Universiteit Leiden

Master Computer Science

Parallel Lexing, Parsing and Semantic Analysis on the GPU

Name:	R. F. Voetter
Student ID:	s1835130
Date:	30/06/2021
Specialisation:	Advanced Data Analytics
1st supervisor:	Dr. K. F. D. Rietveld
2nd supervisor:	Dr. A. Uta

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

In this thesis, we describe the design and implementation of the front end of a compiler of which all major stages can be executed on a Graphical Processing Unit (GPU). The lexical-, syntactic- and semantic analysis stages are implemented in terms of data-parallel primitives provided by the Futhark programming language. A series of experiments show that our implementation scales well for large source input, but suffers from the overhead of scheduling and distribution of work for smaller input.

Contents

1. Introduction

Compilation is the process of translating the textual representation of source code into machine code. This process is traditionally implemented as a series of single-threaded processes. As the size of code bases continues to grow, the need for faster compilers arises. In the recent years, though, single-core performance has reached its physical limits, and this requires any application that wishes to scale up to modern computing demands to take advantage of the parallel architectures the industry is moving to. Typically, compilation is parallelized on a coarse level, for example by processing different files of the code base simultaneously on different cores of a multi-core processor.

Besides the Central Processing Unit (CPU), many modern computer setups also feature a Graphics Processing Unit (GPU), either as a dedicated component or integrated into the CPU chip. As the name suggests, graphical processing units originated to accelerate operations required for graphical applications. These display data-parallel characteristics, and GPUs accelerate this by providing specialized functionality that performs the same operation on large groups of values at once. In time, the fixed function graphics functionality provided by this hardware made way for more programmable stages, and now serves as a general-purpose accelerator for many data-parallel applications. This provides interesting performance opportunities for any problem which can state itself as such, even more than modern-day multi-core processors offer.

This provides an interesting research opportunity. If the traditional stages of a compiler can be expressed in terms of data-parallel operations that can be performed on a GPU, the compilation time of large projects could be improved drastically. These stages are typically implemented as recursive processes over irregular tree structures, which are not friendly to GPU hardware, and so require a major redesign.

In this thesis, we concern ourselves specifically with the *front end* of the compilation process. This part includes transforming the plain source code text into a structure the compiler can reason with, verifying that the program adheres to the syntactic and semantic rules of the language, and computing additional information for program structures. The *back end*, described by Huijben [Hui21], concerns itself with lowering the compiler representation of the program into machine code, so that it may be executed on real hardware.

1.1. Goals

The main goal of this project is to design and implement a compiler of which all major processing happens on a Graphics Processing Unit (GPU). To allow us the creative freedom

to solve problems by changing the programming language, we do not limit ourselves to any existing language, but design our own instead. The scope of this language is relatively limited, but should still bear resemblance to existing programming languages, so that the operations performed in the compiler also relate to existing compilers. In specific, the goal is to obtain a compiler for an imperative, procedural programming language with a static type system, not unlike a very simple version of the C language. Furthermore, the compiler should directly output machine code, and not some form of intermediate language which is interpreted.

1.2. Contributions

In this project, we provide the following scientific contributions:

- The design and implementation of a novel algorithm to perform lexical analysis on GPUs.
- The implementation of an algorithm to perform parallel parsing [VM07]. As far as known to the authors, this is at the time of writing not only the first implementation of this algorithm for GPUs, but also the first implementation in general.
- The parsing algorithm suffers from a number of inherent limitations, and so we also propose and implement a number of methods that provide workarounds. These are consist of a combination of grammar modifications and additional processing, the latter of which run in parallel on the GPU.
- The design and implementation of a parallel semantic analysis pipeline, designed to be executed on a GPU. This includes variable and function resolution, type analysis, and convergence analysis.
- Practical implementations for many sub-problems related to compilation and working with trees on data-parallel architectures. This includes implementations for parallel bracket matching, previous smaller or equal value, tree construction, and parallel boolean expression evaluation.
- Source code of the compiler and experiments are made open-source, and are available at `https://github.com/Snektron/pareas` and `https://github.com/Snektron/pareas-frontend-benchmarks`.

The remainder of this thesis is organized as follows. In Chapter 2, some background information related to our topic will be discussed. Chapter 3 discusses the high-level design of the front end part of the compiler, and Chapter 4 goes further into detail about its implementation. To get an indication of the runtime characteristics of the techniques discussed in this thesis, we describe a series of experiments in Chapter 5. Finally, Chapter 6 concludes the work and discusses the direction further research on the topic of GPU-accelerated compilation may take.

2. Background

In this chapter, we will discuss some of the fundamentals on which our research builds. In Section 2.1, we will briefly discuss the state of graphics acceleration hardware. Section 2.2 outlines some theoretical parallel programming principles, and also discusses some of the primitive parallel algorithms on which our work is built. In Section 2.3, we discuss Futhark, the programming language in which much of our implementation is written. Finally, Section 2.4 discusses some of the existing work which is relevant to the topic of parallel compilation.

2.1. Graphic Programming Units

As the name suggests, Graphics Programming Units (GPUs) are specialized hardware originally dedicated to graphics operations. Initially, these devices consisted of fixed function logic specifically for rendering, for example, by providing hardware-accelerated functionality to draw and shade lists of triangles. Note that this process has a data-parallel nature: typically, the same shading computation needs to be performed for every triangle and for every pixel on the screen but with other inputs, and the result of one pixel does not generally depend on the result of another. The simplest way to accelerate this rendering process was to replicate the rendering hardware multiple times, and perform the computations for each triangle or pixel simultaneously. As rendering tasks became more complex, fixed function logic made way for fully programmable shader stages, still running in parallel. This advancement also allowed one to use the parallel processing capabilities that their GPU provided to perform non-graphics related data-parallel computation, marking the advent of general-purpose GPU (GPGPU) programming.

On the hardware level, a typical modern GPU is split up into a number of *compute units*. Each of these compute units represents an independent SIMD processor, which typically operates on vectors of 32 or 64 elements at a time. Operations are then performed element-wise on multiple vectors, and operations between elements of the same vector are often specialized or less efficient. GPUs are often programmed from the perspective of a single element. That is, the programmer does not have explicit knowledge of the underlying SIMD architecture, and the program is written as if it may be executed on a single-thread machine. In practice, this means that many instances of the same program are executed in parallel on the same compute unit, by performing every operation in lock-step. This type of architecture reduces the amount of hardware circuitry needed to perform many of the same operations a large amount of data, but comes at the cost that certain types of programs cannot be efficiently executed. For example, GPUs respond poorly to heavily branching programs. Unless all instances take the same branch, when a branch such as an if/else statement is encountered both cases must be evaluated, which wastes computing power.

Note that GPU manufacturers typically measure the amount of *shader cores* instead of compute units, each of which refers to an individual vector index within a compute unit. Modern high-end GPUs typically have in the order of thousands of shader cores.

When compared to a regular processor, GPU architecture also differs in the way that global memory is handled. Main memory is much slower than local memory on both CPUs and GPUs. The working set of regular programs is typically quite small, and so processors deal with memory latency by introducing multiple levels of CPU-local caches. In contrast, a GPU needs to process a large amount of data quickly, and so data is typically only fetched once from main memory. In this scenario caches are less efficient, and are also much more complicated to implement for parallel architectures. Instead, the latency of main memory is compensated by focusing on memory bandwidth instead, and so GPUs often feature memory bandwidths an order of magnitude higher than CPU memory bandwidth.

2.2. Parallel Programming

Much literature concerning theoretical parallel programming considers the Parallel Random-Access Memory (PRAM) machine [Wyl79]. This is a theoretical model of computation, where an arbitrary number of processors and memory is available. Memory is uniformly and instantaneously accessible to all processors, and there is no resource contention. There are three general variants of PRAM machine, with different properties regarding parallel access of memory cells:

- Exclusive Read, Exclusive Write (EREW) machines. This is the weakest type of PRAM model, where only a single processor may access a certain memory location at a time.

- Concurrent Read, Exclusive Write (CREW) machines. In this case, any amount of processors may read from a certain memory cell simultaneously, but only one processor may write.

- Concurrent Read, Concurrent Write (CRCW) machines. This is the strongest PRAM machine, and also allows processors to write to the same memory cell simultaneously. In this case, write conflicts are handled through different strategies. For example, it may simply be invalid for multiple processors to simultaneously write different values to a memory cell. In other cases, the resulting value may be determined by selecting any of the processors attempting the write. The resulting value could also be could also be determined by reduction, for example, by summing the values of writing processors.

PRAM machines serve to help reason with the theoretical runtime of parallel algorithms, which are measured in two ways: through *sequential runtime complexity* and *parallel runtime complexity*. The former case measures the asymptotic time complexity of the algorithm in terms of the total work, that is, the time complexity of the algorithm if it were executed on a single-core machine. For example, the total runtime complexity of pairwise addition of two arrays of n values is $\mathcal{O}(n)$, as n additions have to be performed in total. The latter case measures the asymptotic time complexity of the algorithm in terms of

critical chain of computations, that is, the total runtime of the algorithm given infinite processors. In the previous example, if we employ n processors which each compute the result of a different pair of elements, we only need a $\mathcal{O}(1)$ time in total.

While PRAM machines are only a theoretical model, many of the algorithms map efficiently to real hardware by simulating processors. For example, on a multi-core machine with 8 cores, we can assign each core a subsequence of approximately $\frac{n}{8}$ elements. Each core then computes the result of its assigned subsequence. Furthermore, many PRAM algorithms perform the same operations on large parts of the input, which means that many algorithms discussed in PRAM literature map well to GPU architectures. In the following subsections, we discuss a number of parallel algorithms which are fundamental to the implementation of our compiler.

2.2.1. Parallel Map

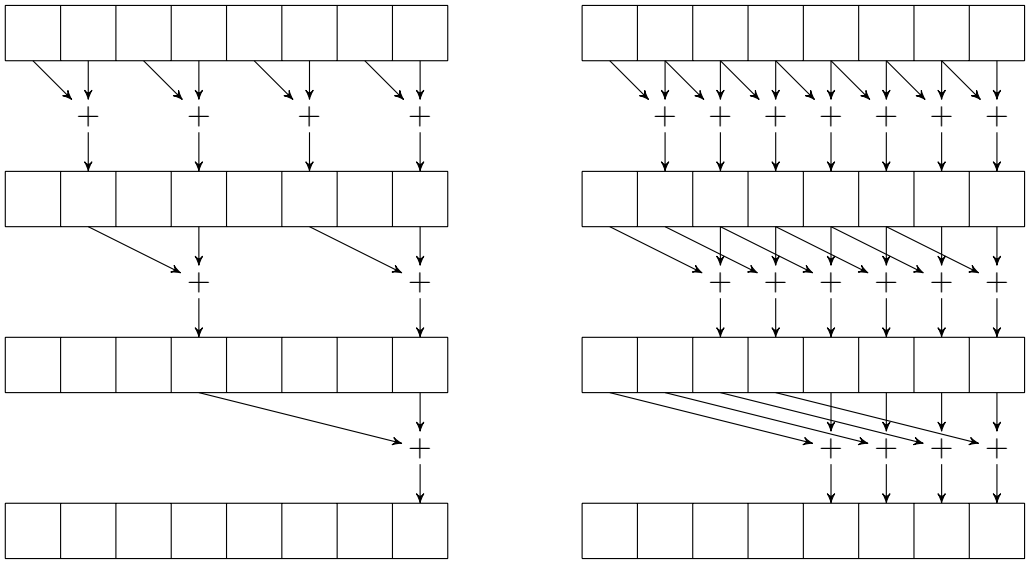
One of the simplest parallel algorithms involves applying a function f pairwise to every element of a number of arrays. This is the generalized version of the example discussed in the previous section, and is commonly referred to as a *map* operations. For arrays of n elements, the total sequential runtime is $\mathcal{O}(fn)$, and the total parallel runtime is $\mathcal{O}(f)$.

2.2.2. Parallel Reduction

A more interesting parallel algorithm is the parallel *reduction*. This concerns applying a binary operator f to accumulate an array of values into a single value. For example, the result of this computation for an array $a = \{a_1, a_2, ..., a_n\}$ can be expressed as $f(...f(f(a_1, a_2), a_3)..., a_n)$. As Hillis & Steele [HS86] show, if the binary operator f is associative, we can reorder the operations in such a way that many of them can be performed in parallel. For example, to compute the result of $((a + b) + c) + d$ in parallel, the computation is first rewritten as $(a + b) + (c + d)$, after which the terms $(a + b)$ and $(b + c)$ can be computed simultaneously. This scales to any associative binary operator, and any amount of values. Figure 2.1a shows an example of the data-flow during the reduction of an array of 8 elements, where each box represents an array element. Notice how the operation in each level are independent from each other, and so can be computed in parallel. To reduce an array of n elements using associative operator f in parallel using this strategy, a total of $\mathcal{O}(fn)$ operations are required, which can be performed in parallel in $\mathcal{O}(f \lg n)$ time.

2.2.3. Parallel Scan

Computing the scan or prefix sum of an array is an operation similar to the reduction, except that in this case a reduction is performed for all prefixes of the array. That is, the prefix sum of the array $a = \{a_1, a_2, ..., a_n\}$ yields $\{a_1, a_1 + a_2, ..., a_1 + a_2 + ... + a_n\}$. Again shown by Hillis & Steele [HS86], this algorithm may also be parallelized, by computing many reductions at the same time. By cleverly re-using the intermediary results, the same



(a) Parallel reduction. Notice how the computations in every level can be performed in parallel.

(b) Parallel sum. Again, the computations in every level can be performed in parallel.

Figure 2.1. Parallel reductions and prefix sums.

asymptotic runtime bounds as the parallel reduction are obtained. Figure 2.1b illustrates the data flow during a parallel prefix sum on an array of 8 elements. Again, the operations shown on each level can be computed simultaneously. Also note that this operation can be performed in-place, and so no extra memory is required.

We recognize two versions of the parallel prefix sum: the *inclusive* and *exclusive* variants. In the former version, the value at some index in the resulting array includes the element originally at that index, while it does not in the latter version. In mathematical terms, the inclusive prefix sum over some array a computes for an element at index i the result of $\sum_1^i a_i$, while the exclusive prefix sum computes $\sum_1^{i-1} a_i$. Similar to the parallel reduction, the parallel prefix sum works with any associative operator f. This operation is referred to as a prefix sum when the addition operator is used, and as a prefix scan or scan when any other operator is used. When an exclusive scan is computed with a custom operator, an extra element is to be supplied to serve as the initial value of the computation.

2.2.4. Parallel Scatters & Gathers

A theoretically less interesting but practically quite useful set of operators is formed by the parallel scatter and gather operations. The parallel scatter writes an array of values in a destination array, where the destination of each element is given by an index array. Operations which are out of bounds are ignored, and if two elements are to be written at the same location in the destination array the result is undefined. The gather operation is the converse of the scatter operation, and fetches elements from one array according to indices in another. Both of these operations are performed over each element in parallel, yielding $\mathcal{O}(n)$ and $\mathcal{O}(1)$ time for index arrays of n elements.

2.3. Futhark

Futhark [Hen+17] is a high-level, functional programming language designed to make GPU programming easier. Rather than having to manually write and dispatch kernels, as is the case with GPGPU programming frameworks such as CUDA and OpenCL, Futhark programs are made by composing a number of parallel primitives which are then compiled into separate kernels and dispatching code. These include for example the parallel map, reduction, prefix scan, scatter and gather operations discussed in the previous section. This saves the programmer writing specialized implementations for these primitives, and means that GPU programs can be prototyped much faster. Of course, this could also be provided as a simple library abstracting over CUDA or OpenCL, but the major advantage of Futhark is that the compiler can optimize the complete program on a global level. For example, when a parallel map operations are performed over similarly sized data, they can be placed in the same kernel and executed simultaneously, which eliminates the non-negligible overhead of launching an additional kernel and improves overall locality of data.

Futhark code can be compiled to one of multiple back ends: the primary and most of interest are the GPU-capable back ends, CUDA and OpenCL. Additionally, single- and multi-core CPU back ends are also supported, which ensures that Futhark programs are portable to many machines. Futhark programs do not support any I/O operations: input is to be provided and output is obtained by invoking one of the *entry points* of the program. This can be done in two main ways: either via the integrated standalone system, where binary or textual input is provided via the standard input and the result of the computation printed to standard output, or with the Python or C API, where input and output is communicated as numpy or native arrays respectively.

2.4. Related work

There is a large amount of related work on the topics of parallel lexical analysis and parallel parsing, but less on the other stages of compilation. Much of this work was performed in the second half of the previous century, but research died out when single-core performance scaled well. With single-core performance at its physical limits, there is a renewed interest in many areas of parallel programming, including lexical analysis and parsing.

One of the most fundamental pieces of related work to our topic is the work of Hillis & Steele [HS86]. In this work, the authors discuss a number of parallel algorithms, which are initially designed to run on their parallel Connection Machine [Hil89]. Besides the parallel reduction and prefix sum algorithms, as discussed in Section 2.2, this work also discusses a number of other algorithms which are fundamental to our research. For example, Hillis & Steele show how the parallel prefix sum can be applied Deterministic Finite-State Automatons (DFAs) to parse a regular language in parallel. While this early variant incurs a large memory overhead, it forms the basis of the lexical analysis algorithm proposed in our work, and is further explained in Section 3.1.2. Furthermore, a number of parallel algorithms involving linked lists are discussed, which include finding the end of a linked list in parallel, performing parallel reductions and scans on linked lists, and pairing up the

elements of two linked lists. In Section 4.7, we show that these algorithms translate to tree operations, and are invaluable to our implementation.

One of the earliest parallel parsing algorithms for a more complicated language class, LL languages, is discussed by Skillicorn & Barnard [SB89]. This parser is in fact designed to be executed on the Connection Machine. In their work the authors claim that any $LL(1)$ grammar, which forms a practical set of languages in which the grammar of many common program languages can be expressed, can be parsed in logarithmic time. This claim was later shown to be invalid by Hill [Hil92], who discusses a practical implementation of the parallel LL parsing algorithm presented by Skillicorn & Barnard, and who also applies the parallel regular language parsing algorithm by Hillis & Steele to perform lexical analysis.

More recent work on parallel parsing algorithms within the class of LL languages is the work by Vagner & Melichar [VM07]. The parsing method outlined in this method is in many ways similar to the algorithm presented by Skillicorn & Barnard, and runs in logarithmic parallel time. In this work, the authors first show that a nondeterministic version of the LL parsing algorithm can be performed in parallel. This is done by dividing the input in segments, which are each parsed by an individual processor. When such a segment is parsed by a processor, the initial state of the parser is unknown. This is handled in a nondeterministic way by iterating over all possible states of the parser, and resolving the actual begin state when the intermediary results of each processor is merged. A deterministic version of the algorithm is produced by limiting the amount of states that the parser can be in at the start of a particular segment to one, which results in the $LLP(q, k)$ grammar class. To help determine the initial state, the parser may look at both the next k symbols on the input, as well as the previous q. This parsing algorithm forms one of the key parts in our work, and is further explained in Section 3.2.

There is also research on paralleling other parsing algorithms and language classes. For example, Johnson [Joh11] presents a parallel implementation of the Cocke-Younger-Kasami (CYK) parsing algorithm that is accelerated using both multi-core CPUs and GPUs. This algorithm also parses context-free languages, similar to the previously discussed parsing algorithms. Unfortunately, the CYK parsing algorithm suffers from a cubic worst case complexity. The authors show that the both the GPU and CPU implementation of this algorithm outperform the baseline implementation, though the CPU version outperforms the GPU version as well.

More recent work on parallel parsing includes the PAPAGENO [Bar+15] tool chain. This work builds on Operator-Precedence Grammars (OPG), which exhibit *local parsability*. Similarly to the algorithms by Skillicon & Barnard and Vagner & Melichar, this algorithm works by splitting up the input into segments and parsing each of the chunks in parallel. The final result is then computed using a parallel reduction. In contrast to the other works, this type of parser works in a bottom-up manner rather than top-down. The authors also propose a method of parallel lexical analysis. This method works similar to the non-deterministic parsing method proposed by Vagner & Melichar, where all the possible starting states for the lexical analyzer are simply enumerated. The implementation is evaluated on different multi-core processors, and the authors show good performance versus the traditional parser- and lexical analyzer generators Bison and Flex.

Many lexical analyzers are implemented by means of a deterministic finite-state automa-

ton. Mytkowicz et al. [MMS14] implement the algorithm discussed by Hillis & Steele to evaluate a DFA in parallel, with a method not unlike that presented by the authors of the PAPAGENO tool chain. Each processor is assigned some subsection of the input, and all starting states for each of those sections are examined. Performance is gained by cleverly enumerating these starting states, and the authors show good performance on modern SIMD hardware.

Sin'ya et al. [SMS13] show an approach to the algorithm by Hillis & Steele which reduces the memory overhead significantly. This method is also fully deterministic, and runs in logarithmic time. Parallelization is obtained by constructing a *Simultaneous Finite Automaton* (SFA) from a DFA. The input is again divided over segments, each of which simulate the SFA. The results can be combined using a parallel reduction. This method, in fact, turns out to be equivalent to our method of lexical analysis, though the authors only show its use for matching regular expressions on multi-core CPU hardware. We offer a different explanation of the underlying concept, and extend the method to lexical analyzers and GPU acceleration.

Skillicorn & Barnard [SB93] also discuss the general topic of compiling in parallel. This work mostly describes the state of parallel lexical and syntactical analysis at the time. Semantic analysis is only mentioned a few time, but it is suggested that this is to be parallelized by simultaneously handling compilation units or functions.

The dissertation of Hsu [Hsu19] describes the implementation of an APL compiler, written in APL itself, which is completely hosted on the GPU[1]. The work does not go much into detail about what methods are used for lexical and syntactical analysis, but does describe a range of semantic analysis passes which verify the semantic correctness of the program and lowers the parse tree into an intermediate format. Note that this compiler does not output machine code, but performs source translation of APL into C++. The author shows that when input is large, the GPU version of the compiler is more performance efficient than the CPU versions. Many of the passes and data structures used in this compiler are relevant to our own work.

Other recent work involving parallel compilation is the Parallel GCC project [Con19], whose aim is to introduce parallel options in the GNU GCC C compiler[2]. In this work, the compiler is parallelized by distributing function-local optimization work over the available processors. While the authors measure a performance increase of more than 2.5 times when 4 cores are used, they also note that this optimization process is only a small part of the total compilation process, over which a performance gain of only 10% less time is obtained.

[1] `https://github.com/Co-dfns/Co-dfns`
[2] `https://gcc.gnu.org/`

3. Design

Architecturally, the design of the compiler front end follows the classical compiler architecture. The source code undergoes a series of transformation passes, each of which translate the program from one intermediary format into the next. The compiler performs a single pass at a time. There is no inter-pass parallelism, and as a result, the main challenge is that single passes are to be highly parallelized in such a way that they can sufficiently use all GPU resources. On a high level, the compilation pipeline of the compiler front end can be divided into a number of different stages, each consisting of multiple passes:

- First, the source code is partitioned into a sequence of *tokens*, through a process known as *lexical analysis*. This serves to make the input more easily digestible for the subsequent stages.

- Second, the newly generated token sequence is parsed according to the language grammar. This serves to both validate the input according to the syntactic rules of the language and to produce a *parse tree*, which represents the program in a more structured and hierarchical manner suitable for further processing.

- In the final stage *semantic analysis* is performed, where a number of passes are applied to derive and verify the semantic structure of the program, ensuring that it satisfies the language rules. These passes also produce auxiliary information required by the compiler back end, such as type, symbol, and literal information. The parse tree is also re-structured to represent the program into a more useful manner, and is combined with the auxiliary information to form an attributed *abstract syntax tree* (AST).

Figure 3.1 shows a schematic overview of the compilation process of the compiler front end. Boxes represent representations of the program, and arrows represent operations and transformations applied to these representations. After the front end passes, the AST is consumed by the back end part of the compiler which transforms the AST into machine code.

The remainder of this chapter elaborates on the design of each of these stages and is structured as follows: Section 3.1 discusses the design of the lexical analysis stage. In

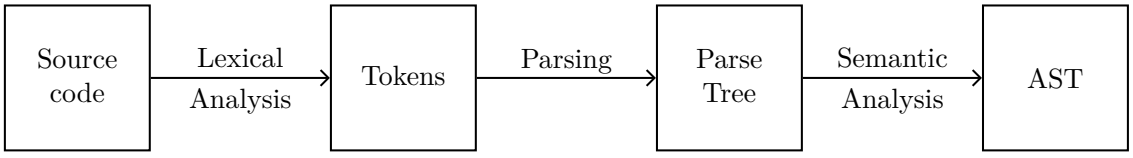


Figure 3.1. Schematic overview of the compiler front end pipeline.

Section 3.2 the design of the parsing stage is laid out, and in Section 3.3 semantic analysis is discussed.

3.1. Lexical Analysis

In the first stage of the compiler, *lexical analysis* is performed. The aim is to partition the input into a sequence of *tokens*. Each of those tokens is a tuple of two values: the *lexeme*, a subspan of the input which corresponds to the token, and a *token type*, an enumerant which identifies the token. Lexical analysis is performed according to a *lexical grammar*, which associates each token with a *pattern*. Lexical analysis is then performed from left-to-right, by repeatedly applying all patterns to the yet-to-be analyzed part of the input. The next token is then determined by whichever pattern produced the longest valid match. If multiple patterns produce a valid match it is disambiguated by some arbitrary rule, for example by assigning priorities to tokens or by picking whichever token matched first. If none of the patterns matched at all an error is yielded, either by producing a special token with an error type or by breaking off the lexical analysis process. Additional tasks of a typical lexical analyzer might be to also filter out certain types of tokens, for example comments and whitespace, or to convert lexemes into a semantic value, such as converting integer lexemes to integer values. In the remainder of this section, we only consider breaking up the input into tokens consisting of a type and a lexeme.

As an example, consider the following lexical grammar:

$$\begin{aligned} \textbf{\textit{int}} &\rightarrow \texttt{int} \\ \textbf{\textit{name}} &\rightarrow \texttt{[a-zA-Z\_]+} \\ \textbf{\textit{number}} &\rightarrow \texttt{[0-9]+} \\ \textbf{\textit{plus}} &\rightarrow \texttt{\textbackslash+} \\ \textbf{\textit{equals}} &\rightarrow \texttt{=} \\ \textbf{\textit{whitespace}} &\rightarrow \texttt{␣+} \end{aligned} \tag{3.1}$$

Patterns here are defined as *regular expressions*. When applied to an example input such as `int cobra=1+python`, the token sequence $\langle \textbf{\textit{int}}, \texttt{int}\rangle$, $\langle \textbf{\textit{whitespace}}, \texttt{␣}\rangle$,$\langle \textbf{\textit{name}}, \texttt{cobra}\rangle$, $\langle \textbf{\textit{equals}}, =\rangle$, $\langle \textbf{\textit{number}}, \texttt{1}\rangle$, $\langle \textbf{\textit{plus}}, +\rangle$, $\langle \textbf{\textit{name}}, \texttt{python}\rangle$ is produced. Note that `int` is matched by two patterns; the pattern for ***int*** and the pattern for ***name***. In this case, we are interested in the more specialized pattern, which could be a reserved identifier in our language, for example.

The purpose of this lexical analysis stage is to ease the implementation of the subsequent stages, parsing and semantic analysis. In a typical compiler lexical analysis can be implemented in a simpler and more efficient manner than the subsequent parsing phase. In turn, the parser can now be implemented in terms of these token types instead of raw characters, which simplifies its design and reduces the amount of work that it needs to do in general. This is no different for our compiler, and in fact, the lexical analysis phase alleviates some of the inherent limitations of the specific parsing algorithm used in the parsing stage of the compiler.

