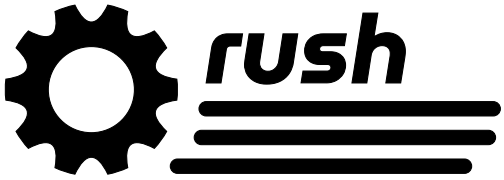
Carl-Fuhlrott-Gymnasium
Wuppertal

School Year 2022/2023
Grade Q2



The Conversion of Source Code to Machine Code

Explaining the Basics of Compiler Construction Using a Self-Made Compiler

Silas Groh, Mik Müller
Supported by Sonja Sokolović

May 13, 2023

Abstract

Programming languages are undoubtedly of great importance for various aspects of modern-day life. Even if they remain unnoticed, digital systems running programs written in some sort of programming language are ubiquitous. However, there are numerous ways of implementing a programming language. A language designer could choose an interpreted or a compiled approach for their language's implementation. Both ways of program execution come with their own advantages and disadvantages.

This paper aims to inform the reader about different means of program execution, focussing on compiler construction. However, we will only focus on the basics since implementing a programming language is often a demanding task. In order to include practical examples, we will explain concepts on the basis of *rush*, our own programming language, which was purposefully designed for this paper. During the implementation of rush, the focus for this paper has shifted slightly. As the title suggests, we originally planned to only implement and explain one compiler. However, there are numerous architectures which a compiler could target and settling on just one felt like the reader would miss out on too much. Therefore, we have implemented rush using two interpreters, one transpiler, and four compilers.

In Chapter 1, we will give an introduction to implementing a programming language. Moreover, the rush programming language and its characteristics are presented. In Chapter 2, the process of analyzing the program's syntax and semantics is explained. Chapter 3 focuses on how interpreters can be used in order to implement a programming language. Here, we will differentiate between a *tree-walking interpreter* and a *virtual machine*. Chapter 4 illustrates how compilation to *high-level* targets[1] works. As examples for high-level targets, we will present a compiler targeting the rush virtual machine, a compiler targeting *WebAssembly*, and a compiler which uses the *LLVM* framework. Chapter 5 focuses on how compilers targeting *low-level* targets[2] can be implemented. For this, we will present a compiler targeting *RISC-V* assembly and another compiler targeting *x86_64* assembly. Lastly, Chapter 6 presents final thoughts and conclusions.

[1] High-level targets: in this case machine independent; many abstractions are provided.

[2] Low-level targets: specific to one CPU architecture and operating system; little abstraction is provided.

Contents

1. Introduction

Nowadays, computer programs are often written in formal, purposefully designed languages. These languages introduce many constraints, such as syntactic and semantic rules, in order to allow programmers to implement algorithms in a structured and precise manner. Advantages of high-level languages are that program development is faster and easier, that programs are easier to maintain, and that programs are portable, meaning that the program can be executed on different architectures [Dan05a, p. 9]. Since programming languages should be easy to write for a human while being easy to understand for a computer, they are often regarded as complicated. The fundamental challenge is that a computer is only able to interpret a sequence of CPU instructions, instead of a written program. Therefore, the source program has to be translated into such a sequence of instructions before it can be executed by the computer. Because programming languages are strictly defined by formal constraints, the translation process can be defined formally as well. Therefore, this translation can be automated and implemented as an algorithm on its own. This process is referred to as *compilation*, and is usually performed by a program called a *compiler*. It is apparent that compilation requires significant effort and must obey complex rules since it should translate the source program precisely, without altering its meaning.

Another common method of program execution is to implement a program, often referred to as an *interpreter*, which evaluates the source code directly. Although compilers and interpreters share some of their core principles, the major difference is that the interpreter omits translation. The implementation of an interpreter is often significantly easier and smaller since the interpreter only has to comprehend the source program in order to execute it. In other words, the step of translating the source program into another form can be avoided completely. However, implementing an interpreter is only sensible if it is written using a high-level language like C or Rust, since the implementation language is able to save the programmer a lot of work. If an interpreter was to be implemented using a low-level language like assembly, there would not be much work done by the implementation language. Compared to interpreters, compilers played an essential role in the early days of computing as high-level languages were yet to be developed.

The first compiler was implemented around 1956 and aimed to translate *Fortran* to computer instructions. However, the success of this programming endeavor was not assured until the program was completed. In total, the project involved roughly 18 man-years of work and is thereby regarded as one of the largest programming projects of the time. To this day, new compilers are created, and innovations in the field of programming languages can be observed regularly. Therefore, compiler construction can still be considered a fundamental and relevant topic in computer science [Wir05, p. 6].

1.1. Stages of Compilation

In order to minimize intricacy and to maximize modularity, compilation often involves several individual steps. Here, the output of a step serves as the input for the next one. However, partitioning the compiler into too many steps is prone to cause inefficiencies during compilation. Separating the process of compilation into individual steps was the predominant technique until about 1980. Due to limited memory of the computers at the time, a single-step compiler could not be implemented. Therefore, only the individual steps of the

compiler would fit, as each step occupied a considerate amount of machine memory. These types of compilers are called *multi-pass compilers*. However, the output of each step would be serialized and written to disk, ready to be read by the next step. It is obvious that this partitioning leads to a lot of performance overhead, since disk access is significantly slower than memory access. Nowadays, one can mitigate these performance issues by implementing the compiler as a single program. Therefore, the compiler can avoid slow disk operations by keeping intermediate structures solely in memory.

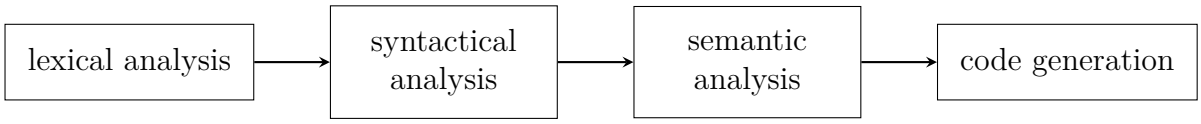


Figure 1.1 – Steps of compilation [Wir05, pp. 6–7].

Usual steps of modern compilers, as shown in Figure 1.1, are as follows [Wir05, pp. 6–7]:

1. The lexical analysis (*lexing*) translates sequences of characters of the source program into their corresponding symbols (*tokens*). Tokens, such as identifiers, operators, and delimiters are recognized by examining each character of the source program in sequential order.

2. The syntactical analysis (*parsing*) transforms the previously generated sequence of tokens into a tree data structure which directly represents the structure of the source program.

3. The semantic analysis (*analyzing*) is responsible for validating that the source program follows the semantic rules of the language. Furthermore, this step often generates a new, similar tree which contains additional type annotations and is affected by early optimizations.

4. Code generation traverses the previously generated tree in order to emit a sequence of target-machine instructions. Due to likely constraints considering the target instruction set, the code generation is often considered to be the most involved step of compilation.

Many modern compilers tend to combine steps 1 and 2 into a single step. Using this approach, the parser accesses a lexer directly, instructing it to return the next token when the parser demands it. Using this approach, memory usage is minimized since the parser only considers a few tokens instead of a complete sequence [Mog17, Chapter 1]. Figure 1.2 shows an altered chart considering this change.

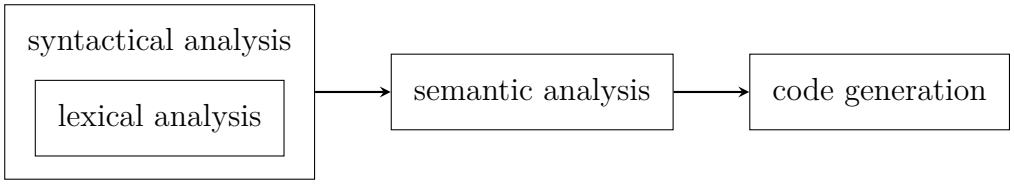


Figure 1.2 – Steps of compilation. (altered)

1.2. The rush Programming Language

For this paper, we have developed and implemented a simple programming language called *rush*[1]. The language features a *static type system*, six different data types[2], *arithmetic operators*, *logical operators*, *local and global variables*, *pointers*, *if-else expressions*, *loops*, and *functions*. In order to introduce the language, we will now consider the code in Listing 1.1.

```
1  fn main() {
2      exit(fib(10));
3  }
4
5  fn fib(n: int) -> int {
6      if n < 2 {
7          n
8      } else {
9          fib(n - 2) + fib(n - 1)
10     }
11 }
```

Listing 1.1 – Generating Fibonacci numbers using rush.

This rush program can be used to generate numbers included in the Fibonacci sequence. In the code, a function named '`fib`' is defined using the '`fn`' keyword. This function accepts the parameter '`n`', which denotes the position in the Fibonacci sequence of the number to be calculated. Since '`int`' is specified as the type of the parameter, the function may be called using any integer value as its argument. However, the constraint $n \in \mathbb{N}_0$ must be valid in order for this function to return the correct result[3]. In this example, the '`main`' function calls the '`fib`' function using the natural number '10'. In rush, every valid program must contain exactly one '`main`' function since that is where program execution will start. Even though rush provides a '`return`' statement, the body of the '`fib`' function does not contain one. This is because blocks, like the one of the '`fib`' function, return the result of their last expression. The if-else construct is also an expression since it represents the last entity in the block, and is not followed by a semicolon. If '`n`' is less than '2', it is returned without modification. Otherwise, the function calls itself recursively in order to calculate the sum of the preceding Fibonacci numbers '$n-2$' and '$n-1$'. The result value of the entire if-expression is calculated by using one of the two branches. Since the if-else construct is also an expression, there is no need for a redundant '`return`' statement. In line 2, the '`exit`' function is called. Even though this function is not defined anywhere, the code still executes without any errors. This is due to the fact that the '`exit`' function is a *built-in* function that is used to exit a program with the specified exit code.

1.2.1. Features

As outlined previously, rush implements various features, which are also found in many other relevant programming languages today. Generally, rush is a *procedural* programming language, meaning that code is partitioned into *procedures*, which are commonly referred to as "functions". This paradigm has been selected as it is also common across most of today's popular programming languages [TN07, p. 278]. The language provides many features which are meant to increase developer productivity. For instance, a program might use a '`loop`' to

[1] Capitalization of the name is omitted intentionally.
[2] Including the '`int`', '`float`', '`bool`', '`char`', *unit*, and *never* type.
[3] Assuming the function should comply with the original Fibonacci definition.

implement code repetition easily. For reference, Table 1.2 shows some features of the rush programming language.

Just like most other compiled languages, rush also includes a *static type system*, meaning that every variable has a type, which cannot change during runtime. For instance, after a new variable, which can hold integers, was defined, only integer values may be assigned to it. This way, the compiler can validate the semantic correctness of the program, showing error messages if an erroneous program is provided. Apart from mandating a program's correctness, a static type system also makes the program easier to read for humans. Therefore, the rush programming language is designed in a way that promotes correctness and programmer efficiency. Table 1.3 provides an overview of all the data types rush provides.

Table 1.1 – Lines of code of the project's components in commit '`f8b9b9a`'.

Component	Lines
lexer and parser	2737
tree-walking interpreter	578
VM compiler and runtime	1288
Wasm compiler	1584
LLVM compiler	1450
C transpiler	1185
RISC-V compiler	2234
x64 compiler	2751

While the first four types are self-explanatory, the last two demand special explanation. The *unit* type is used in order to denote that a function returns no value. This concept is comparable to *void* in other languages like C or Java. In rush however, the unit type is treated like a value at runtime. Even though the unit type can be saved inside a variable or used as a function parameter, all rush compilers simply ignore it since it holds no value. In Listing 1.1, the '`main`' function also returns the unit type. A function named '`foo`' can be defined like '`fn foo() -> () {}`', as well as '`fn foo() {}`', because the unit type is the default. The *never* type is used in order to denote that an expression, or statement, interrupts the normal control flow, so that the expression never yields a value. Because the built-in '`exit`' function terminates a program, its result will never be of relevance at runtime. However, the fact that the never type is returned is relevant as this information is valuable for the semantic analysis and the compilers. Furthermore, '`break`', '`continue`', and '`return`' also result in the never type. It is to be noted that the type cannot be used explicitly. That means the usage of syntax like '`let a: ! = exit(0);`' is illegal.

In the rush project, most of the previously presented stages of compilation are implemented as their own individual crates[4]. This way, each component of the programming language can be developed, tested, and maintained separately. In the Git commit '`f8b9b9a`', the entire rush project consisted of 17460 lines of source code[5]. On the first sight, this might seem like a large number for a simple programming language. However, the rush project includes a lexer, a parser, a semantic analyzer, two interpreters, four compilers, and several other tools like a language server for editor support. Table 1.1 shows the lines of code of the project's individual components. Here, it becomes apparent that the tree-walking interpreter contains the least amount of code, as it is the simplest. The compilers targeting high level targets both require around 1500 lines. As for the low-level compilers, they both require over 2000 lines of code, with x64 requiring around 500 lines more than RISC-V. By considering this table, the complexity of the individual components becomes apparent.

[4] A crate is a software library in Rust terms.

[5] Blank lines and comments are not considered for all specifications of line count.

Table 1.2 – Most important features of the rush programming language.

Name	Description	Example
unconditional loop	Executes code repeatedly.	`loop { }`
while-loop	Like '`loop`', but checks a condition before each iteration.	`while x < 5 { }`
for-loop	Like '`while`', but executes an update-expression after each iteration.	`for i = 0; i < 5; i += 1 { }`
if-expression	Executes different code based on a condition.	`if true { } else { }`
function definition	Defines a function, which can be called later.	`fn foo(n: int) { }`
infix-expression	Performs mathematical and logical operations using two values.	`1 + n; 5 ** 2`
prefix-expression	Performs mathematical and logical operations using one value.	`!false; -n`
let-statement	Defines a variable with an initial value.	`let mut answer = 42;`
cast-expression	Converts a value into a different type.	`42 as float`

Table 1.3 – Data types in the rush programming language.

Notation	Example	Description
'`int`'	`let a: int = 0;`	Values of the *int* type can represent 64-bit signed integers, that is, $[-2^{63};\ 2^{63}-1]$.
'`float`'	`let b: float = 3.14;`	Values of the *float* type can represent 64-bit double precision floating-point numbers, as defined by the *IEEE 754–2008* standard.
'`bool`'	`let c: bool = true;`	Values of the *bool* type can represent either 'true' or 'false'.
'`char`'	`let d: char = 'a';`	Values of the *char* type can be any integer in the range $[0;\ 127]$, representing any valid ASCII character.
'`()`'	`let e: () = main();`	The *unit* type is an empty tuple representing nothing, comparable to *void* in other languages. It is the implicit return type of functions in rush and is commonly used in functional languages like Haskell [Ser19, p. 208].
'`!`'	`let f = exit(42);`	The *never* type can never be materialized, it exists to express that an expression diverges and therefore yields no value.

2. Analyzing the Source

2.1. Lexical and Syntactical Analysis

As previously mentioned, the first step during compilation or program execution consists of the *lexical* and *syntactical analysis*. Program source text is, without previous processing, just *text*, i.e., a sequence of characters. Before the computer can even begin to analyze the semantics and meaning of a program, it first has to *parse* the program source text into an appropriate data structure. This is done in two steps that are closely related and often combined. The *lexical analysis*, performed by a *lexer*, and the *syntactical analysis*, performed by a *parser*.

2.1.1. Formal Syntactical Definition by a Grammar

Just like every natural language, most programming languages also conform to a grammar. However, grammars for programming languages are most often type 2 or 3 in the Chomsky hierarchy, that is, *context-free* and *regular* languages [Wat17, pp. 23f]. Additionally, it is not uncommon for parser writers to formally define the grammar using some notation. Popular options include *BNF*[1] and *EBNF*[2], the latter of which is used here. Although this paper does not fully explain these notations, Listing 2.1 shows a short example grammar notated using EBNF. For reference, Appendix A contains the full grammar of rush.

```
1 Expression = Term , { ( '+' | '-' ) , Term } ;
2 Term       = Factor , { ( '*' | '/' ) , Factor } ;
3 Factor     = ( integer
4               | '(' , Expression , ')' ) , [ '**' , Factor ] ;
5 integer    = { '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' }- ;
```

Listing 2.1 – Grammar for basic arithmetic in EBNF notation.

One thing worth explaining is the '`-`' at the end of line 5. It is used to exclude a set of symbols from the preceding rule. As in this case, there is no symbol following the dash, the empty word, called ε, is excluded from the preceding repetition. Thus, the repetition cannot be repeated zero times here, as that would produce the empty word, which is excluded. So the notation '`{ … }-`' is used to represent repetitions of one or more times, whereas the same without the trailing dash represents repetitions of zero or more times.

2.1.2. Grouping Characters Into Tokens

Before the syntax of a program is validated, it is common to have a lexer group certain sequences of characters into *tokens*. The set of tokens a language provides is the union of the set of all terminal symbols used in context-free grammar rules and the set of regular

[1] "Backus-Naur Form", named after the two main inventors [Bac+60].

[2] "Extended Backus-Naur Form", an extended version of *BNF* with added support for repetitions and options without relying on recursion, first proposed by Niklaus Wirth in 1977 [Wir77] followed by many slight alterations. The version used in this paper is defined by the ISO/IEC 14977 standard.

grammar rules. For the language defined in Listing 2.1 these are the five operators '`+`', '`-`', '`*`', '`/`', '`**`', and the '`integer`' non-terminal.

Although, the specifics of implementing a lexer are not explored in this paper, a basic overview is still provided. The base principle of a lexer is to iterate over the characters of the input to produce tokens. Depending on the target language, it might be required to scan the input using an n-sized window, i.e., observing n characters at a time. In the case of rush, this n is '2', resulting in the '`Lexer`' struct not only storing the current character, but also the next one, as seen in Listing 2.2. For clarity, Table 2.1 shows the values of '`curr_char`' and '`next_char`' while processing of the input '`1+2**3`'. Here, every row in the table represents one point in time displaying the lexer's current state.

crates/rush-parser/src/lexer.rs

```rust
pub struct Lexer<'src> {
    input: &'src str,
    reader: Chars<'src>,
    location: Location<'src>,
    curr_char: Option<char>,
    next_char: Option<char>,
}
```

Listing 2.2 – The rush '`Lexer`' struct definition.

Table 2.1 – Advancing window of a lexer.

Calls	State	`curr_char`	`next_char`	Output Token
0	**1** + 2 * * 3	None	None	
0	**1** + 2 * * 3	None	Some('1')	
0	**1** **+** 2 * * 3	Some('1')	Some('+')	
1	1 **+** **2** * * 3	Some('+')	Some('2')	Int(1)
2	1 + **2** ***** * 3	Some('2')	Some('*')	Plus
3	1 + 2 ***** ***** 3	Some('*')	Some('*')	Int(2)
4	1 + 2 * ***** **3**	Some('*')	Some('3')	
4	1 + 2 * * **3**	Some('3')	None	Pow
5	1 + 2 * * 3	None	None	Int(3)

As explained in Section 1.1, many modern language implementations have the lexer produce tokens on demand. Thus, a lexer requires one public method, '`next_token`' for instance, reading and returning the next token. In Table 2.1, the column on the left displays how many times the '`next_token`' method has been called by the parser. In the first three rows, this count is still '0', as this happens during initialization of the lexer in order to fill the '`curr_char`' and '`next_char`' fields with sensible values before the first token is requested. The '`Pow`' token, composed of two '`*`' characters, requires the lexer to advance two times before it can be returned, which is represented by the two rows in which the call count is '4'. A simplified '`Token`' struct definition for the example language from Listing 2.1 is shown as part of Listing 2.3.

In addition to the current and next character, a lexer also has to keep track of the current position in the source text for it to generate helpful messages with locations in case the program is syntactically malformed. This is done in the '`location`' field, which is incremented every time the lexer advances to the next character. While producing a token, the lexer can read this field, once at the start, and once after having read the token, in order to save the two values as the token's span.

A special case worth mentioning are comments. As explained later in Section 2.1.3, depending on the parser, comments may simply be ignored and skipped during lexical analysis, or could get their own token kind and be treated similarly to string literals.

2.1.3. Constructing a Tree

The parser uses the generated tokens in order to construct a tree which represents the program's syntactic structure. Depending on how the parser should be used, this can either be a *Concrete Syntax Tree* (CST) or an *Abstract Syntax Tree* (AST). The former still contains information about all input tokens with their respective locations, while the latter only stores the abstract program structure, focusing on just the relevant information for basic analysis and execution. Therefore, a CST is usually used for development tools like formatters and intricate linters[3] where it is important to preserve stylistic choices made by the programmer, for which knowing the exact location of every token is beneficial. However, an AST is enough for interpretation and compilation as it preserves the semantic meaning of the program. Figure 2.1 shows an AST for the program '`1+2**3`' in the language notated in Listing 2.1 on page 6. In the case of rush, an AST with limited location information is used because rush's semantic analyzer is basic enough to not require precise locations of every token, and, as discussed, execution and compilation requires no CST.

```
struct Token {
    kind: TokenKind,
    span: Span,
}

enum TokenKind {
    Int(u64),

    Plus,  // +
    Minus, // -
    Star,  // *
    Slash, // /
    Pow,   // **
}

struct Span {
    start: Location,
    end: Location,
}
```

Listing 2.3 – Simplified '`Token`' struct definition.

However, not every parser is the same, and there are different strategies for implementing one, depending on the requirements of the language. These strategies are categorized into *top-down* and *bottom-up* parsers. The main difference between them is the order in which they construct the tree. Top-down parsers construct the syntax tree from top to bottom, starting with the root node. Bottom-up parsers instead construct the tree from the leafs at the bottom upwards to the root.

Top-down and bottom-up parsers are further categorized into many more subcategories. The two discussed here are so-called $LL(k)$ parsers and $LR(k)$ parsers. These are named after the direction in which the tokens are read, L being from left to right, and the kind of derivation they perform. L is the leftmost derivation and R the rightmost derivation [Wat17, pp. 75f]. The parenthesized k represents a natural number with $k \in \mathbb{N}_0$, describing the number of tokens for *lookahead*. Often, k is either '1' or '0', forming a window of width two or one respectively. This window moves just like previously explained for the lexer, and observes k tokens, not characters, simultaneously. Since in most cases k is '1', it is common to omit specifying it and to just speak of LL and LR parsers [Wat17, pp. 86–88].

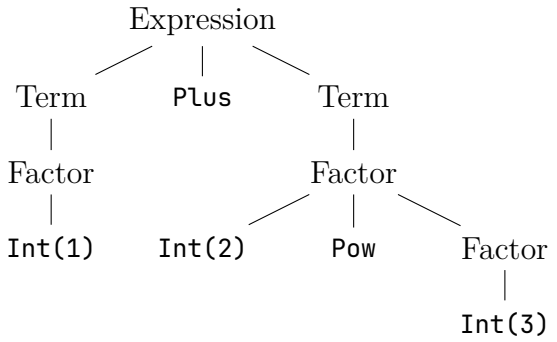


Figure 2.1 – Abstract syntax tree for '`1+2**3`'.

Alternatively to using lookahead tokens, it is possible to create parsers utilizing backtracking. With that approach, a parser has to guess which syntax construct follows if it cannot decide based on the first token. Later, when the parser detects an unexpected symbol, instead of throwing a syntax error, it cancels and returns to the point of decision-making to try the next possible construct. This usually comes with overhead and unclear error messages, which is why this method is rarely used and the superior lookahead method is preferable.

[3] A linter is a tool which suggests how code quality might be improved.

An example for LR parsing is the *shift-reduce* parsing approach, which is outlined later. However, it is to be noted that LR parsers are generally very complicated to implement manually. On the contrary, LL parsers are usually much simpler to implement, but come with a limitation. By design, they must recognize a node by its first $n = k + 1$ tokens, where n is the window size. However, due to that restriction, not every context-free language can be parsed by an LL parser. An example for that is given in Listing 2.4.

```
Expression = Expression , ( '+' | '-' | '*' | '/' | '**' ) , Expression
           | '(' , Expression , ')'
           | integer ;
integer    = { '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' }- ;
```

Listing 2.4 – Example language a traditional LL(1) parser cannot parse.

Here, the problem lies in the left recursion introduced by the '`Expression`' rule. For instance, the input '`1+2*3`' would already cause difficulties. LL parsers must be able to determine which variant of a rule follows based on the first token they encounter. However, when encountering the '`Int(1)`' token, both the '`integer`' and '`Expression`' variants would present valid substitutions, therefore introducing ambiguity. The left recursion causes the '`Expression`' rule to include the same start tokens as the '`integer`' rule since an '`Expression`' can also start with an '`integer`'. In other words, the parser cannot determine whether the '`1`' is an entire '`Expression`' on its own, or whether it is just the start of another '`Expression`' [Wat17, p. 85].

Most LL parsers, including rush's, are *recursive-descent* parsers. Implementation of such a parser is rather uncomplicated. Assuming the grammar respects the mentioned limitation and that an object-oriented approach is used, every context-free grammar rule is mapped to one method on a '`Parser`' struct. In the example grammar from Listing 2.1 on page 6, these are all the capitalized rules highlighted in yellow. Additionally, a matching node type is defined for each context-free rule, holding the relevant information for execution. For Rust, mapping from EBNF grammar notation to type definitions is very simple. Some examples are displayed in Table 2.2.

Table 2.2 – Mapping from EBNF grammar to Rust type definitions.

EBNF	Rust
`A = B , C ;`	`struct A { b: B, c: C }`
`A = B , [ C ] ;`	`struct A { b: B, c: Option<C> }`
`A = B , { C } ;`	`struct A { b: B, c: Vec<C> }`
`A = B , { C }- ;`	`struct A { b: B, c: Vec<C> }`
`A = B | C ;`	`enum A { B(B), C(C) }`
`A = B , ( '+' | '-' ) , C ;`	`struct A { b: B, op: Op, c: C }` `enum Op { Plus, Minus }`
`A = B , [ ( X | Y ) , C ] ;`	`struct A { b: B, c: Option<(XOrY, C)> }` `enum XOrY { X(X), Y(Y) }`

Operator Precedence

As discussed previously, a traditional LL parser cannot parse the language described by the grammar in Listing 2.4. However, when comparing the grammar to the one shown in

Listing 2.1, it might become obvious that the two grammars notate the same language. The first one simply provides additional information about the order of nesting different kinds of expressions, called *precedence*. For example, when parsing the expression '`1+2*3`', the '`2*3`' part should be nested deeper in the tree for it to be evaluated first. In Listing 2.1, this is achieved by recognizing multiplicative expressions as '`Term`'s and having additive expressions be composed of multiple '`Term`'s. Listing 2.4 does not indicate this order itself, so it must be provided externally.

Additionally, a precedence may be either left- or right-associative. The input '`1*2*3`' should be evaluated from left to right, so first '`1*2`', and then the result times three. Now considering '`1**2**3`'[4], the '`2**3`' should be evaluated first, and afterward '1' should be raised to the power of the result. That means, while most operators are evaluated from left to right, that is, they are left-associative, some operators, like the power operator, are evaluated from right to left and are therefore right-associative. In Listing 2.1, left-associativity is achieved by allowing simple repetition of the operator for an indefinite amount of times. Right-associativity instead uses recursion on the right-hand side of the operator.

For LR parsers, the precedence and associativity for each operator is encoded within the parser table, which is explained later. However, there is also a technique called *Pratt parsing*[5] that allows slightly modified recursive-descent LL parsers to parse such languages, given a map of tokens with their respective precedence and associativity. Often, the grammars without included precedence are preferred, because they usually result in a simpler structure of the resulting syntax tree. This can be seen when comparing Figure 2.1 from earlier to Figure 2.2 which shows the resulting AST for the same input using the alternative language representation. Most notably, the rather long sequences of nodes with just a single child, like the path on the left simply resolving to a single '`Int(1)`' token, are gone in Figure 2.2.

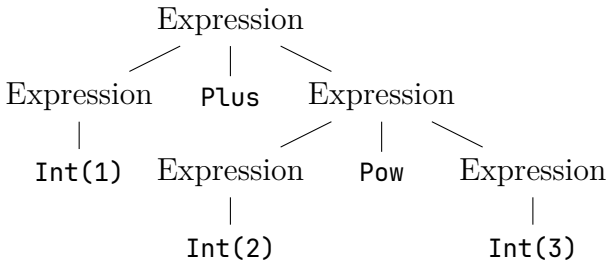


Figure 2.2 – Abstract syntax tree for '`1+2**3`' using Pratt parsing.

Pratt Parsing

As the rush parser makes use of Pratt parsing, the following code snippets are taken from there. First, a mapping from a token kind to its precedence must be defined. The one for rush is found in Listing 2.5. It shows the '`prec`' method implemented on the '`TokenKind`' enumeration. The return type is a tuple of two integers, one for left and one for right precedence. For all but one token kind, the left precedence is lower than the right one, resulting in left-associativity. The higher a precedence is, the deeper in the tree the resulting expression will be, and the earlier it will be evaluated. All unrelated tokens are simply assigned a precedence of '0' for left and right.

The '`expression`' method on the '`Parser`' struct is then modified to take a parameter for the current precedence as seen in Listing 2.6. It then first considers the current token kind to decide which expression to parse, and stores the result in the '`lhs`'[6] variable. Afterward it checks whether the left precedence of the now current token is larger than the '`prec`' argument. When called from elsewhere, like in the condition of a while-loop or in a grouped expression, the '`prec`' argument has its minimum value of '0', as shown in Listing 2.7. In

[4] '`**`' is the power operator here, so the input would be written as 1^{2^3} using mathematical notation.
[5] First introduced by Vaughan Pratt in 1973 [Pra73].
[6] Short for "left-hand side".

crates/rush-parser/src/token.rs

```rust
pub(crate) fn prec(&self) -> (u8, u8) {
    match self {
        // ...
        TokenKind::Plus | TokenKind::Minus => (19, 20),
        TokenKind::Star | TokenKind::Slash | TokenKind::Percent => (21, 22),
        TokenKind::As => (23, 24),
        TokenKind::Pow => (26, 25), // inverse order for right associativity
        TokenKind::LParen => (28, 29), // for calls
        _ => (0, 0),
    }
}
```

Listing 2.5 – Pratt-parser: Implementation for token precedences.

that case, this check will only fail when the whole expression is over, as every operator token is assigned a precedence higher than '0'. If it does not fail, the '`infix_expr`' method is called with the matching operator and '`lhs`'. Afterward, '`lhs`' is overwritten with the returned value.

crates/rush-parser/src/parser.rs

```rust
fn expression(&mut self, prec: u8) -> Result<'src, Expression<'src>> {
    let start_loc = self.curr_tok.span.start;

    let mut lhs = match self.curr_tok.kind {
        TokenKind::Int(num) => Expression::Int(self.atom(num)?),
        // ...
        TokenKind::LParen => Expression::Grouped(self.grouped_expr()?),
        invalid => {
            return Err(Error::new_boxed(/* ... */));
        }
    };

    while self.curr_tok.kind.prec().0 > prec {
        lhs = match self.curr_tok.kind {
            TokenKind::Plus => self.infix_expr(start_loc, lhs, InfixOp::Plus)?,
            TokenKind::Star => self.infix_expr(start_loc, lhs, InfixOp::Mul)?,
            TokenKind::Slash => self.infix_expr(start_loc, lhs, InfixOp::Div)?,
            TokenKind::Pow => self.infix_expr(start_loc, lhs, InfixOp::Pow)?,
            // ...
            _ => return Ok(lhs),
        };
    }

    Ok(lhs)
}
```

Listing 2.6 – Pratt-parser: Implementation for expressions.

The '`infix_expr`' method in Listing 2.8 simply stores the precedence for the right side of the operator token, advances to the next token, and calls the '`expression`' method again for its right-hand side, but this time with the stored '`right_prec`' as the minimum precedence. These simple calls and checks of precedences automatically result in correct grouping and nesting of the expressions.

In order to understand the concept better, Figure 2.3 can be considered. It shows the tokens with their respective left and right precedences for the input '`(1+2*3)/4**5`', which represents the mathematical expression $\frac{1+2 \cdot 3}{4^5}$. Precedences which are irrelevant for this example have been colored in gray. Starting from the left, the parser first encounters an '`LParen`' token and therefore decides to parse a grouped expression. Inside the '`grouped_expr`'

crates/rush-parser/src/parser.rs

```
733 fn grouped_expr(&mut self) -> Result<'src, Spanned<'src, Box<Expression<'src>>>> {
734     let start_loc = self.curr_tok.span.start;
735     // skip the opening parenthesis
736     self.next()?;
737
738     let expr = self.expression(0)?;
739     self.expect_recoverable(
740         TokenKind::RParen,
741         "missing closing parenthesis",
742         self.curr_tok.span,
743     )?;
        // ...
749 }
```

Listing 2.7 – Pratt-parser: Implementation for grouped expressions.

crates/rush-parser/src/parser.rs

```
751 fn infix_expr(/* ... */) -> Result<'src, Expression<'src>> {
757     let right_prec = self.curr_tok.kind.prec().1;
758     self.next()?;
759     let rhs = self.expression(right_prec)?;
        // ...
770 }
```

Listing 2.8 – Pratt-parser: Implementation for infix-expressions.

method, '`expression`' is called again, with '`prec`' being '0' once more. The difference is that now the current token is '`Int(1)`'. That token is then parsed as a simple integer expression and stored in '`lhs`'. Now, the left precedence of the '`Plus`' token, '19', is higher than the value of '`prec`', '0', so the while-loop is entered. In there, the '`infix_expr`' method is called, which queries the right precedence of the '`Plus`' token and calls '`expression`' again, this time with a precedence of '20'. Skipping to the precedence check, the current token's left precedence is higher than '`prec`' again, that is, '21' for the '`Star`' token is higher than '20' for '`prec`'. Therefore, '`infix_expression`' is called once again, which calls '`expression`' again, but now during the next precedence check, the left precedence of the following '`RParen`' token, '0', is not higher than the current precedence of '22'. That means, the innermost '`expression`' call returns with a single '`3`'. This '`3`' is then used as the right-hand side for the multiplicative infix-expression. The same '`RParen`' precedence check fails again for the '`expression`' call with a precedence of '20'. Thus, the '`2*3`' part together forms the right-hand side of the addition expression. Once more, the left precedence of '`RParen`', '0', is not larger than the initial precedence of '0', hence the parser returns to the '`grouped_expr`' call with the entire contents of the parentheses. Here, the right parenthesis is skipped, and the method returns to the outermost '`expression`' call, assigning the entire grouped expression to '`lhs`'.

