Lexical & Syntactic Analysis Improvements

Leveraging Grammar Rules, and Parallelization Against Compilation

CSCI 439 - Advanced Topics in Programming: Compilers
Dave Wessels
December 9th, 2022

Brandon Stanton

Introduction

Compilation Process

Compilation of a source code to executable code is a multiple phase process that converts through various checks and balances from a higher level target language to a target language, usually machine code, which is understood by a target computer architecture. The entire process of compilation typically comprises lexical analysis, syntax analysis, semantic analysis, intermediate code generation, machine independent code optimization, code generation, and machine dependent code optimization (Jha); though in this paper we'll only be covering the first two phases: lexical analysis, and syntactic analysis– for the sake of exploring each in sufficient detail.

Each phase relies on the previous stages, and given a correct, and valid input– will produce output for use for later stages. A given stage will utilize algorithms, and structures to achieve an end goal, and for most compilers used today they were designed and coded in the past which made use of computer architecture, and abstractions of the time.

Revisiting the Compilation Process

With the innovations of the past decade in regards to computer architecture, and an improved understanding of the inner workings of the compilation process, it behooves us to revisit the compilation process, and its various stages to improve, and optimize each stage individually. This will be very much a 'sampler' paper, for this reason, I will not be doing a deep dive into either optimization we'll be looking at, but will rather be presenting the obvious improvements given by each. I encourage the reader to refer to the citations given at the end of this research paper for further study.

In section 1, we start with lexical analysis, whereby instead of a sequential lexical analysis– we leverage multi-core processors given by present computer architecture to shift from iterative analysis to parallel lexical analysis. We will be examining a built-in tool to the GNU compiler, OpenMP, which makes use of the current innovations of multi-core, and multi-thread CPUs. Section 2 focuses on syntax analysis, typically we utilize a recursive method to check that a given token stream abides by production rules laid out by the grammar of the language; we will be exploring an algorithm called "tunnel parsing" that makes use of iteration, and produces a concrete syntax tree. We will also be looking at a suite of tools called Tunnel Grammar Studio, which makes use of the tunnel parsing algorithm, and will look at the adjustments we need to make to writing our grammars to make use of this powerful tool.

1) **Lexical Analysis**

Process & Purpose

A programming statement is a collection of identifiers divided by white space. At the beginning of the tokenizing process, an entire source code is run through a 'lexical analyzer' that breaks each individual statement into their singular terms, and each of these through analysis of the individual characters that make up the terms are then categorized into 'tokens'.

Tokens are the valid building blocks within the context of the programming language; grammar rules are composed of these sequences of tokens. This classification of statement terms is achieved through checking each character in the string that comprises the term, this is accomplished through computation constructs called 'automatons'. The required character sequence to be tokenized in one way could be very exact, as in the case of 'keywords', or loose as in the case of variables, or literals. Whilst this process is underway, the lexical analyzer inserts the legal terms into a 'symbol table' to pair their attribute type, value, scope, line of reference in the original source code. In addition to being used for constructing a symbol table, these tokens can likewise be inserted into a holder called a token stream for use for syntactical analysis, which is described later.

Lexical analysis is solely responsible for ensuring that the names of the identifiers (keywords, or other looser character sequences) are typed correctly, this process of identifying tokens can be done independently of other tokens in the source code . It is this characteristic of tokenization that will lead to lexically analyzing a given source code more efficiently in the context of parallelization

Improvement: Parallel Lexical Analysis

Why Should We Seek to Improve Lexical Analysis?

Some of the tools we utilize for lexical analysis were developed in the past, when computer architecture was not fully optimized, and utilized single core processing which was best suited for sequential processing. Today, multi-core architectures are more prevalent, and thus the compilation process, namely tokenization, and lexical analysis need to be revisited to make best use of these innovations (Farhanaaz and Sanju 258).

But in order to approach parallelism, we first need to identify which constructs within the lexical analyzer can be compartmentalized and run in tandem. For the purposes of simplicity, the process of lexical analysis can be divided into two distinct parts: scanning, and screening.

Scanning is responsible for breaking the source into white-space delimited strings, and screening makes use of the regular expressions laid out by the grammar of the language to tokenize these strings. A given source code can be divided into pieces equal to the number of available threads in a CPU, with one thread being dedicated to the purposes of pre-scanning.

