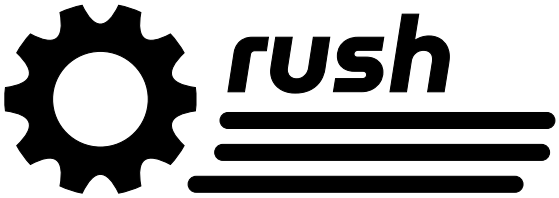
Carl-Fuhlrott-Gymnasium
Wuppertal

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

March 20, 2023

Abstract

Timestamp: Mon Mar 20 09:19:11 UTC 2023 | Commit: 80d4c35

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 one interpreter, one virtual machine, one transpiler, and five 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 an interpreted programming language. Here, we will differentiate between a *tree-walking interpreter* and a *virtual machine*. Chapter 4 illustrates how compilation to high-level[1] targets 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[2] architectures 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 a conclusion on the topic of implementing a programming language.

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

[2] *Low-level* targets: specific to one physical target 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 by a computer, they are often falsely regarded as mysterious. 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 interprets the source code directly. Although compilers and interpreters share some of their core principles, their 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 understand 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. Implementation of an interpreter in a high-level language is often favorable 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 poject 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 of the next one. However, partitioning the compiler into as 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, we 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.

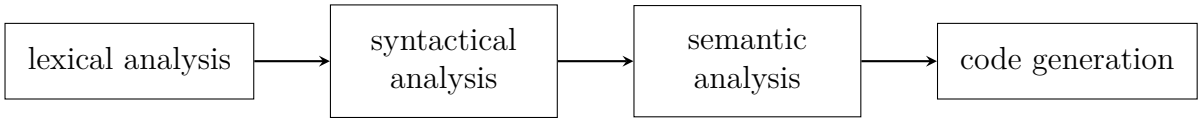


Figure 1.1 – Different steps of compilation.
[Wir05, pp. 6–7]

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*) of the language's vocabulary. 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 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 early optimizations.

4. Code generation traverses the tree generated by step 3 in order to generate 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. In this approach, the parser accesses the lexer directly, instructing it to return the next token as 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.

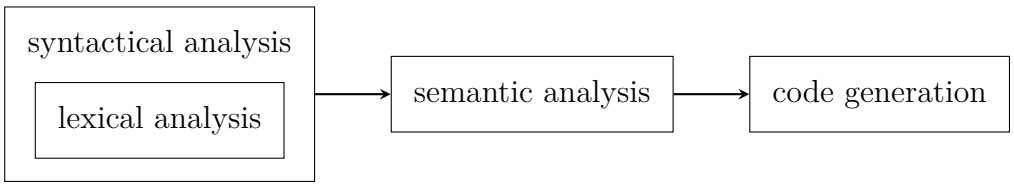


Figure 1.2 – Different steps of compilation (altered).

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 has a '`return`' statement, the body of the '`fib`' function contains no such statement. This is because blocks like the one of the function '`fib`' 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 it is not followed by a semicolon. If the input parameter '`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 using the specified exit code.

1.2.1. Features

As hinted previously, rush implements various features which are also found in most 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]. In order to provide abstractions, rush provides many features which are meant to increase developer productivity. For instance, a program

[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.

might use a '`loop`' to implement code repetition easily. For reference, Table 1.1 shows some features of the rush programming language.

Table 1.1 – Most important features of the rush programming language.

Name	Description	Example
unconditional loop	Executes code repeatedly.	`loop { }`
while-loop	Like '`loop`', checks a condition before each iteration.	`while x < 5 { }`
for-loop	Like '`while`', 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.	`1 + n; 5 ** 2`
prefix-expression	Performs mathematical and logical operations.	`!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`

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 later. Therefore, the rush programming language is designed in a way that promotes correctness and programmer efficiency. Table 1.2 provides an overview about all the types which are present in rush.

Table 1.2 – Data types in the rush programming language.

Notation	Example	Description
‘`int`’	`let a: int = 0;`	Values of the *integer* type can represent 64-bit signed integers, meaning $[-2^{63};\ 2^{63}-1]$.
‘`float`’	`let b: float = 3.14;`	Values of the *floating-point* type can represent 64-bit double precision floating-point numbers.
‘`bool`’	`let c: bool = true;`	Values of the *boolean* type can represent either ‘`true`’ or ‘`false`’.
‘`char`’	`let d: char = 'a';`	Values of the *character* type can represent any ASCII character in the range $0 \leq x \leq 127$.
‘`()`’	`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 and is also 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.

While the first four types are self-explanatory, the last two types 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 the rush code in Listing 1.1, the ‘`main`’ function also returns the unit type. For instance, a function ‘`foo`’ can be defined like ‘`fn foo() -> () {}`’, as well as ‘`fn foo() {}`’. Therefore, manually specifying that a function returns the unit type is redundant. The *never* type is used in order to denote that an expression or statement interrupts the normal control flow, so that the construct never yields a value. Because the built-in ‘`exit`’ function terminates a program, its result will never be of relevance. 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, meaning syntax like ‘`let a: ! = exit(0)`’ is illegal as it would not create any practical benefits.

Table 1.3 – Lines of code of the project’s components in commit ‘`e0fd6f7` ’.

Component	LoC
Lexer / Parser	2737
Tree-walking interpreter	578
VM compiler / runtime	1286
WASM compiler	1584
LLVM compiler	1450
RISC-V compiler	2276
x86 compiler	2751

In the rush project, most of the previously presented stages of compilation are implemented as their own individual code modules. This way, each component of the programming language can be developed, tested, and maintained separately. In the git commit ‘`e0fd6f7` ’, the entire rush project consisted of 17455 lines of source code[4]. 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, five compilers, one interpreter, and several other tools like a language server for editor support. Table 1.3 shows the lines of code of

[4]Every specification of line count does not count blank lines and comments.

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 1,500 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 become apparent.

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 has to first *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 of type 2 or 3 in the Chomsky hierarchy, that is *context-free* and *regular* languages, whereas natural languages often are type 1 or 0 [Wat17, p. 24]. 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 we use 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. **TODO: example for type 1 language '([])'? TODO: explain start symbol? TODO: formal definition of grammars $G = (N, T, S, P)$?**

```
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 there is no symbol following the dash in this case, 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

[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.

the set of all terminal symbols used in context-free grammar rules and the set of regular grammar rules. For the language defined in Listing 2.1 these are the five operators '`+`', '`-`', '`*`', '`/`', '`**`', and the 'integer' non-terminal.

crates/rush-parser/src/lexer.rs

```
10 pub struct Lexer<'src> {
11     input: &'src str,
12     reader: Chars<'src>,
13     location: Location<'src>,
14     curr_char: Option<char>,
15     next_char: Option<char>,
16 }
```

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

The specifics of implementing a lexer are not explored in this paper, but 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 character, as seen in Listing 2.2. For clarity, Table 2.1 shows the values of '`curr_char`' and '`next_char`' during processing of the input '`1+2**3`'. Here every row in the table represents one point in time displaying the lexer's current state.

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, and 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 be simply ignored and skipped during lexical analysis, or get their own token kind and be treated similar to string literals.

```rust
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.

TODO: maybe mention having spans inclusive or exclusive

2.1.3. Constructing a Tree

The parser uses the generated tokens in order to construct a tree representing 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, intricate linters, and analyzers, 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 7. 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.

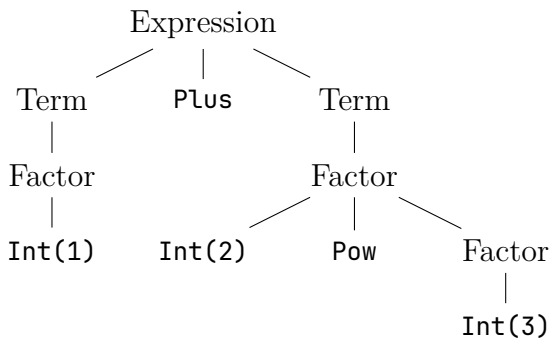


Figure 2.1 – Abstract syntax tree for '`1+2**3`'.

However, not every parser is the same, and there are a number of different strategies for implementing one, depending on the requirements of the language. These strategies are categorized into *top-down parsers* and *bottom-up parsers*. The main difference between them is the kind of tree traversal they perform. Top-down parsers construct the syntax tree in a pre-order manner, meaning a parent node is always constructed before its children nodes. **TODO: well, *construction* uses post-order, but the method calls are pre-order** Hence, the syntax tree is constructed from top to bottom, starting with the root node. Bottom-up parsers instead perform post-order traversal. That way, all child nodes are processed before their parent node. This results in the tree being constructed from the leafs at the bottom upwards to the root. **TODO: a non-exisiting tree cannot be traversed**

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 of reading the tokens, 'L' being from left to right, and the derivation they use. 'L' is the leftmost derivation and 'R' the rightmost derivation. **TODO: what are derivations? + citation can be found here [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 two or one wide window 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].

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.

An example for LR parsing is the *shift-reduce* parsing approach, which is later outlined on page 14. 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. **TODO: explain why**

```
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.

Most LL parsers, including rush's, are *recursive-descent* parsers. Implementation of such a recursive-descent 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 7, 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. **TODO: explain some of them TODO: struct signature example?**

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 from Listing 2.4. However, when comparing the grammar to 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 the order of evaluation 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 '3'. Now consider '`1**2**3`'[3]. Here, the '`2**3`' should be evaluated first, and afterwards '1' should be raised to 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. However, there is also a technique called *Pratt parsing*[4] that allows slightly modified recursive-descent LL parsers to parse such languages, given a map from tokens to precedences and their 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.

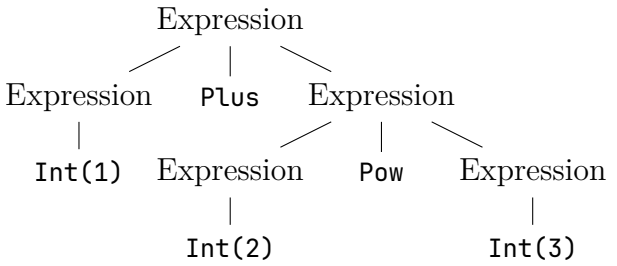


Figure 2.2 – Abstract syntax tree for '`1+2**3`' using Pratt parsing.

Pratt Parsing

As the rush parser makes use of Pratt parsing, most of 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`' enum. 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 the precedences are, the deeper in the tree the resulting expressions will be, and the earlier they 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 matches on the current token kind to decide which expression to parse and stores the result in the '`lhs`'[5] variable. Afterwards it checks whether the left precedence of the now current token is bigger 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 that case, this check will only fail when the whole expression is over, as every non-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 the '`lhs`'. Afterwards, '`lhs`' is overwritten with the

[3] '`**`' is the power operator here, so the input would be written as 1^{2^3} using mathematical notation.
[4] First introduced by Vaughan Pratt in 1973 [Pra73].
[5] 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 – Token precedences in rush.

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 precedence for the input '`(1+2*3)/4**5`', which represents the mathematical expression $\frac{1+2\cdot3}{4^5}$. 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`' 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

crates/rush-parser/src/parser.rs

```rust
fn grouped_expr(&mut self) -> Result<'src, Spanned<'src, Box<Expression<'src>>>> {
    let start_loc = self.curr_tok.span.start;
    // skip the opening parenthesis
    self.next()?;

    let expr = self.expression(0)?;
    self.expect_recoverable(
        TokenKind::RParen,
        "missing closing parenthesis",
        self.curr_tok.span,
    )?;
    // ...
}
```

Listing 2.7 – Pratt-Parser: implementation for grouped expressions.

crates/rush-parser/src/parser.rs

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

Listing 2.8 – Pratt-Parser: implementation for infix expressions.

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, them being '21' for the '`Star`' token and '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 in 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 should then become clear how Pratt parsing achieves parsing with precedences without them being encoded in the Grammar, and with that, the node types.

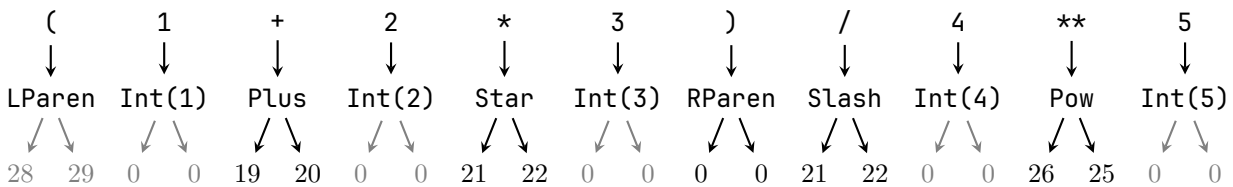


Figure 2.3 – Token precedences for the input '`(1+2*3)/4**5`'.

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 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 from 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. **TODO: cite TODO: mention 'nom'**

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 the grammar cannot specify how the programming language should behave. Therefore, 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 had to be declared before it could be used. Defining the semantic rules of a programming language is often a demanding task since not all requirements are clear from the beginning. Furthermore, the semantic rules of a programming language can usually not be defined formally without significant effort [Hol12, pp. 5f]. a language designer often chooses to write their specification in a natural language, meaning Chomsky type 0 [Wat17, p. 23]. 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 rational to implement a separate compilation step which is responsible for validating the source program's semantics. Among other checks, the so-called *semantic analyzer*[6] 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 at compile time. For obtaining type information, the abstract syntax tree of the source program has to be traversed so that

[6] Later referred to as "analyzer".

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.

In order to understand why type information is required at compile time, Listing 2.9 should be considered. The code in this Listing 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.

```
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.

The analyzer will first check if 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 checks 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 the lines 2 and 3. Since '`let`' statements are used to declare new 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. This information will become useful when line 4 is considered. Here, 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 a legal infix-expression 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 declarations, not just the ones of built-in functions. Since the result of the addition is also an integer, the analyzer accepts this program since both its syntax and semantics are valid.

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 at compile time. However, determining these types requires a complete tree traversal of the operand expressions. Due to the recursive design of the abstract syntax tree, 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. Therefore, implementing type determination in each backend individually would enlarge the compiler source code, thus making it harder to understand. In order to prevent this duplication, rush's semantic analyzer also annotates the abstract syntax tree 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 example from above will now be considered. However, before we can examine how some methods of the analyzer work, its most important struct fields should be discussed first.

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 semantic analyzer. The field '`functions`' in line 13 associates a function name to the function's signature. Therefore, if a function is called at a later point in time, the analyzer checks whether the function exists and 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 diagnostics has a severity, such as *warning* or *error*, for instance. 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 would occur when another 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 correctly. If a function is called from another function, it is marked as used. However, if the call originates from the same function, it only implements recursion without being used from the outside. 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 > 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. After the analyzer has traversed the loop's body, the counter is decremented again. This field is implemented as a counter rather than a boolean in order to support nested loops.

Now that important attributes have been highlighted, we can now consider the example from Listing 2.9. First, the analyzer traverses and examines all functions and their bodies. For every rush function, the analyzer invokes an internal method responsible for validating functions. Among other tasks, this method inserts a new entry into the '`functions`' hashmap, associating the function's name with its signature so that it can be used for validating arguments later. Because a '`main`' function is mandatory in every rush program, the analyzer simply checks that the '`functions`' hashmap contains an entry for the '`main`' function after every function has been traversed. The code in 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`'. Therefore, it is responsible for analyzing and annotating the definition of a function. 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-clause in line 405 checks if the node's '`params`'

vector contains any items. If the vector contained any items, the main-function's declaration would include parameters. If this was the case, the analyzer would generate an error message stating that the main-function must not take any parameters. However, error handling in the analyzer has not been discussed so far. According to the method's signature, it cannot return any errors. Instead, the '`self.error`' method is invoked in line 406. This method uses information about the error to be generated in order to push a new '`Diagnostic`' into the '`diagnostics`' vector.

Secondly, in rush, the main-function's return-type always has to be the unit type. The analyzer validates this constraint in the lines 422 and 423. The first if-clause checks whether the function contains a manually defined result-type. In rush, every function definition without an explicitly defined result type defaults to the unit type. Therefore, the inner if-clause is only executed if the user has manually specified a result type of 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 in line 424 which describes the issue.

After the signature of the main-function has been validated, the method begins traversal of the function's body. In line 490, the '`self.block`' method of the analyzer is invoked using 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 bound to a variable called '`block`'. This variable represents the completely analyzed and annotated function body. Lastly, in the lines 535–541, an '`AnalyzedFunctionDefinition`' is returned. Here, the function's attributes, like its analyzed parameters, return-type, name, and body are specified. The other variables, like '`params`', have been defined in the parts of the code which have been commented for better overview.

During the traversal of the main function's body, the analyzer encounters two '`let`' statements in line 2 and 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 – Beginning of 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 (line 644). Even though the content 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 inside the struct includes whether the variable was declared as mutable. If a variable is mutable, 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 is 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 as a condition inside an if-clause. If the insertion returns '`true`', the variable's name was already present in the current scope and its previous 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.

It is now apparent that the analyzer uses type information of expressions on many occasions. However, how type determination and annotation works in the analyzer has not been discussed yet. In order to get an understanding of these processes work, the traversal of expressions is to 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         }
         // ...
1048     res
1049 }
```

Listing 2.14 – Analysis of expressions during semantic analysis.

This method consumes a non-analyzed expression and transforms it into an analyzed version of itself. For simple types of expressions, like integers, floats, or booleans, further analysis is omitted. Since these types of expressions are constant, the method can directly return an analyzed version of the expression. For more complex types of expressions, like if-expressions, this method calls the appropriate method responsible for analyzing this type of expression.

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 the 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. For instance, since grouped expressions contain another inner expression, another grouped expression inside a grouped expression is a legal construct in rush. Therefore, it is possible that the '`expression`' method calls itself multiple times recursively. Most of the other tree traversing methods implement a similar recursive behavior as most types of AST nodes may contain themselves at some point.

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 – Obtaining the type of expressions.

Since tree traversal and analysis has now been discussed in general, the question of how types are accessed and saved in the annotated syntax tree remains. The code in 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`' enum 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 a constant integer expression, the resulting level of pointer indirection is zero. Therefore, this method is able to return the types of simple constant expressions with no additional effort. 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 and 143. During analysis of block expressions, the responsible function checks if the block contains a trailing expression, and if it does, the result type of the block expression is identical to the one of its trailing expression. In line 144, the result type of a grouped expression is obtained by calling the '`result_type`' method recursively. By using the previously described method, the analyzer is able to get 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 somehow continue the tree traversal. Otherwise, only one error could be reported at a time 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, meaning that it does not exist. 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.

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 on each iteration. Among others, these include that there is an argument with the matching type for each declared 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's hashmap. If an unknown variable was provided, the lookup in the hashmap would yield no value, thus causing an error message to be generated at this point. Because the type of a non-existent variable is unknown, the placeholder '`Unknown`' type would be used to prevent cascading errors.

Because the two variables should be added, the method '`infix_expr`' of the analyzer is

called. This method is responsible for analyzing any kind of infix expression like '`n || m`'. This method validates 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 – Validation of argument type compatibility in the analyzer.

Listing 2.16 shows a part of the '`arg`' function which is responsible for validating that a function call argument is compatible with the declared parameter. This code will produce an error message if the type of the call argument deviates from the one of the declared parameter. 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 displays a 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 overview.

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 this program and returns its syntax-tree with type annotations.

2.2.3. 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 rational 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 often 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

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

Listing 2.17 – Method for determining whether an expression is constant.

In order to make such optimization possible, each expression node in the analyzed AST has a method named '`constant`'. The method shown in Listing 2.17 is responsible for determining whether an expression is constant. It returns '`true`' if the expression is a constant integer, float, boolean or char. Other types of expressions, such as a call-expression 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.

```
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. The Listing 2.18 displays part of a rush program which uses a '`while`' loop even though a '`loop`' would suffice. A '`loop`' implementation is more efficient since the condition check is omitted before each iteration. However, this is only the case because the condition of the loop is a constant '`true`'. If the analyzer detects such a scenario after a while-loop was analyzed, the output node will be converted into a conventional loop. Detection of this scenario is implemented in line 855 of Listing 2.19. In line 853, the match arm is called if the preceding loop will never iterate. If this is the case, the entire loop is deleted 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 into the output program. Furthermore, the resulting output code will also be of slightly smaller 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(&mut self, node: WhileStmt<'src>) -> 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 – Loop transformation in the analyzer.

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 ones discussed in the examples from above. In order to understand how early optimizations may impact the resulting AST, the two trees in Figure 2.4 should be considered.

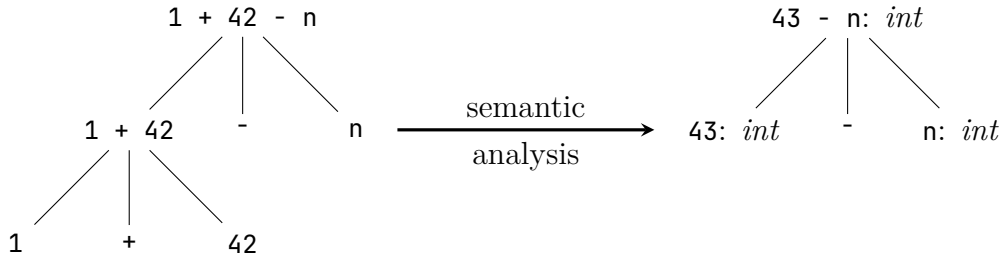


Figure 2.4 – How semantic analysis affects the abstract syntax tree.

