Lazarus: Where Compilers Meet Education

Rui Gonçalves
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

Pedro Rangel Henriques
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

José Carlos Ramalho
ALGORITMI Research Centre / LASI, Dept. of Informatics, University of Minho, Braga, Portugal

Abstract

Lazarus is an intentionally compact platform featuring an optimizing compiler, a lightweight virtual machine and IDE integration. Its small size, lets students explore every compilation stage by experimenting with a compiler inspired by modern systems. At the core lies an incremental parser, that only re-parses the surrounding areas of the changed text in an IDE environment, and an SSA-based optimizer. The target machine is a register-based virtual machine, that mimics native architectures, and provides a debugger. This paper presents both the source language, and its register-based virtual machine, while discussing an often neglected feature in compiler design: the incremental parser.

2012 ACM Subject Classification Software and its engineering → Object oriented languages; Software and its engineering → Parsers; Software and its engineering → Compilers; Software and its engineering → Source code generation; Software and its engineering → Virtual machines

Keywords and phrases Object-Oriented Programming, Compilation, SSA, Optimization, Virtual Machine, IDE

Digital Object Identifier 10.4230/OASIcs.SLATE.2026.19

Supplementary Material *Software (Web Site)*: `https://lazarus.epl.di.uminho.pt`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/2025 – Centro ALGORITMI (ALGORITMI/UM) `https://doi.org/10.54499/UID/00319/2025.`

Acknowledgements The authors would like to thank Neal Gafter and Eric Lippert (from the Roslyn team) for their valuable support, for answering numerous questions, and enabling some features presented in this work; and acknowledge the support and help of our families during this time.

1 Introduction

Computer science degrees often include courses on language design and implementation intended to instruct students on the process of specifying formal languages through the usage of context-free grammars, as well as the processes used to analyze and handle source code. Under this context, programming languages are usually the subject of a course's practical work, although in a simplified form, such as subsets of C or Forth. Given the inherent theory and complexity of compilers, which are not the focus of the class, they are often neglected and replaced by abstract models, such as Intermediate Representations (IRs) and Stack-Based Virtual Machines. While these serve their purpose in education and non-academic environments, the backend components and processes mostly remain as black boxes, creating a very large gap in understanding.

Furthermore, in recent decades, "Programming Languages" have undergone drastic changes facilitated by increased computational resources. Compilers can now (and are expected to) empower code editors with sophisticated tooling tied to the language ecosystem,

such as providing type information, inline documentation, real-time diagnostics, auto-fix suggestions, auto-completion, refactoring primitives, and much more. Moreover, this tooling is the cornerstone of a new wave of revolution in the programming language and framework ecosystems. It is directly exposed via MCP (Model Context Protocol) [2, 3] to AI agents, such as Dart (and Flutter) MCP [24], or in other novel approaches where languages are built specifically for this purpose: the so-called AI-native languages like MoonBit [18].

This shift has created a significant architectural divergence compared to the academic models taught in these courses or classical compiler textbooks regarded as the "gold standard" worldwide, such as The Dragon Book [1] and Appel [4]; these are now quickly becoming out-of-sync with industry practice. This issue is even more severe in other areas of the spectrum: optimization, machine code generation, debugging, etc., because of their increased complexity and lower-level nature. Moreover, while many compilers rely on the LLVM infrastructure [32, 34], which is not beginner-friendly to study and modify without prior knowledge and experience. This led to the creation of a simpler, smaller, and easier-to-modify programming language infrastructure introduced in this paper.

This paper is organized in seven sections. Section 1 presents the project context and motivation. Section 2 provides an overview of related work and industry practice. Section 3 presents a brief description of the source language. Section 4 goes into detail of the architecture of the virtual machine. Section 5 introduces the compiler/infrastructure architecture. Section 6 explores the incremental parser algorithm. Finally, section 7 closes the paper with the conclusions and possible future directions for the project.

2 State of the Art

A compiler is divided into three main parts: the front-end, middle-end, and back-end. The front-end deals with the source language itself; it *reads* the input text (source program), validates it, and extracts its semantic meaning. The middle-end performs optimization passes on the source program aimed at improving the program execution performance. The back-end is responsible for producing assembly code. To complete the Compilation process, there is still the need for an assembler that translates the assembly code generated by the compiler into an object file (LLVM actually bundles the assembler and produces the object file directly from its representation, skipping the translations). Additionally, a mechanism to resolve external symbols is needed, such as a linker in C or dynamic resolution at runtime as in Java. For compiled programs to be executed, they need a runtime environment, such as an operating system or virtual machine, as well as standard libraries that might have been linked with.

2.1 Compiler Front-End

The front-end is the component used for the interface with code editors. Microsoft developed the Language Server Protocol (LSP) [38] as a standardized way for different code editors to request common and expected features from the front-end. Moreover, since the source code document is constantly changing and those changes must propagate to the front-end, meaning the front-end itself must also efficiently represent text in a format that is both easy and convenient for the analysis it performs as well as for user editing. Text editors usually use variants of *Line Arrays*, *Piece Tables*, *Gap Buffers*, and *Ropes* [36, 7, 6]; which may be augmented by extra metadata to help with line breaks and encoding formats. *Line Arrays* keep an array where each position is a string representation of the line. *Gap Buffers* operate on the premise that multiple consecutive user operations are performed around the

same position, and as such when an insert operation is performed it allocates space for the successive insertions. *Piece Tables* try to optimize a requirement of code editors, the undo operation, and essentially keeps the original source and tracks the changes, requiring multiple jumps to query the up-to-date source, but reducing overall editor memory. *Ropes* on the other hand operate on balanced tree structures, making an insertion a join of nodes, this structure has the great advantage of immutability, making it one of the ideal options for front-ends.

The first step in a front-end is to take the input textual stream and produce tokens (like words in English), this is lexical analysis. Then comes parsing, checks the token sequence against grammar rules and produces a representation of the input program in a tree form: the syntax tree. Traditionally, this tree removes redundant information (e.g., spacing, comments, parentheses around conditions, or in math expressions), creating an abstract (not concrete) representation: the Abstract Syntax Tree (AST). On the other side of the spectrum, Roslyn (the C# compiler) [37] produces a tree containing every syntactical detail (a 1:1 mapping), known as Concrete Syntax Tree (CST). Compilers such as Go tend to opt for a middle ground; the tree still contains details like parentheses, and comments (kept in a separate table in Go), but spacing, line breaks, and other non-essential information for the compiler are discarded.

