Performance Testing Owl, Parser Generator for Visibly Pushdown Grammars

LUC TIMMERMAN, University of Twente, The Netherlands

The class of Visibly Pushdown grammars is one between type-3 and type-2 grammars as defined by Chomsky. For this class of languages the only currently publicly available parser generator is Owl, a parser generator written in C. Owl makes numerous claims about performance and lack thereof in certain conditions but does not list any tests to prove these claims. In this paper we run several performance tests with multiple measurements and discover that some of these claims are incorrect.

Additional Key Words and Phrases: Owl, Parser generators, performance testing, Visibly Pushdown

1 INTRODUCTION

Grammars and languages are important parts of Computer Science courses. They explain the theory behind programming languages and much more, like regular expressions. These are powerful tools for the student. In 1956, Noam Chomsky introduced his hierarchy for languages [3], giving us the four main classes known today;

- Type-0: Recursively enumerable languages
- Type-1: Context-sensitive languages
- Type-2: Context-free languages
- Type-3: Regular languages

According to some, this hierarchy is outdated. A paper in 2012 said: "…this fourfold distinction is too coarse-grained to pin down the level of complexity of natural languages along this domain" [8]. The paper claims that gaps exist in the Chomsky hierarchy and aims to fix some of those gaps by providing definitions for new classes of languages. One of the classes not mentioned in that paper is the class of Visibly Pushdown languages, or VPLs [1]. These are the target of the parser generator Owl [5].

In practice, Visibly Pushdown automata are finite state machines which recognize regular languages, with an added stack capable of keeping track of so-called brackets, like the tags in <i>abc </i> and the asterisk pairs in **abc**. As such, the stack allows them to travel arbitrarily deep into a nested structure. This is the minimum automaton capable of handling a Dyck language [9], or languages with "balanced brackets".

The following is the formal definition of VPLs as described by R. Alur and P. Madhusudan:

1. Firstly, we define a pushdown alphabet as a tuple $\widetilde{\Sigma} = \langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$ where Σ_c is a finite set of calls, Σ_r is a finite set of returns and Σ_{int} is a finite set of internal actions.
2. Then we can move on to the Visibly Pushdown Automaton. This is defined by the following: *A visibly pushdown automaton on finite words over $\langle \Sigma_c, \Sigma_r, \Sigma_{int} \rangle$ is a tuple $M = (Q, Q_{in}, \Gamma, \delta, Q_F)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, $\delta \subseteq (Q \times \Sigma_c \times Q \times (\Gamma \setminus \perp)) \cup (Q \times \Sigma_r \times \Gamma \times Q) \cup (Q \times \Sigma_{in} \times Q)$, and $Q_F \subseteq Q$ is a set of final states.*
3. Finally, we can define the Visibly Pushdown Languages: *A language of finite words $L \subseteq \Sigma^*$ is a visibly pushdown language (VPL) with respect to $\widetilde{\Sigma}$ (a $\widetilde{\Sigma}$-VPL) if there is a VPA M over $\widetilde{\Sigma}$ such that $L(M) = L$.*

This is very useful for rich text formats such as HTML or Markdown, and turning one into the other. While doing this is also possible in type-2 languages, these languages and parser generators for them often have quite some overhead because they introduce constructs that are unnecessary for this specific use-case – they have to in order to support the more complex type-2 grammars. VPLs, then, are a sweet spot between type-3 and type-2 that caters to this specific class of problems.

2 OWL

Owl is a parser generator for Visibly Pushdown grammars [5]. It is, at the moment of writing, seemingly the only available parser generator for this class. It is written in C and produces parsers in C.

2.1 Performance claims

Owl makes numerous claims on its performance. It claims to be efficient in its README file, saying: "Owl can parse any syntactically valid grammar in linear time", but it does not back this claim up. Additionally, the README also mentions two limitations:

- Large grammars: "Owl uses precomputed Deterministic Finite Automata (DFA) and action tables, which can blow up in size as grammars get more complex. In the future, it would be nice to build the DFAs incrementally."
- Memory use: "Owl stores a small (single digit bytes) amount of information for every token while parsing in order to resolve nondeterminism. If a decision about what to match depends on a token which appears much later in the text, Owl needs to store enough information to go back and make this decision at the point that the token appears. Instead of analyzing how long to wait before making these decisions, Owl just waits until the end, gathering data for the entire input before creating the parse tree."

None of these claims are elaborated on with any hard numbers or graphs. In the past decade and a half, the importance of replications in empirical software engineering has set in [2, 11, 14] and it is no longer acceptable to make claims without providing replicatable tests proving your claims. As such, this paper aims to create and run those tests and allow readers to run these tests and replicate them themselves at any point in the future. Testing the time Owl takes to parse a grammar is quite simple; we can just time Owl. Testing the size of an internal DFA is more difficult, and testing memory is,

TScIT 37, July 8, 2022, Enschede, The Netherlands
© 2022 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

in practice, also not simple. We aim to get a general idea of these metrics by recording the size of the parsers Owl generates.

