Performance Analysis and Phase Bottlenecks in Abstract Syntax Tree Generation for Simple Expression Parsers

Steve Bradley Hoeij - 13525012
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: stevebradleyhoeij@gmail.com , 13525012@std.stei.itb.ac.id

***Abstract*—An Abstract Syntax Tree is a data structure which is often generated as part of the parsing process in program compilation. This process is often invoked at run-time in interpreted and at Just-In-Time compilers, which may directly impact the performance of the running program. Additionally, as codebases get more complex, the performative overhead caused by parsing may significantly increase. Thus, an analysis towards the algorithmic complexity of parsing may benefit our understanding of the performance of programs in large and non-deterministic codebases. Post analysis, it was found that the asymptotic complexity of a simple expression parser was [O(n)] and that parsing has more overhead within large operation counts.**

Key Words—abstract syntax tree; parsing; simple expression parser; performance

I. Introduction

Abstract Syntax Trees (ASTs) are data structures commonly used to represent code structure in programming languages. Before your computer can execute your code, it must first transform it into a strucutured and logical respresentation. This process, among many others, adds overhead towards the compilation process, which is especially impactful in interpreted and runtime-parsed environments that relies on dynamic AST generation. This is because any memory or latency overhead will directly impact the program's overall performance and responsiveness.

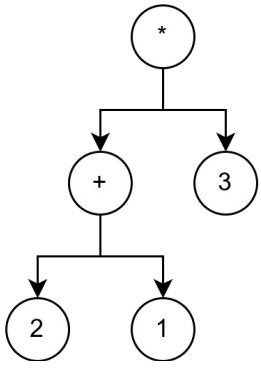


Fig. 1. A simple AST representing (2 + 1) * 3

AST generation is often categorized under the parsing process, but it could also include lexing. In lexing, the computer must first load the program from the hard disk, SSD, network, etc. into memory for processing, either by loading the program in full or in chunks/buffers. Then, the loaded buffer must be traversed to convert it into a stream of tokens for the parser. After that, the parser parses those tokens, analyzing and converting them into AST based on the language's grammar.

In general, the biggest limiting factor towards a program's execution speed is I/O and memory. Most CPUs nowadays have clock speeds greater or equal to 3 GHz, which translates to roughly 0.33 nanoseconds per clock cycle. Sometimes, they are also capable of executing multiple instructions within a clock cycle. This, by comparison, is much faster than the average speed of DRAM. Meanwhile, cache (SRAM & eDRAM), is even faster, costing only 2—4 cycles at the minimum (L1) and around 100 cycles at the maximum (L3) [1]. On the other hand, SSDs, which are on the faster end of I/Os, typically require 50,000—100,000 cycles.

II. Theoretical Foundation

A. Lexical Analysis

Lexical analysis, or tokenization, is the process of converting a contiguous stream of characters into a sequence of lexemes (tokens). The hand-written tokenizer that will be used in this paper has the LL(1) property, which means that it scans the stream from left-to-right, top-to-bottom, with a lookahead of 1. This means the cursor advances monotonically without backtracking, which, on average, translates to a $O(N_i)/O(n)$ linear complexity, where N_i represents the total length of the input buffer. In addition, the lookahead property is used to detect floating points. This is done by utilizing pointer offsets.

B. Operator Precedence (Pratt Parsing)

To resolve the issue of operator precedence without adding extraneous complexity, this system implements a Top-Down Operator Precedence parsing, formalized by Pratt [2]. This is done by assigning a left binding power of $\gamma(t) \in \mathbb{N}$ to token types. In this implementation, the precedence is defined as an enum with 4 values, 0, 1, 2, and 3, which corresponds to

PREC_NONE, PREC_SUM, PREC_PRODUCT, and PREC_UNARY, respectively.

In addition, Pratt's parsing also evaluates tokens based on 2 semantic functions:

1. Null denotation (nud)

   This handles prefix expression where there are no left-hand context, i.e. literals, unary minus (negative numbers/negation), or an opening parenthesis.

2. Left denotation (led)

   This handles infix/postfix expressions that already have a binding to an already parsed left-hand expression, i.e. binary operators.

This means that sub-trees are composed recursively as long as the next/rightward token holds a higher binding power than the current one ($\gamma(t_{next}) > \gamma(t_{current})$). Otherwise, recursion stops, and the parser returns to the sub-tree it has built so far.