In order to understand the right-associativity of the '`Pow`' token, the reader is advised to continue going through this procedure for the rest of the example input. A curious reader might additionally want to parse the inputs '`1*2*3`', '`1**2**3`', and '`1+(2+3)`' by hand. It

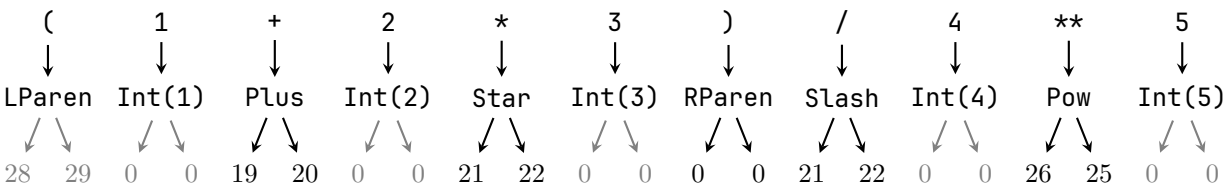


Figure 2.3 – Token precedences for the input '`(1+2*3)/4**5`'.

should then become clear how Pratt parsing achieves parsing with precedences without them being encoded in the grammar, and with that, the node types.

Parser Generators

For most purposes, it is generally not necessary to implement parsers, and with that, lexers, manually. Instead, there are so-called *parser generators* that generate parsers based on some specification of the desired syntax and the required precedences. Often, parser generators define a domain specific grammar notation for the syntax specification, although some parser generators also accept pre-existing grammar notations as their input. Most parser generators target some variation of *table-driven* LR parsers. Table-driven parsers use a table, mapping all possible combinations of tokens on the parse stack and lookahead tokens to an action. For shift-reduce parsers, the possible actions are shifting, that is, reading the next input token and pushing it to the stack, and reducing, which pops some number of elements off the stack, and in place pushes a node to it. The input syntax must therefore either be unambiguous or augmented by precedence rules, so that a complete parser table can be generated [Jef21, pp. 76–81].

2.2. Semantic Analysis

Before compilation can begin, both the syntax and the semantics of the program have to be validated. The *semantic analysis* is responsible for validating that the structure and logic of the program complies with the rules of the programming language. Often, semantic analysis directly follows the syntax analysis since the parser generates the input for the semantic analysis step.

2.2.1. Defining the Semantics of a Programming Language

Often, a programming language is not just defined by its grammar since it cannot specify how the programming language should behave. This behavior can be defined through a so-called *semantic specification*. Apart from describing behavior, the specification regularly states which semantic rules discriminate a valid program from an invalid one. Common rules include *type checking*, *context of statements*, or *integer overflow behavior*. Another example of a semantic rule is that a variable has to be declared before it is used. Defining the semantic rules of a programming language is often a demanding task since not all requirements are clear from the beginning. Furthermore, they usually cannot be defined formally without significant effort [Hol12, pp. 5f]. A language designer typically chooses to write their specification in a natural language. However, due to the specification being written in such a language, it can sometimes be ambiguous. Therefore, a well-written semantic specification should avoid ambiguity as much as possible. Since those rules define when a program is valid, they have to be checked and enforced before program compilation can start [Wat17, p. 21].

2.2.2. The Semantic Analyzer

Because rush shares its semantic rules across all backends, it would be cumbersome to implement semantic validation in each backend individually. Therefore, it is reasonable to implement a separate compilation step which is responsible for validating the source program's semantics. Among other checks, the so-called *semantic analyzer*[7] validates types and variable references while adding type annotations to the AST. The last aspect is of particular importance since all compiler backends rely on type information. For obtaining

[7] Later referred to as "analyzer".

type information, the AST of the source program has to be traversed so that the analyzer can examine and validate the program thoroughly. In order to preserve a clear boundary between the individual compilation steps, the parser only validates the program's syntax without performing further validation.

```
1 fn main() {
2     let two = 2;
3     let three = 3;
4     exit(two + three);
5 }
```

Listing 2.9 – A rush program which adds two integers.

In order to understand why type information is required at compile time, Listing 2.9 should be considered. It displays a basic rush program calculating the sum of two integers, using the result as its exit code. In this example, the exit code of the program is '5', as '2' and '3' are added together. The analyzer will first check whether the program contains a '`main`' function. If this is not the case, the analyzer rejects the program because it violates rush's semantic specification. Furthermore, the analyzer assures that the '`main`' function takes no parameters and returns no value. In this example, there is a valid '`main`' function which complies with the previously explained constraints. Now, the analyzer traverses the function body of the '`main`' function. First, the analyzer examines the statements in lines 2 and 3. Since '`let`' statements are used to define variables, the analyzer will add the variables '`two`' and '`three`' to its current scope. However, unlike an interpreter, the analyzer does not insert the variable's value into its scope. Instead of concrete values, the analyzer only considers types. Therefore, in this example, the analyzer remembers that the variables '`two`' and '`three`' store integer values. In line 4, the analyzer checks that the identifiers '`two`' and '`three`' refer to valid variables. Just like most other programming languages, rush does not allow the addition of two boolean values for example. Therefore, the analyzer checks that the operands of the '`+`' operator have the same type, and that this type is valid in additions. Because this validation requires information about types, the analyzer accesses its scope when looking up the identifiers '`two`' and '`three`'. Since those names were previously associated with the '`int`' type, the analyzer is now aware of the operand types and can check their validity. Calculating the sum of two integers is allowed and results in another integer value. Since rush's semantic specification states that the '`exit`' function requires exactly one integer parameter, the analyzer has to check that it is called correctly. Apart from this call to the '`exit`' function, the analyzer validates all function calls and definitions, not just the ones of built-in functions.

As indicated previously, most compilers require type information while generating target code. For simplicity, we will consider a fictional compiler which can compile both integer and float additions. However, the fictional target machine requires different instructions for addition depending on the type of the operands. For instance, integer addition uses the '`intadd`' instruction while float addition uses the '`floatadd`' instruction. Here, type ambiguity would cause difficulties. If there was no semantic analysis step, the compilation step would have to implement its own way of determining the types of the operands. However, determining these types requires a complete tree traversal of the operand expressions. Due to the recursive design of the AST, implementing this tree traversal would require a large amount of source code in the compiler. However, the implementation of this algorithm would be nearly identical across all of rush's compiler backends. In order to prevent this duplication, rush's semantic analyzer also annotates the AST with type information which is utilized by later steps of compilation.

Implementation

In order to obtain a deeper understanding of how the analyzer works, parts of its implementation and how they behave when analyzing the previous example from Listing 2.9 should be considered. Firstly, the most important struct fields should be discussed.

crates/rush-analyzer/src/analyzer.rs

```
12 pub struct Analyzer<'src> {
13     functions: HashMap<&'src str, Function<'src>>,
14     diagnostics: Vec<Diagnostic<'src>>,
15     scopes: Vec<HashMap<&'src str, Variable<'src>>>,
16     curr_func_name: &'src str,
17     /// Specifies the depth of loops, `break` / `continue` legal if > 0.
18     loop_count: usize,
       // ...
26 }
```

Listing 2.10 – Fields of the '`Analyzer`' struct.

Listing 2.10 displays the struct fields of the analyzer. The field '`functions`' in line 13 associates a function name with the function's signature. Therefore, if a function is called at a later point in time, the analyzer checks whether the function exists so that it can validate the arguments. The next field, '`diagnostics`', contains a list of diagnostics. A '`Diagnostic`' is a struct which represents a message; it is intended to be displayed to a programmer using the language. Each diagnostic has a severity, such as *warning* and *error*. After the analyzer has finished the tree traversal, all diagnostics are displayed in a user-friendly manner. The '`scopes`' field in line 15 is responsible for managing variables. In rush, blocks created with braces ('`{}`') also introduce new scopes. If the analyzer enters such a block, a new scope is pushed onto the '`scopes`' stack. Each scope maps a variable identifier to some variable-specific data. For instance, the analyzer keeps track of variable types, whether variables have been used or mutated later, and the location of their definition. By saving this much information about each declared variable, the analyzer can produce very helpful and accurate error messages or warnings. For reference, compiler output which incorporates diagnostics is displayed in Listing 2.11, it occurs when a new value is assigned to an immutable variable.

```
SemanticError at test.rush:3:5

  2 |     let number = 5;
  3 |     number += 5;
        ^^^^^^^^^^^
  4 | }

cannot re-assign to immutable variable `number`

Hint at test.rush:2:9

  1 | fn main() {
  2 |     let number = 5;
            ~~~~~~
  3 |     number += 5;

variable not declared as `mut`
```

Listing 2.11 – Output when compiling an invalid rush program.

Moreover, the field '`curr_func_name`' saves the name of the current function. This field is used in order to highlight unused functions. If a function is called from a different function, it is marked as used. However, if the call originates from the same function, it is still considered unused. Furthermore, the '`loop_count`' field is used to validate the uses of the '`break`' and '`continue`' statements. Because these statements are only valid inside loop bodies, the value of '`loop_count`' must be greater than '0' at the point when the analyzer encounters such a statement. This counter is incremented as soon as the analyzer begins traversal of a loop body, and is decremented after the body has been traversed. This field is implemented as a counter, rather than a boolean, in order to support nested loops.

Now that important attributes have been highlighted, the example in Listing 2.9 can be considered. First, the analyzer traverses and examines all functions and their bodies. For every rush function, the analyzer invokes an internal method responsible for validating them. Among other tasks, this method inserts a new entry into the '`functions`' map. Because a

'`main`' function is mandatory in every rush program, the analyzer simply checks that the '`functions`' map contains an entry for it after all functions have been traversed. Listing 2.12 shows how validating the '`main`' function's signature works.

crates/rush-analyzer/src/analyzer.rs

```
396 fn function_definition(
397     &mut self,
398     node: FunctionDefinition<'src>,
399 ) -> AnalyzedFunctionDefinition<'src> {
400     // set the function name
401     self.curr_func_name = node.name.inner;
402
403     if node.name.inner == "main" {
404         // the main function must have 0 parameters
405         if !node.params.inner.is_empty() {
406             self.error(/* ... */);
419         }
420
421         // the main function must return `()`
422         if let Some(return_type) = node.return_type.inner {
423             if return_type != Type::Unit {
424                 self.error(/* ... */);
433             }
434         }
435     }
    // ...
490     let block = self.block(node.block, false);
    // ...
535     AnalyzedFunctionDefinition {
536         used: true, // is modified in Self::analyze()
537         name: node.name.inner,
538         params,
539         return_type: node.return_type.inner.unwrap_or(Type::Unit),
540         block,
541     }
542 }
```

Listing 2.12 – Analyzer: Validation of the '`main`' function's signature.

This code displays the '`function_definition`' method of the analyzer. In this listing, only the code relevant for analyzing the '`main`' function is shown. However, this method is used to analyze any function, not just '`main`'. The method takes a '`FunctionDefinition`' as its input and returns an '`AnalyzedFunctionDefinition`'. Since a rush file might contain multiple functions, this method is invoked for each defined function.

In line 401, the method updates the current function name. The if-clause in line 403 checks whether the method is currently analyzing the '`main`' function. If this is the case, additional checks are performed. The next if-expression in line 405 checks whether the node's '`params`' vector contains any items. If this is the case, the '`main`' function's declaration includes parameters, and the analyzer generates an appropriate error message. However, error handling in the analyzer has not been discussed so far. As the method's signature states, the method cannot return any errors. Instead, the '`error`' method is invoked in line 406. This method uses information about the error to be generated in order to append a new '`Diagnostic`' to the '`diagnostics`' vector.

Secondly, in rush, the '`main`' function's return type always has to be '`()`'. The analyzer validates this constraint in lines 422–423. The first if-expression checks whether the function contains a manually defined result type. In rush, every function definition defaults to the unit type. Therefore, the inner if-expression is only executed if the user has manually specified a result type for their '`main`' function. The analyzer then checks whether the manually specified

type differs from the required unit type. In case it does, the analyzer generates another error which describes the issue in line 424.

After the signature of the '`main`' function has been validated, the method begins traversal of the function body. In line 490, the '`block`' method of the analyzer is invoked with the function body as its first argument. The second argument specifies that the method should not push another scope onto the stack since this is already handled by the current method. The return value of this method call is assigned to a variable called '`block`'. It represents the completely analyzed and annotated function body. Lastly, in lines 535–541, an '`AnalyzedFunctionDefinition`' is returned. Here, the function's attributes, like its analyzed parameters, return type, name, and body are specified.

During the traversal of the '`main`' function's body, the analyzer encounters two let-statements in lines 2–3. For analyzing this type of statement, the '`let_stmt`' method, which is shown in Listing 2.13, is invoked.

crates/rush-analyzer/src/analyzer.rs

```
612 fn let_stmt(&mut self, node: LetStmt<'src>) -> AnalyzedStatement<'src> {
        // ...
617     let expr = self.expression(node.expr);
        // ...
644     if let Some(shadowed) = self.scope_mut().insert(
645         node.name.inner,
646         Variable {/* ... */},
657     ) {/* ... */}
        // ...
682     AnalyzedStatement::Let(AnalyzedLetStmt {
683         name: node.name.inner,
684         expr,
685         mutable: node.mutable,
686         used: true,
687     })
688 }
```

Listing 2.13 – Analyzer: The '`let_stmt`' method.

In line 617, the initializing expression of the let-statement is analyzed first in order to obtain information about its result data type. The analyzer now inserts a new entry for the variable's name (e.g., '`two`') into its current scope in line 644. Even though the contents of the pushed variable are hidden, among other information, it includes the variable's type and span. Since the span includes the location of where the variable was defined, it can later be used in error messages, like the one previously displayed in Listing 2.11. Furthermore, the inserted information includes whether the variable was declared as mutable. If it is, reassignments are allowed, meaning that it can be updated to hold another value at a later point. In case a variable was not declared as mutable and reassigning to it was attempted, output comparable to the one in Listing 2.11 would be generated as a result. However, it seems very unusual that the insertion happens in the condition of an if-expression. If the insertion returns 'true', the variable's name was already present in the current scope, and its previously associated data has now been overwritten. The process of overwriting variables by redefining them is sometimes called *variable shadowing* [KN19, p. 34]. Here, the analyzer should display some additional hints or warnings, depending on whether the shadowed variable has been referenced before it was shadowed.

In order to understand how type determination and annotation work, the traversal of expressions should be considered. The code in Listing 2.14 is part of the method responsible for analyzing expressions.

crates/rush-analyzer/src/analyzer.rs

```
1007 fn expression(&mut self, node: Expression<'src>) -> AnalyzedExpression<'src> {
1008     let res = match node {
1009         Expression::Int(node) => AnalyzedExpression::Int(node.inner),
1010         Expression::Float(node) => AnalyzedExpression::Float(node.inner),
1011         Expression::Bool(node) => AnalyzedExpression::Bool(node.inner),
         // ...
1032         Expression::If(node) => self.if_expr(*node),
1033         Expression::Block(node) => self.block_expr(*node),
1034         Expression::Grouped(node) => {
1035             let expr = self.expression(*node.inner);
                 // ...
1040         }
             // ...
1046     }
1047
1048     res
1049 }
```

Listing 2.14 – Analyzer: Analysis of expressions during semantic analysis.

This method consumes a non-analyzed expression and transforms it into an analyzed one. For simple types of expressions, like integer, float, or boolean literals, further analysis is omitted, and an '`AnalyzedExpression`' can be constructed directly. For more complex types of expressions, like if-expressions, this method calls appropriate methods responsible for analyzing specific types of expressions.

In this function, the recursive tree traversal algorithm used in the analyzer is clearly visible. For instance, if the current expression is a grouped expression, like '`(1 + 2)`', the code in lines 1034–1040 is called. In line 1035, the '`expression`' method calls itself recursively using the inner expression of the grouped expression as the call argument. Most of the other tree traversing methods implement a similar recursive behavior.

crates/rush-analyzer/src/ast.rs

```
130 pub fn result_type(&self) -> Type {
131     match self {
132         Self::Int(_) => Type::Int(0),
133         Self::Float(_) => Type::Float(0),
134         Self::Bool(_) => Type::Bool(0),
          // ...
142         Self::If(expr) => expr.result_type,
143         Self::Block(expr) => expr.result_type,
144         Self::Grouped(expr) => expr.result_type(),
145     }
146 }
```

Listing 2.15 – Analyzer: Obtaining the type of expressions.

Listing 2.15 shows how the type of any analyzed expression can be obtained. For constant expressions, like '`Int(_)`', the determination of its type is straight-forward, as seen in line 132. Here, the '`result_type`' method returns '`Type::Int(0)`'. In this implementation, the '`Type`' enumeration saves a count which specifies the amount of pointer indirection. For instance, the rush type '`**int`' is represented as '`Type::Int(2)`' because there are two levels of pointer indirection. However, if the method is called on an integer literal, the resulting level of pointer indirection is zero. For more complex constructs, like if-expressions, the corresponding analyzed AST node saves its result type on its own. In this case, the type can then be accessed as seen in lines 142–143. For instance, during analysis of block expressions, the responsible function checks whether the block contains a trailing expression, and if it does, the result type of the block is identical to the one of its trailing expression. The last

case can be seen in line 144. Here, the result type of a grouped expression is obtained by calling the '`result_type`' method recursively. By using this method, the analyzer is able to determine type information about each node of the tree, assuming that it has been analyzed previously.

In the case of a semantically malformed program, the analyzer must continue tree traversal. Otherwise, only one error could be reported since every traversing method could potentially return an error which would terminate the tree traversal. To mitigate this issue, the '`Unknown`' type was implemented. For instance, the rush expression '`m + 42`' would cause the '`m`' variable to have the '`Unknown`' type if it was undefined. If the analyzer encounters a type conflict where one of the conflicting types is unknown, it does not report another error since the unknown type has to originate from a previous error. Therefore, errors do not cascade, meaning that an undeclared variable will not cause another type error.

In Listing 2.9 on page 14, below the let-statements in the source program, the '`exit`' function is called. Here, the analyzer uses the '`call_expr`' method in order to analyze the validity of this function call. For this, the analyzer iterates over the provided arguments, validating several constraints during each iteration. Among others, these include that each argument matches its corresponding parameter.

In this example, the argument expression '`two + three`' is traversed during this analysis. Since the identifiers on the left- and right-hand side have been declared by the two let-statements previously, obtaining their data types merely involves a lookup of the identifier names inside the current scope. If an unknown variable was provided, the lookup would yield no value, thus causing an error message to be generated.

Because the two variables should be added, the '`infix_expr`' method is called. It is responsible for analyzing any kind of infix-expression by validating several constraints. For instance, the operands must both be of the same type. In this example, both operands of the addition are integers. Therefore, the analyzer accepts this infix-expression and is now aware that it yields another integer. After the infix-expression's result type has been determined, it is saved in its own '`result_type`' struct field. Infix-expressions are another example for tree nodes that save their result type as a struct field on their own. Now that the analysis of the argument expression has completed, its compatibility with the declared parameter must be validated.

crates/rush-analyzer/src/analyzer.rs

```
1807 fn arg(/* ... */) -> AnalyzedExpression<'src> {
1814     let arg_span = arg.span();
1815     let arg = self.expression(arg);
1816
1817     match (arg.result_type(), param_type) {
1818         (Type::Unknown, _) | (_, Type::Unknown) => {}
1819         (Type::Never, _) => {
1820             self.warn_unreachable(call_span, arg_span, true);
1821             *result_type = Type::Never;
1822         }
         // ...
1823         (arg_type, param_type) if arg_type != param_type => self.error(/* ... */),
1829         _ => {}
1830     }
1831
1832     arg
1833 }
```

Listing 2.16 – Analyzer: Validation of argument type compatibility.

Listing 2.16 shows the '`arg`' function, which is responsible for validating that a function call argument is compatible with the declared parameter. This code will produce an er-

ror message if the type of the call argument deviates from the expected one. In order to validate the compatibility between the provided argument and the declared parameter, the method differentiates between several possible scenarios. In line 1818, the method detects the scenario in which the type of either the argument or the parameter is '`Unknown`'. Here, the method should ignore this argument without producing another error.

The next match-arm in line 1819 presents the scenario in which the provided argument has the '`Never`' type. In that case, the analyzer should only add a warning that the call-expression is unreachable. Furthermore, the result type of the entire call-expression is updated to reflect the never type. Line 1823 handles the scenario in which the type of the argument differs from the expected type of the parameter. In that case, the method will generate an error describing the situation. Again, the concrete error message is omitted for better readability.

In the case of the example program, the analyzer did not generate any error messages since the code presents both a syntactically and semantically valid rush program. Therefore, the analyzer accepts it and returns its syntax-tree with type annotations.

Early Optimizations

Another task of the analyzer can be to perform early optimizations. In compiler design, most of the optimizations are often performed with the target machine in mind. Therefore, the effects of these target-machine dependent optimizations can excel the ones caused by earlier optimizations. However, it is still sensible to perform trivial optimizations, such as *constant folding* and *loop conversion* inside the analyzer. For instance, the rush expression '`2 + 3`' evaluates to '5' during compile time instead of run time. This evaluation of expressions during compile time is referred to as constant folding. It is typically used in order to avoid the emission of otherwise redundant arithmetic instructions. As a result of this, the compiled program will run slightly faster since less computation is being performed when the program is executed [Wir05, p. 54].

crates/rush-analyzer/src/ast.rs

```rust
148 pub fn constant(&self) -> bool {
149     matches!(
150         self,
151         Self::Int(_) | Self::Float(_) | Self::Bool(_) | Self::Char(_)
152     )
153 }
```

Listing 2.17 – Analyzer: Determining whether an expression is constant.

In order to make such optimizations possible, each expression node in the analyzed AST has a method named '`constant`'. It is shown in Listing 2.17 and is responsible for determining whether an expression is constant. It returns 'true' if the expression is an integer, float, boolean, or character literal. Other types of expressions, such as call-expressions, cannot be constant since such a function call may cause side effects which cannot be determined during compile time. This method is vital for constant folding since both the left- and right-hand side of infix-expressions need to be constant in order to allow compile-time evaluation, and the analyzer has to check whether they are.

```rust
3 while true {
4     a += 1
5 }
```

Listing 2.18 – Redundant '`while`' loop inside a rush program.

Among other optimizations implemented in the analyzer, loop transformation can also have a positive effect on the program's performance during runtime. Listing 2.18 displays part of a rush program which uses a while-loop even though an unconditional loop would suffice. A '`loop`' implementation is more efficient since the condition check before each

iteration is omitted. However, this is only the case because the condition of the while-loop is a constant 'true'. If the analyzer detects such a scenario during analysis of a while-loop, the output node will be converted into an unconditional loop. Detection of this scenario is implemented in line 855 of Listing 2.19. In line 853, the match-arm is called if the analyzed loop will never iterate; it deletes the entire loop from the AST. This optimization improves runtime efficiency by a small amount since the code performing the very first condition check will not be compiled. Furthermore, the resulting output code will also be slightly smaller in size since the entire loop compilation can be omitted. The last match-arm in line 860 represents the unoptimized fallback case in which the loop is returned without modification.

crates/rush-analyzer/src/analyzer.rs

```
771 fn while_stmt(/* ... */) -> Option<AnalyzedStatement<'src>> {
        // ...
851     match (never_loops, condition_is_const_true) {
852         // if the condition is always `false`, return nothing
853         (true, _) => None,
854         // if the condition is always `true`, return an `AnalyzedLoopStmt`
855         (false, true) => Some(AnalyzedStatement::Loop(AnalyzedLoopStmt {
856             block,
857             never_terminates,
858         })),
859         // otherwise, return an `AnalyzedWhileStmt`
860         (false, false) => Some(AnalyzedStatement::While(AnalyzedWhileStmt {
861             cond,
862             block,
863             never_terminates,
864         })),
865     }
        // ...
866 }
```

Listing 2.19 – Analyzer: Loop optimization.

Implementing such trivial optimizations can significantly contribute to a more efficient output program without relying on the nuances of a target architecture. However, compiler writers often implement significantly more of those early optimizations than the aforementioned examples. In order to understand how early optimizations may impact the resulting AST, the two trees in Figure 2.4 should be considered.

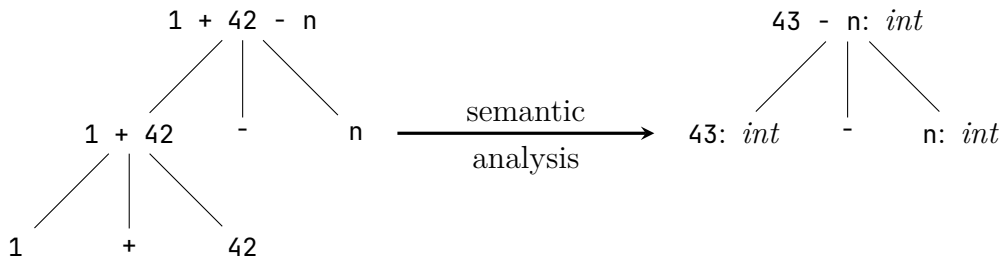


Figure 2.4 – How semantic analysis affects the abstract syntax tree.

Both trees in Figure 2.4 represent the rush expression '`1 + 42 - n`'. The left tree has been generated by the parser, closely representing the structure of the code. On the right, the same tree after it has been transformed by the analyzer is shown. Most notably, the sub-expression '`1 + 42`' has been evaluated to '`43`' during constant folding. Furthermore, all nodes of the right tree now contain type annotations, *int* in this case.

3. Interpreting the Program

After the syntactical and semantic analysis, there are now multiple different ways to continue. The first one shown here is an *interpreter*, which, contrary to a compiler, executes the analyzed program directly. That means, no output to be run by another process is produced and written to disk. Instead, everything is kept in memory and immediately evaluated by the interpreter [Mak09, Chapter 1]. Therefore, compared to a compiler, the *interpreter* not only has to be installed on the developer's machine, but also on the target machine. This chapter presents two fundamentally different ways of implementing an interpreted programming language.

3.1. Tree-Walking Interpreters

A *tree-walking* interpreter is probably the simplest form of programming language implementation, since it accepts the AST as its input directly, which is why it is the first one explained here. No further intermediate steps after the analysis are required. Just like the parser and analyzer, it walks (traverses) the tree, hence the name, and therefore requires one method per node type.

As seen in Listing 3.1, the struct definition is also rather small. It stores a list of '`scopes`', that being maps of variable names to their value at runtime, starting with the global scope at index '0' and ending with the current scope at any point in time. Why the '`Value`' is wrapped inside an '`Rc<RefCell<_>>`' is explained later in Section 3.1.3. Additionally, a map of function names to their tree node is saved. The '`Rc<_>`' here is only necessary to conform to Rust's borrowing rules without unnecessary cloning.

crates/rush-interpreter-tree/src/interpreter.rs

```
8  type ExprResult = Result<Value, InterruptKind>;
9  type StmtResult = Result<(), InterruptKind>;
10 type Scope<'src> = HashMap<&'src str, Rc<RefCell<Value>>>;
11
12 #[derive(Debug, Default)]
13 pub struct Interpreter<'src> {
14     scopes: Vec<Scope<'src>>,
15     functions: HashMap<&'src str, Rc<AnalyzedFunctionDefinition<'src>>>,
16 }
```

Listing 3.1 – Tree-walking interpreter: Type definitions.

In addition to the struct itself, result types for expressions and statements are defined. Statements either return '`()`' on success or an '`InterruptKind`' on failure. Expressions use the same error type, but return a '`Value`', holding the evaluated result, on success. The definitions for both '`Value`' and '`InterruptKind`' are visible in Listing 3.2. It is clearly visible that the interpreter simply makes use of Rust's types and does not need to implement hidden details, like comparisons between floats, manually.

The enumeration '`InterruptKind`' describes the different ways the interpreter can be interrupted. The first three variants, 'return', 'break', and 'continue', are only partial interrupts, i.e., they do not necessarily stop execution of the whole program. For example, when the interpreter encounters a '`break`' statement, it creates an '`InterruptKind::Break`' and tracks

back until it reached the innermost loop, where the interrupt is then caught and execution is continued after the loop. The second to last variant is for runtime errors, produced by events like division by zero, and the last one, '`InterruptKind::Exit(i64)`', is constructed by rush's built-in '`exit`' function and terminates the program. They cause the interpreter to backtrack all the way to the AST's root, and with that, cancel execution, as this is the desired behavior for errors and '`exit`' calls.

The way '`Value`' is implemented also makes it very easy to support dynamic typing[1], since it can be a value of any type. However, the rush interpreter can make use of the analyzer's guarantees of result types in order to omit certain checks internally, which could not be done with dynamic typing.

3.1.1. Implementation

Listing 3.3 displays the '`run`' method of the interpreter that serves as its entry point. Since the order of functions in rush is irrelevant and a function can be called before its definition, the interpreter must first populate its '`functions`' map before any function code is executed. Afterward, the global variables are assigned their initial values. Only then, the code inside the '`main`' function is called via the '`call_func`' method. After it returns, the entire interpreter either returns a runtime error, an explicit exit code, or the default exit code '0' indicating success.

The '`call_func`' method, visible in Listing 3.4, first checks whether a built-in function was called. As the only built-in function in rush is '`exit`', a simple equality check is sufficient. In case the called function is the '`exit`' function, the first call argument is asserted to be an integer via the '`unwrap_int`' method on '`Value`', and an exit interrupt kind containing that integer is returned immediately. Otherwise, the previously saved tree node of the function to be executed is retrieved from the '`functions`' map. A new variable scope for the function body is then initialized and filled with the passed arguments. Afterward, the function's block is evaluated by calling the '`visit_block`' method in a scoped environment. If the block returns a partial interrupt, it is turned into the appropriate return value by the '`into_value`' method call. For 'continue' and 'break', this is '`()`' and for '`InterruptKind::Return(Value)`', it is the wrapped value. The other two fatal interrupt kinds are simply passed along when encountered.

crates/rush-interpreter-tree/src/value.rs

```rust
pub enum Value {
    Int(i64),
    Float(f64),
    Char(u8),
    Bool(bool),
    Unit,
    Ptr(Rc<RefCell<Value>>),
}
// ...
pub enum InterruptKind {
    Return(Value),
    Break,
    Continue,
    Error(interpreter::Error),
    Exit(i64),
}
```

Listing 3.2 – Tree-walking interpreter: '`Value`' and '`InterruptKind`' definitions.

3.1.2. How the Interpreter Executes a Program

To provide a basic overview of a program's execution without too many implementation details, the evaluation of the rush program displayed in Listing 3.5 is now explained.

Firstly, the defined '`main`' function is called by the interpreter's entry point via '`call_func`'. In there, '`visit_block`' is called, using the '`main`' function's block as the argument. The

[1] In contrast to static typing, with dynamic typing the types of variables and results of expressions are only known at runtime rather than during compile time.

crates/rush-interpreter-tree/src/interpreter.rs

```rust
pub fn run(mut self, tree: AnalyzedProgram<'src>) -> Result<i64, Error> {
    for func in tree.functions.into_iter().filter(|f| f.used) {
        self.functions.insert(func.name, func.into());
    }

    let mut global_scope = HashMap::new();
    for global in tree.globals.iter().filter(|g| g.used) {
        global_scope.insert(/* ... */);
    }
    self.scopes.push(global_scope);
    // ...
    match self.call_func("main", vec![]) {
        Err(InterruptKind::Error(msg)) => Err(msg),
        Err(InterruptKind::Exit(code)) => Ok(code),
        Ok(_) | Err(_) => Ok(0),
    }
}
```

Listing 3.3 – Tree-walking interpreter: Beginning of execution.

crates/rush-interpreter-tree/src/interpreter.rs

```rust
fn call_func(&mut self, func_name: &'src str, mut args: Vec<Value>) -> ExprResult {
    if func_name == "exit" {
        return Err(InterruptKind::Exit(args.swap_remove(0).unwrap_int()));
    }

    let func = Rc::clone(&self.functions[func_name]);

    let mut scope = HashMap::new();
    for (param, arg) in func.params.iter().zip(args) {
        scope.insert(param.name, arg.wrapped());
    }

    self.scoped(scope, |self_| match self_.visit_block(&func.block, false) {
        Ok(val) => Ok(val),
        Err(interrupt) => Ok(interrupt.into_value()?),
    })
}
```

Listing 3.4 – Tree-walking interpreter: Calling of functions.

'`visit_block`' method then iterates over the statements contained in the block, and evaluates each one in order using the '`visit_statement`' method. In this case, that is only the call to '`exit`'. Since calls in rush are considered expressions, and expressions can also be used wherever statements are allowed by appending a '`;`', the '`visit_statement`' method only forwards to '`visit_expression`', which itself forwards to '`visit_call_expr`'. The call-expression then evaluates each argument expression by calling '`visit_expression`' again. Here, that involves another call-expression, this time of the '`plus_two`' function. It again evaluates its arguments, that being the access of the global variable '`global`' here, and then runs the function's block. The call stack at the point of reaching the '`return`' statement is displayed in Figure 3.1. Now, the expression '`num + 2`' is evaluated by first evaluating both sides of the '`+`' operator on their own, and then adding the results together.

Following that, an '`InterruptKind::Return(_)`' is constructed, holding the computed sum. By making use of Rust's '`?`' operator[2], the interrupt causes early returns in all function calls up to the bottom '`call_func`'. Thus, the statement '`global += 4;`' is never reached. All

[2]Can be used after expressions returning a '`Result<_, _>`' to return early in case they are the '`Err(_)`' variant.

```
let mut global = 40;

fn main() {
    exit(plus_two(global));
}

fn plus_two(num: int) -> int {
    return num + 2;
    global += 4;
}
```

Listing 3.5 – Example rush program.

other functions up to the top '`call_func`' then also return, as they now resolved to a value. After all this, the '`exit`' function call now has a value for its argument and can be performed. However, as described earlier, '`exit`' is built-in and does not call '`visit_block`', but instead constructs an '`InterruptKind::Exit(_)`' with the result value, so '42' in this case.

3.1.3. Supporting Pointers

Adding support for pointers in a tree-walking interpreter is actually not as straight forward as it is for the other backends, which all have a manually managed memory layout. It also depends a lot on the implementation language. In languages with a garbage collector, for example Go or Java, the pointer functionality of that language can just be reused, and the cleanup is already managed. However, unlike many modern languages, Rust does not have a garbage collector. Instead, it makes use of the so-called borrow checker that validates all references at compile time. Using default Rust references for pointers while conforming to Rust's borrowing rules turns out to be quite complicated though, but it is not required to use them. Rust provides an additional way of having pointers to values besides their reference system. The '`Rc<_>`' type, short for *reference counter*, stores its contained value on the heap and provides shared access to it, without any particular owner. The contained value is freed as soon as no more references to it exist [McN21, pp. 131–133]. This makes it ideal for implementing pointers in a tree-walking interpreter.

