CIS3190W15: A4 Ackermann's Function

by Ryan Wilson-Perkin

Ada

Iterative

Ada lent itself nicely to an iterative implementation of Ackermann's function due to the ability to easily encapsulate global variables, functions, and procedures into a single package. At the professor's suggestion, I created a "stack" package with the recommended subroutines for accessing and modifying the stack.

In Ada, procedures differ from functions by not having an explicit return value. Procedures worked well for creating subroutines like push and pop which pass a value by reference but don't require a return value. Functions work better for subroutines that don't need arguments and return a single value, like stackEmpty and stackFull.

Once the stack package was built, it was a simple matter of combining it in the compiled program, and specifying "with stack; use stack;" at the top of my iterative procedure in order to include the functionality.

Recursive

The professor's code was used for the recursive Ada implementation.

C

Iterative

The professor's code was used for the iterative C implementation, albeit with several modifications to make it simpler. The time library was removed, in favour of tracking the runtime of the executable externally (as with all the other programs). The "struct dataT" type was deemed to be unnecessary, and was removed in favour of a simple array of integers inside of the "struct stack" type. Rather than have explicit checks for the value of st.top inside isEMPTY and isFULL, the program now simply returns the result of the relational expression comparing them to their intended values. Several sections were restructured to be shorter or simpler.

Recursive

The recursive C implementation, follows very closely the provided Ada implementation. The prompt is structured in the same way, and the inputs to the Ackermann function are the same. The main difference is that the variables are not passed by reference, and so we have no

qualms about modifying their values inside the function scope. Recursive calls are chained, in the case of n > 0, with C evaluating the innermost expression first. Given my experience with C, this was by far the simplest “legacy” language to write the implementation in.

FORTRAN

Iterative

FORTRAN felt a bit messy for writing this program in. Much like the iterative Ada implementation, I factored out the stack logic into it’s own module, but the module was decidedly less simple.

Good FORTRAN programs make use of the “implicit none” statement in order to ensure that static type checking catches usage of any undeclared identifiers at runtime. The caveat to this, is that FORTRAN does not resolve function names, and so every function scope must have a forward declaration for any functions it intends to call, as well as a “use stack” statement to have access to the global “stack” module. Additionally, FORTRAN functions and subroutines do not explicitly support the common “return” statement, opting instead to use “intent” in variable declarations, or “result” in the function name to specify what should be returned from the function’s call stack.

The actual Ackermann function implementation however was quite simple. It used the “call” keyword to access the push and pop subroutines and called functions like stackEmpty directly, using the negation of it’s logical return value to specify whether to continue looping.

Because FORTRAN function arguments are passed by reference, it was necessary to specify function-scoped variables to take on the initial values of m and n (read only memory) so that they could be mutated within the algorithm. The “result” keyword was then used to infer “intent(out)” for the function-scoped n variable.

Recursive

An interesting feature of FORTRAN is that it requires functions that will be called recursively to be prefixed with the “recursive” keyword. Presumably this is because, as previously noted, FORTRAN does not seem to be storing the function identifiers in a global scope and forward declarations are required in each function scope. The recursive keyword can act as a hint to the compiler that the function will make use of it’s own identifier within the function scope. It would be fun to dig into the gfortran compiler frontend and see how this functionality is actually handled.

Aside from the “recursive” keyword, there was nothing that made the FORTRAN recursive implementation easier or harder than any of the other recursive implementations, all of which are very straightforward.

Python

Iterative

"Great, kid. Don't get cocky."
Thanks for letting us use a fun language at the end! Certainly the simplest program to implement (although not particularly more difficult than C), my Python script for this was written very quickly. It is an iterative solution as well, and so requires a stack variable to track iterations.

Luckily, lists are a first class object in Python and include all the methods necessary to emulate a stack. The list object can be tested for emptiness directly by evaluating it in the context of a logical expression, it will evaluate to False if empty. The list object also contains both append, and pop methods, used for adding and removing values from the stack.

Of course, the relative overhead of mutating what is effectively an obfuscated array of void pointers is an expensive operation. Python was consistent an order of magnitude worse than all other implementations **combined.** This is the price you pay for convenience.

