The Conversion of Source Code to Machine Code

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

Silas Groh, Mik Müller

February 5, 2023

Sun Feb 5 18:15:29 UTC 2023

CFG Wuppertal

Abstract

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

This paper aims to inform the reader about different means of program execution, focussing on compiler construction. However, we will only focus on the basics since implementing a programming language is often a demanding task. In order to include practical examples, we will explain concepts on the basis of our own programming language called *rush*. During the implementation of rush, the focus for this paper has shifted slightly. As the title suggests, we originally planned on only implementing a 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, 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*. The latter also serves as the target architecture for one of the five compilers. Chapter 4 illustrates how compilation to high-level[1] targets works. As examples for high-level targets, we will present the compiler targeting the virtual machine, a compiler targeting *WebAssembly*, and a compiler which uses the popular *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
[2]low-level targets: specific to one target architecture and operating system

Contents

1. Introduction

Computer programs are often written in formal, purposefully designed languages. These languages introduce many constraints in order to allow programmers to implement algorithms in a structured and precise manner. Since programming languages should be easy to write for a human whilst 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 come with many 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 translation process is referred to as *compilation*, and is usually performed by a program called a *compiler*. However, the output instruction sequence must represent the identical algorithm specified in the source code. 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 which interprets the source code directly. This executor is referred to as an *interpreter*. 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 completely dismissed. However, implementing an interpreter is only rational 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 host 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 host language. Compilers therefore played an essential role in the early days of computing since 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 program 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].

TODO: Advantages of high-level langs /p. 9]Dandamuri2017

1.1. Stages of Compilation

In order to minimize intricacy and to maximize modularity, compilation often involves several individual steps. Here, the output of step s serves as the input for step $s + 1$. However, partitioning the compiler into as many steps as possible 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 if often 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.

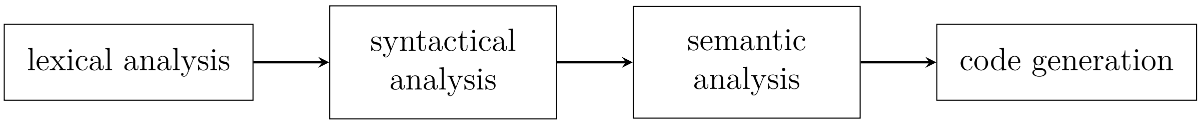


Figure 1.1 – Different Steps of Compilation

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 data structure which contains additional type annotations and early optimizations.
4. Code generation traverses the data structure 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.

[Wir05, pp. 6–7]

Many modern compilers tend to combine steps 1 and 2 into a single step. In this approach, the parser holds a lexer and instructs 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. Figure 1.2 shows an altered chart considering that change. **TODO: citation?**

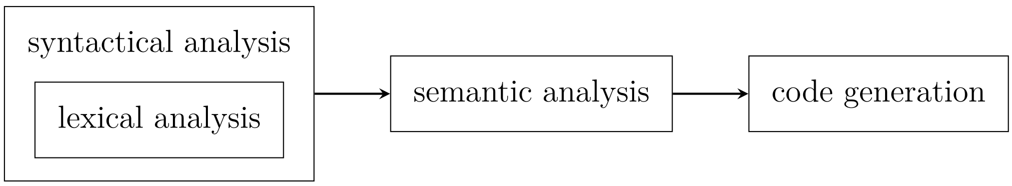


Figure 1.2 – Different Steps of Compilation (altered)

1.2. Characteristics of 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*, *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`, it denotes the position 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 $n \in \mathbb{N}$ must be valid in order for this function to return the correct result[2]. 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 program execution will start there. 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 must therefore also be an expression since it represents the last entity in the block, and it is not followed by a semicolon. In this example, 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 $n - 1$. Therefore, 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 redundant `return` statements. In line 2, the `exit` function is called. However, this function is not defined anywhere. Nevertheless, the code still executes without any errors. This is due to the fact that the `exit` function is a *builtin* function as it is used to exit a program using the specified exit code.

In the git commit `8a45d80`, the entire rush project includes 17168 lines of source code[3]. 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 IDE support. 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.

[1] Capitalization of the name is intentionally omitted
[2] Assuming the function should comply with the original Fibonacci definition
[3] Blank lines and comments are not counted

2. Analyzing the Source

2.1. Lexical and Syntactical Analysis

As previously mentioned, the first step during compilation or program execution is 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 most often are 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. **TODO: citation** 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. This paper does not fully explain these notations, however 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.

[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.

2.1.2. Grouping of 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 has is the union of 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

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

Listing 2.2 – The rush '`Lexer`' Struct Definition

The specifics of implementing a lexer are not explored in this paper, however a basic overview is still provided. The base principal of a lexer is to iterate over the characters of the input to produce tokens. Depending on the target language it might however 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 called something like `next_token` reading and returning, as the name suggests, 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 in requested. The `Pow` token, composed of two '`*`' symbols, 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 in 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 provide helpful diagnostics with locations to the user. 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 then read this field once at the start and once after having read the token and save the two values in the token's span.

```
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

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.

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, whilst the latter only stores the abstract program structure with just the relevant information for basic analysis and execution. Therefore, a CST is usually used for development tools like formatters and intricate linters and analyzers where it is important to preserve stylistic choices made by the programmer or to know the exact location of every token. 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 4. In the case of rush an AST with limited location information is used, because rush's semantic analyzer is still basic enough to work with that, and, as discussed, execution and compilation requires no CST.

Not every parser is the same and there are a few different strategies for implementing one. These strategies are categorized into *top-down parsers* and *bottom-up parsers*. The main difference between them being 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 processed before its children nodes. 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.

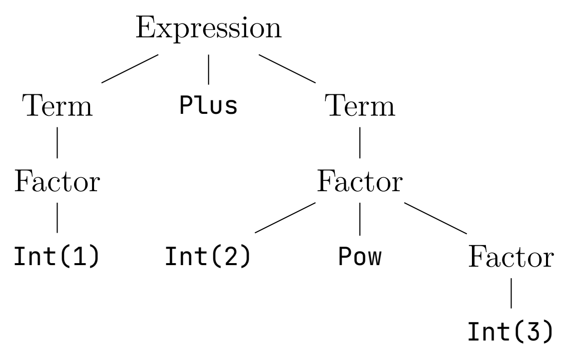


Figure 2.1 – Abstract Syntax Tree for '`1+2**3`'

Top-down and bottom-up parsers are further categorized into many more subcategories. The two we will focus on here are $LL(k)$ parsers and $LR(k)$ parsers. **TODO: 'we will' ok? TODO: citation for naming convention** 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** 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 for 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.

Alternatively to utilizing lookahead tokens, it is possible to create parsers with backtracking. Using that approach, a parser has to guess which syntax construct follows, if it can’t concretely know based on the first token. When it then later 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 of course comes with overhead and usually 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 partially explained later on page 11. One thing to note however, is that LR parsers are generally very complicated to implement manually. LL parsers on the contrary 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 a/an LL parser. An example for that is given in Listing 2.4.

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

Listing 2.4 – Example Language a Traditional LL(1) Parser Cannot Parse

Most LL parsers are *recursive-descent* parsers, including the rush parser. Implementation of such a recursive-descent parser is rather uncomplicated. Assuming the grammar respects the mentioned limitation, every context-free grammar rule is mapped to one method on a `Parser` struct. In the example grammar from Listing 2.1 on page 4 again, 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. In Rust the mapping from EBNF grammar notation to type definitions is very simple as displayed in Table 2.2. **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 previously discussed, a traditional LL parser cannot parse the language from Listing 2.4. However, when comparing it 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 `Terms` and having additive expressions be composed of `Terms`. 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. Consider the input ‘`1*2*3`’ it 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 method called *Pratt-Parsing* 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.

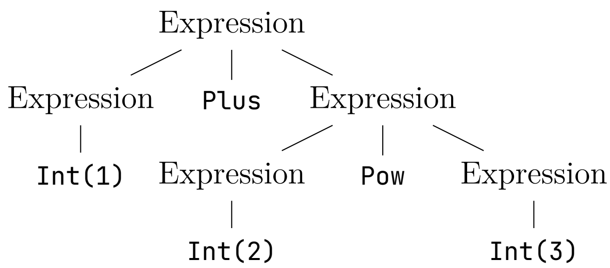


Figure 2.2 – Abstract Syntax Tree for ‘`1+2**3`’ Using Pratt-Parsing

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 are 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`[4] variable. Afterwards it checks whether the left precedence of the now current token is bigger than the `prec` argument.

[3] ‘`**`’ is the power operator here, so the input would be written as 1^{2^3} using mathematical notation
[4] 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

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 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 right **TODO: unclear it's the side 'right', not 'correct'** precedence 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.

This might be made clearer by Figure 2.3. It shows the tokens with their respective left and right precedence for the input '`(1+2*3)/4**5`', which represents the mathematical expression $\frac{1+2\cdot 3}{4^5}$. The for this example irrelevant precedences have been grayed out. Starting 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

from the left, the parser first encounters a/an '`LParen`' token and therefore decides to parse a '`grouped_expr`'. Inside that method, '`expression`' is again called, again with '`prec`' being 0. 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 '`prec`'s value of 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, again the current token's left precedence is higher than '`prec`', the precedences being 21 for the '`Star`' token and 20 for '`prec`'. Therefore, '`infix_expression`' is called once again, which call '`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 again fails 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, '`RParen`'s left precedence of 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 and to improve intuition,

(	1	+	2	*	3	)	/	4	**	5
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
LParen	Int(1)	Plus	Int(2)	Star	Int(3)	RParen	Slash	Int(4)	Pow	Int(5)
28 29	0 0	19 20	0 0	21 22	0 0	0 0	21 22	0 0	26 25	0 0

Figure 2.3 – Token Precedences for Input '`(1+2*3)/4**5`'

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 it being encoded in the Grammar, and with that, the node types.

Parser Generators

For most intents and purposes it is generally not recommended and necessary to implement parsers, and with that, lexers, manually. **TODO: citation** 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 traditional 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: 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 because the grammar cannot specify how programs should behave. Therefore, a programming language's behavior is often defined by a so-called *semantic specification*. This specification often describes how a program should behave during runtime and what semantic rules discriminate a valid program from an invalid one. Common rules include *type checking*, *context of statements*, or *integer overflow behavior*. Another example of a semantic rule is that a variable has to be declared before it is used. Defining the semantic rules of a programming language is often a demanding task since not all requirements are clear from the beginning. Because the semantic rules of a programming language can not be defined formally, a language designer often writes their specification in a natural language, meaning Chomsky type 0. However, due to the specification being written in a natural language, the specification can sometimes be ambiguous. Therefore, a well-written semantic specification should avoid ambiguity as much as possible. Furthermore, this specification is often written in English due to it being a well-adopted language across several academic fields like computer science. 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*[5] validates types and variable references whilst performing type annotations. The last aspect is of particular importance since all compiler backends rely on type information at compile time. In order to obtain type information, the abstract syntax tree of the source program must be traversed, performing numerous other checks during the process. The analyzer behaves exactly like this. In order to preserve a clear boundary between the individual compilation steps, the parser only validates the program's syntax without performing further validation. Therefore, the analyzer traverses the abstract syntax tree previously generated by the parser.

In order to highlight why type information is often required at compile time, we will consider Listing 2.9. The code in this listing displays a basic rush program calculating the sum of two integers and uses the result as its exit code. In this example, the exit code of the program will be 5.

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

Listing 2.9 – A rush Program Which Adds Two Integers

In this example, 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 listed constraints. Now, the analyzer traverses the function body of the `main` function. First, the analyzer examines the statements in the lines two and three. 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 the concrete values, the analyzer only considers the types of expressions. Therefore, in this example, the analyzer remembers that the variables `two` and `three` store integer values. This information will become much more useful when we consider line 4. 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. In this case, calculating the sum of two integers is legal 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. Furthermore, the analyzer validates all function calls and declarations, not just the ones of builtin functions. Since the result of the addition is also an integer, the analyzer accepts this program since both its syntax and semantics are valid.

[5] Later referred to as “analyzer”

As indicated previously, most compilers require type information whilst 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 significant 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. Since code duplication is considered inelegant, outsourcing this algorithm into a separate component is likely the best option. As a result of this, the analyzer implements such a tree-traversal algorithm for determining the types of subtrees. Because of the previously mentioned reasons, rush's semantic analyzer also annotates the abstract syntax tree with type information so that it can be utilized by later steps of compilation.

In order to obtain a deeper understanding of how the analyzer works, we will now consider parts of its implementation and how they behave when analyzing the example from above. However, before we can examine how the analyzer's implementation behaves, we should first highlight which attributes play a vital role in the analyzer.