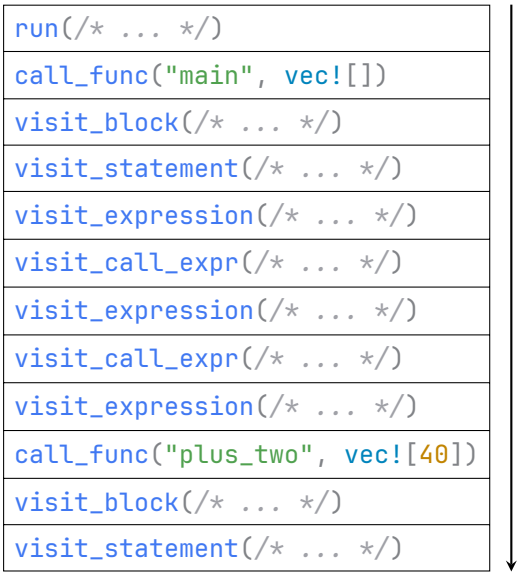


Figure 3.1 – Call stack at the point of processing the '`return`' statement.

However, as mentioned, a reference counter only provides **shared** access to the contained value. In Rust that implies not being able to mutate the value, as that would again break the borrowing rules. In order to still support mutable variables while having pointers to them, one can make use of another type the Rust standard library provides. A '`RefCell<_>`' implements so-called *interior mutability*[3] by enforcing Rust's borrowing rules at runtime. By wrapping values inside both a reference counter and a '`RefCell`', it is possible to support mutable variables and pointers [McN21, pp. 131–133].

[3] Types in Rust that have interior mutability allow mutation through shared references

3.2. Using a Virtual Machine

Just like a tree-walking interpreter, a virtual machine presents a way of implementing an interpreter for a programming language. However, the way a virtual machine operates differs fundamentally from a tree-walking interpreter. In order to compare the two, we have implemented a virtual machine backend for rush.

3.2.1. Defining a Virtual Machine

Often, one might encounter the term *virtual machine* (*VM*) when talking about emulating an existing type of computer using a software system. This emulation typically includes additional devices like the computer's display or its disk, whereas in this context, the term only describes a software entity which emulates how a processor interprets instructions. Since the VM is unable to traverse the AST, it depends on a compiler generating its input instructions.

Because a physical processor and a virtual machine share some fundamental traits, the architecture of a virtual machine is often closely resembling the *von Neumann architecture*. This architecture was first introduced by John Neumann in the year 1945. Von Neumann originally presented a design which allows implementing a computer using relatively few components. Following the von Neumann architecture, a processor would usually contain components like an *arithmetic logic unit* (*ALU*), a control unit, multiple registers, memory, and basic IO [Led20, p. 172]. The ALU's task is to perform logical and mathematical operations. Which ones is being told by the control unit within the *fetch-decode-execute* cycle. This cycle is a simplification of the steps a processor performs in order to execute instructions. The list below explains its individual steps [Led20, pp. 208–209].

- **Fetch**: The processor's control unit loads the next instruction from the adequate memory location. The instruction is then placed into the processor's internal instruction register where it is available for further analysis.
- **Decode**: The processor's control unit examines the fetched instruction in order to determine whether additional steps must be taken before or after instruction execution. Such steps may involve accessing additional registers or memory locations.
- **Execute**: The control unit dispatches the instruction to a specialized component of the processor. The target component is often dependent on the type of instruction since each processor component is designed with one specific type in mind. For instance, the control unit may invoke the ALU to execute a mathematical instruction.

A computer's processor performs this fetch-decode-execute cycle repeatedly from the moment it powers on to the point in time when it shuts down again. For relatively simple processors, each cycle is executed in isolation because instructions are executed in a sequential order. This means that the execution of the instruction i is delayed until execution of $i-1$ has completed [Led20, pp. 208–209]. More sophisticated processors often use many performance-enhancing techniques like *SIMD processing* or *simultanious multithreading* [Led20, pp. 217f].

In most cases, a virtual machine executes instructions similarly to the fetch-decode-execute cycle. Although the von Neumann architecture is relatively simple, one does not always have to adopt it when implementing a virtual machine. Since virtual machines are purely abstract constructs without physical limitations, design constraints are usually kept to a minimum. Therefore, a virtual machine can also be implemented with the high-level abstractions of its input language in mind. For instance, the VM might feature specialized 'break', 'continue', or 'loop' instructions which are not present in modern-day CPUs. Designing the architecture of a virtual machine can sometimes be a challenging task since choosing an adequate set of

features may involve a lot of testing iterations. Because neither the compiler nor the VM exist in the beginning, one should carefully plan the implementation of their VM's architecture.

3.2.2. Register-Based and Stack-Based Machines

One of the main decisions to be made during the design of the VM is how it implements temporary storage. Physical processors often use *registers* in order to make larger computations feasible. Registers are a limited set of very fast, low capacity storage units. On most relevant architectures today, like *x86_64*, each general-purpose register is able to hold as much as 64 bits of information. However, there is always only a limited amount of registers available since they are physical components of the CPU. Therefore, programs typically only utilize registers for storing temporary values, such as intermediate results of a large computation. The main alternative to registers is a stack-based design. A popular example for a stack-based virtual machine is *WebAssembly* [Sen22, p. 44]. For more information on WebAssembly, Section 4.3 presents a compiler targeting it. For compiler writers, managing registers is often a demanding task. This problem is described thoroughly in Chapter 5. On the other hand, targeting stack-based machines is usually significantly easier compared to register-based ones. Therefore, one might choose to implement a stack-based virtual machine in order to minimize the complexity of both the compiler and the interpreter. However, a stack-based design also introduces several issues on its own. For example, register-based machines might regularly outperform stack-based machines. A possible reason for this is that utilizing the stack typically requires a lot of seemingly redundant push and pop operations.

3.2.3. The rush Virtual Machine

The rush virtual machine is a stack-based machine implemented using the Rust programming language. The machine's architecture was solely developed for this project and includes a *stack* for storing short-term data, *linear memory* for storing variables, and another separate *call stack* for managing function calls. Like most virtual machines, the rush VM uses a fetch-decode-execute cycle in order to interpret programs.

crates/rush-interpreter-vm/src/vm.rs

```rust
pub struct Vm<const MEM_SIZE: usize> {
    /// Working memory for temporary values
    stack: Vec<Value>,
    /// Linear memory for variables.
    mem: [Option<Value>; MEM_SIZE],
    /// The memory pointer points to the last free location in memory.
    /// The value is always positive, but using `isize` does not require casts.
    mem_ptr: isize,
    /// Holds information about the current position (like ip / fp).
    call_stack: Vec<CallFrame>,
}
```

Listing 3.6 – Struct definition of the VM.

Listing 3.6 displays the struct definition of the rush VM. The '`stack`' field in line 18 saves the main stack for temporary values. In line 20, the '`mem`' field is declared. This field represents the linear memory used for storing variables. Since the '`Value`' is wrapped in an '`Option`', each cell may also hold a '`None`' value representing uninitialized memory. In line 23, the '`mem_ptr`' field is declared. It holds the *memory pointer*, which saves the index of the last free cell in '`mem`'. Lastly, a field named '`call_stack`' is declared. This field is responsible for managing function calls and returns.

The instructions in Listing 3.11 on page 30 can be interpreted by the rush VM. Programs for the rush VM are always partitioned into functions which each represents a sequence of instructions. Since function and variable names are replaced by indices, human-readable names causing inefficiencies can be omitted entirely. For better understanding, the individual functions have been manually annotated with their human-readable names.

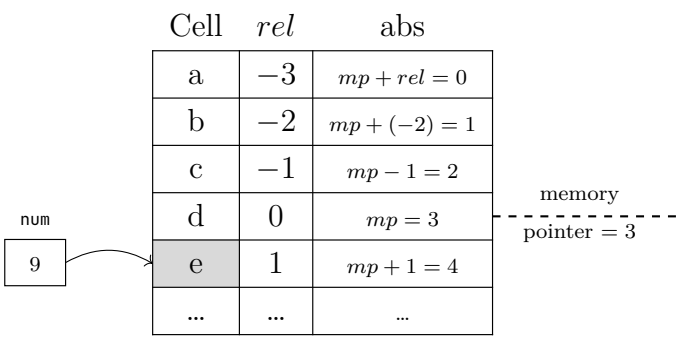


Figure 3.2 – Linear memory of the rush VM.

The first block of instructions is called *prelude*. Its only task is to declare global variables and call the '`main`' function. Global variables need to be initialized at the beginning of a program so that they can be accessed later. If the prelude was omitted, the '`main`' function would instead have to contain these instructions since it is executed at program start. However, recursion of the '`main`' function is allowed in rush. Therefore, each time the '`main`' function recurses, all global variables would be restored to their initial values. In order to prevent this bug, the rush VM uses a prelude function which is guaranteed to only run once.

Linear memory in the VM is represented as an array of runtime values of variables. Each storage cell can be accessed through two different addressing modes. When using the *absolute addressing* mode, the exact index of the memory cell is specified. For instance, if the value of the variable '`num`' in Figure 3.2 was to be retrieved, the VM would need to access the storage cell with the absolute index '4'. However, the absolute position of a variable in memory can only be determined at runtime. For example, in a recursive function, each recursion adds another scope containing variables, thus allocating more memory. However, the rush VM also implements a *relative addressing* mode which can be used without knowledge about the absolute position of the memory cell. For instance, the same variable can also be addressed through index '1' using the relative addressing mode. The mode is called *relative* because all addresses are specified relative to the memory pointer. In Figure 3.2, the memory pointer (shortened to *mp*) is set to '3'. Considering the value of the memory pointer, the absolute address of any relative address can be calculated at runtime. Here, the absolute address is the sum of the relative index *rel* and the runtime memory pointer *mp*. By also implementing this relative addressing mode, compilers targeting the rush VM can generate code without knowing the runtime behavior of a program. For better understanding, Figure 3.2 can be considered more closely. Here, the first column contains a character to identify each memory cell. The second column specifies the relative address of the cell, assuming that *mp* is set to '3'. Lastly, the third column contains the absolute address of the cell, calculated through *mp* and the relative address.

```
1 fn main() {
2     let mut num = 42;
3     let to_tum = &num;
4 }
```

Listing 3.7 – Minimal pointer example in rush.

In order to get a deeper understanding of the addressing modes, a practical example can be considered. The code in Listing 3.7 displays a rush program in which a pointer to a variable is created. First, in line 2, the integer variable '`num`' is created. In line 3, a pointer variable called '`to_num`' is created by *referencing* the '`num`' variable.

In the rush VM, absolute addressing is only used for global variables and pointers. Since a pointer specifies the address of another variable, its runtime value will be the absolute address of its target variable. In the VM, the absolute address of a variable is calculated as soon as it is referenced using the '`&`' operator. For this purpose, the '`reltoaddr`' (**rel**ative **to**

address) instruction exists. This instruction calculates the absolute address of its operand and pushes the result onto the stack. Here, the operand is the relative address of the variable to be referenced. Listing 3.8 shows the VM instructions generated from the rush program in Listing 3.7.

The first instruction '`setmp`' (**set** **m**emory **p**ointer) increases the memory pointer by two since the operand is a positive number. This is because the '`main`' function contains two local variables whose space is to be allocated at the start of the function. Next, the '`push`' instruction pushes the value '42' onto the stack. In line 4, the '`svari`' (**set** **var**iable **i**mmediate) instruction pops the top-most value from the stack, here '42', and assigns it to the specified relative address. Now, the variable '`num`', with an initial value of '42', has been created. Next, the '`to_num`' variable is created by referencing the '`num`' variable. In line 5, the '`reltoaddr`' instruction is used to calculate the absolute memory address of the '`num`' variable. The calculated absolute address is then pushed onto the stack where it can be accessed by the following instruction. Here, the relative address '0' is used since the '`svari`' instruction has previously saved '`num`' at this location. In line 6, another '`svari`' instruction is used to save the value of the '`to_num`' variable, that being the absolute address of '`num`' which is now on top of the stack, at the relative address -1. This is because the compiler targeting the VM assigns variables to higher relative addresses first. The compiler then progresses into lower relative memory as more variables of the function are initialized.

```
1  main:
2      setmp 2
3      push 42
4      svari *rel[0]
5      reltoaddr 0
6      svari *rel[-1]
```

Listing 3.8 – VM instructions for the minimal pointer example.

3.2.4. How the Virtual Machine Executes a rush Program

By considering the previous pointer example, one now has a rough idea of how the VM executes a program. In order to get a better understanding, the execution of the program in Listing 3.9 will now be explained. For this, the instructions in Listing 3.11 on page 30 should be considered again. The first instruction of the prelude function is '`setmp`'. This instruction adjusts the memory pointer by the amount specified in the instruction's operand. In this case however, the memory pointer remains unmodified since the operand of the instruction is '0'. Next, the '`call 1`' instruction calls the '`main`' function. In order to understand how function calls work in this VM, the call stack of the rush VM should be considered. Before the call-instruction, the caller pushes any arguments onto the stack so that they can be used as parameters by the callee. Figure 3.3 displays the state of the VM's call stack after the '`call 1`' instruction has been executed. During execution of a call-instruction, the VM pushes a new stack frame onto its call stack. Listing 3.10 shows how the '`CallFrame`' struct is implemented.

```
5  fn rec(n: int) -> int {
6      if n == 0 {
7          0
8      } else {
9          rec(n - 1)
10     }
11 }
```

Listing 3.9 – A recursive rush program.

crates/rush-interpreter-vm/src/vm.rs

```
29 struct CallFrame {
30     /// Specifies the instruction pointer relative to the function
31     ip: usize,
32     /// Specifies the function pointer
33     fp: usize,
34 }
```

Listing 3.10 – Struct definition of a '`CallFrame`'.

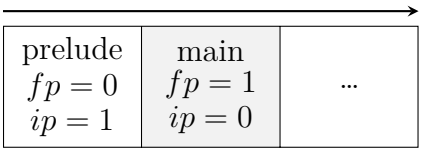


Figure 3.3 – Example call stack of the rush VM.

In this implementation, each call frame holds two important pieces of information. In line 31 of Listing 3.10, the '`ip`' field is declared. It specifies the *instruction pointer*, which saves the index of the current instruction. This index is relative for each function, meaning that the instruction pointer '0' can refer to multiple instructions, each in another function. Since the '`call`' instruction was interpreted previously, the instruction pointer of the new call frame is set to '0' as execution should continue at the first instruction of the called function. The second important field, '`fp`', is declared in line 33. It specifies the *function pointer*, which saves the index of the current function. Therefore, the combination of the instruction and function pointer specifies the instruction to be executed. After the function call, '`fp`' is set to '1' since '`ip`' should now refer to the instructions inside the '`main`' function.

Function calls are managed in a stack in order to allow early returns from functions. If the VM encounters a '`ret`' (**ret**urn) instruction, it should leave the current function immediately. However, it should also know where to resume its fetch-decode-execute cycle. For this, the VM simply pops the top element off its call-stack, meaning that the call frame of the current function is removed. Now, the top element on the stack contains the call-frame of the caller function. In this call frame, '`ip`' still points to the '`call`' instruction which was responsible for calling the function. Since '`ip`' is incremented automatically after most instructions, the VM resumes instruction execution at the first instruction after the '`call`' instruction. Therefore, the call frame of the caller function stays unaffected as long as the called function is executed.

```
1  0: (prelude)
2          setmp 0
3          call 1
4  1: (main)
5          setmp 0
6          push 1000
7          call 2
8          exit
9  2: (rec)
10         setmp 1
11         svari *rel[0]
12         push *rel[0]
13         gvar
14         push 0
15         eq
16         jmpfalse 9
17         push 0
18         jmp 14
19         push *rel[0]
20         gvar
21         push 1
22         sub
23         call 2
24         setmp -1
25         ret
```

Listing 3.11 – VM instructions matching the AST in Figure 3.4.

Now that the '`call`' instruction has been interpreted, the VM begins executing the first instruction of the '`main`' function. Since the '`main`' function only calls the '`rec`' function with the argument '1000', there are no new concepts to consider in this function. When the VM encounters the '`call`' instruction in line 7, execution continues with the instructions of the '`rec`' function. At the beginning of the '`rec`' function, the memory pointer is incremented by '1'. This might seem erroneous since the '`rec`' function contains no visible variable definitions in its body. However, this behavior is correct since function parameters are treated similarly to variables by the rush VM. Since the function takes one parameter, the memory pointer is incremented by one cell. Next, the '`svari`' instruction saves the value of the parameter which was previously pushed onto the stack at the relative address '0'. In line 12, the relative address of the memory cell containing the value of the parameter is pushed onto the stack. At this point, the top element on the stack contains an address value referring to the target of the '`gvar`' (**g**et **var**iable) instruction. It is then popped by the '`gvar`' instruction in line 13. Therefore, the instruction first pops the top element from the stack in order to retrieve the value of the variable at the specified location, and pushes the retrieved value onto the stack. In this case, the value of the popped element is the relative address '0', meaning that the instruction retrieves the value of first parameter.

In line 14, the constant value '0' is pushed onto the stack.

Next, the '`eq`' (is **eq**ual) instruction pops two elements off the stack in order to test them for equality. Then, the result of the equality test is pushed onto the stack as a boolean value. In other words, here, the instruction compares whether the current value of '`n`' is equal to '0' in order to produce a boolean result. The '`jmpfalse`' (**jump** if **false**) instruction in line 16 jumps to the specified instruction index if the boolean value on top of the stack is 'false'. In this example, if the value on the stack is 'false', the parameter '`n`' was not equal to '0'. If this was the case, the VM would jump to the instruction in line 19 as it represents index '9' of the '`rec`' function. Here, the value of the parameter '`n`' is pushed onto the stack using the previously explained '`push`' and '`gvar`' instructions. Now, the top item on the stack is the value of the parameter '`n`'. In line 21, the '`push`' instruction pushes a constant '1' onto the stack. Next, the '`sub`' (**sub**tract) instruction pops the first two elements off the stack in order to subtract their values from each other. In this case, the instruction subtracts '1' from the value of '`n`', pushing the result on the stack at the end. Next, the '`rec`' function calls itself recursively using the aforementioned '`call`' instruction. Since the call argument is the top element on the stack, the result of the subtraction is used as the argument of the recursive call. In line 24, the '`setmp`' instruction decrements the memory pointer in order to deallocate used memory. At the end of a function, the memory pointer is always decremented by the amount it was incremented by at the beginning of the function. By deallocating the now unused memory, the compiler prevents leaking the memory at runtime. Lastly, the '`ret`' instruction is used to return from the '`rec`' function. Now, the case in which the value of '`n`' is not equal to '0' was considered.

On the opposite, if the result of the comparison in line 15 was true, meaning that '`n`' was equal to '0', the '`jmpfalse`' instruction in line 16 would perform no operation. In this case, the VM continues execution at the '`push`' instruction in line 17. Here, the constant value '0' is pushed onto the stack. Since functions also return values by placing them on top of the stack, the return value would be '0' in this case. Next, the VM interprets the '`jmp`' (**jump**) instruction in line 18. Unlike '`jmpfalse`', this instruction performs its jump without any condition. Here, the instruction jumps to the instruction at index '14' of the current function, meaning '`setmp`' in line 24. The instructions in lines 24 and 25 return '0' in case '`n`' was equal to '0'.

3.2.5. Fetch-Decode-Execute Cycle of the VM

Now that the semantic meaning of the instructions in Listing 3.11 has been explained, the fetch-decode-execute cycle of the VM will be explained. Listing 3.12 displays the '`run`' method of the rush VM.

crates/rush-interpreter-vm/src/vm.rs

```
174 pub fn run(&mut self, program: Program) -> Result<i64> {
175     while self.call_frame().ip < program.0[self.call_frame().fp].len() {
176         let instruction = &program.0[self.call_frame().fp][self.call_frame().ip];
177 
178         // if the current instruction exists the VM, terminate execution
179         if let Some(code) = self.run_instruction(instruction)? {
180             return Ok(code);
181         };
182     }
183 
184     Ok(0)
185 }
```

Listing 3.12 – The '`run`' method of the rush VM.

This method manages the entire fetch-decode-execute cycle of the VM. It is immediately apparent that this method looks relatively simple considering that it plays such of a vital

role in the VM. Since the fetch-decode-execute cycle executes instructions repeatedly, the method's main construct is the while-loop beginning in line 175. The condition of the loop checks whether the current instruction pointer refers to a legal instruction inside the current function. This way, the VM comes to a halt as soon as it reaches the end of an instruction sequence. In line 176, the next instruction to be interpreted is saved in the '`instruction`' variable. This line represents the *fetch* step because the next instruction is fetched from memory and placed in a spot where it is accessible to the later steps of the cycle.

In the body of the loop, the current instruction is executed using the '`run_instruction`' method. This method is responsible for executing the previously fetched instruction. If execution of the instruction fails, the method returns a runtime error, such as integer overflow. Furthermore, this method may return an optional integer, representing the exit code of the program. If such an integer is returned, instruction execution comes to a halt instantly and the VM exists using it as the exit code. In case the method returns none of these two possible variants, the fetch-decode-execute cycle continues as usual. However, one cannot observe how the instruction pointer is incremented. Among other tasks, this is done by the code in Listing 3.13. It displays the '`run_instruction`' method. This method mainly consists of a

crates/rush-interpreter-vm/src/vm.rs

```
187 fn run_instruction(&mut self, inst: &Instruction) -> Result<Option<i64>> {
188     match inst {
189         Instruction::Nop => {}
190         Instruction::Push(value) => self.push(*value)?,
            // ...
197         Instruction::Jmp(idx) => {
198             self.call_frame_mut().ip = *idx;
199             return Ok(None);
200         }
            // ...
267         Instruction::Exit => {
268             return Ok(Some(self.pop().unwrap_int()));
269         }
            // ...
278         Instruction::Add => {
279             let rhs = self.pop();
280             let lhs = self.pop();
281             self.push(lhs.add(rhs))?;
282         }
            // ...
363     }
364     self.call_frame_mut().ip += 1;
365     Ok(None)
366 }
```

Listing 3.13 – Parts of the '`run_instruction`' method of the rush VM.

match-expression that determines which code to run depending on the current instruction. Here, the implementation of several instructions is visible. In line 190, the '`push`' instruction is handled. This instruction pushes its first operand onto the stack by calling the internal '`push`' method, which first validates that the stack will not exceed its maximum capacity. If the push operation causes the capacity of the stack to be exceeded, the method returns a runtime error describing the issue. In line 197, the code responsible for executing the '`jmp`' instruction can be seen. This instruction only sets '`ip`' to the target index specified by the first operand. It is to be noted that this instruction does not perform any bound checks, therefore allowing illegal jump destinations. For some special instructions, such as this one, '`ip`' should not be incremented since it would interfere with the jump operation. Since the instruction pointer is incremented at the end of the method, the '`return`' statement in line 199 is used to terminate this method early, preventing modification of '`ip`' at the end.

Line 267 contains the implementation of the '`exit`' instruction, which is inserted by the compiler if it encounters a call to the '`exit`' function. In this case, the instruction leads to the termination of the fetch-decode-execute cycle immediately. Line 278, handles the execution of '`add`' instruction. This instruction first pops two elements of the stack since they represent the operands of the underlying mathematical computation. Then, the '`add`' method is called on the left-hand side of the computation. It then performs the actual addition. This way, the '`run_instruction`' method stays organized and simple. Although just the code for addition is shown, most of the other infix-expressions are implemented similarly. As outlined previously, after an instruction has been executed, '`ip`' is incremented in line 364. If no error occurred during the execution of these instructions, nothing is returned since execution should proceed with the next instruction. This method represents both the *decode* and *execute* steps since it first matches (*decodes*) and then evaluates (*executes*) the current instruction. The compiler targeting the rush VM is presented in Section 4.2.

3.2.6. Comparing the VM to the Tree-Walking Interpreter

One significant benefit of virtual machines is that they execute programs faster compared to tree-walking interpreters. A reason for this speedup is that tree traversal involves a lot of overhead which is omitted when instructions are executed directly. The aforementioned Listing 3.9 on page 29 displayed a recursive function implemented in rush. Figure 3.4 displays a heavily simplified syntax tree representing this function. The root node of the tree represents the '`rec`' function. Since the function only contains a single expression, the if-expression node is the only child of the root node. The if-expression contains a condition, an if-branch, and an else-branch.

Since the function should not call itself again if '`n`' is equal to '0', the if-branch returns '0'. In the else-branch, the '`rec`' function calls itself recursively. When the above program is executed using the tree-walking interpreter, the algorithm traverses the entire tree of the '`rec`' function every time it recurses. In this example, the AST of the program is relatively simple. However, the complexity of the tree grows as the source program evolves. Since loops and recursive functions execute the code in their bodies repeatedly, the repeated tree traversal causes inefficiency. Here, the inefficiency solely lies in the repeated tree traversal, not in the repetition introduced by an iterative or recursive algorithm.

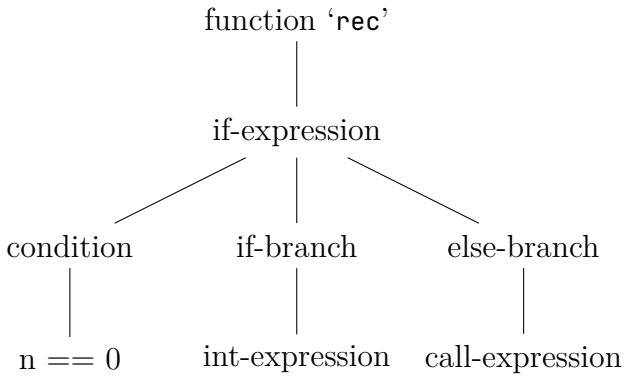


Figure 3.4 – AST and VM instructions of the recursive rush program in Listing 3.9.

In order to improve efficiency, an algorithm could traverse the AST once, saving its semantic meaning in the process. Then, the semantic meaning of the previously traversed tree could be interpreted repeatedly without the additional overhead. This behavior is used in the rush VM since it interprets instructions previously generated by a compiler. The compiler only traverses the AST once, generating VM instructions during the process. As mentioned previously, the instructions in Listing 3.11 on page 30 represent this recursive program. Every time the '`call`' instruction in line 23 is executed, the VM only needs to jump to the instruction in line 10 in order to call the '`rec`' function recursively. Since repeated traversal of the syntax tree is omitted, rush programs will run significantly faster when running on the VM compared to the tree-walking interpreter. Using the VM, executing the '`rec`' function

using an input of $\mathsf{n} = 1000$ took around 160 µs. However, executing the identical code using the tree-walking interpreter took around 427 µs[4]. The rush VM executed the identical code roughly 2.7 times faster than the tree-walking interpreter. However, the initial delay caused by compilation was not considered in this benchmark.

As a conclusion, a VM is often a reasonable approach if an interpreted programming language is to be implemented. The main advantages of a VM are increased speed, reduced memory usage at runtime, and fewer runtime errors due to type checking performed by its compiler. The main downsides include the need for a compiler targeting the VM, thus making its implementation more involved compared to a tree-walking interpreter. Furthermore, debugging the VM is often significantly more demanding than debugging a tree-walking interpreter. In the past, commercial software has shown that implementing a programming language leveraging a virtual machine can indeed be successful and on a larger scale. For instance, the *Java Virtual Machine* (*JVM*) is the software component responsible for executing the compiled *Java bytecode*. Through the JVM, the compiled Java program preserves its platform independence while using the benefits introduced by a VM [Lin+14, Section 1.2].

[4] Average from 10000 iterations. OS: Arch Linux, CPU: Ryzen 5 1500, RAM: 16 GB.

4. Compiling to High-Level Targets

In the previous chapter, the implementation of a programming language using an interpreter has been explained. However, another method of implementing a programming language is to create a compiler. This chapter presents three different compilers for high-level targets, and one transpiler.

4.1. How a Compiler Translates the AST

Often, a compiler traverses an AST generated by the analyzer in order to translate it to some sort of output. For each AST node, the compiler usually calls a separate function or method which is specialized in translating this specific node type, similar to the tree-walking interpreter. These individual methods return some sort of value representing the translated node. Otherwise, each individual method may also insert generated instructions into an internal field of the compiler. Here, the newly generated instructions are inserted into the output sequence. In this case, each method also returns metadata about the previously compiled node. For instance, this data may include the register or memory location of a previously compiled expression, so that other AST nodes can refer to it later. In transpilers, i.e., compilers translating one high-level language into another one, each node-specific method returns another tree node representing code of the output language.

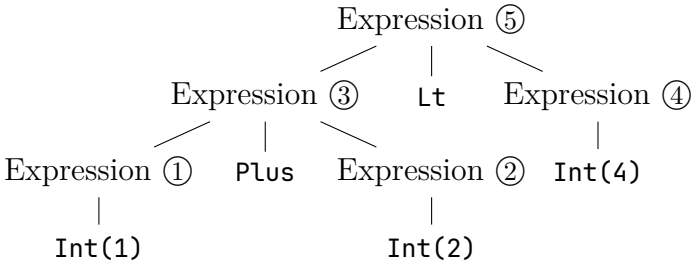


Figure 4.1 – Abstract syntax tree for '`1 + 2 < 4`'.

Figure 4.1 displays a simplified syntax tree of the rush expression '`1 + 2 < 4`'. The encircled number after each expression represents the order in which most compilers would traverse this tree. Compilation of the expression starts at the root node of the tree. Here, most compilers will begin translation by first compiling the child nodes using *post-order* traversal. Post-order traversal is frequently used because the compilation of a node often depends on the output of its child nodes. In this example, translation of the root comparison expression depends on the information returned by compiling its left- and right-hand sides. Therefore, the compiler first considers the node 'Expression ③' which represents the infix-expression '`1 + 2`'. However, due to post-order traversal, this node is not actually processed since the compiler skips straight to its child nodes. Therefore, the left child 'Expression ①' is traversed as the very first node. Next, its sibling node 'Expression ②' is traversed. Since post-order traversal involves considering a root node after the traversal of its children, 'Expression ③' is traversed after ① and ② have been considered. Here, the compiler considers the operator of the expression in order to generate the appropriate output instruction. Therefore, the instruction responsible for the addition is inserted after 'Expression ③' has been traversed. Since it and its children are now completed, and its output instruction has been inserted, the compiler now considers the right-hand side of the comparison. Here, the node 'Expression ④' only consists of the integer literal '4'. Now, all child nodes of 'Expression ⑤' have been traversed, thus the compiler now considers this node itself. Here, the compiler also

inserts the appropriate instruction, that is, a less-than comparison between integers. Since the compiler must be aware of the operands of the instruction, each method involved in the tree traversal returns an entity describing the location of the runtime value of its previously compiled node. For instance, if the target architecture uses registers, every method translating an expression must return the register or memory location containing the value of the expression at runtime. By returning information like this, a node will have information about its children after they have been traversed. It might rely on the values returned by its children, therefore it is traversed at last, thus creating the demand for post-order traversal.

```
1 r0 = 1
2 r1 = 2
3 r2 = add r0, r1
4 r3 = 4
5 r4 = lt r2, r3
```

Listing 4.1 – Simple pseudo-instructions for a fictional architecture.

Listing 4.1 displays a sequence of instructions for a fictional architecture. This sequence could have been generated from the previously discussed tree in Figure 4.1. It is apparent that the order of instructions matches the order in which the tree was traversed. The instructions in lines 1 and 2 represent the tree-nodes 'Expression ①' and 'Expression ②' respectively. Here, the value '1' is assigned to a register named '`r0`' while the value '2' is assigned to the register '`r1`'. The '`add`' instruction in line 3 appears after the instructions in lines 1 and 2 since their tree nodes were traversed first. Furthermore, the instruction uses the registers '`r0`' and '`r1`' as its operands and therefore depends on them containing a value. Therefore, the '`add`' instruction can use the registers returned by compiling its child nodes as its operands. Next, the integer '4' is assigned to the register '`r3`'. Lastly, the comparison instruction '`lt`' uses the result of the addition and the register containing '4' as its operands. Here, it is apparent that the instruction generated by the node which was traversed at last is also inserted at the end.

Therefore, using post-order traversal in order to generate output instructions targeting a register-based architecture is often required. This example illustrates how a simple compiler might operate. However, a similar algorithm is often found in even the most complex compilers.

4.2. The Compiler Targeting the rush VM

Since the rush VM interprets instructions directly, there must be a compiler targeting its architecture. For this purpose, a compiler translating rush source code into instructions, which can be executed by the VM, was implemented. Since the VM's architecture was developed with the features of rush in mind, the compiler sometimes requires surprisingly little effort for translating certain AST-nodes. For instance, the compiler translates infix-expressions, such as '`n + m`', into instructions using the '`infix_expr`' method. This method is displayed in Listing 4.2.

crates/rush-interpreter-vm/src/compiler.rs

```
522 fn infix_expr(&mut self, node: AnalyzedInfixExpr<'src>) {
523     match node.op {
524         InfixOp::Or | InfixOp::And => {/* ... */}
537         op => {
538             self.expression(node.lhs);
539             self.expression(node.rhs);
540             self.insert(Instruction::from(op));
541         }
542     }
543 }
```

Listing 4.2 – VM: Compilation of infix-expressions.

This method differentiates between the '`||`', '`&&`', and other possible operators since the former require extra code to be emitted. Here, the focus only lies on the compilation of the latter, meaning any other type of infix-expression. In line 538, the left-hand side expression is compiled first. In the next line, the right-hand side is compiled too. Finally, in line 540, the appropriate instruction representing the infix-operator is inserted. The instruction is generated by a separate function, '`Instruction::from`', which converts an infix-operator into a matching instruction. It is to be mentioned that most of the other rush compilers required significantly more code in order to implement the translation of infix-expressions. The reason for the simplicity of this implementation is the fact that the rush VM implements one instruction for each infix-operator. Since this VM implements all features of the source language, some complex operations are not required and thereby simplify the compiler's implementation.

crates/rush-interpreter-vm/src/compiler.rs

```
445 fn expression(&mut self, node: AnalyzedExpression<'src>) {
446     match node {
447         AnalyzedExpression::Int(value) =>
    ↪  self.insert(Instruction::Push(Value::Int(value))),
            // ...
457         AnalyzedExpression::Infix(node) => self.infix_expr(*node),
458         AnalyzedExpression::Call(node) => self.call_expr(*node),
459         AnalyzedExpression::Grouped(node) => self.expression(*node),
460     }
461 }
```

Listing 4.3 – VM: Compilation of expressions.

Listing 4.3 shows the '`expression`' method which takes an '`AnalyzedExpresssion`'. However, the method does not return anything which represents the runtime value of the expression. This is possible because the result of the expression is pushed onto the stack directly. In line 447, the code for translating an integer literal can be seen. Here, a '`push`' instruction using the integer as its operand is inserted. By pushing the values of expressions onto the stack, most methods do not need to return values. Due to this, short and elegant code, like the one in Listing 4.2, can be implemented. In other compilers, the method responsible for compiling expressions would usually return the register which will contain the value of the compiled expression at runtime. Due to the values being saved on the VM's stack, other parts of the compiled program can still use the runtime values of compiled expressions. Similar to the '`expression`' method of the analyzer, this method also dispatches the traversal of more complex expressions to specialized methods in order to keep its implementation simple. For instance, the method for compiling infix-expressions is invoked in line 457.

