET
program:  SUBSP     2, i
          DECI      x, d
          LDWA      x, d
          STWA      0, s
          CALL      my_func
          ADDSP     2, i
          .END
```

call_return.py in Pep/9

```
          BR        program
_UNIV:    .EQUATE   42
x:        .BLOCK    2
g_result: .BLOCK    2
;         local variable
l_result: .EQUATE   0
mX:       .EQUATE   4
retVal:   .EQUATE   6
;         function body
my_func:  SUBSP     2, i
          LDWA      mX, s
          ADDA      _UNIV, i
          STWA      l_result, s
          LDWA      l_result, s
          STWA      retVal, s
          ADDSP     2, i
          RET
program:  SUBSP     4, i
          DECI      x, d
          LDWA      x, d
          STWA      0, s
          CALL      my_func
          ADDSP     2, i
          LDWA      0, s
          STWA      g_result,d
          ADDSP     2, i
          DECO      g_result,d
          .END
```

Explanation of Automated Translation

1. The generalized program inherits the top-level program
    a. We created a new class called *GeneralizedProgram* that inherits from the top-level program to implement the additional feature of visiting function definitions. It serves as a more generalized version of *TopLevelPorgram*.
    b. Within the class, a new method *visit_FunctionDef()* is constructed to visit relevant nodes in the ast. The method is responsible for translating python function definitions to pep/9 codes. It first instantiates an instance of itself, which does nothing but visits the FunctionDef node. It also instantiates a local variable visitor and generator to produce Pep/9 codes for memory allocation of local variables, parameters, and return variables. After the instance of *GeneralizedProgram* that was initialized inside the method finishes examining all information within the functional definition, its recorded instructions will be appended to the original *GeneralizedProgram*. Stack operations are recorded at the entrance and exit of the function definitions.
2. Additional visitors and generators
    a. We created a new visitor named *FunctionVariables.py* that inherits from the ast.NodeVisitor and implements *visit_Assign*(), *visit_arg*(), and *visit_Return*() methods from the abstract class. The visitor is responsible for extracting all local variables that are used in a function, and its output will be passed to the new generator to be further processed. Since each function can have only one return value in RBS programs, we treat *self.results['return']* as a set that stores only one variable name, retVal.
    b. We created a new generator called *DynamicMemoryAllocation.py*. Similar to the global variable generator, it possesses a symbol table to which all variables it generates will be added. The *generate()* method primarily makes use of the *.EQUATE* instruction as the addresses of local variables on the stack are constant.
3. Updated symbol table
    a. In the previous version of the symbol table, the variable *__symbols* was represented as a class variable, and it will be shared across all instances of the symbol table. This decision results in a fatal error when we deal with functional variables, particularly when local variables accidentally have the same names as global variables. Therefore, we altered *__symbols* to be an instance variable and defined a symbol table variable for each instance of *TopLevelProgram*. In this way, when the designated *GeneralizedProgram* visitor is created to visit a function, it will have its own symbol table that is completely irrelevant to the symbol table for global variables (except that they share the same universal variables.) The new *func_convert()* method is a specialized variable name converter that ensures the local variables have a different name from all global variables.
4. Updated *visit_call()* method in the top-level program
    a. We altered the *'case _'* the *visit_call()* method to support the assignments with function calls. We handled the stack operation for accessing the memory of the parameters and returning values here. At first, we loaded the required parameters, stored them onto the stack, and pushed certain bytes to the stack. After finishing the visiting of the current function call, if the function has a return value, we popped back only the space for parameters and then popped 2 bytes for return values when we loaded it from the stack. Otherwise, we just popped all the used bytes back.

   b. When loading the parameters to the stack, we used a negative value for assigning stack space, which solves the problem of incorrect access in the case of recursive function calls. For example, when we are trying to access the parameter that is stored at the top of the stack, there will have another function being called and push the corresponding bytes onto the top of the stack, which results in the case that the desired parameter is not stored at the original place. Therefore, instead of using the parameter alias, using a negative value to store the parameter in the stack will handle this problem.

5. Updated *_access_memory()* method in the top-level program
   a. We added an additional Boolean variable to check whether the current execution of memory accessing is in the function calls. If it is in the function calls, then all the accesses should occur in the stack, where using *'s'* as the stack addressing mode. Otherwise, we use direct loading *'d'* and immediate loading *'i'* normally.
