INF-1100: Innføring til programmering og datamaskiners virkemåte

Exam

November 19, 2024

1 Introduction

This report describes the implementation and design of an interpreter for a custom ready made stack based language. We were given an outline on how the stack based language, called 8Inf, should function. Our task was to create an interpreter. We will go into detail on our design and the benefits and drawbacks.

2 Technical Background

2.1 Stacks

Stacks are a linear structure based of the First-In-Last-Out (FILO) principle. Data is added or removed from the top of the stack using operations known as 'Push' and 'Pop' respectively. A stack-based language is a programming language that operates on one or more stacks. [1]

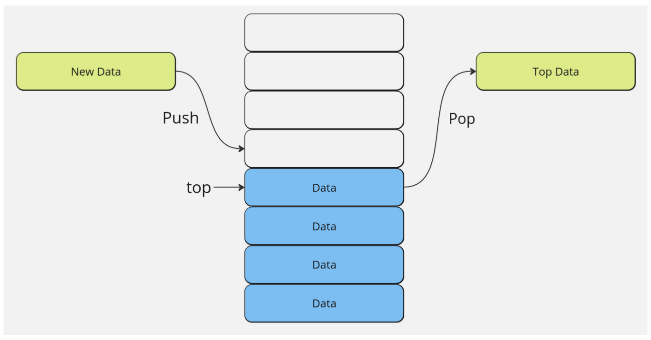

Figure 1: A visualization of a stack and pop/push

2.2 Interpreters

All programming languages need to be translated into machine code for the computer to be able to run it. One of the ways of doing this is by using an interpreter. The interpreter reads the source code line by line, or instruction by instruction, and translates it to machine code and executes it immediately. [2] This can be slower than a complied language, which translates the entire program at once before execution.[3] An example of this is Python who's interpreter (e.g., CPython) is written in C.

3 Design and Implementation

3.1 The stack first implementation

We decided to first implement our stack by using a string array and an integer representing the top index within a struct. This would make it so that if needed two stacks can exist at once. The stack can handle two datatypes, strings and integers. Along with the struct there are 8 functions that make the stack operations possible. Most of which are based on the pop and push function described earlier.

The problem with this implementation was the need the keep converting data back and fourth between integers and stacks. The biggest problem was the use of the function sprintf. Sprintf prints formatted data into a buffer sting. The issue being the size of said buffer. If the buffer is not big enough you can get unexpected behavior which made the interpreter unreliable.

3.2 The stack final implementation

The second and final interpretation is built on the first one. Instead of using an array of stings we decided to create a stack element type with a union and then use that for the array. This makes it so we can have a stack with both integers and strings without having to keep converting them. It also opens up the option to have other variable types in that stack if we were to expand the 8Inf language. We also shortened it to six supporting functions instead of 8.

3.3 Cjump

Cjump is the function that is in charge of the loop mechanism in 8Inf. It works by getting the two top numbers in the stack. If the second number is not 0 the program jumps by as many as the first number. Cjump can jump both forward and backwards in the program. Meaning it can be used for loops but also skip certain sections of the code if a condition is met.

4 Experiments and Results

One common downside to interpreted languages is how slow they run. Python for example is known as a slow interpreted programming language. This can easily lead to bottlenecks so we decided to benchmark the two implementations of 8Inf language and comparing it to Python and C. The way we chose to do this is by making a program in all three languages that counts and prints to a chosen number. Our first test case is 10 000 000 and then scale it up to 100 000 000. This would hopefully highlight any significant differences between the languages.
One thing to be aware of when timing code is how many sources of error there are. These measurements were done on a laptop plugged into power and with minimal background activity. We uses the external command **time** to time the code.

Table 1: Runtime in seconds of a loop counting to two different numbers

Language	**10 000 000 Iterations**	**100 000 000 Iterations**	**Difference in % of C**
Python	20.245	219.698	179.1%
C	11.527	122.661	N/A
Final 8Inf	14.375	153.632	125.3%
First 8Inf	16.106	166.288	135.6%

Above are the results of the tests done. The table shows a considerate difference in runtime. Python is by far the slowest of the three and C the fastest. We have used C as a baseline to show the runtime in percent. Python runs 79.1% slower than C. The first implementation of the 8Inf interpreter runs 10.3% slower than the final implementation. With both versions being significantly faster than Python.

5 Discussion

The results of the experiments were not unexpected. By its nature interpreted languages are slower than complied languages such as C. This made them ideal for testing 8Inf. 8Inf just like python is an interpreted language. The reason 8Inf is so much faster than python is because 8Inf is a much smaller and simpler language. If we were to expand 8Inf we would catch up to the run time of python fairly quickly. This is because our interpreter is not very optimized as Cpython.

The more exciting part of the data is the improvement between the implementations. The final implementation runs just over 10% faster than the first one. But most importantly the final one is much more reliable. When trying to profile the two implementations to check how much time each function uses, the first implementation broke. By the time we fixed the code it was still unreliable and it had been changed enough for it to not be the same implementation as in Figure 1. The sentiment still stands. The final implementation just works much better than the first one.

Clearly there are some big design flaws in the first implementation. This is mainly due to the function sprintf. Sprintf can be unsafe because it can 'print' a lager string than the buffer can hold and this can cause issues. One way to avoid this is to use a function called Snprintf which does protect against an overflow. We decided against using more time to try and fix the first implementation when the final version already works much better.

What makes the final implementation so much more reliable is not using sprintf or something similar. The best way to avoid buffer overflows is by not using any buffers.

6 Conclusion

After two attempts we made a well functioning interpreter for the stack based language 8Inf. Our first implementation, though sometimes functional, was plagued by buffer overflows which made it unreliable. To address this we made new implementation more focused on versatility and reliability. This resulted in a semi-efficient interpreter.

7 Sources

References

[1] GeeksforGeeks Contributors, *Introduction to Stack Data Structure - Algorithm Tutorials*, GeeksforGeeks, 2024. Available at: `https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/`. Accessed: November 5, 2024.

[2] GeeksforGeeks. "Introduction to Interpreters." *GeeksforGeeks.* Available at: `https://www.geeksforgeeks.org/introduction-to-interpreters/`. Accessed on November 11, 2024.

[3] GeeksforGeeks. "Difference Between Compiler and Interpreter." *GeeksforGeeks.* Available at: `https://www.geeksforgeeks.org/difference-between-compiler-and-interpreter/`. Accessed on November 11, 2024.

[4] OpenAI's ChatGPT, *Assistance with refining the structure, content, and clarity of the 8Inf interpreter documentation*, November 2024. Accessed via OpenAI platform.