2.2 Owl's grammar

In order to understand the grammars we generate later on, it is good to understand Owl's grammar. Owl's rules are written a lot like regular expressions, but using tokens. Owl has a few built-in token classes; `integer`, `number`, `identifier` and `string`. Literal text is matched like `'this'`. Rules are quite simple. The following is a grammar that matches an number of integers like this: `1, 2, 3`

```
list = number (',' number)*
```

Another important feature of Owl, needed to support VDLs, is *guarded recursion*. This is recursion inside guard brackets, which ensure the brackets are balanced. The following is an example from Owl's README file demonstrating a grammar that parses `{"arrays", "that", {"look", "like"}, "this"}` using guarded recursion:

```
element = array | string
array = [ '{' element (',' element)* '}' ]
```

A more detailed description, including more features like *expression recursion* can be found in the README of Owl at https://github.com/ianh/owl.

3 EXPECTATIONS

The fact that the creator of Owl specified that it was able to parse in *linear time* as opposed to it just being "fast" leads us to believe this claim is likely to be true. Regarding the claims about limitations, it is likely that the creator of Owl wrote these claims down after having experienced them firsthand, as discrediting your project without reason would nonsensical. The nuance in this lies in where problems begin to arise; do they appear in reasonable grammars or only when grammars get so large that they would rarely, if ever, occur in the real world? We expect problems will arise at large sizes, probably larger than is reasonable.

4 METHODOLOGY

In order to test Owl, we will take some inspiration from the paper on Event-Based Parsing by Vadim Zaytsev [16]. We will generate a series of grammars in Owl's syntax that test several features of Owl to see which feature puts the most stress on the program. To generate the grammars and measure results we will use Python 3.8. The size of the grammar we generate is dictated by input N. For this research, we have limited N to 10,000 in steps of 10 (note that 10,000 is not included, the highest value is 9,990). The tokens used in rules are randomly decided subsets of all tokens provided by Owl, so there can be an individual token or a choice of multiple tokens. The types of grammars are as follows:

- **"Many" grammars**: this is, together with "long"-type grammars, the most straightforward type of grammar; it is simply N lines of declarations, like so:

  ```
  a0 = number
  a1 = string | integer
  a2 = string | number
  a3 = identifier | number | string | integer
  ```

  We expect this type of grammar to be the easiest to parse as Owl does not need to keep track of anything to parse this grammar.
- **"Long" grammars**: these grammars are simply a single rule with a number of terminals behind it, like the following (with `N = 3`):

  ```
  long = identifier integer number string
  ```

  These grammars will likely also be easy for Owl to parse, but do provide additional context for comparing heavier grammars.
- **"Deep" grammars**: this type of grammar uses the calling of other rules, so that there are N + 2 nested calls (one call is for the terminal state which exists for any input, and one is the first call to the terminal rule). For example, for N = 3 we get the following grammar:

  ```
  a0 = a1
  a1 = a2
  a2 = a3
  a3 = a4
  a4 = identifier | string | integer
  ```

  If there is any grammar type that will be parsed in polynomial time instead of linear time, this is a likely candidate as Owl will need to keep track of a lot of these nested states.
- **"Nested" grammars**: nested grammars use Owl's guarded recursion feature to generate very deeply nested grammars. For `N = 3` we see this result:

  ```
  nested4 = [ '(' nested3 ')' ]
  nested3 = [ '[' nested2 ']' ]
  nested2 = [ '{' nested1 '}' ]
  nested1 = [ '(' nested0 ')' ]
  nested0 = number
  ```

  This is another likely candidate for non-linear computation times, especially since we expect the guard brackets to introduce extra complexity in the resulting parsers.
- **"Optionals" grammars**: these grammars strain Owl by using very deeply nested optionals. Since the main rule of this example is quite long, you can find it in Figure 1 using `N = 3`. There are eight terminals here for `N = 3` because it creates four (`N + 1`) "shells" around the main terminal in the middle, in order to make this happen, we need eight terminals.

We will measure performance in time to generate a parser from the grammars and the number of lines in the resulting parser.

The machine we used to measure on was a server with 50 gigabytes of memory and 8 cores (using hyperthreading) from a dual Intel®Xeon®CPU X5650 running at 2.67 gigahertz.

4.1 Replication

Of course, anyone can replicate the results shown in this paper. All code is visible at https://github.com/Luctia/owl_perftest. The

```
if = (terminal6 (terminal4 (terminal2 (terminal0 string | identifier)? terminal1)? terminal3)? terminal5)? terminal7
terminal0 = integer
terminal1 = string | identifier
terminal2 = string | number
terminal3 = identifier | string
terminal4 = string
terminal5 = identifier | number
terminal6 = integer | identifier | string
terminal7 = string | number | integer
```

Fig. 1. Example of "optionals"-type grammar at N = 3

repository includes instructions to perform the tests for yourself in the README file, which is also included in Appendix A.