6. Call-by-value
   a. The "call-by-value" assumption avoids involving anonymous variables when we call a function. If the functions are not "call-by-value" (i.e., binary operations could exist in functional call arguments), the translator has to recognize the need for anonymous variables and allocate memory for them in advance. However, all of the memory allocations were done together at the entrance of the function. Hence, assuming functions are call-by-value ensures consistency across the
   Pep/9 assembly code and enhances their readability.

Stack Overflow Situation

- Due to the nature of the code we translated, each time we make a recursive call, we will grow the stack for a specific number of bytes (depending on the number of parameters, return values, and local variables). Stack overflow typically happens when we make too many recursive calls; in such a program, our Pep/9 code will repeatedly execute the same branching instructions without popping elements on the stack. Therefore, the stack will keep growing until a certain threshold, and the program will essentially crash after the stack's size exceeds the threshold.
- In the Pep/9 simulation software, no output will be produced if the program results in a stack overflow.

F5: Arrays

Explanation of Automated Translation of Global Arrays

1. Alter StaticMemoryAllocation and GlobalVariables
   a. Update the ways of allocating memory for arrays in the files that are specifically for global variables. During the assignment, retrieve the variables for arrays that have a naming format with an underscore at the end and its required memory space. Correspondingly, update the generators to print out the array's memory allocation in the correct format.
2. Remove hard-coded addressing mode specification
   a. In our previous version of the code, the addressing mode (e.g., i, d, etc.) was hard coded using multiple elif statements, resulting in code redundancy and lack of extendability. We updated the approach by adding a new method, *identify_addressing_mode()*, which takes

in the name of a variable as a parameter and outputs the appropriate addressing mode based on the variable name and the current status the visitor is in. For example, if we call *identify_addressing_mode()*, and pass in a variable called *_UNIV*, the method will recognize that the variable is a constant, and output '*i*' to specify we are accessing the variable as an immediate. As another demonstration, if a variable whose name ends with an underscore is passed to the method, the method will realize the variable represents an array, and it should be addressed in '*x*' mode.

3. Additional *slicing_var* attribute in the *TopLevelProgram*
    a. As variables that are used as array indices should be stored in the index register, the program needs to be able to identify those slicing variables and perform computations involving them in the index register (i.e., using LDWX, STWX, etc.). We decided to pick up these variables in the *GlobalVariable* visitor: we add the slicing variables in a set when an encounter an array element update during visit_Assign. The variables collected by GlobalVarible will then be passed to TopLevelProgram, supporting the program in identifying slicing variables.
4. Detect Subscript left-hand side in binary operations and assignment statements
    a. When visiting the assignment and any binary operation or comparison, the variable is an element in the array if the left-hand side element is an instance of Subscript. Therefore, we need to change the loading process of the variable: using 'LDWX' to load the slicing variable, finding the correct position of the element in the array with 'ALXS', and loading/storing the value from/to the index register through addressing mode 'x'.
    b. In order to modify the slicing variable correctly, _modify_index is needed to identify whether the current variable is the slicing variable by checking its existence in the *slicing_vars* set. If the variable is a slicing variable, then we will use 'X' to store/load it from/to the index register (LDWX, STWX, etc) instead of the accumulator.
5. Add *visit_Subscript()* method
    a. If the left-hand side of an operation is a Subscript object, the program immediately knows that we are accessing or updating one of the elements in an array, meaning it has to multiply the value in the index register by 2 to obtain the element in the array. However, since other operations are unknown (we could be using the array element to compare with another value, or we are updating a specific array element), the operations followed by 'ALXS' are translated by other methods.

Explanation of Automated Translation of Local Arrays

1. Alter DynamicAllocation and FunctionalVariables
    a. Previously, in the *FunctionalVariableExtraction* class, only the names of the local variables are stored, and we cannot do proper memory allocation for arrays without storing their lengths. Therefore, we updated the data structure such that the required size of a variable is stored along with its name as a tuple. It is also noteworthy that if an array is initialized with a fixed length, its required size is stored as 2 * length, while any variable-length arrays are initialized with a size of 200 bytes on the stack. This decision is made under the assumption that no array will have a size greater than 100, and further modifications on this are discussed in the unbounded data structure section.