With this division of labour, terms can be scanned in parallel, where the source code input stream is broken into chunks, and additional information on terms (line number, attribute type, etc.) is collected by the pre-scanner at the beginning and end of each term; and then the chunks are screened and added to the symbol table should they be valid.

OpenMP: Leveraging Multi-core Processors to Achieve Parallelism

The tool we'll be discussing to accomplish parallelism 'OpenMP', takes advantage of loops, and processor affinity to achieve parallelism. OpenMP is designed to support multi-platform shared-memory parallel programming in C/C++, and Fortran (GeeksforGeeks). To utilize this tool, OpenMP needs to be available in the compiler of your choosing and be permitted to be used by the system administrator through the compiler settings. OpenMP is available in the most recent build of the GNU Compiler Collection **GCC** by default. The following code can be written in a C/C++ or Fortran program to utilize OpenMP to perform the lexical analysis using parallel processing:

#pragma omp parallel

If able to be used, OpenMP identifies the number of threads available in a given machine, and by default utilizes all of them for execution, but the desired number of threads to be utilized can be specified by the user. Code blocks such as loops that take a considerable amount of time to process are prime candidates for the usage of multi-thread processing. But the usage of the tool falls to the discretion of the programmer, as not all code can be parallelized.

Speedup: How Significant is the Improvement Offered by Parallelism?

$$Speedup = \frac{\text{Sequential exectution time}}{\text{Parallel exection time}}$$

The above is the formula for determining the speed increase given by parallel processing, theoretically in this equation, "Sequential execution time" will be greater than, or equal to "Parallel execution time", resulting in a 'Speedup' value that is ≥ 1.0; excepting in cases where parallelism cannot be employed, or only a single thread is available on the given machine; whereby it will be equal to the traditional sequential method, or a speedup value of 1.0.

A huge benefit to the strategy of using this tool is that for larger programs with more constructs, parallel processing has the tendency to have compound improvements as CPU and thread-count increases. This experimental data (Barve and Joshi 216-217) refers to the time taken, and speedup in Parallel Lexical Analysis of a C Program with 10,000 constructs.

- TSEQ is the time taken using the sequential method, in milliseconds
- TMB is the time taken using OpenMP, in milliseconds
- SPMB is the Speedup value

No of CPUs/No. Of Threads	TSEQ	TMB	SPMB
2	1415.48	456.95	3.09
3	1415.48	312.41	4.53
4	1415.48	295.66	4.78
5	1415.48	229.59	6.16
6	1415.48	220.29	6.42
7	1415.48	206.87	6.84

Theoretically, should the number of available CPUs, and threads continue to increase as computer architecture sees further innovation– we will see further increases in the speedup of parallel lexical analysis. There are other methods of parallelization of lexical analysis not discussed in this paper, such as the round-robin method, but when speedup values are compared against those of OpenMP, their performance has been shown to be slower.

2) Syntax Analysis

Process & Purpose

Syntax Analysis utilizes context-free grammars **CFG** to check whether a sequence of tokens matches the rules of the grammar laid out by the language; this process is called ‘parsing’. Parsing is accomplished by utilizing the production rules of a given grammar and producing a match to the input sequence of tokens produced by the lexical analysis process. This either can be approached from a top-down, or bottom-up approach, where either we take the root production rule and expand into the sequence of tokens given as input, or we take the sequence of tokens and reduce it to the root production rule.

In either approach, it determines where to do the conversion by traversing through the tokens either from left to right, or right to left, and decides which production rule to apply on that token, or which production rule corresponds to the sequence of tokens. In either case, if no such combinations of rules can reproduce the token stream given as input, then the original source has syntax errors. If a successful reproduction is completed, then the production rules that led to this recreation are captured within a parse tree, which acts as a graphical representation of this successful derivation. The syntax analyzer outputs this parse tree for use in the next phase of compilation, Semantic Analysis, not discussed in this paper.

Through this entire process the syntax analyzer accomplishes three goals: parsing the source code, checking for errors, and constructing a parse tree as output as it runs through the production rules and relating it to the production of the token sequence. Syntax analysis will not take into account the correct spelling of identifiers, or the values assigned to those identifiers in the contexts of lexical, and semantic analysis.