C. Asymptotic Time Complexity

Asymptotic time complexity is used to represent time complexity for large order of magnitudes [3]. Commonly used notations are Big-O, Big-Omega, and Big-Theta.

1. Big-O [O(f(n))]

   $T(n) = O(f(n))$ if there exists positive constants C and n_0 such that $0 \leq T(n) \leq C.f(n)$ for all $n \geq n_0$.

2. Big-Omega [Ω(f(n))]

   $T(n) = \Omega(f(n))$ if there exists positive constants C and n_0 such that $0 \leq C.f(n) \leq T(n)$ for all $n \geq n_0$.

3. Big-Theta [Θ(f(n))]

   $T(n) = \Theta(f(n))$ if there exists positive constants C_1, C_2, and n_0 such that $0 \leq C_1.f(n) \leq T(n) \leq C_2.f(n)$ for all $n \geq n_0$.

Additionally, for Big-O notations, the standard hierarchy (fastest to slowest) is:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

III. Methodology

A. Pipeline

The system consists of a lexer/tokenizer and a Pratt parser. Normally, the AST produced is later processed within an intermediate representation (e.g. LLVM IR), but here it is omitted to purely focus on the parser efficiency.

Here, the lexer maps an input string S to a contiguous array of fixed-size Token structures. Character switching is done explicitly using a loop configuration over sequential memory offsets/increments. Memory allocations for string-based tokens (literals and identifiers) are managed dynamically in the heap and freed at the end of the process. Nodes are allocated each time a token matches a defined operation (literals, unaries, and binaries). Each node, depending on the type, may have 0, 1, or 2 childs, represented as a pointer to Node.

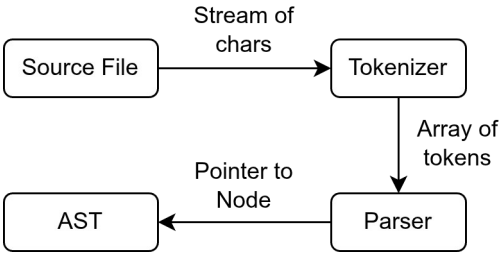


Fig. 2. Pipeline architecture

B. Implementation in C

To reduce unprecedented performance overhead due to linguistic quirks, this implementation of the parser is written in C and it uses C99 grammar. It consists of 5 files: tokenizer.h, tokenizer.c, parser.h, parser.c, and main.c. It was compiled with the command `gcc -O3 main.c tokenizer.c parser.c -o main`.

C. Pseudocode

Implementation of tokenizer is done with the following pseudocode.

```
FUNCTION init_lexer(file)
    file_size ← size of file
    IF file_size = 0
        RETURN error
    END IF

    buffer ← read entire file into memory
    IF buffer allocation fails
        RETURN error
    END IF

    initialize lexer pointers:
        start ← buffer
        cursor ← buffer
        line_start ← buffer

    line_number ← 1
    token_count ← 0

    allocate initial token array
    IF allocation fails
        free buffer
        RETURN error
    END IF
    RETURN success
END FUNCTION
```

```
FUNCTION start_lexer()
    WHILE current character ≠ EOF
        SWITCH current character
            CASE whitespace:
                advance cursor
            CASE newline:
                increment line number
                update line_start
                advance cursor
            CASE single-char operators/delimiters:
                add corresponding token
                advance cursor
```

```
            CASE digit:
                scan complete numeric literal
                add number token
            CASE letter or '_':
                scan complete identifier
                add identifier token
            DEFAULT:
                report lexical error
                add UNKNOWN token
                advance cursor
        END SWITCH
    END WHILE
END FUNCTION
```

Implementation of parser is done with the following pseudocode.

```
FUNCTION build_ast(tokens)
    initialize parser state
    root ← parse_expression(PREC_NONE)

    IF unconsumed tokens remain
        report syntax error
    END IF

    RETURN root
END FUNCTION

FUNCTION parse_expression(precedence)
    token ← consume next token
    IF token is a literal
        left ← create literal node
    ELSE IF token is a unary operator
        operand ← parse_expression(
                      UNARY_PRECEDENCE)
        left ← create unary node
    ELSE IF token is an opening delimiter
        left ← parse_expression(LOWEST_PRECEDENCE)
        consume closing delimiter

        IF delimiter does not match
            report syntax error
        END IF
    ELSE
        report syntax error
        RETURN NULL
    END IF

    WHILE precedence < precedence_of(next token)
        operator ← consume next token
        right ← parse_expression(
                    precedence_of(operator))
        left ← create binary node(operator,
                                  left,
                                  right)
    END WHILE
    RETURN left
END FUNCTION
```

