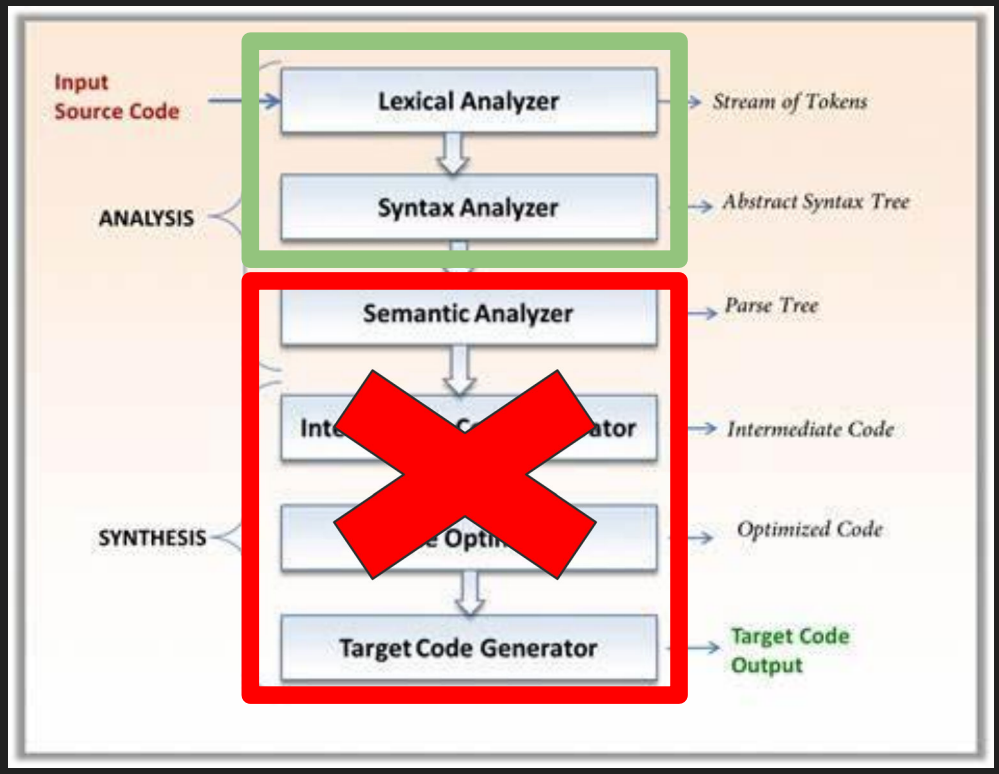
Compilation Improvements:

Lexical Analysis & Syntax Analysis

Compilation Process

For the purposes of this presentation, we will be focusing on the **Lexical**, and **Syntax Analyzer** phases of the compilation process.

There are many improvements that can be made to later phases in the compilation process, but I couldn't find any meaningful research that had these later phases as a focus.



Improvements to Lexical & Syntax Analysis

As we all know: both lexical, and syntax analysis phases focus on analyzing a given source code through tokenization, and the syntactic analysis of programming statements. We typically approach these problems through sequential (one at a time), or recursive (depth) approaches. But by modifying our thought process of how to solve these problems– we can achieve more efficiency and speed!

But what are the attributes of lexical, and syntactic analysis that allow us to make improvements to them in the context of efficiency and speed of this analysis?

Lexical Analysis:

Lexical analysis focuses on the correctness of whitespace delimited words in our source code– these individual words are compared to the individual parts of our grammar called ‘lexemes’. If a given word matches a lexeme in the language, it is classified as a token of that type.

"int" -> matches INT lexeme laid out by grammar -> classified as INT token

These token classifications are independent of other words within the source, so how can we utilize this fact to speed the process of tokenization?

An Example of Parallelization

Say I gave you, a single person– the entire novel “Hitchhiker’s Guide to the Galaxy”, and asked you to look up each word in the dictionary to verify that every word used was part of the English language– how long would this take you?