Another reason for the compiler's simplicity regarding infix-expressions is that the rush VM includes a special instruction for the mathematical power operation. In rush, the expression '`n ** m`' can be used to denote the mathematical expression n^m. Since many real architectures lack an instruction for power operations, most other rush compilers demanded implementation of special edge-cases in order to make compiling power-expressions feasible. The fact that the VM provides one instruction per operator shows how a carefully chosen target architecture simplifies the implementation of its compiler by a great deal.

However, there is also one aspect of the VM which made implementation of the compiler targeting the VM more demanding than usual. For instance, in most assembly dialects, *labels* can be used to allow jumps between blocks of code. However, the VM intentionally does not support the use of such labels. Since the VM would have to look up the exact instruction index of a label at runtime, each jump targeting a label would involve some additional overhead. Like seen in the previous examples, VM jump instruction require the exact index of the destination instruction as their operand. Therefore, the exact target index

to which the instruction should jump must be determined at compile time. To illustrate this issue, Listing 4.4 shows a rush program containing a loop, and Listing 4.5 displays the corresponding compiler output. In the loop's body, the variable '`n`' is incremented by '1'. Next, the '`break`' statement is used to terminate loop execution. Therefore, the total amount of iterations is '1'.

```rush
fn main() {
    let mut n = 0;
    loop {
        n += 1;
        break;
    }
}
```

Listing 4.4 – A rush program containing a loop.

```
setmp 1
push 0
svari *rel[0]
push *rel[0]
clone
gvar
push 1
add
svar
jmp 11
jmp 3
```

Listing 4.5 – Compiler output of the rush program in Listing 4.4.

In lines 2 and 3 of Listing 4.5 the variable '`n`' is initialized. The instructions in lines 4–9 are used to increment '`n`' by '1'. A new instruction which was not covered so far is the '`clone`' instruction. This instruction duplicates the top item on the stack it without prior calls to '`pop`'. Therefore, after the instruction has been executed, two identical values exist on the top of the stack. This instruction is only used in assign-expressions in order to duplicate the address value of the assignee.

After '`n`' was incremented, the instruction in line 10 jumps to the instruction index '11'. However, the last valid index is '10', it is represented by the '`jmp 3`' instruction in line 11. If this jump is executed, the VM has no next instruction to fetch and therefore stops its fetch-decode-execute cycle. Since it jumps to a position outside the loop, it represents the '`break`' statement in line 5 of the source program. The '`jmp`' instruction in line 11 is responsible for the repetition introduced by the loop. It jumps to the first instruction of the loop's body in line 4. Therefore, the instructions inside the loop's body are executed repeatedly. The difficulty presented by this design is that the index of the jump's target instruction must be determined before the target instruction is inserted. Listing 4.6 displays the method responsible for compiling loops.

crates/rush-interpreter-vm/src/compiler.rs

```rush
fn loop_stmt(&mut self, node: AnalyzedLoopStmt<'src>) {
    // save location of the loop head (for continue stmts)
    let loop_head_pos = self.curr_fn().len();
    self.loops.push(Loop::default());
    // ...
    self.block(node.block, true);
    // ...
    self.insert(Instruction::Jmp(loop_head_pos));

    // correct placeholder `break` / `continue` values
    let loop_ = self.loops.pop().expect("pushed above");
    let pos = self.curr_fn().len();
    self.fill_blank_jmps(&loop_.break_jmp_indices, pos);
    self.fill_blank_jmps(&loop_.continue_jmp_indices, loop_head_pos);
}
```

Listing 4.6 – VM: Implementation of loops.

In line 331, the loop's body is compiled and instructions generated during this process are inserted into the output sequence. The next statement in line 337 inserts an instruction responsible for jumping back to the start of the loop's body. The variable '`loop_head_pos`' was previously defined and specifies the index of the first instruction of the loop's body.

In line 340, the top loop is popped off the '`loops`' stack. In line 322, this loop was previously pushed onto the '`loops`' stack. This stack is an internal field used by the compiler in order to save information about loops. The top item on it always represents the loop currently traversed by the compiler. Each loop item saves two lists, each containing the indices of jump instructions whose target index needs to be adjusted. The first list contains the indices of jump instructions generated by '`break`' statements which were encountered during traversal of the loop's body. The second list saves the indices of jump instructions emitted by '`continue`' statements. For instance, if the compiler encounters a '`break`' statement, the code in Listing 4.7 is executed.

crates/rush-interpreter-vm/src/compiler.rs

```
253 fn statement(&mut self, node: AnalyzedStatement<'src>) {
            // ...
268         AnalyzedStatement::Break => {
269             // the jmp instruction is corrected later
270             let pos = self.curr_fn().len();
271             self.curr_loop_mut().break_jmp_indices.push(pos);
272             self.insert(Instruction::Jmp(usize::MAX));
273         }
            // ...
287     }
288 }
```

Listing 4.7 – VM: Compilation of '`break`' statements.

Here, the '`pos`' variable saves the index of the jump instruction to be inserted. In line 271, this index is then inserted into the list containing the placeholder indices of the current loop. Lastly, a '`jmp`' instruction is inserted containing a placeholder target index. In line 342 of Listing 4.6, the '`fill_blank_jmps`' method is used to set the target indices of the specified jump instructions to '`pos`'. The explanation of this method is omitted because it only iterates over the passed list of indices, replacing the target of each jump instruction during the process.

4.3. Compilation to WebAssembly

The first external compilation target[1] presented here is *WebAssembly*, or *Wasm* for short. It aims to be a safe, portable, low-level, efficient, and compact code format, mainly intended for enabling high performance applications in websites. However, unlike the name implies, WebAssembly is not only used in web applications. By itself, it is only a specification that can be implemented by runtimes in any context [Ros22, Section 1.1]. Most modern browsers include such a WebAssembly runtime, but there are also standalone ones, for example *wasmtime* and *wasmer*.

4.3.1. WebAssembly Modules

Every valid WebAssembly file must contain exactly one module. The WebAssembly specification defines two different representations for these modules. First, there is a human-readable text representation, called *WAT*[2], closely resembling S-Expressions[3]. This is comparable to assembly languages for CPU architectures and is the typical target for compilers. Secondly, WebAssembly modules can also be represented in a binary format, which is optimized for size, and comparable to the binary files produced by assemblers [Sen22, pp. 40–44]. Typically, these binary modules are constructed from a text module by using a tool such as *wat2wasm* from the *WebAssembly Binary Toolkit (WABT)* [Sen22, p. 57]. However, the rush WebAssembly compiler instead opts to target the binary format directly, highlighting a few reasons for why most compilers should not follow this approach. Listing 4.8 and Listing 4.9 on page 41 show the same basic WebAssembly module once as WAT and once as a commented hex dump of the same module in its binary representation as produced by wat2wasm.

Focusing on the text representation first, the module contains one function that takes two '`i32`' parameters and returns a single '`i32`'. An '`i32`' in WebAssembly represents an uninterpreted 32-bit integer, meaning that it is not clear whether the integer is signed or unsigned from the type itself. Instead, values of this type can be interpreted as either signed or unsigned by different instructions [Ros22, Section 2.3.1]. For instance, the instruction '`i32.eq`', which checks for equality between two '`i32`' values, behaves the same disregarding how the integer is signed. In contrast, '`i32.lt_s`' and '`i32.lt_u`' are two instructions both querying whether one '`i32`' is less than another, once for signed and once for unsigned integers, as denoted by the suffix [Sen22, p.46].

The mentioned function is exported by the module under the name 'addTwo' to make it accessible externally. What exactly 'externally' means depends on the context the module is executed in. WebAssembly is *stack based* and has one primary stack instructions can operate on. The first two instructions of the '`addTwo`' function retrieve the local variable of the given index and push its value onto the stack. 'Locals' in WebAssembly are simple values separate from the main stack. Function parameters are always the first locals, but additional ones can be added too. After the two instructions have been executed, the stack now contains the values of the two function parameters. They are then added by the '`i32.add`' instruction, which pops the top two elements off the stack and pushes the sum back onto it. The implicit return value is always what remains on the stack at the end of a function body, just like in the rush VM.

Now, the hex dump of the same module in binary, displayed in Listing 4.9, is to be considered. A WebAssembly binary file always starts with the four bytes '`00 61 73 6d`'

[1] Here, "external" describes that the compiler emits an output file which is then executed or further processed by another, already existing tool.

[2] Short for "WebAssembly Text" [Sen22, p. 40].

[3] Short for "symbolic expression", used for representing nested, tree-structured data [Sen22, p. 41].

```
(module
  (func (export "addTwo") (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add))
```

Listing 4.8 – Simple WebAssembly module in text representation.

```
0000000: 0061 736d                 ; WASM_BINARY_MAGIC
0000004: 0100 0000                 ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01                        ; section code
0000009: 07                        ; section size
000000a: 01                        ; num types
; func type 0
000000b: 60                        ; func
000000c: 02                        ; num params
000000d: 7f                        ; i32
000000e: 7f                        ; i32
000000f: 01                        ; num results
0000010: 7f                        ; i32
; section "Function" (3)
0000011: 03                        ; section code
0000012: 02                        ; section size
0000013: 01                        ; num functions
0000014: 00                        ; function 0 signature index
; section "Export" (7)
0000015: 07                        ; section code
0000016: 0a                        ; section size
0000017: 01                        ; num exports
0000018: 06                        ; string length
0000019: 6164 6454 776f    addTwo  ; export name
000001f: 00                        ; export kind
0000020: 00                        ; export func index
; section "Code" (10)
0000021: 0a                        ; section code
0000022: 09                        ; section size
0000023: 01                        ; num functions
; function body 0
0000024: 07                        ; func body size
0000025: 00                        ; local decl count
0000026: 20                        ; local.get
0000027: 00                        ; local index
0000028: 20                        ; local.get
0000029: 01                        ; local index
000002a: 6a                        ; i32.add
000002b: 0b                        ; end
; section "name"
000002c: 00                        ; section code
000002d: 0a                        ; section size
000002e: 04                        ; string length
000002f: 6e61 6d65           name  ; custom section name
0000033: 02                        ; local name type
0000034: 03                        ; subsection size
0000035: 01                        ; num functions
0000036: 00                        ; function index
0000037: 00                        ; num locals
```

Listing 4.9 – Simple WebAssembly module in binary representation.

called the *Wasm binary magic*, representing a zero byte followed by the string 'asm' using ASCII representation. This is used by other programs to easily identify binary files as WebAssembly modules. Following that is the version of the binary format, stored as a 32-bit integer in little-endian[4]. At the time of writing it is always '1' [Ros22, Section 5.5.16].

The binary module is then split into different sections, each containing one kind of information about the entire module. Empty sections can be omitted. Each section begins with its identifier, followed by the section size in bytes. Most sections contain one vector of relevant data, and vectors always start with the count of elements they contain, and continue with the elements themselves. The first section present here is the 'Type' section. It declares different types used by the module, most importantly, the function signatures. The 'Function' section then contains the number of functions of the current module and simply references the 'Type' section for each function's signature. The module's exports are declared in the 'Export' section. Finally, the 'Code' section contains the actual instructions for each function. It is again stored as a vector, containing function bodies for all functions defined in the 'Function' section in the same order. Each function body begins with its size in bytes, continues with the instructions, and ends with an '`end`' instruction represented by a '`0b`' byte [Ros22, Sections 5.5, 5.1.3, 5.4.9].

The wat2wasm tool used here additionally adds a custom 'name' section. Custom sections always have the ID '0' and must provide a custom name using UTF-8 encoding. This 'name' section has its own specification separate from the main module specification, and is used to provide names for functions and variables that can then be used by development tools like debuggers [Ros22, Section 7.4.1].

Apart from exporting, WebAssembly modules can also import external functions. Only the name and type signature must be provided, and the WebAssembly runtime will then have to provide an implementation during program execution. Furthermore, WebAssembly does not only have local variables, but also global ones, accessible from every function. These must be initialized with a constant value and can either be mutable or immutable.

One may have already noticed that except for the version number at the start, all sizes, indices, lengths, and alike, have been stored using just a single byte. But, this is not because those can only reach a maximum of '255', but instead WebAssembly uses the LEB128 encoding for integer literals in binary modules. It is a space efficient way to store integers by only ever needing as many bytes as necessary for a number [Ros22, Section 5.2.2]. However, the encoding details are not explained here, as the rush Wasm compiler simply uses a pre-existing crate called "leb128".

4.3.2. The WebAssembly System Interface

Since WebAssembly itself is independent of a runtime, it cannot provide standardized ways to interact with the environment, except for module imports and exports. That is why an additional specification called the *WebAssembly System Interface*, short *WASI*, was created for WebAssembly modules that need to communicate with an operating system for instance. Any runtime supporting WASI must provide a set of functions comparable to *system calls* on Linux or Windows. These can then be imported from a WebAssembly module to, e.g, write to a console or exit with a specific exit code. Both wasmtime and wasmer implement the WASI interface.

A WebAssembly module making use of WASI must export one function under the name '`_start`' that acts as the entry point of the program. The rush WebAssembly compiler only ever imports WASI's '`proc_exit`' function, which takes one 32-bit integer as an argument and terminates execution with the given code.

[4]Little-endian starts with the least significant byte, whereas big-endian starts with the most significant one.

4.3.3. Implementation

As indicated, the rush WebAssembly compiler targets the binary format directly. This complicates compilation in a few ways, but removes the need for any external dependencies. First, public constants are defined for all instructions and all types in separate files. Listing 4.10 shows an extract containing the '`i64.add`' and '`i64.sub`' instructions.

crates/rush-compiler-wasm/src/instructions.rs

```rust
/// i64.add = 0x7C
pub const I64_ADD: u8 = 0x7C;

/// i64.sub = 0x7D
pub const I64_SUB: u8 = 0x7D;
```

Listing 4.10 – Wasm: Definition of instruction opcodes.

The '`Compiler`' struct has a lot of fields for various purposes. Only some are shown in Listing 4.11 and explained here. To begin, multiple fields regarding the currently compiled function are defined. The field '`function_body`' contains the bytes with instructions for the current function. The field '`locals`' stores which local variables the function has, along with their types. In the binary format, the locals are stored as a WebAssembly vector, starting with the number of locals, followed by each local. Since the compiler cannot know the count of local variables before compiling the function body, it stores them as a vector of byte vectors first. This way, in the end, it can first append the vector's length to the final output and then concatenate the vector's contents. The three following fields all map names to indices. One for local variable scopes, one for the global scope, and one for function names. Each index itself is stored as a vector of bytes, as it uses the aforementioned LEB128 encoding which can vary in length. Finally, one field for every supported section is defined. These are of the type '`Vec<Vec<u8>>`' for the same reason as '`locals`'.

crates/rush-compiler-wasm/src/compiler.rs

```rust
pub struct Compiler<'src> {
    /// The instructions of the currently compiling function
    pub(crate) function_body: Vec<u8>,
    /// The locals of the currently compiling function
    pub(crate) locals: Vec<Vec<u8>>,
    // ...
    /// Maps variable names to `Option<local_idx>`, or `None` when of type `()`
    pub(crate) scopes: Vec<HashMap<&'src str, Option<Vec<u8>>>>,
    /// Maps global variable names to `global_idx`
    pub(crate) global_scope: HashMap<&'src str, Vec<u8>>,
    /// Maps function names to their index encoded as unsigned LEB128
    pub(crate) functions: HashMap<&'src str, Vec<u8>>,
    // ...
    pub(crate) type_section: Vec<Vec<u8>>,     // 1
    pub(crate) import_section: Vec<Vec<u8>>,   // 2
    pub(crate) function_section: Vec<Vec<u8>>, // 3
    // ...
}
```

Listing 4.11 – The '`Compiler`' struct definition of the WebAssembly compiler.

Listing 4.12 contains the compiler's entry point method. Some details are omitted, but essentially, it simply calls another method to compile the program itself, and then concatenates all sections together to form the final binary. It also imports all required functions from WASI and exports blank linear memory as required by WASI. The '`Self::section`' function is used to add the section identifier, byte length, and element count in front of each section's contents.

Inside the '`program`' method, the global variables are defined and initialized, and all function signatures are added to the 'Type' and 'Function' sections and the '`functions`' map. Afterward, it calls '`function_definition`' for every defined function. This has to be parti-

crates/rush-compiler-wasm/src/compiler.rs

```rust
70  pub fn compile(mut self, tree: AnalyzedProgram<'src>) -> Vec<u8> {
        // ...
81      self.program(tree);
        // ...
104     [
105         &b"\0asm"[..],        // magic
106         &1_i32.to_le_bytes(), // spec version 1
107         &Self::section(1, self.type_section),
108         &Self::section(2, self.import_section),
109         &Self::section(3, self.function_section),
            // ...
124     ]
125     .concat()
126 }
```

Listing 4.12 – Wasm: Entry point of the compiler.

tioned into two steps because rush allows a function to be called before its definition. For every function body, all statements and the optional trailing expression are compiled first, and then the values of '`function_body`' and '`locals`', along with their combined length are appended to the 'Code' section.

All other nodes have a matching method defined again, like every time an AST is traversed. Each of those methods simply pushes instructions to '`function_body`'. Because WebAssembly is stack-based, none of them have to return any value, not even the expressions. They simply add their instructions to the resulting code and call methods for nested nodes beforehand. This will result in the intended behavior, just like in the rush VM.

Function Calls

Compared to the other compilers presented later, supporting functions and function calls is very straight forward for WebAssembly. It was already explained that WebAssembly modules have a concept of functions with parameters and return values, so mapping rush functions to these is the obvious strategy. Calling a declared function is as simple as compiling all argument expressions in order, causing all evaluated arguments to be on top of the stack, and emitting a '`call`' instruction with the target function's index.

Logical Operators

One special case to highlight is the compilation of logical operators[5] in infix-expressions. This comes down to the fact that in many programming languages, and likewise in rush, these operators evaluate *lazily*, that is, they only evaluate their right-hand side if the result is not already predeterminable by the left-hand side. Listing 4.13 shows the relevant part of the '`infix_expr`' method. At the start of this method, there are special cases for the '`&&`' and '`||`' operators. Here, only the code for the '`&&`' operator is shown, but the '`||`' operator is handled similarly. It is evident that the operation is compiled as if it were an if-expression, like '`if !lhs { false } else { rhs }`'. Using this strategy, the expected behavior of lazy evaluation is achieved, unlike if both sides had been compiled and the results were compared using an '`i32.and`' instruction. The negation of the left-hand side is done in line 764 using the '`i32.eqz`' instruction, which consumes an '`i32`' from the top of the stack and replaces it with either '0' or '1' based on whether the value was equal to zero. Because the top-most value is already a boolean, the instruction has the effect of negating it. In line 777 an early return is issued to skip the code used for all other infix operators.

[5] Operators that only operate on boolean values, typically '`&&`' and '`||`'.

crates/rush-compiler-wasm/src/compiler.rs

```
758 fn infix_expr(&mut self, node: AnalyzedInfixExpr<'src>) {
759     match node.op {
760         InfixOp::And => {
761             self.expression(node.lhs);
762             self.function_body.extend([
763                 // if lhs is not true
764                 instructions::I32_EQZ,
765                 instructions::IF,
766                 types::I32,
767                 // then return false
768                 instructions::I32_CONST,
769                 0,
770                 // else return rhs
771                 instructions::ELSE,
772             ]);
773             self.expression(node.rhs);
774             // end if
775             self.function_body.push(instructions::END);
776 
777             return;
778         }
779         InfixOp::Or => {/* ... */}
797         _ => {}
798     }
    // ...
864 }
```

Listing 4.13 – Wasm: Compilation of logical operators.

The other compilers explained in this paper, while not mentioning it again, all use the same strategy for these logical operators.

4.3.4. Considering an Example rush Program

Listing 4.14 shows an example rush program containing a global integer variable '`a`', a local boolean variable '`b`', and a call to the built-in '`exit`' function. The generated compiler output, converted to WAT using wasm2wat and manually commented, is shown in Listing 4.15. Since the compiler detects usage of the '`exit`' function, it imports the corresponding '`proc_exit`' function from WASI under the name '`__wasi_exit`' in line 4. Below the 'main' function are the definition and the export of blank memory in lines 23 and 26 respectively, as to conform with WASI's requirements. In line 25, the 'main' function is also exported as '`_start`' and also separately declared as the module's entry point in line 27. Both is done in order to have this function be the one to start execution. Line 24 declares the global '`a`' as a mutable '`i64`' and sets the initial value of '2'.

```
1 let mut a = 2;
2 
3 fn main() {
4     a += 1;
5     let b = true;
6     exit(a + b as int);
7 }
```

Listing 4.14 – Example rush program.

Inside the 'main' function, the first thing is the declaration of the local variable '`b`' with the type '`i32`' in line 6. The type is '`i32`', even though a boolean would really only need one bit because WebAssembly only defines this and '`i64`' as integer types, and therefore uses the smaller of the two for booleans. As indicated by the comments, the first four instructions represent the rush statement '`a += 1;`' from line 4 of the source. They push both the current value of '`a`' and a constant '1' onto the stack, which are then added together by the '`i64.add`'

```
1  (module
2    (type (;0;) (func (param i32)))
3    (type (;1;) (func))
4    (import "wasi_snapshot_preview1" "proc_exit" (func $__wasi_exit (type 0)))
5    (func $main (type 1)
6      (local $b i32)
7      ;; a += 1;
8      global.get $a           ;; push the current value of `a`
9      i64.const 1             ;; push `1`
10     i64.add                 ;; add those two
11     global.set $a           ;; pop and set `a` to the result
12     ;; let b = true;
13     i32.const 1             ;; push `true`
14     local.set $b            ;; pop and set `b`
15     ;; exit(a + b as int);
16     global.get $a           ;; push `a`
17     local.get $b            ;; push `b`
18     i64.extend_i32_u        ;; cast bool to int
19     i64.add                 ;; add those two
20     i32.wrap_i64            ;; convert i64 to i32 (for `__wasi_exit`)
21     call $__wasi_exit       ;; call `__wasi_exit`
22     unreachable)
23   (memory (;0;) 0)
24   (global $a (mut i64) (i64.const 2))
25   (export "_start" (func $main))
26   (export "memory" (memory 0))
27   (start $main))
```

Listing 4.15 – Commented compiler output of the rush program in Listing 4.14.

instruction. The resulting sum is then popped off the stack and set as the new value for '`a`'. To set the value of '`b`' to 'true', a constant '1' is pushed in line 13 and then used as the new value for '`b`' in line 14.

The call to '`exit`' is a bit more complex. First, the values of both '`a`' and '`b`' are retrieved. The cast from a boolean to an integer is performed by the single instruction '`i64.extend_i32_u`', which zero-extends[6] the 32-bit value to a 64-bit one. The two integers are then added together in line 19. Because WASI's '`proc_exit`' function only takes a 32-bit integer for the exit code, but rush uses 64-bit integers for its '`int`' type, the argument value must be converted into an '`i32`' first. This happens in line 20 with the '`i32.wrap_i64`' instruction. In case the value is too large to fit into a 32-bit integer, it is wrapped, effectively only interpreting the lower 32 bits of the 64-bit number. After this, the call to '`__wasi_exit`' can now be performed without issues.

One might notice the trailing '`unreachable`' instruction in line 22. The rush WebAssembly compiler inserts one such instruction after every expression that has the '`!`' type. As the name suggests, the instruction guarantees the WebAssembly runtime that it is never reached. This is helpful in cases where such expressions are used inside others, such as '`1 + exit(2)`', and the semantic analyzer ignores the invalid type of the right-hand side because it knows the outer expression will never be reached during runtime. The same information must be given to the WebAssembly runtime using the '`unreachable`' instruction since the specification requires runtimes to validate types in modules before executing them.

[6] Converting an n-bit sized integer to an m-bit sized one, where $m > n$, by prepending zeros to the binary representation.

4.4. Using LLVM for Code Generation

LLVM is a software project intended to simplify the construction of a compiler generating highly performant output programs. It originally started as a research project by *Chris Lattner* for his master's thesis at the University of Illinois at Urbana-Champaign [Lat02]. Since then, the project has been widely adopted by the open source community. In 2012, the project was rewarded the *ACM Software System Award*, a prestigious recognition of significant software which contributed to science. From the point when popularity of the framework grew, it was renamed from *Low Level Virtual Machine* to the acronym it is known by today. Nowadays, it can be recognized as one of the largest open source projects [CA14, preface]. Among many other projects, the Rust programming language depends on LLVM to generate its target-specific code [McN21, p. 373]. Furthermore, the *Clang* C and C++ compiler leverages LLVM as its code generating backend [Hsu21, preface]. Besides open source projects, many companies have also utilized LLVM in their commercial software. For instance, since 2005, Apple has started incorporating LLVM into some of its products [Fan10, pp. 11-15], including their *Swift* programming language which is mainly used for developing iOS apps [Hsu21, preface].

4.4.1. The Role of LLVM in a Compiler

In a compiler system, LLVM is responsible for generating target-specific code and performing optimizations. The framework is known for performing very effective optimizations during code generation so that the translated program executes faster at runtime, uses less memory and is smaller in size. In order to interact with LLVM, the system provides an API with which is usable by other software. A compiler leveraging LLVM traverses the AST and uses this API in order to construct an *intermediate representation* (*IR*) of the program. This way, the framework will be able to comprehend the semantic meaning of the program. Next, LLVM compiles this IR to an output targeting any of its supported platforms. As of today, LLVM features many target platforms so that a compiler designer does not have to implement portability manually [Hsu21, preface]. Listing 4.2 shows how LLVM integrates into the previously discussed steps of compilation. The framework represents the *backend* component of a compiler.

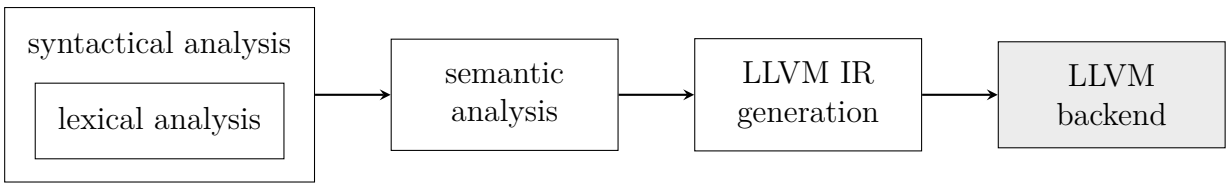


Figure 4.2 – Steps of compilation when using LLVM.

The updated Figure 4.2 now includes two new steps: "LLVM IR generation" and "LLVM backend". The former traverses the AST in order to generate a semantically equivalent program formulated using LLVM IR. The latter uses the IR as its input in order to generate target-specific code. A compiler writer must only implement the first three steps of this figure as the last step is represented by the framework itself.

4.4.2. The LLVM Intermediate Representation

Unlike Wasm, the IR does not provide many high-level abstractions. However, even though this IR is low-level, it is still target-independent, and unlike many other low-level languages, it also contains detailed type information. Therefore, high-level type information is preserved while the benefits of a low-level representation are introduced. This allows LLVM to

perform more aggressive optimizations compared to other compiler solutions or frameworks. Therefore, programs compiled using LLVM as the backend will often run significantly faster compared to those generated by simple compilers.

LLVM provides official APIs for interacting with the IR in memory. This way, IR can be created by a frontend without the need for disk operations. If a file containing the IR is written and read by the individual parts of the compiler, the same performance issues introduced by multi-pass compilation would emerge. Therefore, LLVM provides these APIs for the *C++* and *C* programming languages. However, there are also many unofficial bindings for other languages, such as for Rust, Go, and Python. Since LLVM must be able to perform complex program analysis before it can optimize a program, its IR introduces many rules and constraints. For instance, a program formulated using the IR should always obey the following hierarchy [Hsu21, pp. 211–213]:

- The top-most hierarchical structure is the so-called *module*. It represents the current file being compiled.
- Each module contains several *functions*. Often, each function in the source program is represented using one function in the IR.
- Each function contains several *basic blocks*. A basic block contains a sequence of instructions. Blocks should always be terminated using a *jump*, *return*, or *unreachable* instruction. However, a basic block must never be terminated twice.
- Each *instruction* holds a semantic meaning and represents a part of the source program. For instance, LLVM provides instructions for integer addition or function calls.

Because the LLVM IR provides only little abstraction, numerous optimizations can be achieved in the early stages of compilation. The IR's instruction set describes a virtual architecture which is able to represent an abstraction over most of the common types of processors. Although it is low-level, the IR avoids machine specific constraints like a fixed set of registers or low-level calling conventions. Instead, the architecture provides an infinite set of virtual registers which can hold the values of primitive types, like integers, booleans, floating-point numbers, and pointers [Lat02, p. 14-17].

Structure of a Compiled rush Program

```
1 fn main() { exit(fib(10)); }
2
3 fn fib(n: int) -> int {
4     if n < 2 {
5         n
6     } else {
7         fib(n - 2) + fib(n - 1)
8     }
9 }
```

Listing 4.16 – Generating Fibonacci numbers using rush.

In order to understand how a program can be formulated using the IR, the rush program for calculating Fibonacci numbers in Listing 4.16 should be considered. The code in Listing 4.17 displays the identical rush program formulated in IR, it was generated by the rush compiler targeting LLVM[7]. The code displayed in this listing shows a module named '`main`'.

In lines 5 and 23, two functions are defined using the '`define`' keyword. It is apparent that the functions in the LLVM module represent the functions from the rush program. When examining the signature of the '`fib`' function in line 5 of the IR, it becomes apparent that the function returns a runtime value of the type '`i64`', that is, a signed 64-bit integer, the same as '`int`' in rush. The function takes a parameter named '`%0`' of the type '`i64`'. It represents the '`n`' parameter in the rush source program.

[7] Generated using rush (Git commit '`f8b9b9a`').

```
1  ; ModuleID = 'main'
2  source_filename = "main"
3  target triple = "x86_64-pc-linux-gnu"
4
5  define internal i64 @fib(i64 %0) {
6  entry:
7    %i_lt = icmp slt i64 %0, 2
8    br i1 %i_lt, label %merge, label %else
9
10 merge:                                          ; preds = %entry, %else
11   %if_res = phi i64 [ %i_sum3, %else ], [ %0, %entry ]
12   ret i64 %if_res
13
14 else:                                           ; preds = %entry
15   %i_sum = add i64 %0, -2
16   %ret_fib = call i64 @fib(i64 %i_sum)
17   %i_sum1 = add i64 %0, -1
18   %ret_fib2 = call i64 @fib(i64 %i_sum1)
19   %i_sum3 = add i64 %ret_fib2, %ret_fib
20   br label %merge
21 }
22
23 define i32 @main() {
24 entry:
25   %ret_fib = call i64 @fib(i64 10)
26   call void @exit(i64 %ret_fib)
27   unreachable
28 }
29
30 declare void @exit(i64)
```

Listing 4.17 – LLVM IR representation of the program in Listing 4.16.

In line 6, the start of the '`entry`' block of the '`fib`' function is declared using the block's name followed by a colon. Since LLVM can perform more optimizations on registers if they are declared in the '`entry`' block of a function, the rush compiler places variable declarations solely in the '`entry`' block. In line 7, the '`icmp slt`' (**i**nteger **comp**are **s**igned **l**ess **t**han) instruction is used in order to compare the runtime value of the parameter '`%0`' to '2'. The boolean result is then saved in a virtual register named '`%i_lt`'. Since LLVM's virtual registers may have arbitrary names, the rush compiler uses names which depend on the context of the translation. In line 8, the block is terminated using the '`br`' (**br**anch) instruction which jumps to the beginning of a basic block. The instruction will only jump to the '`merge`' label under the condition that the value of '`%i_lt`' is 'true'. Here, the '`merge`' and the '`else`' labels are used as operands of the '`br`' instruction. Conditional jumps in LLVM always require two jump destinations, one for the case in which the condition is 'true', and one for when it is 'false'. Due to constraints introduced by its internal optimizations, LLVM only allows jumps to target blocks contained in the same function. This '`br`' instruction presents the essential part of the if-expression in the source program. If the condition was 'true' at runtime, the instruction would jump to the '`merge`' block in line 10. What might seem odd is that there is no if-block, even though the rush frontend has previously translated the if-block into LLVM IR. However, since that block would only jump to the '`merge`' block, LLVM's optimizations removed it entirely.

In line 11, in the '`merge`' block, the '`phi`' instruction is used. These so called ϕ-nodes are necessary due to the structure of the IR. In short, a ϕ-node produces a different value depending on the basic block where control came from. Since the if-construct is an expression in rush, LLVM must determine whether the result of the '`entry`' or the '`else`' branch is to

be used as the result of the entire if-expression. As a solution to this problem, these ϕ-nodes associate a value to an origin branch. In this example, the ϕ-node yields the value of the parameter '`n`' (stored in '`%0`') if control came from the '`entry`' block. Otherwise, in case control came from the '`else`' block, the ϕ-node yields the value of the virtual register '`%i_sum3`'. However, it has not been discussed where this virtual register is declared. For this, the instructions in the '`else`' block, starting in line 15 with the '`add`' instruction, should be considered. In this case, the instruction subtracts '2' from the parameter '`%0`' and saves the result in '`%i_sum`'. Here, an addition instruction using a negative operand is used to perform the subtraction. This is done in order to create the argument value for the first recursive call to '`fib`' which is performed by the following '`call`' instruction in line 16. The return value of the function call is saved in the '`%ret_fib`' register. The same behavior is used in order to call '`fib(n - 1)`'.

Next, the '`add`' instruction in line 19 is used in order to calculate the sum of the two return values. This sum is then saved in the virtual register '`%i_sum3`'. Therefore, when this register is used in the ϕ-node in line 11, the result of the recursive calls is used as the result of the entire if-expression. In line 20, the '`br`' instruction jumps to the '`merge`' block unconditionally as there is no condition provided in the operands. After the jump to the '`merge`' block, the previously explained ϕ-node is encountered. Finally, the '`ret`' instruction in line 12 is used in order to use the result of the if-expression as the return value of the function. Since the '`main`' function does not introduce any new concepts, detailed explanation of its contents is omitted. In line 27, the '`unreachable`' instruction is used in order to state that it is never reached. This is necessary because LLVM requires that every basic block is terminated at its end. The '`exit`' function terminates the program and therefore terminates the basic block. However, LLVM does not regard '`call`' instructions as diverging and therefore disallows the call to '`exit`' as a way to terminate the basic block. Since LLVM is not aware of the fact that the '`exit`' function terminates program execution, an '`unreachable`' instruction has to be added in order to signal that a block is terminated.