5 RELATED WORK

A parser generator takes a grammar defined by a user, defining what they want a corresponding parser to do. This can be many things, from recognizing a valid e-mail address to parsing an entire programming language [4].

Parsing can be separated into two categories; top-down parsing and bottom-up parsing. Innovations are still begin made in both categories, like a combination of parsing strategies with top-down parsing at its roots by Terence Parr et al. [10], and research by Guide Wachsmuth et al. describes the current state of the art for bottom-up parsing [13].

5.1 Parser performance (testing)

While some people consider parsing a solved problem, this is not necessarily a consensus as is evident by the innovations still being made as we saw in the previous section. As such, parser performance from one to the other parser can differ greatly, and this is why it is still valuable to evaluate individual parsers' performance as well as parsing techniques on their own [12]. While Owl does give an indication of performance, there is a lack of proof, which could mislead users if the claims are incorrect.

The way we test Owl (generating larger and larger grammars with the same structure) is not the only way to test parser generators. Another possibility is taking existing grammars with actual real applications [6, 7]. This approach was difficult for us as Owl is relatively obscure, so getting a large number of existing grammars would have required too much time. Another automatic approach would be to create a certain grammar and "mutate" it to see what effect different mutations have on the parser. We did not choose for this approach as we were mainly focused on the performance claims by Owl's creators. Therefore our goal was stress-testing Owl, which small changes in grammars are not ideal for. Instead, creating larger and larger grammars provides us with a wider range of results.

6 RESULTS

Results for each grammar type will be displayed in graphs with two lines:

- Time: the blue line, corresponding to the left axis, will represent the time in seconds it took to execute `owl -c {filename} -o {outputfilename}`. This command tells Owl to take in a grammar by the name of `filename`, parse the grammar and output the resulting parser to `outputfilename`.
- Lines: the red line, corresponding to the right axis, will represent the total number of lines in the parser that has been outputted by the command above. While some of these lines will be comments, the difference this makes is negligible when considering the size of the files.

6.1 "Many" grammars

Results of the tests run for "many"-type grammars can be seen in Figure 2. The number of lines is clearly linear, which is no surprise. Time to compute seems slightly exponential, although the curve is quite understated. Especially when looking at the time it took at the highest level (a grammar of size 9,990), it is apparent that this type of grammar does not pose any issue for Owl with a time of around 1.5 seconds.

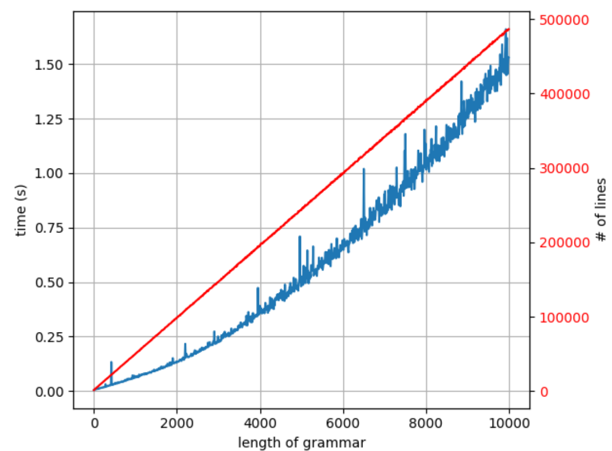

Fig. 2. Results of tests on "many"-type grammars

6.2 "Long" grammars

"Long"-type grammars scale by far the best, as is clear to see in Figure 3. The only real competitor (time-wise) here are the "many"-type grammars, which, at a length of 9,990, takes about 4.5 times

longer. It is unparalleled in terms of linecount; the closest (reliable) competitor here is also the "many"-type grammar, which produces a file 6.4 times longer than the "long"-type grammar at an input size of 9,990. Neither of these things are surprising; for this type of grammar, Owl needs to keep track of only one rule, which is not the case for any other grammar type.