At a first glance, paralleling lexical analysis appears a relatively simple task: tokens are often small and local, and easy to distinguish from another. One approach, for example,

is to simply divide the input string equally over all available processors. Each processor then starts by searching for the next nearest boundary between two tokens, and then runs a single-threaded lexical analysis algorithm which is then performed for all chunks of the string in parallel. This works fine for simple lexical grammars such as Grammar 3.1, but when we introduce more interesting tokens this scheme becomes inadequate. Consider for example Grammar 3.1 again, but now augmented with ***string*** and ***comment*** tokens:

$$\begin{aligned}
\textbf{\textit{int}} &\rightarrow \texttt{int} \\
\textbf{\textit{name}} &\rightarrow \texttt{[a-zA-Z_]+} \\
\textbf{\textit{number}} &\rightarrow \texttt{[0-9]+} \\
\textbf{\textit{plus}} &\rightarrow \texttt{\textbackslash+} \\
\textbf{\textit{equals}} &\rightarrow \texttt{=} \\
\textbf{\textit{whitespace}} &\rightarrow \texttt{␣+} \\
\textbf{\textit{string}} &\rightarrow \texttt{"([\^{}\textbackslash"]|\textbackslash\textbackslash|\textbackslash")+"} \\
\textbf{\textit{comment}} &\rightarrow \texttt{//[\^{}\textbackslash n]\textbackslash n}
\end{aligned} \tag{3.2}$$

Any text after the start of a ***comment*** until the next newline character is included in the token, and any text between two quotes is included in ***string*** tokens. Note that in order for a user to also insert a quote in a string, programming languages often allow *escaping* quotes, in which case they will not end the string. These two tokens generate problems for the naive lexical analysis scheme previously outlined: When a processor starts matching, it will need to look back an arbitrary amount of characters in the input to tell whether the current character is in a string or comment. In fact, it will need to look back all the way to the start of the input to determine the current state, which is of course detrimental for parallel performance.

Our lexical analysis algorithm works by first representing the entire lexical grammar as a single deterministic finite automaton, and then evaluating that on the input in parallel by processing each character simultaneously. While this does not negate that the lexical analysis of a particular subsection of the input is dependent on the state the lexical analyzer is in when it entered the section, we can still sufficiently parallelize it, as we will see in Section 3.1.2. The implementation of our lexical analyzer is split up into two key parts: In the first part, an offline phase transforms the lexical grammar into a number of data structures representing this automaton in a GPU-friendly manner. These are then loaded onto the GPU in the second part, where the actual lexical analysis is performed. Note that we need to build the relevant data structures only once for a particular lexical grammar, so we can ignore any runtime cost associated to the construction process.

3.1.1. The Lexical Analysis Automaton

To perform the analysis, a lexical analyzer is required to evaluate all patterns of the lexical grammar against the start of the remaining input. Instead of simply iterating over each pattern and attempting a match, a typical lexical analyzer contains a preprocessing phase where all patterns are compiled into a more efficient finite automaton, either deterministic or non-deterministic. The automaton is then simulated with the remaining input until a transition is detected from a valid state to the reject state, at which point the generated token is determined by the source state of this transition. If this state was not an accept state, the input is not valid according to the lexical grammar, and an error is emitted.

Otherwise, if the source state is an accept state, the lexical analyzer emits the token type associated to said accept state, and the corresponding lexeme is obtained simply by counting the number of transitions that the state machine performed while matching the token. The automaton is then reset to the start state, and is simulated again to determine the next token. This process is repeated until all of the input has been consumed or until an error occurred. The state machine representing our lexical analyzer works very similar to this process, with one distinct difference: instead of simulating the state machine until a transition to the reject state occurs and then resetting, we build a deterministic finite state machine which is simulated only once for the entire input, and produces tokens along the way. To this end, the classical definition of a deterministic finite automaton is augmented by associating every accepting state with a pattern of the lexical grammar. The idea is that once the automaton passes this state, a token corresponding to the pattern is emitted. Note that we can only know that token should be emitted when examining the next symbol of the input. For example, for the lexical grammar in Example 3.1, if the automaton has processed the symbols `int` and the next symbol is `=`, an ***int*** token should be emitted. When the next symbol is `a`, however, the automaton should continue to match the current token instead. For this reason, a subset of transitions outgoing from accepting states are marked to actually emit a token. For the example the transition reading `*` would be marked to emit a token, whereas the transition reading `a` would not. The lexeme of an emitted token is obtained in a similar way as the classical automaton, simply by counting the number of transitions which were performed since the previous token was emitted, or since the state machine started. This new type of finite state machine allows a simpler implementation of the lexical analyzer, and is more suited to GPU-assisted parallelization.

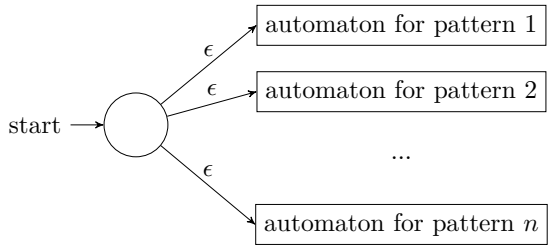
The formal definition of our new type of automaton is close to that of the classical deterministic finite automaton, and is as follows:

$$M = (Q, \Sigma, T, \delta, q_0, q_r, Z, F) \tag{3.3}$$

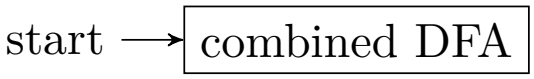
where:

- Q is a finite set of *states*.
- Σ is the *alphabet* of the automaton, a finite set of symbols over which the automaton operates.
- T is a finite set of token types that the lexer may produce.
- $\delta : Q \times \Sigma \to Q \times \{0, 1\}$ is the *transition function*. This function produces the next state, as well as a value indicating whether a token is generated.
- $q_r \in Q$ is the *reject state* of the automaton.
- $F : Q \to T \cup \{\varepsilon\}$ is a function mapping accepting states to token types.

To simulate such an automaton, the current state is first initialized with the start state, q_0. Iteratively, a symbol is read from the input and subsequently discarded. The current state and this input symbol are then applied to δ, which yields two values: The next state of the simulation, and a value indicating whether a token is produced. The type of this token is obtained by passing the old state to F. It is valid for a state to not be associated with any token, in which case F yields ε. Note that if a particular application of δ produces $(q, 1)$ for a $q \in Q$, indicating that a token should be generated, it is invalid for $F(q)$ to yield ε. However, using just the second result of δ to determine when to emit a token results in a boundary condition: The automaton must perform a transition in order to emit a token, as we can only be certain that a token has been fully matched whenever the next symbol



(a) Template nondeterministic finite automaton for combining multiple patterns of a lexical grammar.



(b) The same template compiled into a combined deterministic finite automaton.

is read and determined not to be part of the previous token. If the last symbol has been read, however, and there is no more input available, the lexical analyzer must still emit the final token. In this situation, F is used to obtain the type of the final token. When it is invalid for a state to be the final state of an automaton's simulation, F yields ε. This property is re-used to yield errors during lexical analysis: Whenever an invalid symbol is detected, the automaton moves to the reject state q_r. All transitions from this reject state lead back to itself, and the final error is produced by fixing $F(q_r) = \varepsilon$.

Constructing a lexical analyzer according to Definition 3.3 happens in two steps: first, a deterministic finite automaton is produced which can produce one token according to the lexical grammar, and so is more like the automaton used in a classical lexical analyzer. This is done by first translating each pattern of the lexical grammar into its corresponding nondeterministic finite automaton by using Thompson's construction [Aho+06, Chapter 3.7.4]. Each of these automatons have a single accepting state, which is marked with the token type corresponding to the pattern which produced it. To combine the individual automatons for each pattern into a single automaton which can analyze a single token, they are connected via ε-transitions to a common starting point, which yields an automaton that resembles the template shown in the schematic in Figure 3.2a. Subsequently, the result is transformed into its corresponding deterministic finite automaton using the subset construction algorithm [Aho+06, Chapter 3.7.1]. See the schematic in Figure 3.2b. Note that during the subset construction an ambiguity may appear: final states in the nondeterministic automaton are marked with a token type, while the equivalent state in the deterministic automaton may correspond to multiple accepting states in the nondeterministic automaton. This is resolved by picking whichever token is determined to be more important according to some rule, for example, by picking whichever token type appears first in the lexical grammar.

The final step is to transform the result of the subset construction, an automaton which can only match one token before requiring a reset, into one which can continue to emit tokens after the first. The idea here is as follows: when using an automaton which can match only a single token, the automaton is simulated until a transition to the reject state occurs, at which point it is reset. To transform the automaton, we want to replace this transition with one which resets the automaton by moving it back to the start state, and also emits a token simultaneously. Note that since we would like to keep this automaton deterministic, we cannot simply insert an ε-transition here. Instead, for each accepting state of the deterministic automaton, we copy all transitions outgoing from the start state, assign them to the accepting state, and augment them to produce a token which' type corresponds with said accepting state. Note that if an accepting state already has

an outgoing transition marked with the same symbol as an outgoing transition from the starting state, the one from the starting place will be ignored and the existing transition will be retained instead. This causes the automaton to prioritize continuing with the current match rather than to end the match early.

3.1.2. Parallel Deterministic Finite Automata

In order to simulate the automaton as obtained in the previous section in parallel, we employ the method devised by Hillis & Steele [HS86]. Their algorithm works by recognizing that an automaton may be simulated by first mapping each symbol of the input onto a distinct unary transition function using a partial application of the transition function: for some automaton $M = (Q, \Sigma, \delta, q_0, F)$, each symbol a_i of the input $w = a_1 a_2 ... a_n$ is mapped to its corresponding unary transition function $\delta_{a_i}(q) = \delta(q, a_i)$, where $a_i \in \Sigma$ and $q \in Q$. The final state of the automaton, after simulating the entire input w, may be obtained by evaluating $\delta_{a_n}(...\delta_{a_2}(\delta_{a_1}(q_0))...) = (\delta_{a_n} \circ ... \circ \delta_{a_2} \circ \delta_{a_1})(q_0) = \delta_w(q_0)$. Because the function composition operator is associative, we may compute the result of the composition of all unary transition functions in parallel by employing a parallel reduction. To find out the intermediary states of the automaton instead of only the final state, a parallel prefix scan may be employed instead. Both of these parallel operations are also discussed by Hillis & Steele, in the same article.

In practice, these unary transition functions may be implemented as a simple one-dimensional table. Every entry is a transition, mapping an old state to a new state, as if the automaton were in the old state and performed the transition corresponding to symbol associated with the transition function. The composition operator may then be implemented by replacing each of the new states in the right-hand operand with the corresponding state obtained by a lookup in the left-hand operand. As this lookup needs to be performed for each state of the automaton, the composition operator runs in $\mathcal{O}(|Q|)$ time. This yields a total sequential runtime complexity of $\mathcal{O}(|Q| n \log n)$ and a total parallel runtime complexity of $\mathcal{O}(|Q| \log n)$, by evaluating the entire automaton either using a parallel reduction or parallel prefix sum.

While this is a nice parallel runtime bound, each unary transition function additionally requires $\mathcal{O}(|Q|)$ extra space, and since all of the intermediary results need to be saved when using a parallel prefix sum, we gain a total space complexity of $\mathcal{O}(|Q|n)$. While modern commodity graphics processing units do have memory heaps in the orders of a few gigabytes, a lexical grammar for a typical programming language yields a state machine with a few hundred states, and so the memory overhead makes this solution in practice not very efficient. Instead, we propose to implement this method by computing all reachable compositions of unary transition functions ahead of time and assigning an integer identifier to each one of them. This only requires us to store one integer identifier instead of an entire table for an individual unary transition function. Furthermore, the composition operator can then be implemented by means of a simple two-dimensional table lookup, where two identifiers yield the identifier of the unary transition function of their composition. Note that while in theory the upper bound for the amount of distinct unary transition functions may be very large, in practice only a few thousand are required for the lexical grammar of a typical programming language.

The final parallel simulation of a deterministic finite automaton is then performed as follows:

1. First, each symbol of the input is mapped to the integer identifier of its corresponding unary transition function. This step represents the partial application of the transition function, and can be implemented by means of a one-dimensional table lookup. A total of $\mathcal{O}(n)$ operations are required, each of which can be performed in parallel, and so this step runs in $\mathcal{O}(1)$ parallel time.

2. Next, a parallel reduction is performed over the resulting sequence of integer identifiers, using a simple two-dimensional table lookup as composition operator. This step represents the computation of $\delta_w = \delta_{a_n} \circ ... \circ \delta_{a_2} \circ \delta_{a_1}$. As each table lookup can be completed in constant time, the total sequential runtime for this step is $\mathcal{O}(n \log n)$, and the total parallel runtime is $\mathcal{O}(\log n)$.

3. Finally, the integer identifier produced by the parallel reduction in the previous step is mapped back to a state of the original state machine. This step represents the evaluation of $\delta_w(q_0)$, and can be performed in constant time by precomputing all $\delta_j(q_0)$ and storing them in another one-dimensional table.

Each of these steps can be implemented by means of a simple table lookups, and apart from these no more than a constant amount of extra memory is required. For a lexical grammar generating a total of $m = |\{\delta_j ...\}|$ unary transition functions, this requires tables of the following sizes: The table of the first step maps each symbol to a particular unary transition function identifier, which means that a total of $|\Sigma|$ entries are required. The second step combines two identifiers into a third, requiring a two-dimensional table of m by $m = m^2$ elements. Finally, the table used in the third step maps each unary transition function identifier to a state of the original state machine, and so requires another m elements.

To extend this method from regular deterministic finite state machines to a lexical analyzer automaton $(Q, \Sigma, T, \delta, q_0, q_r, Z, F)$ as defined in Section 3.1.1, only a few modifications need to be made. First, we would like to obtain information about the entire simulation of the automaton, and not just the final state. To that end, the second step is modified to use a parallel prefix scan instead of a parallel reduction to generate a sequence of prefix compositions, $\{\delta_{a_1}, \delta_{a_2} \circ \delta_{a_1}, ..., \delta_{a_n} \circ ... \circ \delta_{a_2} \circ \delta_{a_1}\}$. The final sequence of states is then obtained by performing the third step for each of these elements, which yields $\{\delta_{a_1}(q_0), (\delta_{a_2} \circ \delta_{a_1})(q_0), ..., (\delta_{a_n} \circ ... \circ \delta_{a_2} \circ \delta_{a_1})(q_0)\}$. Each of these prefixes corresponds to applying some sequence of transitions on the start state. To produce the final token stream, each of these prefixes is mapped to a value indicating whether the last of these transitions resulted in a token being emitted. The corresponding token can then be found by precomputing $F(\delta_j(q_0))$ for all δ_j. Note that the type of the token to emit is given by the source state instead of the destination state. If a certain prefix $\delta_j(q_0) = (\delta_{a_i} \circ \delta_{a_{i-1}} \circ ... \circ \delta_{a_1})(q_0)$ yields a token, the token's type is given by $F((\delta_{a_{i-1}} \circ ... \circ \delta_{a_1})(q_0))$. Practically, this means that if a certain element of the sequence of prefix compositions generated a token, we can look at the prior element to find out the token's type. The final element is always supposed to produce a token. If not, when $F((\delta_{a_n} \circ ... \circ \delta_{a_1})(q_0))$ yields ε, the input was not valid according to the lexical grammar. To compute whether a particular prefix yields a token, all we need to do is account for the slightly different definition of the transition

function, as our variant yields both the next state and whether the transition produces a token. To this end, the table representing a unary transition function is modified to also map each source state to a flag indicating whether the transition produces a token. The composition operator does not need to be modified, other than that the flags of the right-hand side are ignored. As each unary transition function now encodes slightly more information, this does cause the total number of distinct reachable compositions to grow, but typically not by a very large amount.

3.2. Parsing

The second major stage of the compiler is *parsing*. This stage further processes the sequence of tokens that is produced in the lexical analysis stage. Note that only the types of the tokens are of interest here; due to the lexical analysis stage we already know the lexeme is of a certain syntactical structure, and so it does not need to be validated again. The parsing stage performs two major operations: first, the syntactic structure of the input is validated according to the language's syntactic grammar, which is a set of rules that describe how well-formed programs should look like. If there is an error during this process, the compilation process is terminated and an error is returned to the user. Second, the stage outputs a *parse tree*, which is a tree data structure that represents the token sequence according to the rules in the grammar. Each node in such a parse tree corresponds with a particular rule of the grammar. The purpose of building this tree is that a structured representation of the input is significantly easier to further process than a sequence of tokens.

There are many different types of languages, and even more different methods to parse them. For example, the lexical analyzer discussed in Section 3.1 forms a parser for regular languages. The syntactical grammar of a programming language, however, is often either an LL- or LR-type grammar. These types of grammar allow more complicated constructs, and thus require a more powerful type of parser. In general, there are two main ways to go about parsing these: either top-down, where the parser starts with the root of the tree, or bottom up, where the parser starts with the leaves. Both of these can be modeled by a *push-down automaton*. These are similar to the finite automatons used for regular languages, like those used for the lexical analyzer, except that they also use a stack. Instead of determining the next transition solely based on the next few tokens in the input, the top of the stack is also taken into consideration. Furthermore, during transitions the automaton may pop and push an arbitrary number of symbols from and onto the stack. This stack is precisely what makes parsing in parallel complicated: in order to parse a particular section of some string of input symbols, the parser has to know the current state of the stack. In a parallel setting, however, each processor parsing a section of the input would have to wait on the result of the previous section. This creates a critical chain of computation spanning the entire input, and so effectively reduces the algorithm to the sequential version.

The basis of the parsing algorithm used in our compiler is formed by the algorithm presented in *Parallel LL Parsing* [VM07]. In their work, the authors define the grammar class LLP, a subset of LL grammars, and introduce a parsing algorithm which can deterministically parse languages of this grammar class in parallel. The parsing algorithm runs in $\mathcal{O}(n \log n)$ sequential time and $\mathcal{O}(\log n)$ parallel time, where n is the number of tokens to

parse. The same general approach is taken as with a traditional predictive parser for an LL grammar, where the top of a stack and the next few symbols of the input are used to determine the next action, when in a particular state. The top of the stack, however, is not determined by keeping track of an actual stack. Instead, it is determined from the *previous* few symbols in the input, relative to the current position in the input. The advantage of this method is of course that it removes the critical dependency on the stack, and so sections of the input can now be parsed in parallel. Note that just dividing the input into sections and parsing them in parallel like this is not sufficient to test whether the input is valid, as it does not check whether the sections agree with each other. This is solved by producing an intermediate result for every section, which are then combined in parallel to check the validity of the entire input. Each section, which may be as small as a single symbol, produces a sequence of *stack configuration changes*: each processor first determines the initial contents of the stack by looking at the symbols before its assigned section of the input. The section is then parsed by a regular predictive LL parser, whose push and pop operations are recorded to form an intermediate sequence of stack configuration changes. To check whether the input was valid, the individual stack configuration changes are concatenated. If the combined configuration is balanced, and every push corresponds with a pop of the same type, the input is valid. In order to build the final parse tree, a *left-most parse* is produced first: A sequence of grammar rules, which form a pre-order traversal of the parse tree according to the grammar. It is produced in much the same way as the sequence of stack configuration changes for each section: when a processor parses a section of the input, it records whenever a rule is expanded. These intermediary results are then concatenated to produce the final result.

Determining the initial contents of the stack from the previous few symbols relative to the start of each section leads to a limitation in the languages which this method can parse: for each q look back symbols and k look ahead symbols, there may at most be one sequence of actions which the parser can take, and hence the $LLP(q, k)$ class of grammars. The parser used in our work is $LLP(1, 1)$, and every section is one symbol, which means that given two consecutive symbols from the input, we must be able to tell what kind of construct we are dealing with. We solve these grammar limitations by working around them using multiple methods; either for example by inventing new syntax which avoids problems, adding additional passes to the semantic analysis stage, or by modifying the lexical analyzer. This is further discussed in Section 3.2.1.

Testing whether a particular sequence of symbols is valid according to the language's syntactical grammar is then done as follows:

1. First, each subsequent pair of symbols is mapped to a sequence of stack pop and stack push operations, each marked with some type. This can be performed completely in parallel for each pair.

2. These stack changes are then packed into a single sequence, using a parallel string packing operation. The implementation of this algorithm is described in Section 4.4.

3. A parallel bracket matching algorithm is used to test whether the input is balanced, and whether each stack push operation corresponds with a stack pop operation marked with the same type. If either of these is false, the input was not valid according to the grammar. This algorithm is discussed in Section 4.5

The final parse is then given by first mapping pairs of subsequent symbols to portions of the parse, which are then concatenated using another parallel string packing operation. This parse is then converted into the parse tree, which is further discussed in Section 4.6.

Similarly to the lexical analyzer, the implementation of the parser is divided into two parts: an ahead-of-time computation transforms the syntactic grammar into a number of data structures, which are then loaded onto the GPU at runtime. The parsing algorithm then uses these data structures to validate the token sequence obtained in the previous stage of the compiler and to build the parse tree. Note that the parsing algorithm presented in *Parallel LL Parsing* is completely table-driven, and so lends itself well to real-world machines. Further discussion about the implementation of the parser generator and specifics about the layout of these data structures is given in Chapter 4.

3.2.1. Language Design

As mentioned in the previous section, the parsing algorithm we have chosen to implement in our compiler suffers from a number of inherent limitations. These derive both from the $LLP(1,1)$ parser itself and from the underlying $LL(1)$ parser which parses each of the individual sections. When designing the syntax for our language, we must make careful consideration that we implement a grammar which can be parsed by both grammar classes. The goal is to obtain a syntax which looks somewhat familiar to common imperative languages like C, but as its exact syntax is not always possible some compromises have to be made.

There are two main categories of solutions employed to mitigate these limitations: either by modifying the lexical analyzer, or by altering the grammar of our programming language. The latter can again be divided into two sub-categories: we can either modify the grammar so that it accepts a super set of the desired syntax which the parser is able to parse, and then add additional passes in the semantic analysis stage which check that generated parse tree matches our actual desired input and not any of the supersede. Alternatively, we can modify the grammar to accept a slightly unorthodox syntax which avoids the problem altogether. While the latter of these solutions is of course simpler and so should be preferred, it could yield some very inconvenient language rules, at which point it might justify adding simpler or more consistent syntax and implement additional passes instead.

A concise overview of the mitigations and transformations we have applied to obtain a grammar which is parsable by an $LLP(1,1)$ parser is as follows. Each mitigation is discussed in depth in Appendix A.

- Curly braces are enforced around all compound statements, even if the block only contains a single statement.
- The `else`-part of an `if`-statement is parsed as a separate statement. The tree is later verified that all ‘else‘-statements follow `if`-statements.
- As the sub-block of `else`-statements require braces, syntactic sugar for `else`-statements containing `if`-statements is provided in the form of an `elif`-statement. Similar to `else`-statements, these are parsed as a separate statement, and the relative

order of `if-`, `elif-`, `else-`, and other statements is verified in a later pass.

- Binary operators are natively expressed by left recursion, but as LLP grammars form a subset of LL grammars, this is not supported. We apply the standard transformation to replace left recursive productions with their right recursive equivalent instead. This changes the final parse tree, though, and an additional stage is discussed in Section 4.8.1 to deal with this.

- The $LLP(1,1)$ parsing algorithm cannot correctly discriminate binary and unary minus. We solve this by extending the lexical analyzer to generate different tokens for unary and binary minus. This method is also used by some other parser implementations, like the parallel parsing algorithm discussed by Barenghi et al. [Bar+15].

- Parenthesis in function applications conflict with parenthesis used to enforce the order of operations in expressions. We solve this by using square brackets for function application instead. We do not allow function application to occur after any expression other than identifiers, and so a function application is parsed as an optional bracket-enclosed list of expressions after an identifier.

- Our programming language supports a total of three data types: integers, floating points, and void. Each type is assigned a different token, which solves many problems similar to those outlined in this list, though this solution will not scale to user-defined types. Because type names are tokens otherwise not allowed in expressions, we can express type casting by writing a type name before a parenthesized expression. This does not conflict with parenthesized expressions.

- The prototype of function declarations conflicts with call expressions, if both use a similar bracket-enclosed syntax for arguments and parameters. To solve this, we parse the prototype of a function declaration as a regular expression, and enforce the required syntax during a later pass.

- Variables are declared by a special keyword, which is only allowed in front of an identifier. This creates a conflict with function applications, though, and so function applications are allowed after variable declarations during parsing. Again, the proper syntax is enforced during a later pass.

- To state that an expression produces a value of a certain type, a binary *ascription* operator is used. When this operator occurs as the parent of a variable declaration, an explicit type is assigned to the variable. Parameter declarations also use this operator, though in this case the variable declaration keyword is not allowed. The return type of function declarations is also stated using the ascription operator.

Figure 3.3 shows an example program defined in our programming language.

```
fn fib[x: int]: int {
    if x < 2 {
        return x;
    } else {
        return fib[x - 1] + fib[x - 2];
    }
}

fn main[a: int, b: float]: void {
    while (a + 2) < int(-b * 3.0) {
        a = fib[a];
    }
}
```

Figure 3.3. An example program in our programming language.

3.3. Semantic Analysis

The third and final stage of the front end of the compiler is *semantic analysis*. In this stage, the parse tree of the previous stage is analyzed to determine whether it fits the semantic rules of the language, and is further augmented and transformed into an *abstract syntax tree* (AST). This tree is similar in structure to the parse tree, but additionally associates nodes with information such as data types and variable resolutions. The output from the lexical analysis stage is also used. It is here that lexemes which are associated with certain tokens of interest, such as identifiers, integer and float literals, are translated from their string representations into a concrete value. This abstract syntax tree is then finally passed to the back end of the compiler, where it is turned into machine code. It is also a complete representation of a valid program; when the AST is constructed, no other checks need to be performed, and the back end can assume that the program and AST are in a valid format. The semantic analysis stage consists of a series of passes, which each perform a small portion of the complete analysis and transformation process: for example, a pass might add, replace or remove some nodes from the tree, or may introduce some additional property which is associated to every node. If during any of these passes the program is determined to be invalid, the compilation process is terminated and an error is returned to the user.

3.3.1. The Semantic Analysis Pipeline

Apart from the passes outlined in Section 3.2.1, the semantic analysis stage consists of the following passes on a high level, which are applied in the respective order. The exact implementation details of each of these passes is further described in Chapter 4:

1. First, explicit dereferencing nodes are inserted in the tree. This new node sits between an expression which produces writable storage location for a variable and node which accept actual values, and is responsible for reading the value from the storage location. The main purpose of this node is to convey more information to the back end: It can now explicitly determine when to transform whatever represents an

l-value into an r-value, and does not need to perform the check manually.

2. Next, lexemes are interpreted into values which are associated with the appropriate nodes. Note that there are only a few token types of which we are actually interested in their semantic value: integer literals, float literals, and identifiers. The former two are parsed into their respective 32-bit values here. Furthermore, each unique identifier is assigned an uniquely identifying number, which removes the need for expensive and irregular string processing operations further down the pipeline as we can simply use these numbers instead.

3. Variable resolution is performed. The purpose of this pass is twofold: first, it verifies that the program is correct with regards to variables. That is, for every variable which is used in some expression there must be a variable or parameter declaration with the same name *before* the location where the variable is used. If there is no such variable or parameter, or the variable is used before it is declared, the program is incorrectly structured and the compiler will return an error. Second, every variable node is associated with a reference to the node where the variable is declared. This will help us during later passes, for example during type checking and while constructing the final abstract syntax tree. Shadowing is allowed; if multiple variables or parameters are declared with the same name, the usage of some variable will be resolved to the most recently declared.

4. In the fourth pass, function resolution is performed. This pass is very similar to the variable resolution pass. It verifies that the program is correct with regards to function application and declarations, and yields an error if a nonexistent function is called. Each function application node is associated with a reference to the corresponding function declaration node, which is also used during type checking. Note that this pass is separate from the variable resolution pass because the resolution rules are slightly different. Variables are resolved to declarations within the same function definition, and must be declared before they are used. In contrast, functions are declared in the global scope and have no required declaration order. Furthermore, shadowing is not allowed for functions; if two or more functions are declared with the same name, the compiler returns an error.

5. Fifth, arguments are resolved. This is again quite similar to the other two resolution passes. It verifies that the number of arguments in a function application expression matches the number of arguments in the called function's declaration, and yields a compile error otherwise. Furthermore, the root node of every argument gains a reference to the corresponding parameter declaration in the called function, which is again useful during type resolution and during constructing the final abstract syntax tree.