Improvement: Tunnel Parsing, with Counted Repetition

Traditionally, the above described process of parsing is accomplished through recursion, this is due to the necessity of 'closure of scope' given by particular rules. However, through the algorithm that we'll discuss, we'll instead be using iteration through "Tunnel Parsing, with Counted Repetition" to accomplish all the goals laid out in the previous paragraph.

The chief differences between the traditional parsing algorithm, and tunnel parsing, is the usage of iteration, over recursion; and the generation of a concrete syntax tree **CST,** as opposed to an abstract syntax tree **AST**. Both of these differences are described in the next two sections.

Iteration vs. Recursion

Why we might want to use iteration over recursion comes down to the time complexity of parsing grammar rules, as well as the overhead of repeated function calls– recursion has both of these as drawbacks. As in the case of left recursive grammars that might call a particular rule multiple times, this has the potential to cause a stack overflow, which may cause a CPU crash. Iteration by comparison is safer, but the code to implement an iterative solution over a recursive one may be prone to more complexity, and length as will be made clear in later sections.

Concrete Syntax Trees vs. Abstract

Concrete syntax trees, and abstract syntax trees differ in a key regard: The latter focuses on simplifying, and representing the structures of the language for analysis; whereas the former is a 1:1 representation of the language, and all aspects of the syntax are represented in the CST. For a language that uses ';' as a termination token– this is not important to keep in a syntax tree to analyze a particular expression, because the tree is structured in such a way that having the termination token would be redundant. A CST would maintain the termination token, which will

be important when discussing how and why tunnel parsing utilizes repetition instead of recursion.

Repetition Stacks, and Stack Operations

Tunnel parsing utilizes a Repetition Stack **RS** to track the repetitions of a given element, where that can either be a token, or a rule– which is a sequence of tokens. There is a minimum, and maximum repetition range that can be specified in the grammar, where ***n*** is the variable attached to minimum repetition, and ***m*** is the maximum repetition. in the form of:

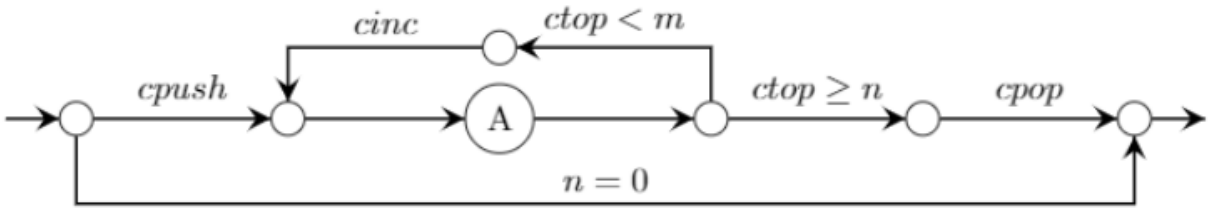
n*m Element

The default values of both n and m, if excluded, is 0, and infinity respectively. If use 'A' to denote the element in question, then an element that has a minimum repetition of zero, and a maximum repetition of one– then the grammar rule would be written as such:

0*1 A

The RS utilized by the tunnel parsing algorithm contains the information of how many times the specified element has been repeated in context with the stated/unstated minimum, and maximum range values. The usage of a stack data structure offers the advantage of simulating scope without the need to make use of recursion; the current scope is represented by the top element in the RS stack, and by pushing, and popping elements from the stack, the scope can either go down, or up by one layer.

The repetition stack, as is typical of the stack data abstraction– has a number of operations that serve the purpose of keeping track of repetitions of elements: ***cpush, cpop, ctop,*** and ***cinc***. The following is the template for building the repetition tracker automaton (Handzhiyski and Somova 447):



In the above diagram, 'A' represents the automaton for the repeatable element; For an n (min. repetition) equal to zero, the lower branch from the starting node can be traversed to exit the diagram. For an n ≥ 1: a 'cpush' places a counter on top of the repetition stack which tracks the number of repetitions of an element, at the point of push– this number is zero. The element is

checked through A, and then reaches the proceeding branch. ‘ctop’ represents the current count or repetitions on top of the stack. If ctop ≥ n, then it can proceed to ‘cpop’ the counter from the repetition stack, and exit the diagram. If further elements of A are present after observing the minimum requirement, it will take the topmost branch, ‘cinc’ will increment the repetition counter by one, and the process will proceed until it satisfies the necessary repetitions, or until no further instances of ‘A’ are present in the token stream.