It is to be mentioned that the original IR generated by the rush compiler looks slightly different because LLVM has already performed aggressive optimizations on this code. By considering this example, it became apparent that the IR represents only some source language constructs in a high-level way. For instance, function calls can be used without considering the complex rules required for a machine-dependent implementation. Here, calling and returning from a function can be implemented using very little effort. Furthermore, virtual registers allow the compiler frontend to omit the process of register management entirely.

4.4.3. The rush Compiler Using LLVM

In order to get acquainted with the LLVM framework, we have implemented a rush compiler which uses the framework as its backend. However, the entire rush project is implemented using Rust, and as mentioned previously, LLVM's official APIs only support C and C++. Therefore, a third-party Rust wrapper around LLVM is required. We have settled on using the *Inkwell* Rust crate since it exposes a safe Rust API for interacting with LLVM [Kol17].

This compiler uses the annotated AST in order to translate it into LLVM IR. Apart from its output, its principles do not differ from the other compilers presented so far. The code

crates/rush-compiler-llvm/src/compiler.rs

```
26 pub struct Compiler<'ctx, 'src> {
27     pub(crate) context: &'ctx Context,
28     pub(crate) module: Module<'ctx>,
29     pub(crate) builder: Builder<'ctx>,
       // ...
31     pub(crate) curr_fn:
   ↪   Option<Function<'ctx, 'src>>,
       // ...
47 }
```

Listing 4.18 – The '`Compiler`' struct definition of the LLVM compiler.

in Listing 4.18 displays the top part of the '`Compiler`' struct definition. The '`context`' field represents a container for all LLVM entities including modules. Next, the '`module`' field contains the underlying LLVM module. The '`builder`' field contains a struct provided by Inkwell which allows generation of IR in memory. In general, these fields provide an abstraction over LLVM's APIs which is used by this compiler. In order to get a deeper understanding of how this compiler works exactly, the rush program in Listing 4.19 should be considered. The LLVM IR generated from this input is displayed in Listing 4.20.

```
1 fn main() {
2     foo(2);
3 }
4
5 fn foo(n: int) {
6     let mut m = 3;
7     exit(n + m);
8 }
```

Listing 4.19 – Simple rush program containing two functions.

```
5  define internal i1 @foo(i64 %0) {
6  entry:
7    %i_sum = add i64 %0, 3
8    call void @exit(i64 %i_sum)
9    unreachable
10 }
11
12 declare void @exit(i64)
13
14 define i32 @main() {
15 entry:
16   %ret_foo = call i1 @foo(i64 2)
17   ret i32 0
18 }
```

Listing 4.20 – LLVM IR generated from the program in Listing 4.19.

The program in Listing 4.19 contains the '`foo`' and the '`main`' functions. The corresponding definitions are found in lines 5 and 14 of the output IR. The '`foo`' function takes the two parameters '`n`' and '`m`'. It calculates their sum in order to use it as the exit code of the program. In line 7 of the IR, the parameter '`n`' and the variable '`m`' are added together. What might seem odd is that the definition of '`m`' cannot be seen in the IR. Instead, the constant value '3' of the variable is used in the addition instruction. The result of this addition is then used for the function call of the '`exit`' function in line 8 of the IR. Because '`n`' is '2', the exit code of the program will be '5'.

At the beginning of translation, the compiler first iterates over all declared functions in order to add them to the LLVM module. Listing 4.21 displays parts of the method responsible for translating the '`main`' function.

In lines 334–336, the '`main`' function is added to the current LLVM module. The variable '`fn_name`' specifies the name of the function to be inserted, while the '`fn_type`' describes its signature. In most cases, the '`main`' function's return type is an integer so that a C standard library can use the function as the entry point. In cases where the generated code should not depend on C libraries, '`fn_name`' will be '`_start`' and '`fn_type`' will state that the function returns '`void`'. This concept is mentioned since the LLVM compiler will not always target the current architecture, meaning a C standard library is not always available. In lines 339 and 340, the '`entry`' and '`body`' basic blocks are created and added to the function. Next, in lines 343–347, the '`curr_fn`' field of the compiler is updated. This field holds information about the current function being compiled. In line 345, the '`llvm_value`' property is of particular importance since all later additions of basic blocks, e.g., during loop compilation, require this value. This property can then later be accessed in case a new basic block should be appended, even though it happens in another method. The '`entry_block`' property in line 346 is required every time a pointer is created, as will be explained later.

Using the Inkwell library, most generated instructions will be automatically appended to

crates/rush-compiler-llvm/src/compiler.rs

```
321 fn main_fn(&mut self, node: &'src AnalyzedBlock) {
        // ...
334     let fn_type = self
335         .module
336         .add_function(fn_name, fn_type, Some(Linkage::External));
337
338     // create basic blocks for the function
339     let entry_block = self.context.append_basic_block(fn_type, "entry");
340     let body_block = self.context.append_basic_block(fn_type, "body");
341
342     // set the current function to `main`
343     self.curr_fn = Some(Function {
344         name: "main",
345         llvm_value: fn_type,
346         entry_block,
347     });
348
349     // compile the body
350     self.builder.position_at_end(body_block);
351     self.block(node, true);
        // ...
369     self.builder.position_at_end(entry_block);
370     self.builder.build_unconditional_branch(body_block);
371 }
```

Listing 4.21 – LLVM: Compilation of the '`main`' function.

the end of the current basic block. For that reason, the position of the instruction builder is changed to the newly created '`body`' block, so that the following call to '`block`' in line 351 adds instructions to it. The '`block`' method first creates a new scope and then compiles all the block's statements and its optional trailing expression. If the contents of the '`main`' function's body do not lead to the insertion of more basic blocks, the '`body`' block will contain the entire body of the function after the method call.

crates/rush-compiler-llvm/src/compiler.rs

```
916 fn call_expr(&mut self, node: &'src AnalyzedCallExpr) -> BasicValueEnum<'ctx> {
        // ...
951     let res = self
952         .builder
953         .build_call(func, &args, &format!("ret_{}", node.func))
954         .try_as_basic_value();
        // ...
957 }
```

Listing 4.22 – LLVM: Compilation of call-expressions.

In line 2 of the example rush program, the '`main`' function calls the `foo` function using the argument value '2'. In order to understand how this compiler translates function calls, Listing 4.22 should now be considered. It displays the '`call_expr`' method. The statement in lines 951–954 inserts a LLVM '`call`' instruction. For this, the '`build_call`' method of the builder is called using the target function, call arguments, and the name of the result register. The value of '`func`' represents the called function, which was previously obtained by looking up the function name in the module. The '`args`' variable contains a vector of Inkwell's '`BasicMetadataValueEnum`' and therefore specifies the type of each argument. This variable was defined previously by compiling each expression in the '`node.args`' vector.

However, the representation of expression values cannot be understood without considering Listing 4.23, which shows the '`expression`' method. When considering the method's

crates/rush-compiler-llvm/src/compiler.rs

```
873 fn expression(&mut self, node: &'src AnalyzedExpression) -> BasicValueEnum<'ctx> {
874     match node {
875         AnalyzedExpression::Int(value) => self
876             .context
877             .i64_type()
878             .const_int(*value as u64, true)
879             .as_basic_value_enum(),
          // ...
910         AnalyzedExpression::Infix(node) => self.infix_expr(node),
911     }
912 }
```

Listing 4.23 – LLVM: Compilation of expressions.

signature, it becomes apparent that it uses an '`AnalyzedExpression`' in order to generate a '`BasicValueEnum`'. This type represents a virtual register which will contain the expression's result at runtime. When using Inkwell, most inserted instructions yield such a value. The lines 875–879 show how an integer literal is compiled. Here, a constant integer value of the '`i64`' type is created and transformed into a '`BasicValueEnum`' which is then returned. For more complex expressions, the '`expression`' method invokes other methods which are specialized on certain types of expressions. For instance, if an infix-expression is compiled, this method calls the '`infix_expr`' method in line 910. Here, the current AST node is passed to the specialized method as a call argument.

crates/rush-compiler-llvm/src/compiler.rs

```
960 fn infix_helper(/* ... */) -> BasicValueEnum<'ctx> {
967     match lhs_type {
          // ...
999         Type::Int(0) | Type::Bool(0) | Type::Char(0) => {
1000            let lhs = lhs.into_int_value();
1001            let rhs = rhs.into_int_value();
1002            match op {
                    // ...
1021                InfixOp::Mul => self.builder.build_int_mul(lhs, rhs, "i_prod"),
1022                InfixOp::Div => self.builder.build_int_signed_div(lhs, rhs,
    ↪  "i_prod"),
1023                InfixOp::Rem => self.builder.build_int_signed_rem(lhs, rhs,
    ↪  "i_rem"),
1024                InfixOp::Pow => self.__rush_internal_pow(lhs, rhs),
                    // ...
1048            }
1049            .as_basic_value_enum()
1050        }
          // ...
1059    }
1060 }
```

Listing 4.24 – LLVM: Compilation of integer infix-expressions.

Listing 4.24 shows a part of this '`infix_helper`' method, which is responsible for compiling parts of infix-expressions. Line 1021 contains the code for inserting the '`mul`' instruction. Here, the variables '`lhs`' and '`rhs`' are used as arguments for the '`build_int_mul`' method call. They, too, represent virtual registers which will contain the values of the left- and right-hand sides at runtime. Furthermore, the string containing '`i_prod`' specifies the name of the virtual register that will contain the resulting product. Compiling basic integer multiplication has proven to be really simple since only one instruction needs to be inserted. This simplicity applies to most infix operations performed on integers. However, compiling mathematical

powers has proven to be more demanding since LLVM does not provide an instruction for performing these operations. Line 1024 is executed if the method needs to compile such an integer power operation. In order to mitigate this issue, the '`__rush_internal_pow`' method is called instead of a method provided by Inkwell. This method first declares the '`core::pow`' function which implements an algorithm for calculating powers given an integer base and exponent. It was manually written in IR by using the appropriate calls to the Inkwell builder. This way, even complex calculations can be implemented, even though LLVM does not provide an existing solution. Although it might seem odd to implement IR functions manually, this approach is used to maintain target independence.

crates/rush-compiler-llvm/src/compiler.rs

```
609 fn let_stmt(&mut self, node: &'src AnalyzedLetStmt) {
610     let rhs = self.expression(&node.expr);
611
612     // if the variable is mutable, a pointer allocation is required
613     match node.mutable {
614         true => {
615             // allocate a pointer for the value
616             let ptr = self.alloc_ptr(node.name, rhs.get_type());
617
618             // store the rhs value in the pointer
619             self.builder.build_store(ptr, rhs);
620
621             // insert the pointer into the current scope (for later reference)
622             let var = Variable::new_mut(ptr, node.expr.result_type());
623             self.scope_mut().insert(node.name, var);
624         }
            // ...
629     };
630 }
```

Listing 4.25 – LLVM: Compilation of let-statements.

In line 6 of the rush program, a let-statement is used to define the mutable variable '`m`' with the initial value '3'. For demonstration purposes, this variable is marked as mutable so that the compiler uses stack memory for it. In order to understand how the compiler translates let-statements, Listing 4.25 should be considered. It shows the '`let_stmt`' method of this compiler. In line 610, the initializing expression of the statement is compiled. The '`rhs`' variable then specifies the virtual register which contains the result of this expression. The code in lines 614–624 is only executed if the variable was declared as mutable. Otherwise, the variable would never change and LLVM would therefore optimize it. Therefore, in order to present relevant code in this example, the '`m`' variable in the source program had to be declared as mutable. In line 616, the '`alloc_ptr`' method is used in order to create a new Inkwell pointer value that is declared in the '`entry`' block. The first argument of the call specifies the name of the pointer to be used, so that is equal to the name of the variable. The second argument passes the type of the initializer expression to the method. In line 619, the expression's result is then stored in the newly created pointer. Since pointers present a way to use stack memory, also non-pointer variables in the source program are internally compiled to pointers. Finally, in lines 622–623, the newly defined variable is inserted into the current scope of the compiler. Every variable inside the scope saves its Inkwell value and its type since these properties are required when the variable is used later.

4.4.4. Final Code Generation: The Linker

After LLVM has compiled a program, it emits an *object file* representing the compiled source program. Object files contain the binary machine code output of a compiler or an assembler. In the case of LLVM, they contain the target-specific machine code generated from the intermediate representation. There are many different formats for representing object files, such as *ELF* on Unix-like systems. However, object files are usually still *relocatable*[8] and not directly executable. A *linker* is used to create and executable program from object files.

A linker is a program which takes one or more object files in order to produce a single executable file. Figure 4.3 displays a simplified overview of this process. During *linking*, a linker often performs numerous tasks, such as *relocation* and *symbol resolution*. For instance, a linker might also include library code in the executable if the object file depends on external functionality provided by that library. A common example for this library code is a C standard library. In order to combine these modules, an essential part of the liker's actions is presented by relocation and code modification [Lev00, pp. 1-15]. During relocation, the linker assigns definitive addresses to numerous parts of the program. Relocation is required, for instance, when the program to be translated consists of multiple modules referencing each other. Here, the order in which the individual parts of the program will be placed in memory is not known. Therefore, any absolute addresses in the program are not determined [Zhi17, p. 74]. However, these concepts are not explained further since they are not of particular relevance for understanding the purpose of a linker.

The shell command in Listing 4.26 presents an example liker invocation. In this example, the LLVM compiler has generated an object file named '`input.o`'. The flag '`-dynamic-linker`' specifies which dynamic linker to use. Next, some library files in the directory '`/usr/lib/`' are included by using the '`-lc`' flag. These files belong to an implementation of the C standard library and are required so that the '`exit`' function works properly. Next, the '`input.o`' file is specified so that the linker includes it. Lastly, the '`-o`' flag is used to specify the output file. This way, the shell command would generate an executable file named '`output`' from an object file named '`input.o`'. A linker often presents the final step of translating a source program into an executable file which the computer can execute. However, the linker is completely independent of the previous stages of compilation and is therefore not displayed in the figures. Even though the linker program *LD* is used in this example, the choice of the linker is completely irrelevant as long as the linker supports the required input and output formats.

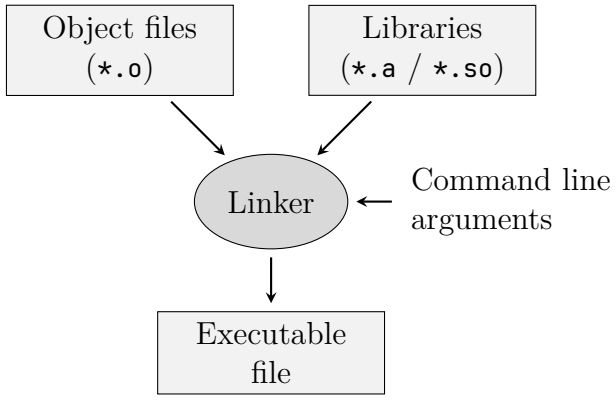


Figure 4.3 – The linking process [Lev00, p. 7].

```
ld -dynamic-linker /lib/ld-linux-x86-64.so.2 \
    -lc /usr/lib/crt1.o \
    -lc /usr/lib/crti.o \
    -lc /usr/lib/crtn.o \
    input.o \
    -o output
```

Listing 4.26 – Invoking LD to link an LLVM output.

[8] Load addresses of position-dependent code may still be changed.

4.4.5. Conclusions

To conclude, implementing a compiler which leverages LLVM presents a lot of advantages, like support for many target architectures and aggressive optimizations. Most of the demanding work is being handled by LLVM, therefore implementing the compiler will prove to be less difficult and error-prone. LLVM presents a robust, production-ready, and scalable backend which is used by very popular compilers. Finally, in order to give an example for how LLVM's optimizations can positively impact performance at runtime, the Fibonacci benchmark should be considered again. In this benchmark, the 42$^{\text{nd}}$ Fibonacci number is calculated using the algorithm displayed in Listing 4.16 on page 48. Running the program compiled using the rush LLVM compiler took around 1.3 seconds. However, executing the binary generated using the rush x86_64 compiler took around 2.17 seconds[9]. This means that the program compiled using LLVM ran roughly 1.7 times faster.

[9] Average from 100 iterations. OS: Arch Linux, CPU: Ryzen 5 1500, RAM: 16 GB.

4.5. Transpilers

One more possible target for compilers is another high-level programming language. A compiler translating one language to another high-level language is usually called a *transpiler*. Historically, the very first version of the C++ compiler was a transpiler which translated C++ into C. The main advantage of a transpiler is that its implementation is usually very simple, as low-level compilation steps can be omitted. A significant downside of a transpiler is that it often generates very inefficient code. Furthermore, the entire project would also depend on the target language [Jef21, p. 5].

For this paper, we have implemented a transpiler which generates C code from rush source programs. Listing 4.27 shows a rush program which terminates with the code '42'. For this, the value of the block-expression in lines 2–5 is saved in the variable '`num`'. In the block, a temporary variable with a value of '40' is created. The block results in a value of '42' since '2' is added to this variable. In line 6, the '`exit`' function is called, using '`num`' as its argument. Listing 4.28 shows the identical program represented using C. The code was generated from the rush program in Listing 4.27 using the rush C transpiler[10].

```
1 fn main() {
2     let num = {
3         let b = 40;
4         b + 2
5     };
6     exit(num);
7 }
```

Listing 4.27 – A rush program containing a block-expression.

```
1  #include <stdlib.h>
2  #include <stdbool.h>
3
4  int main();
5
6  int main() {
7      long long int b0 = 40;
8      long long int num1 = b0 + 2;
9      exit(num1);
10 }
```

Listing 4.28 – Transpiler output of the rush program in Listing 4.27.

When examining the C output, it becomes apparent that the structure of the code has changed slightly. In lines 1 and 2, the file contains '`#include`' directives in order to import certain functions and types from a C standard library. In line 4, the '`main`' function is declared, but not yet defined. The transpiler inserts function declarations at the start in order to allow rush's arbitrary order of function definitions. Furthermore, the C program also does not include the aforementioned block because C does not have a structure comparable to rush's block-expressions. Lines 2–5 in the rush code are represented by lines 7–8 in the C code. This examples demonstrates how a transpiler often alters the program's structure in order to represent the source program in another language with different semantic rules. Since transpilers are often not the best way of implementing a compiler, the internals of this transpiler will not be discussed further. To conclude, transpilers can present an easy but often impractical way of implementing a compiler. Due to its disadvantages, a transpiler is often only considered during the prototype phase of compiler implementation [Jef21, p. 5]. It is to be mentioned that some successful and popular transpilers exist. For instance, the popular programming language *Typescript* uses a transpiler to generate Javascript [Che19, Chapter 2].

[10] Generated using rush (Git commit '`f8b9b9a`').

5. Compiling to Low-Level Targets

In the previous chapter, compilation to high-level targets has been presented. The previously shown compilers generated outputs which were still platform independent and portable. However, compilers can also target a specific physical computer architecture directly, thus removing another layer of abstraction. Implementing compilers targeting the architectures presented in this chapter has proven to be a lot more demanding. Reasons for that mostly include target-specific constraints which were still irrelevant in the previous chapter. The term "low-level" is used to describe such target-specific outputs here.

5.1. Low-Level Programming Concepts

Programming using high-level languages does not require knowledge about the target architecture of the program. In this chapter, two compilers targeting low-level assembly are presented. In order to make the sections in which these compilers are explained more approachable, some of the most important low-level programming concepts are explained here.

5.1.1. Sections of an ELF File

Since a program needs to be representable in binary, special formats for organization are required. *ELF* stands for "executable and linkable format" and is often found on Unix-like systems including Linux. Programs using the ELF format can be represented by three different types of files. Firstly, object files generated by a compiler, like in the previous LLVM section, might use the ELF format. Secondly, dynamic libraries using *shared object files* might also leverage the ELF format. Thirdly, executable programs use the format in order to represent a structured container for instructions, data, and additional information. This way, the unit is mostly self-contained and can be executed by the operating system easily. Therefore, ELF describes the format of a class of files and not just of an individual type of file [Zhi17, pp. 74–76].

Even though a processor only has access to one physical memory unit for both instructions and program data, most assembly programs separate these types of memory into their separate components. Therefore, object files and assembly programs are divided into so-called *sections* [Zhi17, p. 19]. Some of the important ELF sections are listed below [Zhi17, p. 76].

- '`.text`' stores the logic of the program represented using CPU instructions.
- '`.rodata`' stores read-only global data, and is often used for global constants.
- '`.data`' stores mutable global data, such as mutable global variables.

5.1.2. Assemblers and Assembly Language

Assembly language describes a type of low-level programming language which is directly influenced by the target architecture. Since the assembly code provides a slight abstraction over the computer's hardware, it must be translated to machine code before it can be executed. This process is performed by a program called *assembler* and is referred to as the *assembly process*.

For instance, the RISC-V assembly instruction '`addi a0, a0, 2`' would be used for integer addition. Often, the name of the instructions is a mnemonic roughly describing its functionality. In the case of RISC-V, '`addi`' stands for "**a**dd **i**mmediate". Lastly, the instruction for adding integers differs for most CPU architectures. For instance, an equivalent instruction for the x86_64 architecture is '`add %rdi, 2`'.

As described earlier, the application code in form of instructions is placed in the '`.text`' section of an ELF binary. In most assembly dialects, the programmer is able to partition the code manually using sections. If the assembly code is assembled to an ELF object file, the '`.text`' section should contain all the instructions. Just like parts of the LLVM IR, the '`.text`' section of an assembly program can be partitioned into blocks. This section contains many *labels* which mark the beginning of a new block of instructions. In assembly, labels rather serve as targets for jump instructions. However, these blocks do not come with all the constraints which are to be followed when using LLVM IR. For instance, a block in assembly might even contain no instructions, does not have to be terminated and could even be terminated twice. Here, terminating instructions refer to jump and return instructions. Therefore, the constraints in assembly are much weaker than the ones introduced by LLVM.

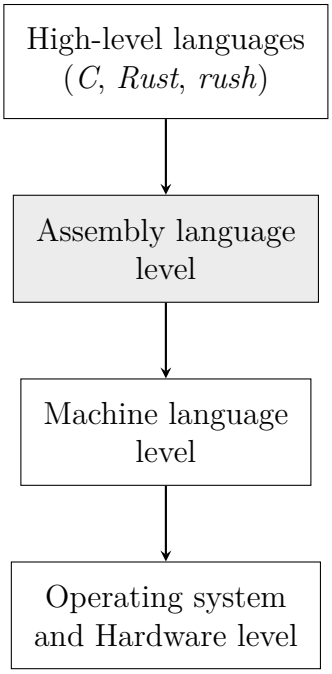


Figure 5.1 – Level of abstraction provided by assembly [Dan05b, p. 5–6].

In order to understand how much abstraction is provided by assembly, Figure 5.1 should be considered. Here, the highest level of abstraction is provided by high-level programming languages like C, Rust, and rush. The next lower level of abstraction is represented by assembly languages. As of today, one only rarely encounters a programmer writing programs using this level of abstraction or less. Here, the program is no longer independent of a target architecture and consists of significantly more code. However, the next lower level of abstraction below assembly is machine language. Since the machine language program is represented in binary, it is very time-consuming for a human to write or understand. However, the machine language is also just an abstraction of the computer's hardware and operating system. Assembly provides enough abstraction to be comprehensible for a human while being a low-level representation of the program. Some benefits of using assembly languages are increased runtime efficiency and decreased code size, while maintaining fine-grained control over the hardware and operating system. However, achieving these results is often very demanding and time-consuming. Due to the aforementioned reasons, it is reasonable for a compiler to generate assembly code from the source program.

One might argue that the compiler could output object files directly. However, doing so rarely creates any significant benefits other than the omitted dependence on the assembler. Furthermore, implementing a compiler using this approach significantly increases the complexity of the compiler since it now has to perform the tasks of the assembler as well.

5.1.3. Registers

Most processors contain numerous registers in order to hold data, instructions, and state information. Unlike external memory, they are very limited in both size and count, and must therefore be used carefully. Since they are a physical part of the CPU, they are much faster to access than the external memory. Due to these traits, registers are very suitable for

storing temporary values in larger computation [Wat17, pp. 212–214]. Figure 5.2 shows a simplified layout containing a CPU, registers, and memory. Here, the CPU and its registers are connected via a thick arrow, indicating a high throughput connection. However, the CPU and the memory are connected via a thin arrow which indicates a slower connection. Nearly every CPU architecture will define an individual register layout, meaning that sizes, count, and types of its registers may vary. As a general rule, architectures with many registers usually outperform the ones that provide fewer registers. However, as a result of this, the CPU will be more expensive to manufacture. Since not every register is identical, there are platforms which distinguish between general-purpose, floating-point, and special-purpose registers. For the majority of architectures, a subset of the registers is additionally designated for use in specific operations like *procedure calls* [Dan05b, Chapter 2].

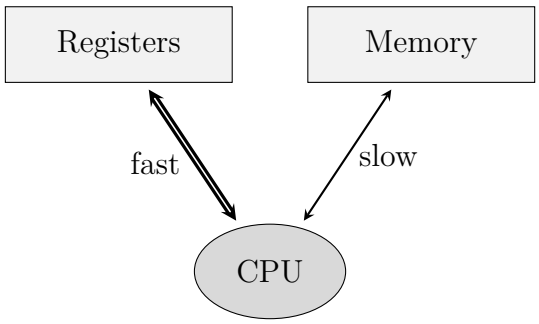


Figure 5.2 – Relationship between registers, memory, and the CPU [Dan05b, pp. 20–21].

For optimizing compilers, the limited number of registers presents a challenge as performance of the output program shall be maximized. Some architectures may require that the operands of arithmetic or logical instructions have been explicitly loaded into registers beforehand. In this case, registers would be required in nearly all computations. In order to understand how register management can present a challenge, Listing 5.1 should be considered. It displays RISC-V assembly instructions which calculate the sum of the two integers '40' and '2'. In line 4, the integer value '40' is placed in the register '`a0`'. Next, in line 5, the integer '2' is placed in '`a1`'. In line 6, the '`add`' instruction is used to calculate the sum of these two integers. Since the first operand of the instruction specifies the register in which the result should be placed, the original value of '40' in the register '`a0`' would be overwritten by the instruction [PW17, Reference Card]. Through this example, it becomes apparent that instructions might use registers in a way that would interfere with other operations. Therefore, subtle bugs can emerge as soon as an instruction unintentionally overwrites a register.

```
4 li a0, 40
5 li a1, 2
6 add a0, a0, a1
```

Listing 5.1 – Example assembly program for explaining register allocation.

In compilers, the process of *register allocation* is responsible for managing how registers are used. This process attempts to use registers in a way that leads to maximized efficiency of the output program. Since accessing registers is faster than accessing memory, a register allocator will try to use registers as often as possible. Real compilers regularly follow this rule to the point at which normal variables are also kept in registers. It is apparent that in most programs, the number of variables will certainly exceed the capacity provided by registers. Therefore, register allocation has to detect when no free registers are available anymore. In this case, the compiler has to save data in memory instead of registers. This process is called *register spilling*. Since register spilling introduces a performance penalty, register allocators typically attempt to use it as infrequently as possible. Therefore, sophisticated algorithms are mandatory if execution performance is regarded as non-trivial. Moreover, register allocation has to ensure that no conflicts between registers are introduced. Such a conflict may be that a register is accidentally overwritten by an instruction in a completely unrelated block. It is apparent that implementation of an algorithm performing all these tasks is usually very demanding. For instance, register allocation often requires complex graph algorithms [Wat17, pp.212-214]. Since register allocation presents a

complex topic, it will not be explained any further. However, there are also heavily simplified approaches to register allocation, such as the ones used by the compilers presented in later sections of this chapter.

5.1.4. Using Memory: The Stack

As registers are limited in both size and count, additional storage, for example for variables is required. For this, most architectures provide a *stack*. In general, a stack is a last-in-first-out data structure, meaning the element that was last added via a *push* operation is the first to be removed by a *pop* operation. In computer hardware, the stack is usually physically separate from the CPU and therefore much slower to access. It is typically implemented as linear memory that can be accessed and modified freely. A so-called *stack pointer*, which is normally stored in a special register, saves the address of the top-most element of the stack. Thus, the principle of this memory being a stack data structure is merely a convention. A stack typically grows towards lower addresses, so that in order to push values, the stack pointer must be decreased [PH17, pp. 68, 99, 100].

It is typical for compiled procedures to *allocate* an entire region of the stack for themselves in a procedure's *prologue* by subtracting the amount of needed bytes from the stack pointer, and freeing it in batch in a procedure's *epilogue*. These regions are called *stack frames*, and some architectures, including both RISC-V and x64, define an additional register for pointing to the other end of the current stack frame. This pointer is usually called the *frame pointer* or *base pointer* [Wal98, p. 94].

Alignment

In order to save space, one might want to save variables of different sizes, say a boolean and an integer, directly next to each other on the stack. However, most architectures require values to be *aligned* to a multiple of their size. Figure 5.3 shows three example memory layouts of one '`u8`', one '`u16`', and one '`i64`'[1], taking up one, two, and eight bytes respectively. The first layout has two of the integers misaligned, marked in dark gray, thus being invalid. The second layout preserves the same order but inserts empty white areas, called *padding*, in two places, to correct the alignment. Firstly, there is one byte just before the '`u16`' to align it to an integer multiple of two bytes, and secondly, there are four bytes before the '`i64`' to have that be aligned to eight bytes. The third layout inverts the order, which causes all three values to be correctly aligned as is.

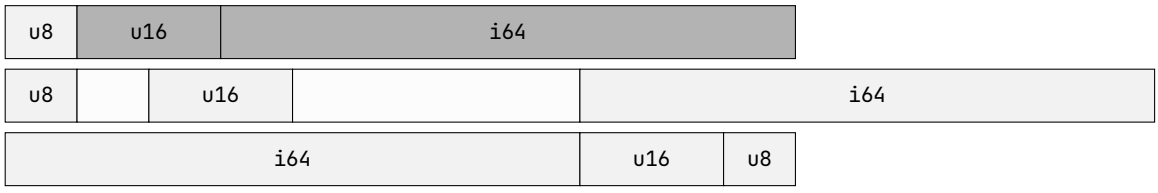


Figure 5.3 – Examples of memory alignment.

Listing 5.2 shows the implementation of the rush x64 compiler for aligning a pointer to a given size. The pointer must only be modified if it is not an integer multiple of the desired byte count yet, hence the outer if-expression. To calculate the amount of padding, the formula '`size - (ptr % size)`' is used. For instance, say '`ptr`' is '22' and '`size`' is '8'. The stack memory is then thought of in chunks of eight bytes each, where the resulting pointer should point to the start of the next empty chunk. With the pointer currently being '22',

[1] These types are defined by Rust, representing unsigned ('`u`') and signed ('`i`') integers with 8, 16, and 64 bits of storage respectively.

there are two full chunks and one chunk with six of the eight bytes occupied. Hence, two bytes of padding are needed to reach the next chunk. The expression '`22 % 8`' yields '6', which is the number of bytes that are already in use in the last chunk. This result is then subtracted from '8' again to result in '2', the number of bytes required for padding, that is then added to the pointer.

crates/rush-compiler-x86-64/src/compiler.rs

```
184 pub(crate) fn align(ptr: &mut i64, size: Size) {
185     if *ptr % size.byte_count() != 0 {
186         *ptr += size.byte_count() - *ptr % size.byte_count();
187     }
188 }
```

Listing 5.2 – Alignment of values on the stack.

5.1.5. Calling Conventions

Programmers often use *procedures* or *functions* in order to structure their programs, both to make the program easier to understand and to allow shared logic to be reused. Procedures allow the programmer to focus on one portion of the tasks at the time, thus representing one way of how a high-level programming language might provide abstraction. Often, parameters and returned values act as the interface between the procedure and the remaining code. During a procedure call in a high-level language like rush, many individual steps need to be performed in the program's low-level representation. Since assembly does not support the use of high-level functions, most architectures specify their own *calling convention* which describes how low-level function calls are to be performed. On most architectures, a low-level procedure call follows these steps [PH17, p. 98]:

1. The caller places the call arguments in a place where the procedure can access them.
2. Control is transferred to the procedure, often using a jump or specialized call instruction. This specialized instruction often saves the *return address*[2] in a special register, so that function returns are possible.
3. First, the procedure acquires local storage resources, like stack memory for its local variables. This step is usually referred to as the *prologue*.
4. Internal code of the procedure is executed by evaluating the instructions in its body.
5. The procedure's result value is placed somewhere so that the caller can access it. Here resources allocated in step 3 are also released again. This step is usually referred to as the *epilogue*.
6. Control is transferred back to the caller using a specialized return instruction that jumps back to the instruction which had initiated the procedure call.

In order to leverage optimal performance during procedure calls, passed arguments are usually kept in registers. The majority of architectures only plan for a portion of their registers to be used as call arguments. If the number of call arguments exceeds the number of available argument registers, the remaining arguments would have to be spilled to memory, meaning that they are placed on the stack so that the called function can access them [PH17, p. 98].

[2] Saves the address of the call-site, allows the called procedure to return back to the caller [PH17, p. 99].

For instance, a target architecture might plan for four registers (‘`r0`’–‘`r3`’) to be used as call arguments. Now, a function is called using six arguments. Due to the fact that there are two more arguments than registers, register spilling is now required. In that case, the registers ‘`r0`’–‘`r3`’ would contain the first four argument values while the fifth and sixth argument are spilled onto the stack.

It is to be mentioned that implementing a compiler which respects the calling convention of its target architecture is not required. However, following the convention is often reasonable in order to preserve *ABI*[3] compatibility of the compiled program.

5.1.6. Referencing Variables Using Pointers

A *pointer* is a value which contains the memory address of a variable. Leveraging pointers often leads to more efficient and compact code. Furthermore, pointers are sometimes mandatory for solving a specific problem [KR88, p. 93]. A problem which can only be solved by using pointers is displayed in Listing 5.3. This program contains the ‘`main`’ and ‘`modify`’ functions. In line 2, in the ‘`main`’ function, the mutable variable ‘`answer`’ is defined with an initial value of ‘42’. Next, the ‘`modify`’ function is called, passing the variable as the call argument. In line 4, the program exits with the value of ‘`answer`’. In the signature of the ‘`modify`’ function, the ‘`n`’ parameter is declared as mutable. Next, in line 8, the value of the parameter is incremented by ‘1’. One might expect that the function call in line 3 causes the variable ‘`answer`’ to be incremented by ‘1’. This seems likely since both the parameter and the variable are declared as mutable using the ‘`mut`’ keyword. However, since rush passes function arguments by value[4] and not by reference, the called function has no direct way of altering the variable behind the passed parameter. In order to solve this problem, pointers are required. Listing 5.4 displays a rush program solving this issue.

```
1 fn main() {
2     let mut answer = 42;
3     modify(answer);
4     exit(answer);
5 }
6
7 fn modify(mut n: int) {
8     n += 1;
9 }
```

Listing 5.3 – A rush program trying to alter the variable behind an argument.

```
1 fn main() {
2     let mut answer = 42;
3     modify(&answer);
4     exit(answer);
5 }
6
7 fn modify(n: *int) {
8     *n += 1;
9 }
```

Listing 5.4 – A rush program altering the variable behind an argument.