6. Next, type checking is performed. This is split into two sub-passes, and uses the resolutions obtained in the previous passes. In the first sub-pass, every expression node is assigned a resulting data type, which is either inherited from one of its children or determined from the type of the node. For example, a bit wise operation always produces an integer. In the second sub-pass, the actual data types are checked to see whether a node's data type agrees with that of its children. The reason for splitting this type checking pass into two sub-passes is both to improve efficiency

and to obtain a simpler implementation. In a naive compiler, data type checking is a GPU-unfriendly recursive process. Furthermore, checking the type of a node requires one to first compute the types of its children. While this can also be implemented by emulating recursion using a complex scheduler, our implementation takes a different approach. The second sub-pass is a node-local operation suitable to be parallelized, and as we will see, restricting the number of children from which a node's data type can be inherited to one allows us to quickly compute an initial data type.

7. After type checking is performed, return types are checked. The resulting data type for every sub-expression of a return statement is compared with the data type that the enclosing function is declared to return. If these are not equal, a compile error is returned.

8. If a function is declared to return a non-void data type, then it is invalid for control flow to reach the end of that function without actually returning a value. To this end, function convergence is analyzed in the eighth pass. We do not attempt to solve the halting problem here. The compiler simply checks that every code path eventually ends in a return statement, and returns a compile error if not. Note that some programming languages integrate this property in the type system, and the compiler can check function convergence during type checking. In our case, this would complicate type checking, and so the passes are kept separate. Furthermore, this pass does not check for dead code; writing code directly after a return statement is allowed, but it will never be executed.

9. In the final pass, the final abstract syntax tree is constructed. This entails computing a number of additional node properties, which are required by the back end part of the compiler. Each node is associated with its depth in the tree, as well as an integer indicating the index of the node in the list of children of its parents. Function, variable and argument declarations are assigned integer which indicates the offset in its local context, depending on the type, which the back end part of the compiler uses to relate declarations with usage. In specific, function declaration nodes are assigned a global offset value; every unique function declaration in the program is assigned a sequentially allocated number, starting from zero. The same is done for variable declarations, though in this case the offset is function-local, which means that the first declaration of a function has offset 0. Parameters declarations are also assigned an offset. Besides that the offset is again function-local, the offset is also determined by categorizing different parameter types, where a different counter is kept for integer and float parameters. Nodes which refer to either of these declarations are then assigned with the same integer value as computed for the referee.

The result is a fully-attributed abstract syntax tree, which is ready to be compiled into machine code by the back end part of the project. If none of the previous passes produced an error, the tree is semantically correct, and so the back end does not need to perform any additional verification.

4. Implementation

The implementation of the compiler is written in a mix of C++ and Futhark [Hen+17] code. The former is mostly used for the offline pre-processing tools and the compiler's driver code, whereas the compilation passes themselves are written in Futhark. Writing GPU code in Futhark greatly reduces the required development effort, as it is much simpler to create efficient programs consisting of many kernels than it would be using low level GPU frameworks such as CUDA and OpenCL. Pre-processing code for the lexical analyzer and parser are combined into a single tool, which generates the appropriate data structures during the build process of the compiler. These are then encoded in the compiler's executable. At runtime, the driver code first loads these data structures onto the GPU, along with the source code of the program to compile in verbatim. Each major pass is associated with a Futhark entry point, and to perform the compilation, the driver code calls each of these entry points in the appropriate order. If the program is determined to be invalid during either of the compilation passes, an integer or boolean value is returned from the related entry point indicating failure. When this happens, the driver code terminates the compilation process, and shows a representative error message to the user. Otherwise, if there are no compile errors, the final entry point returns an array of machine code and an array of offsets representing entry points for the individual functions present in the machine code.

Apart from data structures, which are linked into the executable as a raw binary blob, the lexical analyzer and parser generation tool also emits a C++ and a Futhark source file containing additional information that should be available at compile time. For example, they contain the backing types used for various data structures, and programmer-usable identifiers for token types and grammar rules. Additionally, the C++ source code contains information about where in the blob the different data structures are located, so that they may be retrieved and upload onto the GPU when appropriate. The build system ensures that all parts of the build process are executed in order. That is, the pre-processing tool is compiled first, then executed to produce data and source files. Next, the Futhark source code is processed via source translation into C sources, which are then compiled into the final compiler's executable along with the driver code and generated data blobs. The Futhark compiler can generate sources for a number of different runtimes, some of which can be configured at compile time via the build system. This allows the compilation passes of the compiler for example to be executed on GPU hardware via OpenCL or CUDA, as well as on CPUs by using the single- or multi-core C back ends.

The remainder of this chapter is organized as follows. In Section 4.1, the details of the offline processing related top the lexical analyzer. Subsequently, Section 4.2 describes how the lexical analysis process itself is implemented. Section 4.3 describes the offline processing related to the parser. Sections 4.4 describes the parallel string extraction algorithm, Section 4.5 the bracket matching process, which are both topics related to the parsing process itself. Section 4.6 describes the parse tree construction process. Section 4.7 describes a

```
plus = /\+/
minus = /-/
name = /[a-zA-Z_]+/
number = /[0-9]+/
lparen = /\(/
rparen = /\)/
```

Figure 4.1. Contents of an example lexical grammar file.

number of common tree subroutines routinely used during the parser limitation mitigation passes described in Section 4.8, semantic analysis passes described in Section 4.9 and the abstract syntax tree construction pass described in Section 4.10.

4.1. Lexical Analyzer Pre-processing

The pre-processing part of the lexical analyzer follows the construction process outlined in Sections 3.1.1 and 3.1.2, along with the extension discussed in Section 3.2.1. The lexical grammar is supplied as a simple text file, which associates patterns to token names. Patterns consist of a simple subset of common regex syntax, and supports constructs such as concatenation, alternation, zero or more repetition, once or more repetition, zero or one repetition, and character sets. It is invalid for a pattern to be able to match no characters. See the example lexical grammar in Figure 4.2.

The lexical analyzer is then encoded into three data structures:

- A one-dimensional initial table, which maps input characters to the integer identifiers representing the corresponding unary transition functions. This table is indexed by a character's 8-bit ASCII integer value. Unary transition functions are encoded as 16-bit integers. The lower 15 bits indicate the integer identifier, and the upper bit indicates whether a transition produces a token.

- A two-dimensional composition table, which maps pairs of integer identifiers for unary transition functions to unary transition function encoded similarly to the initial table.

- A one-dimensional final table, which maps integer identifiers of unary transition functions to the token type associated to the state obtained when applying the corresponding unary transition function to the start state. An implicit `invalid` token is added, which is used if a particular state is not associated with a token type. Token types themselves are encoded by assigning each a unique 8-bit integer identifier. Programmer-usable constants which associate token type's name with its assigned integer identifier are also emitted into the generated Futhark source file so that they may be used in other parts of the compiler.

Note that the prefix scan operation available in Futhark requires a neutral element. For this reason, an additional 'identity' unary transition function is generated, which when composed with another unary transition function always yields the other.

4.2. Lexical Analysis

At runtime, the lexical analysis algorithm also follows the process outlined in Section 3.1.2. First, every character of the input is mapped to a unary transition function identifier by looking up the value in the initial table at the character's ASCII value. The array of composed unary transition functions is obtained by using a prefix scan and the composition table. This is further processed into two arrays. One which states whether a particular transition produced a token, by fetching the upper bit of each encoded integer identifier, and another containing the token type associated to each intermediary state by a lookup in the final table. The final token type stream is obtained by first shifting the former array one element to the left, and then masking off the elements which did not produce a transition in the latter array. Note that the final element is never masked off.

To test whether the input was valid according to the lexical grammar, we can simply test whether the final element of the token type stream is the invalid token. In our implementation, however, we know that the parser will never accept the invalid token, and so we simply ignore invalid tokens.

Lexemes are obtained by filtering the token stream for indices which produce token types, which yields an array containing the ending indices of each token. By shifting this array to the right one element, we also obtain the starting index. Finally, the lexical analyzer filters out elements of the token stream which actually generate tokens, and returns this along with arrays containing the starting index and length of the associated token. Note that some tokens should not be passed to the parser, for example, comments and whitespace are to be ignored. These are removed by a filter operation.

4.3. Parser Pre-processing

The generation algorithm employed in the offline pre-processing of the parser is a relatively straightforward implementation of the method discussed in the original discussion of *Parallel LL Parsing* [VM07]. A syntactical grammar is supplied to the tool in the form of a simple text file, similar to the lexical grammar. This file contains a number of simple productions, each consisting of a non-terminal left-hand side and a sequence of non-terminals and token types on the right hand side. Furthermore, each production must be assigned a unique, human-readable *tag*, by which the production is later identified. An example grammar specification can be seen in Figure 4.2. Tags are either specified explicitly between brackets, or equal to the left-hand side if unspecified. Token types are specified between single quotes, which in this example relate to those in the lexical grammar in Figure 4.1. The first production in the grammar file is used as the start rule.

A total of three data structures are generated, which are encoded into seven different arrays:

- A stack configuration change table, which maps every possible pair of token types to a sequence of stack configuration changes. This data structure is encoded into three tables. A one-dimensional table consisting of a supersequence which holds ev-

```
expr -> atom sum;
sum [sum_add] -> 'plus' atom sum;
sum [sum_sub] -> 'minus' atom sum;
sum [sum_end] -> ;
atom [atom_name] -> 'name';
atom [atom_number] -> 'number';
atom [atom_paren] -> 'lparen' expr 'rparen';
```

Figure 4.2. Contents of an example syntactical grammar file, which parses simple expressions.

ery sequence of stack configuration changes, and two similarly sized two-dimensional tables containing an offset and length of the corresponding sequence in the supersequence. Individual stack configuration changes are encoded as simple integers. Push operations are assigned $2x+1$ and pop operations $2x$, where x is an integer identifier for the symbol that is being pushed on or popped from the stack. In order to reduce memory overhead, the pre-processor computes the minimum required size for the integer type backing stack configuration changes. This information is then emitted into the generated Futhark source file, so that it may be used by the implementation of the parser.

- A partial parse table, which is similar in structure to the stack configuration change table. Instead of stack configuration changes, this associates each pair of token types with a partial parse. An individual element of the partial parse is encoded by assigning each production of the input grammar with an integer identifier. The minimum required width for this integer is again computed and emitted into the generated Futhark source code, along with Futhark constants relating the production's tag with the integer identifier.

- A one-dimensional production arity table, which maps each production to the number of non-terminals that appear in its right-hand side. This data structure is not required for the parsing process itself, but will be useful when constructing the parse tree. This table is indexed by the same integer identifier that is used in the partial parse table.

The two-dimensional offset and index tables used for the stack configuration change and partial parse data structures are indexed by a pair of integer identifiers, each corresponding to a token type. If the pre-processing tool is supplied with both a lexical and syntactical grammar, the integer identifiers are synchronized, so no translation needs to be performed. Otherwise, token identifiers are computed and emitted to the Futhark source file in a similar manner as with the lexical analyzer. In either case, two additional pseudo token types are generated for the parser, which are used to represent the start and end of input markers used by the parsing algorithm.

a	b	c	d	e	f	g	h	Superstring
0	1	2	3	4	5	6	7	Superstring indices
				5	1	2	4	Substring offsets
				2	3	1	2	Substring lengths
				0	2	5	6	Offsets of substrings in the result
				5	-5	-1	2	Difference between ends and starts of runs
5	1	-5	1	1	-1	2	1	Scatter differences into array of ones
5	6	1	2	3	2	4	5	Prefix sum to obtain runs
f	g	b	c	d	b	e	f	Final result by gathering

Figure 4.3. Performing a parallel string extract

4.4. Parallel String Extraction

The parallel string extraction algorithm is an important component of the parsing algorithm. As described in Section 3.2, pairs of symbols from the input are first mapped to sequences of stack configuration changes for the verification step, and a partial parse to construct the final parse. Each of the local results are concatenated into a global stack configuration change sequence and global parse respectively through the parallel string extraction algorithm.

There are two inputs to this subroutine: a superstring, corresponding to the supersequence stored in the stack configuration change and partial parse tables, and two arrays containing the offset and length of every string, obtained by mapping pairs of symbols. The goal is to extract and concatenate all the substrings from the superstring, in order, and produce a single array holding the result.

The main idea behind our implementation is to first build an array of indices using a parallel prefix scan. This array consists of a concatenation of *runs*, sequences of integers which represent the indices of some substring in the superstring. These are built by scattering some initial values into an array of ones, after which the prefix sum is executed and each initial value followed by a number of ones are converted into a run. Since we use a prefix sum, the initial values of each run must be the difference between the last value of the previous run and the first value of the next. This is also computed by using a prefix sum. Figure 4.3 illustrates an example of the parallel string extraction process.

4.5. Parallel Bracket Matching

After the complete sequence of stack configuration changes is obtained through the parallel string extraction algorithm discussed in the previous section, it needs to be checked for validity. Every push operation which pushes a certain symbol on the stack needs to correspond with a pop operation that pops the same symbol, and the pop operation needs to performed only after the push operation. As is shown by Vagner & Melichar [VM07], this problem maps to a different problem known as *parallel bracket matching*. Every push

$[^a$	$[^b$	$]^b$	$[^c$	$[^d$	$]^d$	$]^c$	$]^a$	Initial bracket sequence
1	1	−1	1	1	−1	−1	−1	Map to nesting level difference
1	2	1	2	3	2	1	0	Inclusive prefix sum to compute nesting depth
0	1	1	1	2	2	1	0	Decrement nesting depth of opening brackets

Figure 4.4. Computing nesting depth of bracket sequences

$[^a$	$[^b$	$]^b$	$[^c$	$[^d$	$]^d$	$]^c$	$]^a$	Initial bracket sequence
0	1	1	1	2	2	1	0	Nesting depth
$[^a$	$]^a$	$[^b$	$]^b$	$[^c$	$]^c$	$[^d$	$]^d$	Sorted bracket sequence
0	0	1	1	1	1	2	2	Sorted nesting depth

Figure 4.5. Bracket matching by sorting on nesting depth

operation pushing a certain symbol corresponds with an opening bracket marked with that symbol, and the same goes for closing brackets and pop operations. We implement two different ways to check that the sequence of brackets is balanced: by sorting, and by reduction to a different problem.

For both methods, we first need to compute the *nesting depth* for each bracket: an integer value representing the number of pairs of opening and closing brackets that surround the bracket, ignoring bracket's type. This is done by first mapping each bracket to the nesting level it generates. Opening brackets increase the nesting depth by one, and closing brackets decrease the nesting level by one. The nesting depths are then obtained by performing a parallel prefix sum over the differences, and adjusting the depth of opening brackets. See the computation in Figure 4.4. The time complexity of this step is bound by the prefix scan, which runs in $\mathcal{O}(\log n)$ parallel time, where n is the number of brackets in the input.

The first method is relatively simple, and works by first sorting the brackets according to their nesting depth. This will cause all brackets with the same nesting depth to be placed next to each other. If this sort is stable, relative order of brackets with the same nesting depth will be maintained. Note that if the input is valid, the opening bracket corresponding to a closing bracket is the first element prior to the closing bracket which has the same nesting depth, and so checking whether the brackets are balanced is now as easy as checking whether non-overlapping pairs of brackets match. See the computation in Figure 4.5. We sort the input using a GPU-friendly parallel radix sort, which is stable and runs in $\mathcal{O}(\lg h)$ parallel time, where h is the height of the parse tree.

Note that while radix sort can be parallelized very efficiently, it still has a large overhead. For this reason we implement an alternative implementation by reduction to yet another problem, the *previous smaller or equal value* problem, a variation of the *all nearest smaller values* problem. Given an array of values, every element is mapped to the prior element which has a value less than or equal to that of itself. In a valid input, elements between a pair of opening and closing brackets all have a higher nesting depth, and so an opening bracket corresponding to a closing bracket can be found by solving the previous smaller or equal problem for closing brackets over the array of nesting depths. Note that opening

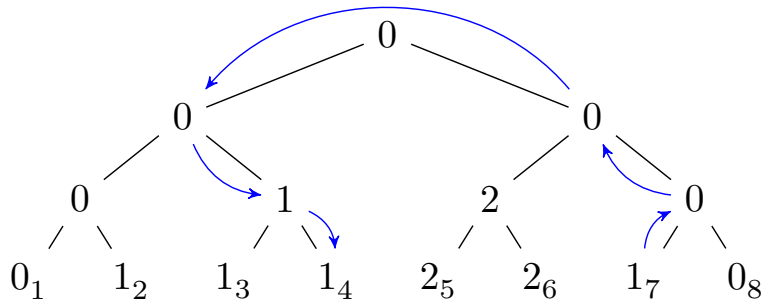


Figure 4.6. Finding the previous smaller or equal value. The subscript shows the leaf's offset.

brackets are ignored, nor do we solve a next smaller or equal value problem for opening brackets. Instead, we first verify that the input is balanced when ignoring the bracket's type by checking that the nesting depth of the last bracket is zero, and check that the nesting depth never becomes negative. This ensures that every closing bracket is paired with exactly one opening bracket, and no opening bracket is left unpaired. Finally, we check that the marking of every paired opening and closing brackets match.

4.5.1. Finding the Previous Smaller or Equal Value

Many parallel algorithms for the all-nearest smaller values problem exists. While Berkman et [BMR98] discusses an optimal parallel algorithm running in $\mathcal{O}(\lg \lg \lg n)$ time, this algorithm is relatively complicated to implement in GPU contexts. Instead, we implement the algorithm proposed by Bar-on & Vishkin [BV85], which uses a binary tree of minima to find the previous smaller or equal value to solve the bracket matching problem. Finding the corresponding previous smaller or equal value for an element is done by traversing the tree in two steps. First, by walking up the tree until we find a previous sibling node which has a smaller or equal value. If this is the case, then either of its descendants must hold a smaller or equal value, and so we proceed to work our way down the tree. This is done by repeatedly selecting the child node which holds the largest value that is still smaller than or equal to the value of the initial element. When a leaf node is reached, the process terminates, and the algorithm returns the offset of the leaf in the original array. See the example traversal in Figure 4.6. As the height of the tree is logarithmic in number of leaves, and since the number of leaves is equal to the amount of brackets in the input, the total runtime time for processing a single element is $\mathcal{O}(\log n)$. We can perform the traversal for each of these elements in parallel, and so this is also the parallel runtime for this step.

We store the tree as a perfect binary tree, to make construction and traversal simpler at the cost of some memory overhead. Construction is performed sequentially level by level, and by parallelizing the work of computing the minima for internal nodes during construction of a level. This again yields a parallel runtime of $\mathcal{O}(\log n)$ for constructing the tree, and also $\mathcal{O}(\log n)$ for the entire bracket matching process.

4.6. The Parse Tree

Once the input has been validated and the complete left parse is obtained, it is time to build the parse tree. We represent the program using an inverted parse tree. Whereas in classical compilers nodes often contain references to their children, we represent our program by giving each node a reference to its parent. This representation was inspired by Hsu's dissertation [Hsu19], and gives us a number of key advantages:

- In a representation where every node contains references to its children, node sizes may differ greatly from another. For example, a binary operator always has two children, while a unary operator only has a single child, and literals have no children at all. This is not much of an issue on architectures where dynamic memory allocation is easy and efficient, but this is not the case on GPUs. Note that while we can compute the maximum amount of children for nodes ahead of time and allocate this amount to every node, this leads to a memory overhead for nodes which do not have the maximum amount of children. The inverted representation requires us to only store a single parent for every node, allowing us to store the parse tree simple and efficiently in two arrays: index i stores the type of node i in the first array, and the index of the parent in the second array.

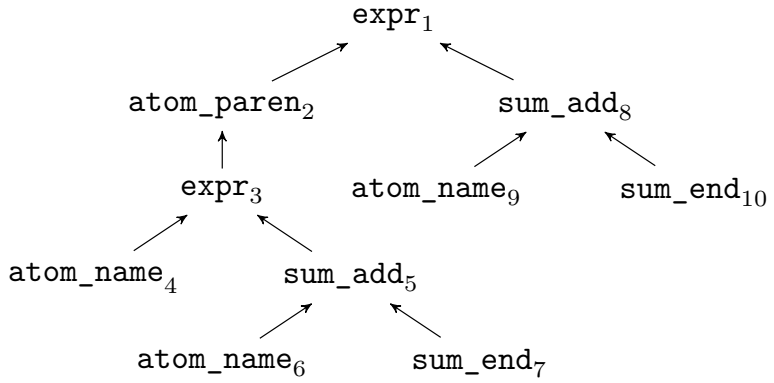
- Classical compilers often process the tree in a top-down manner by using recursion. Child pointers are used to traverse the tree from root to leaves, and a stack is used to store the current path so that the compiler may traverse back to the parent. While recursion is possible on modern GPUs, this requires complex scheduling systems. A representation where nodes only have a single reference eliminates this recursion and transforms it into an iterative process, which can be trivially parallelized by performing it for every node at the same time. Furthermore, a representation in which nodes only have a single pointer to their parent essentially converts the tree into a forest of linked lists, which allows us to process it using the parallel linked list primitives described in the work by Hillis & Steele [HS86].

4.6.1. Construction

We use the left parse produced by the string extraction step to serve as our array of node types, and so all we need to do to construct the parse tree is compute an additional array containing the index of the parent for each node. Note that the left parse forms a pre-order traversal of the nodes of the parse tree, and so a parent always has a lower index than its children. The tree is constructed by a variant of depth-first traversal, recording the stack depth at every node, and then solving the *previous smaller or equal value* problem over this array of depths. During the depth-first traversal, we iteratively pop the topmost node from the stack, and push its children. This corresponds with decreasing the stack depth by one, and then increasing it by the number of children the node has. We compute the final depths in the same way as during the parallel bracket matching algorithm. First the stack depth is constructed, by mapping each node type to the number of children it has using the arity array and subtracting that by one. An exclusive prefix sum then gives the final stack size for each node. Note that since we pop the parent before we push the children, the last child of a node has the same depth as its parent, and all the other

Index	Left parse	Arity	Stack change	Stack depth	Parent
1	`expr`	2	1	0	none
2	`atom_paren`	1	0	1	1
3	`expr`	2	1	1	2
4	`atom_name`	0	−1	2	3
5	`sum_add`	2	1	1	3
6	`atom_name`	0	−1	2	5
7	`sum_end`	0	−1	1	5
8	`sum_add`	2	1	0	1
9	`atom_name`	0	−1	1	8
10	`sum_end`	0	−1	0	8

(a) Derivation of the parent indices for the input $(a + a) + a$



(b) Visualization of the parse tree of $(a + a) + a$. Subscripts show node index.

Figure 4.7. Computation showing the construction of a parse tree for $(a+a)+a$ according to the grammar in Figure 4.2.

descendants of the parent have a higher depth. To construct the final parent array, we re-use the implementation given in Section 4.5.1 to solve the *previous smaller or equal value*. The example Figure 4.7 shows the construction process for the parse tree of the input $(a + a) + a$.

4.7. Common Tree Subroutines

The implementations of the different compiler passes used during the mitigation of parser limitations, semantic analysis and constructing the abstract syntax tree make use of a common set of tree operations, which are described in this section.

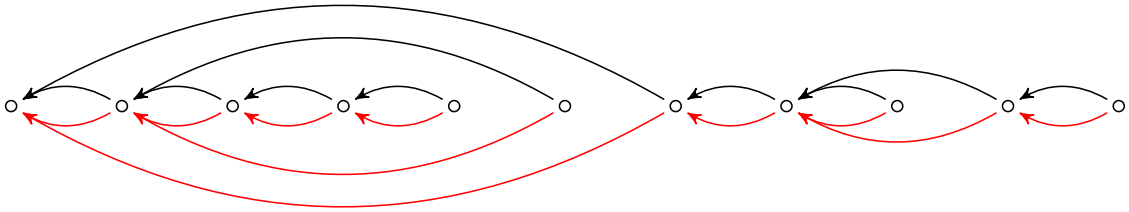
4.7.1. Tree Compactification

Nodes are removed by setting the parent index of the node to a value that indicates that it is no longer part of the parse tree, and by adjusting the parent index of the direct children. Nodes of which the parent equals this special value are ignored during further processing. This makes a quick and efficient method to effectively remove nodes from the tree, at the cost of still having to process them. Some passes remove quite a lot of nodes, and so at some point it becomes more efficient to shrink the arrays backing the parse tree and removing the nodes for good. This operation is performed by first mapping each node to 0 when it is removed, and 1 otherwise. Performing an exclusive prefix sum over the resulting array yields an array holding for each node the new index. The compacted tree is then obtained by permuting the arrays backing the tree. Parent indices are updated by performing a lookup into the array holding new indices. Note that this operation retains relative node ordering.

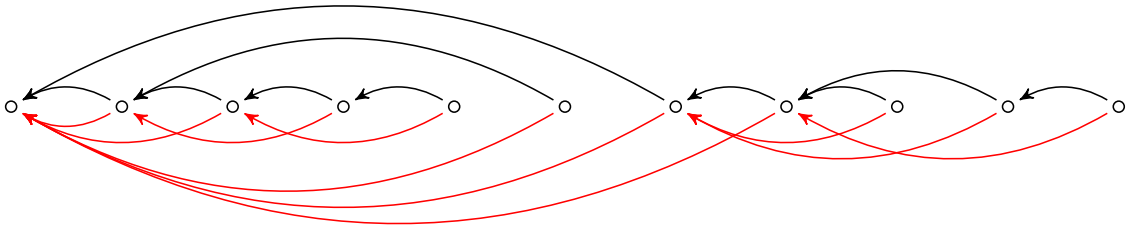
4.7.2. Finding Root Nodes

An essential building brick of several tree operations is to find the root node of a (sub) tree. For example, by partitioning the tree into a forest of trees we can perform operations such as finding the closest parent which satisfies some condition. This can in turn be used for operations like adjusting the parent indices of children of removed parents, by finding the closest parent which is not removed. The most straightforward way to implement this in a parallel manner is to, for each node in parallel, follow the list of parent pointers until we reach the root. This yields an algorithm which runs in $\mathcal{O}(h)$ parallel time, where h is the height of the largest tree. Note that this naive algorithm is not particularly inefficient in practice. A GPU would process the nodes in blocks, each node assigned to a thread. When all roots in a block have been found the loop exits, and so if the depths of nodes in a group are similar not much effort is wasted. No complex scheduling logic is required, since this is already integrated into the GPU hardware.

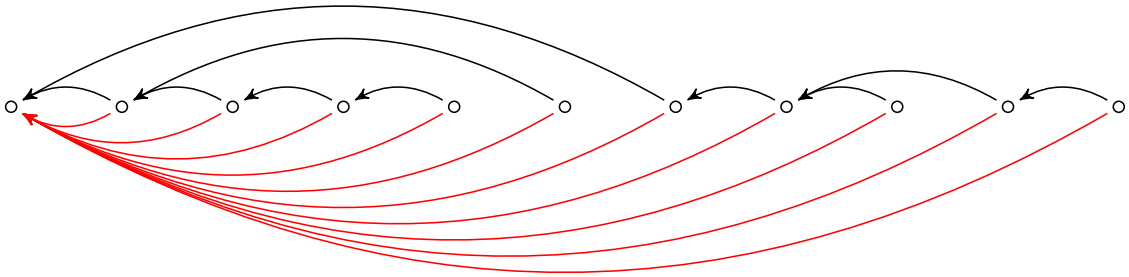
A theoretically more efficient method is obtained by repeatedly mapping each node to its grandparent, if it exists. This effectively doubles the step size every iteration. In the second iteration, the node is again mapped to its current grandparent, but as its grandparent has



(a) Initial tree.



(b) After the first iteration.



(c) After the second iteration.

Figure 4.8. Finding the root node of a tree in parallel. Black (above) edges show the original parents, red (below) shows the new parents at every iteration.

also been mapped to its grandparent, the effective step size is 4 nodes. In the third iteration, this step size is increased to 8 nodes, and so on, yielding a total parallel runtime of $\mathcal{O}(\lg h)$. See the computation in Figure 4.8. This algorithm is in fact a direct application of the algorithm by Hillis & Steele [HS86] to find the end of a linked list. Every path from a node to the root forms a linked list, and we apply the algorithm to each of them simultaneously. Even though different nodes may share a prefix of this list, the intermediary results are the same for both prefixes and so the algorithm works out fine. In our implementation we avoid bookkeeping h by recognizing $h \leq n$, where n is the *total* number of nodes in all trees combined, and simply iterate $\lceil \lg n \rceil$ times. Although this algorithm is theoretically more efficient than the linear approach, in practice it suffers from a large performance overhead due to having to write back the intermediary results to global memory on every iteration. For this reason, we implement both the logarithmic and linear version, and use the former when the expected height of subtrees is large and the latter when it is small.