Tunnel Parsing Algorithm

To implement the tunnel parsing algorithm we need to build a tunnel parsing machine, which requires the **design of the automata**, **extraction of the tunnels**, **construction of routers**, **preparation of segments**, **creation of a control layer**, and **parsing** (Handzhiyski and Somova 447). To give a full explanation, it would require extensive elaboration, but I will attempt to abbreviate, and describe each in a shortened format. For further information about any of these steps, I would encourage the reader to do further reading on Handzhiyski, and Somova’s work.

Design of the Automata, and Explanation of Segments

For the design of the automata, we need to create an automaton for each element in each rule of the grammar, of which will be used in the automaton described in the previous section in place of ‘A’. Each rule can make references to other rules (along with their minimum, and maximum repetition requirements), and the repetition stack will act as a history for the repetition of each element, with the most recent count of repetitions being stored at the top of the stack. The tracking of rules differs from the tracking of elements of a given rule; Rules are tracked utilizing Segments, which are class objects that correspond to each rule in the grammar, and until completed are stored in the Depth Stack (DS). The DS is a record of all currently active rules that are being processed by the tunnel parsing machine, and fulfill a similar role to recursion in that the rule being processed at the top of the DS denotes the current rule scope.

Extraction of the Tunnels

Each rule referenced by another will make use of the same repetition stack automaton, but in order to construct a concrete syntax tree, we will eventually need to extract the number of repetitions from the ‘tunnels’ corresponding to each element. This is facilitated by two additional stacks, the Depth Stack Archive (DSA), which tracks completed rules as they exit their processing scope, as well as the Repetition Stack Archive (RSA), which tracks the completed repetitions of elements. When an element from the repetition stack mentioned in the previous section reaches the ‘cpop’ transition, the element, along with its repetitions are moved to the RSA for later use, and likewise, if this completion of the repetition of an element should see a rule segment completed as well– that rule will be moved to the DSA. Both of these archive stacks will be used later in the construction of the CST.

Construction of Routers, and the Creation of a Control Layer

For error-recovery, as well as facilitating parsing, we need to be able to transition forward, and backwards through our automatons, but in order to do this, we need to have a list of each reachable state. These state transitions are collected utilizing a depth-first search and are stored and sorted in a static read-only table, this transition table is utilized along with segments to form the control layer of the tunnel parsing machine. The segments as they traverse along each state in the constructed automatons use the values in the transition table to facilitate parsing.

Parsing

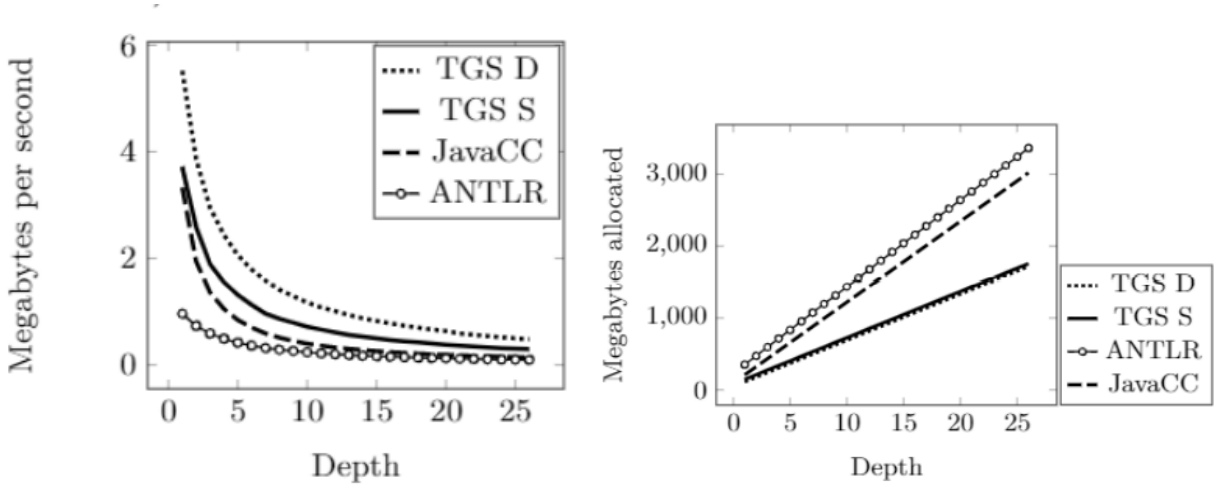
Through the process of moving backwards and forwards in the process of parsing, and utilizing the DS, RS, DSA, and RSA– the tunnel parsing machine is able to construct a concrete syntax tree for a given grammar and source code input. It constructs this concrete syntax tree by utilization of segments, which are object-oriented, and contain enough concrete information from extracting the information pertaining to repetition of elements from the tunnels to perform this task. The concrete syntax tree that is constructed has another advantage: should the user decide to utilize a different form of syntax tree (i.e. an abstract syntax tree) it is possible to convert the CST to an AST with relative ease.