```
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 if the function exists and can compare if the arguments match the declared parameters. The next field, `diagnostics` contains a list of diagnostics. A diagnostic is a struct which represents a message, is intended to be displayed to the user of the compiler. 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 using braces (`{}`) create 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 later, if they are mutated, and the location of where they were declared. By saving this much information about each declared variable, the analyzer can produce very helpful and accurate error messages or warnings. Such a message containing various diagnostics is displayed in Listing 2.11, it occurs 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. 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 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. Due to this design, nested loops do not cause issues while the validity of the above statements can be guaranteed.

Now that important attributes have been highlighted, we can now consider the example from Listing2.9. First, the analyzer traverses and analyzes all functions and their bodies. For every rush function, the analyzer invokes an internal method responsible for validating functions. Among other tasks, this method sets the `curr_func_name` field to the name of the current function and inserts a new entry into the `functions` hashmap, associating the function's name with its signature. 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. Naturally, this lookup is performed after all functions have been analyzed since the `main` function can then exist in the hashmap. The code in listing 2.12 shows how validating the `main` function's signature works.

```rust
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(
                    // ...
418             )
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(
                        // ...
432                 )
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 Validating the Signature of the '`main`' Function

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 function declared.

In line 401, the method updates the current function name. The if-clause in line 403 checks if the method is currently analyzing the main-function. In this case, the code inside the if-clause is executed in order to perform special checks on the main-function. 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, we have not yet explained how error handling in the analyzer works. When examining the signature of this method, it becomes obvious that it cannot return 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`' struct into the `diagnostics` vector. Therefore, the '`diagnostics`' vector will contain any generated errors after analysis has been completed.

In rush, the main-function cannot return a value at runtime. Therefore, the return-type of the function always has to be the unit-type. Thus, the analyzer validates this constraint in the lines 422 and 423. The first if-clause checks if 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. In this case, the analyzer checks if the manually specified type differs from the required unit-type. If this is the case, 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 body (`node.block`) of the method as its first argument. The second boolean argument specifies that the method should not push another scope onto the stack since this is 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, metadata like the function's analyzed parameters, its return-type, its name, or its body are specified. Unknown variables like '`params`' have been defined in the parts of the code which are currently hidden.

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 of the analyzer is invoked. The code in Listing 2.13 shows the '`let_stmt`' method of the analyzer.

```
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 {
            // ...
656         },
657     ) {
        // ...
680     }
        // ...
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. After the subtree of the expression has been traversed and analyzed, its data type is now known. The analyzer now inserts a new entry for the variable's name (e.g. '`two`') into its current scope. This insertion is performed in line 644. The contents of the pushed variable struct are hidden but include its type or its span for instance. Since the span includes the location of where the variable was defined, it can later be used in error messages like the one 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, it can be reassigned to. Because the variable in our example program was not declared as mutable, the error seen in Listing 2.11 was generated as a result. However, what strikes the eye is 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 called *variable shadowing*. Here, the analyzer should display some additional hints or warnings, depending on whether the shadowed variable has been referenced before it was shadowed. If this was not the case, the analyzer will generate a warning message informing the user about an unused and therefore redundant variable. Among the previously mentioned data, the insertion includes the variable's data type which was obtained by prior analysis of the expression.

It is now apparent that the analyzer uses type information of expressions on many occasions just like this one. However, we have not yet explained how type determination and annotation works in the analyzer. In order to get an understanding of how the analyzer determines types of variables, we must consider how expressions are traversed. The code in Listing 2.14 is part of the method responsible for analyzing expressions.

```
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

The '`node`' parameter specifies the expression node generated by the parser, it does not contain any type information since it is yet to be analyzed. It is also apparent that the method returns a value of the type '`AnalyzedExpression`', which represents an analyzed and annotated expression. Therefore, 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 for instance, this method calls the appropriate method responsible for analyzing this type of expression. This way, implementation of this method stays relatively simple and the maintainability of the codebase increases.

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. Since grouped expressions contain another inner expression, a grouped expression inside another 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.

```
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 we have now explained how tree traversal and analysis works 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. This case is displayed 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 directly. For instance, during analysis of block expressions, the responsible function checks if the block contains a trailing expression. If this is the case, the result type of the block expression is identical to the one of its trailing expression. For instance, the result-type saved in a field on an individual AST node is accessed in line 142. 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 return a potential error which would terminate the tree traversal. To mitigate this issue, the `Unknown` type was implemented. 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 could have only been caused by a previous error. Therefore, errors do not cascade, meaning that an undeclared variable will not cause another type error.

```
let args = node
    .args
    .into_iter()
    .zip(func_params.inner)
    .map(|(arg, param)| {
        self.arg(arg, param.type_.inner, node.span, &mut result_type)
    })
    .collect();
```

Listing 2.16 – Validation of Call-Arguments in the Analyzer

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.