Both trees in Figure 2.4 represent the rush expression '`1 + 42 - n`'. The left tree could have been generated by the parser, closely representing the structure of the code. The right tree shows the same tree after it has been transformed by the analyzer. Most notably, the sub-expression '`1 + 42`' has been transformed into '`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. **TODO: mention slowness 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. **TODO: are rush's scoping rules already explained? TODO: is runtime / compile time already explained? TODO: are globals already explained?** Why the '`Value`' is wrapped inside an '`Rc<RefCell<_>>`' here 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 unnecessarily cloning.

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

```rust
type ExprResult = Result<Value, InterruptKind>;
type StmtResult = Result<(), InterruptKind>;
type Scope<'src> = HashMap<&'src str, Rc<RefCell<Value>>>;

#[derive(Debug, Default)]
pub struct Interpreter<'src> {
    scopes: Vec<Scope<'src>>,
    functions: HashMap<&'src str, Rc<AnalyzedFunctionDefinition<'src>>>,
}
```

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 '`()`'[1] on success or an '`InterruptKind`' on failure. Expressions use the same error type, but return a '`Value`' on success, holding the evaluated result. 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.

[1] Rust's 'unit' type, a type containing no data, comparable to 'void' in other languages. Its notation describes an empty tuple.

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

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 a '`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, '`Exit(i64)`', is constructed by rush's built-in '`exit(int)`' function and terminates the program. They cause the interpreter to backtrack all the way back 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[2], since it can already be a value of any type. The rush interpreter only makes use of the analyzer's guarantees of result types, which could not be done with dynamic typing.

3.1.1. Implementation

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

Listing 3.3 displays the public '`run`' method of the interpreter that serves as its entry point. Since the order of functions in rush does not matter and a function can be called before its definition, the interpreter must first populate its '`functions`' map, before any function code is run. Afterwards, 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

[2] 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.

interpreter either returns a produced runtime error, an exit code caused by a rush call to ‘`exit`’, or the exit code ‘0’ indicating success.

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

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’s body is then initialized and filled with the passed arguments. Afterwards, the function’s block is evaluated by calling the ‘`visit_block`’ method in a scoped environment. If the block returns an interrupt that is partial, 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.

3.1.2. How the interpreter executes a program

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

let mut global = 40;

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

Listing 3.5 – Example rush 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.

First, 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 '`visit_block`' method then iterates over the statements contained in the block, and evaluates each one in order with 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 '`+`' sign on their own and then adding the results.

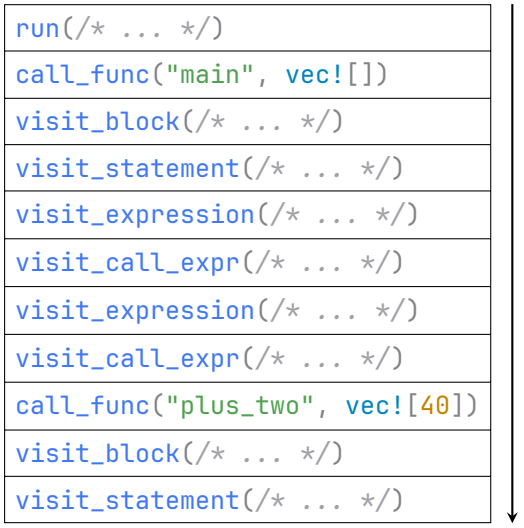


Figure 3.1 – Call stack at the point of processing the '`return`' statement.

Following that, an '`InterruptKind::Return(_)`' is constructed, holding the computed sum. By making use of Rust's '`?`' operator[3], the interrupt causes early returns in all function calls walking up the call stack to '`call_func`'. Thus, the statement '`global += 4;`' is never reached. 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 language implementations, 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 method of having pointers to values besides their reference system. The '`Rc<_>`' type, short for *reference counter*, stores its contained value on the heap[4] and provides shared access to it, without any particular owner. The contained value is freed as soon as no more references to it exist. This makes it ideal for implementing pointers in a tree-walking interpreter.

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<_>`'

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

[4] **TODO: what is the heap / explained in chapter 5?**

implements so-called *interior mutability*[5] 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.

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 a virtual machine to a tree-walking interpreter, we have implemented a virtual machine backend which can execute rush programs.

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 often includes additional devices like the computer's display or its disk, whereas in this context, the term 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]. Because the ALU is designed to perform logical and mathematical operations as fast as possible, it lacks the ability to fetch and execute instructions from memory directly, presumably to keep its implementation simple. Therefore, the processor contains a control unit which manages 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 the individual steps of the fetch-decode-execute cycle.

- **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 of instruction in mind. For instance, the control unit may invoke the ALU in order 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 an isolated manner because instructions are executed in a sequential order. This means that the execution of the instruction i is delayed until

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

execution of $i - 1$ has completed [Led20, pp. 208-209]. More sophisticated processors often use many performance-enhancing techniques like *SIMD processing* or *simultanious multi-threading* [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 often 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, we will present a compiler targeting it in Chapter 4 on page 43. For compiler writers, implementing *register allocation* is often a demanding task. This problem is described thoroughly in Chapter 5. Since register allocation is not required in a compiler targeting stack-based machines, its implementation is often significantly easier compared to a compiler targeting a register-based machine. 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 usually requires a lot of seemingly redundant push and pop operations which could have otherwise been omitted.

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 a 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.

Listing 3.6 displays the struct definition of the rush VM. The '`stack`' field in line 18 saves the main stack for temporary values using Rust's '`Vec`' type. 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.

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.

The instructions in Listing 3.11 on page 33 can be interpreted by the rush VM. Programs for the rush VM are always partitioned into functions which each represent 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.

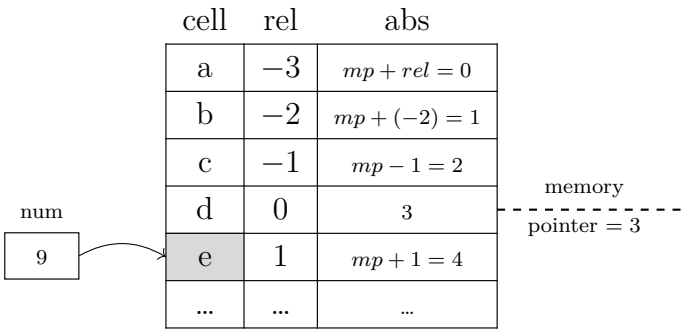


Figure 3.2 – Linear memory of the rush VM.

The first block of instructions is called '*prelude*' since 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 in the program. If global variables were present in the example, the prelude would contain the instructions used for initializing them. If the prelude was omitted, the main function would instead contain these instructions since it is executed at program start. However, recursion of the '`main`' function is legal 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 run only once.

Linear memory in the VM is represented as an array of runtime values of variables. Each storage cell can be accessed by using 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 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 which represents the

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.

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.

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

Listing 3.7 – Minimal Pointer Example in rush

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' (*relative 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.

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

Listing 3.8 – VM Instructions for the minimal Pointer Example

The first instruction 'setmp' (*set memory pointer*) 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 variable immediate*) instruction pops the topmost 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 instructions. Here, the relative address 0 is used since the 'svari' instruction has previously saved 'num' at this location. In line 6, the '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. Now, the absolute address of 'num' is saved 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 declared.

To summarize the previous paragraph, this example uses relative addressing in order to declare local variables of a function. However, absolute addressing is also used when variables are referenced in order to create pointers. Therefore, each addressing mode serves a separate and important purpose.

3.2.4. How the Virtual Machine Executes A rush Program

```
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.

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 33 should be considered. 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, we must consider the call stack of the rush VM. 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 VMs 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.

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`'.

prelude	main	...
$fp = 0$	$fp = 1$	
$ip = 1$	$ip = 0$	

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`'[6] instruction, it should leave the current function immediately.

[6] Short for "return"

However, it should also know where to resume its fetch-decode-execute cycle. For this, the VM simply pops the top element from 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. This way, function calls are implemented in a simple but robust manner.

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 declarations in its body. However, this behavior is correct since function parameters count as variable declarations. 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`' instruction. It is then consumed 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 to push its value on 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.

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

Listing 3.11 – VM instructions matching the AST in 3.4.

In line 14, the constant value 0 is pushed onto the stack. Next, the '`eq`' instruction pops two elements from the stack in order to test them for equality. Then, the result of the equality test is pushed on 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`' 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 on the stack. Next, the '`sub`' instruction pops the first two elements from 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 previously explained '`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. Next, the '`setmp`' instruction in line 24 decrements the memory pointer

in order to deallocate used memory[7]. At the end of a function, the memory pointer is always decremented by the amount it was incremented at the beginning of the function. By deallocating the now unused memory, the compiler prevents the code from leaking 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`' is 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`' 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. Since we have covered what the instructions in the lines 24 and 25 accomplish, we can summarize that the function returns the value 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, we will explain how the fetch-decode-execute cycle works in the VM. The code in Listing 3.12 displays the '`run`' method of the rush VM.

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

```
168                 return Ok(code);
169             };
170         }
171         Ok(0)
172     }
173
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)? {
```

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 169. The condition of the loop checks that 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 170, 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 '`self.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 an *integer-overflow* fault. 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

[7]Since the operand is negative, the instruction decrements '`mp`'

possible enum variants, the fetch-decode-execute cycle continues as usual. However, one cannot observe how the instruction pointer is incremented, therefore suggesting that the VM stays in an endless loop without ever progressing to the next instruction. In order to answer the final question of how the current instruction is executed, and the instruction pointer is incremented, we will now examine the code in Listing 3.13.

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

```
181         };
182     }
183
184     Ok(0)
        // ...
191         Instruction::Drop => {
192             self.pop();
193         }
194         Instruction::Clone => self
            // ...
261             self.call_stack.pop();
262         }
263         Instruction::Cast(to) => {
            // ...
272             self.push(top.neg())?;
273         }
274         Instruction::Not => {
275             let top = self.pop();
276             self.push(top.not())?;
            // ...
357         }
358         Instruction::BitXor => {
359             let rhs = self.pop();
360             let lhs = self.pop();
```

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

The code in Listing 3.13 displays parts of the '`run_instruction`' method. This method mainly consists of a match-expression that determines which code to run, depending on the current instruction. Here, the implementation of several instructions is visible. In line 183, the '`Nop`' instruction is matched. It is apparent that there is no instruction-specific code executed in this case since "nop" stands for "no operation". Therefore, the VM will ignore any '`Nop`' instructions it encounters. Next, in line 184, the code for the '`Push`' instruction is displayed. This instruction pushes its first operand on the stack at runtime by calling the internal helper method '`self.push`'. Therefore, this method pushes the provided parameter on the '`stack`' field of the VM, first validating 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 191, the code responsible for executing the '`Jmp`' instruction can be seen. This instruction only sets the instruction pointer to the target index specified in the first operand. Therefore, a jump involves very little overhead and is implemented using very little effort. It is to be noted that this instruction does not perform any bound checks, therefore allowing illegal addresses. For some special instructions, such as this one, the instruction pointer should not be incremented since it would interfere with the jump. Since the instruction pointer is incremented at the end of the method, the '`return`' statement in line 193 is used to terminate this method early, preventing the increase of '`ip`' at the end.

Line 261 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. In line 272, the code responsible for executing the '`Add`' instruction can be observed. This instruction first pops two

elements from the stack since they represent the operands of the underlying mathematical computation. Then, the '`add`' helper function is called on the left-hand side of the computation. This helper function then performs the actual addition computation. 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. It is apparent that the execution of these instructions involves relatively little effort. Like hinted previously, after an instruction has been executed, '`ip`' is incremented in line 358. 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* step since it first matches (*decode*) and then interprets (*execute*) the current instruction. Surprisingly, the parts of the method shown in this example can all be understood using relatively little effort and show that the implementation of a VM is usually manageable. Now that we have explained how some important parts of the rush VM work, the question of how its input instructions are generated remains. Therefore, the compiler targeting the rush VM is presented in Section 4.1.1 on page 39.

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 interpreted. The code in Listing 3.9 displays a recursive function implemented in rush. Figure 3.4 displays a heavily simplified syntax tree representing the function displayed in Listing 3.9. 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 only direct child of the root node is the single if-expression present in the function's body. 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 of the body presents an inefficiency. Here, the inefficiency solely lies in the repeated tree traversal, not in the repetition introduced by an iterative or recursive algorithm.

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. The instructions in Listing 3.11 represent the program in Listing 3.9. 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 execute the '`rec`' function recursively. Since repeated traversal of the syntax tree is omitted, rush programs will run significantly faster using the VM compared to the tree-walking interpreter. Using the VM, executing the '`rec`' function using an input of $n = 1000$ took around 160 μs. However, executing the identical code using the tree-walking interpreter took around 427 μs[8]. The rush VM executed the identical code roughly 2.6 times faster than the tree-walking interpreter. However, the initial delay caused by compilation was not considered in this benchmark.

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

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 less 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 demanding 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 to run on a virtual machine can indeed be used successfully and on a larger scale. For instance, the *Java Virtual Machine* (*JMV*) 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, Chapter 1.2].

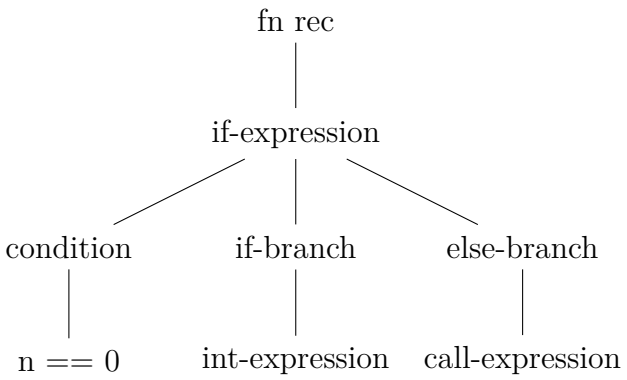


Figure 3.4 – Abstract syntax tree and VM instructions of a recursive rush program.

4. Compiling to High-Level Targets

In the previous chapter, we have learned how an interpreted programming language can be implemented. Another method of implementing a programming language is to create a compiler for this language. However, the question of how such a compiler works exactly still remains.

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. These individual functions often return some sort of value representing the translated node. Otherwise, each individual function may also insert generated instructions into an internal field of the compiler. Here, the newly generated instruction is inserted the output sequence of instructions. In this case, each function often 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, meaning compilers translating one high-level language into another one, each node-specific function often returns a tree node representing code in the output language.

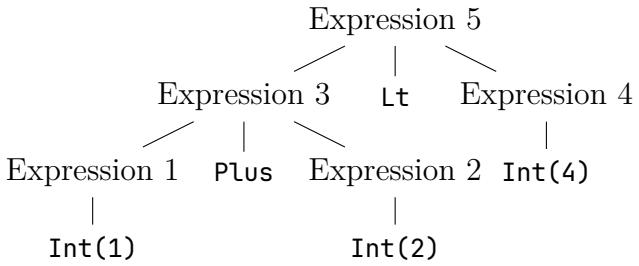


Figure 4.1 – Abstract syntax tree for '`1+2<4`'.

Listing 4.1 displays a simplified syntax tree of the rush expression '`1+2<4`'. The 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 add-expression '`1+2`'. However, due to post-order traversal, this node is not actually traversed 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 4' is traversed too. Since post-order traversal involves considering a root node after the traversal of its children, 'Expression 3' is traversed after its children have been considered. Here, the compiler considers the operand of the expression in order to generate the appropriate output instruction. Therefore, the instruction responsible for the addition is inserted after the Expression node 3 has been traversed. Since 'Expression 3' and its children are now completely traversed, and its output instruction has been inserted, the compiler now considered the right-hand side of the comparison. Here, the node 'Expression 2' only consists of the constant integer value 4. Now, all child nodes of 'Expression 5' have been traversed, thus the compiler now considers this node itself. Here, the compiler notices that the expression should check if the left-hand side is less than the right-hand side.

Therefore, the compiler inserts an instruction performing this comparison using the results of the left and right child as the operands. Since the compiler must be aware of the operands of the instruction, each method or function 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 containing the value of the expression at runtime. Therefore, such a describing entity can either be a register or a memory location describing the location a value which a node may yield. By returning information like this, a root node will have information about its children after they have been traversed. A root node 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 line 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 line 1 and 2 since their tree nodes were traversed first. Furthermore, the instruction uses register '`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 constant integer value 4 is assigned to the register '`r3`'. Lastly, the comparison instruction 'lt' is inserted using 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.

4.1.1. The Compiler Targeting the rush VM

Since the rush VM interprets instructions directly, there must be a compiler targeting its architecture. For this purpose, we have implemented a compiler translating rush source code into instructions which can be understood by the VM. 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.

This method differentiates between the '`Or`' / '`And`' operators and other possible operators since the former require some extra code to be emitted. However, due to simplicity, 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 matching instruction representing the infix-operator is inserted. The final instruction is generated by a helper function 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

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 => {
          // ...
536         }
537         op => {
538             self.expression(node.lhs);
539             self.expression(node.rhs);
540             self.insert(Instruction::from(op));
541         }
542     }
543 }
```

Listing 4.2 – Compilation of infix-expressions targeting the VM.

rush VM implements one instruction for each infix-operator. Since this VM implements all features of the source language, complex hacks for implementing some 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 – Compilation of expressions targeting the VM.

The code in Listing 4.3 shows parts of the '`expression`' method of the rush VM compiler. When we examine the method's signature, it becomes apparent that it consumes an '`AnalyzedExpresssion`'. However, the method does not return anything which represents the runtime value of the expression. This is possible because the results of the expressions displayed in the snipped are pushed onto the stack directly. In line 447, the code for translating a constant integer expression can be seen. Here, a push instruction using the constant int value as its operand is used. By pushing the values of expressions onto the stack directly, most tree traversing 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 contains 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. In line 457, the method for compiling call-expressions would be invoked. Similar to the '`expression`' method in the analyzer, this method also dispatches the traversal of more complex expressions to specialized methods in order to keep this method simple.

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 following mathematical term: n^m. Since many real architectures lack such a power instruction, most other rush compilers demanded implementation of special edge-cases in order to make compiling power-expressions feasible. On the opposite, implementing a rush compiler targeting the VM has proven to be less demanding since the VM supports a power-instruction. Furthermore, the VM also includes an '`exit`' in-

struction which terminates the fetch-decode-execute cycle instantly. These examples showed 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. This overhead is eliminated by the assembler during assembly of a program. Since the assembler performs these lookups during translation, the CPU does not have to deal with label lookups at runtime. Like seen in the previous examples, jumping VM instruction require the exact index of the target instruction as their operands. Therefore, the exact target index to which the instruction should jump must be known. To illustrate this issue, we will consider how loops are implemented in the VM. The rush code in Figure 4.2 presents a program containing a loop. In the loop's body, the variable '`n`' is incremented by 1. Next, the '`break`' keyword is used to terminate loop execution. Therefore, the total amount of iterations is 1.

```
1  fn main() {
2      let mut n = 0;
3      loop {
4          n += 1;
5          break;
6      }
7  }
```

```
1   setmp 1
2   push 0
3   svari *rel[0]
4   push *rel[0]
5   clone
6   gvar
7   push 1
8   add
9   svar
10  jmp 11
11  jmp 3
```

Figure 4.2 – Representation of loops in the VM.

The rush VM instructions of the '`main`' function are displayed on the right side of Figure 4.2. Here, lines 2 and 3 are responsible for declaring a variable named '`n`'. The instructions in the lines 4–9 are used to increment '`n`' by 1. A new instruction which we have not covered so far is the '`clone`' instruction. This instruction *clones* 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 variable.

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 this instruction 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. This instruction 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 known before the target instruction is inserted. The code in Listing 4.4 displays a part of the method responsible for compiling loops for the rush VM.

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

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

```
320 fn loop_stmt(&mut self, node: AnalyzedLoopStmt<'src>) {
321     // save location of the loop head (for continue stmts)
322     let loop_head_pos = self.curr_fn().len();
323     self.loops.push(Loop::default());
        // ...
331     self.block(node.block, true);
        // ...
337     self.insert(Instruction::Jmp(loop_head_pos));
338
339     // correct placeholder `break` / `continue` values
340     let loop_ = self.loops.pop().expect("pushed above");
341     let pos = self.curr_fn().len();
342     self.fill_blank_jmps(&loop_.break_jmp_indices, pos);
343     self.fill_blank_jmps(&loop_.continue_jmp_indices, loop_head_pos);
344 }
```

Listing 4.4 – Implementation of loops in the rush VM compiler.

previously defined and species the index of the first instruction of the loop's body. Therefore, this inserted instruction performs continuous jumps, thereby introducing the repetition for which the loop is desired.