Advantages of Utilizing Tunnel Parsing

Although the complexity of the algorithm of tunnel parsing is readily apparent, utilizing iteration over recursion enables us to take advantage of various optimizations that make the process of parsing much more efficient, and speedy.

One example of an optimization that can be utilized is parallelism. Segments which facilitate parsing can be strategically divided into parts, and be run by different program threads. How this works is that if a given rule requires the repetition of a given element, and there are multiple instances of that particular element, duplicate threads tracking the repetition of that element can make use of the same RS, and increment the stack independently for each thread.

The advantages to both the memory usage, and parsing speed of utilizing parallelism in tunnel parsing is illustrated through the below graphs, where ‘TGS’ denotes a tool that is discussed in the next section that makes use of the tunnel parsing algorithm, and the suffix ‘D’ refers to a dynamically-typed CST, and ‘S’ refers to a statically-typed CST. The performance is compared against ANother Tool for Language Recognition (ANTLR), and Java CC (Handzhiyski and Somova 457):



Tunnel Grammar Studio

While possible to code the aforementioned improvements by hand, there does exist a tool to facilitate tunnel parsing without the necessary effort. Tunnel Grammar Studio TGS (ExperaSoft) creates an object oriented parser, and for all intents and purposes is a parser generator (a compiler compiler). TGS also has the capability of running in a single, or multi-threaded capacity, up to 3 threads, which reduces the compile time on a given piece of source code, especially for longer inputs.

Tunnel Grammar Studio, while not specifically geared towards optimizing syntactic analysis, facilitates the construction of a capable parser that works for a given language. A caveat of using this tool is that the grammar rules of a given language need to be typed in a format that TGS can utilize to construct the language-specific parser. The entire suite offered by Tunnel Grammar Studio includes a decoder, a lexer, parser, optimizer, builder, and glue code as part of its suite of tools (GeeksforGeeks).

A simple example of a set of grammar rules that recognize text expressions, would be:

word = 1* ('A'-'Z' / 'a'-'z')
number = 1* '0'-'9'
text = 1* expression
expression = ({word} / {number}) *(0*1 "," " " ({word} / {number})) "."

The above example recognizes that an expression is a multiple word, or numbers, followed by a single comma, and whitespace, followed by 0 or more groups of words or numbers, ending with a period. Converting a grammar recognized by LEX to one recognized by TGS can be done with relative ease with simple languages such as this, but for more complex languages, it may require more time and effort.

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.

ExperaSoft. “Tunnel Grammar Studio.” *ExperaSoft*, 15 August 2022, https://www.experasoft.com/en/products/tgs/. Accessed 6 December 2022.

Farhanaaz, and Pillai V. Sanju. *An exploration on lexical analysis*. Conference Paper. Chennai, India, ICEEOT, 4 March 2016, pp. 253 - 258, 10.1109/ICEEOT.2016.7755127.

GeeksforGeeks. “OpenMP | Introduction with Installation Guide.” *GeeksforGeeks*, 23 April 2019, https://www.geeksforgeeks.org/openmp-introduction-with-installation-guide/. Accessed 6 December 2022.

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.

Jha, Rajesh K. “Phases of a Compiler.” *GeeksforGeeks*, GeeksforGeeks, 9 November 2021, https://www.geeksforgeeks.org/phases-of-a-compiler/. Accessed 6 December 2022.