D. Data Collection

Experimentation was done by a random sample data generation Python script and a data collecting Python script.

```
# generate_sample_data.py
import random
import sys
def generate_expression(op_count):
    if op_count <= 0:
        return str(random.randint(1, 100))

    if op_count >= 2:
        node_type = random.choice(["UNARY",
                                   "BINARY"])
    else:
        node_type = "UNARY"

    if node_type == "UNARY":
        operand = generate_expression(op_count - 1)
        return f"(-{operand})"

    remaining = op_count - 1
    left_budget = random.randint(0, remaining)
    right_budget = remaining - left_budget

    left_expr = generate_expression(left_budget)
    right_expr = generate_expression(right_budget)

    op = random.choice(["+", "-", "*", "/", "%"])
    return f"({left_expr} {op} {right_expr})"

TARGET_OPERATIONS = int(sys.argv[1])
OUTPUT_FILENAME = f"input{TARGET_OPERATIONS}.txt"

huge_expression =
    generate_expression(TARGET_OPERATIONS)

with open(OUTPUT_FILENAME, "w") as f:
    f.write(huge_expression)
```

Timing was done within the C program with the help of the `clock_gettime(CLOCK_MONOTONIC, &timestp)` function from `time.h`. Because of this, all time measurements from hereon will be in milliseconds. Some measured metrics include:

- Operations (N_{ops}), total number of operator nodes generated in sample data.
- Node count (N_{nodes}), total number of physical nodes instantiated in the resulting AST which is counted only after the AST has been fully formed .
- Lexing initialization time (t_{init}), total duration of file loading (init_lexer).
- Lexing time (t_{lex}), total duration of the tokenizing (start_lexer) loop.
- Parsing time (t_{parse}), total duration of Pratt parsing (build_ast) loop.
- Node counting time (t_{count}), total duration of complete AST traversal.
- Total time (t_{total}), total duration of a parsing cycle during experimentation which includes node counting.

Node counting was done to represent the simplest non-trivial operation you can perform on the AST.

E. Hardware & Testing Environment

The data was collected on the researcher's personal computer, which is an Acer Aspire A515-45-R0RG laptop. Additionally, the testing environment was a closed room with an air conditioning unit set to 27°C and the script was run in the Alacritty terminal emulator with Zsh and Tmux. There were no other active processes other than the benchmark.

TABLE I. HARDWARE SPECIFICATION

Component	Hardware Specification
Processor (CPU)	Ryzen 3 5300U
Core architecture	4 Cores/8 Threads @ 2.6 GHz
Cache capacity	L1: 64 KB per core L2: 512 KB per core L3: 4 MB (shared)
System memory (RAM)	4 GB Internal 16 GB DDR4 SODIMM @ 3200 MHz
Operating System	Arch Linux x86_64 Linux 7.0.12-arch1-1
Compiler	GCC 16.1.1 20260430
Miscellaneous	512 GB SSD PCIe Gen 3, 8 Gb/s, M.2 NVMe

IV. RESULTS

Here, asymptotic analysis is done to verify that both the LL(1) lexical analyzer and the Pratt parser operates within linear boundaries ($\Theta(n)$), meanwhile time measurements were made to address architectural constraints, such as locality, I/O, and memory hierarchy, and compiler optimization.

A. Authors and Affiliations

To enliminate transient timing spikes caused by background tasks and/or thermal scaling, benchmarks were grouped by their mass. In the first dataset, this was done by running 10 different operation sizes, which, for each of them, was tested on 100 randomly generated sample datasets, and aggregating them by the median. Meanwhile, in the second dataset, operation sizes were generated with the following formula to create a graph representing asymptotic growth:

```
sizes = [int(1000 * (1.5**i)) for i in range(20)]
```

which was tested on 10 randomly generated sample datasets and aggregated by the median.