Most parts of this rush program look similar to the one displayed in Listing 5.3. However, some parts of the code have been adjusted so that they use a pointer. First, the signature of the ‘`modify`’ function looks slightly different. Now, the function takes a parameter of type ‘`*int`’ instead of ‘`int`’. The syntax ‘`*type`’ describes a pointer to the type specified after the ‘`*`’. Thus, the function now takes a parameter which is a pointer to an integer variable. In rush, pointers allow full read-write access to the target variable. This time, the parameter itself is not declared as mutable since the target variable behind the pointer and not the pointer itself should be incremented. In line 8, the variable referenced by the pointer is incremented. When assigning to the target variable behind a pointer, the pointer first needs to be *dereferenced*. In rush, dereferencing a pointer is accomplished using the ‘`*`’ operator before the pointer’s identifier. The process of dereferencing a pointer involves accessing the value of the variable the pointer points to [KR88, p. 94].

In rush, only pointers targeting an existing variable can be created. In order to create a pointer to a variable, rush’s ‘`&`’ prefix operator is used. This operator *references* the target

[3] Short for “application binary interface”, allows calling foreign functions from another program.
[4] Passing by value means that the value of the argument is copied for the function call.

variable in order to return a pointer value. Referencing a variable produces the memory address of that variable [KR88, p. 95]. Since the resulting value is only a normal integer, it could also be treated similarly.

However, some target architectures make the implementation of pointers more difficult. The statement in line 8 increments the value of the variable pointed to by '`n`' by '1'. This time, '`answer`' does change since only the address, which still points to the same mutable variable, was copied during the function call. Due to this write-access, the analyzer only allows the '`&`' operator to be used on mutable variables. As seen in the updated program, there is no need for the resulting pointer to be mutable since it only stores a read-only address.

A special trait of pointers in rush is that they are able to introduce *undefined behavior*[5]. However, this only occurs if the pointer is used as a return value of a function in which it references a local variable. This problem occurs because the memory used by that function is freed after the function has executed, resulting in the pointer referencing uninitialized memory. Therefore, pointers should be used with caution since the semantic analyzer cannot[6] guarantee that they are used correctly.

[5]The runtime behavior of such scenarios is undefined and might vary per architecture.

[6]If additional constraints for pointers were introduced, the analyzer could also validate their usage.

5.2. RISC-V: Compiling to a RISC Architecture

The *RISC-V ISA*[7] is a modern *reduced instruction set* architecture focussing on simplicity and expandability. The initial version was developed at *UC Berkely* in the context of another related research project. Since its introduction in 2011, the architecture has been rapidly growing in popularity while being managed and led by the RISC-V foundation. Today, corporate members of the RISC-V foundation include companies like *Google*, *Microsoft*, *Samsung*, and *IBM*. Therefore, the general popularity and commercial attraction of the technology is apparent. However, unlike most previous ISAs, the RISC-V architecture is a completely *open source* project and is therefore not controlled by a single large corporate entity. In the past, many ISAs have failed due to them being too restrictive with their licensing, thus preventing widespread commercial adoption. However, RISC-V is completely open and free to use, so that many companies can leverage the technology commercially while contributing to the project. Unlike most of the previous ISAs, which were developed during the 1970s and 80s, RISC-V is one of the few which were developed in the last decade [PW17, Preface].

5.2.1. Register Layout

Compared to popular CISC architectures, RISC architectures typically have a large number of registers [Dan05b, Chapter 2]. For instance, while the 32-bit variant of the x86 architecture has eight registers, the RISC architecture *ARM-32* provides twice that amount, that is, 16 registers. However, a RISC-V CPU includes 32 registers, only counting the general-purpose ones. Just for floating-point numbers, the ISA provides another 32 registers. Due to this difference, a register allocator targeting the RISC-V architecture could be more aggressive compared to one targeting a CISC architecture [PW17, p. 10].

Table 5.1 – Registers of the RISC-V architecture [WA19, p. 155].

Register(s)	Purpose
`zero`	hardwired zero
`ra`	return address
`sp`	stack pointer
`t0`–`t6`	temporary storage
`fp`	frame pointer
`a0`, `a1`	function arguments, return values
`a2`–`a7`	function arguments
`s1`–`s11`	saved register
`fa0`, `fa1`	float arguments, return values
`fa2`–`fa7`	float arguments
`fs0`–`fs11`	float saved registers
`ft0`–`ft11`	float temporaries

Table 5.1 shows most of the registers provided by the RISC-V architecture. For this table, the official ABI names of the registers have been used. The special '`zero`' register always holds a constant '0'. Unlike other registers, it is read-only, meaning that it can never be overwritten [PW17, pp. 18f, p. 34]. The '`ra`' register saves the *return address* of a procedure. If a return instruction is used, the value in '`ra`' is used as the address to jump back to. The '`sp`' and '`fp`' registers are used for managing stack memory. The registers '`a0`' and '`a1`' both serve as call arguments and return values of functions. The remaining '`a`*n*' registers ('`a2`'–'`a7`') should be primarily used as function call arguments. How functions are called will be explained in Section 5.2.3. The saved registers '`s1`'-'`s11`' are meant to be preserved across function calls, that is, a called function must not overwrite them. What the previously explained registers have in common is that they all hold integer values [PW17, p. 34]. Depending on the used RISC-V extensions, all registers, including the floating-point ones, either hold 32 or 64 bits of information. For this project, the *RV64I* base extension is used, meaning that each register may hold 64 bits. The float registers are able to hold floating-point numbers according to the *IEEE*

[7] Short for: "instruction set architecture", it specifies an abstract model of a CPU.

754–2008 standard [WA19, Chapter 11]. Just like their integer counterparts, the floating-point registers '`fa0`' and '`fa1`' are used as function call arguments *and* as return values, and the other '`fa`n' registers ('`fa2`'–'`fa7`') are only meant to be used as float function arguments. Just like the '`s`n' registers, the '`fs0`'–'`fs11`' registers are usually preserved across function calls. It is apparent that the floating-point registers are provisioned very similarly to the integer registers. Therefore, a compiler targeting the architecture can utilize roughly the same principles, regardless of the data-type [PW17, p. 52].

Now, it has become apparent that RISC-V includes many registers grouped into semantic categories. Every category is meant to be used in the specified manner, however, these groups are mostly only a suggestion of how each register should be used.

5.2.2. Memory Access Through the Stack

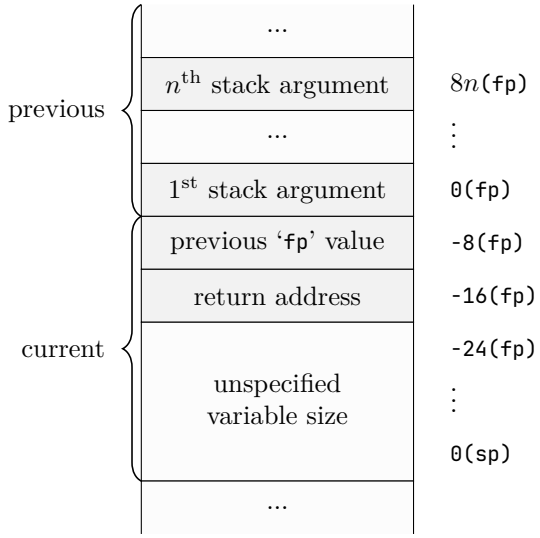


Figure 5.4 – Stack layout of the RISC-V architecture.

Figure 5.4 shows the stack layout used by RISC-V. Stack frames are enclosed by the '`fp`' and '`sp`' registers. Here, '`sp`' always points to the lowest address inside of the current stack frame while '`fp`' points to the address above the current frame. The available space for functions ranges from '`-24(fp)`' to '`0(sp)`'. It is used for storing variables, following the alignment rules that were explained in Section 5.1.4. In the space between '`0(fp)`' and '`-24(fp)`', the frame pointer of the previous stack frame and the return address are saved at the positions '`-8(fp)`' and '`-16(fp)`', respectively. If stack arguments were present, they would be placed upwards of '`fp`'. Each stack argument always uses eight bytes of memory, regardless of the actual size of its content. Thus, the n^{th} stack argument would be placed at the position 8n`(fp)`. To keep track of variable locations on the stack, the compiler maps a variable's name to a memory location. To be precise, the compiler contains a '`HashMap`' which associates a variable's name with its '`fp`' offset. When the variable is accessed at a later point, the compiler performs a simple lookup of the variable's memory address.

5.2.3. Calling Convention

Just like previously explained, most architectures provide a calling convention which dictates how low-level function calls should be managed. In the case of RISC-V, the calling convention consists of four main steps and is specified in a separate document [CCW22].

The first step of calling a function involves placing the arguments in a place where the function can access them. For RISC-V, this involves placing the arguments into certain registers. Like described in Table 5.1, only some registers are used for call arguments. For integer arguments, the first eight arguments are placed in the registers '`a0`'–'`a7`' For instance, the first two arguments of the rush function call '`foo(40, 2, 3.14)`' would be placed in the registers '`a0`' and '`a1`', respectively. However, the third argument is a floating-point number and can therefore not be placed inside an integer register. Instead, the first floating-point argument register '`fa0`' is used. In this case, all arguments can be held in registers and spilling would not be required.

In case the function accepted nine or more integer arguments, all further integer arguments

starting with the ninth one would have to be spilled onto the stack. Here, the successive registers '`a0`'–'`a7`' would contain the first eight integer arguments of the called function. However, the argument at position nine is spilled onto the stack since there are no registers left which could contain the additional argument [CCW22, p. 8].

In the second step, the underlying procedure call is made using a specialized instruction. In RISC-V assembly, one typically uses the '`call`' *pseudoinstruction*[8]. This instruction uses the name of its target label as its operand, resulting in labels being callable, just like functions. Internally, the instruction first saves the address of the following instruction in the '`ra`' register, and then performs an unconditional jump [PW17, p. 22].

The third step takes place in the function's prologue. Here, the previous frame pointer and return address are saved on the stack, and space for storing local variables is allocated by decrementing the stack pointer. Following the prologue, the function's body is executed.

In step four, after the body has also been evaluated, the epilogue restores the changes made by the prologue. The previous values for both the frame pointer and the return address registers are taken from the stack, and the stack pointer is incremented back to its original value. The '`ret`' instruction, which jumps to the address stored in '`ra`', is finally used to return control back to the caller.

In case a function returns a value, it must be made accessible to the caller. For integer types, the first return value of a function is placed in the register '`a0`' while floating-point numbers are placed in the register '`fa0`'. This way, the caller code can obtain a function's return value by accessing the '`a0`' and '`fa0`' registers respectively. If a function does not return a value, these steps are just omitted. It is to be mentioned that char and bool values are also placed inside the '`a0`' register since these types are internally represented by integers.

5.2.4. The Core Library

As outlined in the section about the linker on page 55, a program might use functionality provided by external libraries. In case of the rush RISC-V compiler, external functions are used for character-arithmetic, the mathematical power operator, and the '`exit`' system call. Character-arithmetic requires external functionality since the bounds of the ASCII range should be regarded while performing operations. For instance, in rush, calculating '`'a' + 'a'`' results in '66' because '194' $(97 + 97)$ is larger than the maximum allowed value of '127'. If the result is outside the valid range $[0; 127]$, only the least significant seven bits are considered. This way, the ASCII range is always valid.

Since these concepts must introduce additional logic, the compiler should not emit their instructions every time they are used. In that case, the repeated emission of instructions would result in enlarged and unnecessary complex output code. In order to mitigate these issues, the compiler simply inserts '`call`' instructions referencing the appropriate external function. External functions are called just like any other function. However, their definition is not found in the same assembly file. As described previously, resolving these external calls is later handled by the linker. This target-specific library code is later referred to by the term *corelib*.

For instance, the corelib includes a function for mathematical power operations. Therefore, the compiler can emit a call to this function every time rush's '`**`' operator is used in the source program. For this project, the entire corelib is written in RISC-V assembly. However, it is more reasonable to implement a corelib using a high-level language like C. Since the corelib's functions are specified in separate files, they are packaged into an *archive file* which is later used by the linker. The assembly code in Listing 5.5 shows the implementation of

[8] A macro generating one or more instructions from one pseudoinstruction. This way, the actual count of ISA instructions remains low while convenience features can be used in assembly [Dan05b, p. 68].

the ‘exit’ function in the RISC-V corelib. Its task is to invoke a specific functionality of the Linux kernel by performing a *system call*.

crates/rush-compiler-risc-v/corelib/src/exit.s

```
8  .global exit
9
10 exit:
11     li a7, 93 # syscall type is `exit`
12     ecall     # exit code is already in `a0`
```

Listing 5.5 – The assembly implementation of the ‘exit’ subroutine.

A system call (often abbreviated to *syscall*) is an invocation of a function of an operating system’s kernel. Linux implements more than 90 percent of the available system calls for all architectures [Lov13, p. 3]. One common system call is ‘exit’. It terminates the current process and performs various cleanup steps. Its only parameter is the exit code which can be checked by the shell or other programs [Lov13, p. 148]. In RISC-V Linux, the syscall identifier of ‘exit’ is ‘93’ [Tor91]. In line 8 of Listing 5.5, the ‘exit’ label is declared as global using the ‘.global’ directive. Next, the ‘exit’ label is declared in line 10. In line 11, the ‘li’ (**l**oad **i**mmediate) instruction is used in order to load ‘93’ into the register ‘a7’. On RISC-V Linux systems, the content of the ‘a7’ register specifies the type of syscall to be performed. In line 12, the ‘ecall’ instruction is then used to perform the syscall [PW17, p. 23]

5.2.5. RISC-V Assembly

Listing 5.6 shows a rush program containing two functions and a global variable. In the ‘main’ function, the variable ‘m’ is incremented by ‘1’ and the ‘foo’ function is called with ‘m’ as the argument. The ‘foo’ function then exits with the given argument. When executed, the exit code of the displayed program will be ‘43’. Listing 5.7 shows the output assembly generated from this program by the rush RISC-V compiler. The assembler code of the ‘foo’ function is intentionally omitted.

```
1  let mut m = 42;
2
3  fn main() {
4      m += 1;
5      foo(m);
6      return;
7  }
8
9  fn foo(n: int) {
10     exit(n);
11 }
```

Listing 5.6 – Example rush program containing two functions.

In line 1, the ‘.global’ assembler directive is used to declare the global symbol ‘_start’ [PW17, p. 36]. On most architectures, the ‘_start’ symbol indicates a program’s entry point, marking the first instruction to be executed [Zhi17, p. 19]. In line 6, the ‘call’ instruction is used to call the ‘main..main’ function. Here, the already familiar ‘main’ function name is prepended by the prefix ‘main..’. Since this rush compiler implements name mangling[9], every function defined in a rush program will contain this prefix.

In line 7, the ‘li’ instruction is used to load the constant integer ‘0’ into the register ‘a0’ [PW17, Reference Card]. As explained in the previous section, the register ‘a0’ is used for the first integer call argument. In line 8, the ‘exit’ function from rush’s corelib is called, using ‘0’ stored in ‘a0’ as the exit code. Therefore, the two instructions in lines 7–8 are responsible for terminating the program with the exit code ‘0’. They are always inserted at the end of the ‘_start’ label in order to terminate the program appropriately in case the rush code does not call ‘exit’ on its own[10].

[9] Compilers often *mangle* names in order to guarantee a unique name for every function [Lev00, pp. 119-120].
[10] In order to prevent runtime errors, programs should always be exited properly.

Due to the function call in line 6, the '`main..main`' label in line 10 should now be considered. In line 11, the first line of the '`main`' function, a comment indicates the beginning of the function's prologue. The '`addi`' (**a**dd **i**mmediate) instruction in line 12 adds '−16' to the value stored in '`sp`', thus allocating memory. Here, an addition instruction is used even though subtraction is required. In RISC-V, the '`addi`' instruction's operands are one destination register, one source register and one immediate value. Since this immediate value can be negative, an additional instruction for immediate subtraction would be redundant [PW17, Reference Card]. In this case, 16 bytes are allocated since two 8-byte values are stored on the stack in lines 13–14.

The comment in line 17 indicates the start of the function's body. First, the previously explained '`li`' instruction in line 18 places a constant '1' in the register '`a0`'. Next, the '`ld`' (**l**oad **d**word[11]) instruction in line 19 is used in order to load the value of the global variable '`m`' into the register '`a1`' [PW17, Reference Card]. Global variables, like '`m`', are saved under the '`.rodata`' or the '`.data`' section, depending on their mutability. Since '`m`' is declared as mutable, the compiler saves it under the '`.data`' section. The start of the '`.data`' section is marked by the '`.section`' assembler directive in line 51. Here, a label called '`m`' is defined. In this label, the '`.dword`' directive is used to define the global initializer value of the variable. This directive stores 64 bits of data [PW17, p. 39]. The initializer value of the global variable is '42' and is represented as '`0x2a`' using hexadecimal in the assembly code.

At this point, the register '`a0`' contains '1', while '`a1`' contains '42'. In line 20, the '`add`' instruction is used to save the sum of '`a0`' and '`a1`' in the register '`a2`'. Next, in line 21, the '`sd`' (**s**tore **d**word) instruction saves the value of the register '`a2`' at the memory location of the global variable '`m`', meaning that '`m`' is updated to reflect its new value '43'. It now becomes apparent that these instructions are responsible for the add-assign expression in line 4 of the rush program. Another observation is that the last operand of the '`sd`' instruction specifies the temporary register '`t6`'. The instruction uses this register for saving temporary data during the process of saving data in '`m`' [PW17, Reference Card].

```
.global _start                                  1
                                                2
.section .text                                  3
                                                4
_start:                                         5
    call main..main                             6
    li a0, 0                                    7
    call exit                                   8
                                                9
main..main:                                     10
    # begin prologue                            11
    addi sp, sp, -16                            12
    sd fp, 8(sp)                                13
    sd ra, 0(sp)                                14
    addi fp, sp, 16                             15
    # end prologue                              16
    # begin body                                17
    li a0, 1                                    18
    ld a1, m            # m                     19
    add a2, a1, a0                              20
    sd a2, m, t6                                21
    ld a0, m            # m                     22
    call main..foo                              23
    j epilogue_0        # return                24
    # end body                                  25
epilogue_0:                                     26
    ld fp, 8(sp)                                27
    ld ra, 0(sp)                                28
    addi sp, sp, 16                             29
    ret                                         30
# ...
.section .data                                  51
m:                                              52
    .dword 0x000000000000002a  # =              53
↪   42
```

Listing 5.7 – Compiler output of the rush program in Listing 5.6.

The final instruction of the '`main`' function's body is the '`j`' (**j**ump) instruction in line 24. This instruction will cause the CPU to jump to the address of the specified label, here '`epilogue_0`' would be targeted [PW17, p. 17]. The rush compiler emits the '`call`' and '`ret`' (**ret**urn) instructions for jumps between functions, while the '`j`' instruction is used for jumps between blocks of the current function. In lines 26–30, inside the epilogue, the aforementioned task of restoring register values and returning to the caller are performed.

[11] Here, "dword" indicates that 64 bits of data shall be loaded.

5.2.6. Supporting Pointers

Listing 5.8 shows a rush program in which a variable is referenced to create a pointer which is then dereferenced. In line 3, the previously created variable '`a`' is referenced in order to create the variable '`to_a`' which saves a pointer to '`a`'. Listing 5.9 includes the relevant part of the compiler output which represents this rush program[12]. In line 18, the integer value '42' is placed in the register '`a0`'. Next, '`a0`' is saved on the stack at '`-24(fp)`' using the '`sd`' instruction. As the comment suggests, the instruction in line 20 is used in order to reference '`a`'. It does so by subtracting '24' from the value stored in the '`fp`' register. This results in the absolute address of '`a`' at runtime because the syntax '`-24(fp)`' describes the address stored in '`fp`' offset by '−24'. The calculation has to be performed at runtime since the value of '`fp`' cannot be determined at compile time. In line 21, the '`a0`' register, which contains this memory address, is also saved on the stack. Therefore, the '`a`' variable is saved at '`-24(fp)`' and '`to_a`' is saved at '`-32(fp)`'.

```
1 fn main() {
2     let mut a = 42;
3     let to_a = &a;
4     exit(*to_a);
5 }
```

Listing 5.8 – Example rush program containing a pointer.

```
10 main..main:
       # ...
18     li a0, 42
19     sd a0, -24(fp)     # let a = a0
20     addi a0, fp, -24   # &a
21     sd a0, -32(fp)     # let to_a = a0
22     ld a0, -32(fp)     # to_a
23     ld a0, 0(a0)       # deref
24     call exit
```

Listing 5.9 – Compiler output of the rush program in Listing 5.8.

In order to access the value of '`a`', '`to_a`' is dereferenced in line 4 of the rush program. For this, the memory address stored in the variable '`to_a`' first needs to be loaded from the stack, which is accomplished by the '`ld`' instruction in line 22. Next, in line 23, another '`ld`' instruction is used in order to load the value of the variable at the address stored in '`a0`'. Here, '`0(a0)`' is used instead of 'n`(fp)`' since the address stored in '`a0`' offset by '0' shall be used. Because '`a0`' contains the memory address of the variable '`a`', the instruction loads '42' into the '`a0`' register.

5.2.7. Implementation

Just like the other compilers presented in this paper, this one also traverses the annotated AST using post-order. Unlike the LLVM or Wasm compilers, this one emits RISC-V assembly which can then be assembled by an assembler supporting RISC-V.

Struct Fields

Before any complex code samples can be considered, important struct fields of the compiler need to be explained. Listing 5.10 shows the '`Compiler`' struct definition containing those fields.

The '`blocks`' field in line 15 holds a vector containing values of the type '`Block`'. That type provides an abstraction for representing a label followed by instructions. The Listing 5.11 shows parts the '`Instruction`' enumeration declaration. Although the listing only shows a few of the implemented instructions, this enumeration contains all instructions which the compiler might need at a later point. Depending on the type of instruction, the corresponding

[12] Generated using rush (Git commit '`f8b9b9a`').

crates/rush-compiler-risc-v/src/compiler.rs

```
11 pub struct Compiler<'tree> {
       // ...
15     pub(crate) blocks: Vec<Block<'tree>>,
       // ...
19     pub(crate) curr_block: usize,
20     /// Data section for storing mutabable global variables.
21     pub(crate) data_section: Vec<DataObj>,
22     /// Data section for storing float constants and immutable globals.
23     pub(crate) rodata_section: Vec<DataObj>,
24     /// Holds metadata about the current function
25     pub(crate) curr_fn: Option<Function>,
26     /// Holds metadata about the current loop(s)
27     pub(crate) loops: Vec<Loop>,
28     /// The last element is the current scope.
29     pub(crate) scopes: Vec<HashMap<&'tree str, Variable>>,
30     /// Holds the global variables of the program.
31     pub(crate) globals: HashMap<&'tree str, Variable>,
32     /// Specifies all currentlu used registers.
33     pub(crate) used_registers: Vec<(Register, Size)>,
34 }
```

Listing 5.10 – The '`Compiler`' struct definition of the RISC-V compiler.

crates/rush-compiler-risc-v/src/instruction.rs

```
57 pub enum Instruction {
       // ...
69     Seqz(IntRegister, IntRegister),
70     Li(IntRegister, i64),
71     La(IntRegister, Rc<str>),
       // ...
110 }
```

Listing 5.11 – RISC-V: The '`Instruction`' enumeration.

variant includes fields which define its operands. For instance, the '`Li`' variant in line 70 represents the '`li`' instruction in assembly. Since this instruction loads the integer value specified by the second operand into the register specified by the first operand, it contains a field for a value of the type '`IntRegister`' and a field for a 64-bit signed integer. Listing 5.12 shows a part of the '`IntRegister`' enumeration. It specifies all registers which RISC-V provides, assuming that the required extensions are considered. An equivalent enumeration called '`FloatRegister`' is defined for the float registers of this RISC-V configuration.

crates/rush-compiler-risc-v/src/register.rs

```
98  pub enum IntRegister {
        // ...
130     A2,
131     A3,
132     A4,
        // ...
136 }
```

Listing 5.12 – RISC-V: The '`IntRegister`' enumeration.

The '`curr_block`' field, defined in line 19 of Listing 5.10, saves the index of the block which is currently being inserted into. Next, the '`data_section`' and '`rodata_section`' fields are declared in lines 20–23. They represent the ELF sections '`.data`' and '`.rodata`', respectively. Here, each vector might contain global variables, which are represented by the type '`DataObj`'. Each '`DataObj`' consists of a label, saved as a string, and the initial data. The data itself

is represented by another enumeration holding either a 64-bit float, a 64-bit integer, or an 8-bit integer.

In line 25, the '`curr_fn`' field is declared. It saves a value of the type '`Function`', which contains a counter for the bytes of stack space allocated by the current function. Furthermore, it contains the label of the current function's epilogue block. The former is incremented every time a variable declaration is encountered. This is required in order to allocate the correct amount of stack memory during a function's prologue.

Just like the other compilers, this one also features a '`loops`' field which saves important labels of the current loop being compiled. It stores a vector of '`Loop`' structs in order to support nested loops. Each '`Loop`' struct saves the label of the loop's head and the label of the block which follows after the loop's body.

The '`scopes`' field stores a list of nested scopes, the last element being the current one. Each scope is represented by a '`HashMap`' associating variable names with important metadata. This metadata includes the variable's type and its memory offset relative to '`fp`'. However, these '`HashMap`'s are not used for saving global variables. Instead, the '`globals`' field in line 31 is used. Just like the maps in the '`scopes`' field, this map also associates a variable's name to a value of the type '`Variable`'. This time however, each variable contains the data label under which the global variable was declared. Lastly, the '`used_registers`' field is declared in line 33. It plays a vital role in the compiler's register allocation algorithm.

Data Flow and Register Allocation

The LLVM compiler represents runtime values by passing virtual registers, represented by Inkwell's '`BasicValueEnum`', internally. Similarly, this compiler also passes registers in order to represent the data flow of the program. Unlike in LLVM, there is only a finite amount of registers available. Therefore, this compiler also has to manage register allocation so that programs can be represented using this limited number of registers. In order to understand the implementation, Listing 5.13 and Listing 5.14 are to be considered. The former contains a rush program which adds two integer variables together and uses the result as its exit code. The latter shows a part of the assembly output representing the logic in the '`main`' function.

```
1 fn main() {
2     let a = 1;
3     let b = 2;
4     exit(a + b);
5 }
```

Listing 5.13 – A rush program calculating the sum of two integers.

```
18 li a0, 1
19 sd a0, -24(fp)    # let a = a0
20 li a0, 2
21 sd a0, -32(fp)    # let b = a0
22 ld a0, -24(fp)    # a
23 ld a1, -32(fp)    # b
24 add a0, a0, a1
25 call exit
```

Listing 5.14 – Compiler output of the rush program in Listing 5.13.

In lines 2–3 of the rush program, the integer variables '`a`' and '`b`' are defined. In line 4, the '`exit`' function is invoked, using their sum as the call argument. In line 22 of the assembly code, the runtime value of the variable '`a`' is loaded into the register '`a0`'. Next, the same operation is performed for '`b`'. However, this time, the instruction writes its result into the '`a1`' register instead of the '`a0`' register. This is because the previously loaded value in '`a0`' should not be overwritten. Unlike the register allocation algorithm of a production-ready compiler, like LLVM, this one does not emphasize the performance of the output program. All variables are saved on the stack in order to keep as many registers as possible free for temporary use in calculations and operations like this one. Since this algorithm only

performs rudimentary register allocation, its implementation is also significantly simpler.

crates/rush-compiler-risc-v/src/utils.rs

```
175 pub(crate) fn get_int_reg(&self) -> IntRegister {
176     for reg in INT_REGISTERS {
177         if !self
178             .used_registers
179             .iter()
180             .any(|(register, _)| register == &Register::Int(*reg))
181         {
182             return *reg;
183         }
184     }
185     panic!("out of integer registers")
186 }
```

Listing 5.15 – RISC-V: The '`get_int_reg`' method.

Listing 5.15 shows the method called '`get_int_reg`' which is called every time a register is required by the compiler. The method returns the first available register from the register pool that contains either integer or float registers. Each register pool contains all possible registers for either integer or floating-point numbers. In line 175, in the method's signature, one can see that it returns a value of type '`IntRegister`'. The for-loop in line 176 is used to iterate over the entire '`INT_REGISTERS`' array. This array represents the pool containing integer registers. For each register in that pool, the if-expression in lines 177–180 checks whether the current register '`reg`' is contained in the '`used_registers`' vector. If this is not the case, the current register is still unused and can therefore be returned for usage. Since the '`return`' statement causes the function to return immediately, the loop only iterates until a free register has been found.

Due to the fact that the register allocator only uses as few registers as required, the compiler should not run out of registers, therefore making the '`panic!`' macro call in line 185 unreachable. However, one can see that this method does not mark the returned register as used. This is because doing so is only required in some places, where overwriting a register would introduce a bug in the output program. Therefore, if this method was called repeatedly, it would always yield the same register. In order to mark a register as used, it is appended to the '`used_registers`' vector. For this, the '`use_reg`' method, which only performs this one operation, is implemented. When a register is no longer used, and can be overwritten again, it must be removed from the '`used_registers`' vector. For this, another simple method, called '`release_reg`', is implemented.

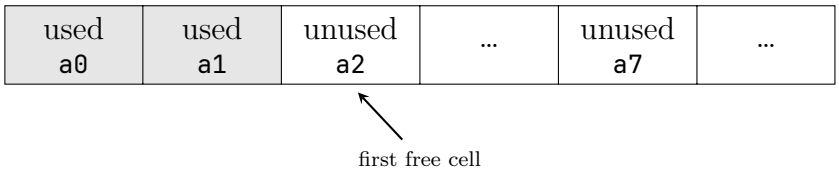


Figure 5.5 – Simplified integer register pool of the RISC-V rush compiler.

Figure 5.5 shows an abstract representation of the compiler's integer register pool. Only the first section of the pool, which contains the '`a`*n*' registers, can be seen. The gray cells specify registers that are currently in use by the compiler. The arrow points to '`a2`', which is the first free register, meaning the one which follows '`a1`', the last used register. Therefore, if the '`get_int_reg`' method was called when the state of the registers is equivalent to the one displayed in Figure 5.5, it would return the register '`a2`'.

In order to understand how expressions are compiled, Listing 5.16, which shows parts of the '`expression`' method, is to be considered. This method is responsible for compiling

crates/rush-compiler-risc-v/src/compiler.rs

```
590 pub(crate) fn expression(&mut self, node: AnalyzedExpression<'tree>) ->
    ↪  Option<Register> {
591     match node {
592         AnalyzedExpression::Int(value) => {
593             let dest_reg = self.get_int_reg();
594             self.insert(Instruction::Li(dest_reg, value));
595             Some(Register::Int(dest_reg))
596         }
            // ...
656     }
657 }
```

Listing 5.16 – RISC-V: Parts of the '`expression`' method.

expressions. As the method's signature suggests, it takes an analyzed expression and returns an '`Option<Register>`', which is '`None`' if the expression results in the '`!`' or '`()`' types. Otherwise, the method returns the register in which the result of its computation will be contained at runtime. The code in lines 593–596 is executed if the expression is an integer literal. First, a target register is retrieved by calling the previously shown '`get_int_reg`' method. Now, the '`dest_reg`' variable holds the register which will contain the result of the expression. Next, an '`li`' instruction is inserted which loads the integer contained in the variable '`value`' into the register specified by '`dest_reg`'. Finally, this register is then returned. For more complex expressions, the specified methods are called, just like in the other compilers.

crates/rush-compiler-risc-v/src/compiler.rs

```
755 fn infix_expr(&mut self, node: AnalyzedInfixExpr<'tree>) -> Option<Register> {
        // ...
791     match (node.lhs, node.rhs, node.op) {
            // ...
841         (lhs, rhs, op) => {
842             let lhs_type = lhs.result_type();
843
844             // mark LHS register as used and compile RHS
845             let lhs_reg = self.expression(lhs)?;
846             self.use_reg(lhs_reg, Size::from(lhs_type));
847
848             let rhs_reg = self.expression(rhs)?;
849
850             // set LHS register as unused
851             self.release_reg(lhs_reg);
852
853             let res = self.infix_helper(lhs_reg, rhs_reg, op, lhs_type);
854
855             Some(res)
856         }
857     }
858 }
```

Listing 5.17 – RISC-V: Parts of the '`infix_expr`' method.

However, it is still not apparent *when* the compiler marks certain registers as used. For this, the compilation of the rush expression '`n+m`' is to be considered. Listing 5.17 shows a part of the '`infix_expr`' method of the rush compiler. It consumes an annotated tree-node representing an analyzed infix-expression. In line 845, the left-hand side expression is compiled, and its result register is saved in the variable '`lhs_reg`'. Next, in line 846, the previously discussed '`use_reg`' method is used in order to mark the register specified by '`lhs_reg`' as used. The second argument to the method specifies the size of the data which

the register holds, 64 bits for '`int`', 8 bits for '`char`' and '`bool`'. This information is also saved in the '`used_registers`' vector and is accessed in case used registers need to be spilled.

In line 848, the right-hand side expression is compiled and its result register is saved in the variable '`rhs_reg`'. After this, '`lhs_reg`' is marked as unused again. Finally, in line 853, the '`infix_helper`' method is called in order to insert the instructions for the actual calculation. As discussed previously, calling the '`get_int_reg`' or '`get_float_reg`' methods repeatedly, without marking registers as used, results in the same register every time. In this scenario, this would create an issue since both the left-hand side and the right-hand side expressions would use the identical register. In order to mitigate this issue, the register of the left-hand side is marked as used before compilation of the right-hand side. Now, the right-hand side will respect that the register returned by the left-hand side is currently in use and will instead use the next free one.

crates/rush-compiler-risc-v/src/compiler.rs

```
861 fn infix_helper(&mut self, lhs: Register, rhs: Register, op: InfixOp, type_: Type)
    ↪  -> Register {
        // ...
864     let dest_regi = self.get_int_reg();
865     let dest_regf = self.get_float_reg();
866
867     match (type_, op) {
868         (Type::Int(0), InfixOp::Plus) => {
869             self.insert(Instruction::Add(dest_regi, lhs.into(), rhs.into()));
870             dest_regi.into()
871         }
            // ...
977         (Type::Float(0), InfixOp::Minus) => {
978             self.insert(Instruction::Fsub(dest_regf, lhs.into(), rhs.into()));
979             dest_regf.into()
980         }
            // ...
990     }
991 }
```

Listing 5.18 – RISC-V: Parts of the '`infix_helper`' method.