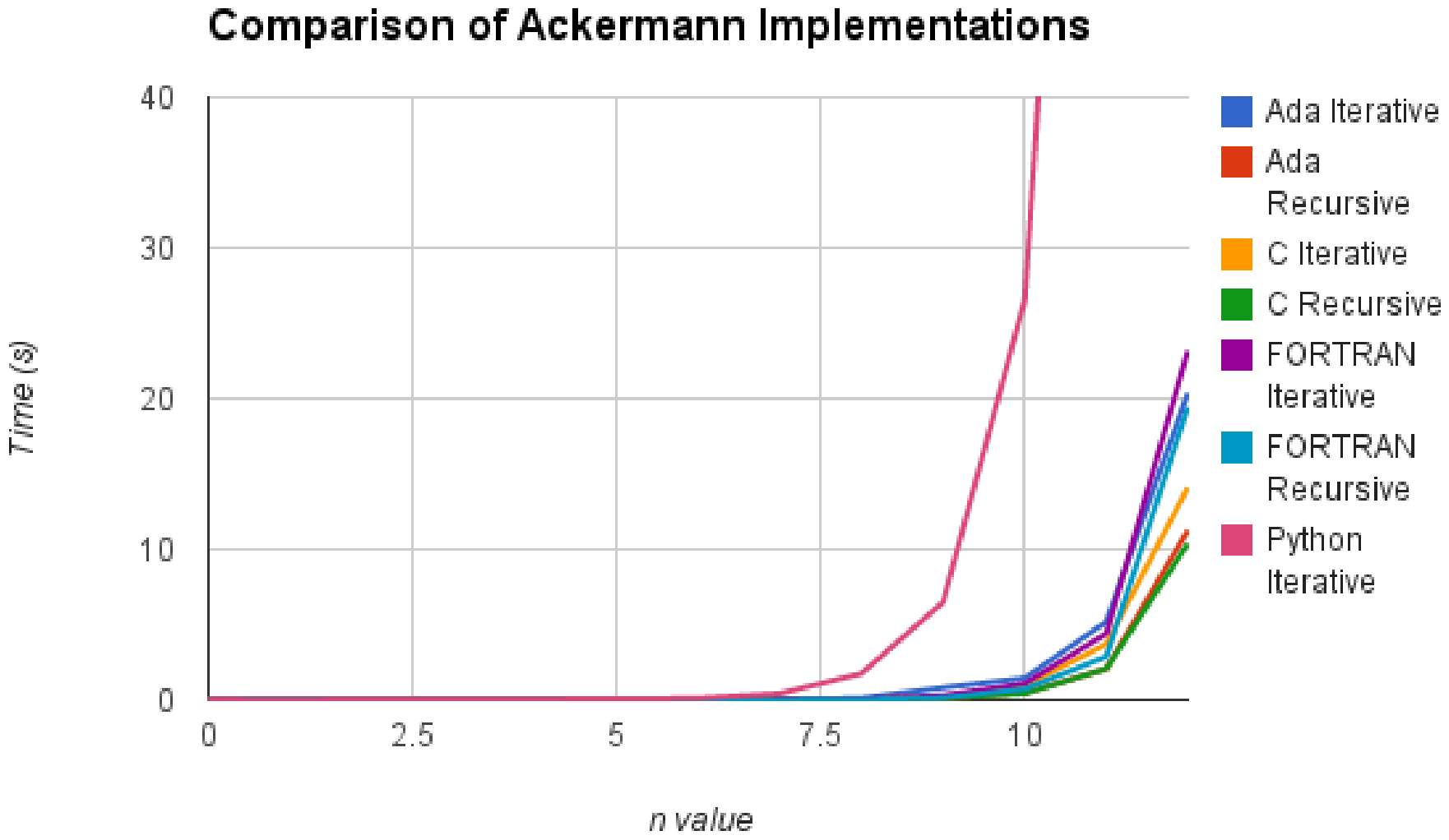
Benchmarking

In order to test the speed of each executable, I built a common interface to them and then wrote a benchmarking tool called test.py. Each executable has the same interface: it prints “Enter m and n:”, then reads two integers from standard input, then runs the algorithm, then prints “Result: “ followed by the resulting value.

Once the executables were verified for correctness of calculation, it was no longer necessary to track their actual output, only the amount of time that each took to run. Python has a great library for this called “timeit” which takes a process to run and measures the execution time. Coupled with the “subprocess” library, I used timeit to measure how long it took to spawn a new subprocess, feed it the arguments to the function, and wait for the subprocess to complete.

As mentioned, Python was consistently an order of magnitude worse than the other implementations combined, and so it is worth noting that likely some of this slowness is due to the fact that the python interpreter needs to be started each time. Python is not a compiled language and so it is almost an unfair test to make here. It is still interesting to see *just how bad* Python performed though. The following chart is truncated, but were it shown to scale, you would see that Python took almost *400 seconds* to complete the last test.

The chart measures the time that each implementation took to compute the Ackermann value given a variable n value and a constant value of m = 3.


Comparison of Ackermann Implementations
Time (s)
40
30
20
10
0
0
2.5
5
7.5
10
n value
Ada Iterative
Ada Recursive
C Iterative
C Recursive
FORTRAN Iterative
FORTRAN Recursive
Python Iterative