The code in Listing 2.16 contains the part of this method which is responsible for validating that the arguments are compatible with the function's parameters. For this, the code in the snipped iterates over the provided arguments. In each iteration, the '`self.arg`' method of the analyzer is invoked. This method validates that the provided argument matches the provided 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 the invalid variable is unknown, the placeholder `Unknown` type would be used in order 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 an example for tree nodes which 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.

```
1814 let arg = self.expression(arg);
1815
1816 match (arg.result_type(), param_type) {
1817     (Type::Unknown, _) | (_, Type::Unknown) => {}
1818     (Type::Never, _) => {
1819         self.warn_unreachable(call_span, arg_span, true);
1820         *result_type = Type::Never;
1821     }
1822     (arg_type, param_type) if arg_type != param_type => self.error(
     // ...
1827     ),
1828     _ => {}
1829 }
```

Listing 2.17 – Validation of Argument Type Compatibility in the Analyzer

Listing 2.17 shows a part of the `arg` function which is responsible for validating that a function call argument is compatible with the declared parameter. In the above example, this means that the `exit` function is to be called with exactly one integer argument. 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 1817, 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 1818 presents the scenario in which the provided argument has the '`Never`' type. In this case, the analyzer should only add a warning that the call-expression is unreachable. Furthermore, the result type of the entire call-expression will also be updated to reflect the never-type. However, this scenario does not cause an error to be generated. Line 1822 displays the final scenario in which the type of the argument differs from the expected type of the parameter. In this 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*. Constant folding is often used in order to avoid the emission of otherwise redundant arithmetic instructions. As a result of this, the compiled program will run faster since less computation is being performed when the program is executed. In order to make such optimization possible, each expression node in the analyzed AST has a method named `constant` [Wir05, p. 54].

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

Listing 2.18 – Method for Determining if an Expression is Constant

This method is responsible for determining whether an expression is constant. The method returns `true` if its expression is a constant integer, float, boolean or char. Other types of expression, 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.

Among other optimizations implemented in the analyzer, loop transformation can also have a positive effect on the program's performance during runtime. The top listing displays part of a rush program which uses a `while` loop even though a `loop` would be faster. The other listing displays the same algorithm implemented using the faster `loop`.

```
while true {
    a += 1
}
```

```
loop {
    a += 1
}
```

The `loop` implementation is more efficient since the condition check is omitted before each iteration. Because the `while` loop checks that its head condition is true before it starts the next iteration, the `while` loop will run slower than the `loop` in this example. However, this is only the case because the condition of the loop is a constant `true`. Therefore, using a condition which is always true is redundant and should therefore be omitted. 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.

```
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 }
```

Listing 2.19 – Loop Transformation in the Analyzer

Another scenario in which a `while` loop can be restructured occurs if the condition always evaluates to `false`. This example is displayed in the listing below.

```
while false {
    // this loop will never iterate
}
```

Since the loop in the above listing never iterates, it is completely redundant and can therefore be omitted entirely. This scenario is detected in line 853 of Listing 2.19. 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. Therefore, implementing such trivial optimizations can significantly contribute to a more efficient output program. However, compiler writers often implement significantly more of those early optimizations than the ones presented in the two examples from above.

TODO: Include tree conversion figure which uses constant folding + has annotated types

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. **TODO: citation** 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, which is why it is the first one explained here. **TODO: mention slowness here?** It requires no further intermediate steps after the analysis but simply accepts the AST as input. Just like the parser and analyzer, it walks the tree, hence the name, and therefore again requires one method per node type.

The struct definition is also rather small as seen in Listing 3.1. 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 mentioned?** Why the ‘`Value`’ is wrapped inside a/an ‘`Rc<RefCell<Value>>`’ 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

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

that the interpreter simply makes use of Rust's types and does not need to worry about the hidden implementation details.

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

'`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 remaining two variants are for one runtime errors, produced by things like division by zero, and '`Exit(i64)`', created by rush's builtin '`exit(int)`' function. 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

[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.

before its definition in the file, 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 is the code inside the main function called via the '`call_func`' method. After it returns, the entire interpreter either returns a produced runtime error, an exit code caused by a rush call to '`exit`', or the success exit code '0' otherwise.

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

```
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 functions was called. As the only built-in function in rush is '`exit`', a simple equality check is enough here. In case the called function is in fact 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 received from the '`functions`' map. A new variable scope for the function's body is then initialized and filled with the arguments passed to the function. 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. Example

```
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 found in Listing 3.5 is presented in the following.

First, the main function gets called by the interpreter's entry point as discussed before. In there, '`visit_block`' is called with the main function's block. The '`visit_block`' method then iterates over its contained statements 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, 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.

```
run(/* ... */)
call_func("main", vec![])
visit_block(/* ... */)
visit_statement(/* ... */)
visit_expression(/* ... */)
visit_call_expr(/* ... */)
visit_expression(/* ... */)
visit_call_expr(/* ... */)
call_func("plus_two", vec![40])
visit_block(/* ... */)
visit_statement(/* ... */)
```

Figure 3.1 – Call Stack at the Point of Processing the Return Statement

Following that, an interrupt of kind 'return' is constructed, holding the sum. By making use of Rust's '`?`'[3] operator, 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. The '`exit`' function now finally has a value for its argument and can be called. 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 following language implementations, which all have a manually managed memory layout. It also depends a lot on the implementing 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 and provides shared access to it, without any **one** owner. The held 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

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

them, we can make use of another type the Rust standard library conveniently provides. A '`RefCell<_>`' implements so-called *interior mutability*[4] by enforcing Rust's borrowing rules at runtime. By now wrapping values inside both a reference counter and a `RefCell`, it is possible to support mutable variables and pointers.

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

3.2. Using a Virtual Machine

TODO: For more general info: [Wat17, p. 209]

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 fundamentally differs from a tree-walking interpreter. For rush, we have implemented a virtual machine backend in order to compare it to the previously explained tree-walking interpreter.

3.2.1. Defining a Virtual Machine

Often, one might encounter the term *virtual machine* when talking about emulating an existing type of computer using a software system. This emulation often includes simulating additional devices like the computer's display or its disk. In this context however, a *virtual machine*[5] is a software entity which emulates how a computer interprets instructions. Just like a real computer, a virtual machine executes low-level instructions directly. However, since the VM is unable to traverse the AST, it depends on a compiler generating its input instructions.

Since a physical processor and a virtual machine share some fundamental traits, the architecture of a virtual machine is often a slight deviation from the *von Neumann architecture*. The von Neumann 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 *ALU*[6], a control unit, multiple registers, memory, and basic IO [Led20, p. 172]. The ALU is often designed so that it performs logical and mathematical operations as fast as possible. However, to keep its implementation simple, it lacks the ability to fetch and execute instructions from memory directly. Therefore, the processor contains a control unit which manages the *fetch-decode-execute* cycle. The fetch-decode-execute cycle is a simplification of the steps a processor performs in order to execute an instruction. The list below explains the individual steps of the fetch-decode-execute cycle.

TODO: Figure which shows the 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 if 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 optimized 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 is powered on until the point in time when it is powered 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

[5] May later be shortened to "VM"
[6] Short for "arithmetic logic unit"

until execution of $i - 1$ has completed [Led20, pp. 208-209].

For virtual machines, executing the input instructions in sequential order is often also the simplest solution. Often, 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, meaning that they are implemented using software, design constraints are usually kept to a minimum. Therefore, a virtual machine can also be implemented with the high-level constructs of the source language in mind. For instance, the VM might feature specialize 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. From the point where the architecture is clear, implementing the VM should normally be a straight-forward task.

3.2.2. Register-Based and Stack-Based Machines

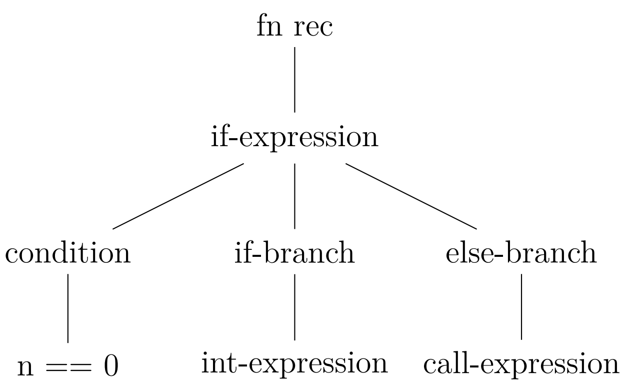
One of the main decisions to be made when designing a VM is how it implements temporary storage. Physical processors often use *registers* in order to make larger computations feasible. Registers are a limited set of very fast, low capacity storage units. On modern architectures, like *x86_64*, each 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 computers CPU. Therefore, programs often only utilize registers for storing temporary values, such as intermediate results of a large computation. The main alternative to using 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 WebAssembly in Chapter 4. For compiler writers, register allocation is often a demanding task. This problem is described in more detail 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 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 reason for this is that use of the stack usually requires a lot of push or pop operations which could have otherwise been omitted.