In line 340, the top loop is popped from the '`loops`' stack. In line 322, this loop was previously pushed onto the identical stack. This stack is an internal field used by the compiler in order to save information about loops. The top item on this stack 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 inside the loop's body. For instance, if the compiler encounters a '`break`' statement, the code in Listing 4.5 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 4.5 – Compilation of '`break`' statements in the rush VM compiler.

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. Therefore, at the end of the compilation of a loop's body, there will be a list containing the indices of instructions whose target indices need to be adjusted. In line 342 of Listing 4.4, the '`self.fill_blank_jmps`' method is used to set the target indices of the specified jump-instructions to '`pos`'. We will omit the explanation of this method because it only iterates over the passed list of indices, replacing the target of the jump-instruction at the current index during the process.

As a conclusion, design, and implementation of the compiler targeting the rush VM has

presented itself as a reasonable task. Altering the target architecture to mitigate difficulties which occurred during the implementation of the compiler was often extremely helpful. Therefore, compared to the rush compiler targeting RISC-V, implementation of this compiler was significantly simpler. Furthermore, the rush VM uses a stack-based design which made implementing its compiler less demanding as well.

4.2. Compilation to WebAssembly

The first external compilation target[1] presented here is *WebAssembly*, or *WASM* for short. **TODO: research origins and goals of WASM and explain them here, [Ros22, Section 1.1.1]** 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. Most modern browsers include such a WebAssembly runtime, but there are also standalone ones, for example *wasmtime* and *wasmer* **TODO: cite**.

4.2.1. 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]. Most often, 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 4.6 and Listing 4.7 on page 44 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 no matter the integer's sign. 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 run in. WebAssembly is *stack based* and has one primary stack each instruction operates on. The first two instructions of the '`addTwo`' function retrieve the local variable of the given index and push its value to 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,

[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 4.6 – 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 4.7 – Simple WebAssembly module in binary representation.

which pops the top two elements off the stack and pushes the sum back on to it. The return value implicitly 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 in Listing 4.7 is to be considered. A WebAssembly binary file always starts with the four bytes '`00 61 73 6d`' 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 whole. 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 to 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 ended by 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 so on, 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]. The encoding details are not explained here however, and our implementation for this rush compiler simply uses a pre-existing crate[5] called "leb128".

4.2.2. The WebAssembly System Interface

Since WebAssembly itself does not provide any guarantees about the runtime environment, it does not provide 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

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

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

WebAssembly module to, e.g, write to a console and exit with a specific exit code. Both wasmtime and wasmer implement the WASI interface. **TODO: citation for WASI?**

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.

4.2.3. Implementation

As indicated, the rush WebAssembly compiler directly targets the binary format. 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 4.8 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 4.8 – Definition of instruction opcodes.

The '`Compiler`' struct has a lot of fields for various purposes. Only a few are shown in Listing 4.9 and explained here. To begin, a few 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 **TODO: capitalization** 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 4.9 – The '`Compiler`' struct definition of the WebAssembly compiler.

Listing 4.10 contains the compiler's entry point function. Some details are left out, but essentially, it simply calls another method to compile the program itself, and then concate-

nates 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. **TODO: WASI citation** The '`Self::section`' helper function is used to add the section identifier, byte length, and element count in front of each section's contents.

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

```
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 4.10 – Entry point of the WASM compiler.

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. Afterwards, it calls '`function_definition`' for every defined function. This has to happen in these two steps, because rush allows a function to be called before its definition. **TODO: this was already explained in the tree interpreter section** 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. By the stack's nature, this will result in correct 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 already 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 interesting special case to highlight is the compilation of logical operators[6] in infix expressions. This comes down to the fact that in many

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

```
fn infix_expr(&mut self, node: AnalyzedInfixExpr<'src>) {
    match node.op {
        InfixOp::And => {
            self.expression(node.lhs);
            self.function_body.extend([
                // if lhs is not true
                instructions::I32_EQZ,
                instructions::IF,
                types::I32,
                // then return false
                instructions::I32_CONST,
                0,
```

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

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 clear by the left-hand side. Listing 4.11 shows the relevant part of the '`infix_expr`' method in the WASM compiler. At the start of this method, before the normal logic is reached, there are special cases for the '`&&`' and '`||`' operators. Shown here is only the code for the '`&&`' operator, but that for '`||`' works alike. It is evident that the operation is compiled as if it were an if-expression like '`if !lhs { false } else { rhs }`' where '`lhs`' and '`rhs`' stand for 'left-hand side' and 'right-hand side' respectively. Using this strategy, the expected behavior of lazy evaluation is achieved, unlike if it were simply compiling both sides and comparing the results with 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 weather 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 logic used for all other infix operators.

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

4.2.4. Example

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

Listing 4.12 – Example rush program.

Listing 4.12 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 4.13. 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 3. Below the 'main' function is the definition and export of blank memory in line 20 and line 23 respectively, as to conform with WASI's requirements. The 'main' function is also exported as '`_start`' in line 22 and also separately declared as the module's entry point in line 24, both two have this function be the one to start execution. Line 21 declares the global '`a`' as a mutable *i64* and sets the initial value of '2'.

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

```
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 4.13 – Commented compiler output of the rush program in Listing 4.12.

the smaller of the two for booleans. **TODO: was this already explained earlier?** 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`' 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[7] 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 back down to '0', effectively only interpreting the lower 32 bits of the 64-bit number. After this happened, the call to '`__wasi_exit`' can now be performed without problems.

One might notice the trailing '`unreachable`' instruction in line 22. The rush WebAssembly compiler inserts one such instruction after every expression that was analyzed to have the '`!`' type. As the name suggests, the instruction asserts to the WebAssembly runtime that this instruction is never reached. This is helpful in cases where such expressions are used inside other expressions, 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 modules for correct types before executing them.

[7] **TODO: what is zero-extension**

TODO: loops?

4.3. 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 in order to generate its target-specific code [McN21, p. 373]. Furthermore, the *Clang* C and C++ compiler uses 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]. A recent example of software developed by Apple which uses LLVM is the *Swift* programming language which is mainly used for developing IOS apps [Hsu21, preface].

4.3.1. 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 interact with which is usable by other software. Typically, a compiler frontend only analyzes the source program to create an AST. Next, a separate step of compilation invokes the LLVM framework which carries on from this point. This component traverses the AST and uses the LLVM API in order to construct an *intermediate representation* (IR) of the program. This way, the framework will be able to understand 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 4.3 shows how LLVM integrates into the previously discussed steps of compilation. Therefore, the framework represents the *backend* component of a compiler.

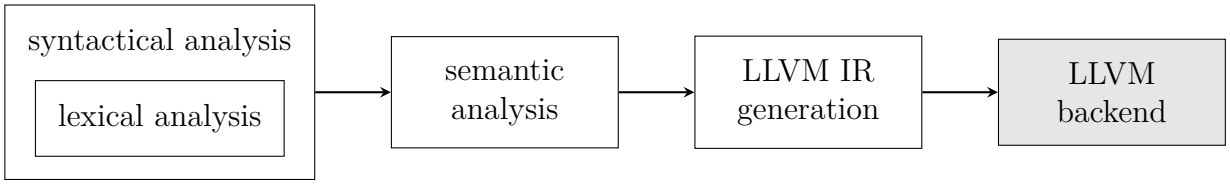


Figure 4.3 – Steps of compilation when using LLVM.

The updated Figure 4.3 now includes two new steps: "LLVM IR generation" and "LLVM backend". The first step traverses the AST in order to generate a semantically equivalent program formulated using LLVM IR. The second step 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.

4.3.2. The LLVM Intermediate Representation

The LLVM *IR* represents the source program in a low-level manner. 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 significantly 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 a separate file. If a file containing the IR is written and read by the individual parts of the compiler, the same performance issues introduced by multipass 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. Furthermore, due to the high-level type information contained in the IR, LLVM is able to perform more optimizations on the IR during later stages of compilation. Therefore, the virtual instruction set of LLVM is designed as a low-level representation with high-level type information. This 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 virtual architecture provides an infinite set of virtual registers which can hold the values of primitives like integers, booleans, floating-point numbers, and pointers. All registers in the IR use the *SSA*[8] form in order to allow more optimizations. In order to enforce the correctness of the type information included in the IR, each program has to obey these rules [Lat02, p. 14-17].

Structure of a Compiled rush Program

In order to understand how a program can be formulated using the LLVM IR, we consider the rush program for calculating Fibonacci numbers again. For reference, the rush program used in this example can be found in Listing 1.1 on page 3. The code in Listing 4.14 displays the identical rush program formulated in LLVM IR, it was generated by the rush compiler targeting LLVM[9]. The code displayed in this listing shows a module named '`main`'.

[8] Short for "static single assignment", widely used in optimizing compilers.
[9] Generated using Git commit '`e0fd6f7 `'.

```
1  ; ModuleID = 'main'
2  source_filename = "main"
3  target triple = "x86_64-alpine-linux-musl"
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 4.14 – LLVM IR representation of the program in Listing 1.1.

In the 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 source 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`', meaning a signed 64-bit integer. In rush, the '`int`' type represents a 64-bit signed number. Furthermore, we can observe that the function takes a parameter named '`%0`' of the type '`i64`'. It represents the '`n`' parameter in the rush source program.

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 variables 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`'[10] instruction is used in order to compare the runtime value of the parameter '`%0`' to a constant '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`'[11] instruction. 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 branch-instruction. Conditional jumps in LLVM always require two jump targets, 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 branch-instruction presents the essential part of the if-expression in the source program. If the condition was true at

[10] Short for "integer compare (signed less than)".
[11] Short for "branch", used to jump to the beginning of a basic block.

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 of the '`merge`' block, the '`phi`' instruction is used. These so called ϕ-nodes are necessary due to the SSA form used in 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 know 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 phi-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, we have not covered where this virtual register is declared. For this, we consider the instructions in the '`else`' block, starting in line 15 with the '`add`' instruction. 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 can be 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 this instruction is never reached. This is necessary because LLVM requires that every basic block is terminated at its end. The '`exit`' function terminates the program using a system call 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 all of its aggressive optimizations on this code. By considering the example from above, it became apparent that the IR represents many source language constructs in a high-level way. For instance, function calls can be used without considering the complex rules introduced by low-level calling conventions. 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 allocation entirely. Lastly, the LLVM IR can subjectively be seen as very readable since registers, basic blocks, and functions may contain custom, human-readable labels. Moreover, most instructions have a relatively reasonable name which allows readers to guess what the instruction is doing without them reading any LLVM documentation.

4.3.3. The rush Compiler Using LLVM

In order to get acquainted to the LLVM framework practically, we have implemented a rush compiler which uses the framework as its backend. However, the first problem emerged soon since the LLVM project only provides official C / C++ bindings to be used by other programs. Nonetheless, the entire rush project is written in the Rust programming language. 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 using LLVM for code generation [Kol17].

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>,
       // ...
47 }
```

Listing 4.15 – Parts of the struct definition of the rush LLVM '`Compiler`'.

This compiler uses the annotated AST generated by the semantic analyzer in order to translate it into LLVM IR. Here, each type of AST node is translated using its own individual function. For instance, an expression AST node is translated into IR by the '`expression`' method of the compiler. This way, translation of individual AST nodes can be organized in order to increase maintainability. To understand how this rush compiler leverages LLVM in order to translate programs, we should first consider some implementation details. The code in Listing 4.15 displays the top part of the '`Compiler`' struct definition.

The '`context`' field in line 27 represents a container for all LLVM entities including modules. Next, the '`module`' field contains the underlying LLVM module. In line 29, the '`builder`' field contains a helper struct provided by Inkwell which allows generation of IR solely in memory. All the types of the above fields are provided by the Inkwell crate and are therefore used to interact with the framework. In order to get a deeper understanding of how this compiler works exactly, we will now consider how the program in Figure 4.4 is translated into IR.

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

```
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 }
```

Figure 4.4 – Translation of a simple rush program to LLVM IR.

The source program on the left side contains the '`foo`' and the '`main`' functions. These functions are declared in the lines 5 and 14 of the output IR. The '`foo`' function takes two parameters ('`n`' and '`m`'). It uses the two parameters and 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 strikes the eye is that the declaration of '`m`' cannot be seen in the IR. Instead, the constant value 3 of the variable is used in the addition instruction. Therefore, the program uses less memory since a redundant mutable variable is not saved in memory. This again shows how advanced LLVM optimization is and how it benefits the program. The result of this addition is then used in order to call the '`exit`' function. This

function call takes place in line 8 of the IR. Therefore, the exit code of the program will be 5. During translation, the compiler first iterates over all declared functions in order to add them to the LLVM module. Listing 4.16 displays parts of the method responsible for translating the 'main' function.

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 4.16 – Compilation of the 'main' function using LLVM.

In the lines 334–336, the 'main' function is added to the current LLVM module. The definitions of the variables 'fn_name' and 'fn_type' are not visible. The first variable specifies the name of the function to be inserted, while the latter describes the function's signature. The return type of the function is specified by the 'fn_type' variable. In most cases, the return-type of the function is an integer since C libraries can then use the function as its 'main' function. 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*. In the lines 339 and 340, this method adds the 'entry' and 'body' basic block. Next, the 'entry' and 'body' block are appended to the newly created function. Therefore, the main-function now contains these two basic blocks. In the 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' field is of particular importance since all later additions of basic blocks, e.g., during loop compilation require an Inkwell 'FunctionValue'. Therefore, this field of the current function can later be accessed if a basic block should be appended. Furthermore, the 'entry_block' field in line 346 is used every time a pointer is declared. However, the reason for this behavior is explained later.

Using the Inkwell crate, most instructions generated will be automatically appended to the end of the current basic block. Therefore, the position of the instruction builder is changed to the end of the newly created 'body' block. Since this block contains the beginning of the main-function's body, the 'block' method of the compiler is called in line 351. In this case, this method first creates a new scope, then compiles all the statements which the block contains. Lastly, the method attempts to compile the block's optional expression. If the content of the body of the main-function does not lead to the insertion of more basic blocks, the 'body' block will contain the entire contents of the function after the method call.

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, we will now consider Listing 4.17.

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).as_str())
954         .try_as_basic_value();
            // ...
957 }
```

Listing 4.17 – Compilation of call-expressions using LLVM.

The code in Listing 4.17 displays a small part of the '`call_expr`' method of the rush LLVM compiler. This snipped shows the statement inserting the 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. Since the variable '`func`' represents the called function, it was previously declared by looking up the function name in the module. The '`args`' variable is of type '`Vec<BasicMetadataValueEnum>`' and therefore represents a list of Inkwell values representing the arguments used for the call. This variable was also defined previously by iterating over the '`node.args`' vector containing expressions. This vector is contained in the provided AST node representing the call-expression. Each argument expression is then compiled, and its result is placed into the '`args`' output vector. However, we cannot understand how results of expressions are handled in this compiler without considering Listing 4.18.

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 4.18 – Compilation of expressions Using LLVM.

The code in Listing 4.18 shows parts of the `expression` method of this compiler. When consider the method's signature, it becomes apparent that it uses an '`AnalyzedExpression`' in order to generate a '`BasicValueEnum`'. The return type of the function is of particular importance. Using Inkwell, most inserted instructions yield a symbolical value at compile time. This value represents a virtual register which will contain a real *value* at runtime of the program. Therefore, the '`BasicValueEnum`' returned by the function represents the virtual register which will hold the result of the expression at runtime. This way, symbolical values can be used at compile time, thus presenting a high-level abstraction for generating the IR. The lines 875–879, show how a constant integer expression is compiled. Here, a constant int value of the '`i64`' type is created and transformed into a '`BasicValueEnum`' which is then used as the method's return value. For more complex expressions, the '`expression`' method invokes other methods which are specialized on this type of expression. For instance, if an

infix-expression like '`3 * n`' is compiled, this method calls the '`infix_expr`' method in line 910. Here, the current AST node is passed to the specialized function as a call argument.

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

```
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),
```

Listing 4.19 – Compilation of integer infix-expressions using LLVM.

The code in Listing 4.19 shows a part of the '`infix_helper`' method which is responsible for compiling parts of infix-expression. Line 1021 contains the code for inserting the '`mul`' multiplication instruction. Here, the variables '`lhs`' and '`rhs`' are used as arguments for the '`build_int_sub`' method call. They, too, represent virtual registers which will contain the value of the left- and right-hand side at runtime. Furthermore, the string containing '`i_prod`' specifies the name of the virtual register containing the product of the multiplication performed by the instruction. In this example, 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 power operations 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 in order to call it directly after. This function implements an algorithm for power operations given an integer base and exponent. However, this function is implemented in IR directly by hardcoding the required calls to Inkwell into this function. Therefore, even complex calculations like this one can be implemented even though LLVM does not provide a straight-forward way to accomplish them directly.

In line 6 of the source program, a let-statement is used to declare the mutable variable '`m`' with the initial value 3. However, there is never a value assigned to this variable. This variable is only mutable so that the compiler has to use stack memory for it. Non-mutable variables are inlined by the compiler in order to save resources during runtime. In order to understand how the compiler translates let-statements, we will now consider Listing 4.20.

The code in Listing 4.20 displays parts of the '`let_stmt`' method of this compiler. This method is responsible for compiling let-statements. In line 610, the initializer expression of the statement is compiled. The '`rhs`' variable then specifies the virtual register which contains the result of the expression at runtime.

The code in the block after line 614 is only executed if the variable was declared as mutable. Otherwise, the variable would be constant and therefore require no space in memory. 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. The first argument of the call specifies that the name of the pointer should be identical to the name of the variable. The second argument passes the type of the initializer expression to the method. The statement in line 619 is used in order to insert a store instruction. Here, the instruction should store the value of the initializer expression 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 an IR program using pointers. Finally, in line 622, 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 fields are required when the variable is used later. The code in Listing 4.21 shows the '`alloc_ptr`' method of the compiler.

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

```rust
fn let_stmt(&mut self, node: &'src AnalyzedLetStmt) {
    let rhs = self.expression(&node.expr);

    // if the variable is mutable, a pointer allocation is required
    match node.mutable {
        true => {
            // allocate a pointer for the value
            let ptr = self.alloc_ptr(node.name, rhs.get_type());

            // store the rhs value in the pointer
            self.builder.build_store(ptr, rhs);

            // insert the pointer into the current scope (for later reference)
            let var = Variable::new_mut(ptr, node.expr.result_type());
            self.scope_mut().insert(node.name, var);
        }
        // ...
    };
}
```

Listing 4.20 – Compilation of let-statements using LLVM.

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

```rust
fn alloc_ptr(&mut self, name: &str, llvm_type: BasicTypeEnum<'ctx>) ->
↪  PointerValue<'ctx> {
    // save current insertion point
    let curr_block = self.curr_block();

    // insert at `entry` block
    self.builder.position_at_end(self.curr_fn().entry_block);

    // allocate the pointer
    let res = self.builder.build_alloca(llvm_type, name);

    // jump back to previous insert position
    self.builder.position_at_end(curr_block);

    res
}
```

Listing 4.21 – Pointer allocation in the LLVM compiler.

This method exists in order to create a new Inkwell pointer value. Like hinted previously, pointers are declared in the '`entry`' block of each function in order to allow for more aggressive optimizations. In line 640, this method places the builder cursor at the end of the entry-block of the current function. Next, in line 643, an '`alloca`' LLVM instruction is inserted. This instruction is responsible for allocating a new pointer which points to stack memory. After the instruction has been inserted, the builder position is reset to where it was before the method was called. Finally, the pointer is returned so that it is usable for other parts of the compiler.

4.3.4. Final Code Generation: The Linker

After LLVM has compiled a program, it outputs 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*[12] and not directly executable. In order to create an executable program from object files, a *linker* is used.

A linker or *link editor* is a program which takes one or more object files in order to combine them into a single file. Often, the output of the linker is a file which can be executed by the operating system. For instance, a linker might take an object file generated by a compiler in order to create the final executable program. During *linking*, a linker often perform numerous tasks, such as *relocation* or *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 the functionality provided by 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, we will not explain these concepts further since they are not of particular relevance for understanding the purpose of a linker.

The shell command in Listing 4.22 presents an example liker invocation. In this example, the LLVM compiler has generated an object file named `input.o`. The flag `-dynamic-linker` is used in order to tell the linker which dynamic linker should be used. Next, some library files in the directory '`/usr/lib/`' are included. These files belong to an implementation of the C standard library and are required so that the '`exit`' function works properly. Furthermore, the '`input.o`' file is specified so that the linker includes it.

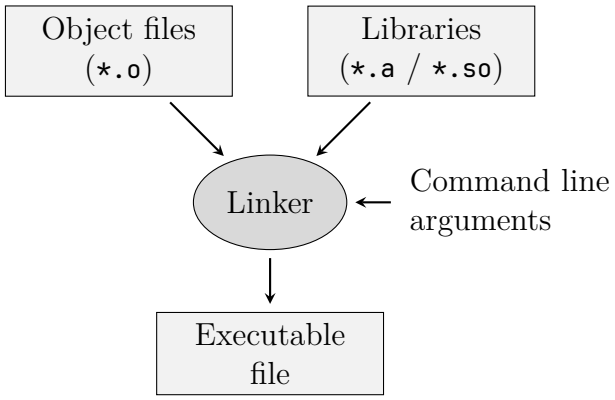


Figure 4.5 – How a linker works.
[Lev00, p. 7]

```
1 ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
2   /usr/lib/crt1.o \
3   /usr/lib/crti.o \
4   -lc input.o \
5   /usr/lib/crtn.o -o output
```

Listing 4.22 – Using LD to link the LLVM output.

This way, the shell command would generate an executable program named '`output`' from an object file named '`input.o`'. Therefore, a linker often presents the final step of translating a source program into an executable which the computer can understand. 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 program generated by the compiler.

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

4.3.5. Conclusions

As a conclusion, implementing a compiler which leverages LLVM presents a lot of advantages. For instance, the language will be able to support many backend architectures. Most of the demanding work is being done by LLVM, therefore implementing the compiler will proof to be less difficult and error-prone. Moreover, LLVM performs a lot of very effective optimizations which would otherwise have to be implemented by the compiler designer. However, these optimizations often involve a lot of work and are therefore unpractical to implement for simpler languages. Therefore, LLVM presents a robust, production-ready, and scalable backend which is even used in real-world compilers. However, by depending on LLVM, the resulting compiler will often be less portable since cross-compilation still presents an issue if used across programming language boundaries.

Finally, in order to understand how LLVM's optimizations can positively impact application performance at runtime, we will consider the Fibonacci benchmark again. In this benchmark, the 42nd Fibonacci number is calculated using the program displayed in Listing 1.1 on page 3. However, the 10 in line 2 was replaced by a 42. Running a binary 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[13]. Therefore, the program compiled using LLVM ran roughly 1.66 times faster.

4.4. Transpilers

Another, more esoteric target of a compiler is presented by another high-level programming language. A compiler translating one language to another 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 often that it generates very inefficient code. Furthermore, the entire project would also dependent on the target programming language [Jef21, p. 5].

For this paper, we have implemented a transpiler which generates *ANSI C* from rush source programs. Listing 4.23 shows a rush program which terminates using 42 as its exit code. For this, the value of the block-expression in the 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 the value of the temporary variable. In line 6, the '`exit`' function is called, using '`num`' as its argument. On the right side, Listing 4.24 shows the identical program represented using ANSI C. The code was generated from the rush program in Listing 4.23 using the rush C compiler[14].

[13] Average from 100 iterations. OS: Arch Linux, CPU: Ryzen 5 1500, RAM: 16 GB.
[14] Generated using Git commit '`e0fd6f7` '.