Next, the implementation of the '`infix_helper`' helper method shall be considered. This method exists because translation of the underlying mathematical or logical operation is required during compilation of both infix-expressions and certain assign-expressions, '`a += 1`' for instance. Listing 5.18 shows parts of this method. It takes the registers of the left- and right-hand sides, the operator, and a type. As the registers do not specify one concrete type, it has to be provided additionally. The analyzer guarantees equal types on both sides, hence only one type parameter is needed. In lines 864–865, an integer and a floating-point destination register are created, even though only one of them is used later. However, both are defined beforehand in order to avoid code duplication. In lines 868–871, the code responsible for inserting an integer addition instruction can be seen. For this, the '`insert`' method is used. It inserts a new instruction into the current block. Here, the '`add`' instruction is used with the integer destination register specified in '`dest_regi`' and the left- and right-hand side registers as operands. Since the '`add`' instruction demands it, the operand registers are asserted to be integer registers via the '`into`' method. In lines 977–980, the code for inserting a float subtraction instruction can be seen. Since floating-point operations require different instructions from the ones used for integers, the additional type information is passed to this method. Here, the '`fsub`' (**f**loat **sub**tract) instruction is used for subtraction. In this case, the variable '`dest_regf`' is used as the first operand because the resulting value is a float, and these '`into`' calls assert the registers to be float registers.

Functions

However, before the infix-expression in Listing 5.13 on page 72 is compiled, the compiler iterates over the functions of the program. In this case, only the '`main`' function is present. Listing 5.19 shows the '`define_main_fn`' method. It is only responsible for compilation of the '`main`' function. All other rush functions are compiled by the method '`function_definition`'. However, these two methods only mainly differ in two ways. Firstly, the latter contains code which handles function parameters. Secondly, the special instructions for calling the '`main`' and '`exit`' functions are not required.

crates/rush-compiler-risc-v/src/compiler.rs

```
123 fn define_main_fn(&mut self, node: AnalyzedBlock<'tree>) {
124     let start_label = "_start";
125     self.blocks.push(Block::new(start_label.into()));
126     self.exports.push(start_label.into());
127
128     let main_label = "main..main";
129     self.blocks.push(Block::new(main_label.into()));
130
131     // call the `main` function from `_start`
132     self.insert(Instruction::Call(main_label.into()));
133     // exit with code 0 by default
134     self.insert(Instruction::Li(IntRegister::A0, 0));
135     self.insert(Instruction::Call("exit".into()));
136
137     // add the epilogue label
138     let epilogue_label = self.gen_label("epilogue");
139     self.curr_fn = Some(Function::new(Rc::clone(&epilogue_label)));
140
141     // compile the function body
142     self.insert_at(main_label);
143     self.push_scope();
144     self.function_body(node);
145     self.pop_scope();
146
147     // insertion after body was compiled; frame size is now known
148     let mut prologue = self.prologue();
149     self.insert_at(main_label); // resets the current block back to the fn block
150     prologue.append(&mut self.blocks[self.curr_block].instructions);
151     self.blocks[self.curr_block].instructions = prologue;
152
153     self.blocks.push(Block::new(Rc::clone(&epilogue_label)));
154     self.epilogue()
155 }
```

Listing 5.19 – RISC-V: The '`define_main_fn`' method.

The method takes a block, that is, a list of statements as its input. In lines 124–126, the '`_start`' label is created and exported so that the linker will later be able to refer to it. Next, in lines 128–129, a new block with the '`main..main`' label is created. As explained previously, this block represents the start of the body of the '`main`' function. In line 132, a '`call`' instruction, which is responsible for calling the '`main`' function at the start of program execution, is inserted. In lines 133–135, instructions for calling the '`exit`' function using '0' as the exit code are inserted. Next, in lines 137–139, a new epilogue label is generated, and the '`curr_fn`' field is updated so that it contains this newly created epilogue label. However, this label is not directly added to the '`blocks`' vector since it should appear after the function body.

Next, the '`main`' function's body is compiled. For this, the current insertion position is first updated to the end of the previously created '`main..main`' block by calling the method

'`insert_at`'. Following that, a new variable scope is added for the function body. Then, the function body's contents are compiled by calling the '`function_body`' method. After the body's compilation, the previously added scope is removed again.

In line 148, the '`prologue`' method returns the instructions for the prologue. It is only inserted *after* the function body was compiled, even though it will execute *before* the body during runtime. This is because the prologue needs to allocate a certain amount of stack space, which the compiler can only determine after the code for the body has been generated. Next, in line 149, the insertion position is moved back to the end of the '`main..main`' label. The lines 150–151 are then responsible for prepending the generated prologue instructions to the instructions representing the function body. This is achieved by first appending the contents of the current block to the end of the prologue instructions, and then overwriting the current block with the result. Finally, in lines 153–154, the code responsible for inserting the trailing epilogue code can be seen. For this, the previously created epilogue label is now appended to the '`blocks`' vector, and the '`epilogue`' method of the compiler is called.

crates/rush-compiler-risc-v/src/call.rs

```
66 pub(crate) fn epilogue(&mut self) {
67     let epilogue_label = Rc::clone(&self.curr_fn().epilogue_label);
68     self.insert_at(&epilogue_label);
69     // restore `fp` from the stack
70     self.insert(Instruction::Ld(
71         IntRegister::Fp,
72         Pointer::Register(IntRegister::Sp, self.curr_fn().stack_allocs + 8),
73     ));
74     // restore `ra` from the stack
75     self.insert(Instruction::Ld(
76         IntRegister::Ra,
77         Pointer::Register(IntRegister::Sp, self.curr_fn().stack_allocs),
78     ));
79     // deallocate stack space
80     self.insert(Instruction::Addi(
81         IntRegister::Sp,
82         IntRegister::Sp,
83         self.curr_fn().stack_allocs + BASE_STACK_ALLOCATIONS,
84     ));
85     // return control back to caller
86     self.insert(Instruction::Ret);
87 }
```

Listing 5.20 – RISC-V: The '`epilogue`' method.

Listing 5.20 shows this '`epilogue`' method. It generates the instructions for a function's epilogue. First, in lines 67–68, the insertion position of the compiler is set to the epilogue block. The epilogue must exist in a separate label in order to be used when '`return`' statements are encontered. The instruction insertion in lines 69–73 is responsible for an '`ld`' instruction. Here, the '`ld`' instruction is used to restore the previously saved '`fp`' register. In lines 74–78, another '`ld`' instruction for restoring the '`ra`' register is inserted. In lines 79–84, an '`addi`' instruction is inserted in order to free the memory used for the function's stack frame. Finally, a '`ret`' instruction is inserted which is responsible for transferring control back to the caller.

Let Statements

In lines 2–3 of the rush program in Listing 5.13 on page 72, let-statements are used in order to save values on the stack. The method responsible for compiling these statements is called '`let_statement`'. It saves the initializing expression on the stack and saves the resulting pointer in the current scope. The main work is accomplished by the '`save_expr_on_stack`'

method. Listing 5.21 displays parts of this method. The method takes an expression in order to compile it and save the resulting value on the stack. It returns an '`Option<Pointer>`', where '`None`' again represents the '`!`' and '`()`' types. A '`Pointer`' stores a relative memory address given a base register, '`fp`' in this case.

crates/rush-compiler-risc-v/src/compiler.rs

```
557 fn save_expr_on_stack(/* ... */) -> Option<Pointer> {
562     let type_ = node.result_type();
563     let reg = self.expression(node)?;
564     let comment = format!("{comment_prefix} = {reg}").into();
565     let offset = self.get_offset(Size::from(type_));
566
567     match reg {
            // ...
580         Register::Float(reg) => self.insert_with_comment(
581             Instruction::Fsd(reg, Pointer::Register(IntRegister::Fp, offset)),
582             comment,
583         ),
584     };
585
586     Some(Pointer::Register(IntRegister::Fp, offset))
587 }
```

Listing 5.21 – RISC-V: The '`save_expr_on_stack`' method.

In line 563, the expression is compiled, and the resulting register is saved in the variable '`reg`'. In line 565, the '`get_offset`' method is used in order to calculate the frame pointer offset of a variable. Since types like '`char`' or '`bool`' require less memory compared to '`int`', the size of the expressions's data type must also be available to this method. Now, the '`offset`' variable contains the frame pointer offset at which the variable should be saved. Since the '`get_offset`' method is used every time values need to be saved on the stack, it also increments the '`stack_allocs`' variable, so that the prologue and epilogue are aware of this stack allocation.

The '`match`' block in line 567 executes different code based on the register type which the expression yielded. In lines 580–583, the code responsible for float registers can be seen. Here, the '`fsd`' (**f**loat **s**tore **d**ouble) instruction is used in order to save the resulting register at the memory position relative to '`fp`' which is specified in the variable '`offset`'. The code for integer registers is similar, but also differentiates between 8-bit and 64-bit data. In line 586, the '`Pointer`' abstraction representing the memory location of the expression is returned. Here, it is apparent that this variant of a '`Pointer`' only saves a register and an offset to this register. In this case, '`fp`' is specified as the register and the value of '`offset`' is used as the offset.

Function Calls and Returns

For compiling call-expressions, also the ones of builtin functions, the compiler uses the '`call_expr`' method. Listing 5.22a shows the first part of this method. Due to the aforementioned low-level calling conventions, its implementation has proven to be very complex and demanding. The part of code shown here is only preparing the compilation of the function call. The method consumes an analyzed call-expression and optionally returns a register representing the call return value. For instance, if the '`exit`' function was called, this method would return no register as the '`exit`' function results in the '`!`' type.

In lines 93–99, all registers which are used at the time of the function call are spilled and then marked as unused. This is required since the register allocator of this compiler is so simple that it only considers one function at a time. In production-ready compiler systems,

crates/rush-compiler-risc-v/src/call.rs

```rust
pub(crate) fn call_expr(&mut self, node: AnalyzedCallExpr<'tree>) ->
    Option<Register> {
    // before the function is called, all currently used registers are saved
    let regs_on_stack = self
        .used_registers
        .clone()
        .iter()
        .map(|(reg, size)| (*reg, self.spill_reg(*reg, *size), *size))
        .collect();

    // save the previous state of the used registers
    let used_regs_prev = mem::take(&mut self.used_registers);

    // specifies the argument position of the specified register type
    // type dependent: (`a0` -> `int_cnt = 0`), (`fa0` -> `float_cnt = 0`)
    let mut float_cnt = 0;
    let mut int_cnt = 0;
    // calculate the total byte size of spilled params
    let mut spill_param_size = 0;
    // needed for freeing registers later
    let mut param_regs = vec![];
    // specifies the count of the current register spill
    let mut spill_cnt = 0;
    // ...
```

Listing 5.22a – RISC-V: The '`call_expr`' method.

a register allocation algorithm could also work interprocedurally. It becomes apparent that saving *all* used registers does not produce very efficient output code. However, throughout the extensive testing conducted on the rush backends, this approach has presented itself as simple and reliable[13]. The register spilling is performed by iterating over the currently used registers and calling the '`spill_reg`' method for each register. That method is structured very similarly to the aforementioned '`save_expr_on_stack`' method. After the register spilling, the contents of the '`used_registers`' vector are moved into the local '`used_regs_prev`' variable. Therefore, after the assignment, the '`used_registers`' vector will be empty.

The next part of the '`call_expr`' method is shown is Listing 5.22b. In line 136, the '`spill_param_size`' variable is incremented. It specifies the additional storage required for placing arguments on the stack. For instance, a call with nine integer arguments would need eight additional bytes of memory since the ninth argument would have to be placed on the stack. The optional placement of arguments on the stack is performed by the loop in line 115. It iterates over each argument expression of the call, and for each expression, the match-block in line 116 executes different code depending on the expression's result data type. For instance, expressions which yield in '`()`' or '`!`' do not require registers and can thus be compiled without further consideration. However, for float values, the code in lines 120–139 is executed. Firstly, the argument's expression is compiled and the resulting register is saved in the variable '`res_reg`'. Next, the '`nth_param`' method is called on the '`FloatRegister`' enumeration. This method associates an index of a parameter to a register. For instance, the index '2' represents the third parameter, which is represented by the register '`fa2`'. For unsupported indices, the method returns '`None`'. Therefore, the if-expression in line 123 checks whether there still is a register which could represent the current argument position. This position is saved in the '`float_cnt`' and '`int_cnt`' variables which are incremented whenever an argument of the respective type is encountered, in line 138 for instance. In case a register is available, the code in lines 124–125 is executed. Here, '`res_reg`' is only marked as used

[13]For each new commit, 41 integration tests are conducted on every backend

crates/rush-compiler-risc-v/src/call.rs

```
    // ...
115     for arg in node.args {
116         match arg.result_type() {
117             Type::Unit | Type::Never | Type::Unknown => {
118                 self.expression(arg);
119             }
120             Type::Float(0) => {
121                 let res_reg = self.expression(arg).expect("type is float");
122
123                 if let Some(reg) = FloatRegister::nth_param(float_cnt) {
124                     param_regs.push(reg.to_reg());
125                     self.use_reg(reg.to_reg(), Size::Dword);
126                 } else {
127                     // no more param registers: spilling required
128                     self.insert_with_comment(
129                         Instruction::Fsd(
130                             res_reg.into(),
131                             Pointer::Register(IntRegister::Sp, spill_cnt * 8),
132                         ),
133                         format!("{} byte param spill", /* ... */).into(),
134                     );
135                     spill_cnt += 1;
136                     spill_param_size += 8;
137                 }
138                 float_cnt += 1;
139             }
                // ...
160         }
161     }
    // ...
```

Listing 5.22b – RISC-V: The '`call_expr`' method. (cont.)

and pushed in the '`param_regs`' vector. There is no code validating that the correct register for the argument position is returned. This check is redundant since all registers have been marked as unused prior to this block. Therefore, the register allocator will always return the registers for the argument expressions in the correct order. Otherwise, if there is no register for the current argument, the value has to be spilled onto the stack. For this, an '`fsd`' instruction (or '`sd`' for integers) is inserted in order to save the result of the current argument on the stack. As described in Section 5.2.3 about the RISC-V calling convention, spilled arguments are placed on the stack in a way so that the first spilled argument is at the lowest memory position. In order to calculate this increasing offset easily, the '`spill_cnt`' variable was declared in line 113 of Listing 5.22a. Every time an argument is spilled, this counter is increased. In line 131 of Listing 5.22b, the code calculating the offset relative to '`sp`' can be seen. Here, the value of '`spill_cnt`' is simply multiplied by eight in order to obtain the correct offset, as each argument should use eight bytes. In line 135, the '`spill_cnt`' variable is incremented as described previously. Lastly, the '`spill_param_size`' variable is incremented by eight bytes, as floats require eight bytes of storage. Although just the code for floats is shown here, the algorithm for integer-based types only deviates slightly.

The final part of the '`call_expr`' method is shown in Listing 5.22c. The code in lines 165–173 is responsible for mangling the name of the called function so that it includes the '`main..`' prefix. However, if the '`exit`' function is called, the name is not mangled. In line 174, the '`call`' instruction responsible for performing the actual call is inserted. At this point, the arguments are in their correct places so that the called function will be able to access them. In line 177, the previously emptied '`used_registers`' vector is restored to its state before the call. This is done in order to allow the register allocation for the call argu-

crates/rush-compiler-risc-v/src/call.rs

```rust
    // ...
    self.curr_fn_mut().stack_allocs += spill_param_size;

    // perform function call
    let func_label = match node.func {
        "exit" => {
            // mark the current block as terminated (avoid useless jumps)
            self.curr_block_mut().is_terminated = true;
            "exit".into()
        }
        func => format!("main..{func}").into(),
    };
    self.insert(Instruction::Call(func_label));

    // restore the old list of used registers
    self.used_registers = used_regs_prev;

    let res_reg = match node.result_type {
        Type::Float(0) => Some(FloatRegister::Fa0.to_reg()),
        Type::Int(_) | Type::Char(_) | Type::Bool(_) | Type::Float(_) => {
            Some(IntRegister::A0.to_reg())
        }
        Type::Unit | Type::Never | Type::Unknown => None,
    };

    // restore all caller saved registers again
    self.restore_regs_after_call(res_reg, regs_on_stack)
}
```

Listing 5.22c – RISC-V: The '`call_expr`' method. (cont.)

ments to happen independently of the rest of the compiled code. In line 188, the method '`restore_regs_after_call`' later reloads the old register values from the stack. The code in lines 179–185 is responsible for selecting an appropriate output register based on the function's return type, storing it in '`res_reg`'. For instance, functions which return '`float`' values place their return value inside the register '`fa0`', and for functions which return the '`()`' or '`!`' types, no register is selected and '`None`' is returned. This return register is required as an argument to the following '`restore_regs_after_call`' method call. The reasons why that method has to access the return register is that this register might have been in use before the call. In that case, restoring the register would result in the function's return value to be overwritten. In order to mitigate this issue, the method will alter the result register by inserting a '`mv`' (**mo**ve) instruction if necessary. Since the method returns the final output register which contains the call's result, it is used as the last expression of the block, thus also making it the return value of the '`call_expr`' method.

The last thing to mention is how a called function returns a value. Whenever an explicit or implicit return in a function is encountered, the compiler checks whether the expression already saved its result in the appropriate return register. In case it did not, a '`mv`' instruction is used to move it there. Afterward, a jump to the function's epilogue is issued, causing the function to return.

Loops

Since assembly dialects do not provide high-level control-flow constructs like loops, a compiler targeting assembly has to generate code which reproduces the behaviour. Listing 5.23 shows a rush program containing a while-loop. Next to it, Listing 5.24 shows the part of the compiler output that represents the loop and the '`exit`' call.

```
fn main() {
    let mut count = 0;
    while count < 10 {
        count += 1;
    }
    exit(count);
}
```

Listing 5.23 – A rush program containing a while-loop.

```
while_head_0:
    # while condition
    ld a0, -24(fp)    # count
    li a1, 10
    slt a0, a0, a1
    beqz a0, after_while_0
    # while body
    li a0, 1
    ld a1, -24(fp)    # count
    add a2, a1, a0
    sd a2, -24(fp)
    j while_head_0

after_while_0:
    ld a0, -24(fp)    # count
    call exit
```

Listing 5.24 – Compiler output of the rush program in Listing 5.23.

The rush program contains a while-loop which iterates ten times. At the beginning of the program, the integer variable '`count`' is declared. Before each iteration, the while-loop checks whether '`count`' is less than '10'. If it is, count is incremented by '1', and the loop begins its next iteration. Otherwise, the execution of the loop stops and the code after the loop is executed. In the assembly code, the content of the loop is represented by the instructions below the '`while_head_0`' label. As the comment in line 22 suggests, the first four instructions represent the loop control code. These instructions are executed every time the loop begins a new iteration. They check the condition before any of the code inside the loop's body is executed. If it results in 'false', the execution of the loop should stop. This is accomplished by the '`beqz`' (**b**ranch if **eq**ual to **z**ero) instruction which jumps to the specified label if the value inside the first operand is zero [WA19, p. 105]. Here, this means that the instruction would jump to the '`after_while_0`' label if the condition of the loop evaluated to 'false'.

This label starts a block with the code that follows the while-statement, here only the call to the '`exit`' function. Otherwise, if the condition evaluated to 'true', the '`beqz`' instruction would not perform a jump and the CPU would just continue to the following instruction. The instructions following that conditional jump represent the loop body, i.e., the statement '`count += 1`'. In line 32, and unconditional jump, which jumps back to the '`while_head_0`' label, follows. Therefore, the loop begins its next iteration. Using this approach, the compiler is able to generate instructions for the '`loop`', '`while`', and '`for`' statements.

5.3. x86_64: Compiling to a CISC Architecture

In addition to *reduced instruction set computers* (*RISC*), there are also *complex instruction set computers* (*CISC*). The main differences lie in the instruction count and complexity. For the purposes of this paper, the still widely used CISC architecture *x86_64*[14] is used as an example, just like RISC-V was used to represent RISC architectures in the previous section. While RISC-V with all extensions used by rush[15] provides about 100 different instructions [WA19, Chapter 24], x64 is estimated to have about 3600 instructions [RA17]. This is due to the fact that at the time when most CISC architectures were developed, many programmers still created assembly programs by hand, without the help of compilers. Additional instructions for higher-level concepts were therefore very helpful [Dan05a, p. 9].

One example for such an instruction is the x64 '`leave`' instruction. It is used to release the current stack frame at the end of a procedure. Listing 5.20 on page 77 shows how multiple RISC-V instructions are used to accomplish the same manually. First, the frame pointer register is copied into the stack pointer register, freeing the stack space allocated at the start of the procedure. Then, the old frame pointer, that was saved on the stack, is loaded from the stack back into the frame pointer register. Restoring the return address and returning to the caller is both done by the '`ret`' instruction in x64.

Another similar example is the '`call`' instruction. Although it is also usable in RISC-V assembly, it is a pseudoinstruction provided by the assembler and resolves to multiple other RISC-V instructions during the assembly process. In x64 however, '`call`' is a conventional instruction.

For x64 specifically, there are also some instructions that only operate on certain specific registers. For instance, the '`idiv`' instruction divides a signed 128-bit integer stored in the two 64-bit registers '`%rdx`' and '`%rax`' by the given operand, storing the result in the '`%rax`' register and the remainder in the '`%rdx`' register. This makes usage of such instructions significantly more demanding, as values in these registers must be spilled onto the stack in case they have to be preserved.

The substantial growth of the x64 instruction set and its commitment to backwards compatibility have also led to a number of instructions that have long become obsolete. For example, '`aaa`' (ASCII Adjust after Addition) is used in the context of adding two *Binary Coded Decimal* (*BCD*) values, but that technology is rarely used nowadays [PW17, p. 4].

5.3.1. x64 Assembly

There are multiple different dialects of x64 assembly and multiple assemblers each supporting a different set of dialects. The rush x64 compiler emits assembly code using Intel syntax that can be assembled by the *GNU Assembler* (*GAS*).

Listing 5.25 contains another simple rush program. It defines a global variable called '`a`' and assigns it the integer '2', increments it by one, defines a local boolean called '`b`', using 'true' as its initial value, and exits with the sum of '`a`' and '`b`'. The corresponding assembly code generated by the x64 rush compiler is shown in Listing 5.26. It begins with the '`.intel_syntax`' directive, indicating that the following assembly code uses the Intel syntax. Then, similar to the RISC-V assembly code shown in the previous section, the '`_start`' symbol is marked as global, as it

```
let mut a = 2;

fn main() {
    a += 1;
    let b = true;
    exit(a + b as int);
}
```

Listing 5.25 – Example rush program.

[14] Later shortened to *x64*.
[15] The used extensions are RV32I, RV64I, RV32M, RV64M, RV32D, and RV64D.

again represents the entry point. The sections are also the same, as these are defined by the ELF format which is independent of ISAs.

The main differences between x64 assembly and the previously shown RISC-V assembly, and also the other x64 assembly dialects, are the instruction mnemonics and the register names. When using the Intel syntax for x64, register names are always prepended by a percentage sign. Another difference is the syntax used for pointers. While RISC-V assembly uses '`-1(fp)`' to specify the value in the '`fp`' register as a memory address with an offset of '−1', Intel x64 uses '`byte ptr [%rbp-1]`', as seen in line 16. It is obvious that the latter provides an additional constraint, the size of the value pointed to, here '`byte`'. RISC-V assembly instead encodes this information in the instruction mnemonic. To store a byte, one would use '`sb`' (**s**tore **b**yte), whereas for storing a 64-bit integer, '`sd`' (**s**tore **d**word) is used instead.

```
1  .intel_syntax
2  .global _start
3
4  .section .text
5
6  _start:
7      call        main..main
8      mov         %rdi, 0
9      call        exit
10
11 main..main:
12     push        %rbp
13     mov         %rbp, %rsp
14     sub         %rsp, 16
15     inc         qword ptr [%rip+main..a]
16     mov         byte ptr [%rbp-1], 1  # let b = 1
17     mov         %rdi, qword ptr [%rip+main..a]
18     mov         %sil, byte ptr [%rbp-1]
19     and         %rsi, 255
20     add         %rdi, %rsi
21     call        exit
22 main..main.return:
23     leave
24     ret
25
26 .section .data
27 main..a:
28     .quad       0x0000000000000002  # = 2
```

Listing 5.26 – Compiler output of the rush program in Listing 5.25.

The naming of sizes other than 'byte' was not explained yet, and this, too, differs between RISC-V and x64. Architectures usually define the size of one so-called *word*. In RISC-V, a word is defined as 32 bits, so 4 bytes, in x64 it is defined as 16 bits, so 2 bytes. All other sizes, except for the byte, are then named based on the word size. Therefore, RISC-V assigns 16-bit, 32-bit, and 64-bit the names *half word*, *word*, and *double word* respectively [WA19, p. 6]. For x64, they are instead called *word*, *double word*, and *quadruple word* [Kus18, p. 3]. These names are often shortened to only include the first letter of the factor. For instance, a double word is called *dword*. In some instances, like in line 28, alternative short forms are used. Here, the directive '`.quad`' denotes a quadruple word.

Another difference between x64 and RISC-V is the typical operand structure of instructions. RISC-V requires three operands for '`add`', '`sub`', and alike, these being the destination, the first source, and the second source location. The norm in x64 is using just two operands where the first represents both the first source and the destination [Dan05a, pp. 14–20].

5.3.2. Registers

The base x64 ISA provides 16 general-purpose registers capable of holding 64-bit integers. These are shown in Table 5.2. Additionally, three more sets of 16 registers are defined, each half the size of the previous one. However, instead of these being additional storage, they simply provide an alias to the lower half of the respective larger register. For instance, when writing the byte $(2a)_{16}$ into '`%al`', the least significant byte of '`%ax`', '`%eax`', and '`%rax`' changes to $(2a)_{16}$, too.

Table 5.2 – General-purpose registers of the x64 architecture [Lu+22, pp. 20, 26].

64-bit	32-bit	16-bit	8-bit	Caller-Saved	Purpose
`%rbp`	`%ebp`	`%bp`	`%bpl`		base pointer / frame pointer
`%rsp`	`%esp`	`%sp`	`%spl`		stack pointer
`%rax`	`%eax`	`%ax`	`%al`	✓	1st return register
`%rbx`	`%ebx`	`%bx`	`%bl`		
`%rcx`	`%ecx`	`%cx`	`%cl`	✓	4th argument register
`%rdx`	`%edx`	`%dx`	`%dl`	✓	2nd return register 3rd argument register
`%rsi`	`%esi`	`%si`	`%sil`	✓	2nd argument register
`%rdi`	`%edi`	`%di`	`%dil`	✓	1st argument register
`%r8`	`%r8d`	`%r8w`	`%r8b`	✓	5th argument register
`%r9`	`%r9d`	`%r9w`	`%r9b`	✓	6th argument register
`%r10`	`%r10d`	`%r10w`	`%r10b`	✓	
`%r11`	`%r11d`	`%r11w`	`%r11b`	✓	
`%r12`	`%r12d`	`%r12w`	`%r12b`		
`%r13`	`%r13d`	`%r13w`	`%r13b`		
`%r14`	`%r14d`	`%r14w`	`%r14b`		
`%r15`	`%r15d`	`%r15w`	`%r15b`		

Considering line 18 of Listing 5.26, the move of the value behind the '`byte ptr`' requires the destination register to be one byte in size, too. For this reason, the compiler uses the '`%sil`' register here, instead of its 64-bit equivalent '`%rsi`'. Line 19 then performs the cast from the boolean, saved as a byte, to a 64-bit integer. It does this by taking the 64-bit variant of the register, in this case '`%rsi`', and assuring that all bits, except for the least significant eight, are zeros. The latter is done using the *bitwise AND* operation with the number $2^8 - 1 = 255$, which in binary is represented by eight ones.

Another special register is '`%rip`'. It stores the instruction pointer, that is, the address of the current instruction being executed. The register is usually not modified directly, as branching can be achieved with instructions like '`call`', '`ret`', or '`jmp`'. Nevertheless, it is sometimes accessed manually, as seen in lines 15 and 17. In these lines, read and write operations on the global variable '`a`', defined at the '`main..a`' label, are performed. Pointers to such global variables and constants in the '`.data`' and '`.rodata`' sections require the instruction pointer as their base in x64 assembly.

In addition to these general-purpose registers, x64 also defines 16 registers for floating-point numbers, each being 128 bits in size and labelled '`%xmm0`' through '`%xmm15`'. For the purposes of rush, only the lower 64 bits are ever used, as this is enough to represent double precision floating-point numbers. The additional 64 bits are provided for *single instruction multiple data* (*SIMD*) instructions. They can be used by more optimized compilers to operate on multiple values at a time. Any single one of these registers would then for instance hold two 64-bit values or four 32-bit values [Kus18, pp. 5–7].

5.3.3. Stack Layout and Calling Convention

Alongside the instructions and registers defined by x64 itself, an additional ABI document defines how programs should structure the stack during execution. Figure 5.6 shows this layout. The overall outline is similar to that of RISC-V, but some details differ. For one, the order of the saved return address and the saved '`%rbp`' register is inverted. Secondly, the address '`%rbp`' points to differs by 16 bytes. Therefore, in x64 the first stack argument is accessed by offsetting '`%rbp`' by 16, whereas in RISC-V the '`fp`' register already points to the first stack argument without any offset. In both calling conventions, every stack argument uses eight bytes of space. However, x64 has no special return address register, and instead only stores the return address on the stack and retrieves it from there when needed. At the moment of calling a function, the stack frame's size must be aligned to a multiple of 16 bytes. The unspecified space between '`%rbp-8`' and '`%rsp`' can be freely used by the current function to save local variables. Its internal layout may differ between programming languages [Lu+22, p. 21].

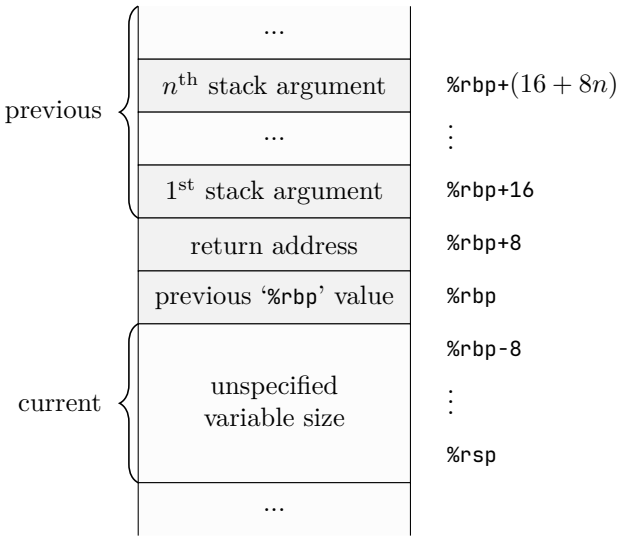


Figure 5.6 – Stack layout of the x64 architecture [Lu+22, p. 21].

5.3.4. Implementation

crates/rush-compiler-x86-64/src/instruction.rs

```
14  pub enum Instruction {
        // ...
20      Section(Section),
21      Label(Rc<str>, bool),
        // ...
46      Add(IntValue, IntValue),
47      Sub(IntValue, IntValue),
        // ...
104 }
```

Listing 5.27 – x64: The '`Instruction`' definition.

Even though many implementation details could be the same as for the RISC-V compiler, there are some differences because we each created one separately without using any reference implementation. One such difference is that the x64 compiler's '`Instruction`' definition, which is shown in Listing 5.27, not only includes instructions, but also directives and labels. This way, an entire assembly file can be represented as one '`Vec<Instruction>`' with every '`Instruction`' representing one line of the assembly code. A matching '`Display`' implementation[16] then simply emits the respective assembly representation as a string.

Line 20 shows the definition of the '`.section`' directive, simply holding an instance of another enumeration describing the valid sections. Line 21 defines an assembly label, holding its name and a boolean indicating whether a blank line should be added above.

Instructions, e.g., '`add`' and '`sub`', also cannot simply take registers as operands, as x64 often allows memory pointers and immediate values as operands. Instead, another enumeration, called '`IntValue`', is introduced to allow these three variants. Listing 5.28 shows its definition.

[16] In Rust, implementing the '`Display`' trait on a type defines how instances of that type can be presented in text form to a user.

Struct Fields

The compiler struct itself, shown in Listing 5.29, is also a bit different. Both compilers have a '`used_registers`' field, but this compiler does not need to store the size for each register because the registers themselves already have a defined size (line 19). Both compilers have a '`scopes`' field for storing variable locations, but the RISC-V compiler has the '`Option<_>`' inside the '`Variable`' struct (line 26). Both compilers store the global variables, but this compiler has separate fields for every size (lines 46–56). The same applies for constants in the '`.rodata`' section (lines 58–70). And both compilers have one field for accumulating the final assembly code, but this compiler saves a list of '`Instruction`'s instead of '`Block`'s (line 44).

crates/rush-compiler-x86-64/src/value.rs

```rust
pub enum IntValue {
    Register(IntRegister),
    Ptr(Pointer),
    Immediate(i64),
}
```

Listing 5.28 – x64: The '`IntValue`' definition.

crates/rush-compiler-x86-64/src/compiler.rs

```rust
pub struct Compiler<'src> {
    /// The instructions for the current function body
    pub(crate) function_body: Vec<Instruction>,
    /// The currently used [`IntRegister`]s
    pub(crate) used_registers: Vec<IntRegister>,
    // ...
    /// Maps variable names to `Option<Variable>`, or `None` for types `()` and `!`
    pub(crate) scopes: Vec<HashMap<&'src str, Option<Variable>>>,
    /// Internal stack pointer, separate from `%rsp`, used for pushing and popping
    /// inside the stack frame. Relative to `%rbp`, always positive.
    pub(crate) stack_pointer: i64,
    /// Size of current stack frame, always `>= self.stack_pointer`
    pub(crate) frame_size: i64,
    // ...
    /// The text section (aka code section)
    pub(crate) text_section: Vec<Instruction>,

    //////// .data section ////////
    /// Globals with 64-bits
    pub(crate) quad_globals: Vec<(Rc<str>, u64)>,
    // ...
    /// Globals with 8-bits
    pub(crate) byte_globals: Vec<(Rc<str>, u8)>,

    //////// .rodata section ////////
    /// Constants with 128-bits (maps value to name)
    pub(crate) octa_constants: HashMap<u128, Rc<str>>,
    // ...
    /// Constants with 8-bits (maps value to name)
    pub(crate) byte_constants: HashMap<u8, Rc<str>>,
}
```

Listing 5.29 – The '`Compiler`' struct definition of the x64 compiler.

