# SFWRENG 3XB3

## *L3 – The Red-Bellied Snake Language*

McMaster University

ENGINEERING

Computing & Software

## GENERAL INFORMATION

| | | |
|---|---|---|
| • Author: | Dr. Sébastien Mosser | mossers@mcmaster.ca |
| • Start date: | 14/11/2022 | |
| • Delivery date: | 08/12/2022 | |
| • Weigh in final grade: | 10% | |

## LEARNING OBJECTIVES

At the end of this assignment, students should:

- **Know and understand:**
  - How a low-level assembly language works
  - How a high-level language can be translated to a lower level of abstraction
- **Be able to:**
  - Manually translate several programs into Pep/9 assembly language
  - Implement a translator from python-like language to assembly

## THE RED-BELLIED SNAKE (RBS) LANGUAGE

### A SUBSET OF PYTHON

We are going to work on a subset of Python, named RBS in this document. *Reb-bellied Snakes* (RBS) are a non-venomous kind of snake, endemic to North-America. They are quite small, but can be spotted when hiking in Coote's Paradise for example (maybe related to Lizadillos?).

Our RBS language holds the following properties:

- It is a subset of Python. Thus, RBS code can be run by any Python interpreter;
- It only works with integers, and only natively supports binary additions and subtractions;
- Conditional instructions are *if*, *elif* and *else;*
- The only available loop instruction is *while;*
- It supports global and locals variables. Anonymous computations (e.g., print(1+3)) are not legit (1+3 must be stored in a named variable before being printed).;
- Functions calls are only "by-value" (no side effects on objects outside of a function scope);
- Arrays are of fixed size, decided at initialization. Arrays can be global or local to functions;
- We support basic I/O with *input* and *print* functions.

Here is an example of a RBS program that computes n!.

1. The program defines two global variables (*n*, *value*), and two functions (*mult*, *fac*)
2. As the language only supports + and -, we need to write our own multiplication (*mult*);
3. The value of *n* is read from the keyboard (*input*), casted as an *int*
4. We store n! in *value*, and then *print* it out.

```
def mult(a,b):
    mult_r = 0
    while b > 0:
        mult_r = mult_r + a
        b = b - 1
    return mult_r

def fac(n):
    i = 1
    result = 1
    while i <= n:
        result =  mult(result, i)
        i = i + 1
    return result

n = int(input())
value = fac(n)
print(value)
```

We consider RBS as a subset of Python because it is way more limited in terms of expressiveness.

## ASTS, VISITORS AND GENERATORS

Your job as a software engineer here is dual:

- Understand how the Pep/9 virtual machine works so that you can translate high-level code into assembly machine and understand the impact of some deign choices in high-level languages;
- Put into motion your knowledge gained in 2AA4 about design pattern by leveraging the Visitor pattern, and 2CO3 with tree analysis.

A program can be represented as an Abstract Syntax Tree (AST), which is basically a tree representation of the program code. Python gives us access to its AST thanks to the AST module. Consider the following example:

```
x = 3 + 2
print(x)
```

```
Module(
  body=[
    Assign(
      targets = [Name(id='x', ctx=Store())],
      value = BinOp(
        left  = Constant(value=3),
        op    = Add(),
        right = Constant(value=2))),
    Expr(
      value = Call(
        func = Name(id='print', ctx=Load()),
        args = [Name(id='x', ctx=Load())], keywords=[]))],
  type_ignores=[])
```

On the left-hand side, you find the content of a legit Python file. On the right-hand side, you find the equivalent AST. A file is a ***Module***, which contain a list of instructions in its body. The first instruction is an ***Assign***ment, targeting x, and resulting into the execution of a Binary Operation (***BinOp***), which is an ***Add*** between two ***Constant***s. The second instruction is an ***Expr***ession which is a ***Call*** to a function ***Name***d print, using the variable ***Name***d x as first argument.

In software engineering, we often consider tree-structures as visitable elements (remember your Visitor design pattern). It is an elegant way to provide a tree-traversal mechanism once and for all, and to let engineers build their own visits on top of such mechanism. Python provides such a Visitor approach in its ast module, by providing the *ast.NodeVisitor* class.

