- Authors:
  - Cameron Dunn, dunnc11@mcmaster.ca
  - Rafeed Iqbal, iqbalr8@mcmaster.ca
- Group ID on Avenue: 23
- Gitlab URL: https://gitlab.cas.mcmaster.ca/dunnc11/l-3-assembly-group-23

F4

Ranking complexity of translation for the given examples:

The simplest example would be call_void, it is a simple function with no parameters. More complex than that is call_param, it takes in a parameter to be called. More complex than that is call_return, which takes in a parameter and returns a value when called, which needs to handled when translating. Moving on, fibonacci is more complex as it has a while loop in the body, followed by factorial which has both conditionals and function calls within the body. Fib rec and factorial rec are the most complex as they have all the complexities of the previous programs, alongside recursive calls which the compiler needs to keep track of on the stack.

Translation for call_param.py, call_return.py and call_void.py:

call_param.pep:

```
          BR      program
x:        .BLOCK 2
result:   .BLOCK 2
_UNIV:    .EQUATE 42
value:    .EQUATE 2           ;value #2d
my_func:  LDWA 2,s
          ADDA _UNIV,i
          STWA result,d
          DECO result,d
          RET
program:  SUBSP 2,i           ;Allocate #value
          DECI x,d
          LDWA x,d
          STWA 0,s
          CALL my_func
          .END
```

call_return.pep:

```
          BR      program
_UNIV:    .EQUATE 42
x:        .BLOCK 2
result:   .BLOCK 2
retVal:   .EQUATE 2           ;return #2d
value:    .EQUATE 0           ;value #2d

my_func:  LDWA 4,s
          ADDA _UNIV,i
          STWA retVal,s
          RET
program:  SUBSP 2,i           ; push #value
          DECI x,d
          LDWA x,d
          STWA value,s
          SUBSP 2,i           ; push #retVal
          CALL my_func
          LDWA 0,s
          STWA result,d
          ADDSP 4,i           ; pop #value #retVal
          DECO result,d
          .END
```

call_void.pep:

```
          BR      program
_UNIV:    .EQUATE 42
value:    .EQUATE 0           ; value #2d
result:   .EQUATE 2           ; result #2d
my_func:  SUBSP 4,i           ; push #result #value
          DECI value,s
          LDWA value,s
          ADDA _UNIV,i
          STWA result,s
          DECO result,s
          ADDSP 4,i           ; pop #result #value
          RET
program:  CALL my_func
          .END
```

Implementation of functions in our translator

The translator is implemented by adding to the visitor TopLevelProgram. As there are now more node types that are visited due to functions being added, we implemented visit_functiondef, visit_return, visit_call and visit_expr. Memory also has to be allocated for the parameter, so that we can call functions by value, rather than having to pass in a pointer. We allocate memory in the generator StaticMemoryAllocation. As we iterate over the AST

when we reach a function definition we generate the instructions for it, and add it alongside a label. This label can be used to call the function, and the result from the function will be properly passed through by the return. Everything is stored with SUBSP before a function then added a return value then extracted everything after. Once it is time for the value to be returned, the value is removed and the memory de-allocated to avoid overflow. As the stack is not properly utilized, the order of function calls may not work properly.

What happens if there is a 'Stack Overflow'?

If a "stack overflow" occurs in a program, it means that the program has attempted to use more memory space on the stack than is available. This can happen if the program contains an infinite loop or if the program has a large number of nested function calls without proper management of the stack.

In general, a stack overflow will result in the program crashing or becoming unstable. Depending on the specific circumstances, it may be possible for the program to continue running, but it will likely produce incorrect results or produce other errors.

To avoid stack overflows in assembly programs, it is important to properly manage the stack and ensure that there is enough space available for all necessary function calls and data. This can be done by carefully planning the use of the stack, using stack-related instructions in the program, and properly allocating and deallocating stack space as needed.