To create a parser, a language must first be described by a grammar, which defines the productions (rules) used both for generation (the derivation of language sentences) and for recognition. The rules make use of tokens (terminal symbols), and non-terminal symbols. Two main core parsing stratagies exist: one is Top-Down (TD) – usually implemented by parsing algorithms of the LL family – where the structure is built from the beginning (top of the tree) to the end (bottom); for example, a sentence must contain a subject and a predicate, and so on down to the word. The other is Bottom-Up (BU) – usually implemented by parsing algorithms of the LR family – that analyze input terminals until recognize the right-side of a production reducing, at this moment, the input sequence to the non-terminal in the left-side of that production; and proceed so far, so on, until be able to reduce the complete input sequence to a non-terminal that is the root of the tree. For example, a name is a subject, and the subject plus a predicate form a sentence (the sentence is only recognized (accepted) when both elements are found). The needed grammars can be represented in common formats such as EBNF [26], which will be adopted throughout this paper.

Given that creating a lexer/parser for a set of grammar rules is a systematic and repetitive work, several amazing parser (and lexer) generators were developed. The most famous ones are Lex/Yacc (the eldest, inspirational, and the most used around the world by the C community[1]), Tree Sitter [9] (an incremental parsing library/generator developed for the Atom code editor based on Generalized LR theory) and ANTLR [43] (Another Tool for Language Recognition based on Adaptive LL theory)[2]. Incremental parsers are those capable of being aware of a change (from a code editor) and producing a brand-new CST by piecing together a previously built CST with minimal lexing/parsing, covering just the changed region. While these generators are very capable, most commercial-grade compilers end up handwriting a Recursive Descent Parser (follows the TD strategy, but resorts to several tricks to solve ambiguities, including but not limited to arbitrary look-ahead, LR-like behavior, and semantic/syntactical aware parsing/lexing). The choice for RD parsers happens because they are easy to make, capable of better/custom error recovery and messages, and feature gating – the ability to control access to language features, as is the case of versioning, variants (e.g., a single compiler for all C-family languages), enabling experimental language features.

[1] Nowadays available for Python as a module: PLY.
[2] A similar version is also, at present, available for the Python community as a module: Lark.

The next step is semantic analysis or binding, where the syntax tree is processed and types and other contextual conditions (semantic constraints) are checked. Most compilers tend to store this information in one of two places: the syntax tree itself (also called the annotated syntax tree) or by producing a brand-new bound tree. Most compilers end up doing multiple passes through the entire tree, performing different semantic analysis tasks: declaring classes, fields, and methods; then expression-level type checking. In these pipelines, each step creates/augments a symbol based on previously known symbols (variables, functions, classes, or anything relevant) and stores them in a centralized structure: the symbol table. Symbols generally use lazy evaluation, and there are multiple passes to avoid ordering problems (e.g., a function using another function declared later). In recent times, several compilers have also adopted Hindley-Milner [25, 40] based type systems that employ constraint solving systems to automatically deduce types. Moreover, most compilers cache the values computed along these passes (or parts of the passes; or much more simply on a per file/library basis) so they can be used for incremental semantic analysis.

2.2 Compiler Middle-End

For the rest of the compilation pipeline, most brand-new compilers tend to rely on the LLVM infrastructure, although MSVC, GCC, Go, and other compilers still prefer to build their own (for better compilation time, flexibility, simplicity, and other reasons). These systems work on an Intermediate Representation (IR) that the front-end must emit, to allow for optimizations to be constructed on a common infrastructure and tooling. In the past, IRs were high-level and quite similar to the source language, but nowadays most IRs opt for being low-level. This generalizes and canonicalizes the source, making it uniform (no semantic-specific constructs), enabling easier reasoning and transformation, which in turn improves the correctness of optimizations and the generalization (more powerful) and composability of them. Furthermore, some optimizations always require target-specific information (e.g., vectorization); as such, it is common to specify target details, such as architecture, extensions, operating system, and others; and pass them along the optimization pipeline.

IRs nowadays also use sparse representation that encode directly variable *def-use chains* (what variable uses a certain definition) and *use-def chains*. This is augmented by variable immutability (and a special Phi function for merging control flow branches), known as Static Single Assignment (SSA) form [16, 45]. To convert to this form, it is necessary to insert Phi instructions at control flow join points to merge incoming values; the set of nodes where such insertions may be required is the dominance frontier [44, 35]. This set can be calculated in multiple ways and may use sophisticated algorithms and data structures such as the DJ-Graph [47, 33]. Given the complexity of this task, LLVM introduced a convenient way for front-ends not to have to implement this logic every time; it uses a "memory-based" form (stack allocations and read/write operations) whose variables are then promoted to SSA registers via the *mem2reg* pass. This form is also convenient for calculating liveness information, which indicates what variables are live at the end of a block (needed by a successor block) and in the beginning of a block (used in the block or a successor), as well as other data-flow analyzes.

Most compilers also implement hundreds of small optimizations for different use cases and patterns, making their study an impossible task. As such, only a handful of optimizations considered to be some of the main ones will be discussed. Sparse Conditional Constant Propagation [51] performs a form of Dead Code Elimination (removing unused branches/variables) and evaluates constant expressions together to improve the accuracy of constant value detection. Global Value Numbering (GVN)/Global Code Motion (GCM) [10] merge redund-

ant operations that compute the same value and hoist code to better places (e.g., outside loops), performing a form of redundancy elimination and loop-invariant code motion. More general forms of redundancy elimination, such as partial redundancy, where a computation may occur in one but not all execution paths, can augment the GVN algorithm [48]. This is the reasoning behind the deprecation of more classical algorithms for the task, such as SSAPRE [41, 27, 28]. With the widespread use of streams in languages, algorithms have also been developed to perform stream fusion, which combines multiple passes over data into a single pass and conversion to standard imperative constructs [29].