Thus, to support the automatic translation of RBS programs into Pep/9, your job become simpler:

- Write a ***Visitor*** to extract the relevant information from the RBS program (e.g., identify the global variable to allocate memory)
- Push these pieces of information to a ***Generator*** to create the relevant Pep/9 code

## MAIN CHANGES WITH RESPECT TO LAB #1 & LAB #2

- You are more autonomous. This is the end of the term, you should be comfortable working with Python.
- The lab contains a manual part, as well as an automatic translation one.
- It is designed as an incremental and iterative lab. Your capacity to work in an incremental and iterative way will be evaluated.
- "***This lab is only 10%, I don't care***". That is correct, the lab worth only 10% of the final grade. But remember that the assembly part is a third of the final exam**. Not working on this lab is a good way to lose 30% of your score in the final, which assume diligent work**.

# PREAMBLE

## PROJECT PREPARATION

1. Find a teammate in your lab section.
    a. If your lab section contains an odd number of students, then one (and only one) group can be composed of three students. It'll require manual action on Avenue, please check the situation with your TA.
2. One member has to fork the "L3 – Assembly" project on the GitLab interface:
    a. URL: https://gitlab.cas.mcmaster.ca/sfwreng-3xb3-fall-22/l3-assembly
    b. Make sure your project is "private"
    c. Add your teammate as "maintainer" of the project
    d. Add your TA and the instructor as a "reporter"
3. **Declare your group on Avenue by the end of your first lab**
    a. Check that you have access to the submission interface

## WORK TO DO

The assignment is designed as a sequence of features implementation. Your job is to work on these tasks and report your results in a PDF file name `**report.pdf**` in the repository. You will be evaluated on the quality of your code as well as the quality of your report.

**Warning**: This is not a language design or compilation course. Thus, you are allowed to take "shortcuts" when translating, and leverage the assumptions made when defining the RBS language. It is fine if your translator only works for program similar to the given examples.

## F$_1$: GLOBAL VARIABLES AND FIRST VISITS

- Look at the contents of the directory _samples/1_global. It contains five RBS programs that only rely on global variables, assignments and while loops.
    - Manually translate the simple.py and add_sub.py program into Pep/9, using the rules seen during the lecture;
    - Explain in natural language what is the definition of a "global variable" in these programs. If you randomly pick one variable in an RBS program, under what condition can you decide it is a global one or a local one
- The repository contains some python code to kick-off the translation process
    - Run the following command:
        - `python translator.py –ast-only –f _samples/1_global/simple.py`
        - You should recognize the contents of the previous example.
        - When invoked with the –ast-only flag, the translator script will load the file (-f) given as input, and print its equivalent AST on the standard output.
    - Now run the following command:
        - `python translator.py –f _samples/1_global/simple.py`
        - The translator produces Pep/9 code for you!
    - Load this code into Pep/9, and execute it
    - The translator uses **NOP1** instructions. Any ideas why?
- Look at the code of translator.py
    - It relies on two visitors and two generators. Explain the role of each element.
    - Explain the limitations of the current translation code in terms of software engineering.

**Tasks:**

- Answer to all these question into your report.
- **Deliver a first version of your report on Avenue**.

## F$_2$: ALLOCATION, CONSTANTS AND SYMBOLS

- *Examples for this task are in the directory named _samples/2_mem_alloc*

For this task, you have to improve the translator to address three limitations in the translation process:

- **Memory allocation**: the current code always allocate global memory using a ***.BLOCK 2*** instruction. The canonical way of allocating memory when the interger value is known is to use ***.WORD n***, with n the value. This saves a ***LDWA*** and then ***STWA*** instruction at runtime.
- **Constants**: In RBS, we will consider constant variable named using uppercase letter only, and starting with an underscore (e.g., ***_UNIV***). These elements need to not be stored in memory to ensure they will never be modified at runtime, by using the ***.EQUATE n*** instruction.
- **Symbol table**: RBS variable names are not length limited. But Pep/9 symbols are limited to eight (8) characters. Thus, a direct translation from an RBS variable name into a Pep/9 symbol is too naïve. This can affect variables names, but also loop labels.