4.7.3. Computing Node Depths

In the same way that the parallel algorithm to find the end of a linked list can be applied to the parse tree, the parallel algorithm to compute the prefix sum of a linked list which Hillis & Steele [HS86] discuss may also be used. Again, even though nodes may share a prefix of the linked list, the intermediary results are the same in both cases, and so the algorithm works out. Computing a prefix sum over nodes in a tree is useful in a few occasions. Most notably, by initializing an array with ones and performing a parallel prefix sum over the tree using these values we obtain an array where the index of a node gives its depth in the parse tree.

4.7.4. Computing Sibling Indices

During some operations it is necessary to know relational information about siblings. For example, some operation might only need to be performed for the left or right child of a node, or an operation might even depend on the type of another sibling. For this reason, we compute a *sibling array*, which relates each node with an index of the previous or next sibling, if it exists.

The sibling array which contains indices of previous siblings is computed as follows:

1. First, pack the arrays backing the parse tree using the Compactification algorithm as described in Section 4.7.1.

2. Second, the depth of every node is computed, using the algorithm described in Section 4.7.3.

3. Next, the nodes are sorted by depth using a stable radix sort. So long as the siblings maintain their relative order in the array backing the parse tree, and siblings of different nodes with the same depth are not interleaved, this operation yields an array where all children of a certain node are positioned consecutively and in order.

4. Finally, to find the previous sibling for a node, one only has to look at the node in the previous position in the sorted array. If both nodes have the same parent, the previous node is the previous sibling. If not, the node was the left-most child of its parent.

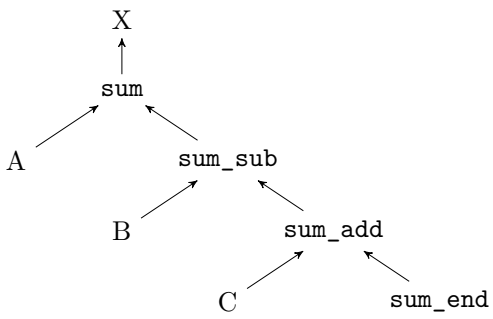
In order for this algorithm to work, the property discussed in the second step needs to be guaranteed. Since the array of node types is directly formed by the productions in the left parse of the input, the parse tree is initially laid out in pre-order, which satisfies this requirement. Sibling arrays are not initially required, and to avoid recomputing or updating the sibling arrays they are only computed after the passes which remove nodes from the parse tree. While some passes manipulate the tree in such a way that the backing arrays no longer from a pre-order traversal of the parse tree, care is taken that none of the passes violate the properties required to compute sibling arrays.

4.7.5. Computing the Right- or Left-Most Descendant

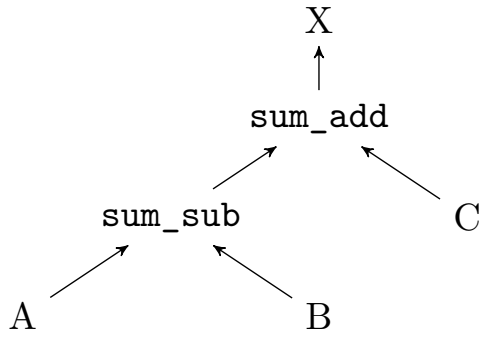
While performing some passes, it is required to know the index of the node which is the right- or left-most descendant of another node; the node obtained by repeatedly selecting the respectively right- or left-most child until we find a leaf node. This is for example useful during variable analysis or re-ordering the tree into pre- or post-order, as the right-most descendant of the previous node gives the previous node in pre-order ordering. Using the previously obtained computational primitives, this operation is relatively simple. First, we compute an array holding for each node the index of its first or last child for finding the left- or right-most descendant respectively. This can be found by writing the indices of nodes which are the first child of their parent, tested by checking the sibling arrays obtained using the algorithm discussed in the previous section. This yields a forest of linked lists, which when traversed end in the target node. To obtain the final array of descendants, we simply use the algorithm discussed in Section 4.7.2 to map each node to the root of the linked list it appears in.

4.8. Mitigating Parser Limitations

As described in Section 3.2.1, the parallel parsing method used in our compiler suffers from a number of fundamental limitations. Many of these limitations are mitigated by introducing additional compilation passes. These verify that the original program is syntactically valid by checking that the parse tree is in the right structure, and restructure the parse tree to a form that is more suitable for further processing. In general, these operations either consist of small, node local operations which are performed on all nodes of the tree in parallel, or of tree-wide operations implemented by variations of the parallel tree algorithms described in Section 4.7. Navigating the parse tree is mostly implemented by scattering and gathering. For example, to fetch the type expression related to a particular cast node, we can check for every node whether it is a type expression and whether its parent is a cast expression, and write the index of that node at the location of its parent in an auxiliary array. During these operations, we avoid race conditions by carefully considering the current structure of the parse tree. For example, a cast node always has



(a) The resulting subtree after parsing $A - B + C$.



(b) The desired subtree representing $A-B+C$ correctly.

Figure 4.9. The initial and desired parse trees of $A - B + C$ according to the grammar in Figure 4.2.

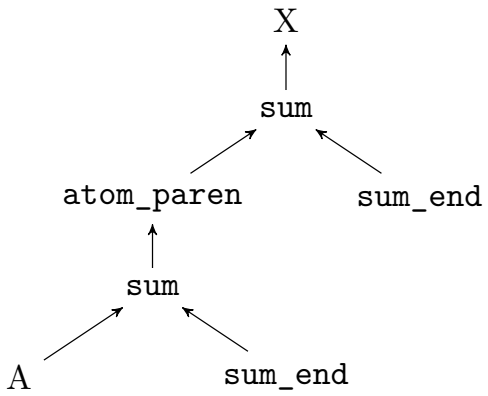


Figure 4.10. Parse tree of (A). Some nodes have been omitted for the purpose of this example.

exactly one type expression as child, and so this child can safely write values at the index of its parent in auxiliary arrays. In the case of binary operators, though, both children may be of the same type, and so these would need to be handled in another way.

4.8.1. Restructuring Binary Operators

As described in Section 3.2.1, the parser cannot parse left-recursive binary expressions in a correct manner, which is resolved by a modification to the grammar and an additional semantic analysis pass. Initially, when the program contains an expression such as $A - B + C$, where A, B and C are arbitrary subexpressions, we obtain a parse tree as shown in Figure 4.9a. This has a number of problems. First, the tree does not represent the expression correctly. As `sum_add` is a child of `sum_sub`, this represents the expressions $A - (B + C)$, whereas we would like perform the subtraction first. Secondly, the `sum` and `sum_end` do not carry much semantic information, and can be ignored. These kinds of nodes are always generated even if there are no actual addition or subtraction operators, as they are a side effect of the transformation we applied to replace left-recursive grammar productions with their right-recursive equivalent. See for example the parse tree of the

expression (A) as shown in Figure 4.10. In this case, there is only one precedence level, but if there would be more then there would be a pair of nodes for each precedence level in every subexpression. Instead, we would like to obtain a tree as shown in Figure 4.9b, where the nodes representing operators are properly structured and the other nodes have been removed.

To this end, we first categorize the nodes of the parse tree into four categories:

1. List head nodes. These are formed by nodes like `sum`, and form the initial node of a precedence level.
2. List intermediary nodes. These are the actual operators, like `sum_add` and `sum_sub`.
3. List end nodes, like `sum_end`.
4. Other nodes, which are not of interest in this pass.

Collectively, a chain consisting of head, intermediary and end nodes is called an *operator list*. Note that the list initially only consists of right children, with the exception of the head. While any of the left subtrees of the list may in itself be list nodes, these are considered a different list. The basic restructuring process is then as follows, also shown in Figure 4.11:

1. First, note that in the desired result in Figure 4.9a, nodes A and B point to the same parent, but this is not the case in the original. To fix that, we set the parent of left children of list intermediaries to their grandparent, obtaining the tree in Figure 4.11a.

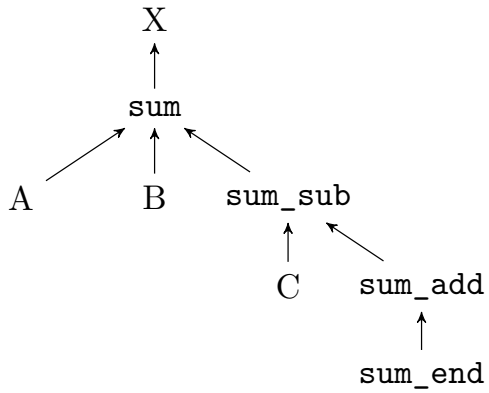
2. In the previous step, left children of intermediary nodes have become dissociated with their proper parent. This is fixed by moving all the node types of the list intermediaries up to their parent, which also eliminates the head node. The original end node is also removed. This yields the tree in Figure 4.11b.

3. Next, we will temporarily use the new end node to function as the list head. To this end, we use the linear algorithm discussed in Section 4.7.2. The resulting tree is shown in Figure 4.11c.

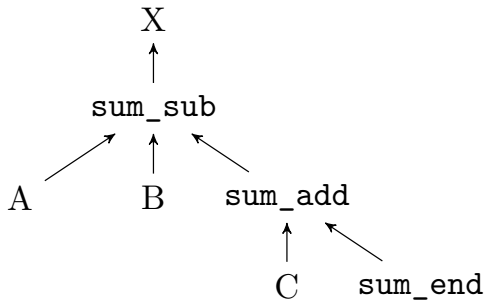
4. To perform the final restructuring, the parent pointer of list intermediaries is inverted to that of their (original) left child. This transforms the chain of left children into an inverted chain of right children, imposing the right order of operations. See the result in Figure 4.11d.

5. Finally, the end nodes are removed, and leaves us with the desired tree as shown in Figure 4.9b. This and the second step in particular deal with the superficial nodes as those in Figure 4.10.

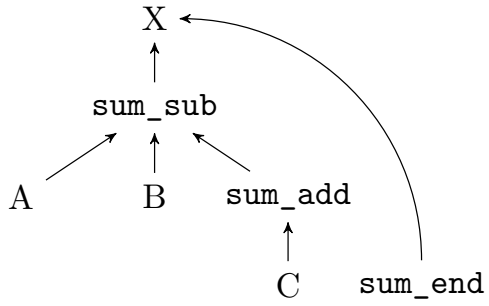
Note that initially the nodes in the arrays backing the parse tree were laid out in pre-order. The operation performed in step 4 breaks this order partially, as parents now will no longer strictly have a lower index than their children. This will have implications further down the pipeline. See an example of the effect in Figure 4.12. Finally, in our implementation there are quite a number of precedence levels, and all of these generate pairs of superficial nodes. This results in that a large part of the tree is removed in this pass. To improve efficiency, we perform the algorithm described in Section 4.7.1 after this pass to shrink the



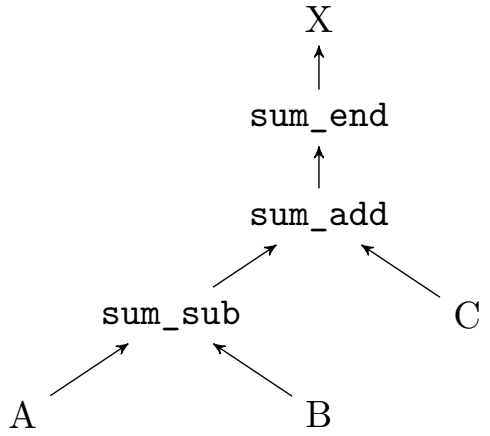
(a) Step 1: Make children of intermediaries point to the right parent node.



(b) Step 2: Move the types of intermediary nodes one up, and remove the original end.

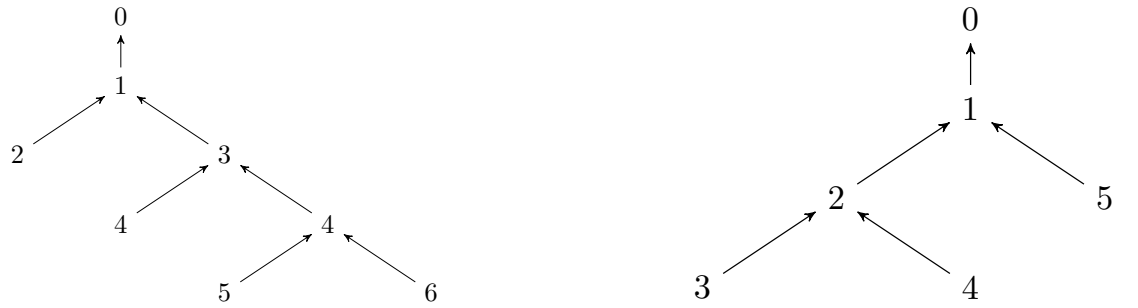


(c) Step 3: Make the list end node temporarily point to the parent of the original head.



(d) Step 4: Make list intermediaries point to their original right child.

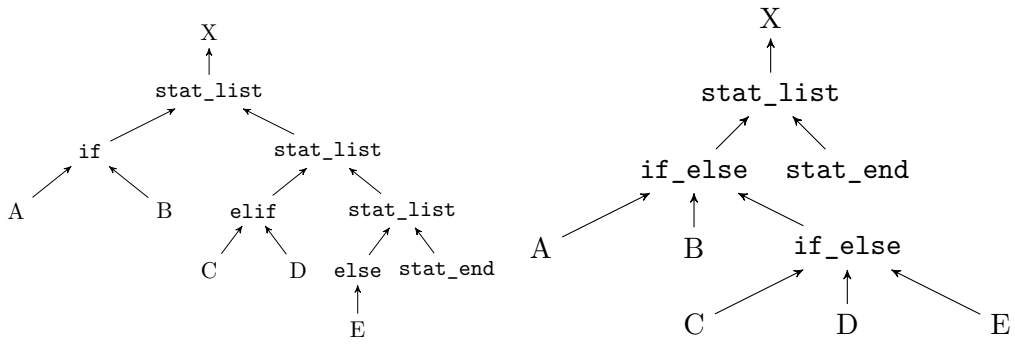
Figure 4.11. Restructuring binary operators.



(a) Node indices before restructuring.

(b) Node indices after restructuring.

Figure 4.12. Node indices before and after restructuring binary operators.



(a) The initial parse tree after parsing.

(b) The desired parse tree.

Figure 4.13. The initial and desired parse tree of `if A { B } elif C { D } else { E }`.

size of the backing arrays.

4.8.2. Restructuring Conditional statements

Similar to binary operators, `else` statements can also not natively be expressed by the grammar accepted by the parser. As described in Section 3.2.1, this is also fixed by altering the grammar, and introducing extra passes to both verify and restructure the parse tree. With this modification, `if`, `elif` and `else` statements are separated into different statements. For example, for the input `if A { B } elif C { D } else { E }` the parser now generates the parse tree shown in Figure 4.13a. There are few operations that need to be performed. First, we need to verify that the input is valid. This entails checking that an `elif` node follows a `if` or another `elif` node, and checking whether an `else` node follows either an `if` or `elif` node. Second, we would like for the tree to more accurately represent the input, by making the `elif` and `else` blocks children of the `if` node. With that modification, there is not much use for a `else` and `elif` node. It is useful to know whether or not an `if` node has a `else`-branch though, and so we replace `if` nodes which have an `else` block with a `if_else` node.

The restructuring and verification process happens as follows:

1. First, the tree is restructured by changing the parents of `elif` and `else` nodes to point to the previous statement, which is the left child of their grandparent. If there exists no such node, the input is invalid.

2. Now that the parents of `elif` and `else` nodes are correct, checking the structure can be done by checking whether the parent of these nodes is the right type. For `elif` nodes, the parent must be an `if` or `elif` node, and for `else` nodes the parent must be an `elif` or `if` node.

3. The parent of `else` and `elif` nodes is set to `if_else` to indicate that this `if` node has a third child. Note that `if` nodes which do not have an `else`-branch are unaffected by this step.

4. Finally, superficial nodes are removed: `elif` and `else` do not carry any useful semantic information. Furthermore, the `stat_list` nodes which were initially parents of `elif` and `else` nodes are removed as well.

Applying this process to the example parse tree shown in Figure 4.13a yields the tree shown in Figure 4.13b. Note that even though this pass restructures the tree, the pre-order layout of the nodes in the arrays backing the parse tree is not invalidated.

4.8.3. Other Syntax-Related Processing

Besides restructuring binary operators and conditional statements, Section 3.2.1 describes a number of other problems related to the parser which require additional processing after the parsing stage. These passes are relatively similar to the previous two passes, though a somewhat simpler. A concise overview of the other parser mitigation passes is as follows:

- Lists of which the intermediary nodes bear no semantic information, like function declaration-, argument- and statement-lists, are flattened by removing the intermediary and end nodes.

- Function application expressions without arguments are cleaned up.

- We verify that no variable declaration nodes are the parent of function application expressions.

- Identifiers with a function application are merged into a single `atom_fn_call` node.

- Type ascriptions are cleaned up.

- Function declarations are checked for validity and the corresponding subtrees of the parse are cleaned up.

- Argument- and parameter lists are discriminated by introducing `param` and `param_list` nodes.

- Variable declarations with type ascriptions are merged into a `atom_decl_explicit`

node.

- Finally, the validity of assignment operators is checked by verifying that the left-hand side of every assignment operator is either a declaration or identifier node.

Appendix B goes further into detail on each of the above operations.

4.9. Semantic Analysis

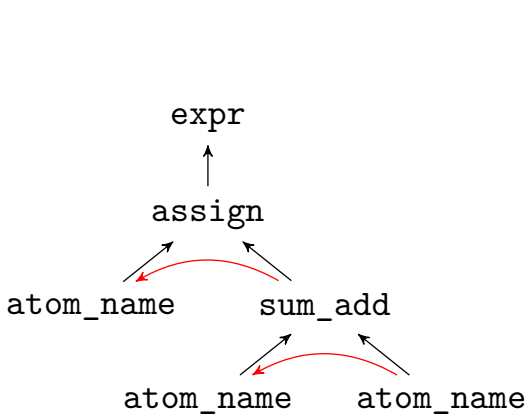
After the parse tree has been restructured and validated, the semantic analysis pipeline is performed. This consists of the implementations for the passes described in Section 3.3. Apart from one operation, most of the transformation passes have been performed, and many of the semantic analysis passes consist of validating the semantic structure of the program or computing additional properties required to perform this validation. These operations are implemented similar to how the parsing mitigations are implemented; either by small, node-local operations which can be executed for all nodes of the tree simultaneously, or by parallelizing large, tree-wide operations in a clever way.

4.9.1. Inserting Dereference Nodes

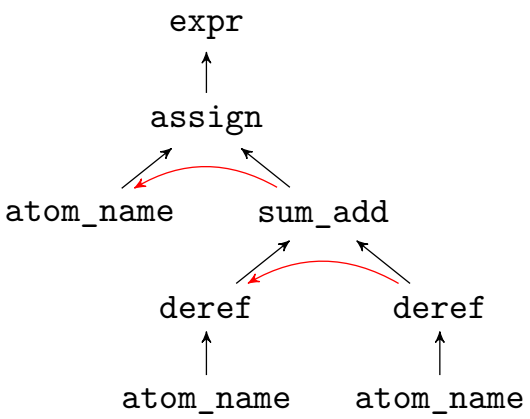
The back end stage of the compiler explicitly requires to know at which point an l-value, an expression which produces a writable storage location for a value of some type, transforms into an r-value, the actual value. This node represents the act of reading the value stored by the variable into more local memory, so that arithmetic and other operations may be performed on it, which is referred to as 'dereferencing' throughout the remainder of this research. There are three node types which produce l-values: variable declarations (both the implicit and explicit version), and regular variable references. In general, all nodes which have expressions as children require r-value expressions, except:

- The left-hand child of assignment expressions.
- Nodes wrapping expressions as statements. In this case, the result is discarded, and so the dereference is superficial.
- The child of parameter nodes, which are handled separately.

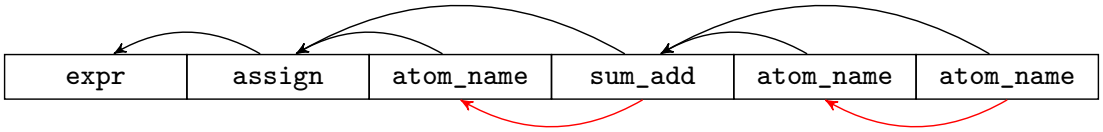
In most of the previous operations, except the pass which checks the validity of assignment operators briefly described in Section 4.8.3, left and right children could be distinguished from each other by the context and by the syntactical grammar. For example, to check whether the left child of the ascription in a function declaration is a function application, we first compute a mask of function application nodes whose parent and grandparent have the required structure. The function declaration node associated to these application nodes is then marked, and we verify that every function declaration is marked to verify that the program is correct. While the ascription node does have multiple children, whose parents also satisfy the required structure, the right child is always a type expression and not a function application expression. This is not the case for the assignment operator, however, as the syntactical grammar allows expressions on both sides. Instead, we use the



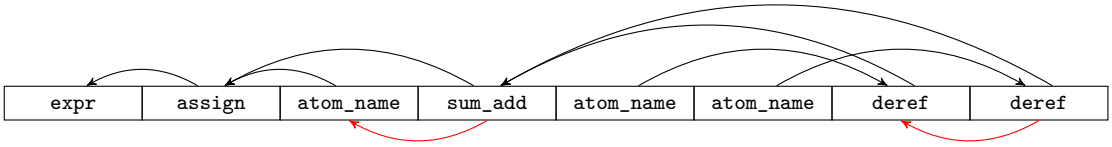
(a) The initial parse tree of $a = b + c$, after previous transformations.



(b) The tree after inserting dereference nodes.



(c) The initial array backing the parse tree.



(d) After applying the transformation.

Figure 4.14. The parse tree before and after inserting explicit dereferencing nodes. Red edges indicate the previous sibling.

sibling arrays described in Section 4.7.4 to determine whether a node is the left-hand child of its parent. Note that this array is also required during the earlier mentioned pass which verifies the validity of assignment operators. As there are no other passes which transform the tree between that and the current pass, the array can be re-used.

This is the only pass which transforms the parse tree in such a way that it adds *new* nodes. This is generally done by first computing arrays of the new nodes, and then appending it to the existing array. To avoid recomputing the sibling array, we also update it also. This means that three arrays need to be updated to account for the new parents: the node types array, the array with indices of parents, and the array of indices of siblings. The updating process is performed as follows: First, the newly inserted dereference nodes copy the parent and sibling index of their child. Children are now the only child of their parent, as dereference nodes only have a single child, and so the index of the sibling of these nodes are set to invalid. Finally, the parents are updated so that they reference the newly inserted nodes. Note that this operation invalidates the ordering of the nodes in the arrays backing the tree as required by the process discussed in Section 4.7.4, however, after this pass no other nodes are removed from or inserted into the parse tree. Furthermore, the order is now given by the parent and sibling indices, which will be used in the other passes to infer relative order of siblings and to finally re-order the nodes of the tree. Figure 4.14 shows an illustration of this process, where dereference nodes are inserted in the expression $a = b + c$. Note that these nodes are only inserted between the parents of b and c, as these variables are used in a context where they are read from. a is written to, and so does not require to be dereferenced.

4.9.2. Extracting Lexemes

Semantic values obtained from lexemes are associated to the parse tree by creating a new *data array*. In this array, each node is associated with one unsigned 32-bit integer containing semantic information. For integer literals, this holds a 32-bit signed integer encoded as an unsigned integer. Similarly, floating point values are also reinterpreted into a 32-bit unsigned integer value. Both the front end and the back end of the compiler do not require these values to be more than raw bits: The values of integer and float literals are not of interest for semantic analysis, and are loaded as raw bits by the final machine. Identifiers are also translated to an integer, which is arbitrary apart from that equal identifiers have an equal value, and different identifiers have a different value. This reduces the amount of passes that deal with strings to the current one. For nodes which do not carry semantic information other than their type, such as operators for example, the corresponding element in the data array has an undefined value and is used.

When the parse tree is first constructed, the nodes in the backing array follow a pre-order traversal of the nodes in the tree. Furthermore, for each of the token types of interest, there is no subtree-producing non-terminal left of them in the right-hand side of a production. This means that the relative ordering between token types of interest is the same as the ordering of the associated nodes in the pre-order traversal. The basic idea of associating information obtained from lexemes to nodes in the parse tree is then as follows:

1. First, the lexemes of tokens of interest are extracted by partitioning the token sequence in three subsequences: an array containing lexemes of integer literals, an

array containing lexemes of float literals, and an array of lexemes of identifier tokens. Other tokens are not of interest. Note that the partitioning is stable, and so the relative order of lexemes within the arrays of each type are maintained.

2. Elements of each array are mapped to integers. Integer and float literals by parsing, and identifiers by assigning an integer to each unique string.

3. For each node which is to be associated with semantic information obtained from a lexeme, we compute an index in one of the previous arrays. As the nodes in the backing array and corresponding lexemes follow the same relative order, this can simply by done by computing an exclusive prefix sum. For example, performing the prefix sum after mapping integer literal nodes to 1 and other nodes to 0 yields for each integer literal node the corresponding index in the array of converted integers.

Although the parse tree is initially in pre-order, some of the passes transform the parse tree in such a way that it no longer is in pre-order. Furthermore, some passes even modify or remove nodes which are to be associated with lexemes: name nodes transform into function application nodes when it has an application expression, function application nodes are removed when they are part of a function declaration, among others. Both problems turn out to be easy to resolve. In the latter case, nodes are always merged or representable by another node. Function applications always take the place of a name node, and so these are also partitioned as nodes with identifiers associated. Function declarations are merged with function applications, and there cannot be any other node associated with lexemes between them in the original parse tree, and so these are also counted as nodes with identifiers. Furthermore, none of the previously applied passes invalidate the order between nodes which are to be associated with semantic information from lexemes. This means that the basic approach still works even after the tree has been manipulated. In Figure 4.15 an example computation can be seen, which extracts lexeme information from the token stream and associates the resulting values to the parse tree.

Integers and Floats

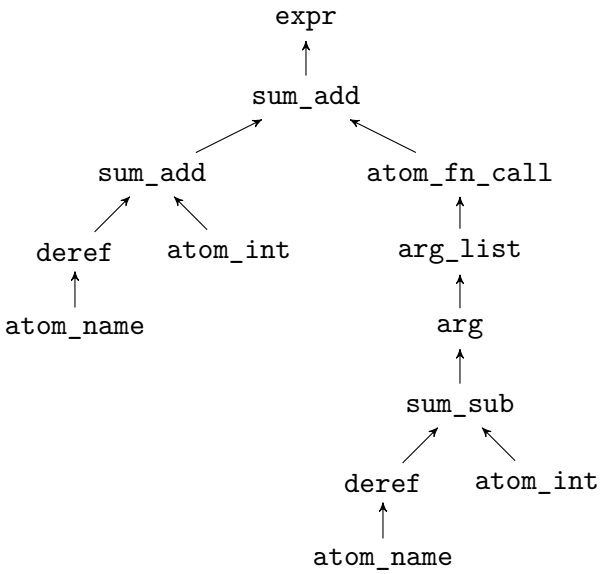
Parsing integers and floats is relatively straightforward. Integers consist of one or more of digits 0 to 9, and there is no syntax for integer literals in a base other than decimal. As integer literals are expected to be relatively short, each integer is parsed using a simple for loop, which is done for all integers in parallel. Floating points follow a similar procedure: we only allow floating points to consist of one or more digits, followed by a dot and another one or more digits. These are also parsed by a simple for loop, executed in parallel for all float literals. Note that while our implementation does not adhere to the formal standards of floating point parsing algorithms, it is sufficient for our compiler. Furthermore, no overflow checks are performed while parsing either integer or float literals.