3.2.3. Comparing the VM to the Tree-Walking Interpreter

One significant benefit of virtual machines is that they execute programs much faster compared to most tree-walking interpreters. A reason for this speedup is that tree-traversal involves a lot of overhead which is omitted when instructions are interpreted directly. The code in Listing 3.6 displays a recursive function implemented in rush.

```
fn rec(n: int) -> int {
    if n == 0 {
        0
    } else {
        rec(n - 1)
    }
}
```

Listing 3.6 – A Recursive rush Program




```
0: (prelude)
        setmp 0
        call 1
1: (main)
        setmp 0
        push 1000
        call 2
        exit
2: (rec)
        setmp 1
        svari *rel[0]
        push *rel[0]
        gvar
        push 0
        eq
        jmpfalse 9
        push 0
        jmp 14
        push *rel[0]
        gvar
        push 1
        sub
        call 2
        setmp -1
        ret
```

Figure 3.2 – Abstract Syntax Tree and VM Instructions of a Recursive rush Program

Figure 3.2 displays a heavily simplified syntax tree and rush VM instructions representing the function displayed in Listing 3.6. The root node of the tree represents the `rec` function. Since the function only contains a single expression, the if-expression node is the only child of the root node. The if-expression contains a condition, an if-branch, and an else-branch. Since the function should not call itself again if `n` is equal to 0, the if-branch returns 0. In the else-branch however, 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. Since `rec` is a recursive function, the tree-walking interpreter would have to traverse it n times. 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 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. A compiler targeting the virtual machine's architecture first traverses the AST and outputs a sequence of instructions. The instructions on the right side of Figure 3.2 represent the program in Listing 3.6. 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[7]. Therefore, 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.

3.2.4. The rush Virtual Machine

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

```
16 pub struct Vm<const MEM_SIZE: usize> {
17     /// Working memory for temporary values
18     stack: Vec<Value>,
19     /// Linear memory for variables.
20     mem: [Option<Value>; MEM_SIZE],
21     /// The memory pointer points to the last free location in memory.
22     /// The value is always positive, however using `isize` is beneficial in order
   ↪ to avoid casts.
23     mem_ptr: isize,
24     /// Holds information about the current position (like ip / fp).
25     call_stack: Vec<CallFrame>,
26 }
```

Listing 3.7 – Struct Definition of the Vm

Listing 3.7 displays the struct definition of the rush VM. In line 18, a field called '`stack`' is declared. Like explained above, the rush VM uses a stack for storing temporary values. Even though the stack is implemented using a '`Vec`', it behaves identical to a LIFO stack data structure. In line 20, the '`mem`' field is declared. This field represents the linear memory capable of storing variables. However, each memory cell is implemented to hold an option of a value. Therefore, each memory cell may also hold a `None` value representing uninitialized memory. In line 23, the '`mem_ptr`' field is declared. It, too, serves an important role in managing the linear memory. The exact responsibilities of the so-called *memory pointer* will be explained shortly. Lastly, a field named '`call_stack`' is declared. This field also behaves like a stack and is responsible for managing function calls and returns. More information on this field will be provided in later parts of this section.

The instructions in Figure 3.2 can be interpreted by the rush VM. Here, the output program is structured as functions which each contain a list of their instructions. Since function and variable names are replaced by indices, strings can be entirely omitted in the output instructions. This often leads to a decrease in code size and an increase in runtime

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

speed. For better understanding, we have annotated the individual functions with their human-readable names.

The first block of instructions can be called the '*prelude*' since its only task is to call the '`main`' function and declare global variables. 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 which saves the runtime value of a variable in each index. Since an array is used, the memory of the VM is limited. However, a large memory size if often enough to run most of the possible programs. In the rush VM, each storage cell can be accessed using two addressing modes. When using the *absolute addressing* mode, the exact index of the memory cell is specified. For instance, if the value of variable '`d`' of Figure 3.3 was to be retrieved, the VM would need to access the storage cell with the index 4. However, the absolute position of a variable in memory can only be determined at runtime. In a recursive function, each recursion adds more variables to the scope, thus allocating more memory. Here, the exact number of recursions the function performs would have to be known at compile time. Of course, this presents an impossible task, thus making writing a compiler targeting the VM impossible. 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 variable '`d`' can be addressed by index 0 using the relative addressing mode. In Figure 3.3, the memory pointer (shortened to "mp") is set to 4. Considering the value of the memory pointer, the absolute address of any relative address can be calculated at runtime. Here, the absolute address a is the sum of the relative index i and the runtime memory pointer m. Therefore, the absolute address of any relative address can be calculated at runtime like this: $a = i + m$. By also implementing this relative addressing mode, compilers targeting the rush VM can generate code without knowing the runtime behavior of a program.

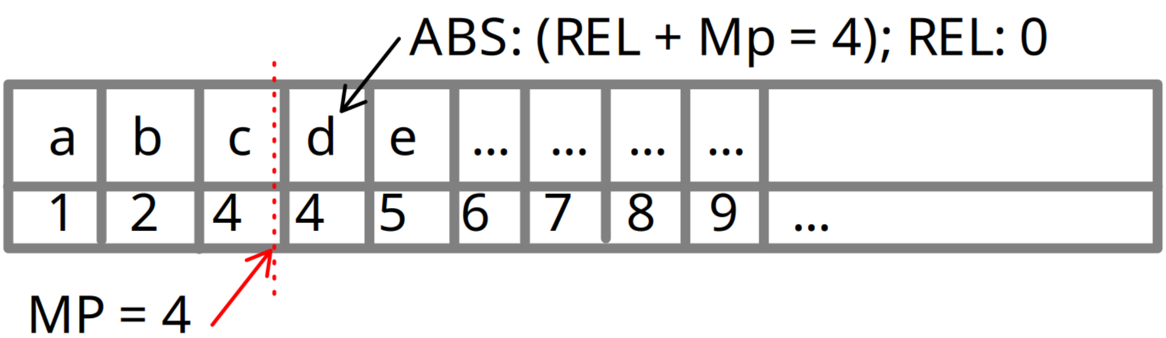


Figure 3.3 – DRAFT: Linear Memory of the rush VM

In order to get a deeper understanding of the addressing modes, a practical example can be considered. The code in Listing 3.8 displays a rush program in which a pointer to a variable is created. First, the integer variable `num` is created. In line 3, a pointer variable called `to_num` is created by *referencing* the `num` variable.

```rust
fn main() {
    let mut num = 42;
    let to_tum = &num;
}
```

Listing 3.8 – 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` 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 3.9 shows the VM instructions generated from the rush program in Listing 3.8.

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

Listing 3.9 – VM Instructions for the minimal Pointer Example

The first instruction `setmp` (*set memory pointer*) increases the memory pointer by two. This is because the `main` function contains two local variables whose space is to be allocated at the start of the function. For instance, one might encounter `mp` being incremented by 0 since the corresponding function contains no local variables. The next instruction `push` pushes the value 42 onto the stack. In line 4, the `svari` (*set variable immediate*) assigns the top value on stack to the specified relative address. Here, 42 is popped off the stack since it is used by the `svari` instruction. Next, the instruction stores the previously popped value at the relative address at the relative address 0 specified in the operand. 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` (*relative to address*) instruction is used to calculate the absolute memory address of the `num` variable. This instruction calculates the absolute address of its operand at runtime using the algorithm described above. Then, the instruction pushes the calculated address onto the stack so that it can be used by following instructions. Here, the relative address 0 is used since the `svari` instruction has previously saved 42 at this location. Therefore, the value of the variable `num` is saved at the relative address 0. In line 6, the `svari` instruction is used again. This time, it is used to save the value of the `to_num` variable. Since the absolute address of the referenced variable was previously calculated, it now exists on top of the stack. Now, the instruction saves the absolute address of `num` 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 above 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.