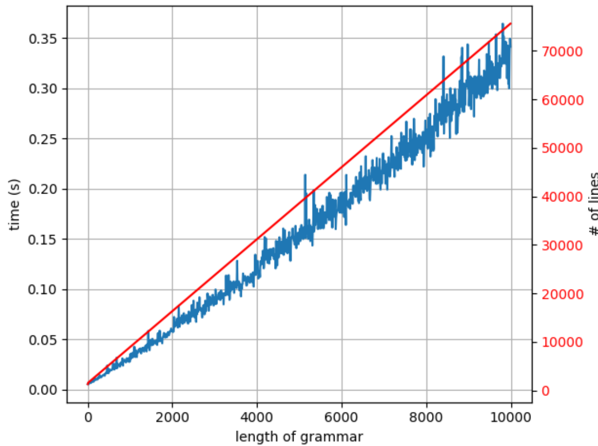

Fig. 3. Results of tests on "long"-type grammars

6.3 "Deep" grammars

For grammars of type "deep", as expected, the curve is more exponential as visible in Figure 4. As a matter of fact, with our specific setup, Owl gives up outputting at around `N = 1470` entirely. This is due to limits to memory usage; as mentioned before, we ran these tests on a server with eight cores and 50 gigabytes of memory, giving each core about 6.25 gigabytes of memory. When running single threaded (which was not feasible due to time limits), it becomes clear that Owl uses about 6.6 gigabytes of memory when parsing a "deep"-type grammar with `N = 1470`, which is too much for the memory allocated to the thread. What is interesting here is that Owl does seem to continue parsing at higher values of `N`, as the time measurements do not hit some ceiling reached at `N = 1460`. This could be improved upon in further research, but given the graph peaks at 81,739 lines, it is not necessary in order to get the full picture: the generated parsers for this type of file grow extremely fast.

Look closely and you will notice a small bump in the time measurements right after the crash in linecount. The bump seems unrelated to that crash, as it occurs at a slightly larger grammar size. It also seems like it is not caused by a one-time error, as several results after each other are lower than the previous one. It was not immediately obvious to us what could cause this bump, and further investigation is outside the scope of this research.

6.4 "Optionals" grammars

This type of grammar was held back by an error from Owl: `error: operators are nested too deeply`. We found out that

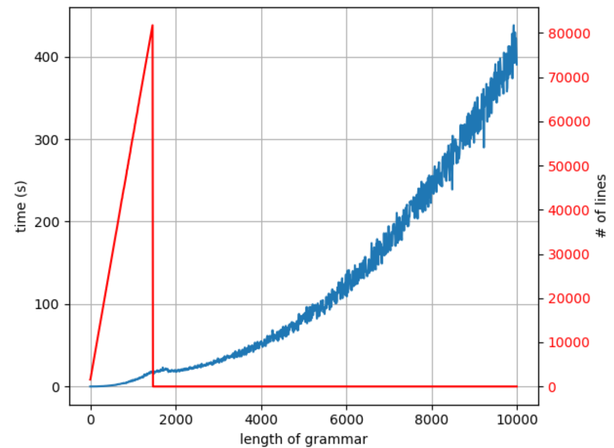

Fig. 4. Results of tests on "deep"-type grammars

Owl does not accept nested optionals of over 331 levels. This also explains the results shown in Figure 5; the randomness in number of lines from 331 forward is because the terminal values are randomly determined; when setting the terminal to a consistent token, the line becomes straight. The line for times remains jagged because host systems do not always execute jobs with the same number of interruptions, for example.

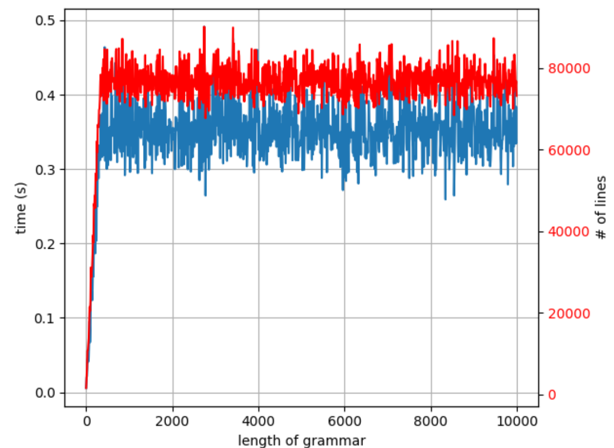

Fig. 5. Results of tests on "optionals"-type grammars

When we zoom in on the first few results, we can see a more nuanced result in Figure 6. We can see that this type of grammar also grows quite quickly in terms of time and size, but we will touch more upon that in section 6.6: Comparing grammars.

6.5 "Nested" grammars

"Nested"-type grammars are by far the most tolling on Owl from a time standpoint. Due to this, we set the maximum input to lower than the one we used for other grammar types, we went up to a size

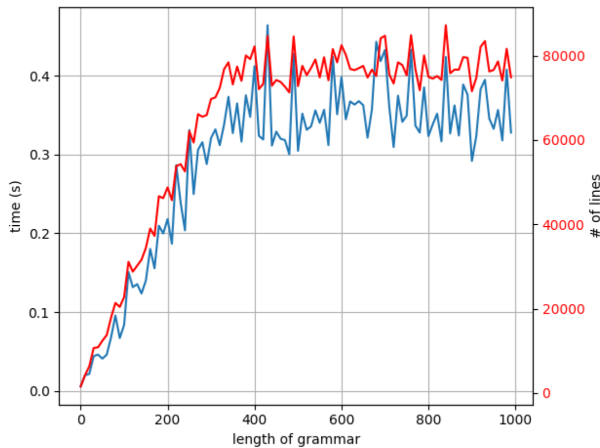

Fig. 6. Zoomed in version of Figure 5

of 3990 in steps of 10. Despite this, we believe this is still sufficient; a grammar with 1000 levels of nesting in one rule will likely never occur in real applications. A study by V. Zaytsev in 2015 found the biggest to be the Open Document Format, containing some 1000 nonterminals, 700 terminals and 2000 production rules [15]. The picture is still as clear though; this type of grammar is definitely computed in exponential time as is clear from Figure 7.