Names

Assigning an integer value to unique names is done by first sorting and then deduplicating. There exist a number of sophisticated parallel string sorting implementations, some of

Index	Token	Partition	Offset in Partition	Data Value
0	$\langle \textbf{\textit{name}}, \texttt{a} \rangle$	identifiers	0	0
1	$\langle \textbf{\textit{plus}}, + \rangle$	–	–	
2	$\langle \textbf{\textit{int}}, \texttt{10} \rangle$	integers	0	10
3	$\langle \textbf{\textit{plus}}, + \rangle$	–	–	
4	$\langle \textbf{\textit{name}}, \texttt{b} \rangle$	identifiers	1	1
5	$\langle \textbf{\textit{lbracket}}, [\rangle$	–	–	
6	$\langle \textbf{\textit{name}}, \texttt{a} \rangle$	identifiers	2	0
7	$\langle \textbf{\textit{minus}}, - \rangle$	–	–	
8	$\langle \textbf{\textit{int}}, \texttt{20} \rangle$	integers	1	20
9	$\langle \textbf{\textit{rbracket}},] \rangle$	–	–	

(a) The tokenization and partitioning of $a + 10 + b[a - 20]$.



(b) Parse tree of $a + 10 + b[a - 20]$.

Index	Node Type	Partition	Offset in Partition	Data Value
0	`expr`	–	–	–
1	`sum_add`	–	–	–
2	`sum_add`	–	–	–
3	`deref`	–	–	–
4	`atom_name`	identifiers	0	0
5	`atom_int`	integers	0	10
6	`atom_fn_call`	identifiers	1	1
7	`arg_list`	–	–	–
8	`arg`	–	–	–
9	`sum_sub`	–	–	–
10	`deref`	–	–	–
11	`atom_name`	identifiers	2	0
12	`atom_int`	integers	1	20

(c) Relating nodes to lexemes.

Figure 4.15. Extracting lexemes and relating them with nodes.

which can be modified to also perform deduplication like the algorithm by Deshpande & Narayanan [DN13]. In order to reduce the scope of the project, however, our implementation uses a simple GPU-friendly parallel radix sort. The average program is not expected to contain very long identifiers, and so the authors believe not much performance is lost. The string equality algorithm which is used during deduplication, evaluated in parallel on all subsequent pairs of nodes after sorting, is implemented using a simple linear loop. This produces an array of masks for unique strings, strings which are different from the previous string in the sorted array. The final integer is obtained by a parallel prefix sum. Sorting and deduplicating the strings run in $\mathcal{O}(m)$ parallel time, and assigning the final integer runs in $\mathcal{O}(\lg n)$ parallel time, for a total parallel runtime of $\mathcal{O}(m + \lg n)$.

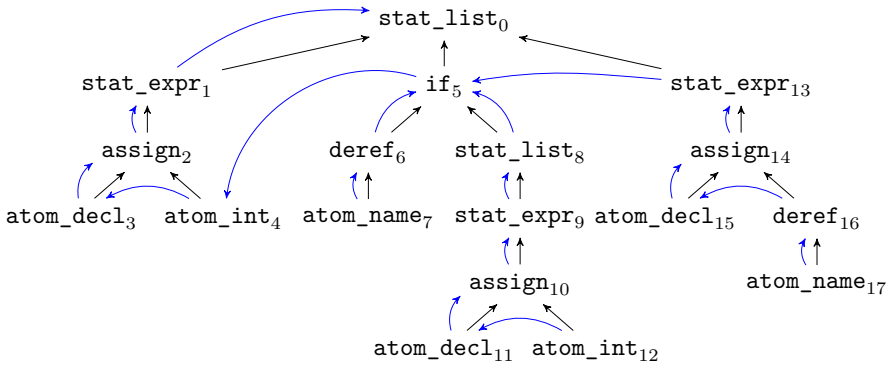
4.9.3. Variable Resolution

Variable resolution is the act of resolving places where a particular variable is referenced to the expression where the variable is declared. To this end, we introduce an additional value associated to each element of the parse tree, stored in the *resolution array*: This array stores for each node which represents a variable the index of the declaration node where the variable is stored. Recall that variable resolution should enforce the proper program structure: variables need to be declared *before* they are used, variable usage does not resolve to variables declared in sub-scopes, and so on. The basic process of resolving variables is relatively simple:

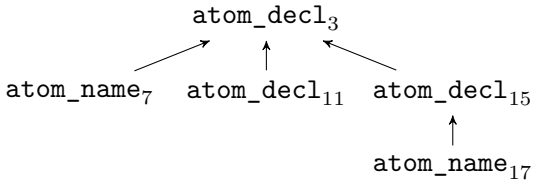
1. First, we construct a new tree representing the declaration search order: for each node an index is generated which represents the next node that needs to be searched when looking for the declaration for a particular variable, following the variable lookup rules of the programming language. Indices are generated from node-local rules. For example, if the node is the child of a statement list and the previous sibling is an expression, then the next node to search is the right-most child of that previous sibling. An example of this declaration search order is shown in figure 4.16a, where the search order is drawn in blue in the parse tree of **var** $a = 1$; **if** a {**var** $a = 1$;} **var** $b = a$;. Notice how the children of the if condition are not searched for the initialization of the latter variable.

2. The algorithm discussed in Section 4.7.2 is used to find, for each node, the nearest parent which is a declaration node. This results in a tree where all internal nodes are variable declarations, and other nodes are leaves. Figure 4.16b illustrates the resulting tree of the example shown in Figure 4.16a, after the unrelated nodes have been removed.

3. To find the declaration for a particular variable reference, we iteratively walk up the tree until we find a declaration node with the same name. Note that names are now represented by integers in the node data array, and checking if variables have the same name reduces to integer comparison. As shadowing variable names is allowed, the process stops when the corresponding declaration is found. If the root node is reached, there is no corresponding declaration to a particular variable reference, in which case the program is invalid.



(a) The declaration search order shown in blue in the original parse tree of the expression. Subscript shows node index.



(b) The same declaration order after unrelated nodes have been removed. Subscript shows the node index again.

Figure 4.16. The declaration search order of the expression **var** $a = 1$; **if** a {**var** $a = 1$;} **var** $b = a$; in the original tree and with unrelated nodes removed.

4.9.4. Function Resolution

Function resolution works by filling entries of function application nodes in the resolution array with the index of the corresponding function declaration node. Note that since function application nodes and variable referencing nodes are mutually exclusive, the array filled by the process discussed in the previous subsection can be used here. In contrast to variable resolution, functions declarations do not follow a structured order. All function are defined on global scope, and code in functions may call any other defined function. The only restriction is that there may not multiple functions declared with the same name. Function resolution is performed as follows:

1. First, we check that functions have unique names by extracting the identifier integer associated to function declaration nodes, and compute a histogram using the integer as index and one as value. This yields an array which for each identifier integer contains the number of function declarations with that name. If any of the array entries are equal to or higher than two, the program is determined invalid. The histogram is computed using Futhark's `reduce_by_index` function, which is described by Henriksen et al. [Hen+20].

2. Next, we compute an array which maps identifier integers to node indices of the functions declared using the corresponding name.

3. Finally, for each function application node in parallel a lookup is performed in the previously computed array to fetch the node index of the function declaration. If

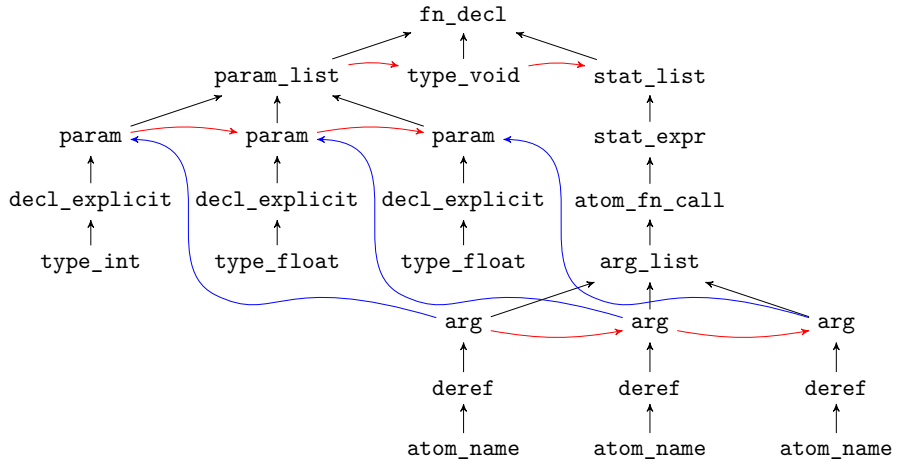


Figure 4.17. Resolving arguments to parameters in the parse tree of **fn** $a[b$: **int**$, c$: **float**$, d$: **float**$]$: **void** $\{\ a[b, c, d];\}$. Blue edges show the argument resolution, red edges shows the next sibling.

the resulting node index is invalid, there was no function declaration with the corresponding name, and the program is again determined invalid.

4.9.5. Argument Resolution

During argument resolution, each argument node of a particular function application is augmented with the index of the parameter node in the corresponding function declaration. This is again stored in the resolution array. Argument nodes are mutually exclusive from function application nodes and variable references, and so the corresponding element in the resolution array is as of yet not used. After the list flattening transformation discussed in Section 4.8.3, all parameter nodes of a particular function declaration are the child of the same parameter list node. This means that the indices in the sibling arrays form linked lists of parameter nodes, and this is also the case for argument nodes. To obtain the argument resolution, we apply the parallel linked list element matching algorithm described by Hillis & Steele [HS86]. This algorithm runs in logarithmic parallel time, and handles all argument lists of the program simultaneously. See Figure 4.17.

The complete process is as follows:

1. First, the initial element of argument lists and parameter lists is written to the location of the function application node and function declaration node respectively, in a temporary array.

2. The index resolved in the function resolution pass discussed in Section 4.9.4 is used to fetch the index of the first parameter node corresponding to the first argument node. These two indices are used to initialize the linked list matching process.

3. After the arguments have been resolved, the program is checked for validity with regards to the number of parameters in the function definition and the number of

arguments in the function application.

In the latter step three cases need to be considered:

1. If there are more arguments in the argument list of the callee than there are parameters in the parameter list of the function declaration, then one of the argument nodes will not be paired up.

2. If there are less arguments in the argument list of the callee, the next sibling of the last argument will not be invalid.

3. Both of the previous cases rely on the fact that there is at least one argument node and parameter node in the respective lists. To handle cases where either have zero, we verify that either both the application and declaration have a nonzero amount of arguments and parameters, or that both have no arguments and parameters.

4.9.6. Type Analysis

The type system of the programming language implemented by our compiler is relatively simple. Every expression node produces a value of some data type, which has to be known statically and is inferred from and must satisfy node-local rules. For example, both children of an addition node must produce the same data type, and the resulting data type will be the same as that of the children. During type analysis, the compiler associates each node with such a data type, and verifies that they follow the program rules. The only form of type coercion that is allowed in our programming language is the implicit conversion from storage locations to their actual value, which is handled in the pass described in Section 4.9.1. Besides integers and floats, there are four other, internal, data types:

1. The invalid type, to indicate that a node does not produce a value or to indicate that the resulting data type of an expression could not be computed.

2. Void, only used to indicate that a function does not return a value and as resulting data type of return statements without expression.

3. Integer and float references, collectively referred to as reference types. These are the resulting type of an expression which yields a writable storage location, like referencing a variable, and are also used as data type of variable declaration nodes. Internally, variables may only hold reference types.

The type analysis process consists of two steps. First, we assume that the semantic structure of the program is correct, and use that to determine a data type for every node. Only in the second step we verify that the data types obtained in the first step satisfy the program rules. To determine the resulting data type of a particular node, five cases need to be considered:

1. For some node types, the resulting data type is known ahead of time. For example, bit wise operators always produce integers, and float literals always produce floats.

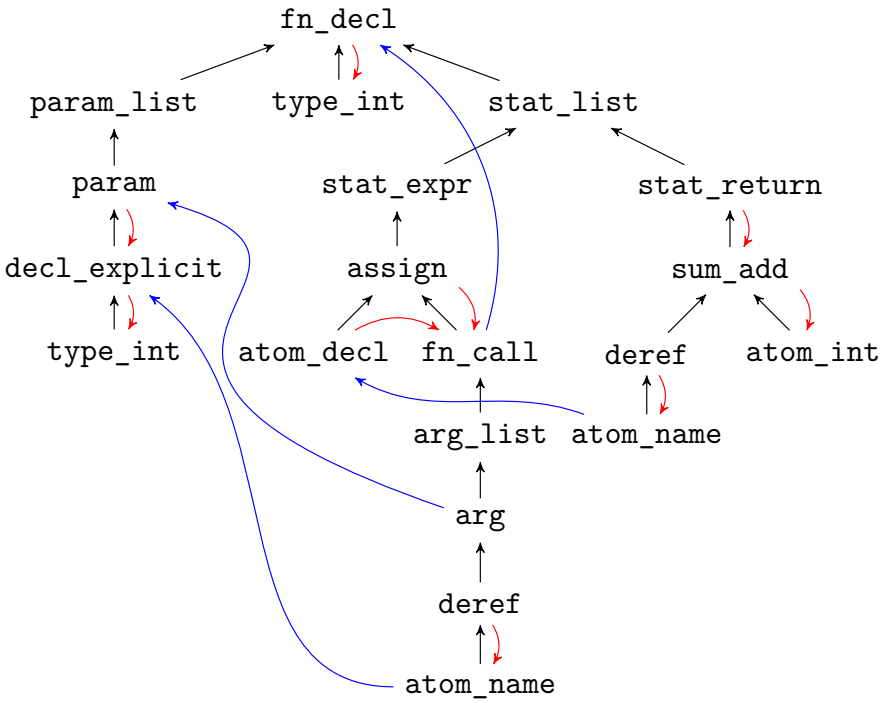


Figure 4.18. Type resolution order for the program **fn** $a[b$: **int**$]$: **int**$\{$ **var** c = $a[b]$; **return** $c+1$; $\}$. Red arrows indicate that a data type is obtained from a child or sibling, blue arrows indicate data types obtained through the result of variable, function or argument resolution.

2. Some nodes are the parent of a type expression, such as for example function declarations, ascriptions, and casts. In these cases, the resulting data type is given by the type expression.

3. For other node types, the resulting data type is determined by the data type of the children. For example, assuming the program is correct the resulting data type of an addition operation is the same as the resulting data type of either child.

4. In some special cases, the resulting data type is determined by that of another sibling. The only such case are variable declarations without an explicit type, as the resulting data type is given by the right-hand side of the parent assignment expression.

5. Finally, for remaining node types, the resulting data type is determined by another node that is not one of its direct children or siblings. These cases include variable usage, where the resulting data type is given by the corresponding declaration. The same holds true for function applications, where the resulting data type is given by the return type of the corresponding function declaration.

Note that in the case that resulting data type of a node is determined by another node, the type may also undergo a modification. For example, the child of a dereferencing node is a reference type, while the result is the dereferenced version of that type.

The final type analysis process is as follows:

1. First, we fill the data type array with data types which can be computed in constant time: data types determined from the node type, and data types determined from type expressions. This corresponds with cases 1 and 2 of the previously discussed cases. Other entries are set to invalid.

2. Next, we construct an array which for each node with invalid data type contains the index of a node from which the data type is to be determined, according cases 3, 4 and 5. These indices may for example consists of the index of a child, sibling or another node. The resulting array forms a forest of trees, referred to as the *data type resolution tree*. Note that in a semantically correct program data types of root nodes have already been determined, in the previous step. See the example in Figure 4.18.

3. We then account for nodes which modify data types by adding or removing references. Each node is mapped to the amount of references it adds or removes, after which a prefix sum is performed as described in Section 4.7.3. The result holds for each node the amount of references that is added or removed to the data type.

4. The data type resolution tree may contain cases where a node without a data type points to another node without a data type. We resolve this similar to how variable resolution is performed. The logarithmic algorithm described in Section 4.7.2 is used to find the nearest parent which does not have an invalid data type.

5. The initial data type can now be fetched from parent in the data type resolution tree, and is combined with the array obtained in step 3 by adding or removing the corresponding amount of references to produce the final data type. Note that if the resulting amount of references is not 0 or 1, or the data type does not support the amount of references, the invalid type is produced.

6. Next, the resulting data types are checked for correctness. This is done by evaluating the node-local rules for all nodes of the tree in parallel. Depending on node type, this includes for example verifying that the resulting data type is not the invalid type or void type, verifying that another sibling has the same data type, verifying that the parent has the same data type, and so on.

7. Finally, we check that the type of return statements matches the type of the function declaration they are part of. This is done simply by finding the ancestor function declaration node for every return statement node using the logarithmic algorithm described in Section 4.7.2, and verifying that the associated data types of the two nodes are equal. Note that return statements without expression are assigned the void type.

Note that all of the substeps run in logarithmic or constant parallel time, and so the total parallel runtime of the type analysis process is also logarithmic.

4.9.7. Function Convergence

We implement a relatively simple form of function convergence analysis. Every code path through a function must eventually result in a return statement, and we assume that every

loop eventually terminates. The latter does not forbid infinite loops, but simply requires that the programmer still writes a return statement after them. The precise set of rules to determine whether a program is valid in this regard is as follows:

- The root statement of all function declarations whose return type is not void must eventually return.
- Return statements return a value, namely the result of the associated expression. At this point, the type analysis is already performed and we know the expression is of the right type.
- In order for a statement list to return a value, at least one of its children must return a value.
- In order for a conditional statement *with alternative* to return a value, both the consequent and the alternative must return a value.
- While loops and conditional statements without alternative are assumed to not return a value: the condition may hold false in which case the corresponding code block is not evaluated, and any return statement in it is never reached.
- Expression nodes also never return a value, as the only way to return a value from a function is by a return *statement*.

In order to determine whether the program is correct, we transform the parse tree to a boolean expression tree. Every node is transformed into an operator or value which implements the appropriate rule for that node. The result of this expression then indicates whether all non-void functions in the program are guaranteed to terminate.

- Return statements transform to a leaf node with value `true`.
- Statement lists transform to an `or` node, as either of its children should yield `true`.
- Conditional statements with alternative transform to an `and`-node, as both children should evaluate to `true`. Note that the child node representing the condition should not be considered.
- While loops and conditionals without alternative are transformed to a leaf node with value `false`.
- Function declaration lists also transform to an `or` node, as the property must hold for all functions in the program.
- The function declaration nodes, which sit between function declaration list nodes and the root statement of the function, either inherit the value from their child if the function does not return void or are transformed to leaves with value `true` if it does.
- All other nodes transform to a leaf node with value `false`.

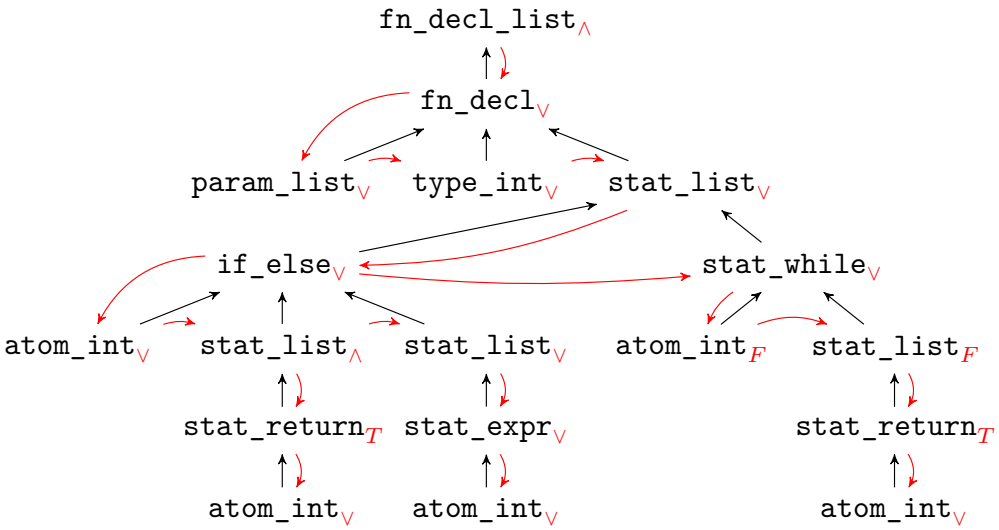
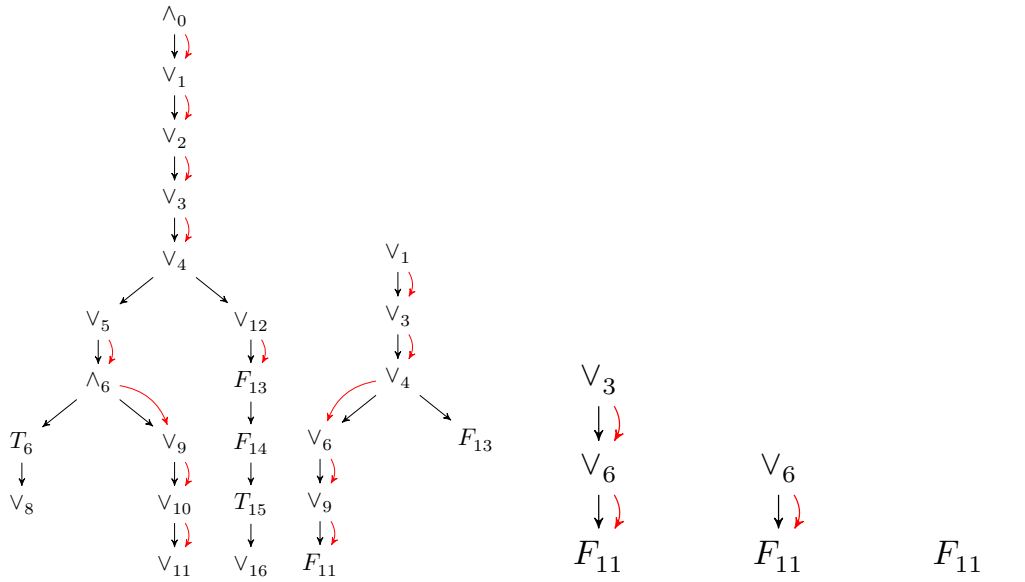


Figure 4.19. Converting a parse tree to a boolean expression to determine whether it converges.

In reality, the conversion is done a slightly different manner, as the parse tree is converted to a binary tree. The left child of a node in the resulting binary tree is formed by the first child of the original node, and the right tree is formed by the original node's next sibling. Note that when a node has multiple children, the first child has the second child as sibling, and so the second sibling will also be considered. When the node in the parse tree does not have children or siblings, the respective child is omitted, and the corresponding value will be assumed to `false`. In these cases, an `or`-node is typically used to pass the value up to the parent. See the tree in Figure 4.19. Note that in this version, nodes which simply pass up their values are encoded as `or`-nodes. Furthermore, nodes which would otherwise yield leaf nodes with value `false` are also mapped to `or`-nodes. This allows the conversion to be much simpler. For example, by encoding the left child of the `if_else` node in Figure 4.19 as an `or`-node, it simply passes up the value from the consequent of the condition statement. Finally, internal nodes may also compose of `true` or `false` values, in which case the value of the child will be ignored during evaluation.

We store the tree in a top-down fashion, where every node has a type and the indices of the two children. To evaluate the tree in parallel, we repeatedly try to replace all nodes in parallel for which we can compute a result, until the value for the root node is obtained. In the most naive version of this algorithm, we only replace nodes for which both children have known values so for which we can compute a concrete result. This works out fine when the tree is balanced. In that case, every iteration two leaf nodes are replaced by one leaf node. As the number of leaf nodes in a binary tree is equal to the number of internal nodes plus one, this would halve the size of the tree every iteration, and so we can compute the result in logarithmic parallel time. When the tree is not balanced, as is often the case, this method becomes slow. In the worst case, when the tree forms a linked list, we can only replace one every iteration and yield linear parallel time.

We obtain a faster algorithm by recognizing that internal nodes which have only one known child can also be replaced, by applying boolean identities. For example, the subtree which represents `true` $\vee\, x$, where x is an arbitrary subexpression, can be replaced by a leaf with



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Step 5

Figure 4.20. The evaluation process of the binary tree corresponding to the parse tree in Figure 4.19. The red arrows show which nodes are replaced in the step, and the subscript shows the index of the node.

value `true`. Furthermore, the subtree which represents `true` $\wedge\, x$ can be replaced by just x. By this extension, we can remove all leaf nodes during an iteration, and so the size of the tree is halved regardless of its structure. This means that we can compute the value of the root node in $\lceil \lg n \rceil$ iterations, where n is the amount of nodes in the original tree, which is equal to the number of nodes in the parse tree. As the work within an iteration can be parallelized over all nodes, the total parallel runtime of this method is $\mathcal{O}(\lg n)$. See Figure 4.20 for the intermediary trees after every iteration while computing the boolean expression corresponding to the parse tree in Figure 4.19. Note that while replacing, we copy the child node including the indices of its children *at the start* of the iteration.

4.10. AST Construction

In the final pass of the front end, the abstract syntax tree is constructed from the parse tree. This process consists of transforming the data structures used by the front end part of the compiler, which include the parents, node types, node data, data types, resolution and sibling arrays into the format required by the back end as described in Section 3.3. The back end expects the abstract syntax tree in a structure similar to how the parse tree is stored in the front end. That is, a set of arrays each of which associated to a single node property, and where a node consists of the elements at a particular index in the arrays. The abstract syntax tree consists of the following arrays:

- The node types, parents and data types arrays, which already contain the appropriate values.

- An array which holds for each node its depth in the abstract syntax tree. This is computed by performing the depth computation algorithm discussed in Section 4.7.3.

- An array associating each node with its *child index*, a number which indicates the offset of the node in the list of children of its parent. This is computed by performing the same depth computation algorithm, but this time over the previous sibling array instead of the parent array.

- The node data array, which holds a different type of value based on the node type it is associated with.

Elements of the node data array holds the following values, based on the node type:

- Elements which are associated with integer- and float literal nodes hold their binary value. This was already the case in the original data array, as obtained by the process discussed in Section 4.9.2.

- Function declaration nodes are associated with an integer identifier for the function, and the element associated to function applications should hold the same integer as the called function. Each identifier is assigned sequentially and starts from 0. We compute this value by mapping each function declaration node in the original node array to 1 and all other nodes to 0, and performing an exclusive prefix sum. After this operation, indices corresponding with function declaration nodes hold the appropriate identifier. Identifiers for function application nodes are fetched from the function declaration node via the resolution array, as this holds for each application node the index of the corresponding declaration node.

- Variable declaration nodes and corresponding variable usage nodes gain a similar sequential identifier, starting from 0 for the identifier in each function. This is computed by observing that none of the passes move any declaration nodes outside of the region between the parent function declaration node and the declaration node of the next function, and computing an exclusive *segmented* prefix sum. This variant of prefix sum accepts a special value which resets the count of the prefix sum at particular indices. Variable declaration nodes increase the counter by one, and function declaration nodes reset it. The identifiers for variable usage nodes are again fetched via the resolution array.

- Similar to function and variable nodes, parameter and argument nodes gain a sequential integer starting from 0. This is computed in the same way as the integer for variable declaration nodes, although in this case, the identifier should be counted separately for parameters of different types.

The order of the nodes of the tree in the arrays backing the abstract syntax tree must follow post-order traversal. This is obtained by first constructing a linked list of nodes, each of which holding the index of the previous node in the post-order traversal. If the node is the first node of its parent, then the previous node in post-order traversal is the parent. Otherwise, the previous node is the right-most descendant of the previous sibling, obtained by the algorithm discussed in Section 4.7.5. The index of each node in the final, ordered version of the tree is obtained by computing the depth of each node according to

the previously obtained linked list using the algorithm outlined in Section 4.7.3.

The back end also expects a separate array which holds for each function declaration the number of local variables which are declared in it. This array is indexed by the function identification integers obtained previously, and the corresponding number of variables is found by extracting the value in the variable declaration identifier counter just before it is reset.

The abstract syntax tree is now in a form suitable for further processing by the back end part of the compiler. This concludes the lexical analysis, parsing and semantic analysis stages.

5. Experiments

In order to get an indication of the runtime efficiency of the techniques described in the previous chapters, we perform a set of empirical experiments. First and foremost, we would like to know the runtime characteristics of the compiler itself. To that end, we measure the time it takes for individual passes to be performed while the compiler is processing input of various sizes.