```
fn main() {
    let num = {
        let b = 40;
        b + 2
    };
    exit(num)
}
```

Listing 4.23 – A rush program containing a block-expression.

```
#include <stdbool.h>
#include <stdlib.h>

int main();

int main() {
    long long int b0 = 40;
    long long int num1 = b0 + 2;
    exit(num1);
    return 0;
}
```

Listing 4.24 – C output of rush program in Listing 4.23.

When examining the C output, it becomes apparent that the structure of the code has changed slightly. In line 1 and 2, the file contains '`#include`' directives in order to import files from the C standard library. In line 4, the '`main`' function is declared, but not defined. Although this might not seem very rational in this example, declaring functions before their declaration allows rush functions to be called before their definition. Furthermore, the C program also does not include the previously mentioned block. The lines 2–5 in the rush code are represented by the lines 7 and 8 in the C code because C does not have a structure comparable to rush's block-expressions. This examples demonstrates how a transpiler often alters the program's structure in order to represent the source program in another language. Due to the practical irrelevance of transpilers, the internals of this transpiler will not be discussed further. As a conclusion, transpilers can present an easy but limited way of implementing a compiler. Due to its disadvantages, a transpiler is often only considered during the prototype phase of compiler implementation.

5. Compiling to Low-Level Targets

In the previous chapter, we have learned how compilation to high-level targets can present an easy way of implementing a compiler. These compilers generated outputs which were still platform independent and portable. However, compilers can also target a specific computer architecture directly, thus removing another layer of abstraction. The concept of compiling to a specific architecture directly is similar to the VM since its compiler also targets its architecture directly. However, implementing a compiler targeting the architectures presented in this chapter has proven to be a lot more demanding since the VM uses a fictional architecture purposefully developed for this paper. Reasons for this chapter's difficulty mostly include target-specific constraints which were still irrelevant in the previous chapter. This chapter contains the term "low-level" in its name since the presented compilers generate code for specific target-architectures directly.

5.1. Low-Level Programming Concepts

Programming using high-level languages does not require knowledge about the target architecture of the program. However, in this chapter, two compilers targeting the low-level assembly language are presented. In order to make sections in which these compilers are explained more approachable, some of the most important low-level programming concepts are explained in this section. We will only explain concepts which play a significant role later on.

5.1.1. Sections of an ELF File

Since a program needs to be representable in a low-level manner, special formats are often required. ELF stands for "executable and linkable format" and is often found on unix-like systems like Linux. Programs using the ELF format can be represented in three different types of files. For instance, object files generated by a compiler, like in the previous LLVM section, might use the ELF format. Furthermore, libraries using *shared object files* might also leverage the ELF format. Most executable program 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 like to separate these types of memory into their separate components. Therefore, an object file and assembly program is divided into so-called *sections* [Zhi17, p. 19]. Some of the important ELF sections are displayed below [Zhi17, p. 76]:

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

Even though a typical ELF file also contains other sections, this list only includes entries which are of importance later in later sections of the paper.

5.1.2. Assemblers and Assembly Language

Assembly language describes a type of low-level programming languages which are directly influenced by the target architecture. Since the assembly code provides a slight abstraction over the computer's hardware, the assembly code must be translated to machine code before it can be executed. This process is performed by a program named the *assembler* and is often called *assembly*. A typical characteristic of assembly code is that it seems relatively cryptic to a human reading it. Furthermore, compared to high-level languages like C, assembly code is relatively low-level since it can be used to interact with the hardware directly.

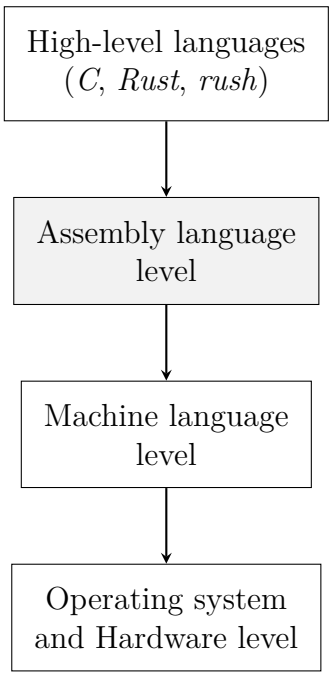


Figure 5.1 – Level of abstraction provided by assembly.

For instance, the RISC-V instruction '`add a0, a0, 2`' would be used in integer addition. This example contains most characteristics of an assembly language program. Like hinted previously, the name of the instruction is a mnemonic. In this case, '`addi`' stands for "add immediate". Furthermore, the exact semantic meaning of the instruction is not immediately apparent. At last, the instruction for adding integers differs for most CPU architectures. For instance, an equivalent instruction for the x_86 architecture could be '`add %rdi, 2`'. Therefore, the fact that instructions differ on each target architecture is clearly apparent.

As seen above, 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 of the assembly should contain all the instructions. Just like parts of the LLVM IR, the '`.text`' section of an assembly program obeys a hierarchy. This section contains many labels which mark the beginning of a new basic block. However, these basic 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 mean jump- and return-instructions. Therefore, the constraints in assembly are much weaker than the ones introduced by LLVM. Because an assembly usually omits complex optimizations, struct rules constraining the assembly code can be omitted too.

Since assembly provides significantly less abstraction than high-level languages, the question of how much abstraction is lost emerges. In order to understand how much abstraction is provided by assembly, we should consider Figure 5.1. Here, the highest level of abstraction is provided by high-level programming languages like C or Rust. In the context of this paper, rush presents roughly the same level of abstraction like these languages. The next lower level of abstraction is provided by assembly. Now, the program is no longer independent of its target architecture and is much more demanding to furmulate. However, the next lower level of abstraction below assembly is represented by code in machine language. As of today, one only rarely encounters a programmer writing programs using this level of abstraction. Since the machine language program is represented in binary, it is nearly impossible for a human to write or understand. However, the machine language is also just an abstraction of the computer's hardware and operating system. Therefore, assembly provides enough abstraction to be comprehensible for a human while being a low-level representation of the program. Although assembly provides more abstraction than the two levels at the bottom of the figure, programmers rarely program in assembly directly. Some benefits of using assembly language to formulate a program are increased runtime efficiency and decreased code size while having fine-grained control over the hardware and operating system. Therefore, it

might be reasonable for a compiler to generate assembly code from the source program. This way, the program is translated into a low-level, target-specific representation which allows the program to be executed on the target machine directly. However, an assembler is still required in order to translate the assembly output of the compiler. Since most assemblers output object files, a linker is required to create the final executable program. Therefore, a compiler targeting the assembly of a specific architecture depends on these two additional steps before the program can be executed [Dan05b, p. 5-6].

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 often significantly increases the complexity of the compiler since it now has to perform the role of the assembler as well. However, the compiler could also emit binary data directly, thus making its implementation significantly more demanding.

5.1.3. Registers

Most processors contain numerous registers in order to hold data, instructions, and state information. In a computer, registers are sometimes used to hold data stored in variables. However, most of the time, registers are used as temporary storage in large computations. Registers are used in the latter case because they are way faster than memory. Therefore, an algorithm using registers instead of memory will usually run faster. However, even though registers are faster than memory, they are not used all the time. This is because CPU registers are a limited resource, meaning every register-based CPU only has a finite amount of registers available. Registers should therefore be used carefully and only when they are really needed [Wat17, pp. 212-214].

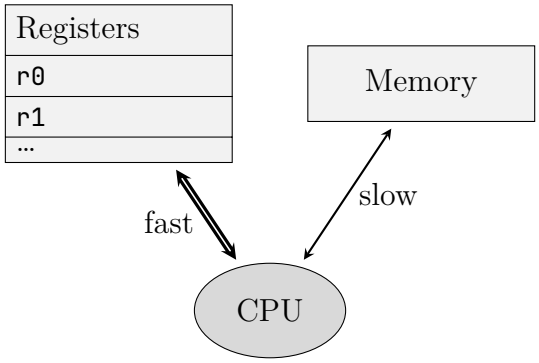


Figure 5.2 – Relationship between registers, memory, and the CPU.

Figure 5.2 shows how the CPU interacts with its registers and the computer's memory. The connection between the set of registers and the CPU is marked as a short and thick arrow because the connection between the CPU and its registers is very short and fast. Although the registers are often physically parts of the CPU, they are displayed as separate entities in this example. For the memory, the arrow is long and thin. This represents the long and therefore slow connection between the CPU and the memory modules. In most modern computers, this connection often spans across several centimeters on the motherboard. Therefore, higher latency during memory access is inevitable [Dan05b, pp. 20-21].

Nearly every CPU architecture will include an individual register layout, differing in size, count, and type. Therefore, the exact information about registers always depends on which CPU is used. As a rule of thumb, having more registers in an architecture will almost always improve performance of that architecture. Furthermore, not every register is identical. There might be platforms with some general-purpose, some floating-point, and some special-purpose registers. For instance, there may be registers which contain information about the CPU's current instruction. Furthermore, common architectures also include special-purpose registers for modifying the machine's memory. Therefore, there is nearly always an appropriate register matching the requirement [Dan05b, Chapter 2].

For a compiler, the limited amount of registers presents a big challenge. 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 registers management can present a challenge, Listing 5.1 should be considered. This snippet 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`'. In line 5, another integer, this time 2 is placed in '`a1`'.

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

Listing 5.1 – Example assembly program for explaining register allocation.

Next, 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]. Through this example, it becomes apparent that other instructions might use registers unexpectedly. Therefore, subtle bugs can be created if an instruction overwrites registers.

In compilers, the process of *register allocation* is responsible for managing how registers are used. This process attempts to use registers in a way leading to maximized efficiency of the output program. Since accessing registers is often faster, a register allocator will try to use registers as often as possible. Production-ready compilers often try to keep as much of the frequently accessed data in registers. For instance, this frequently accessed data may also include variables which are normally saved in memory. 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 of saving excess data in memory instead of registers is called *register spilling*. Since register spilling introduces a performance penalty, register allocation algorithms often attempt to prevent it as much as possible. Therefore, sophisticated algorithms for register allocation are often mandatory as long as the factor of output code performance is non-trivial.

Apart from just managing the use of registers, most allocation algorithms are responsible for many other register-related tasks. For instance, register allocation should detect when a variable is no longer needed so that its register can be freed. 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 basic block. It is apparent that an algorithm performing all these tasks can not be implemented in an ad-hoc manner. Instead, this process often requires complex graph algorithms for determining which registers can be used and freed. Therefore, implementing register allocation in a compiler is often a very demanding task [Wat17, pp.212-214]. Since register allocation represents a complex topic, it will not be explained any further.

5.1.4. Using Memory: The Stack and the Heap

As registers are limited in both size and count, additional storage for long-term memory is required. This is what the *stack* is for. A stack in general 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 separate from the CPU and therefore much slower to access. It is usually implemented as a linear memory that can be freely accessed and modified, where a *stack pointer*, which is usually stored in a special register, saves the address of the 'top' of the stack. Thus, the principal of this memory being a stack data structure is merely a convention. A stack typically grows towards lower addresses, so that to push values the stack pointer must be decreased [PH17, pp. 68, 99, 100]. **TODO: ugly line wrap**

It is typical for compiled procedures to *allocate* and 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 a stack frame. This pointer is usually called *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`', taking up one, two, and eight bytes respectively. **TODO: do we need to explain these Rust types?** The first layout has two of the integers not correctly aligned, marked in dark gray, thus being invalid. The second layout preserves the same order but inserts empty white areas in two places, called *padding*, to correct the alignment. First one byte just before the '`u16`' to align it to an integral multiple of two bytes, and secondly 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. **TODO: cite?**

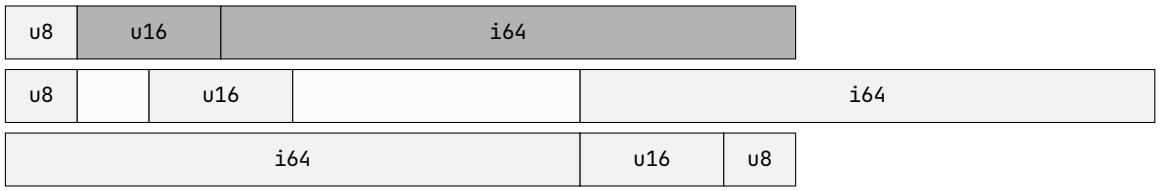


Figure 5.3 – Examples of memory alignment.

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 integral multiple of the wanted 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', 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 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.

TODO: some figure of a stack?
TODO: example for array?
TODO: The section title contains 'Heap'. Can we just leave that out?

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. Therefore, procedures allow the programmer to focus on one portion of the tasks at the time. Often, parameters and returned values act as the interface between the procedure and the remaining code. Therefore, procedures present one way of how a high-level programming language might provide abstraction. 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 might take place like the following steps:

- The caller places the call arguments in a place where the procedure can access them.
- Control is transferred to the procedure, often using a jump or specialized call instruction. This specialized instruction often saves the *return address*[1] in a special register so that function returns are easier.
- The first task the called function performs is acquiring local storage resources needed by the procedure. Often, registers are spilled if required. Furthermore, stack memory for the procedure's local variables is often allocated in this step.
- Internal code of the procedure is run by executing the function body's instructions.
- 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.
- Control is transferred back to the caller using a specialized return-instruction. A specialized instruction is often required since procedures can be called from multiple places in the program. Therefore, the return-instruction jumps back to the instruction which had initiated the function call.

In order to leverage optimal performance during function calls, passed arguments are usually kept in registers. However, in the previous subsection about registers, we have learned that most architectures state that only subset of their registers are to be used as function arguments. Therefore, if a function is called using more than eight arguments, every remaining argument has to be spilled to memory. In this case, each additional parameter of the function is placed inside the stack frame of the called function so that it can access it. [PH17, p. 98].

For instance, a target architecture might provide *four* registers which can hold function arguments. Now, a function is called using *six* arguments. Here, register spilling is mandatory since there are two more arguments than registers. In that case, the registers '`r0`' — '`r3`' would contain the first four argument values while the fifth and sixth argument is spilled on the stack.

For managing the calling convention in assembly, most functions contain a *prologue* and an *epilogue*. These blocks of the function are responsible for allocating and deallocating resources which the function might use. For instance, the stack- and frame-pointer is often adjusted in the prologue. This way, space on the stack is allocated for variable declarations found in the called function. However, the modification offset always depends on how much memory the called function will require during runtime. Therefore, a compiler would have to keep track of the count of variable declarations made by a function. Apart from allocating stack space, the prologue is also often used for storing special registers on the stack.

[1] A link to the caller site, allows the called procedure to jump back to the caller [PH17, p. 99].

The epilogue however is required for deallocating all resources which were previously allocated by the function's prologue. For instance, the stack-related pointers are modified to reflect their state before the function call. Furthermore, any special registers previously saved on the stack are now restored. Obviously, a '`return`' statement should always jump to the function's epilogue instead of terminating the function directly. At the end of most epilogues, a return-instruction jumping back to the caller location is often found. Therefore, the prologue is executed as the first code when calling the function and the epilogue is called as the last code before this function terminates.

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*[2] compatibility of the compiled program.

5.1.6. Referencing Variables Using Pointers

A *pointer* is a variable which contains the memory address of another variable. Sometimes, pointers are mandatory for solving a specific computational problem. Furthermore, leveraging pointers often leads to more efficient and compact code [KR88, p. 93]. A problem which can only be solved by using pointers is displayed in the rush program in Listing 5.3.

```
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.

This rush program contains the '`main`' and '`modify`' functions. In line 2 of 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, using the value of '`answer`' as its exit code. In the signature of the '`modify`' function, the '`n`' parameter is declared as a mutable integer. Next, in line 8 of this function, 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 and not by reference, the called function has no direct way of altering the variable behind the passed parameter [3]. In order to solve this problem, pointers are required. The code in Listing 5.4 displays a rush program which is able to solve this problem.