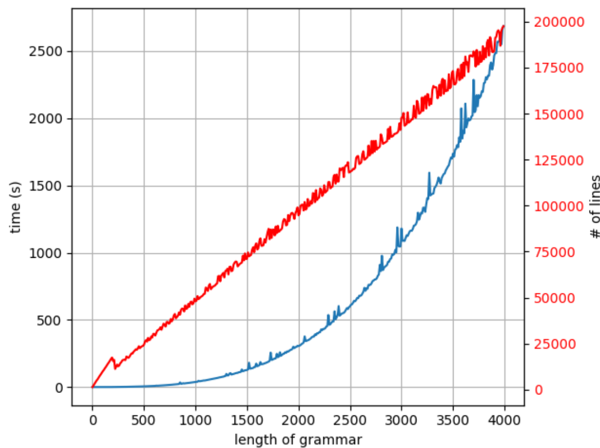

Fig. 7. Results of tests on "nested"-type grammars

At a size of 3990, Owl takes 2685 seconds to parse and generate the grammar. One thing stands out in this graph; the apparent drop-off in line count starting at `N = 200`. From this point on, Owl gave an error:

```
free(): invalid next size (fast)
fish: “owl -c tests/nested_990.owl. . . ”
  terminated by signal SIGABRT (Abort)
```

This is a C memory error which can happen, for example, when attempting to `free` a pointer that was not allocated or when attempting to delete a pointer more than once. This is seemingly an error in Owl, which raises two questions:

(1) Why does this error only occur when parsing grammars of this type and from a certain size?
(2) If this error occurs from a certain size, how come that the size of the resulting grammar does not stagnate from that point onward but instead keeps growing, although producing much less predictable output sizes?

Something else to note: parsing these types of grammars barely uses any memory, as was observed during the tests. Answering these questions above is outside the scope of this research.

6.6 Comparing grammars

6.6.1 Comparing times. When looking at the time Owl takes to parse grammars and generate their corresponding parsers, displayed in Figure 8, we can clearly see "nested"-type grammars are by far the most tolling and clearly exponential.

In fact, the difference is so extreme, that we can barely see the other types. For example, it is not immediately clear here that "deep"-type grammars parse in exponential time, as is clearly visible in Figure 4. To get a better look at the different grammar types, a zoomed-in version is displayed in Figure 9.

When looking at Figure 9, we should not forget the notable anomaly in the "deep"-type grammar (see Figure 4), which is not visible in this figure.

6.6.2 Comparing linecounts. In Figure 10 the number of lines for each grammar type is displayed, in which we have limited the horizontal axis to a size of 2500, since anything larger than this only shows more of the same. We can see here which type of grammar grows the fastest, which is obviously the "optionals"-type grammars. Something else worth noting is that, after its crash, "nested"-type grammars grow remarkably similarly to "many"-type grammars. We also see here that linecount does not necessarily correspond to memory usage; we encountered memory issues with the "deep"-type grammars, but here, there are other grammars without such problems with more lines than the "deep"-type grammars.