Because we have designed the compiler for a custom programming language, there is no suitable baseline implementation available to compare GPU performance against. Writing a naive single-threaded implementation would yield an unfair comparison, as a proper baseline implementation would not be naive. In order to yield a good comparison, the CPU implementation would additionally have to be optimized for all the resources the host machine has to offer, including usage of all available cores. Of course, Futhark supports single- and multi-core CPU targets besides GPU targets, however, many of the passes outlined in Chapter 4 have been designed with the high memory bandwidth that GPU environments offer in mind. GPU memory bandwidth is often much higher than CPU bandwidth, and although the individual cores of CPUs are often more efficient than individual GPU cores, we hypothesize that the CPU version will trash on memory accesses when many cores are used. All multi-core CPU-based experiments are performed with 1, 4, 16, and 64 cores where applicable.

We can still compare some individual stages of the compilation process with other work. In particular, we compare the lexical analyzing and parsing methods with those described by Barenghi et al [Bar+15]. The implementation of this work is called PAPAGENO, which is written by the original authors and publicly available on GitHub[1]. This work features parallel lexical analyzer and parsers for three languages: Lua[2], JSON[3] (JavaScript Object Notation), and simple arithmetic expressions. We compare this work on two fronts:

- First, we implement a PAPAGENO parallel parser for the grammar of our programming language. Note that the parsing method used by this research suffers from similar but different limitations as $LLP(1,1)$ grammars, and so does also not accept the full grammar of our programming language. As it turns out, however, the transformations outlined in Section 3.2.1, along with some other transformations which do not change the accepted language, is enough to get a language accepted by PAPAGENO. For this reason, when comparing the parsers for this language, we do not include the additional restructuring and verification passes. The main parsing function of a parser generated by PAPAGENO returns a parse tree, and so we do include the parse tree construction pass. Barenghi et al also describe a method for parallel lexical analysis, but this method was not applied to our language. The

[1] `https://github.com/PAPAGENO-devels/papageno`
[2] `https://www.lua.org/`
[3] `https://json.org`

lexical analyzer is instead generated by Flex.

- The process to obtain the lexical analyzer and parser for our programming language are in principle generic, and can also be used to parse other inputs. To further investigate the runtime characteristics of these stages, we also write a JSON lexical analyzer and parser, and compare this with the versions written by the authors of PAPAGENO. Note that this includes a parallel lexical analyzer. The grammar of JSON is also not $LLP(1,1)$, and because the PAPAGENO implementation only accepts valid JSON, a number of restructuring and validation passes are employed to ensure a fair comparison.

In both cases, we used the logarithmic PAPAGENO recombination strategy. Additionally, we perform a similar set of benchmarks on simdjson[4] [LL19]. Modern CPUs feature SIMD instructions not unlike those found in GPU architectures, but on a smaller scale. Simdjson effectively uses these to accelerate parsing of JSON documents. Note that simdjson does not take advantage of multi-core CPUs, and only uses a single thread to parse a document.

All runtime measurements are obtained by querying for timestamps on the CPU, using the c++ `std::chrono::high_resolution_clock`. Care is taken that asynchronous GPU and CPU work is synchronized before a timestamp is collected in order to ensure accurate measurement. Note that this introduces a small amount of overhead, both because recording timestamps takes a little time itself and since without synchronization work between multiple stages may be executed simultaneously. In order to reduce noise, we repeat every individual experiment 30 times.

During experimentation, we also gather a number of statistics dependent on the input which may be interesting or help explain some results. These include statistics about the intermediary and generated trees, and memory usage during processing. Note that Futhark does not offer very intrusive memory debugging, but does provide a debug trace which includes allocation information. This information is also not completely accurate, but should serve as a rough indication of the amount of memory used during processing.

The remainder of this chapter is structured as follows. In Section 5.1, we describe the setup of the computers on which the experiments were performed. In Section 5.2, the experimentation inputs are discussed. Finally, in Section 5.3, the results of our experiments are presented.

5.1. Benchmark Machines

Two different machines are used to gather the results of our experiments, where one is specialized in CPU-intensive computations, and the other is intended for GPU-intensive work. The former is equipped with with two AMD EPYC 7601 processors, each running at 2.4 GHz and with 32 physical and 64 logical cores, and 1 TB of system memory. Each EPYC 7601 has a maximum theoretical memory bandwidth of 170 GB/s. The latter has two Intel Xeon Silver 4214R processors, each running at 2.4 GHz with 12 physical and 24 logical cores, 256 GB of system memory. It also features two Nvidia GeForce RTX

[4] `https://simdjson.org`

Data set	Size
`twitter_api_response`	15.2 KB
`spirv.core.grammar`	423 KB
`gsoc-2018`	3.33 MB
`refsnp-chrMT`	66.0 MB
`refsnp-other-100K`	442 MB

(a) Statistics of JSON test data.

Data set	Size	Lines	Functions
`pareas-0`	5.17 KB	183	51
`pareas-1`	596 KB	15 190	23
`pareas-2`	3.23 MB	82 608	24
`pareas-3`	20.6 MB	538 477	63
`pareas-4`	183 MB	4 698 065	744

(b) Statistics of Pareas test data.

Table 5.1. Statistics of benchmark data files.

3090 GPUs, each with 24 GB of video memory and 10 496 shader cores, though only one of the two GPUs is used for our experiments. Each GPU boasts a memory bandwidth of 936 GB/s. Both machines run CentOS 7, powered by Linux kernel version 5.4.126. All executables are compiled using GCC version 10.2.0 in release mode (`-O3`), and are statically linked to the same version of libstdc++. For GPU experiments we use the Futhark CUDA back end, which uses CUDA version 11.2.

5.2. Test Data

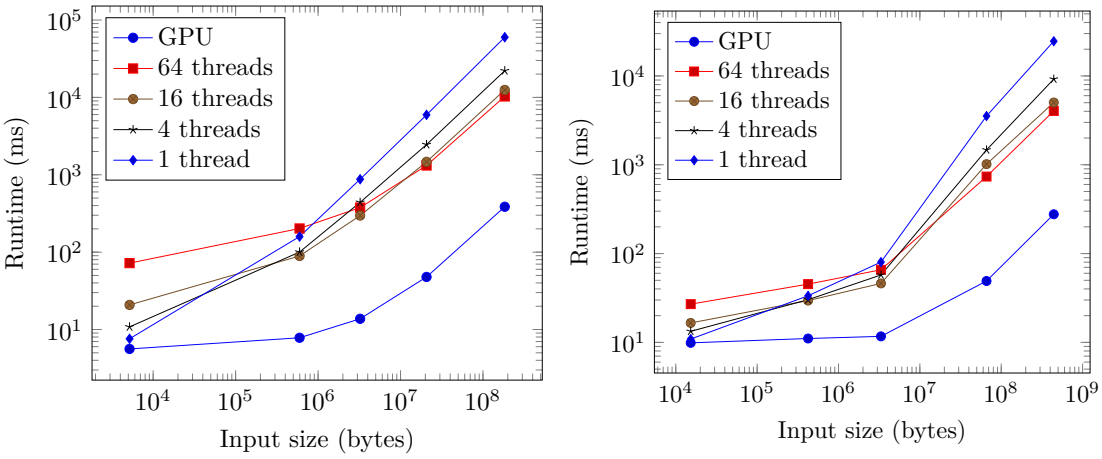
In the case of JSON, we perform our experiments on five different real-world data sets of various sizes:

- The response of a call to the Twitter API, obtained from the simdjson [LL19] test data repository[5].
- SPIR-V machine-readable grammar definitions, obtained from the Khronos SPIR-V headers repository[6].
- Google Summer of Code 2018 data, also obtained from the simdjson test data repository.
- Two data sets of single nucleotide polymorphisms, obtained from the Single Nucleotide Polymorphism Database (dbSNP) [She+01]. Data was retrieved from the dbSNP the ftp repository[7], originally uploaded on 2021-05-25. We use `refsnp-chrMT.json` and the first 100 000 entries of `refsnp-other.json`.

Note that the two data sets obtained from dbSNP were originally in JSONlines[8] format, where each line contains a separate JSON document. This is not accepted by our implementation, and so these files were re-processed into valid a JSON document by constructing a single array containing the individual documents of the original data set. Some statistics about these data sets are outlined in Table 5.1a.

Since we have designed our own programming language, which is also quite limited, there

[5] `https://github.com/simdjson/simdjson-data/tree/a5b13babe65c1bba7186b41b43d4cbdc20a5c470`
[6] `https://github.com/KhronosGroup/SPIRV-Headers/tree/8aec5fcf95cc69c54321007ba959ff6b1954a0dd`
[7] `https://ftp.ncbi.nih.gov/snp/latest_release/JSON/`
[8] `https://jsonlines.org`



(a) Throughput of the compiler.

(b) Throughput of the JSON parser.

Figure 5.1. Throughput in bytes per millisecond when compiling source on different architectures with different sizes.

is not a sufficient amount of source code available which represent real-world use cases. Furthermore, the simplicity of the language also makes source translation complicated. For this reason, we opt to randomly generate source code which we believe will yield results indicative of the performance on hypothetical real-world source code. By tweaking generation parameters, we gain five different data sets. All files are syntactically and semantically correct. See the statistics of these data sets in Table 5.1b.

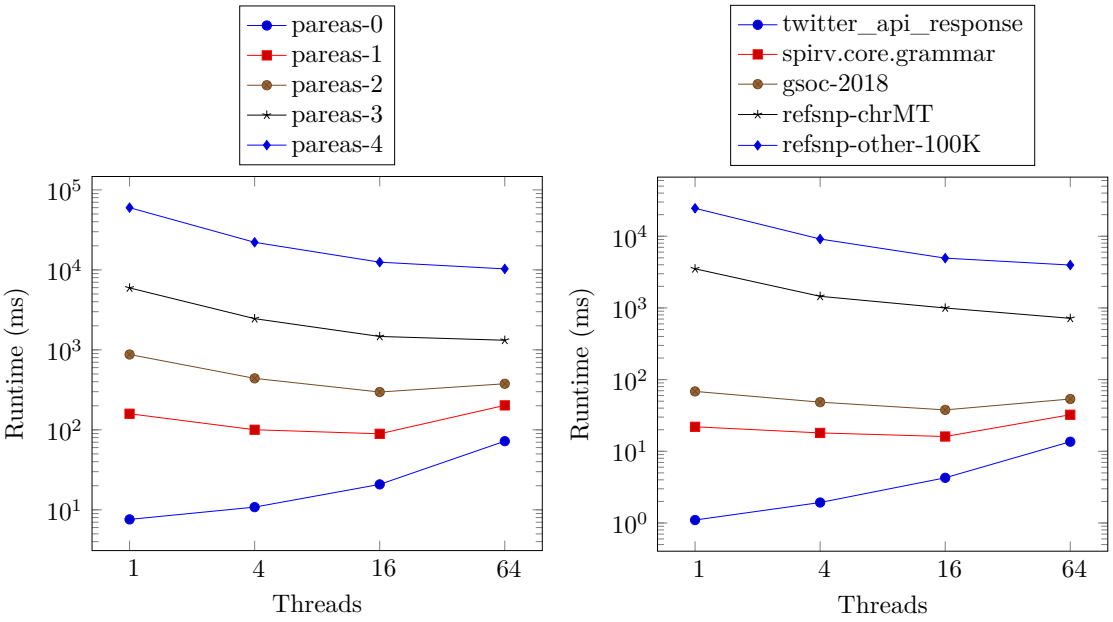
5.3. Results

In the following sections, we give an overview and discuss some of the results we obtained from our experiments. See Appendix C for detailed listings of all gathered results, including standard deviations.

5.3.1. Throughput

Figure 5.1 relates the input size of the data set used in each experiment with the amount of time it takes to process it into an abstract syntax tree, for different Futhark back ends. It is immediately evident that as suspected, running the compiler on the GPU is much faster than when running on the CPU. Again, this can be attributed to how the individual passes of the compiler are written. A compiler written for CPU targets would try to keep the working set as small as possible, for example by performing all operations that need to be performed on one node first, before processing the next. In our compiler, however, there are many relatively small passes which all process the entire tree. This creates a relatively large amount of memory pressure, and as discussed in Section 2.1, GPUs are more easily able to deal with this style of processing than CPUs.

From Figure 5.1, we can also see that when sufficient work is available, scaling appears to



(a) CPU scaling of the compiler.

(b) CPU scaling of the JSON parser.

Figure 5.2. Scaling effects of increasing the amount of threads when performing experiments on the CPU.

be linear on CPU targets. This is to be expected, as many of the parallel primitives used in our compiler can be executed in linear time sequentially for each thread, after which the logarithmic strategy is only used to combine the results of the individual threads. For example, in the case a reduction, each thread can linearly compute the result for the assigned subsequence, after which only relatively a small amount of work is required to compute the final results. This last part can also explain why the linear scaling does not extend to the smaller data sets, especially when a larger amount of threads is used. While the processing of each subsequence can happen without synchronization, combining the final results does require explicit synchronization. When a large amount of threads is used, the overhead of this synchronization work starts to outweigh the benefits of having more threads to distribute work over.

This linear scaling is not immediately evident for the experiments involving the GPU back end, also shown in Figure 5.1. This can be explained by the amount of threads that a GPU has. This causes the sequential part that each shader processor is assigned to be relatively small, and results in that a relatively large amount of intermediary results need to be merged.

5.3.2. CPU Scaling

To further investigate the performance of the compiler on CPUs, Figure 5.2 compares just the results of the CPU experiments with one another. From this figure we observe that in the case of the larger inputs, for both the compiler and the JSON parser, more threads generally means faster execution. This is of course expected of a fully parallel compiler, however, we also observe that the speedup does not scale very well. For example, the total

Data Set	Size	Uploading	Lexical Analysis	Parsing	Building Parse Tree	Restructuring	Semantic Analysis
pareas-0	5.17 KB	7.48 MB	114 KB	188 KB	98.3 KB	92.9 KB	59.7 KB
pareas-1	597 KB	8.07 MB	13.1 MB	21.8 MB	12.6 MB	10.4 MB	5.15 MB
pareas-2	3.24 MB	10.7 MB	71.2 MB	151 MB	50.3 MB	56.6 MB	28.2 MB
pareas-3	20.7 MB	28.2 MB	455 MB	719 MB	403 MB	354 MB	178 MB
pareas-4	184 MB	191 MB	4.04 GB	6.05 GB	3.22 GB	3.18 GB	1.59 GB

(a) Results of the compiler.

Data Set	Size	Uploading	Lexical Analysis	Parsing	Building Parse Tree	Restructuring
twitter_api_response	15.3 KB	34.9 MB	336 KB	93.5 KB	24.6 KB	30.9 KB
spirv.core.grammar	423 KB	35.3 MB	9.3 MB	5.07 MB	1.57 MB	1.39 MB
gsoc-2018	3.33 MB	38.2 MB	73.2 MB	5.65 MB	1.57 MB	1.8 MB
refsnp-chrMT	66 MB	101 MB	1.45 GB	816 MB	201 MB	294 MB
refsnp-other-100K	443 MB	478 MB	9.74 GB	6.06 GB	1.61 GB	2.01 GB

(b) Results of the JSON parser.

Table 5.2. Breakdown of the maximum allocated bytes per major stage.

runtime when using four cores is less than half of that when using only a single core in the case of the largest two inputs, but when comparing 64 cores with 16 cores the runtime is only marginally smaller. This could have a number of root causes. First, the scalability of our algorithms could be not as well as initially assumed. Second, synchronization and scheduling overhead may start to outweigh the improvements of more threads. This hypothesis is supported by the cases of smaller input files, where we see a negative scaling with the amount of threads used. Finally, the limited memory bandwidth capabilities of the CPU may be hindering more efficient scaling.

5.3.3. Initialization Time

During experimentation, it became evident that one major drawback of processing input on the GPU not shown in the previous plots was the initialization time. In contrast to many CPU architectures, the machine code of GPU architectures is not standardized and often proprietary. This results in that programs often ship kernels in the form of source code or an intermediary format, which has to be compiled at runtime by a compiler integrated with the GPU API. While there sometimes also exist ahead-of-time compilers for hardware platforms, Futhark generates source code for both OpenCL and CUDA targets, which is compiled when the Futhark context is initialized. The implementation of our compiler features many distinct passes, all of which have to be compiled, and so we report an average initialization time of 4.6 seconds. The JSON parser does not have as many passes, but still requires an average of 2.0 seconds to be initialized. In contrast, the multi-core back ends take on average less than 2 milliseconds to initialize, as the concrete implementations for these can be compiled ahead of time along with the rest of the host program.

5.3.4. Memory Usage

Table 5.2 shows a rough breakdown of the total number of bytes that is allocated during different stages of the implementations. Note that this includes new allocations made during that stage, and already existing allocations from data that persists from the previous

Data Set	Size	Tokens	Initial Nodes	After restructuring binary operators	Final Nodes
pareas-0	5.17 KB	1 357	6 189	1 749	853
pareas-1	597 KB	116 870	690 209	168 389	80 158
pareas-2	3.24 MB	640 202	3 773 477	921 509	439 044
pareas-3	20.7 MB	4 051 688	23 621 648	5 785 412	2 764 364
pareas-4	184 MB	36 024 031	211 942 480	51 769 852	24 658 753

(a) Numbers of tokens and nodes after the initial parse, after restructuring binary operators, and after all other transformations of the compiler.

Data Set	Size	Tokens	Initial Nodes	Final Nodes
twitter_api_response	15.3 KB	1 439	1 625	714
spirv.core.grammar	423 KB	57 223	73 215	28 613
gsoc-2018	3.33 MB	75 841	94 803	37 922
refsnp-chrMT	66 MB	12 710 519	15 465 337	6 290 756
refsnp-other-100K	443 MB	87 223 028	105 848 178	42 989 691

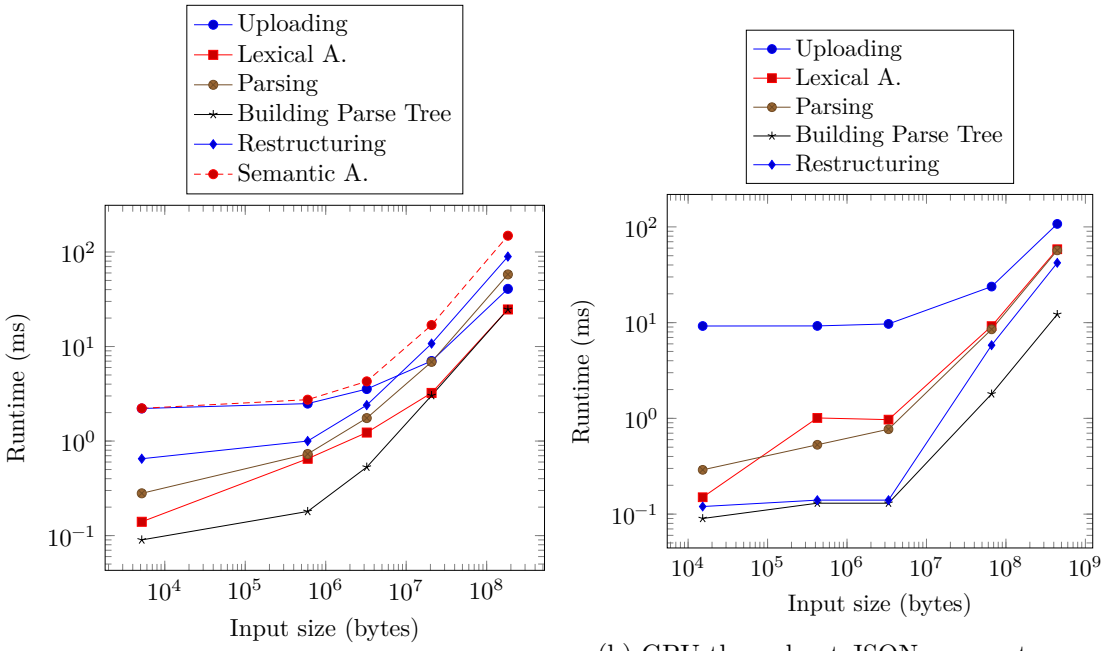
(b) Number of tokens and number of nodes before and after restructuring in the JSON parser.

Table 5.3. Numbers of tokens and nodes during various stages of the implementations.

stages. There are several things of note. First, we see that there is a difference between the size of the data set and the actual amount of bytes that is sent to the GPU during the upload stage. This is because the offline precomputed data, the tables generated by the lexical analyzer and parser tools, are also uploaded during this stage. In the case of the compiler, there is about 8 MB of precomputed data, and in the case of JSON there is about 35 MB. This data mostly consists of the merge table used in the lexical analyzer. In the case of the compiler, there are 1 927 unique unary transition functions. As discussed in Section 4.1, each unary transition function is encoded in 16 bits, and so this yields about 7.5 MB of precomputed data. In the case of the JSON parser this table is much larger, and has 4 176 unique unary transition functions, yielding 34.9 MB of precomputed information.

From Table 5.2, we can also see that there is a large amount of memory that is allocated during the lexical analysis stage in both implementations, up to 20 times the size of the input. This can be explained by the implementation of the lexical analyzer. After the simulation of the state machine has been computed, each element of the input array has a state associated to it in another array. States which produce tokens are associated a start and end index, yielding another two elements associated with each input character. As these indices are 4 bytes each, this yields $4 + 4 + 2 = 10$ bytes per input character. During filtering of states which do not produce a token, these intermediary results need to be duplicated, yielding a total of 20 bytes per character.

Finally, we can see in Table 5.2 that a similar large amount of memory is allocated during parsing, though relatively less so in the case of the JSON parser. This is explained by the grammar of our programming language, and the characteristics of the parser. As discussed in Sections 3.2.1 and 4.8.1, binary operators cannot be expressed natively by the parser. The method which is used to work around this issue yields a node for every class of binary operators, however. There are about 10 classes of operators in our programming language, which causes a large amount of superficial nodes to be generated. Indeed, when we compare the initial amount of nodes with the amount of nodes after removing these superficial nodes using the process described in Section 4.8.1, shown in Table 5.3a, we see that only about a quarter remains. In the other passes, the amount of nodes is halved again.



(a) GPU throughput of major compiler stages.

(b) GPU throughput JSON parser stages.

Figure 5.3. GPU Throughput of the compiler and JSON parser, broken down by major stage.

5.3.5. GPU Throughput Breakdown

Figure 5.3 breaks down the scaling of some individual major stages on the GPU target. In the case of the compiler, of which the results are shown in Figure 5.3a, we can see that almost all stages exhibit the same type of scaling observed for the total runtime. Again, when the input is small, a relatively large amount of time spent combining the results of the individual shader processors. When the input is larger, however, almost all stages exhibit linear scaling in runtime with the size of the input.

When comparing the results from the compiler shown in Figure 5.3a with the results from the JSON parser in Figure 5.3b, we can see that the latter shows much less nice scaling. In this case, we see that for many stages, linear scaling is only achieved after the third smallest data set, `gsoc-2018`. We believe that this can be explained by the fact that the JSON files used in this experiment come from real-world sources, while the experiments involving the compiler used synthetically generated sources. In the latter case, the structure of sources of different sizes are expected to be relatively the same. In the former case, both documents have a different purpose, and as such are structured differently. Indeed, when querying the number of tokens and the number of nodes in the final parse tree, shown in Table 5.3b, we can see that even though `gsoc-2018` is almost an order of magnitude larger than `spirv.core.grammar` it only has a relatively small amount more tokens and nodes. It turns out that `gsoc-2018` features relatively long strings, which causes the size of the file to be much larger relative to the amount of nodes. This shows us that even though individual passes may scale well on generic input, some variation is still to be expected when processing real-world input.

5.3.6. Comparison with Simdjson and PAPAGENO

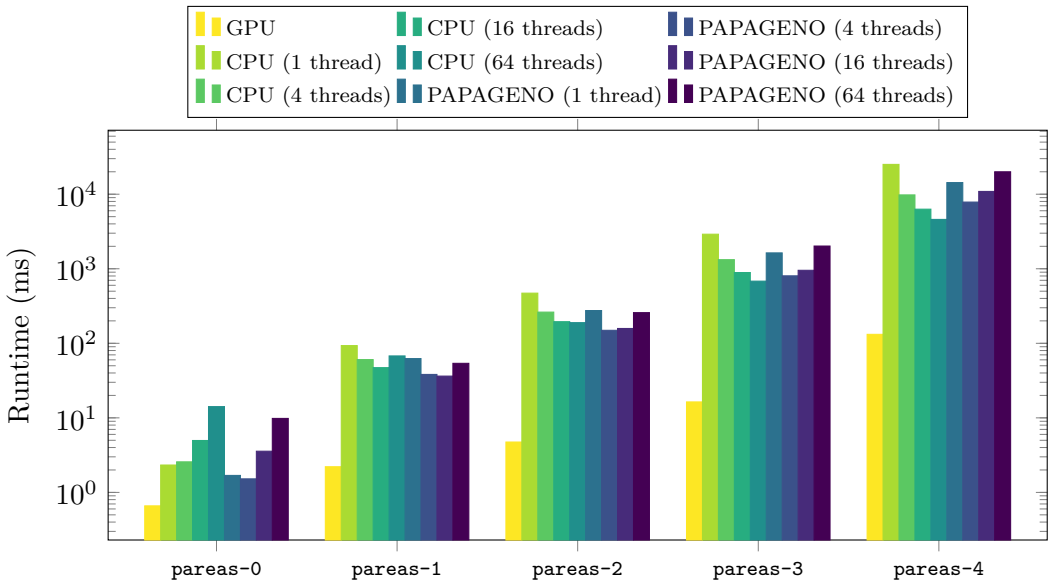
Figure 5.4 compares the runtimes we obtained for our own implementation with those of the other projects. In order to get a fairer comparison, in the case of the compiler we only count the runtime of the lexical analyzer, parser, and the runtime of the process to construct the parse tree, as the PAPAGENO parser performs the same work as these three stages. In the case of the JSON parser, the PAPAGENO version parses the exact JSON grammar, and so we do count the restructuring and validation stages. The plot in Figure 5.4a shows the runtimes of the parser for our programming language, in which case a sequential lexical analyzer was used for the PAPAGENO experiments. For the runtimes obtained for the JSON parser, shown in Figure 5.4b, the parallel PAPAGENO JSON lexical analyzer was used. Note that this parallel lexical analyzer requires more than one thread, and so the relevant experiments with one thread could not be completed.

When comparing the results of parser for our programming language with the CPU version of our implementation and PAPAGENO, we can see that when the amount of threads is small, PAPAGENO outperforms our implementation. In the cases where the input is sufficiently large and a sufficient amount of threads is used, we can see that our implementation outperforms PAPAGENO. This is also the case in all experiments involving the JSON parser. In fact, we can see that when 4 or more threads are used, the PAPAGENO implementation scales negatively with the number of threads on larger inputs for both the grammar of our programming language and for JSON. Finally, we can see that in all cases, the GPU implementation outperforms the CPU-based implementations for our programming language, and for the majority of experiments involving the JSON parser.