**Tasks:**

- For each improvement, explain in natural language how you extract (visit) the necessary information from the AST, and how you translate (generate) it into Pep/9.
- Implement these improvements in the code.

- Compute “large” Fibonnaci or factorial numbers. Explain how overflows should be handled in a “real” programming language.
- **Place and push a tag named ‘t2’ in your repository when your coding is done. Deliver a second version of your report on Avenue**.

## F3: CONDITIONALS

- *Examples for this task are in the directory named _samples/3_conditionals*

For this task, you will have to translate conditionals instructions (if, elif, else).

**Tasks:**

- Look at the programs in the example directory. Provide a manual translation for the gcd.py program.
- Explain in natural language how you will automate the translation of conditionals, and how it impacts the visit and generation.
- Code this new feature. You are free to modify the given code structure, e.g., add new generators, visitors, introduce new classes, …
- **Place and push a tag named ‘t3’ in your repository when your coding is done. Deliver a third version of your report on Avenue**.

## F4: FUNCTION CALLS

- *Examples for this task are in the directory named _samples/4_function_calls*

For this task, you will have to translate function calls.

**Tasks:**

- Look at the programs in the example directory. Rank them in terms of “translation complexity”, and explain the rationale behind the order relation you have chosen.
- Provide manual translations for call_param.py, call_return.py and call_void.py.
- Explain in natural language how you will automate the translation of function calls, and how it impacts the visit and generation. Emphasize how the “call-by-value” assumption helps here.
- Code this new feature. You are free to modify the given code structure, e.g., add new generators, visitors, introduce new classes, …
- We do not provide any “stack overflow” mechanism. What will happen if such a situation happen in a program?
- **Place and push a tag named ‘t4’ in your repository when your coding is done. Deliver a fourth version of your report on Avenue**.

## F5: ARRAYS

- *Examples for this task are in the directory named _samples/5_arrays*

For this task, you will have to translate arrays, being global ones or local ones. Arrays are always statically initialized (e.g., [0]*25 creates an array containing 25 cells, initialized with a 0), and stored in a variable that ends with an underscore. These two conventions will help you to translate more easily.

**Tasks:**

- Explain in natural language how you will automate the translation of global arrays, and how it impacts the visit and generation.
- Explain in natural language how you will automate the translation of local arrays, and how it impacts the visit and generation.
- Code this new feature. You are free to modify the given code structure, e.g., add new generators, visitors, introduce new classes, …
- The real python language handles lists/arrays in an non bounded way. Without doing it, can you envision how such unbounded data structure can be managed on a virtual machine like Pep/9?
- **Place and push a tag named 't5' in your repository when your coding is done. Deliver a fifth version of your report on Avenue**.

## BONUS TASK (UP TO 10 BONUS POINTS)

The bonus task will only be considered if the five feature are delivered and falls into the "Meet" or "Exceed" categories of the grading rubrics. Maximum grade is still 100. Pick one, no need to cover them all.

- **Support for for loops**. You could for example use an adst.NodeTransformer visitor to first preprocess all for loops into while loops, and then run your translator
- **Support for anonymous assignments**. It is tedious to have to store manually in a variable a value before being able to use it. One would like to write "print(3+2)" instead of having to first store the result of 3+2 into a named variable, and then print this variable.
- **Support for other data types**. RBS only supports integers. Add support for strings (**.ASCII**), chars and Booleans (**.BYTE**).
- **Support for switch statements**. RBS allows conditionals to be written with the if/elif/else structure. Introduce a way to support switch statements.
- **Peephole optimization**. Consider a sequence of Pep/9 instruction. Can't you "combine" some instructions so that you can save some CPU cycles? E.g., why loading a value if it is already present in the accumulator? How to remove some NOPs?

# DELIVERING YOUR ASSIGNMENT

## MSAF POLICY

**As this assignment is four weeks long, submitting an MSAF for the day of the deadline does not cancel it**. When submitting an MSAF on Mosaic:

- An automatic 3-days extension is granted to this assignment on Avenue, and your GitLab repository will be automatically cloned at the new deadline.
- The instructor implements these relief actions when receiving the automated MSAF email from Mosaic. If your case is different from "classical" situations, please communicate with the instructor using direct message on Teams.