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

```
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 pointers. 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 of the variable. The parameter 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 stored in the pointer is incremented. When assigning to the target variable behind a pointer, the pointer first needs to be dereferenced using the '`*`' prefix operator.

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]. For instance, a pointer variable named '`p`' would be dereferenced by using the following rush expression: '`*p`'.

In rush, only pointers targeting an existing variable can be created. In order to create a pointer to an existing variable, rush's '`&`' prefix operator is used. This operator *references* the target variable in order to return a pointer value. *Referencing* a variable produces the memory address of that variable, often in form of an integer value [KR88, p. 95]. Theoretically, since the resulting value is only a normal integer, it can also be treated similarly. Therefore, creating a new pointer variable involves following steps:

1. Producing the absolute memory address of a variable
2. Saving the resulting integer as a variable on the stack as if it was a conventional integer

It is apparent that implementing pointers should often be relatively straight-forward since the underlying principle consists of only these two simple steps. However, some target architectures of a compiler might make the implementation of pointers more difficult. Therefore, the statement in line 8 increments the value of the variable stored in '`n`' by 1. Here, only the value of the pointer is copied when being used as an argument. Since the pointer only saves the address of its target, the copy does not affect the target variable by any means. Because rush pointers allow write-access, '`answer`' can be incremented by using a pointer.

Due to this write-access, the analyzer only allows the referencing ('`&`') 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*[4] into a program. This however 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 often freed after the function has executed. Therefore, pointers should be used with caution since the semantic analyzer cannot guarantee that they are used correctly.

[4]The runtime behavior of such scenarios is undefined and might vary on every architecture

5.2. RISC-V: Compiling to a Modern RISC Architecture

The *RISC-V ISA*[5] is a new and 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. Since the beginning, the project has been managed and led by the *RISC-V foundation*, consisting of many individuals contributing to the project. 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. This can be regarded as a large competitive advantage over other popular RISC architectures like *ARM*. 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 like *Google* can leverage the technology commercially while contributing to the project. Unlike most of the previous ISAs, which were developed during the 1970s or 80s, RISC-V is one of the few which were developed this decade. Therefore, it seems like RISC-V could be a significant architecture to be used in all sorts of devices in the near future [PW17, preface].

5.2.1. Register Layout

Most RISC architectures typically have a large count of registers [Dan05b, Chapter 2]. When compared to other popular architectures, the truth of this statement becomes clear. For instance, the *x86_32* architecture has 8 registers. The popular RISC architecture *ARM-32* provides twice that amount, meaning 16 registers. However, a RISC-V CPU includes 32 registers, which is drastically more than the previously mentioned architectures. Moreover, these 32 registers only include registers holding integer values. Just for floating-point numbers, the ISA even provides another 32 registers. Like previously explained, using more registers usually leads to increased efficiency of the output program. Therefore, a register allocation algorithm targeting the RISC-V architecture could be more aggressive compared to one targeting x_86 for instance [PW17, p. 10].

Table 5.1 – Common registers of the RISC-V architecture.

[WA19, p. 155]

Register	Purpose
`zero`	Hardwired zero
`ra`	Return address
`sp`	Stack pointer
`t0` — `t6`	Temporary
`fp`	Frame Pointer
`a0`, `a1`	Function argument, return value
`a2` — `a7`	Function argument
`s1` — `s11`	Saved register
`fa0`, `fa1`	FP args, return value
`fa2` — `fa7`	FP args
`fs0` — `fs11`	FP saved registers
`ft0` — `ft11`	FP temporaries

The Table 5.1 shows most of the registers which the RISC-V architecture provides. For this table, the official ABI names of the registers have been used in order to make this section easier to read. The first column of the table contains a register's name while the second column describes its purpose.

The first row of the table contains the '`zero`' register. On RISC-V, this register is special. Like its name suggests, it holds the value of a constant 0. Unlike other registers, it is read-only, meaning that it can never be overwritten, therefore preventing accidental writes. In the next row, the '`ra`' register is shown. It saves the *return address* of a function or subroutine. If a return-instructions is used, the value in

[5] Short for: "instruction set architecture"

'ra' is read as it is used to jump to a specific instruction. The purpose of this register is elaborated further in Subsection 5.2.3 about RISC-V's calling convention. The 'sp' and 'fp' registers are used for managing stack memory. Their purpose is explained in Subsection 5.2.2 about stack memory. In the fourth row, the 't0' — 't6' are displayed. These registers are often used to store temporary values used in larger computations. In row six, the registers 'a0' and 'a1' can be seen. These both serve as call arguments and return values of functions. The remaining a-registers 'a2' — 'a7' can only be used as function call arguments. How functions are called using registers will be explained in Subsection 5.2.3. The next row contains the *saved* registers 's1' — 's11'. These registers are typically preserved across function calls, meaning a called function must not overwrite them. What the previously explained registers have in common is that they all hold integer values. Depending on the exact RISC-V architecture, all registers, including floating-point registers, either hold 32 or 64 bits of information. For floating-point number values, RISC-V provides other registers. These registers are able to hold floating-point numbers according to the *IEEE 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. However, the other fa-registers 'fa2' — 'fa7' can only be used as function arguments holding floating-point numbers. Just like the 'sx' registers, the 'fs0' — 'fs11' registers are usually preserved across function calls. Last, the 'ft0' — 'ft11' registers can be used as temporary registers for floating-point numbers. It is apparent that the floating-point registers are provisioned very similarly to the integer registers. Therefore, a programmer or compiler targeting the architecture can utilize roughly the same principles, regardless of the data-type stored in each register [PW17, pp. 18f, p. 34], [WA19, p. 155].

Now, it has become apparent that RISC-V includes many registers which are 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. Although this subsection provides a good overview over the registers of the architecture, the purpose of some special registers is still not known. These special registers, like 'sp', 'fp', and 'ra', are thoroughly explained in the next sections.

5.2.2. Memory Access Through the Stack

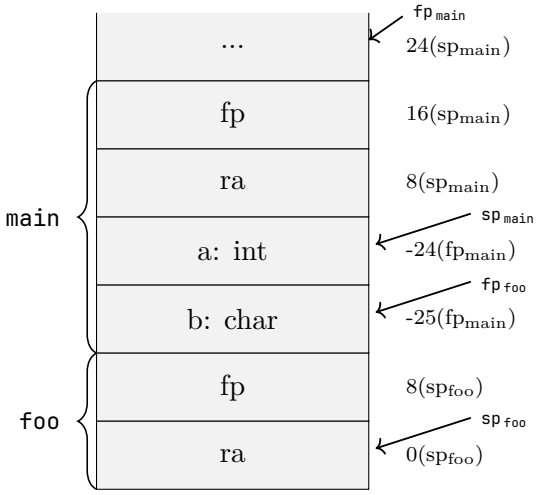


Figure 5.4 – Example stack layout in RISC-V.

As mentioned in the dedicated subsection about the stack, it presents a way to save data outside of registers. This subsection explains special conventions followed when using stack memory on RISC-V. Just like previously explained, most stacks are accessed through the specialized pointers. On RISC-V, there is a *stack pointer* which is saved in the 'sp' register. Furthermore, the 'fp' register contains the *frame pointer*.

Like most memory stacks, the RISC-V stack grows downwards, meaning it progresses into lower memory regions. In the current implementation of the rush RISC-V compiler, the stack pointer points to the last legal memory cell of the current stack frame. Therefore, 'sp' points on the cell with the lowest address of the current stack frame. On the other hand, the frame pointer 'fp' points to the cell above the end of the current stack frame. Therefore, the frame pointer always points to a memory cell which is illegal to use by the current function. Thus, a stack frame is defined by its upper and lower bounds, represented

by ‘fp’ and ‘sp’, respectively.

In order to understand how the above behavior is used in practice, Figure 5.4 is to be considered. Furthermore, the rush program in Listing 5.5 should be considered.

```
1 fn main() {
2     let a = 42;
3     let b = 'z';
4     foo();
5 }
6
7 fn foo() {}
```

Listing 5.5 – A rush program containing two variables.

This snipped shows a rush program which contains two functions. In the ‘main’ function of this program, two variables are defined. The ‘a’ variable holds the integer value 42 while the ‘b’ variable contains the char value ‘z’. In line 4, the ‘foo’ function is called. Figure 5.4 shows the state of the stack at the point when this function was called. The braces on the left side of the stack group the stack into frames. On the right side of each stack cell, its relative address can be observed. As explained previously, each stack cell is accessible either by offsetting ‘sp’ or ‘fp’. The stack pointers of each stack frame can are displayed by the arrows on the right side of the stack.

The stack shown in this figure contains two frames, one for each function involved in the call. Since the ‘main’ function is called first, it is displayed at the top of the figure. Normally, the last pushed element of a stack is located at the top, however, just as described, this stack progresses towards lower memory, meaning that it grows downwards. Since the ‘main’ function calls the ‘foo’ function, the stack frame for the ‘foo’ function is located at the bottom of the stack.

It is apparent that every stack frame saves the ‘fp’ and ‘ra’ registers at its top position. These two special registers are saved at the two top positions of each call before any of the code in the function’s body is executed. Why these registers are saved on the stack is explained in Subsection 5.2.2 about the calling convention. Since every stack frame contains these two elements, the minimum size s of a RISC-V stack frame in bytes must be $s = \frac{2 \times \text{Size}_{\text{int}}}{8}$. Since s is dependent on the integer-size of the RISC-V architecture, the minimum required memory differs per RISC-V architecture. For instance, if the 64-bit version of RISC-V was used, the minimum size would be 16 bytes ($s = \frac{2 \times 64}{8} = 16$). However, if the 32-bit version of the architecture was used, s would be only 8 bytes ($s = \frac{2 \times 32}{8} = 8$).

Additionally, one can observe that the ‘main’ function’s stack frame contains cells which save the two variables which are defined in the body of the function. Since the code in the function’s body (where the variables are defined) is executed after ‘fp’ and ‘ra’ have been saved on the stack, the cells containing these variables appear lower in the stack. Another interesting observation is that the more recently declared variables are also saved in lower cells of the stack frame. Therefore, the order in which variables are saved in the stack follows the *LIFO* principle which is common in stacks.

In this example, in the stack frame of the function ‘main’, the registers ‘fp’ and ‘ra’ require 16 bytes of memory together. Therefore, the variable ‘a’ can be saved at the next 8 bytes of memory, meaning -24(fp). This way, the variable is saved at the memory region from -24(fp) until the start of -16(fp) or 16(sp) in this example. Therefore, each stack cell requires exactly as much memory as shown in the figure. Just like described earlier, different rush types require different quantities of memory. A character for instance only uses 1 byte of memory. Thus, the entire variable ‘b’ is saved at -25(fp), meaning one byte below the end of the variable ‘a’. This way, each variable only uses as much memory as it actually requires.

However, the question of how the compiler is able to keep track of saved variables remains. For this, 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. In case of the program in Listing 5.5, the variable '`a`' is associated with a '`fp`' offset of -24. If the variable is referenced 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. For most architectures, the calling convention is part of the ISA's official specification. In the case of RISC-V, the calling convention is specified in a separate document [22].

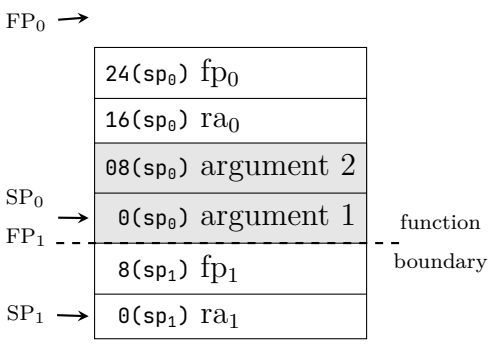


Figure 5.5 – Spilled registers during a RISC-V function call.

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 specialized registers. Like described in the Table 5.1, only special classes of registers can be used as call arguments. For integer arguments, the first 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`'. However, the third argument is a floating-point number and can therefore not be placed inside an integer register. Therefore, the first floating-point argument register '`fa0`' contains the argument '`3.14`'. In this case, all arguments can be held in regisers and spilling would not be required.

In case the function accepted nine or more integer arguments, all further integer arguments upward of the ninth position would have to be spilled on the stack. Here, the successive registers '`a0`'–'`a7`' would contain the first eight integer arguments of the called function. The argument at position 9 however is then spilled on the stack since there are no registers left which could contain the additional argument [22, p. 8].

The Figure 5.5 displays a possible state of the call stack during a function call which uses ten integer arguments. If ten integer arguments are used, two arguments would have to be spilled on the stack. In the figure, the spilled registers are placed in the stack cells "argument 1" and "argument 2". Here, the cell "argument 1" would hold the ninth argument while "argument 2" holds the tenth argument. Therefore, all spilled argument registers will be placed in the stack frame of the caller function. Normally, variables saved on the stack are aligned to reflect their sizes. In case of spilled argument registers however, every argument will occupy exactly 8 bytes on the stack, even if the data type itself requires less space.

Now that the first step of a procedure call is explained, the question of how the second step works in RISC-V remains. In the second step, the underlying procedure call is made using a specialized instruction. In RISC-V assembly, one typically uses the call *pseudoinstruction*[6]. Due to a lack of functions in assembly, the call-instruction uses the name of its target label as one operand. Therefore, labels can be called as if they were functions. This instruction will jump to the first instruction of the specified target label while saving the address of the next instruction after the '`call`' instruction in the register '`ra`' [PW17, p. 22]. As hinted

[6] A macro generating multiple instructions from one pseudoinstruction. Therefore, the actual count of ISA instructions remains low while convenience features can be used in assembly [Dan05b, p. 68].

previously, the '`ra`' register saves the *return address*. Therefore, the return address is set every time a function call is performed.

During the third step of the function call, the called function acquires local storage resources. To be precise, the function decrements the stack pointer by the amount required by the stack frame. Therefore, the function allocates as much stack space as required for storing local variables and other data. Additionally, the frame pointer and return address are saved on the stack so that nested function calls do not cause issues. For instance, if the return address was not saved on the stack, a nested function call would overwrite its stored value. In this case, the parent function could no longer return since the return address now holds an incorrect value. In order to mitigate issues like this, the return address and frame pointer are saved on the stack. Figure 5.5 shows that the two registers are saved at the two positions on the top of the new stack frame. This part of the function is often called the *prologue* as it is executed before any of the function's internal code.

After the code of the function has been executed, the so-called *epilogue* is executed. Since the frame pointer and return address have been saved on the stack during the prologue, the epilogue restores these registers by loading their values from the stack. Furthermore, the amount which was subtracted from the stack pointer in the prologue is now added to the pointer in order to restore it to its original state before the function call. Here, incrementing the stack pointer represents deallocating the previously acquired stack space. However, the memory in the stack frame is not actually deleted since only the stack pointer is modified. Even though the used memory is not explicitly deleted, it is still freed since it will probably be overwritten by the next function call. If the prologue would not save the return address on the stack, a nested function call would overwrite the return address of the parent function, therefore creating a bug [PW17, p. 33]. By saving the return address on the stack, nested function calls do not cause difficulties. It is apparent that this design contains a lot of similarities to the call-stack of the rush VM. However, in the VM, the process of saving and restoring the return address was managed automatically by the VM, whereas, here, the programmer has to manually pay attention to saving and restoring this important piece of data. Therefore, implementing function calls is definitely more demanding in RISC-V assembly than in the rush VM.

In case a function returns a value, it must be communicated to the caller so that it can access it. For integer-based 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 character and boolean values are also placed inside the '`a0`' register since these types can be represented by integers.

Lastly, the epilogue contains a return-instruction which should jump to the place where the function was called. This *ret* instruction reads the value stored in '`ra`' in order to jump to this address.

5.2.4. The Core Library

Like hinted in the section about the linker on page 58, 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 system *exit* call. Since these concepts must introduce additional logic, the compiler should not be emitted their instructions every time they are used. In that case, the repeated emission of redundant instructions would result in enlarged and unnecessary complex output code. In order to mitigate these issues, the compiler simply inserts call-instructions referencing external functions. External functions can be 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. For this compiler, we later refer to this target specific library code by the term *corelib*.

For instance, a function for mathematical power operations is implemented in the corelib. Therefore, the compiler can emit a procedure call to this method 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 often rational to implement a corelib or standardlib 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 5.6 shows the implementation of the '`exit`' subroutine in the RISC-V corelib. The task of this subroutine is to invoke a specific functionality 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 5.6 – The assembly implementation of the '`exit`' subroutine.

A *system call* (often abbreviated to *syscall*) is an invocation of a function of operating system's kernel. In Linux, more than 90 percent of the available system calls are implemented on all architectures. A common system call is '`exit`'. This function terminates the current process and performs various cleanup steps. Its first parameter is the exit status (or code) which can be checked by the shell or other programs [Lov13, p. 148]. In RISC-V Linux, the integer representation of a call to '`exit`' is 93 [Tor91].

In line 8 of Listing 5.6, 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`' instruction is used in order to place 93 in the value '`a7`'. On RISC-V Linux systems, the content of the '`a7`' register specifies the type of syscall to be performed. Here, 93 is placed inside this register since it represents the '`exit`' syscall. In line 12, the '`ecall`' instruction is used in order to call the program's environment [PW17, p. 23] Here, this call invokes the Linux kernel. In this example, some RISC-V assembly code was shown. The next subsection explains these concepts and idioms in more detail.

5.2.5. RISC-V Assembly

The Listing 5.7 shows a rush program containing two functions and a global variable. In line 1 of the rush program, the mutable global variable '`m`' is defined with the initial value 42. In line 4 of the main-function, '`m`' is incremented by 1. Next, in line 5, the '`foo`' function is called using '`m`' as the only call argument. In line 6, a '`return`' statement is used to terminate the main-function explicitly. The body of the '`foo`' function only contains a call to the '`exit`' function. Therefore, the '`foo`' function only exits using the specified parameter '`n`' as the exit code. In this case, the exit code of the displayed program will be 43. The code in Listing 5.6 on page 76 shows the output assembly generated from this program by the rush RISC-V compiler[7]. Because the assembler code of the '`foo`' function would take up too much space in the assembler program, it is intentionally omitted from this listing. Since the excluded function does not introduce any new concepts anyway, omitting it will not lead to a loss of explained concepts.

[7]Generated using Git commit '`e0fd6f7` '.

```
let mut m = 42;

fn main() {
    m += 1;
    foo(m);
    return;
}

fn foo(n: int) {
    exit(n)
}
```

Listing 5.7 – Example rush program containing two functions.

```
1  .global _start
2
3  .section .text
4
5  _start:
6      call main..main
7      li a0, 0
8      call exit
9
10 main..main:
11     # begin prologue
12     addi sp, sp, -16
13     sd fp, 8(sp)
14     sd ra, 0(sp)
15     addi fp, sp, 16
16     # end prologue
17     # begin body
18     li a0, 1
19     ld a1, m            # m
20     add a2, a1, a0
21     sd a2, m, t6
22     ld a0, m            # m
23     call main..foo
24     j epilogue_0        # return
25     # end body
26
27 epilogue_0:
28     ld fp, 8(sp)
29     ld ra, 0(sp)
30     addi sp, sp, 16
31     ret
   # ...
53 .section .data
54
55 m:
56     .dword 0x000000000000002a  # = 42
```

Figure 5.6 – Compiler output from the rush program in Listing 5.7.

In line 1, the '`.global`' assembler directive is used to declare the global symbol '`_start`' [PW17, p. 36]. On most architectures, the '`_start`' label indicates a program's entry point, therefore marking the first instruction to be executed [Zhi17, p. 19]. In line 5, the '`_start`' label is defined by placing a colon after its name. In line 6, the '`call`' instruction is used to call the '`main..main`' function. What strikes the eye here is that the already familiar '`main`' function is prepended by the '`main..`' prefix. Since this rush compiler implements name mangling[8], every function declared in a rush program will contain this prefix. However, unlike high-level function calls in LLVM, this call instruction is used alongside the previously explained low-level calling conventions of RISC-V.

In the next line, the '`li`' instruction is used to load the constant integer 0 into the register '`a0`' [PW17, reference card]. Like explained in the previous section about the RISC-V calling convention, the register '`a0`' is used for the first integer call argument. In line 8, the '`exit`' function is called, however, one cannot see the definition of this function in the current file. This is because the exit function is provided by the rush RISC-V corelib which was explained previously. Since 0 was previously placed inside the register for the first integer call argument, the '`exit`' function is called using 0 as the argument. Therefore, the instructions in the lines 7–8 are responsible for terminating the program using the exit code 0. These two instructions are always inserted at the end of the '`_start`' label in order to terminate

[8] Compilers often *mangle* names in order to create a unique name for every function [Lev00, pp. 119-120]

the program appropriately in case the rush code does not call '`exit`' on its own. This is required in order to prevent a segmentation fault which occurs if the program is not terminated properly.

Due to the function call in line 6, we will now shift our focus on the '`main..main`' label in line 10. In line 11, the first line of the '`main`' function, a comment indicates the beginning of the function's prologue. Just like demanded by the RISC-V calling convention, the rush compiler emits code for a *prologue* and an *epilogue* for each function.

As described in the previous sections about calling conventions, one task of the prologue is allocating stack space. In this prologue, the '`addi`' instruction in line 12 subtracts 16 from the value stored in '`sp`'. Since subtraction is used, the stack pointer is decremented, leading to the stack progressing into lower memory. Therefore, this instruction increases the size of the stack, thus allocating memory. Here, an addition instruction is used even though subtraction is required. In RISC-V, the '`addi`' instruction requires one register and one immediate value as its operands. Due to the third operand being an *immediate* value, the trailing '`i`' (*immediate*) appears in the instruction's name. Since this immediate value can be negative, an additional instruction for immediate subtraction is redundant [PW17, reference card]. This example shows how the RISC-V ISA omits redundant instructions in places where it is feasible. In this case, the stack pointer is decremented by 16 since two 8-byte values are stored on the stack in the lines 13 and 14. Just like described in the previous subsection about the stack, these registers are always saved on the stack.

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 register '`a0`'. Next, the '`ld`' instruction in line 19 is used in order to load the value of the global variable '`m`' into the register '`a0`' [PW17, reference card]. Global variables, like '`m`' in this example are saved under the '`.rodata`'. section or under the '`.data`' section if they are mutable. In this example, '`m`' is not declared as mutable and therefore saved under the '`.rodata`' section. The start of the '`.rodata`' section is represented by the '`.section`' assembler directive found in line 53. Here, a label called '`m`' is defined. In this label, the '`.dword`' directive is used to define the global initializer value of the variable. In RISC-V, this directive stores 64 bit of information in successive memory doublewords [PW17, p. 39]. The initializer value of the global variable is 42 and is represented as '`0x2a`' using hexadecimal in the assembly code. Since these data labels require their contents to be specified in hexadecimal, the trailing comment shows the base 10, human-readable version of the number. Because global variable are not saved on the stack, special instructions like '`ld`' are required to interact with global variables stored in the program's data sections.

At this point, the register '`a0`' would contain 1 and '`a1`' would contain 42. In line 20, the '`add`' instruction is used in order to save the sum of '`a0`' and '`a1`' in the register '`a2`'. Now, the value saved in '`a2`' would be 43. Next, the '`sd`' instruction in line 21 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 interesting 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].

In line 22, the previously explained '`ld`' instruction is used in order to load the value of the same variable into the register '`a0`'. Then, the '`call`' instruction in line 23 is used in order to call the '`foo`' function using the value of m as its argument. However, one cannot easily observe how call arguments are passed here. Like explained previously, the first integer argument of a function call must be placed in the register '`a0`'. Since '`m`' was loaded into '`a0`' previously, it will be used as the call argument for '`foo`' automatically. Therefore, the '`foo`' function is called using 43 as the first argument.

Since the ‘foo’ instruction only calls the ‘exit’ function, its explanation will not be beneficial for introducing new concepts. Therefore, we will omit the explanation of the assembler output of the ‘foo’ function. The final instruction of the main-function’s body is the ‘j’ instruction in line 24. This instruction will cause the CPU to jump to the address of the specified label. In this example, the CPU will jump to the first instruction of the ‘epilogue_0’ label [PW17, p. 17]. Therefore, the rush compiler uses the ‘call’ instruction for jumps caused by function calls and the ‘j’ instruction for jumps between blocks of the current function.

Like explained previously, every function has a *prologue* and an *epilogue*. Since one of the tasks handled by the epilogue is releasing resources allocated by the prologue, the function’s stack pointer is incremented in line 30. Finally, the ‘ret’ instruction in line 31 is used in order to jump back to the instruction whose address is specified in the ‘ra’ register [PW17, reference card].

5.2.6. Supporting Pointers

In Subsection 5.1.6, we have explained how pointers can be used in rush. Since every rush backend should support all features of the language, pointers also need to be implemented for the RISC-V architecture. In order to get a rough understanding of how pointers work in this compiler, the code in Listing 5.8 and 5.9 are to be considered.