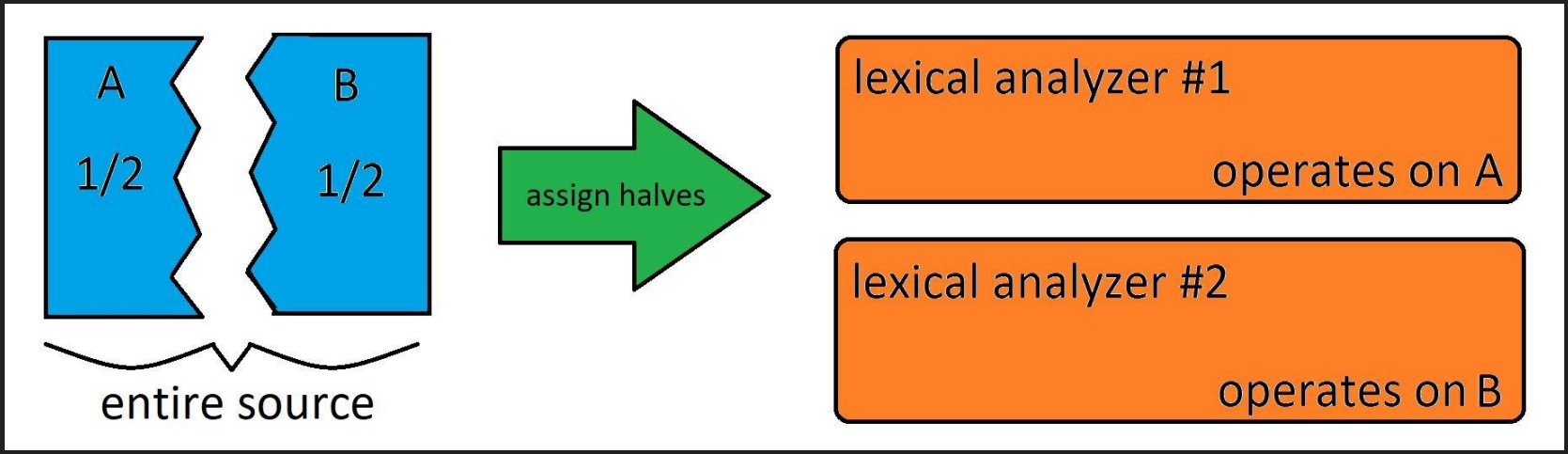
If you ripped the book in half, and handed it to your neighbour, and asked them to do the same task– how much would this speed up the process?

In the context of a source code, could we do the same thing? **YES;** because we’re focusing on individual tokens, not the meaning, or correctness of entire statements in the source.

Multi-thread approach to Lexical Analysis

Modern CPUs have multiple processor threads that can perform tasks in tandem utilizing the round-robin model of computation time allocation.

Below is a simplified model of how one might allocate a source code to lexical analyzers working on multiple process threads, with a control layer on top to return an error should one lexical analyzer encounter a discrepancy.



OpenMP: A Tool to Facilitate Parallel Lexical Analysis

By default in the GNU Compiler Collection, there is a tool called “OpenMP” (Open Multi-Processing) that can facilitate our need to compute programming functions on multiple threads. It affects run-time behavior, so for example if we were to utilize this tool in our course’s tokenizer code for a given source code input– we can parallelize the process of tokenization. By adding the below line, and making use of the library documentation available, we can optimize our tokenizer code to make use of parallelization.

```
#pragma omp default
```

Parallelization Performance Gain

If a given task takes ‘ts’ time to compute sequentially, and then ‘tp’ time to compute through parallelization– then the ‘speedup’ value is equal to:

speedup = ts / tp

Through experimentation done by Barve and Joshi: a 10,000 token source took 1415.48 ms to tokenize sequentially, and through parallelization using OpenMP this time was reduced to 206.87 ms; the result of which was a 6.84x speedup value. Nearly seven times faster!

Other uses of Parallelization for Optimization

Local/Block Optimization: In the process of code generation and structure– there exists the opportunity to run certain operations in parallel to reduce the computation cycles of a given block of code– this, much like the process of tokenization is subject to the same speedup benefits.

Syntax Analysis:

In our course, we utilized functions that made use of recursion to verify the syntactic correctness of programming statements. Recursion has the benefit of simplifying the process of syntactic analysis, but limits us to needing to perform this process sequentially.

In the aforementioned example of taking a novel and verifying it for correctness, can we verify whether a written work has all of the attributes that are required for it to be syntactically valid as a book by splitting into parts by using recursion? **NO;** because a book requires us to consider each sentence/chapter/volume (multiple layers) in the context of the rest of the book– so how can we restructure our process to allow us to improve the speed/efficiency of syntactically analyzing it?

Continuing the book metaphor…

An entire novel requires a specific number of chapters to qualify it as a book, i.e. a book with no chapters ≠ a book.

A chapter with no sentences cannot qualify as a chapter, i.e. a chapter with no sentences ≠ a chapter.

A sentence with no words cannot qualify as a sentence, i.e. a sentence with no words ≠ a sentence.

What then, is a suitable alternative to recursion to verify whether each layer of the book qualifies as what they are? **ANSWER: Iteration, and counting the repetition of necessary elements to qualify.**

Alternative grammar rules, with repetition requirements.