## SELF-REFLECTION QUESTIONS

- How much did you know about the subject before we started? (backward)
- What did/do you find frustrating about this assignment? (inward)
- If you were the instructor, what comments would you make about this piece? (outward)
- What would you change if you had a chance to do this assignment over again? (forward)

## CODE AVAILABLE ON GITLAB

- Your GitLab repository will be automatically cloned at the deadline.

## SUBMIT YOUR REPORT ON AVENUE

- Avenue is configured to keep each version of your report. **We expect SIX (6) deliveries**: one incremented after each feature delivery, and a final one including the self-reflection questions.

**Late deliveries will not be considered. Too late is too late.**

**END OF ASSIGNMENT.**

*Instructor reserves the right to adapt the marking scheme (while preserving its spirit) during the marking process. Parts contribute equally to each category*

## Appendix – Grading Rubrics*

| Category | Part | Below [0-49] | Marginal [50-69] | Meets [70-84] | Exceed [85-100] |
|---|---|---|---|---|---|
| F1 (10%) | *Questions* | *No answers, or wrong answers* | *Shallow answers, no clear evidence of understanding* | *Clear description of the code and its rationale.* | *Justification of the code (limitations) above and beyond* |
| F2 (20%) | *Questions* | *No explanations, or wrong ones* | *Unclear description of the translation process* | *Understandable description of the translation process and the overflow situation* | *Depth of thoughts in the description is above and beyond* |
| | *Code & design* | *Does not execute, or produces pep/9 code that does no assemble* | *Pep/9 execution provides an approximate answer. Unreadable code.* | *Pep/9 execution is OK. Translation code is readable and explained, follows SOLID principles. Symbol table explicitly defined in the code.* | *Memory allocation is optimized when possible. Symbol table reified as an actionable object with translation-oriented methods.* |
| F3 (10%) | *Questions* | *No explanations, or wrong ones* | *Unclear description of the translation process* | *Understandable description of the impact of introducing conditionals* | *Depth of thoughts in the description is above and beyond* |
| | *Code & design* | *Does not execute, or produces pep/9 code that does no assemble* | *Pep/9 execution provides an approximate answer. Unreadable code.* | *Pep/9 execution is OK. Translation code is readable and explained, follows SOLID principles. Conditional labels can be mixed (if can contain sub-ifs)* | *Global effort to make the generated code readable. Translator code reusable and extensible.* |
| F4 (20%) | *Questions* | *No explanations, or wrong ones* | *Unclear description of the translation process* | *Clear description of how function calls are translated into Pep/9* | *Depth of thoughts in the description is above and beyond (e.g., call-by-balue)* |
| | *Code & design* | *Does not execute, or produces pep/9 code that does no assemble* | *Pep/9 execution provides an approximate answer. Unreadable code.* | *Pep/9 execution is OK. Translation code is readable and explained, follows SOLID principles.* | *Global effort to make the generated code readable. Translator code reusable and extensible.* |
| F5 (20%) | *Questions* | *No explanations, or wrong ones* | *Unclear description of the translation process* | *Clear description of how arrays are translated into Pep/9* | *Depth of thoughts in the description is above and beyond* |
| | *Code & design* | *Does not execute, or produces pep/9 code that does no assemble* | *Pep/9 execution provides an approximate answer. Unreadable code.* | *Pep/9 execution is OK. Translation code is readable and explained, follows SOLID principles.* | *Global effort to make the generated code readable. Translator code reusable and extensible.* |
| Misc. (20%) | *Iterative and incremental* | *One final delivery* | *No evidence that development was made incrementally* | *6 reports delivered, tags available on GitLab* | *Workload distributed properly over the duration of the assignment* |
| | *Self-reflection* | *No answers* | *Binary (or super short) answers* | *Answers describe the experience of the students adequately. Plan for action to support improvements* | *Depth in the self-reflection and plan for action* |