```
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 – RISC-V assembly output generated from Listing 5.8.

The code in Listing 5.8 shows a rush program in which a variable is referenced to create a pointer which is then dereferenced. In line 2, the mutable variable ‘a’ is defined using an initial value of 42. Next, in line 3, the variable is referenced in order to use the resulting address to define the variable ‘to_a’. In line 4, the ‘to_a’ pointer variable is dereferenced in order to use the value of ‘a’ as the exit code of the program.

Listing 5.9 includes the most significant part of the compiler output which represents this rush program[9]. In line 18 of this listing, the integer value 42 is placed in the register ‘a0’. Next, ‘a0’ is saved on the stack at ‘-24(fp)’ using the ‘sd’ instruction. Like the comment suggests, the instruction in line 20 is used in order to reference a. Here, the ‘addi’ instruction is used to subtract 24 from the value stored in the ‘fp’ register. The result of this subtraction is saved in the register ‘a0’. Therefore, the register now contains the absolute memory address of the ‘a’ variable. Since the syntax ‘-24(fp)’ means that the variable is saved at $fp - 24$, the subtraction uses the exact same information which is already known about the variable. Here, instead of using the saved memory location of the variable like ‘x(fp)’, the information is used in order to calculate the absolute address of the target variable. This computation can only be performed at runtime since the value of ‘fp’ is not known at compile time. In line 21, the ‘a0’ register which contains the memory address is also saved on the stack. Therefore, the ‘a’ variable is saved at ‘-24(fp)’ while ‘to_a’ is saved at ‘-32(fp)’.

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.

[9] Generated using Git commit ‘e0fd6f7 ’.

Here, the '`ld`' instruction in line 22 of the assembly output is used. The instruction will load the memory address stored at '`-32(fp)`' (in '`to_a`') into the '`a0`' register. Next, another '`ld`' instruction in line 23 is used in order to load the value of the variable saved at the previously fetched memory address. What strikes the eye is that '`0(a0)`' instead of '`x(fp)`' is used for specifying the target memory address of the load instruction. In this case, '`0(a0)`' means that the instruction should load its value from the address saved in '`a0`' with an offset of 0. Since an offset of 0 is used, the practical description of the instruction is that it loads a value saved at the memory address specified in '`a0`'. Because '`a0`' contains the memory address of the '`a`' variable, the instruction loads 42 into the '`a0`' register.

5.2.7. Implementation: The rush compiler targeting RISC-V assembly

Just like the other compilers presented in this paper, this one also traverses the annotated AST using the postorder technique. Unlike the LLVM or WASM compiler, this compiler emits RISC-V assembly files which are later assembled by the assembler.

Struct Fields

Before any complex code samples can be considered, important struct fields of the compiler first need to be explained. The rust code in Listing 5.10 shows important struct fields of the rush compiler targeting RISC-V.

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

```rust
pub struct Compiler<'tree> {
    // ...
    pub(crate) blocks: Vec<Block<'tree>>,
    // ...
    pub(crate) curr_block: usize,
    /// Data section for storing global variables.
    pub(crate) data_section: Vec<DataObj>,
    /// Read-only data section for storing constant values (like floats).
    pub(crate) rodata_section: Vec<DataObj>,
    /// Holds metadata about the current function
    pub(crate) curr_fn: Option<Function>,
    /// Holds metadata about the current loop
    pub(crate) loops: Vec<Loop>,
    /// The first element is the root scope, the last element is the current scope.
    pub(crate) scopes: Vec<HashMap<&'tree str, Variable>>,
    /// Holds the global variables of the program.
    pub(crate) globals: HashMap<&'tree str, Variable>,
    /// Specifies all registers which are currently in use and may not be
    ↪ overwritten.
    pub(crate) used_registers: Vec<(Register, Size)>,
}
```

Listing 5.10 – Fields of the RISC-V '`Compiler`' struct.

In line 15, the field '`blocks`' is declared. This field holds a vector containing values of the type '`Block`'. That type provides an abstraction representing a label with its basic block in the assembly output. Therefore, this struct needs to contain a string field for its label and a vector for its instructions. The Listing 5.11 shows parts of the Rust code which declares the '`Instruction`' enum.

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

```
62  Jmp(Rc<str>),
    // ...
74  Not(IntRegister, IntRegister),
75  Add(IntRegister, IntRegister, IntRegister),
76  Addi(IntRegister, IntRegister, i64),
        // ...
114     match self {
```

Listing 5.11 – The '`Instruction`' enum in the RISC-V compiler.

Although the listing shows only a few of the implemented instructions, this enum contains all instruction variants which the compiler might need at a later point. Depending on the type of instruction, the corresponding enum includes fields which define its operands. For instance, the '`Li`' variant in line 74 represents the '`li`' instruction in assembly. This instruction loads the immediate integer value specified in the second operand into the register specified in the first operand. Due to this, the enum also contains a field for a value of the type '`IntRegister`' and a field for a 64-bit signed integer. The Rust code in Listing 5.12 shows parts of the declaration of the '`IntRegister`' enum. Like the name implies, this enum holds all possible registers which the architecture provides.

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

```
107     // temporaries
        // ...
137
138 impl IntRegister {
139     pub(crate) fn nth_param(n: usize) -> Option<Self> {
                // ...
145             4 => Self::A4,
```

Listing 5.12 – The '`IntRegister`' enum in the RISC-V compiler.

Even though just the integer registers are shown in this listing, a similar enum for floating-point registers also exists in this implementation. Another important struct field is declared in line 19 of Listing 5.10. The '`curr_block`' field saves the index of the basic block which is currently being inserted to. Next, the '`data_section`' and '`rodata_section`' fields in line 21 and 22 are declared. Just like the ELF sections, these vectors contain declarations of global variables in the program. Here, each vector holds items of the type '`DataObj`'. Since this type specifies data which should be saved in these sections, it contains a string field for the label and an enum field for the actual data saved in the object. For instance, if the compiler encounters a global variable declaration, a new data object is inserted into the correct section, depending on whether the global variable has been declared as mutable or not.

In line 25, the '`curr_fn`' field is declared. This field saves a value of the type '`Function`'. The '`Function`' struct contains a counter of the stack allocations of the current function in bytes and the label of the epilogue block of the current function. The former is incremented every time a variable declaration is compiled. This is required in order to allocate the correct amount of stack memory during a function's prologue.

Just like in the other compilers, this one also features a '`loops`' field which saves important labels of the current loop being compiled. This field is declared in line 27 and holds a vector of '`Loop`' structs. Each '`Loop`' struct saves the label of the loop's head and the label of the basic block which follows after the loop's body.

Furthermore, the '`scopes`' field in line 29 is managed to associate a variable to some important metadata. This metadata includes the variables type and its *stack memory position*. Just like explained in the previous sections, each cell of the stack memory is accessible by

specifying a unique index. The compiler saves this unique index in this HashMap so that it can refer back to the variable later. Of course, this only works for variables which are saved on the stack. However, these HashMaps are not used for saving global variables. Instead, the '`globals`' field in line 31 is used. Just like the HashMaps 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. Last, the '`used_registers`' field in line 33 is declared. It plays a vital role in the compiler's register allocation algorithm.

By only considering the compiler's struct fields, it has become apparent that this implementation provides several abstractions over the bare strings in which assembly is normally formatted. Due to this, implementation of the actual compiler is a lot more structured and reliable. During development of the compiler, this approach has often proven itself to be optimal.

Data Flow and Register Allocation

An important characteristic of a compiler is how it represents runtime data at compile time. In assembly, runtime data is represented by registers which will contain values at runtime. Since this rush compiler emits RISC-V assembly, it also represents data by using registers internally. In the previous paragraphs, we have learned how this implementation uses abstractions in order to represent assembly constructs, including registers.

For reference, the LLVM compiler represents runtime values by passing virtual registers internally. Similarly, this compiler also passes abstractions representing registers in order to represent the data flow of the program being compiled. Unlike in LLVM, there is only a finite amount of registers available. Therefore, this compiler also manages register allocation so that programs can be represented using this limited number of registers. In under to understand the implementation, the code in Listing 5.13 and Listing 5.13 is to be considered. The former listing contains a rush program which adds two integer variables together in order to use the result as its exit code. The latter shows parts 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 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 – Assembly output of the rush program in Listing 5.13.

In the lines 2 and 3 of the rush program, the integer variables '`a`' and '`b`' are declared. In line 4, the '`exit`' function is invoked, using the sum of these variables as the call argument.

In the line 22 of the assembly output, the runtime value of the '`a`' variable is loaded into the register '`a0`'. Next, the same operation is performed for the '`b`' variable. However, this time, the instruction writes its result in the '`a1`' register instead of the '`a0`' register. This is because the previously loaded value in '`a0`' would be overwritten if this was the case. Since both the value of the variable '`a`' and '`b`' are required for the addition, overwriting this register would result in a wrong calculation. This is an example of how register allocation is required in order to manage registers and to prevent such bugs.

Unlike the register allocation algorithm of a production ready compiler like LLVM, this one only aims to use registers without causing conflicts like the previously explained one. There-

fore, this algorithm does not emphasize performance and instead only performs mandatory task which are required for making a program work. For instance, all variables are saved on the stack in order to keep all registers unused and free for use in temporary calculations and operations like this one. Since this algorithm only performs rudimentary register allocation, its implementation is also significantly easier.

The core principle of the register allocator is that each method which can return a register decides which register it returns itself. In order to choose an output register, each of those methods uses the helper method '`alloc_ireg`'. The rust code in Listing 5.15 shows the '`alloc_ireg`' method of the RISC-V rush compiler.

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

```
175 pub(crate) fn alloc_ireg(&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     unreachable!("out of registers!")
186 }
```

Listing 5.15 – The '`alloc_ireg`' method of the RISC-V rush compiler.

This method returns the first available register from the register pool containing either integer registers[10]. Such a register pool contains all possible registers which can hold the data type of that class of registers. This compiler contains two pools: one for integer registers and one for floating-point registers. In line 175 of Listing 5.15, the signature of the method is shown. Here, one can see that the method returns a value of type '`IntRegister`'. The for-loop in line 176 is used to itrate through the entire '`INT_REGISTERS`' array. This array is constant and therefore represents the pool containing integer registers. In the lines 177–180, the if-expression checks if the current register ('`reg`') is not found in the '`used_registers`' vector. If this was the case, the current register would be unused and could therefore be returned. Since the '`return`' statement in line 182 returns the current register if it is unused, the loop only runs until a free register has been found. Due to the passive nature of the register allocation algorithm used in this compiler, the unreachable-macro in line 185 should never be called since the compiler should not run out of registers. However, one can see that this method does not mark the newly returned register as used. This is because marking a register as used is only required in some sections of the compiler where overwriting a register would introduce a bug in the output program. Therefore, if this method was to be called repeatedly without calls to other methods, it would always yield the same result register.

In order to mark a register as used, it is simply pushed into the '`used_registers`' vector. For this, a helper method called '`use_reg`' which only performs this simple call is implemented. Due to the simplicity of this method, a listing of its code is intentionally omitted. If a register is no longer used and can be overwritten again, it is also simply removed from the '`used_registers`' vector. For this, another simple helper method called '`release_reg`' is implemented. Just like the previous method, its implementation is so simple that a code listing is omitted.

Figure 5.7 shows an abstract representation of the compiler's integer register pool. Here, the pool contains all integer registers of the compiler. In this figure, only the first section of the pool which contains the '`ax`' registers can be seen. The gray cells at the beginning

[10]The "i" in '`alloc_ireg`' hints that it allocates integer registers

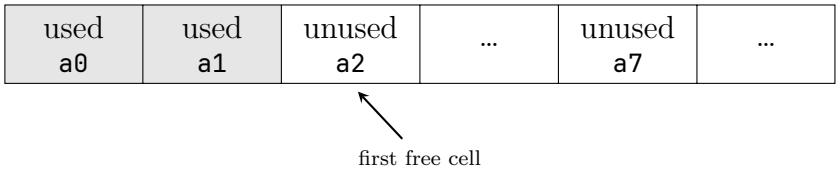


Figure 5.7 – Integer register pool of the RISC-V rush compiler.

indicate that these registers are currently in use by the compiler. Of course, this information is not saved in the register pool since the '`used_registers`' vector saves this data. In the figure, the arrow points to the first free register, meaning the one which follows the last used register '`a1`'. Therefore, if the '`alloc_ireg`' method was called when the state of the registers is equivalent to the one displayed in Figure 5.7, it would return the register '`a2`'. By considering this example, it has become apparent that the method always returns the first free integer register. In the compiler, an equivalent pool for float registers also exists.

Before register allocation can be explained using code samples, how registers are passed around in the compiler needs to be explained first. Here, we should focus on the compilation of simple expressions. For this, the rust code in Listing 5.16 is to be considered.

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

```
596 pub(crate) fn expression(&mut self, node: AnalyzedExpression<'tree>) ->
    ↪  Option<Register> {
597     match node {
598         AnalyzedExpression::Int(value) => {
599             let dest_reg = self.alloc_ireg();
600             self.insert(Instruction::Li(dest_reg, value));
601             Some(Register::Int(dest_reg))
602         }
            // ...
662     }
663 }
```

Listing 5.16 – Parts of the '`expression`' method in the RISC-V rush compiler.

The code in Listing 5.16 shows parts of the '`expression`' method. This method is responsible for compiling expressions in the RISC-V rush compiler. As the method's signature suggests, it takes a node representing an analyzed expression. This method returns an '`Option<Register>`' which is '`None`' if the infix-expression contained a call to the '`exit`' function. Otherwise, the method returns the register in which the result of its computation will be contained at runtime. The code in the lines 598-602 is executed if a constant integer expression is compiled. First, a target integer register is allocated by calling the previously shown '`alloc_ireg`' method. Now, the '`dest_reg`' variable contains the register which will contain the result of the expression. Next, an '`li`' instruction is inserted. At runtime, this instruction would load the constant integer contained in the variable '`value`' into the register specified '`dest_reg`'. Finally, the register which now contains the loaded value is returned. For more complex expressions, the corresponding methods are called, just like in the other compilers. Now it has become apparent how basic expressions work and how registers are used to hold values.

However, we still do not know when the compiler marks certain registers as used. In order to understand this problem, the compilation of the rush expression `n + m` is to be considered. The rust code in Listing 5.17 shows parts of the '`infix_expr`' method of the rush compiler.

Like the signature of the method suggests, it consumes an annotated tree-node representing an analyzed infix-expression. In line 853, the left-hand side expression is compiled, and its result register is saved in the variable '`lhs_reg`'. Next, in line 854, the previously discussed '`use_reg`' method is used in order to mark the previously allocated register as used. The

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

```
761 fn infix_expr(&mut self, node: AnalyzedInfixExpr<'tree>) -> Option<Register> {
                // ...
849             (lhs, rhs, op) => {
850                 let lhs_type = lhs.result_type();
851
852                 // compile RHS and mark the LHS register as used
853                 let lhs_reg = self.expression(lhs)?;
854                 self.use_reg(lhs_reg, Size::from(lhs_type));
855
856                 let rhs_reg = self.expression(rhs)?;
857
858                 // set LHS register as unused
859                 self.release_reg(lhs_reg);
860
861                 let res = self.infix_helper(lhs_reg, rhs_reg, op, lhs_type);
862
863                 Some(res)
864             }
                // ...
868 }
```

Listing 5.17 – Parts of the '`infix_expr`' method in the RISC-V rush compiler.

second argument to the method specifies the size of the data which the register holds, 64 bits for integers or 8 bits for characters for instance. This information is also saved in the '`used_registers`' vector and is used in case used registers need to be spilled. In line 856, the right-hand side is expression is compiled, and its result register is saved in the variable '`rhs_reg`'. After this, the register returned by the compilation of the left-hand side is marked as unused again. Finally, in line 861, the '`infix_helper`' method is called in order to insert the instruction for the actual calculation.

However, the reason the left-hand side register is being marked as used is not immediately apparent. As discussed previously, calling the methods responsible for allocating registers, like '`alloc_ireg`', repeatedly without marking registers as used results in the allocator method returning the same register. In this scenario, this would create an issue since both the left-hand side and the right-hand side would result in the identical register, '`a0`' for instance. Now, the compilation of the right-hand side expression would overwrite the value stored in the register of the left-hand expression, thus creating invalid output code. In order to mitigate this issue, the register of the left-hand side is marked as used as seen in Listing 5.17. Therefore, the allocator call caused by the compilation of the right-hand side will respect that the register returned by the left-hand side is currently in use and will therefore return the next one.

Next, we will consider the implementation of the '`infix_helper`' helper method. This method exists since translation of infix operators is required during compilation of both normal infix-expressions and assign-expressions with additional operations, '`a += 1`' for instance. The rust code in Listing 5.18 shows parts of this method.

As the method's signature suggests, the left-hand and right-hand side's registers and a type are used its parameters. Since this register abstraction does not contain type information, as registers are usually untyped, the type information conveys the type of either the left-hand or right-hand side expression. In case of infix-expressions, which side the type specifies is irrelevant since the analyzer demands that both are identical. Here, the type information is required order to insert the correct instruction.

In line 874 and 875, an integer and a floating-point output register is allocated. These registers are saved in a variable for each type, '`dest_regi`' for integers for instance. Of course, only one of these registers is later used. However, both are allocated in order to

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

```
871 fn infix_helper(&mut self, lhs: Register, rhs: Register, op: InfixOp, type_: Type)
    ↪  -> Register {
        // ...
874     let dest_regi = self.alloc_ireg();
875     let dest_regf = self.alloc_freg();
876
877     match (type_, op) {
878         (Type::Int(0), InfixOp::Plus) => {
879             self.insert(Instruction::Add(dest_regi, lhs.into(), rhs.into()));
880             dest_regi.into()
881         }
            // ...
987         (Type::Float(0), InfixOp::Minus) => {
988             self.insert(Instruction::Fsub(dest_regf, lhs.into(), rhs.into()));
989             dest_regf.into()
990         }
991         (Type::Float(0), InfixOp::Mul) => {
            // ...
1000    }
1001 }
```

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

avoid code duplication. In line 878-881, the code responsible for inserting an integer addition instruction can be seen. For this, the '`self.insert`' helper method is used. This inserts a new instruction into the current basic block. Due to its simplicity its explanation will be omitted. Here, the '`Instruction::Add`' enum variant is used. Since the first operand specifies the output reqister, the previously '`dest_regi`' variable is specified. For the first and second operands, the registers provided as arguments to the '`infix_helper`' method are used. The '`Add`' enum can only use integer registers, like '`a0`' as its operands. However, the parameters '`lhs`' and '`rhs`' can be registers of any type. For this, the conversion method '`into`' is used. In line 987–990, the code for inserting the float subtraction instruction can be seen. Since floating-point operations usually require different operators from the one used for integer operations, it becomes apparent why type information is passed to this method. Here, the '`fsub`' instruction enum variant is used to represent the addition of floating-point numbers. In this case, the variable '`dest_regf`' is used as the first operand since it can hold floating-point numbers.

Functions

However, before the infix-expression in Listing 5.13 is compiled, the compiler considers the functions of the program. In this case, only the '`main`' function is present. The rust code in Listing 5.19 shows the '`declare_main_fn`' method of the rush RISC-V compiler.