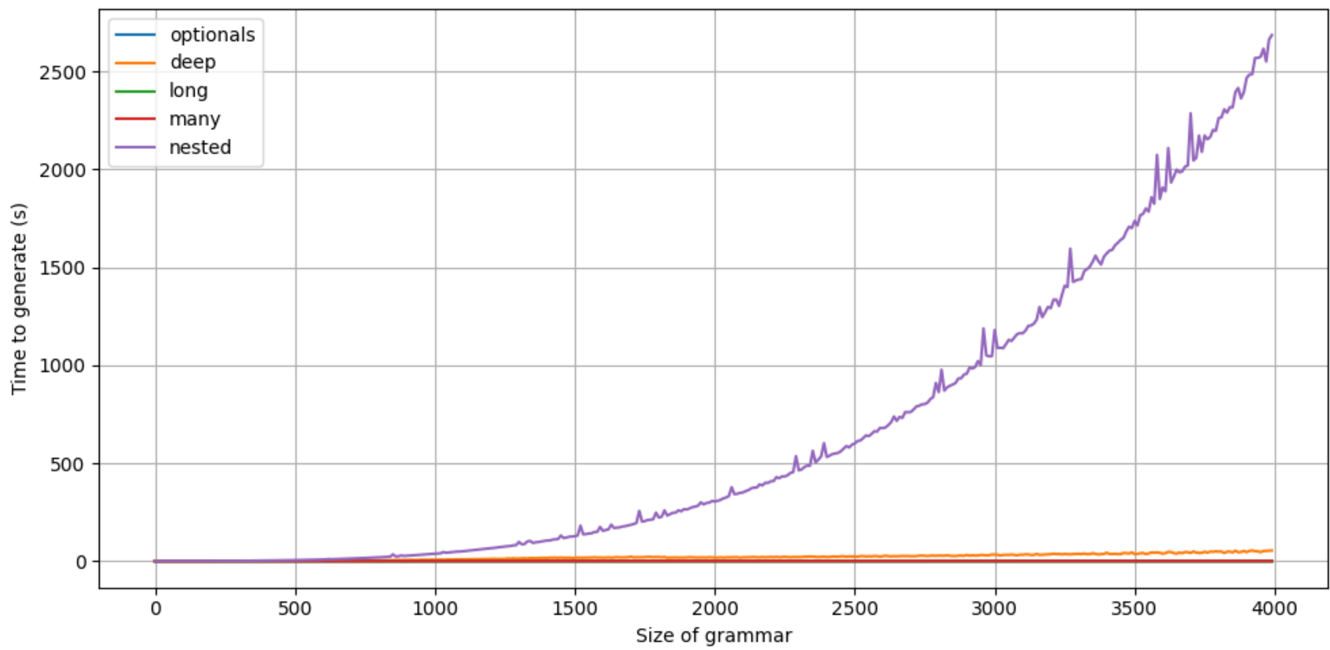

Fig. 8. Comparison of time at certain sizes

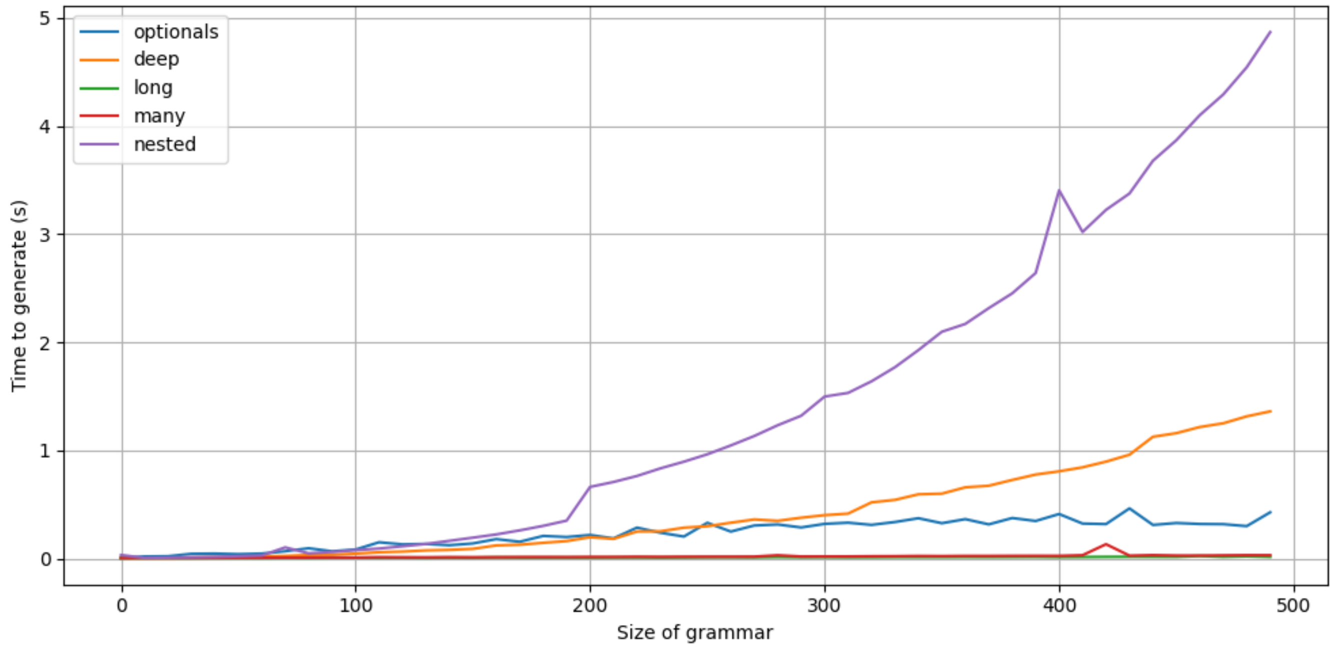

Fig. 9. A zoomed in version of Figure 8

7 NOTES ON AMBIGUITY DETECTION

One of the things we tried to test as well, but which is harder to automate, is how well Owl can handle ambiguity or otherwise confusing grammars. It was able to detect ambiguity in some grammars, like the following one, which can be problematic for other parser generators:

```
A = B | C | D
B = E | F | G
C = H | I | J
D = K | L | M
E = number | string
F = number | string
...
M = number | string
```

We did find one grammar which proved to have some interesting behaviour:

```
A = B*
B = C | D | '.'
C = integer
D = number
```

When inputting `1.5` in this parser, it concludes that this is a case of D. This is not necessarily correct, as it could also be interpreted as C, then `'.'` and then C again. When we remove the D state entirely, this is indeed exactly what the parser comes up with. However, when we remove the option D from B, but we leave D in as a rule, something peculiar happens; we get an error saying `error: unexpected number`. It is as if the parser always recognizes `1.5` as being a number when any rule mentions the `number` token, but is otherwise fine parsing it as a combination of two integers and a string.