The question of the order of optimizations led to the combination of different sorts of optimizations into singular passes, in an attempt to improve compilation speed and effectiveness [13, 15]. Moreover, GVN/GCM optimizations require a particular internal structure: a graph where the notion of definitions is discarded since edges represent usage. This internal structure (which still holds to the SSA property) has been adopted as the core intermediate representation of some compilers and is named the Sea of Nodes [14, 11, 12]. This representation led to the development of new SSA conversion algorithms that require no prior analysis [8], and more advanced peephole optimizations (optimizations requiring no prior analysis), enabling optimizing one-pass front-ends where parsing directly produces the IR (which is automatically optimized on creation), and type checking is also performed on the IR itself before iterating through more advanced optimizations. This approach generally produces much higher-quality code faster, but its main disadvantage for traditional compilers is that GCM is non-optionally integrated, making debugging much more difficult.

2.3 Compiler Target Machine

Before continuing with the pipeline it is important to better understand the general target of a compiler. A CPU or virtual machine can only execute a defined set of instructions, which operate on specific data types and registers and access memory in particular ways [42]. All instructions must also follow a specific encoding format, and the Instruction Set Architecture (ISA) defines all of these aspects. On top of the hardware layer itself, there is usually an operating system [5]; for all intents and purposes, it is just like a big virtual machine that exposes an interface to access system resources via system calls. The program itself is not just code; it also contains other information such as imported/exported symbols, libraries to load, signatures, debug information, and other metadata. That referred information is stored in the image/container format (e.g., PE (.exe/.dll) in Windows and ELF in Linux/other Unix-like systems). Moreover, there are several other details that need to be specified, such as naming conventions, calling conventions, stack layout, exception models, among others. These details about how a compiled program interacts with the OS, hardware, and other binaries are called the Application Binary Interface (ABI).

Virtual machines provide the ability to run isolated environments of an ABI that may correspond to a physical ABI (e.g., x86_64, ARM) or a completely virtual one. Several compilers for well-known programming languages (e.g., Java, C#, F#, Visual Basic, Python, etc.) resort to a virtual machine approach for some of its advantages, such as *portability* – which Java popularized with the "write once run everywhere" slogan – or *performance* reasons (in comparison to tree-walk interpreters). These virtual machines can include extra facilities compared to real architectures, such as an unlimited number of registers or "locals", or no registers at all (just a stack). Furthermore, the virtual machine can be augmented to compile the running application to the native architecture on demand. Essentially, this "bundles a compiler" (a Just-in-Time (JIT) Compiler) which can use its knowledge about the running program (e.g., frequently executed paths) that would not be known without runtime support, to *safely* optimize the program more aggressively than an ahead-of-time compiler ever could.

Beyond the image that the runtime environment executes, there is an additional layer of metadata designed to enable debugging. This information provides a mapping between logical language constructs (e.g., variables and parameters) and physical ones (e.g., memory/stack locations and registers), as well as mapping addresses to line information, among others [17]. Virtual machines allow for much richer debug information since they do not have the constraints of physical architectures. This flexibility enables specialized ways to simplify this mapping, such as custom debug-specific instructions that provide fine-grained information, or even high-level constructs such as classes being delayed into the runtime with all their information. Furthermore, virtual machines typically execute de-optimized code during debugging, rolling back from highly optimized code that is executing. This allows the usage of an almost direct correspondence from the source code, significantly enhancing the debugging experience and enabling more complex features [31, 30]. While it is technically possible to achieve the same in native architectures, doing so would introduce substantial instrumentation overhead, even more than the already present one (the operating system's middle-man overhead). To bridge the gap between debuggers and code editors, there are several protocols, such as the Debug Adapter Protocol (DAP) [39], that standardize common functionality; but given the versatility of some debuggers and the features they provide, the usage of these protocols can be limited. Certain debuggers provide powerful tools – such as, time-travel debugging (stepping backwards), predictive debugging (seeing future execution paths when execution is paused), custom data visualizers (displaying 3D models) – which are highly specific to certain use cases of that specific debugger and cannot be expected to be standardized.

2.4 Compiler Back-end

Returning to the compilation pipeline, the back-end must take the mostly architecture-agnostic IR and produce the final target-dependent assembly. The journey begins with instruction selection, where IR instructions are converted into their target equivalents. This task presents some interesting aspects. The first is the possibility of a non-direct mapping: a singular native instruction might merge multiple IR ones (e.g., in x86_64, the Load Effective Address instruction is usually used to replace multiple arithmetic operations that follow the pattern $base + index * scale + offset$), or vice versa. The other interesting detail is that the order in which instructions are presented to the CPU might affect performance (because of hardware details like pipelining). It is very common to perform scheduling after selection for this purpose, or to take these patterns into account during selection itself. Conceptually speaking, this step is quite simple, and even simple tree-pattern matching [19] can do it. Register allocation usually follows, and this is arguably the most intricate step of the back-end. In this step, the compiler must assign a place to variables, and when there is no room for the variables, they must be temporarily stored in the memory stack. This problem is NP-complete, so heuristics and other mechanisms are used. There are some simplistic algorithms that yield excellent results, like Linear Scan [47], which have been used successfully in both compilers and JIT-based virtual machines. Other approaches involve the equivalence to the graph-coloring problem and do exactly that, such as Chaitin-Briggs style, Iterated Register Allocation [22, 46], and some of its generalizations. Other approaches even take advantage of the SSA property and use chordal graphs to solve the problem. This last approach raises an interesting question: where is the best place to destroy (remove) SSA? This destruction process involves replacing Phis with copy instructions on incoming blocks, and some compilers prefer to perform this even before instruction selection. Although destroying during the register allocation process has clear advantages for analysis, and some

back-ends opt for this approach as well. While there are always variations to the pipeline described, and every single compiler ends up making significant architectural changes to it, this is especially true in the back-end.