Additional fields of this compiler are '`function_body`', '`stack_pointer`', and '`frame_size`'. The '`function_body`' field stores the instructions of the current function. During compilation of statements and expressions, emitted instructions are always appended to this vector. The method for compiling a function body then produces the prologue, and afterward appends the vector contents to '`text_section`'. This is required because some values that are needed in the prologue, like the '`frame_size`', cannot be known before compilation of the function body has completed, but must appear before the body in the assembly code.

Memory Management

The '`stack_pointer`' field is used to keep track of the current stack frame. Every time a variable definition is encountered, or a register is spilled, stack space for it is reserved by increasing this pointer. This process is done by the '`push_to_stack`' method visible in Listing 5.30. It takes a value, its size, and an optional comment as parameters. At first, it aligns the stack pointer to a multiple of the values' size by using the '`align`' function. Then, it adds the number of bytes required for the value to the stack pointer, and increases the frame size in case it is not big enough yet. Now, the pointer to this value is constructed using '`%rbp`' as the base and the negated stack pointer as the offset. Finally, a '`mov`' instruction for inserting the value at the location of the pointer is added to the assembly code, and the pointer is returned for further use.

crates/rush-compiler-x86-64/src/compiler.rs

```
254 pub(crate) fn push_to_stack(
255     &mut self,
256     size: Size,
257     value: Value,
258     comment: Option<impl Into<Cow<'static, str>>>,
259 ) -> Pointer {
260     // add padding for correct alignment
261     Self::align(&mut self.stack_pointer, size);
262     // allocate space on stack
263     self.stack_pointer += size.byte_count();
264     // possibly expand frame size
265     self.frame_size = self.frame_size.max(self.stack_pointer);
266     // get pointer to location in stack
267     let ptr = Pointer::new(
268         size,
269         IntRegister::Rbp,
270         Offset::Immediate(-self.stack_pointer),
271     );
    // ...
277     self.function_body.push(match comment {/* ... */});
281
282     ptr
283 }
```

Listing 5.30 – x64: Stack space reservation for values.

Register Allocation

crates/rush-compiler-x86-64/src/register.rs

```
257 pub fn next(&self) -> Self {
258     match self.in_qword_size() {
259         Self::Rax => Self::Rdi,
260         Self::Rdi => Self::Rsi,
261         Self::Rsi => Self::Rdx,
        // ...
271         Self::R15 => Self::Rbx,
272         reg => panic!(/* ... */),
273     }
274 }
```

Listing 5.31 – x64: Register allocation.

The register allocation algorithm of the x64 compiler differs quite a lot from the RISC-V version. Instead of marking individual registers as used or unused and then searching through a specified order for the first free one, this compiler uses an almost stack-like behavior. Again, one specific order of registers is defined, starting with the return register '`%rax`', followed by the argument registers in their correct order, and ending with the remaining general-purpose registers. Whenever a free register is required, the '`next`' method, displayed in Listing 5.31, is called on the last used register, returning the next register in line, which is then

added to the '`used_registers`' field. For freeing a register, the compiler just removes the last register from this list. This only works because the rest of the compiler guarantees used registers to be freed in exactly the reverse order they were acquired in. With this guarantee, it is apparent that at any point in time during compilation, the registers in the '`used_registers`' field are in the exact order specified by Listing 5.31 without gaps. The same is done separately for the float registers.

Functions

To explain the process of compiling functions and function calls, there is another example program to consider in Listing 5.32, along with the produced assembly code in Listing 5.33. Although not explicitly separated by comments, the x64 compiler also uses the concept of a function prologue, body, and epilogue. Since the '`main`' function in this example does not require any stack space for variables or register spilling, the prologue is automatically omitted, and the epilogue does not contain a '`leave`' instruction for deallocating the memory. The '`foo`' function however does declare a local variable and thus requires stack space to be allocated in the prologue. The lines 20–22 are responsible for this. As previously shown in Figure 5.6, the stack should first contain the return address and then the previous base pointer, the former of which is already handled by the '`call`' instruction. Therefore, the function's prologue begins with pushing '`%rbp`' onto the stack. Afterward, the base pointer's value is set to the current stack pointer. The stack pointer itself is then decremented by the amount of bytes needed for the function, which is saved in '`frame_size`', rounded up to a multiple of 16.

```
fn main() {
    exit(foo(21));
}

fn foo(n: int) -> int {
    return n * 2;
    let b = n;
}
```

Listing 5.32 – Another example rush program with two functions.

```
11 main..main:
12     mov         %rdi, 21
13     call        main..foo
14     mov         %rdi, %rax
15     call        exit
16 main..main.return:
17     ret
18
19 main..foo:
20     push        %rbp
21     mov         %rbp, %rsp
22     sub         %rsp, 16
23     mov         qword ptr [%rbp-8], %rdi
   ↪   # param n = %rdi
24     mov         %rax, qword ptr [%rbp-8]
25     imul        %rax, 2
26     jmp         main..foo.return
   ↪   # return %rax;
27     mov         %rax, qword ptr [%rbp-8]
28     mov         qword ptr [%rbp-16], %rax
   ↪   # let b = %rax
29 main..foo.return:
30     leave
31     ret
```

Listing 5.33 – Trimmed compiler output of the rush program in Listing 5.32.

The epilogue ranges from line 29 to line 31. It defines a label where '`return`' statements can jump to, releases the stack frame using the '`leave`' instruction, and returns control back to the caller with the '`ret`' instruction. Usage of the return label can be seen in line 26, which performs an unconditional jump to that label, rendering all intermediate instructions unreachable. This is the expected behavior because the jump instruction was emitted due to the '`return`' statement in line 6 of the rush program, which should skip to the end of the function. The return value was moved into the '`%rax`' register, or its appropriately sized variant, in advance.

Function Calls

To support function calls, two main aspects must be considered: passing arguments from the caller, and retrieving arguments in the callee. Generally, for passing arguments, the compiler simply iterates over the argument expressions, compiles each one, and moves its result into the appropriate register, or onto the stack if no registers are left. As the first six integer arguments are always passed through the same registers, a function call always requires access to these exact registers, or at least a subset of them. That means if some of the registers happen to be in use already, their contents have to be spilled onto the stack and replaced with the argument value. An example for when this might happen is a function call as a non-first argument to another function call, like in '`foo(42, bar('x'))`'. When the compiler has to move the first argument for the inner function call into '`%rdi`', here the argument '`'x'`' for the function '`bar`', it notices this register is already used for the first argument of the outer function call, '`42`'.

```
5  fn bar(a: bool) {
6      let b = if a { 42f } else { 3.14 };
7      let c = if b == 42f { 11 } else { 3 };
8      loop {
9          break;
10         continue;
11         3 / c;
12     }
13 }
```

Listing 5.34 – A rush example function containing if-expressions and a loop.

Each function that takes parameters should move all passed arguments from their registers onto the stack in order to make the registers available for the rest of the function body. In Listing 5.33 this happens in line 23 for the one parameter called '`n`'. More optimized compilers can of course try to keep the arguments in the registers for as long as they are not specifically needed elsewhere, but as the rush compilers only serve educational purposes, this is not done.

Control Flow

Just as with RISC-V, control flow constructs, like loops and if-expressions, must be represented using simple jumps in assembly. There are both conditional and unconditional jumps, the latter of which was already outlined during explanation of '`return`' statements. An unconditional jump, done in x64 with the '`jmp`' instruction, only takes a label to jump to, and does so, as the name implies, without checking for any condition. Conditional jumps are rather a set of different instructions, each testing a different condition. Again, the way x64 handles conditions differs from RISC-V. It provides a set of *flags*[17] indicating various relations between two values, which are set by dedicated compare instructions like '`cmp`' for integers, '`test`' for booleans, and '`ucomisd`' for floats. A conditional jump instruction, like '`je`' (**j**ump if **e**qual), then only has to query the flag that indicates equality, and optionally performs a jump based on its value.

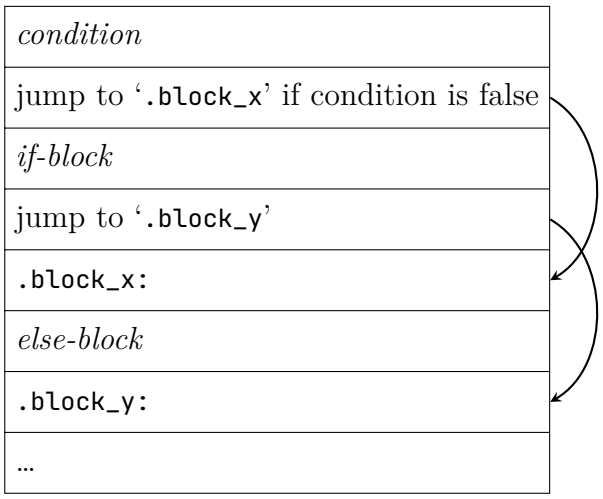


Figure 5.7 – Structure of if-expressions in assembly.

Listing 5.34 shows an example rush function containing if-expressions and a loop

[17] A flag is a boolean value, typically stored in batch with one bit per flag.

```
main..bar:
    push        %rbp
    mov         %rbp, %rsp
    sub         %rsp, 32
    mov         byte ptr [%rbp-1], %dil                 # param a = %dil
    test        byte ptr [%rbp-1], 1
    je          .block_0                                # if a {
    movsd       %xmm0, qword ptr [%rip+.quad_constant_0] #   42f
    jmp         .block_1
.block_0:                                               # } else {
    movsd       %xmm0, qword ptr [%rip+.quad_constant_1] #   3.14
.block_1:                                               # }
    movsd       qword ptr [%rbp-16], %xmm0              # let b = %xmm0
    movsd       %xmm0, qword ptr [%rbp-16]
    ucomisd     %xmm0, qword ptr [%rip+.quad_constant_0]
    jp          .block_2
    jne         .block_2                                # if b == 42f {
    mov         %rax, 11                                #   11
    jmp         .block_3
.block_2:                                               # } else {
    mov         %rax, 3                                 #   3
.block_3:                                               # }
    mov         qword ptr [%rbp-24], %rax               # let c = %rax
.block_4:                                               # loop {
    jmp         .block_5                                #   break;
    jmp         .block_4                                #   continue;
    mov         %rax, 3
    cqo
    idiv        qword ptr [%rbp-24]                     #   3 / c;
    jmp         .block_4                                #   continue;
.block_5:                                               # }
main..bar.return:
    leave
    ret

.section .rodata
.quad_constant_0:
    .quad       0x4045000000000000  # = 42.0
.quad_constant_1:
    .quad       0x40091eb851eb851f  # = 3.14
```

Listing 5.35 – Trimmed compiler output of the rush function in Listing 5.34.

with one ‘`break`’ and one ‘`continue`’ statement. The resulting assembly code is displayed in Listing 5.35. The displayed function serves no purpose, and the analyzer rightfully warns about unreachable statements and unused variables, but it still serves to show the translation of these constructs.

Compiled if-expressions always follow the same outline, as it is shown in Figure 5.7. First, the condition is compiled and an initial jump instruction that jumps to the else-block in case the condition evaluates to ‘false’ is emitted. The instructions for the if-block, concluded by an unconditional jump to behind the else-block, follow. Now, the first label is inserted, here ‘`.block_x`’, followed by the else-block. The entire construct ends with the second label, here ‘`.block_y`’. Any following code goes after that. It is apparent that this structure encodes the expected if-else branching behavior. The code from the if-block is only ever reached if the condition is ‘true’, and it then skips over the else-block to the following instructions. If the condition is ‘false’, the if-block is immediately skipped, and only the else-block is executed which then also continues with the following instructions, causing the two branches to merge.

Starting with the if-expression in line 6 of Listing 5.34, the condition here is merely a

boolean variable. In order to execute a jump based on the value of this variable, the '`test`' instruction is used with both the variable pointer and a constant '1' as operands in line 22 of Listing 5.35. This results in the equality flag being set to '1' if a bitwise AND operation of these operands resulted in '0', that is, if the boolean variable was 'false'. The following '`je`' instruction in line 23 therefore skips to the else-block in case the boolean value was 'false'.

The compiler could now simply compile all condition expressions as usual, always resulting in one '`IntValue`' holding a boolean, and use the same simple boolean check every time. However, since x64 provides additional jump instructions for basic conditions like '`==`', '`!=`', '`>=`', or '`<`', the compiler will try to use one of these when the source rush expression is simple enough. This can be seen in lines 30–33 of the assembly code, which represent the condition of the if-expression in line 7 of the rush code. The compiler detects the condition to be a simple equality check between two floats, and therefore issues a '`ucomisd`' instruction in line 31 to compare the floats, and a '`jne`' (**j**ump if **n**ot **e**qual) instruction in line 33 to jump if the floats are not equal. The reason for the '`jp`' (**j**ump if **p**arity) instruction in between is explained later.

The representation of the loop is much simpler. Only two labels must be defined, one before and one after the loop's contents, and an unconditional jump back to the former must be performed at the end of every iteration. This second point is the reason for the comments in Listing 5.35 showing two '`continue`' statements, even though the rush function only contains one. An unconditional jump back to the loop's start is exactly the behavior of '`continue`' statements. And '`break`' statements are not much different either as they just jump to behind the loop instead. For conditional loops, that is, while-loops and for-loops, there is an additional *head* effectively performing '`if !condition { break; }`' at the start of each iteration. For-loops also initialize a variable before the first iteration and call an update-expression at the end of each one.

Integer Division and Float Comparisons

This subsection's title might seem a little specific compared to the others, but it has the reason that this subsection highlights two unexpectedly complex difficulties which we encountered during the creation of this compiler. They were both already hinted at before. Firstly, there is integer division with the '`idiv`' instruction, which was shortly explained in the introduction to x64 on page 83. It is used when compiling rush infix-expressions between two integers that use either the '`/`' or the '`%`' operator.

Listing 5.36 shows this compilation process. As '`idiv`' always operates on the '`%rax`' and '`%rdx`' registers, it must first be assured that they are free for use. The left-hand side of the division, the numerator, is then moved into '`%rax`' and sign-extended[18] to 128 bits via the '`cqo`' (**c**onvert **q**uadruple word to **o**ctuple word) instruction. Now, the right-hand side, the denominator, is made sure to either be a register or a memory pointer, which is then used in the call to '`idiv`' in line 87. The division result, either '`%rax`' or '`%rdx`' depending on the operator, is then moved into the register that previously contained the numerator, as this is guaranteed to be available now. To finish up, possibly spilled registers are reloaded, and the register is returned as the expression's result. The resulting assembly of this can be seen in lines 43–45 of Listing 5.35.

The second unexpected difficulty was presented by floating-point number comparisons. Floating-point numbers, as defined by the *IEEE 754–2008* standard, have the special property of not forming a total order. That means, not any two arbitrary values are comparable. For instance, the float standard defines a 'NaN'[19] value, which cannot be compared to it-

[18] Converting an n-bit sized integer to an m-bit sized one, where $m > n$, by prepending either zeros or ones to the binary representation, depending on the number's sign.

[19] Short for "not a number".

crates/rush-compiler-x86-64/src/infix.rs

```
12  pub(crate) fn compile_infix(/* ... */) -> Option<Value> {
19      match (lhs, rhs, op) {
            // ...
56          (Some(Register::Int(left)), Some(Value::Int(right)), InfixOp::Div |
    ↪   InfixOp::Rem) => {
                // ...
63              // make sure the rax and rdx registers are free
64              let spilled_rax = self.spill_int_if_used(IntRegister::Rax);
65              let spilled_rdx = self.spill_int_if_used(IntRegister::Rdx);
66              // move lhs result into rax
67              // analyzer guarantees `left` and `right` to be 8 bytes in size
68              self.function_body
69                  .push(Instruction::Mov(IntRegister::Rax.into(), left.into()));
70              // sign-extend lhs to 128 bits (required for IDIV)
71              self.function_body.push(Instruction::Cqo);
72              // get source operand
73              let source = match right {/* ... */};
86              // divide
87              self.function_body.push(Instruction::Idiv(source));
88              // move result into result register
89              self.function_body.push(Instruction::Mov(
90                  left.into(),
91                  // use either `%rax` or `%rdx` register, depending on operator
92                  IntValue::Register(match op {
93                      InfixOp::Div => IntRegister::Rax,
94                      InfixOp::Rem => IntRegister::Rdx,
95                      _ => unreachable!("this arm only matches with `/` or `%`"),
96                  }),
97              ));
                // ...
101             // reload spilled registers
102             self.reload_int_if_used(spilled_rax);
103             self.reload_int_if_used(spilled_rdx);
104
105             Some(Value::Int(IntValue::Register(left)))
106         }
            // ...
249     }
250 }
```

Listing 5.36 – x64: Compilation of integer division.

self [PH17, pp. 215f]. In rush an expression comparing two incomparable float values should result in 'false'. This is the reason for the additional '`jp`' instruction in line 32 of Listing 5.35 which was previously hinted at on page 92. When comparing two floats with the '`ucomisd`' instruction and there is no clear order defined, the *parity* flag is set to '1'. The '`jp`' instruction then jumps straight to the else-block if this flag is set, hereby interpreting an expression like '`NaN == NaN`' as 'false'.

How this is achieved in the compiler is visible in Listing 5.37. The shown code snippet is inside the '`condition`' method which is used during compilation of if-expressions and conditional loops. After pushing the '`ucomisd`' instruction, two additional checks are performed. Firstly, if the condition is '`==`', then a conditional jump to the else-block is added for when the result is unordered. Secondly, if the condition is '`!=`', then a conditional jump to the else-block is added for when the result is *not* unordered. An expression like '`NaN != NaN`' is therefore considered to be true.

crates/rush-compiler-x86-64/src/compiler.rs

```
927 (Value::Float(lhs), Value::Float(rhs)) => {
        // ...
944     self.function_body.push(Instruction::Ucomisd(lhs, rhs));
945
946     if cond == Condition::Equal {
947         // if floats should be equal but result is unordered, jump to end
948         self.function_body.push(Instruction::JCond(
949             Condition::Parity,
950             Rc::clone(&false_label),
951         ));
952     } else if cond == Condition::NotEqual {
953         // if floats should not be equal and result is not unordered, jump to end
954         self.function_body.push(Instruction::JCond(
955             Condition::NotParity,
956             Rc::clone(&false_label),
957         ));
958     }
959 }
```

Listing 5.37 – x64: Compilation of float comparisons.

Pointers

As for pointers, the x64 compiler operates based on the same principles as the RISC-V one. However, the exact implementation differs slightly as x64 provides a specialized '`lea`' (**l**oad **e**ffective **a**ddress) instruction for calculating a pointer address at runtime. For referencing a variable stored at an offset of '−24' from the frame pointer, the RISC-V compiler produces the instruction '`addi a0, fp, -24`' in order to manually subtract '24' from the address stored in '`fp`'. The x64 compiler instead emits '`lea %rax, qword ptr [%rbp-24]`'. Here, the same syntax for pointers can be used as for accesses to the values behind pointers. In fact, in order to load the value instead of the address into '`%rax`', one only has to substitute '`lea`' with '`mov`'.

5.4. Conclusion: RISC vs. CISC Architectures

During implementation of the RISC-V and x64 compilers, it has become apparent how each type of architecture affects the project's complexity. One metric for the that is line count. At the state of Git commit '`f8b9b9a`', the RISC-V compiler consisted of 2234 lines of code while its x64 counterpart required 2751, meaning that the latter required roughly 500 lines more than the former. Apart from objective metrics, implementation of the x64 compiler has presented itself as slightly more demanding compared to the one targeting RISC-V. For instance, the previously explained issues of integer division and float comparison presented just an example of how the architecture complicated development.

Furthermore, documentation and learning resources also showed a significant discrepancy. While there are many resources on the RISC-V architecture available, complete guides for implementing programs in its assembly were sparse. For x64, online resources covering assembly were slightly more prominent due to its larger popularity. However, easy to comprehend instruction references were only provided by third-party individuals, as the official Intel references lack structure to the point where it can be considered unhelpful. For RISC-V, the official ISA specification is held up-to-date and is easily accessible, making it very helpful during the design of a compiler. Overall, it seemed like the RISC-V community put a lot of effort into documentation, making the architecture available for everyone.

Due to the discussed factors, mainly the latter one, RISC-V seems like a very promising architecture which will probably gain a lot more popularity in the future. Simultaneously, CISC architectures, like x64, seem like they are becoming increasingly obsolete. Furthermore, significant licensing fees introduced by large corporate entities like Intel lead to the architecture being unsuitable for third-party chip manufacturers. Since RISC-V is an open source architecture, no licensing fees are required.

It is to be mentioned that CISC architectures, like x64, itself cannot be considered bad. For instance, since the introduction of x64, it has served in millions of devices. However, since it dates back to the time when assembly programs were still written by hand, its features are just not up-to-date in a world where code is mostly generated by compilers. Since CISC architectures provide a lot more, increasingly complex instructions, handwritten assembly programs will run faster at the expense of programs generated by a compiler. The differences in documentation and reference material can also be explained when considering the age of the architectures. Because RISC-V intents to attract cooperations to use their architecture, good documentation is vital.

To summarize, during creation of the rush project and this paper, RISC-V has presented itself as the more attractive architecture, both in objective metrics and subjective experiences. Therefore, if the reader intents to develop a compiler on their own, we strongly recommend choosing a RISC over a CISC architecture as the compiler's target.

6. Final Thoughts and Conclusions

Chapter 2 explained how the syntactical and semantic analysis play a vital role before program execution can start. It has been explained how the *parser* is used for analyzing the program's syntax in order to construct an *abstract syntax tree*, and how the *semantic analyzer* validates the program's semantic rules.

Next, in Chapter 3, we have explained how a tree-walking interpreter can be used to interpret the AST directly. Furthermore, a virtual machine has been presented as a way to leverage some advantages of interpreters while achieving greater runtime speed. Here, the tree-walking interpreter has presented itself as the easier solution, both in planning and implementation. The virtual machine's implementation instead was slightly more demanding as it also required a small compiler.

Chapter 4 provided an overview of compilation to high-level targets. Here, *WebAssembly* and *LLVM* have been used as examples. Although the former is one specific architecture, it is very portable and provides many high-level constructs by itself. This, combined with the fact that it is stack-based rather than register-based, resulted in a rather simple compiler. The more difficult aspects are mostly caused by our decision to target WebAssembly's binary format directly, which could have been avoided by targeting the text format instead. Leveraging LLVM has proven itself to make implementation of a high-performance, multi-target compiler both feasible and easy. Due to helpful literature and guides on LLVM being rather sparse, we have used the implementation of some other Rust LLVM compilers in order to learn about its principles and concepts. To summarize, writing a compiler targeting these high-level targets presented a relatively accomplishable task. If we were to construct a compiler for a more complicated language, LLVM (or a similar framework) would present an attractive solution considering its current relevance in the software industry.

In Chapter 5, low-level programming concepts and compilation to low-level targets have been covered. During the research and software implementation phase of this project, this chapter has proven itself to be the most demanding by far. Reasons for this are that writing a compiler targeting a low-level architecture requires detailed knowledge about the target machine. Programming in assembly allows for many subtle bugs which are hard to track down later. As a result of this, we have both spent many hours trying to locate subtle bugs which were rooted in the assembly program. Moreover, creating a compiler that emits efficient code has proven to be very difficult. Therefore, our compilers only focus on the minimal principles without regarding optimization techniques. Like initially expected, implementation of the x86_64 compiler has proven to be more demanding than the implementation of the RISC-V compiler.

To summarize, this paper has presented two entirely different means of program execution: interpreters and compilers. In order to demonstrate the differences using practical examples, we have implemented our own programming language called *rush*. In total, two interpreters, one transpiler, and four compilers were developed. Furthermore, we have created additional tooling like a language server, a web playground, and a command line interface for the backends. Production-ready programming languages often implement a lot more tooling, including dependency managers, build systems, intricate analyzers, and formatters. Since this paper is primarily about methods of program execution, a reader interested in this tooling might browse the rush GitHub organization at '`https://github.com/rush-rs`'.

Acknowledgements

We would like to express our sincere gratitude towards *Sonja Sokolović* for her invaluable supervision and support during the creation of this paper. Our gratitude extends to our school, the CFG Wuppertal, which allowed us to pursue this research project as part of our final exams. During the creation of this paper, both of us gained lots of invaluable knowledge about compilers, interpreters, and low-level programming. Even though the past months were very stressful, this project was very enjoyable. Additionally, we would like to thank our classmate *Fatima* for inspiring the rush logo.

List of Figures

List of Tables

List of Listings

Bibliography

[Bac+60] J. W. Backus et al. “Report on the algorithmic language ALGOL 60”. In: 3.5 (May 1960). Ed. by Peter Naur, pp. 299–314. DOI: 10.1145/367236.367262.

[Pra73] Vaughan R. Pratt. “Top down Operator Precedence”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 41–51. ISBN: 978-1-4503-7349-4. DOI: 10.1145/512927.512931.

[Wir77] Niklaus Wirth. “What can we do about the unnecessary diversity of notation for syntactic definitions?” In: *Communications of the ACM* 20.11 (Nov. 1977), pp. 822–823. DOI: 10.1145/359863.359883.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd ed. Murray Hill, New Jersey: Prentice Hall, 1988.

[Tor91] Linus Torvalds. *Linux*. 1991. URL: https://github.com/torvalds/linux.

[Wal98] John Waldron. *Introduction to RISC Assembly Language Programming*. Pearson Education, 1998. ISBN: 0-201-39828-1.

[Lev00] John R. Levine. *Linkers and Loaders*. San Francisco, CA: Morgan Kaufmann, 2000.

[Lat02] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[Dan05a] Sivarama P Dandamudi. *Guide to RISC processors*. Ottawa, Canada: Springer International Publishing, Feb. 2005. ISBN: 0-387-21017-2.

[Dan05b] Sivarama P. Dandamudi. *Introduction to Assembly Language Programming: For Pentium and RISC Processors*. 2nd ed. Springer International Publishing, 2005. ISBN: 0-387-20636-1.

[Wir05] Niklaus Wirth. *Compiler Construction*. Zürich, 2005. ISBN: 0-201-40353-6.

[TN07] Allen B. Tucker and Robert E. Noonan. *Programming Languages: Principles and Paradigms*. 2nd ed. McGraw-Hill Education, Nov. 2007. ISBN: 978-007-125439-7.

[Mak09] Ronald Mak. *Writing Compilers and Interpreters*. 3rd ed. Indianapolis, IN: Wiley Publishing, 2009. ISBN: 978-0-470-17707-5.

[Fan10] Dominic Fandrey. *Clang/LLVM Maturity Report*. Karlsruhe, Germany, June 2010.

[Hol12] Nils M. Holm. *Practical Compiler Construction*. Raleigh, NC: Lulu Press, 2012.

[Lov13] Robert Love. *Linux System Programming*. 2nd ed. Sebastopol, CA: O’Reilly Media, May 2013. ISBN: 978-1-449-33953-1.

[CA14] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*. Birmingham, UK: Packt Publishing, Aug. 2014. ISBN: 978-1-78216-692-4.

[Lin+14] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 8 edition*. Boston, MA: Addison-Wesley Professional, May 2014. ISBN: 978-0-13-390590-8.

[Kol17] Daniel Kolsoi. *Inkwell*. 2017. URL: https://github.com/TheDan64/inkwell.

[Mog17] Torben Ægidius Mogensen. *Introduction to Compiler Design*. 2nd ed. Copenhagen, Denmark: Springer International Publishing, 2017. ISBN: 978-3-319-66965-6.

[PH17] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, Apr. 2017. ISBN: 978-0-12-812275-4.

[PW17] David A. Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon LLC, Oct. 2017. ISBN: 978-0-9992491-0-9.

[RA17] Steven Rodgers and Richard A. Uhlig. "X86: Approaching 40 and Still Going Strong". In: (June 2017).

[Wat17] Des Watson. *A practical approach to compiler construction*. en. 1st ed. Undergraduate Topics in Computer Science. Cham, Switzerland: Springer International Publishing, Apr. 2017. ISBN: 978-3-319-52787-1.

[Zhi17] Igor Zhirkov. *Low-Level Programming: C, Assembly, and Program Execution on Intel® 64 Architecture*. 1st ed. Saint Petersburg, Russia: APress, June 2017. ISBN: 978-1-4842-2403-8.

[Kus18] Daniel Kusswurm. *Modern X86 Assembly Language Programming*. 2nd ed. Geneva, IL, USA: APress, Dec. 2018. ISBN: 978-1-4842-4063-2.

[Che19] Boris Cherny. *Programming TypeScript*. Sebastopol, CA: O'Reilly Media, May 2019.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. San Francisco, CA: No Starch Press, 2019. ISBN: 978-1-7185-0044-0.

[Ser19] Alejandro Serrano Mena. *Practical Haskell: A Real World Guide to Programming*. 2nd ed. Utrecht, The Netherlands: APress, Apr. 2019. ISBN: 978-1-4842-4479-1.

[WA19] Andrew Waterman and Krste Asanović. "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA". In: (Dec. 2019). Ed. by Andrew Waterman, Krste Asanović, and Sivive Inc. URL: https://riscv.org/technical/specifications/.

[Led20] Jim Ledin. *Modern Computer Architecture and Organization*. Birmingham, UK: Packt Publishing, Apr. 2020. ISBN: 978-1-83898-439-7.

[Hsu21] Min-Yih Hsu. *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries*. Birmingham, UK: Packt Publishing, Apr. 2021. ISBN: 978-1-83882-495-2.

[Jef21] Clinton L Jeffery. *Build Your Own Programming Language*. Birmingham, UK: Packt Publishing, Nov. 2021. ISBN: 978-1-80020-480-5.

[McN21] Timothy Samual McNamara. *Rust in Action*. In Action. New York, NY: Manning Publications, Aug. 2021. ISBN: 978-1-61729-455-6.

[CCW22] Kito Cheng, Jessica Clarke, and Andrew Waterman. *RISC-V ABIs Specification*. Nov. 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc.

[Lu+22] H.J. Lu et al. *System V Application Binary Interface. AMD64 Architecture Processor Supplement*. Version 1.0. Dec. 2022. URL: https://gitlab.com/x86-psABIs/x86-64-ABI.

[Ros22] *WebAssembly Core Specification*. Version 2.0. W3C, Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/.

[Sen22] Kumar N Sendil. *Practical WebAssembly*. Birmingham, UK: Packt Publishing, May 2022. ISBN: 978-1-83882-800-4.

A. Complete Grammar of rush in EBNF Notation

grammar.ebnf

```
Program = { Item } ;

Item               = FunctionDefinition | LetStmt ;
FunctionDefinition = 'fn' , ident , '(' , [ ParameterList ] , ')'
                   , [ '->' , Type ] , Block ;
ParameterList      = Parameter , { ',' , Parameter } , [ ',' ] ;
Parameter          = [ 'mut' ] , ident , ':' , Type ;

Block = '{' , { Statement } , [ Expression ] , '}' ;
Type  = { '*' } , ( ident
                  | '(' , ')' ) ;

Statement    = LetStmt | ReturnStmt | LoopStmt | WhileStmt | ForStmt
             | BreakStmt | ContinueStmt | ExprStmt ;
LetStmt      = 'let' , [ 'mut' ] , ident , [ ':' , Type ] , '='
             , Expression , ';' ;
ReturnStmt   = 'return' , [ Expression ] , ';' ;
LoopStmt     = 'loop' , Block , [ ';' ] ;
WhileStmt    = 'while' , Expression , Block , [ ';' ] ;
ForStmt      = 'for' , ident , '=' , Expression , ';' , Expression
             , ';' , Expression , Block , [ ';' ] ;
BreakStmt    = 'break' , ';' ;
ContinueStmt = 'continue' , ';' ;
ExprStmt     = ExprWithoutBlock , ';'
             | ExprWithBlock , [ ';' ] ;

Expression       = ExprWithoutBlock | ExprWithBlock ;
ExprWithBlock    = Block | IfExpr ;
IfExpr           = 'if' , Expression , Block , [ 'else' , ( IfExpr
                                                          | Block ) ] ;
ExprWithoutBlock = int
                 | float
                 | bool
                 | char
                 | ident
                 | PrefixExpr
                 | InfixExpr
                 | AssignExpr
                 | CallExpr
                 | CastExpr
                 | '(' , Expression , ')' ;
PrefixExpr       = PREFIX_OPERATOR , Expression ;
InfixExpr        = Expression , INFIX_OPERATOR , Expression ;
(* The left hand side can only be an `ident` or a `PrefixExpr` with the `*`
↪  operator *)
AssignExpr       = Expression , ASSIGN_OPERATOR , Expression ;
CallExpr         = ident , '(' , [ ArgumentList ] , ')' ;
ArgumentList     = Expression , { ',' , Expression } , [ ',' ] ;
CastExpr         = Expression , 'as' , Type ;

ident = LETTER , { LETTER | DIGIT } ;
```

```
int   = DIGIT , { DIGIT | '_' }
      | '0x' , HEX , { HEX | '_' } ;
float = DIGIT , { DIGIT | '_' } , ( '.' , DIGIT , { DIGIT | '_' }
                                   | 'f' ) ;
char  = "'" , ( ASCII_CHAR - '\'
              | '\' , ( ESCAPE_CHAR
                      | "'"
                      | 'x' , 2 * HEX ) ) , "'" ;
bool  = 'true' | 'false' ;

comment = '//' , { CHAR } , ? LF ?
        | '/*' , { CHAR } , '*/' ;

LETTER      = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I'
            | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R'
            | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' | 'a'
            | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j'
            | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's'
            | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' | '_' ;
DIGIT       = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
            | '9' ;
HEX         = DIGIT | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a'
            | 'b' | 'c' | 'd' | 'e' | 'f' ;
CHAR        = ? any UTF-8 character ? ;
ASCII_CHAR  = ? any ASCII character ? ;
ESCAPE_CHAR = '\' | 'b' | 'n' | 'r' | 't' ;

PREFIX_OPERATOR     = '!' | '-' | '&' | '*' ;
INFIX_OPERATOR      = ARITHMETIC_OPERATOR | RELATIONAL_OPERATOR
                    | BITWISE_OPERATOR | LOGICAL_OPERATOR ;
ARITHMETIC_OPERATOR = '+' | '-' | '*' | '/' | '%' | '**' ;
RELATIONAL_OPERATOR = '==' | '!=' | '<' | '>' | '<=' | '>=' ;
BITWISE_OPERATOR    = '<<' | '>>' | '|' | '&' | '^' ;
LOGICAL_OPERATOR    = '&&' | '||' ;
ASSIGN_OPERATOR     = '=' | '+=' | '-=' | '*=' | '/=' | '%='
                    | '**=' | '<<=' | '>>=' | '|=' | '&=' | '^=' ;
```