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

Figure 3.4 – Call Stack of the rush VM

3.2.5. How the Virtual Machine Executes A rush Program

By considering the minimal pointer example from above, we now have a rough idea how the VM might execute instructions. In order to get a better understanding of how the rush VM works exactly, we will explain how it executes the program in Listing 3.6. For this, we should consider the instructions in Figure 3.2 again. The first instruction of the prelude function is '`setmp`'. This instruction adjusts the memory pointer by the amount specified in the instruction's operand. In this case however, the memory pointer remains unmodified since the operand of the instruction is 0. Next, the '`call 1`' instruction calls the '`main`' function. In order to understand how function calls work in this VM, we must consider the call stack of the rush VM. Before the call-instruction, the caller pushes any call-arguments onto the stack so that they can be used as parameters by the callee. Figure 3.4 displays the state of the VMs call stack after the '`call 1`' instruction would be executed. During execution of a call-instruction, the VM pushes a new stack frame onto its call stack. Listing 3.10 shows how a call frame is implemented.

```
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`

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 holds the index of the current instruction. 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 '`fp`' field is declared in line 33. This field specifies the *function pointer* which holds the index of the current function.

After the function call, '`fp`' is set to 1 since the main function is called and instruction should start at the first instruction of the main function. Figure 3.4 shows how the call stack of the VM looks like after the '`call`' instruction has been interpreted. Function calls are managed in a stack in order to allow returning from functions. If the VM encounters a '`ret`'[8] instruction, it should leave the current function. However, it should also know where to resume its fetch-decode-execute cycle. For this, the VM just simply pops the top element from its call-stack. 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. 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

[8] Short for "return"

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. After the call instruction in line 7, the VM starts executing 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 instruction `svari` 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. It is then consumed by the `gvar` instruction in line 13. At this point the top element on the stack contains an address-value referring to the target of the `gvar` instruction. Therefore, the instruction first pops the top element from the stack. In this case, the value of the popped element is the relative address 0. Then, the instruction retrieves the value of the target variable and pushes it onto the stack.

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 comparison is pushed onto the stack as a boolean value. In this case, the instruction compares if the current value of `n` is equal to 0. In line 16, the `jmpfalse` instruction is executed. This instruction jumps to the specified instruction index if the value on top of the stack is `false`. In this case, if the value on the stack is false, the parameter `n` was not equal to 0. Now, the VM would jump to the instruction in line 19. Here, the value of the parameter `n` is pushed onto the stack using the previously explained `push` and `gvar` instructions. Now, the top item on the stack is the value of the parameter `n`. In line 21, the `push` instruction pushes a constant 1 onto the stack. Next, the `sub` 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` and pushes the result onto the stack. Next, the `rec` calls itself recursively using a 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 function decrements the memory pointer in order to deallocate used memory using the `setmp` instruction in line 24. 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 function. Now we have considered what happens if the value of `n` was not equal to 0.

However, if the result of the comparison in line 15 is true, meaning that `n` is equal to 0, the `jmpfalse` instruction in line 16 does nothing. In this case, the VM continues to the `push` instruction in line 17. Here, the constant value 0 is pushed onto the stack. Next, the VM interprets the `jmp` instruction in line 18. Unlike `jmpfalse`, this instruction performs its jump without any condition. In this case, the instruction jumps to the instruction at index 14 of the current function. The instruction at index 14 is `setmp` in line 24. Since functions also return values by placing them on top of the stack, the return-value would be 0 in this case. Since we have covered what the instructions in the lines 24 and 25 do, we can summarize that the function returns the value 0 in this case.

3.2.6. Fetch-Decode-Execute Cycle of the VM

Now that the semantic meaning of the instructions in Figure 3.2 has been explained, we will explain how the fetch-decode-execute cycle works in the VM. The code in Listing 3.11 displays the '`run`' method of the rush VM.

```
168 pub fn run(&mut self, program: Program) -> Result<i64> {
169     while self.call_frame().ip < program.0[self.call_frame().fp].len() {
170         let instruction = &program.0[self.call_frame().fp][self.call_frame().ip];
171
172         // if the current instruction exists the VM, terminate execution
173         if let Some(code) = self.run_instruction(instruction)? {
174             return Ok(code);
175         };
176     }
177
178     Ok(0)
179 }
```

Listing 3.11 – 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 main construct in the function is a 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 if it reaches the end of an instruction sequence. In line 179, the next instruction to be interpreted is saved as the variable '`instruction`'. This line represents the *fetch* step since the next instruction is fetched from memory and placed in a spot where it can be used by the following steps.

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* error. Furthermore, this method may return an optional integer representing the exit code of the program. If the method returns such a code, instruction execution comes to a halt instantly and the VM exists using the retrieved code. However, if the method returns none of these two possible types, the fetch-decode-execute cycle continues. What might seem odd is that in this code, one cannot observe the instruction pointer being incremented. Therefore, one might think 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 3.12.

```
181 fn run_instruction(&mut self, inst: &Instruction) -> Result<Option<i64>> {
182     match inst {
183         Instruction::Nop => {}
184         Instruction::Push(value) => self.push(*value)?,
            // ...
191         Instruction::Jmp(idx) => {
192             self.call_frame_mut().ip = *idx;
193             return Ok(None);
194         }
            // ...
261         Instruction::Exit => {
262             return Ok(Some(self.pop().unwrap_int()));
263         }
            // ...
272         Instruction::Add => {
273             let rhs = self.pop();
274             let lhs = self.pop();
275             self.push(lhs.add(rhs))?;
276         }
            // ...
357     }
358     self.call_frame_mut().ip += 1;
359     Ok(None)
360 }
```

Listing 3.12 – Parts of the `run_instruction` Method of the rush VM

The code in Listing 3.12 displays parts of the `run_instruction` method. This method mainly consists of a match-expression which determines which code to run depending on the current instruction. In this example, the implementations of several instructions are 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. In line 184, the code for the '`Nop`' instruction is displayed. Since this instruction should push the operand onto the stack at runtime, the helper method '`self.push`' is invoked. This method pushes the provided parameter onto the '`stack`' field of the VM. However, the function first validates that the stack will not exceed its maximum capacity. If the call to push would cause the capacity of the stack to be exceeded, the method returns a runtime error describing the issue. In line 261, the code for executing the '`Exit`' instruction can be seen. This instruction can be used to terminate the execution of the program prematurely. It is inserted by the compiler if it encounters a call to the '`exit`' function.

In line 191, the code responsible for executing the jumping instruction '`Jmp`' can be seen. This instruction only sets the instruction pointer to the target index specified in the instruction operand. Therefore, a jump involves very little overhead and is implemented using very little effort. For some special instructions, such as the jump-instructions, 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 prematurely.

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 later executed similarly. It is apparent that the execution of these instructions involves relatively little difficulty. After the instruction has been executed, the instruction pointer is finally 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 implementation of a VM is usually not that demanding. 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 the following chapter.

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 whilst using the benefits introduced by a VM [Lin+14, Chapter 1.2].

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.

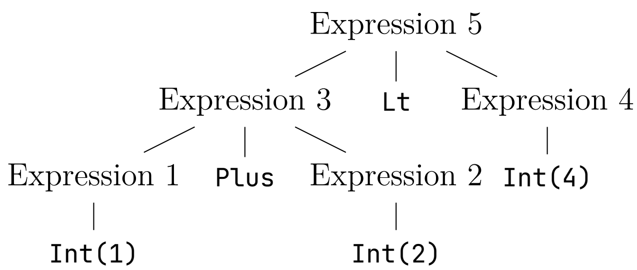


Figure 4.1 – Abstract Syntax Tree for ‘`1+2 < 4`’

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`’ whilst 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.

```
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

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 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.

```
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`’ instruction 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.

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

```
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.

```
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

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

```
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

1. what is WASM and why
2. modules
   - binary and text format (sections, leb128)
   - globals, functions, imports, exports, …
   - uncommon to target binary format
3. WASI
4. basic implementation
   - leb128
   - sections
   - `self.function_body`
5. example program

The first ‘external’ compilation target presented here is *WebAssembly*, or *WASM* for short. **TODO: research origins and goals of WASM and explain them here, CITATION** 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*.

4.2.1. WebAssembly Modules

Every valid WebAssembly file must contain exactly one module. **TODO: confirm, CITATION** The WebAssembly specification defines two different representations for these modules. First, there is a human-readable text representation, called *WAT*[1], closely resembling S-Expressions[2]. This is comparable to assembly languages for CPU architectures and is the typical target for compilers. Secondly, WebAssembly modules can also be represented using its binary format, which is optimized for size and comparable to the final binary files produced by assemblers. Most often these binary modules are constructed from a text module by using a tool such as *wat2wasm* from the *WebAssembly Binary Toolkit (WABT)*. However, the rush WebAssembly compiler instead opts to target the binary format directly, highlighting a few reasons for why most compilers should not do this. Listing 4.6 and Listing 4.7 on page 45 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 `https://webassembly.github.io/wabt/demo/wat2wasm/`.