3 Language

The high-level programming language to be compiled, named Lazarus (after the infrastructure[3]), is designed as a statically typed, imperative, and object-oriented programming language with a focus on familiarity and simplicity. The built-in atomic types include `String`, `Number`, and `Boolean`. The numbers are represented by double-precision floating-point values (IEEE 754), just like JavaScript. Moreover, the language has array support and is also garbage collected. The standard control-flow structures are supported, such as the `if-else`, `for`, `while` and `do` statements. The variables are mostly type-inferred, although the return type and argument types still require explicit type declarations. The language also provides some extra native functionality for writing to the console and some basic graphical capabilities. It uses braces and semicolons, allowing for an easy transition from C# (the language in which the infrastructure was written), although it deviates from some syntax choices of C-style to simplify the compiler itself. The most significant change is found in the generic syntax, which removes the *less than* (`<`) and *greater than* (`>`) symbols, given that they would introduce ambiguities into the language. The generic support is used to provide the built-in types, in particular the `Nullable@(T)` and `Array@(T)`. Given the large scope of the language, only a small subset is described. Currently, the language accepts only a single input file, which must be composed of declarations (classes or functions), described by the top level grammar in Listing 1.

Listing 1 Top-level language grammar.

```
ClassDeclaration →
| "class" ID GenericParameterDeclaration? ClassExtends? ClassBody
ClassExtends → "extends" TypeExpression
ClassBody → "{" ClassMember* "}"
ClassMember → Field | Method
Field → ID Typing ";"
Method → ("virtual" | "override")? Function
FunctionDeclaration → "fun" Function
Function →
| ID GenericParameterDeclaration? "(" ParameterList? ")" Typing? Block
ParameterList → Parameter ("," Parameter)*
Parameter → ID Typing
Typing → ":" TypeExpression
TypeExpression →
| "fun" "(" TypeExpressionList? ")" Typing? "?"?
| ID GenericArgument? "?"?
TypeExpressionList → TypeExpression ("," TypeExpression)*
GenericParameterDeclaration → "@" "(" ID ("," ID)* ")"
GenericArgument → "@" "(" TypeExpression ("," TypeExpression)* ")"
```

[3] It came to our attention that Lazarus is an established name in the industry, yet we opted to retain it, as it was chosen as an allusion to the biblical narrative, and the project's revival from a prior attempt.

The language intentionally leaves out access modifiers, which are helpful in large-scale systems but mostly useless in small-scale "scripts", which this language aims for. Having said that, by convention, fields starting with the underscore character are considered private (but are accessible nonetheless). The only modifiers available in the language are *virtual* and *override*, which are applicable only to methods. A virtual method indicates that the method can be overwritten by another class; such a class must specify the intent using the *override* keyword. This choice allows for exact control over the methods present in the tables required for virtual dispatching. The typing system also natively uses nullable types, which is syntactic sugar for the $Nullable@(T)$ class type, and function pointers, although it can be easily extended to support closures down the road. Regarding generics, generic arguments cannot be refined as of now, since there is no interface or trait support; this presents a challenge for real-use cases of the language.

Listing 2 shows a simple program written in Lazarus to illustrate the basics of the language. The example is a function (the main one) to print a header text and the numbers between 1 and 5, one per line, using a "for" loop. Listing 3 shows a more complex example, by computing the number of primes from 1 to 20 using the Eratosthenes Sieve.

Listing 2 Minimal program sample.

```
fun main() {
    print("Hello World!\n");

    for (var i = 1; i <= 5; i = i + 1) {
        print("Number $i\n");
    }
}
```

Listing 3 Eratosthenes Sieve in Lazarus.

```
class PrimeEratosthenesSieve {
    n: Number;
    isPrime: Array@(Boolean);

    init(nbr: Number) {
        n = nbr;
        isPrime = new Array@(Boolean)(n + 1, true);
        isPrime.set(0, false);
        isPrime.set(1, false);
    }

    run() {
        for (var i = 2; i * i <= n; i = i + 1) {
            if (!isPrime.at(i)) continue;

            for (var j = i * i; j <= n; j = j + i)
                isPrime.set(j, false);
        }
    }

    countPrimes(): Number {
        var count = 0;

        for (var i = 0; i < isPrime.length; i = i + 1)
            if (isPrime.at(i))
```

```
                count = count + 1;

        return count;
    }
}

fun main() {
    var nbr = 20;
    var sieve = new PrimeEratosthenesSieve(nbr);
    sieve.run();
    var count = sieve.countPrimes();

    print("There are $count primes from 1 to $nbr");
}
```

3.1 Language Analysis

The language itself is parsed by a handwritten recursive descent parser, which makes use of a precedence table for correctly parsing expressions. Moreover, this parser employs a custom error recovery mechanism that attempts to never consume tokens beyond a statement, by keeping a global recovery synchronization set with the token that close and open new statements, and not crossing that boundary when already in panic mode. The parser produces a CST (Concrete Syntax Tree) [23], by keeping every error and whitespace (trivia tokens) attached the tokens in the tree, enabling the code formatter and the incremental parser, improving the developer experience and productivity. The exact internals of the incremental nature of the parsing algorithm will be discussed later.

The semantic analysis is composed of multiple passes. First, the class declaration pass declares all available classes, making it possible to reference them later even if their declaration is present after usage. Next is the class inheritance pass, which maps and checks the possibility of the class hierarchy. Symbol information is created in a lazy fashion; meaning generic class fields and methods are only physically created when they are queried, making it possible to write code such as `class A extends B@(C)` even though the fields and methods were not declared (this occurs in the next pass). Moreover, since virtual functions might not have been declared, it is not possible to determine if an override method is possible; therefore, a following pass dedicated solely to these checks is necessary. Last, the function pass performs type checking and produces a bound tree with this information.

4 Virtual Machine

The Virtual Machine was designed to be a lightweight alternative to native architectures, allowing cross-platform execution without heavy emulators. Moreover, this approach proves useful as it allows for the study of debuggers without relying on operating systems and their complex debugging APIs. This machine is register-based; most instructions cannot operate on memory directly, requiring values to be loaded into registers or being immediate. Furthermore, the program has access to a stack where temporary values (e.g., spilled variables) can be placed and arguments can be passed to functions with a variable number of parameters (which are called variadics). While this stack is traditionally also used for call management in native architectures, it was opted to separate it into an implicit stack, ensuring that a program cannot accidentally corrupt this information and minimizing function prologues and epilogues. The program can also allocate memory on the heap; in which case, the memory is automatically garbage collected. Since no symbolic names are used by the virtual

machine, access to data in these heap-allocated structs must be done by offsetting the base value. Moreover, since all kinds of values are considered boxed (reference types), this offset corresponds to the field index. A program itself has multiple sections, mainly the code and data sections, which contain the instructions to be executed and constant strings respectively. This data section can be extended in the future to allow for constant data besides strings to enable jump tables commonly used by switch statements, virtual dispatch tables (which are currently created at instantiation time), richer typing tables, among other common use cases. Figure 1 contains a visual representation of the described architecture.