    b. *DynamicMemoyAllocation* class is altered according to *FunctionVariableExtraction*, and it can now distinguish array variables from normal variables. The counter variable is also updated, such that it keeps track of the current stack that is taken and assigns new stack point values to the next variable.
2. Alter stack operations at the entrance and exit of a function called
    a. Since the list containing the local variable is altered to a list containing tuples with the variable name and its corresponding required memory size, we do not need to iterate through the local variable list to count the total required memory size for allocating stack memory. We will use a variable *num_local_vars* to store the total required size by summing up all the required memory sizes in the tuples.
3. Addressing mode identification - sx
    a. A variable should be accessed in sx-mode when we are performing an operation within a function (implies the requirement of a stack) and on an array variable. For example, say the following line exists in a Python function: word_[i] = data + key.
        i. The program will understand that it is visiting a function by the variable *_in_function*.
        ii. The program can recognize *word_* is the name of an array since it is already stored in the *array_names* set
        iii. The fact that *word_* is initialized in a function can be concluded from the translated name, which is stored in the symbol table.
    b. Concluding the three criteria above, we can construct an elif statement in the *_identify_addresing_mode*() method to help the visitor to do the correct translation.

Unbounded Data Structure

To implement an unbounded data structure in Pep/9, we could make use of a structure similar to a linked list. That is, we could initialize a fixed-size array first using the instruction **a**: .BLOCK 50 (assuming we are working with global arrays and use arrays with a fixed length of 25 elements). We store elements in the array until it is one element from being full. At the last element, we store the memory address that points to the following fixed-length array (b) that serves as a continuation of the current array (a). Then, that array can be constructed with **b**: .BLOCK 50, and its memory address can be stored at the last element in **a** with the indirect addressing mode in Pep/9. This chain of operation can repeat infinitely, and with that, we manage to implement an unbounded data structure in the Pep/9 virtual machine. Note the size of the array can also be shrunk by removing later arrays. For example, if all elements in **b** are removed, we shall also remove **b**'s address that is stored in the last element of **a**.

Self-Reflection

How much did you know about the subject before we started? (backward)

- WY: I did not have any experience translating high-level programming to assembly language through coding before starting this project. However, I did do some manual translation and learned some related knowledge about the assembly language and CPU architecture from the computer architecture course in the last term. The 3XB3 lecture also gave me a thoughtful overview and understanding of this project.
- XZ: I had some knowledge about how Python compilers and interpreters work due to a computer science course in high school where we did elementary programming. I have also worked with the ARM64 architecture family from a second-year computer architecture course. The 3XB3 lectures on Pep/9 grammar were also very helpful in starting this project.

What did/do you find frustrating about this assignment? (inward)

- WY: In this assignment, the most frustrating part is that we always need to modify the previous code and ways of implementation when we are trying to work on the later part. It is hard to predict and comprehensively consider every problem that may be encountered in the later development. Therefore our previous solution will have defects frequently. It takes a lot of time to look back at our original code and resolve these defects.
- XZ: The debugging process of this project is a bit different from what we did in the past, as we have to look at our code at a lower level and trace the value in the accumulator, stack, and index register. Also, the debugging process is a bit more complicated than before since the program does not output with an error, and we have to identify the problem by ourselves.

If you were the instructor, what comments would you make about this piece? (outward)

- WY: The current program can correctly translate all the given sample codes, and the implementation is trying to be consistent with the provided code structure to reduce redundant checking. It also contains several helper functions to enhance readability and single responsibility, though it still can have improvements for reducing hard-coded components.
- XZ: The program contains some features that are hard-coded, but since we are primarily focusing on the given sample codes, it does not affect the overall performance of the program. Additionally, the use of helper functions does a good job of decreasing redundancy code and ensuring the extendability of the program.

What would you change if you had a chance to do this assignment over again? (forward)

- WY: If I had a chance to do the assignment over again, instead of finishing each task directly after correctly translating the given code files, I would write more unit tests to test the current program before starting the new one. It would be helpful to simplify the debugging process and find more problems that we haven't considered before, which reduces later troubles about alteration.
- XZ: I would start the project after I understand the visitor design pattern better. The more I worked with the visitor design pattern in this project, the more I realized its power as it can modify its status and extract various information in different formats based on its status. If I can start with a more thorough understanding of the design pattern, I will be able to take advantage of it and make the code more efficient and clear.