8 CONCLUSION AND FUTURE WORK

In this paper, we have attempted to prove or disprove statements by the creator of Owl, a parser generator for Visibly Pushdown grammars. We have discovered that Owl does not always parse in linear time, in particular when many/large "deep"- and "nested"-type statements are used. We have also seen that for some grammar types, the size of the resulting parsers is rather big. We had hoped this corresponded to memory usage and size of DFA's, but this turned out to be false, although not a useless measure. We have also discovered some interesting limitations of Owl, like grammar size and the amount of nested optionals one can apply, although these limitations only show up when dealing with grammars so large that they likely will not appear in real applications [15] and so the results are not particularly damning for Owl.

One thing that we would have liked to measure but had difficulty doing was memory usage when parsing a grammar. Not only was the software built into Python a bit lacking in this area, several processes can also fight over memory, so the memory usage will be dependent on the memory size of the machine the measurements are running on as well as among how many threads this memory is divided. Given more time and a more powerful machine, perhaps a more complete image could be given by running one thread at a time with a lot of memory. It might be valuable to note that we did keep an eye on our memory during parsing, and we did notice that memory usage was often quite high, but eyeballing memory is obviously not a good way of measuring it.
Another part of Owl that would be good to test, is the performance of the resulting parsers. In this research, we wanted to focus completely on the claims by Owl's creator, but this aspect could also be very interesting.

9 DISCUSSION

We would like to touch on the performance claim Owl made: "Owl can parse any syntactically valid grammar in linear time". While we did record the time Owl took to run on different grammars, the times we recorded were, in fact, the time it took for Owl to parse the grammars *and generate the corresponding parsers*. In practice, Owl cannot be used another way; it has two modes: interpreter mode and compilation mode, which both need to generate a parser, so if they did mean only the parsing part, this would not mean that much in practice.

Another possibility is that they meant something else altogether; they could have meant that a parser generated by Owl parses in linear time. While this is not technically what the statement says, and thus not what this paper focused on, this is another reason why it might be valuable to look into the generated parsers in the future.

REFERENCES

[1] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing* (Chicago, IL, USA) *(STOC '04)*. Association for Computing Machinery, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[2] Jeff Carver, Natalia Juristo, Maria Baldassarre, and Sira Vegas. 2014. Replications of software engineering experiments. *Empirical Software Engineering* 19 (04 2014). https://doi.org/10.1007/s10664-013-9290-8

[3] N. Chomsky. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 2, 3 (1956), 113–124. https://doi.org/10.1109/TIT.1956.1056813

[4] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing Techniques — A Practical Guide* (second ed.). Addison-Wesley. https://dickgrune.com/Books/PTAPG_2nd_Edition/

[5] Ian Henderson. 2017. Owl. https://github.com/ianh/owl.

[6] Grzegorz Herman. 2020. Faster General Parsing through Context-Free Memoization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3385412.3386032

[7] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

[8] Gerhard Jäger and James Rogers. 2012. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences* 367, 1598 (2012), 1956–1970. https://doi.org/10.1098/rstb.2012.0077 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rstb.2012.0077

[9] Jens Liebehenschel. 2003. Lexicographical Generation of a Generalized Dyck Language. *SIAM J. Comput.* 32, 4 (2003), 880–903. https://doi.org/10.1137/S0097539701394493 arXiv:https://doi.org/10.1137/S0097539701394493

[10] Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 579–598. https://doi.org/10.1145/2660193.2660202

[11] Forrest J Shull, Jeffrey C Carver, Sira Vegas, and Natalia Juristo. 2008. The role of replications in empirical software engineering. *Empirical software engineering* 13, 2 (2008), 211–218. https://doi.org/10.1007/s10664-008-9060-1

