Recursive Descent Parsing P0 Compiler in Golang

Bartosz Kosakowski, Justin Vu, Melissa Gonzales
Department of Computing And Software
April 2020

Research Challenge

The goal of our project was to use Golang's concurrency features, primarily Goroutines and WaitGroups to improve the performance the Winter 2020 implementation of the P0 compiler. The original implementation provided was written with Python 3.6.9, running on JupyterHub using a single threaded recursive descent parser. Our implementation is based off of files provided in lecture five of CS 4TB3.

What is concurrency?

Concurrency can be defined as computing multiple processes at the same time. A process lives in its own part of memory and is separated from the rest of the program. Several threads can exist inside a process, and can share the memory within a process. A process can exist belong to one to many threads. [1]

How did we make use of concurrency?

A `WaitGroup` was defined to wait for `goroutines` to finish executing. We made use of Golang's goroutines to execute processes concurrently:

- A goroutine to remove whitespace (`EatWhiteSpace`).
  - Reduces the number of times the move ahead function `GetChar` is called and streamlines the source code to only provide information that is necessary to the compiler.
- A goroutine to remove comments (`EatComments`) to further reducing the burden of the parser
- A goroutine for the parser (`ParseInput`), with the data be condensed by the other threads, the parser only has to make sense of non-empty characters.

We found the idea of using threads on each of these tasks for the lexical analyzer from the *Improved Parallel Lexical Analysis Using OpenMP on Multi-core Machines* [2] paper by Amit Barvea and Brijendra Kumar Joshi.

Testing

- Each function was individually tested as we wrote it, testing it under various conditions.
- We then tested our solution as intersecting pieces of code came together to see that it still functions as intended.
- Finally, we tested the program as a whole by passing in various code snippets from the P0 Test Jupyter file, tweaking our solution as needed.

Challenges

- Developer experience - unfamiliar with Golang, however syntax was simple to pick up.
- Managing packages in Golang - we can only save and maintain our solution from the directory where Golang is installed. Results in a lack of modularity and simplicity in executing the application.
- Cannot use whitespace to delimit as indicator for when to identify a string of characters - for instance, `program test;` is read as `programtest;`.For certain keywords, we had to include a special delimiter (such as in the case of `do`).
- The original implementation of the P0 compiler is written in Python, a language that is dynamically typed - this allows for certain "shortcuts" to be made, which was impossible for our solution to emulate, leading to much more verbose code.

Statistics

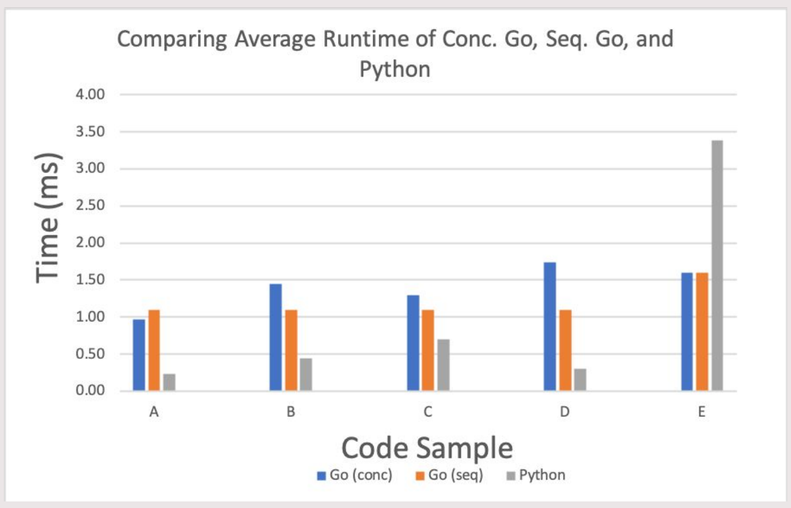
Lines of Code Per Component	
Component	Lines of Code
Code Generator	462
Lexical Analyser	75
Scanner	162
Symbol Table	164
Symbol Table Functions	57
Parser	762
Keywords	67
InputData	61
Main	26
Comments in Project	133
TOTAL	1836

Test Data		
Sample	Description	Performance Improvement Estimate
A	6 lines, var declaration, assignment, while loop, one arithmetic operation. 34 Whitespace, 58 Chars	36.95%
B	12 lines, multiple var declarations, more arithmetic and var access than sample 1. 111 whitespaces, 82 chars.	57.51%
C	14 lines, multiple var declarations, more arithmetic and var access than sample 1, multiple while loops. 115 whitespace, 149 chars.	43.56%
D	9 lines, var declaration, one arithmetic operation, multiple calls to standard procedures 36 whitespaces, 84 chars.	30.00%
E	46 lines, multiple const declarations, multiple var declarations, multiple assignments, many if-then-else statements, and many calls to standard procedures. 34 whitespaces, 997 chars.	28.73%

Performance Benchmarks

Each of these comparisons were done on five test cases pulled from the P0test.ipynb file from lecture 5 of CS 4TB3; the result is the average time over 10 runs of each case.

- The tests were of varying length and feature sets.
- The Go (concurrent and sequential) tests were performed locally by `go build` on Goland build #GO193.6911.30 running version 1.14 of Go.
- The Python tests were also performed locally on JupyterHub using Python version 3.6.9.



System Specifications of PC used for Testing	
CPU	AMD Ryzen 5 1600 Six-Core Processor ~3.2GHz
RAM	16GB DDR4
Operating System	Windows 10 Education 64-bit (10.0 build 17763)

Conclusion

- Our results demonstrate that our implementation tended to be slower than the Python compiler and sequential Go compiler in most cases.
- A possible reason for the slowdown is that the overhead of creating goroutines and the system automatically managing data locking to prevent mutex issues was more impactful than anticipated (i.e., the scale of the project might make parallelism negligible).
- The project proved to be a valuable source of knowledge on compiler design, the Golang language, and insight into how to approach such projects in the future.
- Potential areas of improvement could be to reduce the number type conversions, investigate other possible ways to implement functionality (for instance, appending strings with a plus compared to using a string builder object), and consider pursuing a non-recursive solution.

References

- [1] Software Construction - Concurrency https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/
- [2] Barvea, Amit *Improved Parallel Lexical Analysis Using OpenMP on Multi-core Machines* https://www.sciencedirect.com/science/article/pii/S1877050915007553#!

Kosakowski, Vu, Gonzales
McMaster University, Computing and Software
kosakowb@mcmaster.ca
vu4@mcmaster.ca
gonzam4@mcmaster.ca

BRIGHTER WORLD