In general, the higher the number of operations in the sample data, the higher the total processing time. Most of the latency is caused by lexing and parsing, which in both datasets, are the primary contributors towards total time. In dataset 2, as the operation counts increase, parsing time seems to grow

TABLE II. DATASET 1 SUMMARY

N_{ops}	N_{nodes}	t_{init}	t_{lex}	t_{parse}	t_{count}	t_{total}
1000	1386	0.019	0.122	0.061	0.009	0.204
2000	2774	0.027	0.405	0.086	0.018	0.509
5000	6935	0.038	0.650	0.297	0.044	0.999
10000	13864	0.050	1.046	0.647	0.088	1.769
20000	27730	0.071	1.970	1.339	0.240	3.439
50000	69341	0.172	4.745	3.401	0.864	8.503
100000	138620	0.503	8.310	6.744	1.754	15.642
200000	277269	0.848	15.076	13.392	3.4170	29.340
500000	693156	1.261	34.313	33.467	8.4095	69.096
1000000	1386322	1.977	64.423	66.662	16.726	133.214

TABLE III. DATASET 2 SUMMARY

N_{ops}	N_{nodes}	t_{init}	t_{lex}	t_{parse}	t_{count}	t_{total}
1000	1386	0.021	0.125	0.067	0.012	0.223
1500	2080	0.026	0.177	0.120	0.018	0.325
2250	3132	0.024	0.355	0.097	0.020	0.475
3375	4676	0.031	0.452	0.189	0.030	0.666
5062	7004	0.037	0.677	0.308	0.047	1.059
7593	10536	0.042	0.853	0.510	0.068	1.414
11390	15801	0.053	1.219	0.753	0.109	2.041
17085	23687	0.067	1.672	1.163	0.199	2.911
25628	35517	0.096	2.709	1.804	0.367	4.654
38443	53353	0.145	3.944	2.574	0.619	6.839
57665	79958	0.268	4.990	3.867	1.006	9.131
86497	119911	0.423	7.215	5.985	1.506	13.710
129746	179838	0.557	10.233	9.021	2.267	19.710
194619	269738	0.808	13.780	13.236	3.330	27.744
291929	404787	1.093	19.743	19.714	4.978	40.620
437893	607020	1.513	27.346	29.210	7.434	57.957
656840	910575	1.398	39.880	43.565	11.068	84.567
985261	1365919	1.916	57.552	65.872	16.608	125.332
1477891	2048533	2.444	83.564	98.379	24.797	184.537
2216837	3073189	2.763	123.797	147.085	37.249	274.726

quicker than lexing time. However, for small operation counts, lexing time dominates the total time.

Additionally, lexer init seems to cause little overhead, which is strange because I/O is supposed to be much slower than DRAM. For instance, a randomly generated sample data with 1,000,000 operator counts would roughly occupy 4.5 MB on the disk. From Fig. 3 we know that the testing machine has an SSD with a 8 Gb/s speed.

$$8 \text{ Gb/s} = 1 \text{ GB/s}$$

$$4.5 \text{ MB} / 1 \text{ GB/s} = 0.0045 \text{ s} = 4.5 \text{ ms}$$

From these calculations, we know that the SSD has a theoretical max speed of 4.5 ms, which is a discrepancy from dataset 1's median which is 1.977 ms. This may be because of operating system's page caching. Another possibility is the SSD used may have had multiple lanes (which is commonly the case in PCIe Gen 3 SSDs), so its performance may not necessarily be limited by the link speeds.

Next, to view the algorithm's complexity, both datasets can be represented as a scatter plots between operation count and total time to measure their relationship. Based on previous theories, the lexer's complexity should be linear because it posesses the LL(1) property. On the other hand, Pratt parsing utilizes recursion which can make it difficult to evaluate from just seeing its source code.

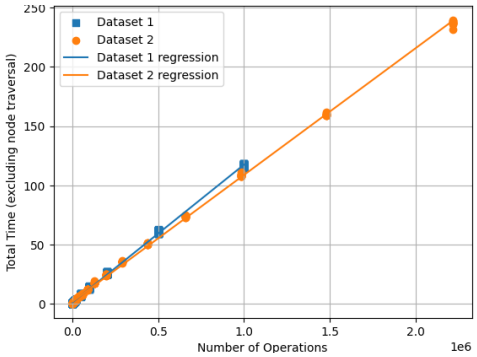


Fig. 3. Scatter plot between operation count and total time

If the scatter plot and regression describes a linear asymptotic complexity, we can infer that the complexity of the parsing algorithm has a O(n) complexity. However, to be certain, regression has to be done to measure the relationship between the number of operations and total time. Here, it is done with ordinary least squares (OLS) regression which resulted in the following time complexity formula for datasets 1 and 2.

$$T_1(n) = 1.163\times10^{-4}\, n + 1.148 \quad (ms)$$

$$T_2(n) = 1.070\times10^{-4}\, n + 1.759 \quad (ms)$$

On top of analyzing linear growth, linearity can be verified by computing the coefficient of determination (R^2) which results in the values 0.9991 and 0.9993 for dataset 1 and 2, respectively. This confirms that a linear regression can be used to represent the recorded data. The same method can be applied to individual processes like lexing or parsing, to analyze their regression separately.