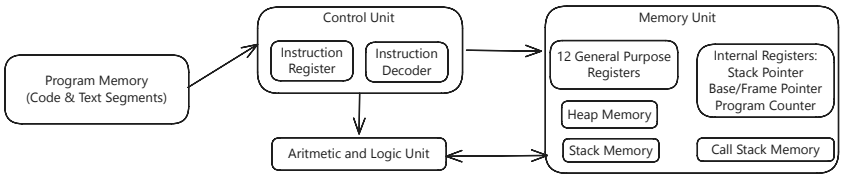

Figure 1 Virtual Machine Architecture Diagram.

The Instruction Set Architecture of this machine contains several opcodes to perform standard **arithmetic and logic operations**, to perform **calling** and **returning**, **nullability checks**, **stack management**, **jumping**, to **allocate** and **operate on heap structures**, and some extra special instructions meant to interface with the user (the **system call** instruction) and to call the **debugger** (equivalent to the *int3* instruction in x86, which is used to call the debug exception handler and is inserted by the debugger; this will be explained later). The encoding schema for the instructions has a variable length depending on the variant of the instruction. An instruction can contain *up to three arguments*, and only *two of them can vary* (such as *immediate* or *register*). This variant information is stored in the two upper bits, leaving the rest of the byte for the opcode; this, in turn, enables up to $2^6 = 64$ opcodes. The actual argument encoding follows this byte, and immediate values are stored in little-endian order. The architecture follows a Reduced Instruction Set Architecture (RISC) style, preferring simple instructions rather than more compact and efficient ones. In the future, this machine can be extended to support a second register class for vector registers and the corresponding arithmetic and logic instructions to operate on them, enabling the study of Single Instruction Multiple Data (SIMD) used by vectorization optimizations. The compiled code is loaded from a container format that can hold more than the two aforementioned sections (the code and strings). These two extra sections hold linking and debugging information. The linking section contains information about exported symbols and imported symbols, enabling the linker to effectively resolve locations and produce an executable. The debug tables contain line, variable, typing, and other information based on a subset of the DWARF standard. Listing 4 contains an implementation in the virtual machine assembly for the program in Listing 2.

Listing 4 Minimal assembly sample.

```
.strings
    "Hello World!\n"
    "Number "
    "\n"
.end

entry main:
    mov #0 r8           ; moves "Hello World!\n" to register 8
    syscall 0           ; call print (register 8 is printed)
    mov 1 r9            ; i = 1
```

```
loopCond:
    leF64 r9 5 r8       ; i <= 5
    jumpIf @loopBody @loopExit r8
loopBody:
    mov #1 r8           ; moves "Number " to register 8
    f64ToStr r9 r10     ; converts variable i to string
    addStr r8 r10 r8    ; interpolates the string
    mov #2 r10          ; moves "\n" to register 10
    addStr r8 r10 r8    ; interpolates the string
    syscall 0           ; call print
    addF64 r9 1 r9      ; i = i + 1
    jump loopCond
loopExit:
    return
```

The virtual machine's debugging approach also mimics that of production-ready native debuggers. The typical way these debuggers handle breakpoints is by replacing the instruction with an interrupt (like *int3* in x86, already mentioned). When the Control Unit hits that program instruction, the operating system is called and transfers execution to the debugger. That instruction must be the shortest possible (*int3* is one byte) to ensure it does not overlap with a second instruction, while the original is kept in the debugger, ensuring it can then continue execution. This virtual machine uses the same approach, just streamlined by the fact that there is no middleman between the debuggee and the debugger. Standard stepping is implemented based on the same principles: *step in* executes instructions until the line changes, while *step over* does the same but also decodes the current instruction to take into account calls. Native debuggers sometimes end up disassembling the machine code and speculating where the control flow is going to end (faster than single-step instruction execution because of operating system overhead), which was again streamlined in this debugger. *Step out* is the most interesting one: it is not possible to simply take the return address and insert a temporary breakpoint there because of recursion; so, instead, it modifies the call stack value to a virtual machine-controlled address (a breakpoint instruction outside code space) so that when hit, the debugger knows it can replace the program counter. Native architectures use a similar approach but tend to replace it with an invalid address instead (since the operating system notifies the debugger).

5 Architecture

The compiler front-end follows the same architecture as Roslyn and is quite similar in many details, with several internal APIs being almost identical. The middle-end and back-end were inspired by LLVM, but the chosen architecture significantly differs from it. Figure 2 contains a diagram of the high-level overview of the final architecture of the compiler infrastructure. The "Common" submodule contains the internal data structures for text representation and diagnostic information. These structures are used by the rest of the infrastructure, including the actual LSP/DAP, which automatically translates to/from the protocol specific to each. The Frontend submodule contains everything related to the Lazarus language itself, ranging from representations to the documentation processing engine, and some services, such as the code formatter. The Middle/Back-end contains the IR representation, as well as the implementation of different optimization passes and machine code generation. The virtual machine assembler implements the architecture-specific parts for the backend, such as defining available registers and instructions and producing the actual machine code. Lastly,

the virtual machine submodule contains the execution and debugging engines as well as the bytecode representation. Complementary to the Compiler Framework is the suite of tools containing the Command Line Interface (CLI) (*lz*) to run and execute Lazarus code from the terminal, as well as the LSP/DAP server (*lzs*). The latter tool uses the Server module, which contains a generic implementation of the Base Protocol, the LSP and DAP protocols, enabling code reuse, as well as the Lazarus-specific implementation. Lastly, the Plugin module contains the Visual Studio Code-specific code for launching *lzs* and connecting to it as a client. The LSP uses some custom messages to provide extra functionality, such as the syntax tree visualizer, the control flow graph viewer, and the optimization pipeline inspector. These extra tools provide direct access to internal structures used within the compiler for several tasks, enhancing understanding of the inner workings by streamlining their visualizations, and serving as debug tools for compiler engineers.