Focusing on the text representation first, the shown module contains one function that takes two '`i32`'s as parameters and returns a single '`i32`'. **TODO: CITATION for statements below** An '`i32`' in WebAssembly represents an uninterpreted 32-Bit integer, that is, 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. For instance, the instruction '`i32.eq`', which checks for equality between two '`i32`' values, behaves the same no matter the integer's signedness. 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.

The mentioned function is exported by the module under the name 'addTwo' to make it accessible from outside. What exactly 'outside' is 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 ran, the stack now contains the values of the two function parameters. They are then added by '`i32.add`' which pops the top two elements off the stack and pushes the sum back on. The return value implicitly is always what remains on the stack at the end of a function body.

Now focusing on the hex dump of the same module in binary in Listing 4.7. A WebAssembly binary file always starts with the four bytes '`00 61 73 6d`' called the *WASM binary magic* and 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[3]. At the time of writing it is always '1'.

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

[1] WebAssembly Text

[2] **TODO: what are S-Expressions**

[3] Little-Endian starts with the least significant byte first, whereas Big-Endian starts with the most significant byte

```
(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 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.

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 ASCII. 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 'wasm2wat'.

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

One may already have 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. The encoding details are not explained here however, and our implementation for the rush compiler simply uses a pre-existing crate[4] called 'leb128'.

TODO: lacking CITATIONS for simple claims like "WebAssembly uses the LEB128 encoding..."

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, of course, for module imports and exports. That is why an additional specification called the *WebAssembly System Interface*, short *WASI*, was created for WebAssembly modules that wish to communicate with an operating system. Any runtime supporting WASI must provide a set of functions comparable to *system calls* on Linux or Windows. These can then be imported from a WebAssembly module to do things like writing to a console and exiting with a specific exit code. Both wasmtime and wasmer implement the WASI interface.

A WebAssembly module making use of WASI must export one function under the name '`_start`' that acts as the entry point. 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.

TODO: anything else to explain/mention here?

4.2.3. Implementation

[4] A crate is a library in Rust terms

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.

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 '`function_body`' contains the bytes with instructions for this function, and '`locals`' stores which locals the function has along with their types. In the binary format the locals are stored as a WebAssembly vector, that is, it starts with the number of locals, followed by each local. Since the compiler cannot know the count of local variables beforehand, 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 contents. The three following fields all map names to indices. One for local variable scopes, one for the global scope, and one for function names. Each index itself is stored as a vector of bytes, as it uses the aforementioned LEB128 encoding which can vary in length. Finally, one field for every supported section is defined. They are of 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 – '`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 concatenates all sections together to form the final binary. It also already imports all required functions from WASI and exports blank linear memory as required by WASI. The '`Self::section`' helper function is used to add the section identifier, byte length, and element count for each section.

Inside the '`self.program`' method, the global variables are defined and initialized, and all function signatures are added to the 'Type' section and 'Function' section and the '`self.functions`' map. Afterwards, it calls '`self.function_definition`' for every defined function. This has to happen in these two steps, because rush allows functions to be called before their definition in the file. For every function body, first all statements and the op-crates/rush-compiler-wasm/src/compiler.rs

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

Listing 4.10 – Entry Point of the WASM Compiler

tional trailing expression are compiled by their respective methods, and then the values of '`self.function_body`' and '`self.locals`' along with their combined length are appended to the code section.

All other nodes have a matching method defined again, as every time an AST is traversed. Each of those methods simply pushes instructions to '`self.function_body`'. Because WebAssembly is stack-based, none of them has 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 VM.

TODO: function calls?

4.2.4. Example

TODO: example?

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. Today, it can be recognized as one of the largest open source projects [CA14, preface]. Among many other projects, the Rust programming language depends on the LLVM compiler in order to generate its target-specific code [McN21, p. 373]. Furthermore, the *Clang* C / C++ compiler uses LLVM as its code generating backend [Hsu21, preface]. Therefore, production ready compilers for popular programming languages have been implemented using the LLVM framework. Besides open-source projects, many companies have also incorporated 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 runs faster at runtime and uses less memory. In order to use LLVM, the system provides and API which is usable by earlier steps of compilation. 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 API of LLVM in order to construct an intermediate representation of the program. This way, the framework will be able to understand the semantic meaning of the program. Next, LLVM compiles the input program to an output which is specific to an arbitrary target architecture. As of today, LLVM features many target architectures so that a compiler designer does not have to worry about portability of the output program [Hsu21, preface]. Listing 4.3 shows how LLVM integrates into the previously discussed steps of compilation. Therefore, the framework represents the *back end* component of a compiler.

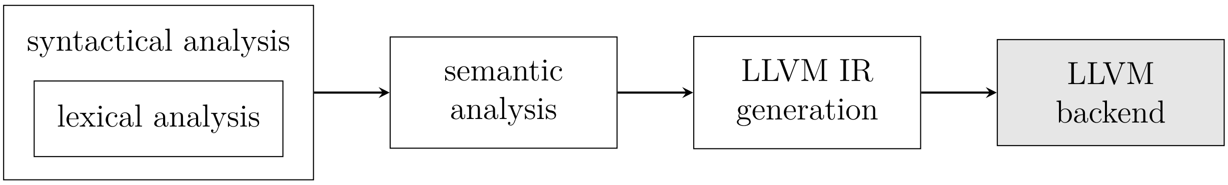


Figure 4.3 – Steps of Compilation When Using LLVM

TODO: continue here

The updated Figure 4.3 now includes two new steps: "LLVM IR generation" and "LLVM backend". The first step generates the target-independent input used by the second step to generate target-specific code. This step traverses the AST in order to generate a semantically equivalent program formulated using LLVM's intermediate representation. 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 intermediate representation (*IR*) represents the source program in a low-level manner. However, even though this IR is low-level, it is still target-independent. Furthermore, the IR also contains detailed type information which is usually uncommon for low-level representations of a program. Therefore, high-level type information is preserved while the benefits of a low-level representation are introduced. This allows LLVM to perform significant more aggressive optimizations compared to other compiler solutions or frameworks. Therefore, programs compiled using LLVM as the backend will often run significantly faster due to the many aggressive optimizations introduced by the framework.

LLVM provides many APIs for interacting with the IR in memory. This way, it can be created by a frontend without the need for a separate file containing the IR. If the 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 official APIs for the *C++* and *C* programming languages. However, there are many unofficial bindings, such as for Rust, Go, or Python. For instance, a compiler frontend written in Rust can leverage LLVM, although the system is written in C++. 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:

- 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 a 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 basic block contains a sequence of *instructions*. 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.

[Hsu21, p. 211-213].

The IR provides a low-level enough representation in order to allow optimizations in the early stages of compilation. However, due to the high-level type information contained in the IR, LLVM is able to perform many aggressive optimizations on the IR during later stages of compilation. This way, the framework can communicate a lot of information to the linker which can then use this information for *link-time* optimizations. The virtual instruction set of LLVM is therefore 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 for most of the common types of processors. Although it is low-level, the IR avoids machine specific constraints like registers or calling conventions. Instead, the virtual architecture provides an infinite set of virtual registers which can hold the value of primitives like integers, booleans, floating-point numbers, and pointers. All registers in the IR use the *SSA*[5] form in order to allow more optimizations. In order to enforce the correctness of the type information included in the IR, the operands of each instruction all obey LLVM's type rules. Therefore, LLVM only processes a program which contains valid type information [Lat02, p. 14-17].

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 4.11 displays the identical rush program formulated in LLVM IR. The IR was generated by the LLVM targeting rush compiler[6]. This compiler is presented in a later chapter since right now, only its output is of relevance. The IR displayed in this listing shows a module named '`main`'.

[5]Short for "static single assignment", widely used in optimizing compilers

[6]Generated in Git commit `8a45d80`, automatically built with this document

```
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 4.11 – LLVM IR Representation of the Program in Listing 1.1

In the lines 5 and 23, two functions are defined using the ‘`declare`’ keyword. It is apparent that the functions in the LLVM module represent the functions from the source rush program. The function’s name in the IR matches the name in the source file as it increases readability of the 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`’. In rush, the ‘`int`’ type represents a 64-bit signed number. Therefore, the `i64` LLVM type represents the rush `int` type. 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 uses the ‘`entry`’ block solely for variable declarations. In line 7, the ‘`icmp slt`’[7] 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 will make reading of the generated IR easier. In line 8, the block is terminated using the ‘`br`’[8] instruction. The instruction will only jump 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 an alternative jump to perform if the condition is false at runtime. Due to constraints introduced by its internal optimizations, LLVM only allows jumps to target blocks contained in the same function.