1. Lexing

$$T_1(n) = 6.444\times10^{-5}\, n + 0.9793 \quad (ms)$$

$$T_2(n) = 5.586\times10^{-5}\, n + 1.452 \quad (ms)$$

$$R_1^2 = 0.9978$$

$$R_2^2 = 0.9985$$

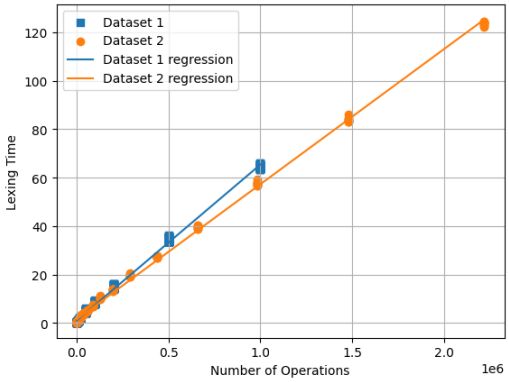


Fig. 4. Scatter plot between operation count and lexing time

2. Parsing

$$T_1(n) = 6.663\times10^{-5}\, n + 0.06193 \quad (ms)$$

$$T_2(n) = 6.645\times10^{-5}\, n + 0.1251 \quad (ms)$$

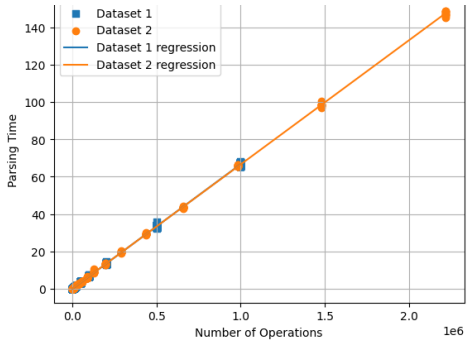


Fig. 5. Scatter plot between operation count and parsing time

$$R_1^2 = 0.9998$$

$$R_2^2 = 0.9999$$

As we have predicted earlier, the lexer's initial time cost is higher compared to the parser. However, due to the parser's steeper gradient, it is less efficient than the lexer on larger operation counts. Both shows a linear ansymptotic complexity.

V. Conclusion

It has been found that both the lexer and the parser scale linearly. Its total runtime grows proportionally to the number of operations, which can be seen in the derived regression lines. Results stayed consistent despite differing benchmark methodologies. Additionally, those regressions possessed a very high coefficient of determination, which suggests that a linear model is the best fit to for the observed variance.

Apart from the time complexity, it was also discovered that for large inputs, parsing becomes more costly than lexing due to the higher slope/gradient, meanwhile for smaller operation counts lexing is more costly.

VI. Appendix

C source code for the simple expression parser: https://codeberg.org/EnnEffDee/simple_arithmetic_parser

Acknowledgement

The author thanks God for His blessings and grace, their family for their support, their lecturer, Prof. Dr. Ir. Rinaldi, M.T., for his teachings and lectures, and their fellow classmates in IF1220.

References

[1] T. J. Johnson, "Cache Memory," Faculty Lecture Notes for ELE 655, Milwaukee School of Engineering. https://faculty-web.msoe.edu/johnsontimoj/ELE655/files655/cache1.pdf. Accessed 18 June 2026.

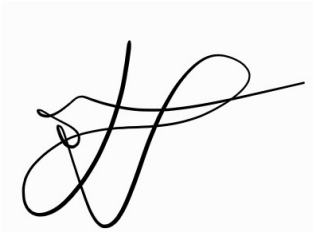
[2] V. R. Pratt, "Top down operator precedence," Proceedings of the 1st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 41–51, 1973.

[3] M. Rinaldi, “Kompleksitas algoritma (bagian 2)”, https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf, 2026. Accessed 14 June 2026.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2025



Steve Bradley Hoeij 13525012