If we consider what qualifies as a ‘word’ in a written work, we can determine that a word can be comprised of individual alphabet characters– with a minimum repetition of 1 character (e.g. “a”), and (theoretically) no upper bound to how many letter can constitute a word (eg. “supercalifragilistic”).

If in our grammar rule for a word, if we specified the expected repetitions of a specific element (in this case: characters) to be between 1, and infinity– we could write the rule as:

WORD = 1 * ∞ (‘A’ - ‘Z’ OR ‘a’ - ’z’)

Continued…

The rule for a WORD is fine and well, but when we move up to evaluating whether a sentence is correct– we need to be more explicit in terms of what qualifies!

A complete sentence requires: Starting with a capital letter, having a subject, and a predicate (verb), it must convey a complete thought (fairly abstract), and ends with a period, question mark, or exclamation point. Stripping away the abstract requirements– we might write a sentence thusly, (with the in-between rules from WORD up to the elements in the below rule being implied):

SENTENCE = (1*1 CAPITAL-WORD) AND (1*∞SUBJECT) AND (1*2 SUBJECT-ACTIONS) AND (1*1 END-SENTENCE-PUNCTUATION).

Continued x2 …

If we keep working our way through the grammar, specifying what constitutes each element/rule in our language, we eventually end up with a completely deterministic language, which specifies how many repetitions (none, up to infinity) we expect to see of tokens, and sub-rules for each parent rule.

This format of grammar rules eliminates infinitely recursive rules and allows us to count the elements required for each rule, but comes with the cost of grammar rule length, and complexity.

Why would we exchange a simplified version of a grammar for a more complicated and explicit one?

Relating the Book Metaphor to Code

If we take the book metaphor and relate it to source code, we would discover that syntactically, we can approach the problem in a similar format. Whereby a source code can have 0 to infinity statements, each statement requires a specific repetition of tokens, and tokens require a specific repetition of characters to qualify as syntactically correct.

If the context of statements doesn’t matter in relation to other statements in the source, (unlike in semantic analysis) then could we not examine each layer of the program (source, declaration block, statement block, etc.) in pieces to verify whether it is syntactically valid?

Enter Tunnel Parsing, with Counted Repetitions

Tunnel Parsing is an algorithm that utilizes repetition, and stack data structures to simulate scope, and count the number of repetitions that the rules of a given language need to be correct.

Because the algorithm is iterative in nature, it allows us to take advantage of optimizations such as parallelization like in the case of lexical analysis– but due to the need to consider scope, requires the use of stacks to simulate this scope of rules being processed at the same time.

NOTE: The explanation given in this presentation is simplified heavily, and the inner workings of the algorithm are much more complicated than what is shown.

Stacks used in Tunnel Parsing Algorithm

There are two primary stacks used in the Tunnel Parsing algorithm for the sake of parsing:

- Repetition Stack (RS), tracks the repetition of individual elements
- Depth Stack (DS), tracks the repetition of specific rules (this simulates scope)

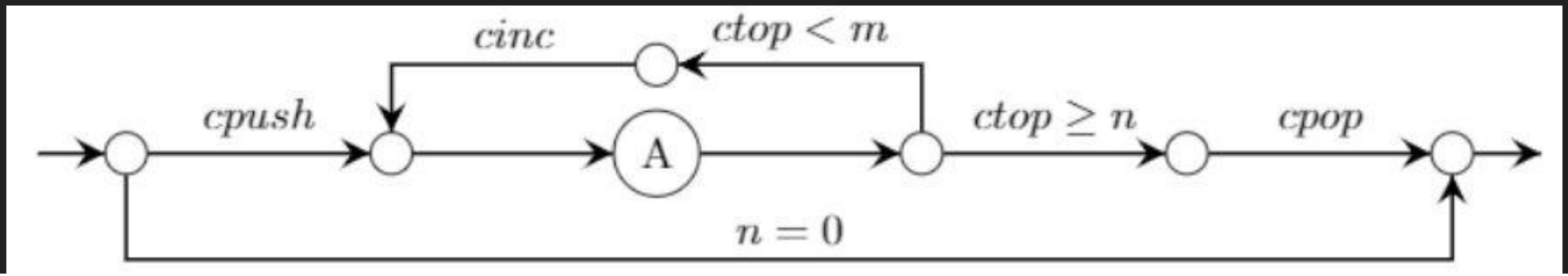
These, along with the ‘archive stacks’, i.e. RS Archive, DS Archive– allow us to utilize the tunnel parsing algorithm correctly; the archive stacks allow us to construct a parse tree by ‘unwinding’ the layers of the archive stacks to construct a concrete syntax tree, or CST.