Like its signature suggests, the method takes a block, meaning a list of statements as its input. In the lines 124–126, the '`_start`' label is created and marked as exported so that the linker will later find it. After this insertion, the '`curr_block`' index is 0, meaning that the block in which is inserted to is the one of the '`_start`' label.

In the 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 rush. Next, the line 132 inserts a '`call`' instruction which is responsible for calling the '`main`' function when the program is executed. In the lines 134–135, instructions for calling the '`exit`' function using 0 as the exit code are inserted. These two instructions are necessary in order to prevent a segmentation fault at the end of the program.

In the lines 137–139, a new epilogue label is generated. Then, the current function is

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

```
123 fn declare_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     // prologue is inserted after the body (because it now knows about the frame
    ↪ size)
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 – The '`declare_main_fn`' method of the RISC-V rush compiler.

updated so that it contains this newly created epilogue label. However, this label is not directly added to the '`blocks`' vector since it should ideally appear after the body of the function.

Next, in the lines 142–145, 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 using the helper method '`insert_at`' in line 142. In line 143, a new variable scope is added for the function's body. Then, the function's body is compiled by calling the '`function_body`' method in line 144. After the body's compilation, the previously added scope is removed again in line 145.

Now, the code responsible for inserting the prologue is executed. The prologue is inserted after the function's body even though it introduces each function. Since the required memory of the function is not known before the function's body is compiled, no stack space can be allocated before the body of the functions was traversed. In order to mitigate this issue, the code responsible for a function's prologue is generated after the entire function body has been traversed.

The call to the method '`prologue`' in line 148 generates the instructions which represent the prologue. Next, in line 149, the insertion position is moved back to the end of the '`main..main`' label. In line 150, the instructions generated from the function's body are concatenated to the end of prologue instructions. This way, the variable '`prologue`' now represents a vector containing the instructions of the main function's prologue followed by the

ones of its body. In line 151, the instructions of the current block, meaning the '`main..main`' block, are overwritten with the contents of the variable. In other words, the current block now contains the function's prologue and body.

In the lines 152–155, the code responsible for inserting the trailing epilogue code can be seen. For this, in line 153, the previously created epilogue label is now appended to the '`blocks`' vector. In line 154, the '`epilogue`' method of the compiler is callled. The rust code in Listing 5.20 shows the '`epilogue`' method of the rush RISC-V compiler.

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 – The '`epilogue`' method of the RISC-V rush compiler.

As the context of the previous code snipped suggests, this method is called in order to generate the instructions for a function's epilogue. First, in the lines 67–68, the insertion position of the compiler is updated to the end of the epilogue label. This is done since the epilogue must exist in a separate label in order to be reused when '`return`' statements are encontered. The instruction insertion in line 70 is responsible for a '`ld`' instruction. As described previously, this particular instruction restores the previously saved '`fp`' register. Next, in line 75, another '`ld`' instruction for restoring the '`ra`' register is inserted. In line 80, an '`addi`' instruction is inserted in order to restore the previously allocated stack space. Finally, a '`ret`' instruction is inserted which is responsible for transferring control back to the caller. Now it becomes apparent that the inserted instructions resemble the already known patters observed in previously shown listings.

As for the code in Listing 5.19, the presented method is very similar to the one which translates other rush functions. However, the '`function_declaration`' method which is responsible for other functions differs in two ways. First, it contains code which handles function parameters. Second, the special instructions for calling to the '`main`' and '`exi`' functions are not required for other functions.

Let Statements

In line 2 and 3 of the rush program in Listing 5.13, let statements are used in order to save values on the stack. For this, the rush RISC-V compiler implements a separate method named '`let_statement`'. The rust code in Listing 5.21 shows the '`let_statement`' method of the rush RISC-V compiler.

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

```
554 fn let_statement(&mut self, node: AnalyzedLetStmt<'tree>) {
555     let type_ = node.expr.result_type();
556
557     // save the expression result on the stack & insert the variable into the
    ↪  current scope
558     let ptr = self.save_expr_on_stack(node.expr, format!("let {}", node.name));
559     self.scope_mut()
560         .insert(node.name, Variable { type_, value: ptr });
561 }
```

Listing 5.21 – The '`let_statement`' method of the RISC-V rush compiler.

This method only calls the '`save_expr_on_stack`' method in order to save the resulting memory address in the current scope. The purpose of the '`save_expr_on_stack`' is explained later. That method returns an abstraction which represents a memory location on the stack. In line 558 of Listing 5.21, this method is called and the resulting relative memory address is saved in the variable '`ptr`'. In the next line, this memory address is inserted into the current scope, thus associating the name of the compiled variable in to its relative memory location. This data is saved so that the variable can be accessed later. Since the main work is accomplished by the '`save_expr_on_stack`' method, it will now be considered in detail. The rust code in Listing 5.22 displays parts of the '`save_expr_on_stack`' which was called in line 558 of the previous listing.

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

```
563 fn save_expr_on_stack(/* ... */) -> Option<Pointer> {
568     let type_ = node.result_type();
569     let reg = self.expression(node)?;
570     let comment = format!("{comment_prefix} = {reg}").into();
571     let offset = self.get_offset(Size::from(type_));
572
573     match reg {
        // ...
586         Register::Float(reg) => self.insert_w_comment(
587             Instruction::Fsd(reg, Pointer::Register(IntRegister::Fp, offset)),
588             comment,
589         ),
590     };
591
592     Some(Pointer::Register(IntRegister::Fp, offset))
593 }
```

Listing 5.22 – The '`save_expr_on_stack`' Method in the RISC-V rush compiler

This method takes an expression in order to save it on the stack. As its signature suggests, the method returns an '`Option<Pointer>`'. Just like other methods, this one will return '`None`' if the passed expression yields no value. Otherwise, the method returns the relative memory address which will contain the value of the saved expression at runtime. In line 569, the passed expression is compiled, and the resulting register is saved in the variable '`reg`'. In line 571, the '`get_offset`' method is used in order to calculate the fp-offset of variable. Since types like char or bool require less memory, the data type of the expression is also passed to the helper method. Now, the '`offset`' variable contains the fp-offset at which the variable can 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 components like the prologue or epilogue are aware of this stack allocation. The '`match`' block in line 573 executes different code based on the register type which the expression yielded. In the lines 586–589, the code which is executed if the expression yielded a float register can be seen. Here, the '`fsd`'

instruction is used in order to save the yielded register at the memory position relative to '`fp`' which is specified in the variable '`offset`'. The code for integer registers is omitted since it also differentiates between 8-bit and 64-bit data. In line 592, 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

In line 4 of the rush program in Listing 5.13, the '`exit`' function is called. Even though the presented '`exit`' function is special, the rush RISC-V compiler is able to translate all sorts of different call-expressions using one internal method. The rust code in Listing 5.23 shows the first part of the '`call_expr`' method in the RISC-V rush compiler. This listing only shows the first part of the method since it is too large to be included in a single-page listing.

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

```rust
92  pub(crate) fn call_expr(&mut self, node: AnalyzedCallExpr<'tree>) ->
    ↪  Option<Register> {
93      // before the function is called, all currently used registers are saved
94      let mut regs_on_stack = vec![];
95
96      for (reg, size) in self.used_registers.clone() {
97          let offset = self.spill_reg(reg, size);
98          regs_on_stack.push((reg, offset, size));
99      }
100
101     // save the previous state of the used registers
102     let used_regs_prev = mem::take(&mut self.used_registers);
103
104     // specifies the argument position of the specified register type
105     // type dependent: (`a0` -> `int_cnt = 0`), (`fa0` -> `float_cnt = 0`)
106     let mut float_cnt = 0;
107     let mut int_cnt = 0;
108
109     // calculate the total byte size of spilled params
110     let mut spill_param_size = 0;
111     for arg in &node.args {
112         match arg.result_type() {
113             Type::Unit | Type::Never | Type::Unknown => continue,
114             Type::Float(0) => {
115                 if FloatRegister::nth_param(float_cnt).is_none() {
116                     spill_param_size += 8;
117                 }
118                 float_cnt += 1;
119             }
120             Type::Int(_) | Type::Bool(_) | Type::Char(_) | Type::Float(_) => {
                // ...
127     }
128
    // ...
235 }
```

Listing 5.23 – The '`call_expr`' method in the RISC-V rush compiler.

Since this method is responsible for translating function calls, its implementation has proven to be very complex and demanding. As the method's signature suggests, it consumes an analyzed call-expression in order to return an optional register. No register is returned if a function which returns the '`()`' type or the '`!`' type is called. In line 4 of the code in Listing 5.13, this method would return no register because the '`exit`' function is called.

The parts of code in the shown listing are responsible for preparing the compiler for compilation of the function call. In the lines 93–102, all registers which are used at the time of the function call are saved on the stack. This is required since the register allocator of this compiler is so simple that it only considers a function at the time. In production-ready compiler systems, a register allocation algorithm could also work in an interprocedural manner. When considering this practical example, it becomes apparent that saving **all** used registers does not produce the most efficient output code. However, the presented algorithm remains relatively simple. Furthermore, throughout the extensive testing conducted on the rush backends, this approach has presented itself as reliable[11]. In line 95, a new vector named '`regs_on_stack`' which will later contain registers is created. The for loop in the lines 96–99 is used to spill every used register to the stack. For this, the helper method '`spill_reg`' is used. Due to its similarity to the previously explained '`save_expr_on_stack`' method, this method will not be explained further. After a register has been saved on the stack, it is added to the previously declared vector. After the loop, in line 102, the '`used_registers`' vector is moved into the local '`used_regs_prev`' variable. Therefore, after the assignment, the '`used_registers`' vector will be empty due to its contents being moved into the '`used_regs_prev`' variable. Through this, the compilation of the argument expressions can use more registers since the spilled registers can be overwritten.

In the lines 104–128, the code determines how much additional memory is required in order to place arguments on the stack, which could not fit into registers. For most calls, this required memory is usually 0. However, a call with 9 integer arguments would need 8 additional bytes of memory because the ninth argument would have to be placed on the stack. The final amount of additional memory is saved in the '`spill_param_size`' variable. Determination of the required size is performed by the for-loop in line 111. This loop iterates over each argument expression of the call. For each expression, the match-block in line 112 executes different code depending on the rush data type of the expression. For instance, expressions which yield in '`()`' or '`!`' do not require registers and can thus be skipped using the '`continue`' keyword in line 113. Expressions yielding float variables however are considered in the line 114. Here, the '`nth_param`' method on the '`FloatRegister`' enum is called.

The rust code in Listing 5.24 shows parts of this method.

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

```
215 pub(crate) fn nth_param(n: usize) -> Option<Self> {
216     Some(match n {
217         0 => Self::Fa0,
          // ...
224         7 => Self::Fa7,
225         _ => return None,
226     })
227 }
```

Listing 5.24 – The '`nth_param`' method of the RISC-V rush compiler.

This method associates an index of a parameter to a register which could represent this position. For instance, the index 2 represents the third parameter, which is represented by the register '`fa2`'. For any other indices, the method return '`None`'. Therefore, the if-expression in line 115 of Listing 5.23 checks if there is no longer a register which could represent the current argument position. This position is saved in the '`float_cnt`' and '`int_cnt`' variables which are incremented during some iterations of the for-loop. Here, two counters are used in order to preserve independence between integer and float arguments. If there is no register for representing the current '`float_cnt`', the variable's value is greater than 7 and thus

[11] For each new commit, 38 integration tests are conducted on every backend

exceeds the capacity provided by the eight float parameter registers. If this is was case, the '`spill_param_size`' variable is incremented by 8 bytes since in rush, a floating-point number requires 64 bits of information. At the end of the float-specific block, the '`float_cnt`' is incremented by one. This way, the for loop is able to determine if there are any additional memory requirements of the current function. In line 4 if the code in Listing 5.13, the '`exit`' function is called with one integer argument. After the code displayed in Listing **??** has executed, the runtime value of the '`int_cnt`' variable is 1 while the '`spill_param_size`' variable holds 0. If the exit function was to be called using two integer registers, the '`int_cnt`' variable would hold the value 2. Through this example, it becomes apparent what the task of the code displayed in Listing 5.23 are.

However, only the first part of the '`call_expr`' method is shown in this listing. The rust code in Listing 5.25 shows the second part of the method.

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

```
130     int_cnt = 0;
131     float_cnt = 0;
132
133     // will later contain all registers used as params (needed for releasing locks
    ↪ later)
134     let mut param_regs = vec![];
135
136     // if there are caller saved registers, allocate space on the stack
137     if spill_param_size > 0 {
138         self.insert(Instruction::Addi(
139             IntRegister::Sp,
140             IntRegister::Sp,
141             -spill_param_size,
142         ));
143     }
144
145     // specifies the count of the current register spill
146     let mut spill_count = 0;
147
148     for arg in node.args {
149         match arg.result_type() {
150             Type::Unit | Type::Never | Type::Unknown => {
151                 self.expression(arg);
152             }
153             Type::Float(0) => {
154                 let res_reg = self.expression(arg).expect("`None` filtered above");
155                 match FloatRegister::nth_param(float_cnt) {
156                     Some(curr_reg) => {
157                         param_regs.push(curr_reg.to_reg());
158                         self.use_reg(curr_reg.to_reg(), Size::Dword);
159                     }
160                     None => {
161                         // no more param registers: spilling required
162                         self.insert_w_comment(
163                             Instruction::Fsd(
164                                 res_reg.into(),
165                                 Pointer::Register(IntRegister::Sp, spill_count *
    ↪ 8),
166                             ),
167                             format!("{} byte param spill",
    ↪ Size::Dword.byte_count(),).into(),
168                         );
169                         spill_count += 1;
170                     }
171                 }
172                 float_cnt += 1;
173             }
                // ...
198         }
199     }
    // ...
235 }
```

Listing 5.25 – The second part of the '`call_expr`' method of the RISC-V rush compiler.

In the lines 130–131, the '`int_cnt`' and '`float_cnt`' variables are reset to 0. In line 134, a new empty vector named '`params_regs`' is created. The lines 137–143 are responsible for inserting an '`addi`' instruction in order to allocate stack space. However, this instruction is only inserted if the size of the spilled parameters is non-zero, meaning that parameters have been spilled. If this was the case, inserting an instruction in order to allocate memory is required.

Next, the for-loop in line 148 iterates over the argument expressions. Just like in the last listings, the match-block in line 149 executes different code based on the data type of the argument expression. In case of unit or never types, the expression is compiled and the resulting value is ignored. This is reasonable since these types of expressions yield no register. For other data types, like floats, other code is executed. Expressions which yield floats for instance are handled in line 153. In the next line, the expression is compiled and the resulting register is saved in the variable '`res_reg`'. In line 155, different code depending on whether the current argument needs to be spilled is executed. If the current expression does not need to be spilled, the code after the line 156 is executed. Here, '`res_reg`' is only marked as used and pushed in the '`param_regs`' vector. Interestingly, 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 resulting value of argument expressions will always be in the correct registers. However, if there is no register for the current argument, the value has to be spilled onto the stack. For this, a '`fsd`' instruction (or '`sd`' instruction for integers) is inserted in order to save the result of the current argument on the stack. As described in a previous section 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. For reference, this process is described in detail in Figure 5.5 in Subsection 5.2.3 on page 73. In order to calculate this increasing offset easily, the '`spill_count`' variable is declared in line 146. Every time an argument is spilled, this counter increases. In line 165, the code calculating the offset relative to '`sp`' can be seen. Here, the value of '`spill_count`' is simply multiplied by 8 in order to obtain the correct offset. Therefore, the memory offset o is calculated like this: $o = \text{count}_{\text{spilled}} \times 8$. In line 169, the '`spill_count`' variable is incremented like described previously. Although just the code for floats is shown here, the algorithm for integer-based types only deviates slightly from the one for floats.

Now it has become clear how the argument expressions are compiled. Even until now, not the entire method has been shown. The rust code in Listing 5.25 shows the last part of the method.

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

```
201     // perform function call
202     let func_label = match node.func {
203         "exit" => {
204             // mark the current block as terminated (avoid future useless jumps)
205             self.curr_block_mut().is_terminated = true;
206             "exit".into()
207         }
208         func => format!("main..{func}").into(),
209     };
210     self.insert(Instruction::Call(func_label));
211 
212     // if there were spilled params, deallocate stack space
213     if spill_param_size > 0 {
214         self.insert(Instruction::Addi(
215             IntRegister::Sp,
216             IntRegister::Sp,
217             spill_param_size,
218         ));
219     }
220 
221     // restore the old list of used registers
222     self.used_registers = used_regs_prev;
223 
224     let res_reg = match node.result_type {
225         Type::Float(0) => Some(FloatRegister::Fa0.to_reg()),
226         Type::Int(_) | Type::Char(_) | Type::Bool(_) | Type::Float(_) => {
227             Some(IntRegister::A0.to_reg())
228         }
229         Type::Unit | Type::Never => None,
230         Type::Unknown => unreachable!("analyzer would have failed"),
231     };
232 
233     // restore all caller saved registers again
234     self.restore_regs_after_call(res_reg, regs_on_stack)
235 }
```

Listing 5.26 – The third part of the '`call_expr`' method of the RISC-V rush compiler.

The code in the lines 201–209 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 201, the '`call`' instruction responsible for invoking the function call is inserted. At this point, the arguments are in their correct places so that the called function will be able to access them. Therefore, the instruction can be inserted without additional effort.

The code in the lines 212–219 is responsible for restoring stack space which was allocated in order to fit spilled arguments. Of course, the responsible '`addi`' instruction is only inserted if there are indeed spilled arguments. In line 222, 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 arguments to happen independently of the rest of the compiled code. The code in the lines 224–231 is responsible for selecting an appropriate output register based on the function's return type. For instance, functions which return '`float`' values place their return value inside the register '`fa0`'. Therefore, the variable '`res_reg`' represents the register containing the called function's return value. For functions which return the '`()`' or '`!`' type, no register is selected and '`None`' is returned. The invocation of the helper method '`restore_regs_after_call`' is used in order to insert instructions which restore the previously spilled registers after the call. The '`used_registers`' vector could only be completely emptied due to the prior save of all used registers, these restored registers need to be restored after the call has completed. What strikes the eye here is that the method takes the '`res_reg`'

variable as its first argument.

For instance, a function might return a value of type '`int`', therefore making '`a0`' the content of '`res_reg`'. If the register '`a0`' was used before the function call, it would have to be restored by the helper method. However, restoring the register would result in the function's return value to be overwritten, therefore creating a bug. In order to mitigate this issue, the method will alter the result register by inserting a '`mv`' 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 making it the return value of the '`call_expr`' method.

Now it has become apparent how function calls are implemented in the rush RISC-V compiler. However, the question of how a called function returns a value still remains. In the previous sections, we have covered that every generated function contains an '`epilogue_x`'. Every time the '`return`' keyword is used in rush, the compiler generates an instruction which jumps to the epilogue label of the function being compiled. If the function should a value, meaning that the '`return`' statement contains an expression, it is first compiled in order to observe its output register. If the output register is already the desired register, like '`a0`' for integer values, the compiler can carry on. However, if the output register of the compiled expression deviates from the register in which return values shall be passed, the compiler inserts a '`mv`' instructions prior to the instruction jumping to the epilogue label. This way, values can be returned at any time while stopping execution of the function. For implicit returns, meaning trailing expressions which are not terminated by a semicolon, this procedure is identical.

Loops

Since most assembly dialects do not provide high-level control-flow scaffolds like loops, a compiler targeting assembly has to generate code which behaves like a loop. The code in Listing 5.27 shows a rush program containing a while-loop. On the right side, Listing 5.28 shows parts of the assembler output generated from the rush program.

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

Listing 5.27 – 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 5.28 – Assembly output of rush program in Listing 5.27.

The rush program in Listing 5.27 contains a while loop which will iterate ten times. At the beginning of the program, the integer variable '`count`' is declared. Before each iteration, the while-loop checks that the condition in its head is true. If the condition was true, the code inside the loop is executed. Otherwise, the execution of the loop stops and the code after the loop is executed. In the assembly output in Listing 5.28, the contents of the loop is represented by the instructions inside the '`while_head_0`' label. The instructions generated

from code before the loop are not visible but appear before the code shown in the listing. As the comment in line 22 suggests, the first instructions after the label declaration represent the loop control code. These instructions are executed every time time loop begins a new iteration, therefore they check if the condition is true before any of the code inside the loop's body is executed. If the condition is false, the execution of the loop should stop.

This is accomplished by the '`beqz`' instruction in line 26. This instruction is a pseudoinstruction which jumps to the target label if the value inside the first operand is zero [WA19, p. 105]. Here, this would mean that the instruction would jump to the '`after_while_0`' label if the condition of the loop evaluated to false. Just like the name of the '`after_loop`' label suggests, it contains code which follows the while-statement. Therefore, in this example, it contains the instructions calling the '`exit`' function with the code saved in the variable '`count`'. However, if the condition evaluated to '`true`', the '`beqz`' instruction would execute without causing a jump. In this case, the instructions following the comment in line 27 are executed. As the comment suggests, these instructions represent the code inside the loop's body. The instructions in the lines 28–31 represent the expression '`count += 1`' in line 4 of the rush code. In line 32, an unconditional '`j`' instruction jumps back to the beginning of the '`while_head_0`' label. Since this jump happens unconditionally, this instructions introduces the iteration for which the loop is desired. Therefore, at the end of a loop, an instruction jumps back to the start, where the condition is then evaluated again. This way, the compiler is able to generate instructions for the '`loop`', '`while`', and '`for`' statements. However, '`loop`' statements lack the control code which checks a condition as there is no condition in these loops. Furthermore, '`for`' loops contain code representing its update expression at the end of their body. By using this approach, the compiler is also able to generate nested loops reliably. Due to the simplicity of the presented approach, rust code from the compiler is omitted.

5.3. x86_64: Compiling to a CISC Architecture

In addition to *reduced instruction set computers* (*RISC*), there are also *complex instruction set computers*, short *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*[12] 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[13] has about 100 different instructions [WA19, Chapter 24], x64 is estimated to have about 3600 instructions [RA17]. This has the simple reason that at the time when most CISC architectures were developed, many developers still programmed in assembly languages by hand, without the use of compilers. Additional instructions for higher level concepts were therefore very helpful [Dan05a, p. 9].

One example for such an instruction is the '`leave`' instruction from x64. It is used to release the current stack frame at the end of a procedure. For RISC-V, Listing 5.20 on page 87 shows how the same is accomplished 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 popped off 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 normal instruction, just like every other one.

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, and stores the result in the '`%rax`' register and the remainder in the '`%rdx`' register. This makes any usage of such instructions significantly more demanding, as values in these registers that should be preserved must be spilled to the stack temporarily.

The substantial growth of the x64 instruction set and its commitment to backwards compatibility has also led to a number of instructions that have long become redundant. 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].
TODO: maybe explain what BCD is?

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).