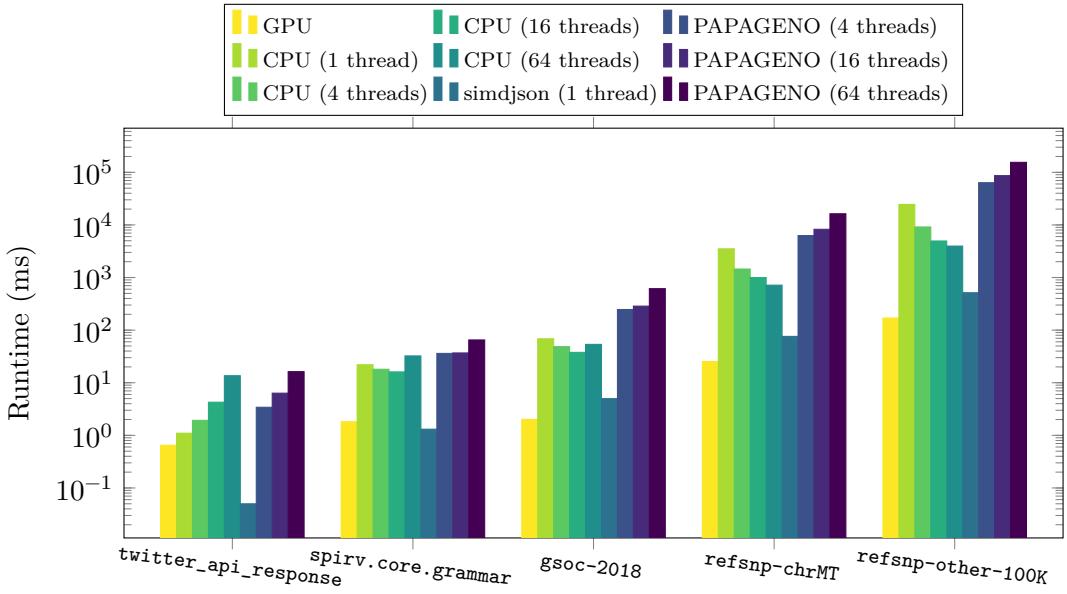
From Figure 5.4b we can see that in most cases, simdjson outperforms the other implementations. Even though simdjson only utilizes a single core, it is a production-grade high performance JSON parser, so this makes sense. When the input is sufficiently large, however, we can see that the GPU-powered JSON parser is able to outperform simdjson.

5.3.7. Summary

To summarize, analyzing the throughput of the implementations tells us that the runtime seemingly scales linearly with input size on both the GPU and CPU targets when the input is large enough, and that the GPU experiments outperforms the CPU experiments in all cases. The latter phenomenon may be caused by large memory pressures generated by many passes over the input, which is less efficient on CPU targets. This may also cause the degradation of performance observed when increasing the amount of cores on CPU experiments. When the input is small, synchronization and distribution overhead becomes a major part of the total runtime, and so this is less efficient. A noted drawback of GPU implementations is that kernels are compiled at runtime, which introduces a relatively large overhead. Furthermore, observe that the lexical analyzer and parser utilize a large amount of memory, which is explained by a combination of the implementations and the respective grammars. We see that our solution offers competitive performance and better scaling with number of threads with the CPU back end when compared with PAPAGENO. Furthermore, our GPU target outperforms the PAPAGENO CPU target. Finally, simdjson outperforms most other implementations, except our GPU-based parser on larger inputs.



(a) Comparison of runtimes for the lexical analyzer and parser of our programming language.



(b) Comparison of runtimes for the JSON lexical analyzer and parser.

Figure 5.4. Comparisons of our implementation with other parsers.

6. Conclusions

In this thesis, we described the design and implementation of the front end part of a compiler for an imperative, procedural and statically typed programming language, for which all major processing can be performed on a GPU. The textual representation of the program undergoes a series of compilation passes, which transform it into an abstract syntax tree. During this process, the structure of the program is validated, and a number of additional properties are computed. To this end, the source code is first broken up into a sequence of tokens using a parallel lexical analyzer, after which a parse tree is constructed by a parallel parsing process. The parsing algorithm suffers from a number of inherent limitations, though, which are mitigated by combination of grammar transformations and additional verification passes. The parse tree is then verified for semantic correctness, where variable, function and argument resolution is performed, as well as type analysis and function convergence. Finally, the parse tree and its attributes are exported to the abstract syntax tree, which is then converted into machine code in the back end part of the compiler as described by Huijben [Hui21].

A series of experiments serve to obtain some performance characteristics of the techniques used in this project. These are preformed on both the compiler itself, as well as a JSON parser implemented using similar techniques. While our results show a seemingly linear scaling of throughput versus input size, this behavior is only attained for relatively large input. When the input is smaller, the overhead of distribution and synchronization becomes evident. Our results also show that the GPU target outperforms the CPU target even when up to 64 threads are used, however, this can mostly be explained by the particular implementation of the compiler. Many small passes which process the entire tree creates a large memory pressure, which GPUs are better designed to deal with. When comparing CPU targets running with different amounts of threads on sufficiently large input, we do not see a linear scaling. This may have a number of explanations, for example the scalability of our algorithms, overhead of work distribution and synchronization, or memory pressure.

When breaking down the runtime characteristics of the different stages in our implementations we see that most of them scale similarly, where a linear correlation between throughput and input size is achieved when the input is sufficiently large. Note that this behavior is also dependent on the characteristics of the input file, though, and some variation is to be expected on real-world data sets. Comparing memory usage of the different stages, we see that a large amount is allocated during the lexical analysis and parser processes. In the former case, up to 20 times the size of the input is allocated, which is explained by the particular implementation of the lexical analyzer. In the case of the parser, it is explained by the parsing process and the features of the syntactical grammar.

Our work is also compared to a different parallel lexical analyzer and parser implementation [Bar+15] and a production-grade high performance SIMD based JSON parser [LL19]. Both of these are CPU based only. When comparing CPU targets of our implementation

with the former, we observer similar performance and better scaling. The latter implementation uses only a single core, but out performs all other CPU parsers. When the input is sufficiently large, however, the GPU target outperforms this implementation. Comparing the massively parallel GPU architecture with a single-core CPU based implementation, which can be easily replicated over multiple cores to parse different documents in parallel, is not very fair, but it does give us a hint that our techniques show promise in absolute terms.

The lack of a proper baseline makes it hard to draw a final conclusion from our experiments. The linear scaling shows is that this research area is interesting particularly when the size of the input is large, and developing GPU-based tools for these cases may prove efficient. A drawback of the techniques presented in this research is the amount of memory overhead, precisely because it limits its applications on very large input files.

6.1. Future Work

This project is obviously still in its infancy, and there are many improvements that could be made. The ultimate goal of course is to be able to compile programs in an existing programming language like C. In the following sections, a non-exhaustive list of methods is outlined to generally improve the compiler and bring us closer to that goal.

6.1.1. Improving the Lexical Analyzer

While the current approach to lexical analysis shows promise, many improvements to the algorithm could be made. First, the merge table of the lexical analyzer is in the current implementation of the compiler relatively large, and becomes even larger with a more complicated lexical grammar. As is discussed in Section 5.3, the lexical analyzer requires almost 35 MB of pre-computed data in the case of the JSON parser. The size of this table largely depends on the amount of syntactic structures a character can be part of. For example, in the grammar for our programming language, the plus character can either be interpreted as an operator, or it can be part of a comment. When more of these structures are introduced, for example in a programming language which also supports strings, the amount of entries in the merge table blow up drastically, and makes the entire system infeasible. If this method is to be used for more complicated textual languages, the size of this table needs to be reduced.

One method that might be of interest is to process the input using *two* lexical analyzers. The first would break up and disambiguate the input for the second. For example, the first lexical analyzer can be used to determine whether a character is part of a string, a comment, or regular code. This increases the alphabet of the second lexical analyzer, but allows it to more easily split up the input into the final sequence of tokens.

When the merge table required by the lexical analyzer is in an order of megabytes, as is the case in both the lexical analyzer for our programming language and the JSON parser, it needs to reside in the global memory heap. This memory is relatively slow, especially

for the random access patterns that accesses to this memory would exhibit. If the size of the merge table can be reduced to only tens of kilobytes, it could be placed in shared memory instead. This is fast memory local to compute units, and if the merge table is small enough to fit in this section, lexical analysis speed could be improved drastically. We note that many of the entries of the merge table are likely to lead to similar states, for example the reject state, and so the size of this table may be reduced by employing sparse matrix storage techniques.

6.1.2. Improving the Parsing Method

As the results of the experiments described in Chapter 5 show, a large part of the total runtime of the compiler is dedicated to parsing and mitigating parser limitations. Furthermore, these parser limitations also form a large part of the implementation of the compiler itself. This is largely attributed to the relatively limited set of languages that $LLP(1,1)$ includes. Parsing in parallel is a complicated problem, but if a parallel parsing method is devised that accepts a larger set of languages, the runtime of the compiler could be reduced.

One relatively simple method to increase the set of languages which the current implementation accepts is to increase the amount of look ahead and look back tokens. This is currently hard coded to one token in both cases, but the algorithm discussed by Vagner & Melichar [VM07] scales to any amount. For example, using two look back and look ahead tokens is enough to distinguish unary and binary minus, and to distinguish function applications from variable declarations.

Another problem with the parsing method is the amount of memory overhead. As discussed in Section 5.3, a leading factor of this fact is the pair of nodes that is generated for each precedence level. If the parsing method could be extended in such a way that some nodes can be attributed to be ignored during construction of the initial parse, the size of the initial tree and thus the total memory usage of this phase could be reduced.

6.1.3. Improved Type Analysis

One of the most important parts of a compiler is the type analysis system. In our programming language, the type system is relatively simple: there are only a few core types, and there is only a single way that a type can be modified, which is by either gaining or losing a reference. Type checking can be performed in parallel because the operations which modify types are associative, as counting the amount of gained references is reduced to a parallel prefix sum. It remains to be seen whether this is also the case for more complicated programming languages, and so more research needs to be performed in this area.

Bibliography

[Hui21] M Huijben. “Parallel Code Generation on the GPU”. MSc Thesis. Universiteit Leiden, 2021.

[VM07] Ladislav Vagner and Bořivoj Melichar. “Parallel LL parsing”. In: *Acta informatica* 44.1 (2007), pp. 1–21.

[Wyl79] James C Wyllie. *The complexity of parallel computations*. Tech. rep. Cornell University, 1979.

[HS86] W Daniel Hillis and Guy L Steele Jr. “Data parallel algorithms”. In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

[Hen+17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. “Futhark: purely functional GPU-programming with nested parallelism and in-place array updates”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 556–571.

[Hil89] W Daniel Hillis. *The connection machine*. MIT press, 1989.

[SB89] David B Skillicorn and David T Barnard. “Parallel parsing on the connection machine”. In: *Information Processing Letters* 31.3 (1989), pp. 111–117.

[Hil92] Jonathan MD Hill. “Parallel lexical analysis and parsing on the AMT distributed array processor”. In: *Parallel computing* 18.6 (1992), pp. 699–714.

[Joh11] Mark Johnson. “Parsing in parallel on multiple cores and GPUs”. In: *Proceedings of the Australasian Language Technology Association Workshop 2011*. 2011, pp. 29–37.

[Bar+15] Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Federica Panella, and Matteo Pradella. “Parallel parsing made practical”. In: *Science of Computer Programming* 112 (2015), pp. 195–226.

[MMS14] Todd Mytkowicz, Madanlal Musuvathi, and Wolfram Schulte. “Data-parallel finite-state machines”. In: *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 2014, pp. 529–542.

[SMS13] Ryoma Sinya, Kiminori Matsuzaki, and Masataka Sassa. “Simultaneous finite automata: An efficient data-parallel model for regular expression matching”. In: *2013 42nd International Conference on Parallel Processing*. IEEE. 2013, pp. 220–229.

[SB93] David B. Skillicorn and David T. Barnard. “Compiling in parallel”. In: *Journal of Parallel and Distributed Computing* 17.4 (1993), pp. 337–352.

[Hsu19] Aaron Wen-yao Hsu. “A data parallel compiler hosted on the gpu”. PhD thesis. Indiana University, 2019.

[Con19] GNU Contributors. *ParallelGcc*. 2019. URL: `https://gcc.gnu.org/wiki/ParallelGcc` (visited on 07/21/2021).

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[BMR98] Omer Berkman, Yossi Matias, and Prabhakar Ragde. "Triply-logarithmic parallel upper and lower bounds for minimum and range minima over small domains". In: *Journal of Algorithms* 28.2 (1998), pp. 197–215.

[BV85] Ilan Bar-On and Uzi Vishkin. "Optimal parallel generation of a computation tree form". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7.2 (1985), pp. 348–357.

[DN13] Aditya Deshpande and PJ Narayanan. "Can GPUs sort strings efficiently?" In: *20th Annual International Conference on High Performance Computing*. IEEE. 2013, pp. 305–313.

[Hen+20] Troels Henriksen, Sune Hellfritzsch, Ponnuswamy Sadayappan, and Cosmin Oancea. "Compiling Generalized Histograms for GPU". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia: IEEE Press, 2020. ISBN: 9781728199986.

[LL19] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (2019), pp. 941–960.

[She+01] Stephen T Sherry, M-H Ward, M Kholodov, J Baker, Lon Phan, Elizabeth M Smigielski, and Karl Sirotkin. "dbSNP: the NCBI database of genetic variation". In: *Nucleic acids research* 29.1 (2001), pp. 308–311.

A. Language Design

In the interest to keep the scope of the project manageable, the programming language is kept relatively simple. On a global level, a program is a single file of source code consisting of a sequence of function definitions. There are no import statements, nor a way to reference some externally declared function. This keeps the compilation process simple: the compiler can simply read the source code into host memory, upload it to GPU memory, and call the appropriate compilation kernels.

Each function definition consists of a sequence of statements, whose actions are executed when the function is invoked. These include for example expressions, conditional statements, loop statements, and return statements. Functions refer to each other by a program-wide unique name which is assigned in the function's declaration. There is no particular required order for function declarations, such as with some other programming languages.

Our programming language is statically typed, meaning that every value has an explicitly or implicitly assigned data type. There are two primitive data types: integers and floating points, following signed 32-bit and 32-bit IEEE 745 semantics respectively. Furthermore, expressions and sub-expressions is classified as one of two mutually exclusive value categories: when the result of the expression may appear on the left-hand side of an assignment operator, so when the result is writable, it is classified as an l-value. If the result may only appear on the right-hand side of the assignment operator, when the result is not writable, it is classified as an r-value instead. Note that at any point, an l-value may transform into an r-value, but not vice versa. For example, in the expression `a = 1 + b`, `a` and `b` are l-values, and `1` is an r-value. Before the addition operator is executed, `b` transforms from an l-value into an r-value. The result of the addition operator is also an r-value.

As the only allowed top-level structure in our programming language are function definitions, there is no way to declare a global variable which is shared between functions. Instead, functions communicate through parameters and return values: each function definition includes a list of parameter variables, with an explicit data type, and an explicit return type. When the function is invoked with some list of arguments, the parameter variables are assigned the corresponding values. A `return`-statement may be invoked to terminate function execution, and the return value is the result of an expression which the `return`-statement is associated with. A function may also return no value, which is indicated using the special `void` data type. In such functions, there is no expression associated with `return`-statements.

A.1. Compound Statements

Some statements guard other statements. For example, a conditional statement is associated with some other statements which are executed only when the condition evaluates to a truthy value. When a statement guards multiple statements, they have to be delimited by some syntactic structure to indicate which statements belong together. These are known as *compound statements*. In our programming language, and many others, compound statements are delimited by curly braces:

$$\langle \textit{compound} \rangle \quad \rightarrow \quad \{\ \langle \textit{statement list} \rangle\ \}$$

Some programming languages allow omitting braces when a statement guards only a single other statement. This allows the programmer to save a few key strokes on occasion, and yields minor simplifications in the programming language's syntactical grammar. This can be expressed as follows, for example:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & \rightarrow & \langle \textit{compound} \rangle \\ & | & \textbf{\textit{while}}\ (\ \langle \textit{expr} \rangle\ )\ \langle \textit{statement} \rangle \\ & | & \langle \textit{expr} \rangle\ ; \\ & | & \ldots \end{array} \tag{A.1}$$

When parsing the sub-statements of the `while`-statement, if an opening curly brace appears on the input the parser will proceed to parse a sequence of statements until the matching closing brace. Otherwise, it will simply parse a single statement. This works fine for a regular predictive parser, however, this type of syntactic sugar is problematic for LLP parsers. We must be able to unambiguously determine a sequence of stack changes from any pair of consecutive symbols, however, from the pair $\langle ;, \textbf{\textit{while}} \rangle$ it is not clear how many nested statements are exited. For example in the context `while (1) while (1) while (1) 1; while (1) {}` three nested statements are exited, which would require us to pop three $\langle \textit{statement} \rangle$ states from the stack. Alternatively, it could appear in the context `1; while (1) {}`, where none at all were exited.

The solution is to simply enforce braces on every statement which guards another. Note that in Grammar A.1, disambiguation is required between the controlling expression and the sub-statements, which is done by enforcing parenthesis around the controlling expression. Enforcing races around the sub-statements instead also solves this ambiguity, and so these are not required. This yields the following grammar:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & \rightarrow & \langle \textit{compound} \rangle \\ & | & \textbf{\textit{while}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle \\ & | & \langle \textit{expr} \rangle\ ; \\ & | & \ldots \end{array}$$

A.2. Conditional Statements

It is common for programming languages to allow the alternative of a conditional statement to be omitted. This can be represented by a formal grammar as follows:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & \rightarrow & \textbf{\textit{if}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle \\ & | & \textbf{\textit{if}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle\ \textbf{\textit{else}}\ \langle \textit{compound} \rangle \\ & | & \ldots \end{array}$$

This grammar cannot be parsed by the backing $LL(1)$ parser, however, and the common way to fix that is by transforming the grammar via *left factoring* into the following:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & \rightarrow & \textbf{\textit{if}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle\ \langle \textit{maybe else} \rangle \\ & | & \ldots \\ \langle \textit{maybe else} \rangle & \rightarrow & \textbf{\textit{else}}\ \langle \textit{compound} \rangle \\ & | & \varepsilon \end{array}$$

This syntax is not $LLP(1,1)$, however, as after $\langle \textit{compound} \rangle$ it cannot be decided whether to pop both $\langle \textit{maybe else} \rangle$ and $\langle \textit{statement} \rangle$ from the stack, or just the latter. To solve this, we introduce a separate `else`-statement for the alternative, which on a grammar level is not required to follow the condition and consequent. Furthermore, as blocks are required to be delimited by braces, we introduce syntactical sugar in the form of an `elif`-statement for chaining an alternative with another conditional. Similarly to the alternative of a regular conditional, it is also parsed as a separate statement. An additional semantic analysis ensures that the input has the proper structure: Both the `else`- and `elif`-statement must follow either an `if`- or `elif`-statement. This pass also re-structures the parse tree so that conditionals appear as proper nodes instead of sequences of statements. This yields the following grammar for conditionals:

$$\begin{array}{lcl} \langle \textit{statement} \rangle & \rightarrow & \textbf{\textit{if}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle \\ & | & \textbf{\textit{elif}}\ \langle \textit{expr} \rangle\ \langle \textit{compound} \rangle \\ & | & \textbf{\textit{else}}\ \langle \textit{compound} \rangle \\ & | & \ldots \end{array}$$

A.3. Binary Operators

As most binary operators are left-to-right associative, expressing binary operators naturally in a formal grammar typically requires *left recursion*. Productions are left recursive when the first symbol of the right-hand side of a certain production in the grammar may derive to the same non-terminal as on the left-hand side. For example, a grammar which can parse simple expressions containing addition and multiplication operators may be defined as follows:

$$
\begin{array}{lcl}
\langle sum \rangle & \rightarrow & \langle sum \rangle + \langle prod \rangle \mid \langle prod \rangle \\
\langle prod \rangle & \rightarrow & \langle prod \rangle * \langle atom \rangle \mid \langle atom \rangle \\
\langle atom \rangle & \rightarrow & \textbf{\textit{number}} \mid (\ \langle sum \rangle\ )
\end{array}
\tag{A.2}
$$

Unfortunately, top-down parsers such as the backing $LL(1)$ parser used in our compiler cannot deal with this left recursiveness. The typical way to deal with that is to transform each of the offending productions to be right recursive instead of left recursive, by introducing extra rules as follows:

$$
\begin{array}{lcl}
\langle sum \rangle & \rightarrow & \langle term \rangle\ \langle sum' \rangle \\
\langle sum' \rangle & \rightarrow & +\ \langle prod \rangle\ \langle sum' \rangle \mid \varepsilon \\
\langle prod \rangle & \rightarrow & \langle atom \rangle\ \langle prod' \rangle \\
\langle prod' \rangle & \rightarrow & *\ \langle prod' \rangle \mid \varepsilon \\
\langle atom \rangle & \rightarrow & \textbf{\textit{number}} \mid (\ \langle expr \rangle\ )
\end{array}
\tag{A.3}
$$

This modified grammar is $LL(1)$, and accepts exactly the same language. Unfortunately though, the parse tree produced by this grammar is slightly different: The expression as $1+2+3$ would be parsed as $(1+2)+3$ according to left recursive Grammar A.2, but would be parsed as $1 + (2 + 3)$ by modified Grammar A.3. As the multiplication and addition operators are associative, this does not form a problem for the above grammars, but the situation changes when we introduce non-associative operators such as subtraction and division. We solve this by introducing an additional pass in the semantic analysis stage which re-structures the parse tree so that expressions follow the expected behavior. The implementation of this pass is further described in Section 4.8.1.

A.4. Unary and Binary Minus Expressions

Mathematical notation uses the same minus symbol to mean either of two operations, depending on context: when the minus operator is applied as a unary prefix operator to a single argument it performs negation, and when it is used as an binary infix operator it performs subtraction. This syntax is implemented in many programming languages, and during parsing the distinction is usually quite easy to make. This is not so with $LLP(1,1)$ grammars: we must be able to tell from any two consecutive symbols with what construct we are dealing with, but from the pair $\langle -, a \rangle$ we cannot tell whether the minus is used as the binary or unary version. After all, this pair might for example appear in the context `b-a`, in which case the subtraction operator is intended, or in the context `b=-a`, where the minus means negation instead.

Note that this problem only manifests itself for pairs where the minus is the first symbol. When the minus is the second symbol, we can always infer which operator it is: for example, in the pair $\langle a, - \rangle$ it is clear that the binary version is used, and in $\langle =, - \rangle$ the minus represents the unary operation. This points at one of two possible solutions: either extending the grammar to $LLP(2,1)$, where the extra look back symbol can be used to disambiguate the true version of the minus operator, or by extending the lexer to distin-

guish unary and binary minus by keeping track of the previous token. In our compiler, we have chosen for the latter solution.

Instead of actually keeping track of the previous token though, we extend the construction process for the lexical analyzer discussed in Section 3.1.1. It does not require further extension of the machine outlined in Definition 3.3, nor does it require altering the parallel lexical analysis process. Instead, each pattern in a lexical grammar may optionally specify an exhaustive list of tokens that must precede it. These patterns are still converted into a corresponding nondeterministic finite automaton, but are not connected to the start state. For every pattern which has a successor, a new root state is created, and the successors are connected to that instead. Every nondeterministic automaton is then converted into the corresponding deterministic automaton via the subset construction, and only after this are the successors connected. This happens in a similar process as the process which converts the automaton for a single token into an automaton which can match many tokens: for every accepting state in every automaton, all outgoing states from the corresponding successor root node are copied to the accepting state, and are marked as producing a token. After this, the original loop construct is performed as usual. In both cases, existing transitions are left intact to make the automaton prioritize continuing with the current and successor patterns over terminating a match early and starting with the next.

A.5. Function Application

A few conflicts appear when parsing function application syntax. The mathematical notation for application, and the syntax which many popular programming languages have therefore adopted, is by writing a list of expressions delimited by parentheses behind some expression which evaluates to a function. As our programming language only allows functions to be referenced by name instead of complete expressions, it yields the following grammar:

$$\begin{array}{lcl} \langle atom \rangle & \rightarrow & \langle name \rangle \\ & | & \langle name \rangle \ (\ \langle expression\ list \rangle\ ) \end{array}$$

Note that this has the same problem as parsing the alternative of a conditional statement, and so it is transformed using left factoring into the following grammar:

$$\begin{array}{lcl} \langle atom \rangle & \rightarrow & \langle name \rangle\ \langle maybe\ app \rangle \\ \langle maybe\ app \rangle & \rightarrow & (\ \langle expr\ list \rangle\ ) \\ & | & \varepsilon \end{array}$$

This works fine, but when we introduce parenthesized expressions, a conflict appears: it is ambiguous whether the symbol pair $\langle), ; \rangle$ means to close the application or to close a parenthesized expression. While these mostly pop the same states in the same order from the stack, the former also pops $\langle maybe\ app \rangle$ and the latter does not. We solve this ambiguity simply by using brackets instead of parentheses for function call syntax, which yields the following grammar:

$$
\begin{array}{lcl}
\langle atom \rangle & \rightarrow & (\ \langle expr \rangle\ ) \\
& | & \textbf{\textit{name}}\ \langle maybe\ app \rangle \\
& | & \ldots \\
\langle maybe\ app \rangle & \rightarrow & [\ \langle expr\ list \rangle\ ] \\
& | & \varepsilon
\end{array}
$$

A.6. Function & Variable Declarations

Syntax for function prototypes in function declarations consists of a function name, a list of parameters, and a return type. In most programming languages, function declaration syntax mirrors function application syntax: they use the same delimiter for parameter and argument lists, and the same separator between parameters and between arguments. For example, C uses parenthesis and commas respectively. This causes problems in the context of LLP grammars, however, as we for example cannot tell whether $\langle add, [\rangle$ is part of a function declaration or a function application. One method to solve this issue is to add an alternative syntax, but to keep the language more consistent we simply parse the prototype as an expression, and perform an additional semantic analysis pass which checks that the user has entered the proper structure:

$$
\langle fn\ decl \rangle \quad \rightarrow \quad \textbf{\textit{fn}}\ \langle expr \rangle\ \langle compound \rangle
$$

As our programming language is statically typed, the user needs to explicitly state the type of each variable, and the resulting type of a function. To this end, we introduce a *type ascription* syntax, which states that the expression which it is applied to evaluates to a value of a certain type. The same semantic analysis pass then checks that each 'argument' of a function application expression in a function declaration is then a name with a type ascription. The return type is written by applying a type ascription to the entire function application expression.

$$
\begin{array}{lcl}
\langle ascript\ expr \rangle & \rightarrow & \langle sum \rangle\ \langle maybe\ ascript \rangle \\
\langle maybe\ ascript \rangle & \rightarrow & :\ \langle type \rangle \\
& | & \varepsilon
\end{array}
$$

In order to make a programmer able to declare variables, we introduce the ***var*** keyword. When this token appears before a ***name*** token, it indicates that a new variable is declared here. When this declaration is a sub-expression of a type ascription expression an explicit type is assigned to the variable, and otherwise, the compiler will attempt to infer a type. This yields the following grammar:

$$
\begin{array}{lcl}
\langle atom \rangle & \rightarrow & \textbf{\textit{name}}\ \langle maybe\ app \rangle \\
& | & \textbf{\textit{var name}} \\
& | & \ldots \\
\langle maybe\ app \rangle & \rightarrow & [\ \langle expr\ list \rangle\ ] \\
& | & \varepsilon
\end{array}
$$

This grammar contains a new parsing conflict, however. For example, it is unclear whether the partial parse generated for the pair $\langle a, ; \rangle$ should include the production $\langle \textit{maybe app} \rangle \rightarrow \varepsilon$. To solve this, we allow $\langle \textit{maybe app} \rangle$ behind ***name*** in both productions, and check that variable declarations have no applications in the same semantic analysis stage as mentioned previously.

B. Parser Mitigations

B.1. Flattening Lists

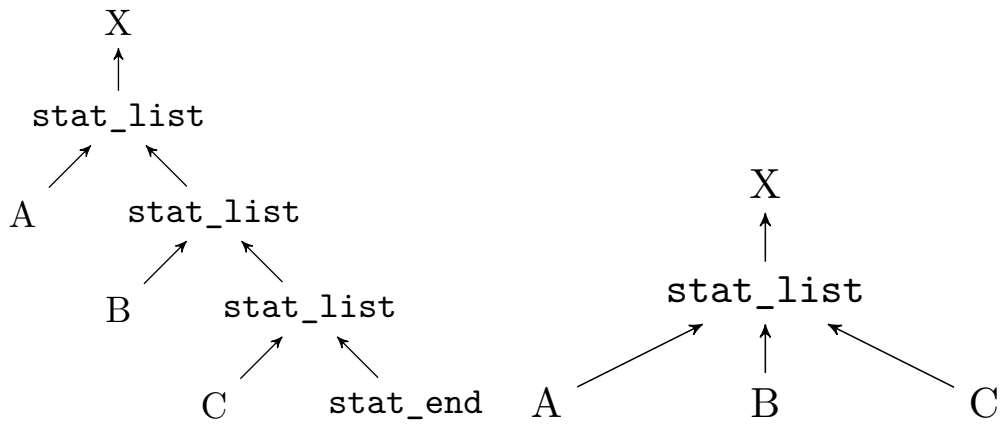
The parser cannot natively represent lists of varying sizes, and so these are represented by right recursive rules instead. This is used in most places where a varying number of items is allowed: parameter- and argument lists, statement lists, function declaration lists, operator lists. In the case of the latter, the intermediary list items themselves carry semantic information, for example, whether a node in an operator list is a subtraction or addition operator. The other node types do not carry such information. For example, every intermediary list item of an argument list is the same type of node, and so these can be safely removed. Furthermore, the presence of these nodes makes it harder to navigate the tree during further processing. For this reason, all but the first node of every list is removed, including the ending node. This results in a new 'flattened' list, where every item of the list is a child of the original first node of the list. See the example in Figure B.1. Note that this operation also preserves the pre-order ordering between children of the list in the backing array.