Stack operations

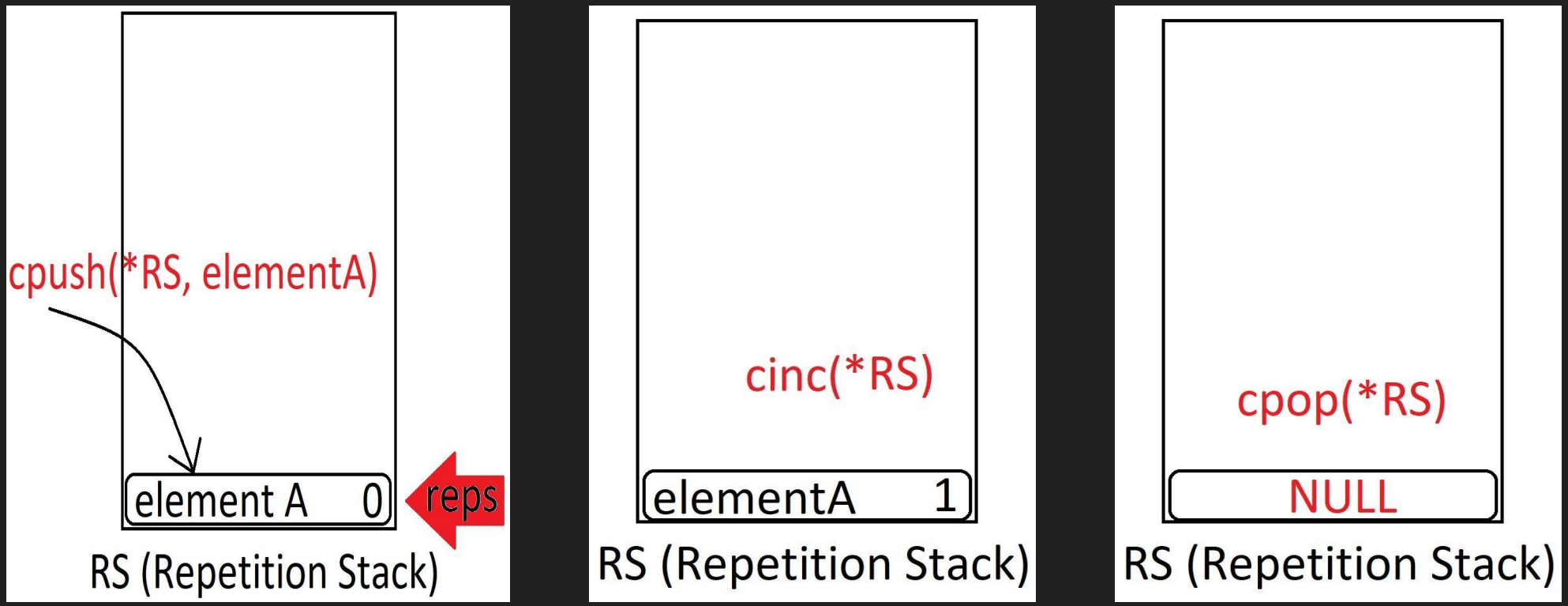
As is typical of stacks, there are a number of operations that we can perform on them in order to track repetition!

- ‘cpush’: pushes an element onto the stack to track repetition
- ‘cpop’: removes an element from the stack when repetitions cease
- ‘cinc’: increments the count of repetitions when a particular element is encountered

The following is a template for an automaton simulating the operations done on the stack, with ‘A’ being the element we are checking for repetition…



RS Stack Operation Visualization



RS and DS Interactions…

The following is a simple rule to demonstrate how the depth, and repetition stacks interact:

NUMBER = 1*1 (‘1’ - ‘9’)
NUMBER-LIST = 1 * 5 (NUMBER)

To test, we are given a source “5 7 1 2”...

Sequence:

1. ‘NUMBER-LIST’ rule is pushed onto Depth Stack
2. NUMBER-LIST expects a minimum of 1, and a maximum of 5 NUMBER, it pushes NUMBER onto the Repetition Stack
3. We hit the first character in the source, identify it as a NUMBER, and increment the repetition counter on the RS by 1
4. We identify that the minimum count expected by NUMBER-LIST is fulfilled, but there is still more input, so we continue the process three more times Until there is no more input.
5. We pop the NUMBER off the RS, as well as the NUMBER-LIST off the DS– as it is complete, and within the range NUMBER-LIST requires.