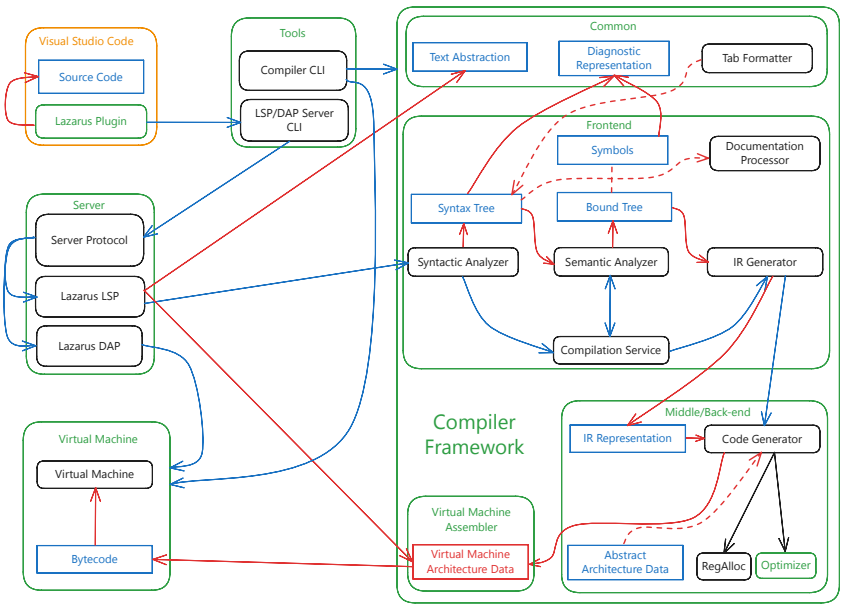

Figure 2 Lazarus Infrastructure Architecture Diagram.

The entire architecture was also built around the principle of providing access to the internals of the system in a more visual and easy to understand way, which is accomplished by exposing this information via Visual Studio Code extension. For brevity, it is only going to be enumerated some of them. The syntax tree is exposed to the user, and it highlights the nodes changed by the incremental parser. The entirety of the pipeline stages (i.e., optimizations, and code generation) are exposed in a textual form that highlights the changes. A function's control flow graph used by the bound tree can be displayed.

6 Incremental Parser

As already mentioned, an incremental parser is the technology that enables efficient re-use of previous Syntax Trees, by only reparsing changed regions of the source code; making a crucial feature for the compiler's integration in an IDE. The system is built on top of two very important properties of the `Concrete` Syntax Tree. The first one being immutability. After creation, it is not possible to change the tree, hence the usage of bound trees, making it possible to completely reuse subtrees in other places, as is the case with the refactoring tools,

the code formatter, and of course the incremental parser. Another advantage of this property is trivial concurrency, although it has not been exploited by the compiler yet. The second property is that the used tree is actually not the one produced by the parser. If every node knew its exact position in the source code (i.e., line and column information), any edit action would force rebuilding the entire parse tree after the change. Instead, the parser produces a persistent tree, which is informally called the "green" tree, that keeps track of the width of the node and not its position. Since knowing the exact position is often required, a new tree (the "red" tree) is created, which is just a thin wrapper over the green one, and does the position calculations [49, 50]. This red tree is not fully created at once; doing so would negate the effects of the incremental parser, but rather when a child of a node is accessed, all the children of that node are created. Since not all parsed trees need to be traversed in their entirety, for instance, a new change has been received while the user is typing, not all red tree's nodes are going to be fully created. While the actual implementation uses an index based position, which simplifies the algorithm, it can be modified to also include line and column spans, enabling the red tree to present the position in a more convenient format; though the current approach just computes this information on-request and is incrementally updated by the text representation (a modified rope data structure with line and column cache). The definitions of these two trees are generated by a Source Generator.

Note that the lexical analyzer, now, does not keep track of position, and can also be called independently of the position, as long as its source is confined to token boundaries. In turn, the incremental parse tree rebuilding algorithm, called the `Blender`, can call the lexer only when it is needed. This also means that the parser cannot use the lexer, doing so would produce brand-new tokens instead of re-using from the previous tree. As such, the Blender and the Lexer implement a common interface, which also enables non-terminal nodes to be exposed to the parser. This interface can be found in Listing 5.

Listing 5 ITokenStream interface.

```
internal interface ITokenStream {
    GreenToken PeekToken();
    GreenToken ConsumeToken();
    bool TryPeekNonTerminal([MaybeNullWhen(false)] out GreenNode? n);
    bool TryTakeNonTerminal([MaybeNullWhen(false)] out GreenNode? n);
}
```

The Blender can take the green tree of a previous compilation (the baseline), and locate the change by using a work stack. For this process, the work stack is initialized with the root node of the baseline tree. While the change has not been dealt with, the top of the stack is examined, and if the range overlaps with the change, the node is decomposed into its children. If the top node range is located before the change it is stored into another stack (the left stack), and discarded from the work one. When the top of the stack is a token that falls within the change range, then it is discarded and the algorithm stops. The left stack contains the nodes that fall behind the change, and the work stack the ones that lay after the change. A final stack is produced with these nodes, so that when popped they remain in the order the parser expects, with a caveat the range in the source text change entry is also included in the appropriate place in this stack. Rather than requesting tokens directly, the parser uses an *ITokenStream* interface, that can be found in Listing 5. This enables the parser to operate identically whether it is reading from the lexer (which reads from the source text directly), or the Blender (reading the tokens of the baseline tree and only from the lexer for the changed region). Furthermore, this interface was later extended for the Blender to expose non-terminal nodes, allowing the parser to check in key points (e.g., in a *ParseStatement*) if the next node matches the expected type, avoiding parsing entire sub-trees.

Since in the Blender operations, a token may indeed provoke other tokens to change (e.g., commenting code), the same sorts of interface is needed for the base source and the Blender that the lexer uses, enables following nodes/tokens to be broken up until its textual representation. The *PeekCharacter* that it would contain, cannot modify the current stack, since it may be used by the lexer to detect a token boundary; the same principle applies to the *PeekToken*. The parser can then simply check while parsing if there is a node of the appropriate kind (e.g., a statement) at the beginning of parsing that particular kind, and reusing if that is the case. Figure 3 contains the execution of the Blender in an incremental parser; the added code fragment is highlighted in red, as well as the newly created nodes in the end.