[7] Short for “integer compare (signed less than)”
[8] Short for “branch”

Therefore, two labels of blocks in the current function are used as the operands of this instruction. As the names '`merge`' and '`else`' suggest, this branch-instruction presents the essential part of the if-expression in the source program. If the condition was true at runtime, the instruction would jump to the '`merge`' block in line 10. What might seem odd is that there is no '`if`' block. In fact, the rush compiler has even compiled this block into LLVM IR. However, since that block only jumped 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 *phi-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 if 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 phi-nodes associate a value to an origin branch. In this example, the phi-node yields the value of the parameter '`%0`' (`n`) if control came from the '`entry`' block. In the source program, '`n`' should be returned without modification if it is less than 2. Therefore, the runtime result of the phi-node is '`%0`' if it is less than 2 at runtime. Otherwise, if control came from the '`else`' block, the phi-node's result is taken from 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`'. However, an addition instruction using a negative operand is used since LLVM's optimization decided that this instruction is likely beneficial. This is done in order to create the argument value for the first recursive call to '`fib`'. Next, the '`call`' instruction is used in order to perform the recursive call. Here, the '`\%i_sum`' register is used as an argument to the call-instruction. 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)`'. However, in that case, 1 is subtracted from the parameter and saved in '`\%i_sum1`'.

Next, the '`add`' instruction in line 19 is used in order to calculate the sum of the return values of the recursive calls This sum is then saved in the virtual register '`\%i_sum3`'. Therefore, this register is used in the phi-node in line 11 so that the result of the recursive calls is used as the result of the if-expression. Next, the '`br`' instruction jumps to the '`merge`' block. However, this jump happens unconditionally since the instruction does not consider a condition and only has one target label. 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, we will omit detailed explanation of its contents. However, in line 27, the '`unreachable`' instruction is used in order to state that it is never executed. 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 does not know that the '`exit`' function terminates program execution, an '`unreachable`' instruction is inserted to communicate a block termination to LLVM.

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 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].

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 4.12 displays the top part of the '`Compiler`' struct definition.

```
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 4.12 – Parts of the Struct Definition of the rush LLVM `Compiler`

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.

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 4.13 displays parts of the method responsible for translating the '`main`' function.

```
fn main() {
    foo(2)
}

fn foo(n: int) {
    let mut m = 3;
    exit(n + m)
}
```

```
define internal i1 @foo(i64 %0) {
entry:
  %i_sum = add i64 %0, 3
  call void @exit(i64 %i_sum)
  unreachable
}

declare void @exit(i64)

define i32 @main() {
entry:
  %ret_foo = call i1 @foo(i64 2)
  ret i32 0
}
```

Figure 4.4 – Translation of a Simple rush Program to LLVM IR

```
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 4.13 – 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, whilst 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 4.14.

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

Listing 4.14 – Compilation of Call-Expressions Using LLVM

The code in Listing 4.14 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 4.15.

```rust
fn expression(&mut self, node: &'src AnalyzedExpression) -> BasicValueEnum<'ctx> {
    match node {
        AnalyzedExpression::Int(value) => self
            .context
            .i64_type()
            .const_int(*value as u64, true)
            .as_basic_value_enum(),
            // ...
        AnalyzedExpression::Infix(node) => self.infix_expr(node),
    }
}
```

Listing 4.15 – Compilation of Expressions Using LLVM

The code in Listing 4.15 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.

```
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 4.16 – Compilation of Integer Infix-Expressions Using LLVM

The code in Listing 4.16 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 4.17.

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

Listing 4.17 – Compilation of Let-Statements Using LLVM

The code in Listing 4.17 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 4.18 shows the '`alloc_ptr`' method of the compiler.

```
635 fn alloc_ptr(&mut self, name: &str, llvm_type: BasicTypeEnum<'ctx>) ->
    ↪ PointerValue<'ctx> {
636     // save current insertion point
637     let curr_block = self.curr_block();
638
639     // insert at `entry` block
640     self.builder.position_at_end(self.curr_fn().entry_block);
641
642     // allocate the pointer
643     let res = self.builder.build_alloca(llvm_type, name);
644
645     // jump back to previous insert position
646     self.builder.position_at_end(curr_block);
647
648     res
649 }
```

Listing 4.18 – 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, a '`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

TODO: MORE ON LINKERS: [PW17, p. 40]

After LLVM has compiler 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*[9] 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.

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

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

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.

[Lev00, pp. 1-15]

The shell command in Listing 4.19 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.

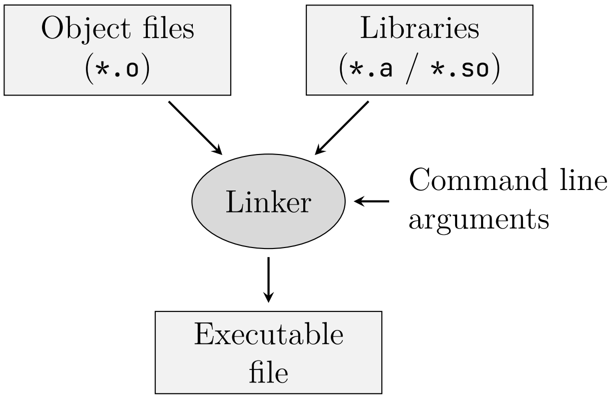


Figure 4.5 – How a Linker Works

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

Listing 4.19 – 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.

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[10]. Therefore, the program compiled using LLVM ran roughly 1.66 times faster.

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

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, p. 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]. In ELF programs, some of the important sections are:

- `.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 [Zhi17, p. 76].

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.

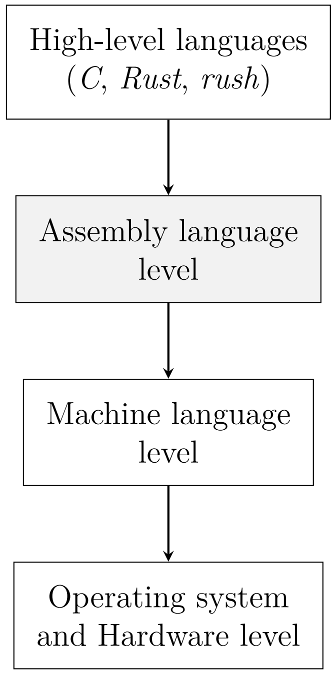


Figure 5.1 – Level of Abstraction provided by Assembly

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 whilst 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 whilst 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 [Dan05, 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].

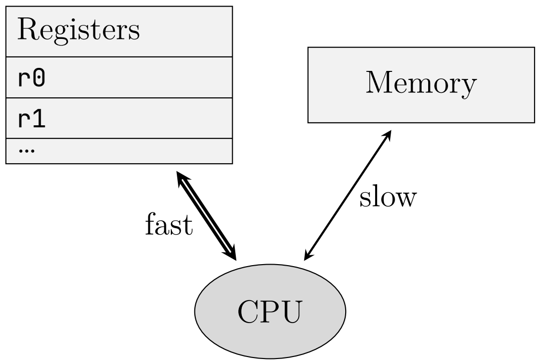


Figure 5.2 – Relationship Between Registers, Memory, and the CPU

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 [Dan05, 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 [Dan05, Chapter 2].

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

Listing 5.1 – Example Assembly Program for Explaining Register Allocation

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`'.

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

TODO: @RubixDev: write this subsection
TODO: Useful information: [PH17, p. 99]
TODO: Insert figure which uses '`drawstack.sty`'
TODO: More information: [Wal98, p. 85]

TODO: Stack frames: [Wal98, p. 93-95]

- Repetition: When is the stack used instead of registers?
- Insert very figure of the stack.
- Only mention that there are stack pointers and why they are needed; do not explain how they point / behave exactly
- Stack can be indexed

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. Figure 5.3 illustrates how such a call would be managed on this fictional architecture.

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

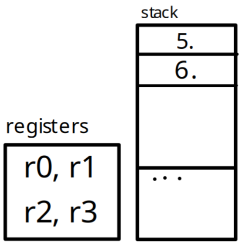


Figure 5.3 – DRAFT: Spilling Registers for Function Calls

Here, the registers '`r0`' — '`r3`' contain the first four argument values. The excess arguments which could not fit into registers are now placed on the stack. Even though argument 5 and 6 are placed on top of the stack in this example, the exact location of spilled call arguments differs for every architecture.

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.

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 5.2.