```
let mut a = 2;

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

Listing 5.29 – Example rush program.

Listing 5.29 contains another simple example 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 5.30. It begins with the '`.intel_syntax`' directive, indicating that the following assembly code uses the Intel syntax.

[12] later shortened to *x64*
[13] The used extensions are RV32I, RV64I, RV32M, RV64M, RV32D, and RV64D.

Then, similar to the RISC-V assembly code shown in the previous section, the '`_start`' symbol is marked as global, as this is again the entry point. The sections are also the same, as these are defined by the ELF file format which is independent of ISAs.

```
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 5.30 – Compiler output from the rush program in Listing 5.29.

The main differences between this and the 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 uses '`-1(fp)`' to use the value in the '`fp`' register as a memory address, offset by '-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.

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 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 usually shortened to only include the first letter of the factor, for instance a double word is called *dword*. In some instances, see line 28, alternative short forms are used. Here the directive '`.quad`' inserts a quadruple word.

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

5.3.2. Registers

The base x64 ISA provides 16 general purpose registers for 64-bit integers. These are shown in Table 5.2. Additionally, three more sets of 16 registers are defined, each half the size as 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

Table 5.2 – Integer registers on 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`		

byte $(2a)_{16}$ into '`%al`', the least significant byte of '`%ax`', '`%eax`', and '`%rax`' changes to $(2a)_{16}$, too. The upper bytes are untouched, however. That means, if for example '`%ax`' contains the value $(a455)_{16}$ before writing, it will contain $(a42a)_{16}$ afterwards, not $(002a)_{16}$.

These registers for smaller sizes are not present for legacy reasons though. Considering line 18 from Listing 5.30, the move of the value behind the '`byte ptr`' requires the destination register to 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 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 to 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. These read and write to the global '`a`' variable, defined at the '`main..a`' symbol. Accesses like this simply require the offset by the instruction pointer in x64 Assembly.

In addition to these general purpose registers, x64 also defines 16 registers for floating-point number calculations, each 128 bits in size and labelled '`%xmm0`' through '`%xmm15`'. For the purposes of rush, only the lower 64 bits are ever used, since this is enough for 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.

TODO: explain pointers? (should be the same as RISC-V)

5.3.3. Stack Layout and Calling Convention

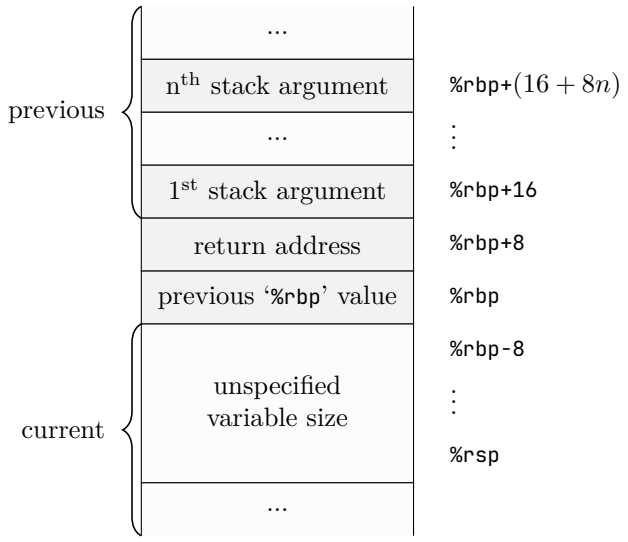


Figure 5.8 – Stack layout of the x64 architecture [Lu+22, p. 21].

Alongside the instructions and registers defined by x64 itself, the *System V* ABI defines how programs should structure the stack during execution. Figure 5.8 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 is different 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, but 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].

5.3.4. Implementation: The rush compiler targeting x_64

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

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

Listing 5.31 – The '`Instruction`' definition in the rush x64 compiler.

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 and did not have any reference implementation. One such difference is that the x64 compiler's '`Instruction`' definition which is shown in Listing 5.31 not only includes actual instructions, but also directives and symbols. 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 then simply emits the respective assembly representation.

Line 20 shows the definition of the '`.section`' directive, simply holding an instance of another enumeration containing the valid sections. Line 21 defines an assembly symbol, holding its name and a boolean whether a blank line should be added above. **TODO: did I say what symbols / labels are?** The name is saved as an '`Rc<str>`' instead of a '`String`' here to prevent some unnecessary cloning in some places.

The normal instructions, e.g., '`add`' and '`sub`', also cannot simply take registers as operands, as x64 also allows memory pointers and immediate values in many places. Instead, another enumeration, called '`IntValue`', is introduced to allow these three variants. Listing 5.32 shows its definition.

Struct Fields

The compiler struct itself shown in Listing 5.33 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. Both compilers have a '`scopes`' field for storing variable locations, but the RISC-V compiler has the '`Option<_>`' inside the '`Variable`' struct. Both compilers store the global variables, but this compiler has separate fields for every size for easier alignment in the static memory which is not required for RISC-V. **TODO: is this true?** The same applies for constants in the '`.rodata`' section. Both compilers have one field for accumulating the final assembly code, but this compiler saves a list of '`Instruction`'s instead of '`Block`'s.

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

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

Listing 5.32 – The '`IntValue`' definition in the rush x64 compiler.

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 5.33 – The x64 '`Compiler`' struct definition.

Additional fields of this compiler are '`function_body`', '`stack_pointer`', and '`frame_size`'. The '`function_body`' field stores the instructions for the currently compiled function. During compilation of statements and expressions, emitted instructions are always appended to this list. The method for compiling a function body then produces the prologue and afterwards appends the list contents onto '`text_section`'. This is required, because some values like the '`frame_size`' that are needed in the prologue cannot be known beforehand, but must appear before the function 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 saved by increasing this stack pointer. This process is done by the '`push_to_stack`' method visible in Listing 5.34. It takes a value, its size, and an optional comment as arguments. At first, the stack pointer is aligned to a multiple of the values' size by the '`align`' function. Then, the number of bytes required for the value is added to it, and the frame size is increased 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 an 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 5.34 – Stack space reservation for values.

Register Allocation

The register allocation algorithm of the x64 compiler differs quite a lot, too. 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 from Listing 5.35 on the last used register is called, returning the next register in line, which is then added to the '`used_registers`' field. For freeing a register, the compiler just pops the last register off this list. This only works, because the rest of the compiler guarantees used registers to be freed in exactly the reverse order they were reserved 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 5.35 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 be considered in Listing 5.36 along with the produced assembly code in Listing 5.37. Though not explicitly labelled, 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 left out, and the epilogue does not contain a '`leave`' instruction for freeing the memory. The '`foo`' function however does declare a local variable and thus requires the stack space to be allocated in the prologue. Lines 20–22 are responsible for this. As previously shown in Figure 5.8, 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. Afterwards, the base pointer's value is set to the current stack pointer. The stack pointer itself is then decremented by the number of bytes needed for the function which is saved in '`frame_size`', rounded up to a multiple of 16.

The epilogue ranges from line 29 to line 31. It defines a symbol where '`return`' statements can jump to, releases the stack frame using the '`leave`' instruction, and returns control to the caller with '`ret`'. Usage of the return symbol can be seen in line 26, which does an unconditional jump to there, rendering all intermediate instructions as unreachable. This is the expected behavior, because the jump instruction was emitted for 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 correspondingly sized variant, in advance.

TODO: '`function_body`' definition?

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

```rust
pub fn next(&self) -> Self {
    match self.in_qword_size() {
        Self::Rax => Self::Rdi,
        Self::Rdi => Self::Rsi,
        Self::Rsi => Self::Rdx,
        Self::Rdx => Self::Rcx,
        Self::Rcx => Self::R8,
        Self::R8 => Self::R9,
        Self::R9 => Self::R10,
        Self::R10 => Self::R11,
        Self::R11 => Self::R12,
        Self::R12 => Self::R13,
        Self::R13 => Self::R14,
        Self::R14 => Self::R15,
        Self::R15 => Self::Rbx,
        reg => panic!(/* ... */),
    }
}
```

Listing 5.35 – Register allocation in the rush x64 compiler.

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

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

Listing 5.36 – Another example rush program with two functions.

```
main..main:
    mov         %rdi, 21
    call        main..foo
    mov         %rdi, %rax
    call        exit
main..main.return:
    ret

main..foo:
    push        %rbp
    mov         %rbp, %rsp
    sub         %rsp, 16
    mov         qword ptr [%rbp-8], %rdi  # param n = %rdi
    mov         %rax, qword ptr [%rbp-8]
    imul        %rax, 2
    jmp         main..foo.return          # return %rax;
    mov         %rax, qword ptr [%rbp-8]
    mov         qword ptr [%rbp-16], %rax # let b = %rax
main..foo.return:
    leave
    ret
```

Listing 5.37 – Trimmed compiler output for the rush program in Listing 5.36.

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 in turn, and moves their result into the appropriate register, or onto

the stack if no registers are left. As the passing of the first six integer arguments is done 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 content has to be spilled to 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 inner function call, here '`bar`', wants to move its first argument '`'x'`' into '`%rdi`', it notices this register is already used for the first argument of the outer function call, '`42`'.

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 5.37 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.

TODO: Write something one Corelib?

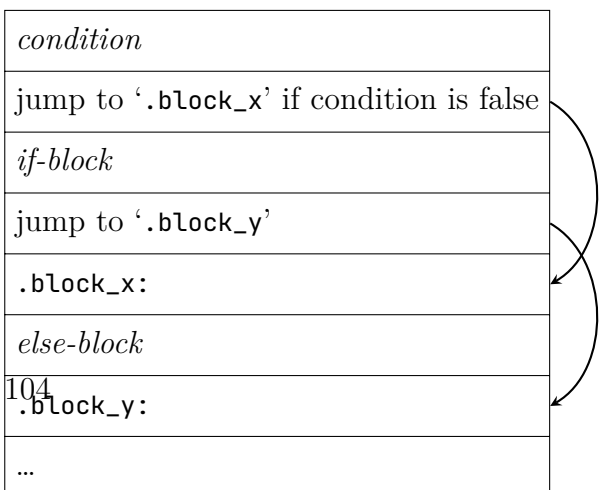
Control Flow

```
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 5.38 – A rush example function containing if-expressions and a loop.

Just as with RISC-V, control flow constructs like loops and if-branches must be represented using simple jumps in assembly. There are both conditional and unconditional jumps, the latter of which was already slightly touched upon when explaining '`return`' statements in a previous section. An unconditional jump, done in x64 with the '`jmp`' instruction, only takes a symbol 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. The way x64 handles conditions again differs from RISC-V. It provides a set of flags 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.

Listing 5.38 shows one last example rush function with the produced assembly code in Listing 5.39, this time containing if-expressions and a loop with one '`break`' and one '`continue`' statement. The purpose of this function is nonexistent, and the semantic analyzer rightfully complains about unreachable statements and unused variables, but it still serves to show translation of these constructs to assembly.



Compiled if-expressions always follow the same outline as it is shown in Figure 5.9. 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. Following that are the instructions for the if-block, concluded by an unconditional jump to behind the else-block. Now, the first symbol is inserted, here '`.block_x`', followed by the else-block. The

```
17 main..bar:
18     push        %rbp
19     mov         %rbp, %rsp
20     sub         %rsp, 32
21     mov         byte ptr [%rbp-1], %dil                 # param a = %dil
22     test        byte ptr [%rbp-1], 1
23     je          .block_0                                # if a {
24     movsd       %xmm0, qword ptr [%rip+.quad_constant_0] #   42f
25     jmp         .block_1
26 .block_0:                                                # } else {
27     movsd       %xmm0, qword ptr [%rip+.quad_constant_1] #   3.14
28 .block_1:                                                # }
29     movsd       qword ptr [%rbp-16], %xmm0              # let b = %xmm0
30     movsd       %xmm0, qword ptr [%rbp-16]
31     ucomisd     %xmm0, qword ptr [%rip+.quad_constant_0]
32     jp          .block_2
33     jne         .block_2                                # if b == 42f {
34     mov         %rax, 11                                #   11
35     jmp         .block_3
36 .block_2:                                                # } else {
37     mov         %rax, 3                                 #   3
38 .block_3:                                                # }
39     mov         qword ptr [%rbp-24], %rax               # let c = %rax
40 .block_4:                                                # loop {
41     jmp         .block_5                                #   break;
42     jmp         .block_4                                #   continue;
43     mov         %rax, 3
44     cqo
45     idiv        qword ptr [%rbp-24]                     #   3 / c;
46     jmp         .block_4                                #   continue;
47 .block_5:                                                # }
48 main..bar.return:
49     leave
50     ret
51
52 .section .rodata
53 .quad_constant_0:
54     .quad       0x4045000000000000  # = 42.0
55 .quad_constant_1:
56     .quad       0x40091eb851eb851f  # = 3.14
```

Listing 5.39 – Trimmed compiler output for the rush function in Listing 5.38.

whole thing ends with the second symbol, 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 5.38, 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 in line 22 of Listing 5.39 with both the variable pointer and a constant '1' as operands. This results in the equality flag being set to '1' if a bitwise AND operation of these operands results in '0', that is, if the boolean variable is '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`' instruction in line 33 to jump if the floats are not equal. The cause for the '`jp`' instruction in between is explained later.

TODO: show rust snippets

The representation of the loop is much simpler. It must only define two symbols, one before and one after the loop contents, and unconditionally jump back to the former at the end of every iteration. This second point is the reason for why the comments in Listing 5.39 show two '`continue`' statements, even though the rush function only had 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, 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 the 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 simple reason that this subsection highlights two unexpectedly complex hurdles 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 97. It is used when compiling rush infix expressions between two integers with either the '`/`' or the '`%`' operator.

Listing 5.40 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[14] to 128 bits via the '`cqo`' 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 93. 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 in Listing 5.39.

The second unexpected hurdle were floating-point number comparisons. Floating-point numbers as defined by the IEEE standard have the special property of not forming a total order **TODO: cite**. That means, not any two arbitrary values are comparable. For instance, the float standard defines a 'NaN'[15] value, which cannot be compared to itself. 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 5.39 that was previously hinted at on page 106. 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 5.41. The shown code snippet is inside the '`condition`' method which is used during compilation of if-expressions and condi-

[14] **TODO: what is sign-extension**

[15] Short for "not a number".

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

```
56  (Some(Register::Int(left)), Some(Value::Int(right)), InfixOp::Div | InfixOp::Rem)
    ↪  => {
        // ...
65      // make sure the rax and rdx registers are free
66      let spilled_rax = self.spill_int_if_used(IntRegister::Rax);
67      let spilled_rdx = self.spill_int_if_used(IntRegister::Rdx);
68
69      // move lhs result into rax
70      // analyzer guarantees `left` and `right` to be 8 bytes in size
71      self.function_body
72          .push(Instruction::Mov(IntRegister::Rax.into(), left.into()));
73
74      // sign-extend lhs to 128 bits (required for IDIV)
75      self.function_body.push(Instruction::Cqo);
76
77      // get source operand
78      let source = match right {/* ... */};
91
92      // divide
93      self.function_body.push(Instruction::Idiv(source));
94
95      // move result into result register
96      self.function_body.push(Instruction::Mov(
97          left.into(),
98          // use either `%rax` or `%rdx` register, depending on operator
99          IntValue::Register(match op {
100             InfixOp::Div => IntRegister::Rax,
101             InfixOp::Rem => IntRegister::Rdx,
102             _ => unreachable!("this arm only matches with `/` or `%`"),
103         }),
104     ));
        // ...
110     // reload spilled registers
111     self.reload_int_if_used(spilled_rax);
112     self.reload_int_if_used(spilled_rdx);
113
114     Some(Value::Int(IntValue::Register(left)))
115 }
```

Listing 5.40 – Compilation of integer division in x64.

tional loops. After pushing the '`ucomisd`' instruction, two additional checks are performed. Firstly, if the condition is '`==`', then add a conditional jump to the else-block for when the result is unordered. Secondly, if the condition is '`!=`', then add a conditional jump to the else-block for when the result is *not* unordered. An expression like '`NaN != NaN`' is therefore considered to be true.

5.4. Conclusion

5.4.1. RISC vs CISC Architectures

TODO: RISC vs. CISC [Dan05b, p. 5-6]

- RISC-V was less demanding and better

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_symbol),
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_symbol),
957         ));
958     }
959 }
```

Listing 5.41 – Compilation of float comparisons in x64.

6. Final Thoughts and Conclusions

In Chapter 2, we have learned how the syntactical and semantic analysis play a vital role before program execution can start. We have explained how the *parser* is used for analyzing the program's syntax in order to construct an AST. Furthermore, it has been explained how the *semantic analyzer*, validates the program's semantic rules.

Next, in Chapter 3, we have explained how a tree walking interpreter and a virtual machine can be leveraged to execute programs using an interpreter. Here, the tree walking interpreter has presented itself as the easier solution, both in planning and implementation. The virtual machine however has proven itself to be slightly more demanding as it requires a custom-made compiler to execute programs. However, we have concluded that a VM often executes a program faster.

Chapter 4 provided an overview of compilation to high-level targets. As examples for high-level targets, *WASM* and *LLVM* have been used. The former has proven itself to be **TODO: What about WASM**. During planning and implementation of rush, leveraging LLVM has proven itself to make implementation of a high-performance, multi-target compiler both feasible and easy. To summarize, writing a compiler targeting these high-level targets presented a demanding but accomplishable task. If we were to construct a compiler for a more complicated language, LLVM 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 implementation phase of this paper, 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, thus making the implementation process much harder. Furthermore, programming on the assembly level often creates bugs which are hard to eliminate for a programmer who is used to writing software in high-level languages. As a result of this, we have both spent many hours trying to locate subtle bugs in our compiler which were rooted on the assembly level. Moreover, creating a low-level compiler which emits efficient code has proven to be very difficult. Therefore, our low-level compilers only focus on the minimal principles without regarding optimization techniques. Like initially expected, implementation of the x86 compiler has proven to be more demanding than the implementation of the RISC-V compiler. Among other factors, this is because RISC-V is a modern architecture which was designed with the compilers targeting them in mind. On the contrary, x86 is a very old architecture which has evolved over time but was created when assembly programs were still written by hand.

To summarize, this paper has presented two vastly different means of program execution, the former being an interpreter and the latter being a compiler. In order to demonstrate the differences using practical examples, we have implemented our own programming language called *rush*. In summary, we have implemented two rush compilers, two interpreters, four compilers, and one transpiler. Additionally, we have implemented additional tooling like a language server, web playground, and command line interface combining all components. Real programming languages often implement a lot more tooling like 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 (`https://github.com/rush-rs`).

Acknowledgements

We would like to express our sincere gratitude towards our supervisor – *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 finals. Additionally, we would like to thank our classmate – *Fatima* for designing the initial version of 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: 9781450373494. 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: 0-201-39 82 8-1.

[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.

[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: 9781617294556.

[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.

[22] *RISC-V ABIs Specification*. Nov. 2022. URL: https://github.com/riscv-non-isa/riscv-elf-psabi-doc.

[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     = '=' | '+=' | '-=' | '*=' | '/=' | '%='
                    | '**=' | '<<=' | '>>=' | '|=' | '&=' | '^=' ;
```