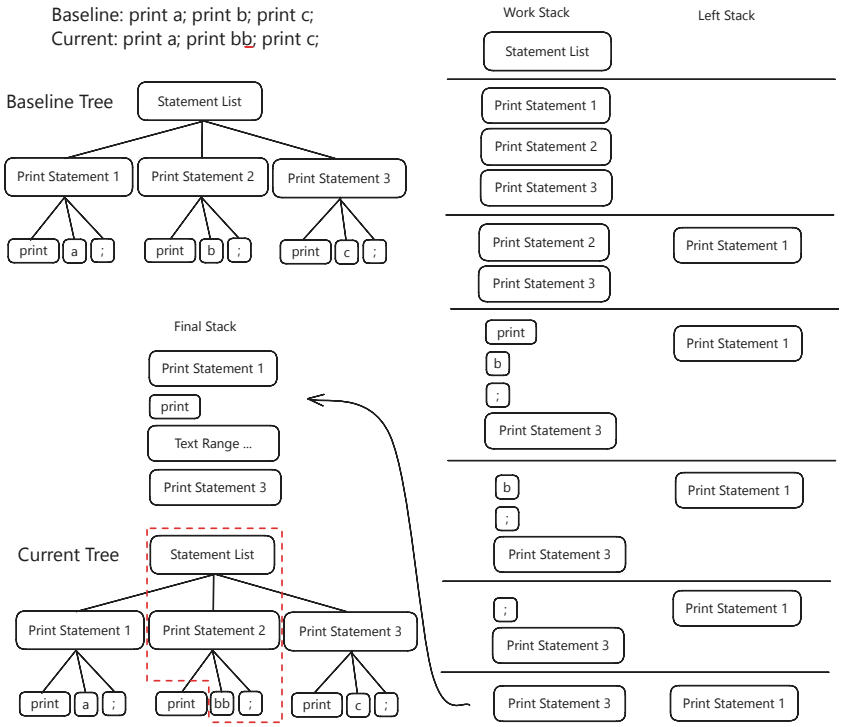

Figure 3 Blender Algorithm.

This algorithm can be trivially extended to support multiple changes. First the list of changes must be in order, and when the first iteration of the specified algorithm finishes, the text change is added to the left stack, and the cycle continues until the end. Furthermore, there is no need to construct a stack at all; one can simply create a tree cursor that knows its position in the tree and is capable of moving to the first child, to the next child (sibling), and to the parent, this way the incremental parse tree rebuilding algorithm can be performed while parsing. This incremental parsing algorithm is based on the algorithm found in the Roslyn compiler, that started as a piece of work of Neal Gafter [20, 21].

7 Conclusion

The initial aim for the project was to build a pedagogical tool that students could use to modify, debug, and self-learn from their tinkering, while empowering them with the background knowledge required to go into production compilers and learn. Given the small

size (<20k lines of code) in comparison to the rich feature set, and the inspiration and similarity to production compilers, we consider that this goal was attained from the design and implementation perspective. However, tests with end users are missing to prove this learning thesis. The incremental parser on its own already pushes the envelopes of what is traditionally thought and discussed in academic environments.

Moreover, the system was built in a modular fashion, following closely the architecture already detailed, which allows for easy usage and swapping of components, enabling other compiler designers to use the platform and extend it without actually modifying it. Furthermore, although the target audience is a master's class on compilers, which is taken after an introductory class of language processing, it also enables other classes besides compiler-centric ones to use this resource as well. Computation systems classes can employ this tool chain to demonstrate the basics (essential concepts) of computer architectures and machine coding in Assembly language. More introductory classes such as language processing can use the intermediate representation, providing students with hands-on experience with LLVM-like infrastructures without the heavy-weight and complexity usually associated with them.

The size of the infrastructure as a whole is showing to be an advantage way beyond the educational sphere. As it currently stands, it is already being used to experiment with new technologies, such as Model Context Protocol built-in to the compiler itself, enabling further research that may one day power the next generation of the programming language ecosystem.

The Lazarus language, while somewhat complex, spanning through all the core concepts of Imperative and Object-Oriented styles, still is a very capable language capable of rather complex "scripts" and graphical user interface capabilities, but fundamentally, it is only intended to provide the means of experimenting with the system and aiding in the research and experimentation taken by students. Though the pedagogical strategy was not discussed in this paper, given it would shift the focus of the paper away from the technical topics and use the necessary space in the process, it is hopefully going to be discussed on a future paper.

Regarding future work, some immediate goals are to continue the development of the webpage and make the entire tool chain generally available. We envision this site, not only as a way to provide the platform as a whole, but also as a means of sharing learning resources, samples and documenting the project with visual and practical explanations to better illustrate the inner workings and the architectural choices made. Furthermore, we want to start conducting diversified testing with end-users to comprehend the learning curve of both using the platform and as a means of a study material. Long-term, the main goal is to evolve the infrastructure to a point where it would be able to self-host itself (i.e., mostly written in the Lazarus language).

References

1 Alfred V. Aho, Mónica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques, [and] tools*. Pearson Addison-Wesley, Boston, 2nd. ed edition, 2007. OCLC: 1341171760.

2 Anthropic. What is Model Context Protocol? Connect AI to your world, 2025. URL: `https://claude.com/blog/what-is-model-context-protocol`.

3 Anthropic. What is the Model Context Protocol (MCP)?, November 2025. URL: `https://modelcontextprotocol.io/docs/getting-started/intro`.

4 Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, Cambridge, 1997. `doi:10.1017/CBO9781139174930`.

5 Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.10 edition, November 2023.

6 Thorsten Ball, Nathan Sobo, Antonio Scandurra, and Max Brunsfeld. Rope & SumTree - Zed Blog, 2024. URL: https://zed.dev/blog/zed-decoded-rope-sumtree.

7 Hans-J. Boehm, Russ Atkinson, and Michael Plass. Ropes: An alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995. doi:10.1002/spe.4380251203.