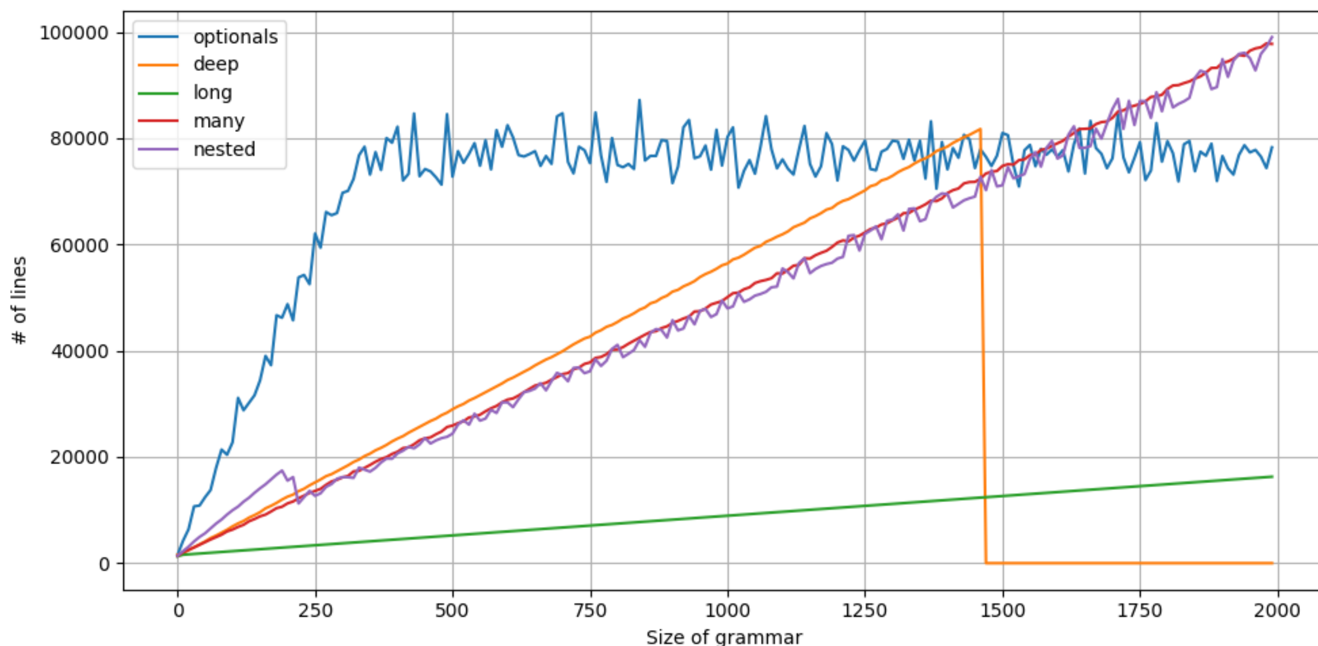

Fig. 10. Comparison of linecounts at certain sizes

[12] L. Thomas van Binsbergen, Elizabeth Scott, and Adrian Johnstone. 2020. Purely functional GLL parsing. *Journal of Computer Languages* 58 (2020), 100945. https://doi.org/10.1016/j.cola.2020.100945

[13] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. 2014. Language Design with the Spoofax Language Workbench. *IEEE Softw.* 31, 5 (2014), 35–43. https://doi.org/10.1109/MS.2014.100

[14] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated. https://doi.org/10.1007/978-3-642-29044-2

[15] Vadim Zaytsev. 2015. Grammar Zoo: A corpus of experimental grammarware. *Science of Computer Programming* 98 (2015), 28–51. https://doi.org/10.1016/j.scico.2014.07.010 Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012).

[16] Vadim Zaytsev. 2019. Event-based parsing. *REBLS 2019: Proceedings of the 6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 31–40. https://doi.org/10.1145/3358503.3361275

A REPLICATION INSTRUCTIONS

Running these tests for yourself is simple (these instructions should work for any operating system with:

(1) Clone the repository (at https://github.com/Luctia/owl_perftest);
(2) Edit the variables at the top of `main.py` to your liking:
- `TOTAL_MEMORY_GB`: the total memory in gigabytes which you would like to dedicate to the testing;
- `TOTAL_WORKER_COUNT`: the total number of threads you would like to dedicate to the testing;
- `TYPES`: the types you would like to test for (by default, this is all of them);
- `TEST_COUNT`: the number of tests;
- `STEP_SIZE`: the size of the steps in size between tests;
- `REMOVE_AFTERWARDS`: whether or not the generated grammars and resulting parsers should be removed after running.

As such, when using the default values (`TEST_COUNT` of 1000 and `STEP_SIZE` of 10), 1000 tests will be run with sizes `N = 0`, `N = 10`, `N = 20` up to `N = 9990`;
(3) Install dependencies (like `matplotlib` and numpy) using the method of your choice;
(4) Running can be done by simple executing `python main.py`. When testing large sample sizes, it might be beneficial to use `nice` (or `START` on Windows systems) to set a priority for the threads created by using `nice -n 10 python3 main.py`.

Depending on the setting of `REMOVE_AFTERWARDS`, this will generate a number of files:

- `{grammar-type}_result.json` files: these are files containing the results of the tests with an entry for every input size per grammar type including:
  - `time`: time to parse and generate in seconds;
  - `lines`: number of lines in the resulting parser.
- `{grammar-type}.png`: a fitted graph displaying the results for a certain grammar type;
- `tests/`: the grammars generated. The filenames are structured as follows: `{grammar-type}_{N}`;
- `parsers/`: the parsers generated by Owl. Their filenames correspond to those of the generated grammars.