More Complicated Rules

For rules that require the repetition of child rules, the top of the depth stack keeps track of what scope we are at in the syntactic analysis process– because rules below the top cannot be accessed- if something is encountered that is not within the requirements of the current scope being evaluated– it will throw an error on unexpected output.

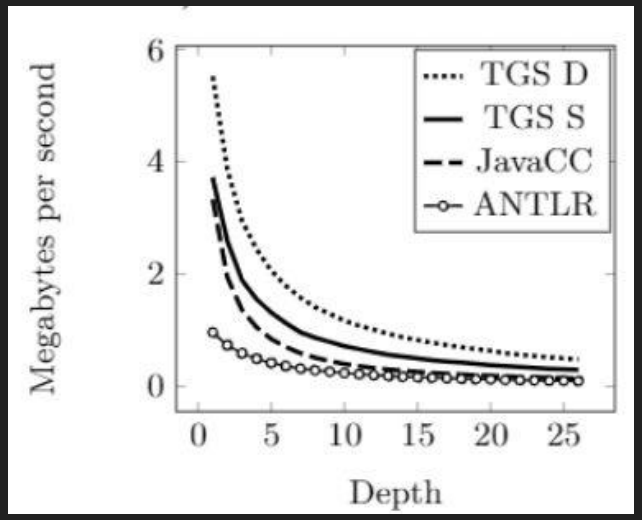
(I would show an example of such an instance, but I will not have the time to demonstrate it in full.)

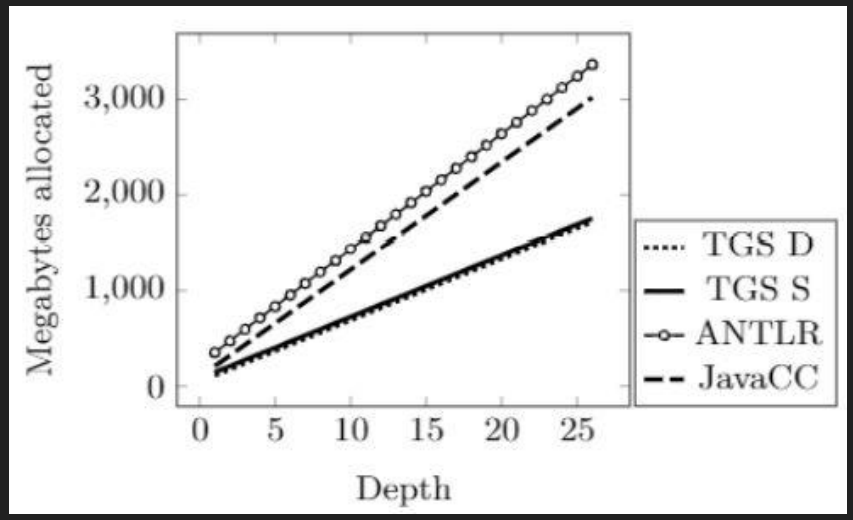
How Parallelization Can be Applied

When we encounter a particular rule with expected repetitions of a specific element, we can run multiple instances of our algorithm to recognize repeated elements on different lines of code, as they will operate using the same repetition stack– in the case that more than one repetition of a specific element is expected, this is when parallelism can be employed to speed up the process of syntactic analysis by scanning the element immediately after the current and incrementing the RS in the case that another instance of the element is found.

Gains to speed and efficiency given by the Tunnel Parsing Algorithm.

Through the use of the tunnel parsing algorithm in the construction of a dynamic (TGS D), and static (TGS S) Concrete Syntax Tree, we see a significant reduction of memory usage, and speed of parsing compared to ANTLR, and JavaCC, as is shown in the following graphs (Handzhiyski and Somova):





Conclusion:

We discussed improvements that can be made to both the processes of lexical analysis, and syntactic analysis. By leveraging parallelism against the task of tokenization, we were able to observe significant speedups in processing source code, and verifying the legality of typed identifiers through the use of multiple processing threads. By the usage of stacks to simulate scope, and tracking the repetition of grammar elements, we were able to convert the traditionally recursive process of parsing to an iterative one; by extension enabling us to leverage parallelism to produce favourable results in both the speed of parsing, and the memory usage of the tunnel parsing algorithm.

References

Barve, Amit, and Brijendra K. Joshi. “Improved Parallel Lexical Analysis Using OpenMP on Multi-Core Machines.” *Procedia Computer Science*, vol. 49, no. 1, 2015, pp. 211 - 219. 1877-0509.

Handzhiyski, Nikolay, and Elena Somova. “Tunnel Parsing with Counted Repetitions.” *Computer Science*, vol. 21, no. 4, 2020, pp. 441-462. 10.7494/csci.2020.21.4.3753.

Questions?