B.2. Variables, Variable Declarations and Function Applications

Due to parser limitations, the syntactical grammar allows both the variable declaration and regular variable nodes to have an application. When this is the case, either of these nodes have subtree with as root the `app` node, and otherwise, there is a `no_app` child to indicate absence of an application. See example parse tree in Figure B.2a. We perform several operations on these nodes:

- First, `no_app` nodes don't contain useful information, and so these are removed.
- Next, variable declaration nodes may not have an application, and so the parse tree is checked for any such structures.
- Finally, we merge variable and application nodes into a new `atom_fn_call` node. This simplification helps further down the pipeline, as we can now search for the new node type instead of a specific structure of multiple nodes.

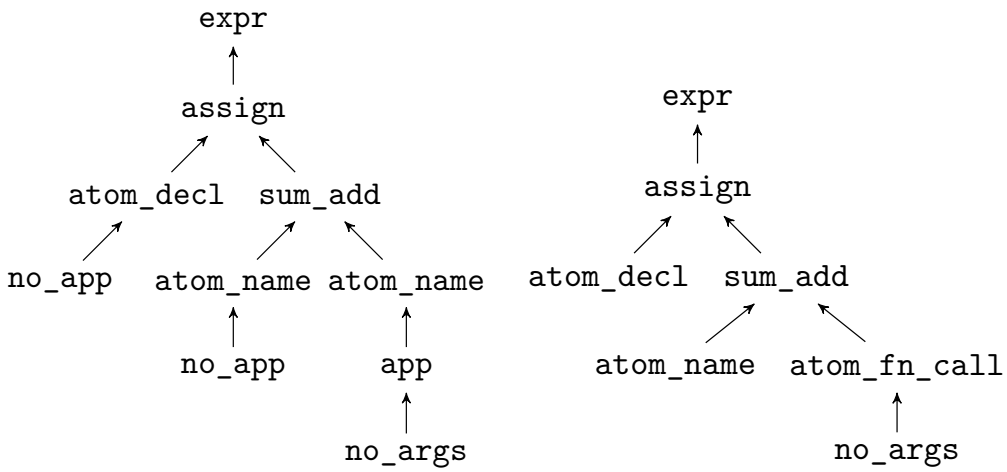
Applying these operations to the parse tree in Figure B.2a yields the parse tree in Figure B.2b.



(a) A list of 3 statements, parsed as a right recursive list.

(b) After flattening the lists.

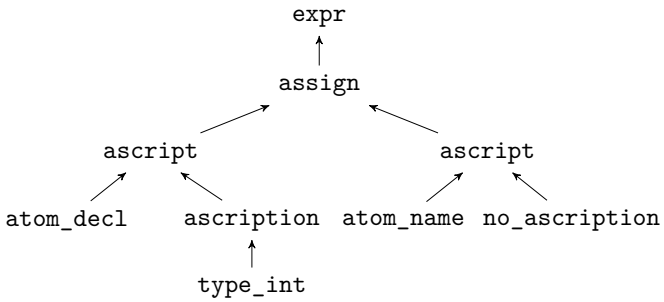
Figure B.1. Before and after flattening right-recursive lists.



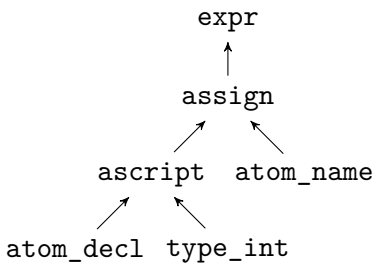
(a) Initial parse tree

(b) Desired parse tree

Figure B.2. The initial and desired parse tree of **var** $a = b + c[\,]$, after the operations from previous sections. Note that some nodes not related to this pass have been omitted.



(a) The parse tree of **var** $a : \mathbf{int} = b$, after previous passes have been applied.



(b) The same parse tree, cleaned up.

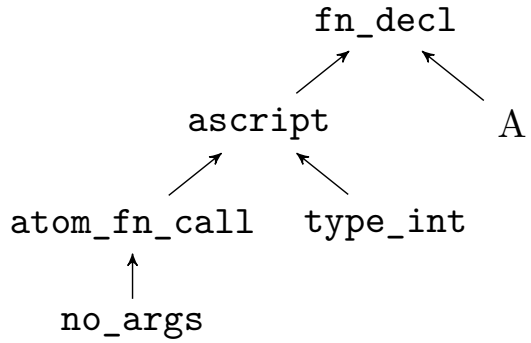
Figure B.3. The initial and final parse tree of **var** $a : \mathbf{int} = b$.

B.3. Type Ascriptions

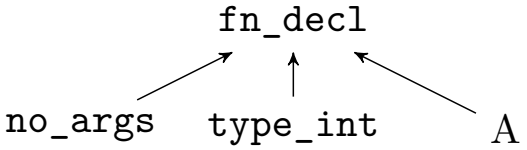
Another small process involves cleaning up expressions generated by type ascriptions. These are used in certain contexts to guard that an expression must yield a certain type, and in other contexts declares a variable with a specific types. Regardless, the parse tree generated by this operator suffers from similar problems as binary operators. Even though lists of ascriptions do not make sense, there is still an `ascript` node that is generated in every expression. This node has two children: an expression and a child indicating the type or absence thereof. Cleaning up the tree involves removing any of the latter type. Furthermore, the node indicating that there *is* an ascription does not carry useful information, as presence of an ascription is now already indicated by the presence of the `ascript` node, and so these are also removed. See the initial parse tree generated by parsing and processing **var** $a : \mathbf{int} = b$ using the previous passes in Figure B.3a, and the final parse tree after the ascriptions have been cleaned up in Figure B.3b.

B.4. Function Declarations

As described in Section A.6, the prototype of a function declaration is parsed as an expression. After the transformations from previous passes have been applied, the subtree of a function declaration consists of the initial declaration node, with as left child an ascription node, which in turn has a function application node as left child. In the context of function declarations, the latter two nodes do not contain any useful information, as a function declaration always consists of a list of function arguments, a return type, and a function body. The semantic analysis pass which deals with these then consists of two things. First, we verify that the tree forms the expected structure, and second, the superficial ascription and function application nodes are removed. See the example in Figure B.4.

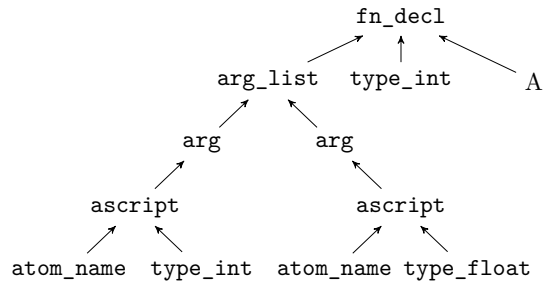


(a) The initial parse tree, after applying the previous transformations.

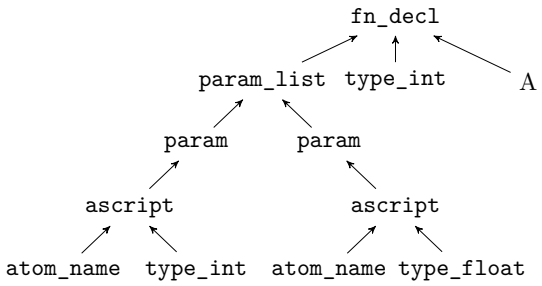


(b) The parse tree after restructuring function declarations.

Figure B.4. The parse tree of **fn** $a[\,] : \mathbf{int}\{A\}$ before and after processing.

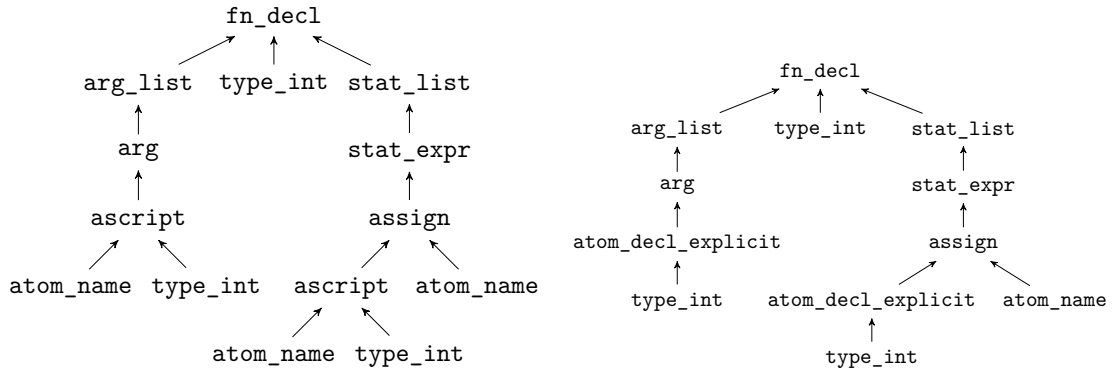


(a) The parse tree of **fn** $a[b : \mathbf{int}, c : \mathbf{int}] : \mathbf{int}\{A\}$, after applying previous transformations.



(b) The same parse tree after transforming the argument list into a parameter list.

Figure B.5. The parse tree before and after transforming argument lists to parameter lists.



(a) Initial parse tree of **fn** $a[b : \mathbf{int}] : \mathbf{int}\{\mathbf{var}\ c : \mathbf{int} = d; \}$.

(b) The parse tree after replacing variable declarations.

Figure B.6. Transforming variable declarations into a dedicated node.

B.5. Argument and Parameter Lists

There are a few small items that are restructured and verified with regards to argument and parameter lists:

- First, note that function applications and definitions feature a `no_args` node to indicate the absence of any arguments. This is transformed into an `arg_list` node, so that it is similarly structured to applications and definitions with one or more arguments.

- Next, it is useful to tell parameter and argument lists apart, so `arg_list` nodes which are the child of a function declaration are transformed into a `param_list` node. Furthermore, note that the child of each `arg_list` node is a `arg` node. While this does not carry much additional information, it will be useful further down the pipeline to associate additional semantic information to items of argument and parameter lists, and so this node is not removed. In the case of parameter lists though, it is transformed into a `param` node. See Figure B.5.

B.6. Implicit and Explicit Variable Declarations

In the programming language accepted by the compiler, both *implicitly typed* and *explicitly typed* variable declarations are allowed. In the former version, the actual type of the values that the variable may store is automatically determined by the compiler from its initializing expression. In the latter case, the programmer explicitly assigns the type by placing a type ascription after a variable declaration. Another type of variable declaration are function parameter declarations. These follow a similar structure, except that the **var** keyword normally used for variable declarations is superficial in this context, and instead the compiler expects a `atom_name` node instead. Parameter declarations do not have an initializing expression, and so parameter are not allowed to be implicitly declared.

We implement two operations which deals with these structures:

- First, we check that all parameter declarations, which since the transformation discussed in Section B.5 can be identified as children of `param` nodes, follow the expected structure. That is, an `ascript` node with a `atom_name` as left child.

- Second, we replace the name and ascription of variable and parameter declarations with a single `atom_decl_explicit` node. This node always has one child, which previously belonged to the `ascript` node. The storage type of the variable. See the transformation in Figure B.6.

B.7. Assignment Operators

Assignment operators store the value produced by the right-hand side of the operator in the variable on the left-hand side. This requires the left-hand side to be an expression which produces an l-value, as the expression $1 = 2$ does not make sense in a language where $=$ means assignment. While in a more complicated programming language there may be many expressions which yield a writable value, in our language, there are only three types of which generate a writable location: variable declarations with implicit type (`atom_decl`), variable declarations with explicit type (`atom_decl_explicit`), and regular variables (`atom_name`). To check whether the program is correctly structured in this regard, we simply check that every left child of an assignment node is either of these three.

Similar to the operation described in Section 4.9.1. we are required to distinguish left and right children of the assignment operator, and so we require the sibling arrays as computed by the algorithm explained in Section 4.7.4. Note that this pass is in fact executed just before the pass described in Section 4.9.1, and the sibling arrays computed in this pass are re-used by the passes following it.

C. Result Tables

Table C.1 shows the results of the PAPAGENO parser configured for our programming language. Tables C.2 and C.3 show the results of the PAPAGENO JSON parsers, respectively with and without parallel lexical analyzer. Table C.4 shows the results of the simdjson JSON parser. Finally, Tables C.5 and C.6 show the results of our compiler and JSON parsers implementations, broken down by major stage. All results are the average of 30 runs, $\pm$ standard deviation. All values are in milliseconds (ms).

Data Set	Threads	Lexical Analysis	Parsing
`pareas-0`	1	0.33 ± 0.03	1.36 ± 0.13
	4	0.34 ± 0.03	1.18 ± 0.18
	16	0.34 ± 0.05	3.23 ± 0.38
	64	0.33 ± 0.06	9.50 ± 0.90
`pareas-1`	1	13.46 ± 2.91	48.93 ± 5.98
	4	12.03 ± 2.70	26.23 ± 2.15
	16	14.59 ± 1.24	21.74 ± 2.25
	64	16.89 ± 1.92	37.00 ± 2.18
`pareas-2`	1	44.57 ± 5.12	230.52 ± 6.83
	4	45.49 ± 5.76	103.93 ± 11.99
	16	43.47 ± 2.96	114.67 ± 11.73
	64	51.99 ± 2.14	205.89 ± 15.07
`pareas-3`	1	250.32 ± 4.85	1382.39 ± 6.52
	4	248.55 ± 5.59	555.30 ± 44.21
	16	245.81 ± 4.88	707.96 ± 96.11
	64	259.02 ± 4.25	1758.06 ± 242.06
`pareas-4`	1	2149.46 ± 17.63	12173.75 ± 84.43
	4	2151.77 ± 7.73	5696.15 ± 428.63
	16	2160.48 ± 21.94	8734.19 ± 730.25
	64	2150.56 ± 14.10	17839.69 ± 1437.51

Table C.1. Results of the PAPAGENO parser for our programming language, without parallel lexical analyzer.

Data Set	Threads	Lexical Analysis	Parsing
twitter_api_response	1	1.01 ± 0.26	3.26 ± 0.66
	4	1.15 ± 0.14	1.79 ± 0.22
	16	1.10 ± 0.21	3.62 ± 0.39
	64	1.13 ± 0.18	9.94 ± 0.68
spirv.core.grammar	1	13.71 ± 3.86	41.67 ± 5.08
	4	15.73 ± 2.49	19.40 ± 1.67
	16	18.51 ± 2.43	22.05 ± 1.76
	64	19.65 ± 1.17	31.66 ± 2.04
gsoc-2018	1	81.10 ± 5.36	299.17 ± 11.01
	4	74.25 ± 5.15	122.98 ± 6.57
	16	72.73 ± 3.87	138.82 ± 9.74
	64	81.34 ± 1.18	284.78 ± 33.13
refsnp-chrMT	1	1464.29 ± 48.02	5926.85 ± 186.54
	4	1474.39 ± 62.53	2742.36 ± 224.45
	16	1457.24 ± 6.39	4003.40 ± 190.47
	64	1460.10 ± 5.58	7615.88 ± 759.20
refsnp-other-100K	1	9599.97 ± 14.85	39628.21 ± 1469.34
	4	9604.63 ± 18.92	17338.01 ± 1000.34
	16	9599.09 ± 10.49	32705.76 ± 2288.17
	64	9619.25 ± 297.19	53819.57 ± 726.85

Table C.2. Results of the PAPAGENO JSON parser, without parallel lexical analyzer.

Data Set	Threads	Lexical Analysis	Parsing
twitter_api_response	4	1.50 ± 0.21	1.91 ± 0.24
	16	2.99 ± 0.33	3.35 ± 0.28
	64	7.25 ± 0.64	9.01 ± 0.91
spirv.core.grammar	4	17.20 ± 1.72	18.92 ± 1.77
	16	13.26 ± 1.50	23.61 ± 2.58
	64	26.67 ± 1.39	38.44 ± 2.89
gsoc-2018	4	115.84 ± 15.06	130.97 ± 9.06
	16	136.69 ± 19.31	149.44 ± 12.85
	64	295.99 ± 34.67	318.25 ± 51.85
refsnp-chrMT	4	2829.77 ± 175.51	3442.25 ± 817.63
	16	4295.52 ± 270.86	3966.81 ± 275.85
	64	8377.71 ± 840.81	7915.11 ± 646.19
refsnp-other-100K	4	22106.40 ± 939.97	41377.34 ± 6414.07
	16	36464.44 ± 826.34	50023.41 ± 2350.92
	64	64337.36 ± 5115.35	90481.90 ± 6075.06

Table C.3. Results of the PAPAGENO JSON parser, with parallel lexical analyzer.

Data Set	Parsing
twitter_api_response	0.05 ± 0.02
spirv.core.grammar	1.31 ± 0.23
gsoc-2018	4.99 ± 0.34
refsnp-chrMT	76.09 ± 1.80
refsnp-other-100K	514.64 ± 4.67

Table C.4. Results of the simdjson JSON parser.

Data Set	Experiment	Upload	Lexical Analysis	Parsing	Building parse tree	Syntax processing	Semantic analysis	Total
`pareas-0`	cpu (1 thread)	3.16 ± 0.55	0.39 ± 0.06	0.96 ± 0.17	0.60 ± 0.10	1.01 ± 0.18	1.25 ± 0.22	7.59 ± 1.18
	cpu (4 threads)	3.43 ± 0.67	0.59 ± 0.06	0.95 ± 0.09	0.44 ± 0.05	2.03 ± 0.23	3.16 ± 0.36	10.81 ± 0.97
	cpu (16 threads)	3.90 ± 0.70	1.09 ± 0.13	1.92 ± 0.19	0.88 ± 0.15	5.69 ± 0.61	7.03 ± 0.64	20.80 ± 1.12
	cpu (64 threads)	3.88 ± 0.60	2.94 ± 0.24	5.13 ± 0.41	3.14 ± 0.89	29.69 ± 3.50	27.03 ± 4.51	72.14 ± 7.77
	gpu	2.21 ± 0.05	0.14 ± 0.00	0.28 ± 0.01	0.09 ± 0.00	0.65 ± 0.01	2.22 ± 0.02	5.61 ± 0.08
`pareas-1`	cpu (1 thread)	2.99 ± 0.65	16.29 ± 1.93	36.10 ± 0.37	24.53 ± 0.11	35.53 ± 0.14	43.27 ± 0.28	158.94 ± 2.40
	cpu (4 threads)	3.49 ± 0.47	12.84 ± 2.29	23.28 ± 2.97	11.54 ± 1.57	20.10 ± 3.71	28.76 ± 6.15	100.30 ± 11.60
	cpu (16 threads)	3.94 ± 0.63	9.86 ± 1.02	19.11 ± 1.23	8.33 ± 1.12	16.15 ± 2.38	31.49 ± 4.73	89.21 ± 8.77
	cpu (64 threads)	4.22 ± 0.82	14.22 ± 1.74	26.27 ± 2.36	13.01 ± 1.08	40.84 ± 3.54	103.20 ± 7.80	202.18 ± 11.61
	gpu	2.49 ± 0.02	0.65 ± 0.01	0.73 ± 0.01	0.18 ± 0.00	1.00 ± 0.01	2.74 ± 0.02	7.82 ± 0.03
`pareas-2`	cpu (1 thread)	2.80 ± 0.28	70.51 ± 1.76	197.77 ± 1.74	132.09 ± 1.26	203.89 ± 0.28	267.83 ± 1.64	875.30 ± 3.21
	cpu (4 threads)	4.06 ± 0.56	54.36 ± 7.45	100.70 ± 6.47	52.72 ± 4.18	91.04 ± 10.88	136.61 ± 16.57	440.28 ± 25.69
	cpu (16 threads)	4.38 ± 0.41	41.98 ± 2.49	82.89 ± 3.23	28.16 ± 2.98	56.85 ± 4.31	82.14 ± 8.05	297.10 ± 14.49
	cpu (64 threads)	4.72 ± 0.71	40.98 ± 3.26	75.68 ± 3.73	31.69 ± 3.67	61.84 ± 5.68	160.84 ± 13.00	376.72 ± 22.77
	gpu	3.55 ± 0.52	1.23 ± 0.55	1.75 ± 0.18	0.53 ± 0.00	2.40 ± 0.06	4.28 ± 0.55	13.76 ± 1.72
`pareas-3`	cpu (1 thread)	6.30 ± 0.39	424.42 ± 2.61	1153.38 ± 5.47	906.13 ± 11.10	1385.47 ± 16.93	2072.58 ± 7.93	5948.72 ± 32.98
	cpu (4 threads)	6.62 ± 0.81	250.32 ± 11.35	458.66 ± 23.61	364.68 ± 17.66	493.35 ± 30.09	881.37 ± 75.07	2455.87 ± 116.57
	cpu (16 threads)	7.14 ± 0.59	164.14 ± 8.61	345.89 ± 14.88	215.81 ± 11.40	283.99 ± 26.60	452.64 ± 43.71	1470.52 ± 82.89
	cpu (64 threads)	7.29 ± 0.83	100.50 ± 6.25	282.90 ± 10.23	198.34 ± 15.29	253.53 ± 14.60	475.23 ± 20.32	1318.77 ± 40.65
	gpu	7.03 ± 0.26	3.23 ± 0.04	6.88 ± 0.03	3.03 ± 0.00	10.74 ± 0.04	16.87 ± 0.04	47.79 ± 0.38
`pareas-4`	cpu (1 thread)	28.78 ± 0.27	3688.70 ± 39.34	9870.49 ± 37.59	7874.13 ± 29.02	13895.99 ± 40.99	24600.59 ± 55.80	59960.99 ± 156.62
	cpu (4 threads)	30.50 ± 1.75	1624.17 ± 17.18	3595.96 ± 62.09	2945.98 ± 85.71	4458.61 ± 84.33	9459.53 ± 403.38	22117.60 ± 560.69
	cpu (16 threads)	30.59 ± 1.82	1057.77 ± 94.87	2277.58 ± 105.75	1906.30 ± 265.39	2164.98 ± 92.92	5041.75 ± 121.45	12482.23 ± 354.97
	cpu (64 threads)	30.92 ± 1.45	531.75 ± 19.67	2013.68 ± 67.23	1514.92 ± 88.40	1907.04 ± 57.14	4289.22 ± 106.66	10291.09 ± 174.31
	gpu	40.74 ± 0.33	24.66 ± 1.67	57.90 ± 0.04	24.71 ± 0.04	89.43 ± 0.81	148.64 ± 0.59	386.10 ± 1.95

Table C.5. Results of compiler experiments.

Data Set	Experiment	Upload	Lexical Analysis	Parsing	Building parse tree	Restructuring	Total
`twitter_api_response`	cpu (1 thread)	9.80 ± 1.24	0.65 ± 0.11	0.26 ± 0.03	0.07 ± 0.01	0.12 ± 0.01	1.10 ± 0.16
	cpu (4 threads)	11.40 ± 0.96	0.81 ± 0.07	0.55 ± 0.04	0.22 ± 0.03	0.35 ± 0.05	1.93 ± 0.10
	cpu (16 threads)	12.30 ± 1.31	1.14 ± 0.14	1.41 ± 0.11	0.78 ± 0.13	0.93 ± 0.49	4.27 ± 0.49
	cpu (64 threads)	13.35 ± 0.69	2.98 ± 0.21	4.71 ± 0.42	2.97 ± 0.87	2.94 ± 0.44	13.61 ± 1.42
	gpu	9.22 ± 0.53	0.15 ± 0.01	0.29 ± 0.01	0.09 ± 0.00	0.12 ± 0.00	0.65 ± 0.03
`spirv.core.grammar`	cpu (1 thread)	11.51 ± 1.14	11.55 ± 2.12	6.44 ± 0.08	1.47 ± 0.00	2.51 ± 0.01	21.98 ± 2.13
	cpu (4 threads)	12.20 ± 0.71	9.06 ± 1.03	5.66 ± 0.52	1.25 ± 0.11	2.08 ± 0.27	18.05 ± 1.68
	cpu (16 threads)	13.48 ± 0.86	7.14 ± 0.39	5.30 ± 0.64	1.50 ± 0.16	2.13 ± 0.16	16.08 ± 1.10
	cpu (64 threads)	12.94 ± 0.62	9.38 ± 0.77	10.16 ± 1.74	5.14 ± 1.32	7.65 ± 1.05	32.36 ± 3.41
	gpu	9.24 ± 0.46	1.01 ± 2.35	0.53 ± 0.05	0.13 ± 0.01	0.14 ± 0.01	1.82 ± 2.41
`gsoc-2018`	cpu (1 thread)	11.74 ± 0.61	56.52 ± 0.30	7.24 ± 0.15	1.59 ± 0.00	3.12 ± 0.02	68.48 ± 0.34
	cpu (4 threads)	8.93 ± 1.21	40.92 ± 7.48	4.52 ± 0.60	1.15 ± 0.18	1.90 ± 0.36	48.50 ± 8.19
	cpu (16 threads)	8.33 ± 0.51	30.01 ± 1.79	4.39 ± 0.48	1.32 ± 0.16	2.09 ± 0.26	37.81 ± 1.61
	cpu (64 threads)	12.08 ± 1.50	28.22 ± 3.13	10.87 ± 1.61	6.26 ± 1.35	8.29 ± 1.12	53.64 ± 4.12
	gpu	9.69 ± 0.25	0.97 ± 0.25	0.77 ± 1.33	0.13 ± 0.01	0.14 ± 0.01	2.01 ± 1.59
`refsnp-chrMT`	cpu (1 thread)	21.52 ± 0.15	1167.66 ± 6.18	1139.20 ± 2.96	328.44 ± 2.10	879.77 ± 1.82	3515.09 ± 8.22
	cpu (4 threads)	18.51 ± 1.96	575.62 ± 23.75	450.50 ± 18.51	155.20 ± 12.54	270.60 ± 14.37	1451.96 ± 45.77
	cpu (16 threads)	17.63 ± 1.32	398.12 ± 21.97	326.94 ± 20.44	102.58 ± 8.68	170.56 ± 21.58	998.21 ± 57.17
	cpu (64 threads)	20.40 ± 1.95	183.61 ± 9.82	276.53 ± 11.86	99.96 ± 10.73	156.14 ± 12.20	716.26 ± 34.12
	gpu	23.80 ± 0.32	9.19 ± 0.17	8.51 ± 2.68	1.80 ± 0.00	5.81 ± 0.00	25.32 ± 2.86
`refsnp-other-100K`	cpu (1 thread)	74.03 ± 0.28	7826.78 ± 52.42	7818.78 ± 38.07	2374.48 ± 11.03	6524.65 ± 12.96	24544.71 ± 81.52
	cpu (4 threads)	71.80 ± 1.00	3372.89 ± 38.33	2926.22 ± 51.51	1011.56 ± 33.93	1837.31 ± 39.27	9148.07 ± 116.77
	cpu (16 threads)	73.18 ± 1.77	1624.25 ± 122.42	1853.54 ± 66.35	687.21 ± 57.78	784.61 ± 59.02	4949.69 ± 186.14
	cpu (64 threads)	73.50 ± 1.94	905.19 ± 41.37	1774.56 ± 56.09	651.16 ± 37.99	633.12 ± 34.44	3964.12 ± 92.82
	gpu	107.43 ± 1.66	58.41 ± 0.36	56.86 ± 1.73	12.20 ± 0.24	42.11 ± 0.36	169.59 ± 1.71

Table C.6. Results of JSON parser experiments.