```
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 5.2 – 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

[2]Short for "application binary interface", allows calling foreign functions from another 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 5.3 displays a rush program which is able to solve this problem.

```
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 5.3 – Rush Program Altering the Variable Behind an Argument

Most parts of this rush program look similar to the one displayed in Listing 5.2. 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

[3]Passing by value means that the value of the argument is copied for the function call

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 whilst 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 [Dan05, 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 in RISC-V [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 whilst 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

[5] Short for: "instruction set architecture"

subroutine. If a return-instructions is used, the value in '`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. However, this concept will also be explained in more detail later. **TODO: Will it?** What the previously explained registers have in common is that they all hold integer values. Depending on the exact RISC-V architecture, all registers either hold 32 or 64 bits of information. For floating-point number values, RISC-V provides other registers. 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

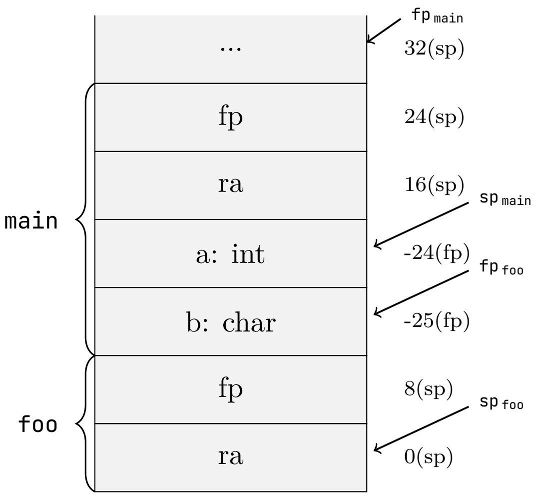


Figure 5.4 – Example Stack Layout in RISC-V

As mentioned in the dedicated **TODO: (sub)** section 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 5.4 should be considered.

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

Listing 5.4 – 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 whilst 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 5.4, 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.

TODO: How can we observe alignment here?

5.2.3. Calling Convention

TODO: did I mention that arguments are saved on the stack?

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].

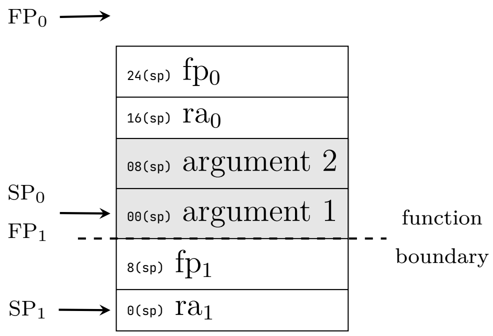


Figure 5.5 – Spilled Registers During a RISC-V Function Call **TODO: Every row should be equal in height**

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 whilst "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 whilst saving the address of the next instruction after the '`call`' instruction in the register '`ra`' [PW17, p. 22]. As hinted 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`' whilst 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.

[6] A macro generating multiple instructions from one pseudoinstruction. Therefore, the actual count of ISA instructions remains low whilst convenience features can be used in assembly [Dan05, p. 68].

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 *kernel*[7] system-calls, like *exit*[8]. Since these concepts must introduce additional logic, the compiler should not be emitted their instructions everytime 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*[9] which is later used by the linker.

5.2.5. RISC-V Assembly

The Listing 5.5 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. **TODO: continue here** 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 75 shows the output assembly generated from this program by the rush RISC-V compiler[10]. 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.

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

Listing 5.5 – Example rush Program Containing Two Functions

[7]**TODO: Explaination + Citation**

[8]Terminates the execution of the program with an arbitrary exit code

[9]**TODO: What are archives + Citation**

[10]Generated in Git commit `8a45d80`, automatically built with this document

```
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 5.5

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[11], 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]. 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 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

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

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]. 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]. 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].

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].

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 5.6 and 5.7 are to be considered.

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

Listing 5.6 – 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 5.7 – RISC-V Assembly Output from Listing 5.6

The code in Listing 5.6 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 5.7 includes the most significant part of the compiler output which represents this rush program[12]. 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)`' whilst '`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. 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.

[12]Generated in Git commit `8a45d80`, automatically built with this document

5.2.7. 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 5.8 shows important struct fields of the rush compiler targeting RISC-V.

```rust
11  pub struct Compiler<'tree> {
        // ...
15      /// Labels and their basic blocks which contain instructions.
        // ...
19      /// Points to the current section which is inserted to.
20      pub(crate) curr_block: usize,
21
22      /// Data section for storing global variables.
23      pub(crate) data_section: Vec<DataObj>,
24      /// Read-only data section for storing constant values (like floats).
25      pub(crate) rodata_section: Vec<DataObj>,
26
27      /// Holds metadata about the current function
28      pub(crate) curr_fn: Option<Function>,
29      /// Holds metadata about the current loop
30      pub(crate) loops: Vec<Loop>,
31
32      /// The first element is the root scope, the last element is the current scope.
33      pub(crate) scopes: Vec<HashMap<&'tree str, Variable>>,
34      /// Holds the global variables of the program.
```

Listing 5.8 – 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 5.9 shows parts of the Rust code which declares the '`Instruction`' enum.

```rust
pub enum Instruction {
    // ...
    Li(IntRegister, i64),
    La(IntRegister, Rc<str>),
    Mov(IntRegister, IntRegister),
    // ...
}
```

Listing 5.9 – 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 5.10 shows parts of the declaration of the 'IntRegister' enum. Like the name implies, this enum holds all possible registers which the architecture provides.

```rust
pub enum IntRegister {
    // ...
    A0,
    A1,
    A2,
    // ...
}
```

Listing 5.10 – 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 5.8. 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. **TODO: describe let and prologue later**

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. **TODO: Explain loop labels later**

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 5.12 and Listing 5.12 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 5.11 – 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 5.12 – Assembly Output of Rush Program in Listing **??**

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 am 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 tries to use the registers without causing conflicts like the previously explained one. Therefore, 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.

- If a register cannot be overwritten, mark it as "used"
- Each function decides which target register will hold its result

- Registers are acquired from a pool of registers
- the first free register from this pool is taken when a new register is requested
- For instance, a new integer register can be acquired by calling the ‘`self.alloc_ireg`’ method.
- If this function is called repeatedly, it would always return the same register, ‘`a0`’ for instance.
- Therefore, registers can be marked as ‘`used`’.
- In this case, the method would return the next free register after the last used one
- **TODO: include graphic here**
- Such a pool exists for both integer and floating-point registers
- Include rust code which calls the ‘`alloc_ireg`’ and ‘`use_reg`’ methods (in the ‘`infix_expr`’ method)
- The latter inserts a register into the ‘`used_registers`’ vector in order to block it temporarely
- Show code of the ‘`alloc_ireg`’ method and explain how it works and why it works like this

- Traverses the tree X
- Show struct fields X
- Explain register allocation and flow of expressions / values
- Explain functions
- Explain let statements
- Explain calls
- Explain loops
- Explain floats

5.3. x86_64: A Compiler for a CISC Architecture

- Present Some example instructions
- Register layout & count
- Calling convention
- INFORMATION: x86: An intel blog stated ca. 3600 Instructions in 2015 [RA17]
- RISC-V Reader: *ASCII adjust after addition* → *aaa* → obsolete, collectively occupy 1.6% of opcode space[PW17, p. 4]
- "Modular vs. Incremental ISAs Intel was betting its future on a high-end microprocessor, but that was still years away. To counter Zilog, Intel developed a stop-gap processor and called it the 8086. It was intended to be short-lived and not have any successors, but that's not how things turned out. The high-end processor ended up being late to market, and when it did come out, it was too slow. So the 8086 architecture lived on—it evolved into a 32-bit processor and eventually into a 64-bit one. The names kept changing (80186, 80286, i386, i486, Pentium), but the underlying instruction set remained intact. —Stephen P. Morse, architect of the 8086"[Mor17]. ⇒ only meant as a temporary architecture

5.3.1. Register Layout

For registers: [Kus18, p. 7]

5.3.2. Memory Access Through the Stack

5.3.3. Calling Convention

5.3.4. X86_64 Assembly

5.4. Conclusion

5.4.1. RISC vs CISC Architectures

TODO: RISC vs. CISC [Dan05, p. 5-6]

- RISC-V was less demanding and better

6. Final Thoughts and Conclusions

Acknowledgements

We would like to express out 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 paper as an additional research project.

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`.

[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.

[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.

[Dan05] 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.

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

[CA14] Bruno Cardoso Lopes and Rafael Auler. *Getting started with LLVM core libraries*. Birmingham, England: 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`.

[Mor17] Stephen P. Morse. "The Intel 8086 Chip and the Future of Microprocessor Design". In: (Apr. 2017), pp. 8–9. DOI: `10.1109/MC.2017.120`.

[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.

[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, England: 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, England: Packt Publishing, Apr. 2021. ISBN: 978-1-83882-495-2.

[McN21] Timothy Samual McNamara. *Rust in Action*. In Action. New York, NY: Manning Publications, Aug. 2021. ISBN: 9781617294556.

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

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

A. Complete Grammar of rush in EBNF Notation

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