8 Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. Simple and Efficient Construction of Static Single Assignment Form. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Ranjit Jhala, and Koen De Bosschere, editors, *Compiler Construction*, volume 7791, pages 102–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-642-37051-9_6.

9 Max Brunsfeld. Tree Sitter, 2026. doi:10.5281/ZENODO.19357078.

10 Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 246–257, New York, NY, USA, June 1995. Association for Computing Machinery. doi:10.1145/207110.207154.

11 Cliff Click. From Quads to Graphs: An Intermediate Representation's Journey. Technical report, Rice University, 1997.

12 Cliff Click. Simple, 2025. original-date: 2023-08-19T15:52:57Z. URL: https://github.com/SeaOfNodes/Simple.

13 Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995. doi:10.1145/201059.201061.

14 Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, 1995. doi:10.1145/202530.202534.

15 Clifford Noel Click. *Combining Analysis, Combining Optimizations*. PhD thesis, Rice, 1995. URL: https://hdl.handle.net/1911/96451.

16 Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. doi:10.1145/115372.115320.

17 DWARF. DWARF Debugging Information Format Version 5, 2017. URL: https://dwarfstd.org/doc/DWARF5.pdf.

18 Haoxiang Fei, Yu Zhang, Hongbo Zhang, Yanlin Wang, and Qing Liu. MoonBit: Explore the Design of an AI-Friendly Programming Language. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, LLM4Code '24, pages 79–83, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643795.3648376.

19 Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992. doi:10.1145/151640.151642.

20 Neal Gafter. *Parallel incremental compilation*. PhD thesis, University of Rochester, 1990. URL: https://urresearch.rochester.edu/institutionalPublicationPublicView.action?institutionalItemId=4743&versionNumber=1.

21 Neal Gafter. Toy Incremental Parser, 2025. URL: https://github.com/gafter/Toy-Incremental-Parser.

22 Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3):300–324, 1996. doi:10.1145/229542.229546.

23 Peter Golde, Matthew J. Warren, Neal M. Gafter, and Heejae Chang. Full fidelity parse tree for programming language processing, June 2013. URL: https://patents.google.com/patent/US20130152061A1/en.

24 Google. Dart and Flutter MCP server, 2026. URL: https://docs.flutter.dev/ai/mcp-server.

25 R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29, December 1969. doi:10.2307/1995158.

26 ISO. ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF, 1996. URL: https://www.iso.org/standard/26153.html.

27 Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. Partial redundancy elimination in SSA form. *ACM Trans. Program. Lang. Syst.*, 21(3):627–676, 1999. doi:10.1145/319301.319348.

28 Robert Kennedy, Fred Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, and Mark Streich. Strength reduction via SSAPRE. In Kai Koskimies, Gerhard Goos, Juris Hartmanis, and Jan Van Leeuwen, editors, *Compiler Construction*, volume 1383, pages 144–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998. doi:10.1007/BFb0026428.

29 Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 285–299, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009880.

30 Matthias Koch. Another Look into the Future with Rider's Predictive Debugger | The .NET Tools Blog, 2023. Running Time: 920 Section: .NET Tools. URL: https://blog.jetbrains.com/dotnet/2023/12/04/another-look-into-the-future-with-riders-predictive-debugger/.

31 Matthias Koch. Introducing Predictive Debugging: A Game-Changing Look into the Future | The .NET Tools Blog, 2023. Section: .NET Tools. URL: https://blog.jetbrains.com/dotnet/2023/07/27/introducing-predictive-debugging-a-game-changing-look-into-the-future/.

32 C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE. doi:10.1109/CGO.2004.1281665.

33 Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. doi:10.1145/357062.357071.

34 LLVM Foundation. The LLVM Compiler Infrastructure, 2025. original-date: 2016-12-07T09:39:33Z. URL: https://github.com/llvm/llvm-project.

35 Edward S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969. doi:10.1145/362835.362838.

36 Peng Lyu. Text Buffer Reimplementation, 2018. URL: https://code.visualstudio.com/blogs/2018/03/23/text-buffer-reimplementation.

37 Microsoft. Roslyn, 2021. URL: https://github.com/dotnet/roslyn.

38 Microsoft. Language Server Protocol, 2022. URL: https://microsoft.github.io/language-server-protocol.

39 Microsoft. Debug Adapter Protocol, 2024. URL: https://microsoft.github.io/debug-adapter-protocol.

40 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. doi:10.1016/0022-0000(78)90014-4.

41 E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, 1979. doi:10.1145/359060.359069.

42 Noam Nisan. *The elements of computing systems: building a modern computer from first principles*. MIT Press, Cambridge, Massachusetts, 2005.

43 Terence Parr. ANTLR, 2024. original-date: 2010-02-04T01:36:28Z. URL: https://github.com/antlr/antlr4.

44 Reese T. Prosser. Applications of Boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '59 (Eastern)*, pages 133–138, Boston, Massachusetts, 1959. ACM Press. doi:10.1145/1460299.1460314.

45 Fabrice Rastello and Florent Bouchez Tichadou, editors. *SSA-based Compiler Design*. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-030-80515-9.

46 Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. *SIGPLAN Not.*, 39(6):277–288, 2004. doi:10.1145/996893.996875.

47 Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing phi-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/199448.199464.

48 Thomas VanDrunen and Antony L. Hosking. Value-Based Partial Redundancy Elimination. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Evelyn Duesterwald, editors, *Compiler Construction*, volume 2985, pages 167–184. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science. doi:10.1007/978-3-540-24723-4_12.

49 Matthew J. Warren, Avner Y. Aharoni, Mads Torgersen, Renaud Paquay, Neal M. Gafter, Jared Parsons, David N. Schach, Aleksey V. Tsingauz, Peter Golde, Kevin Pilch-Bisson, and Karen Liu. Efficient immutable syntax representation with incremental change, February 2020. URL: https://patents.google.com/patent/US10564944/en.

50 Matthew J. Warren, Mads Torgersen, Renaud Paquay, Neal M. Gafter, Jared Parsons, David N. Schach, Aleksey V. Tsingauz, Peter Golde, Kevin Andrew Pilch, and Karen Liu. Efficient immutable syntax representation with incremental change, June 2022. URL: https://patents.google.com/patent/US11372630B2/en.

51 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991. doi:10.1145/103135.103136.