The Conversion of Source Code to Machine Code

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

Silas Groh, Mik Müller

January 23, 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 way. 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].

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.

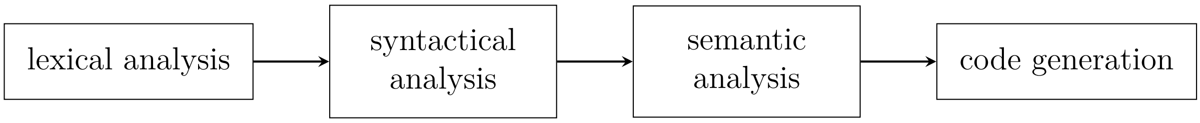


Figure 1.1 – Different Steps of Compilation

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?**

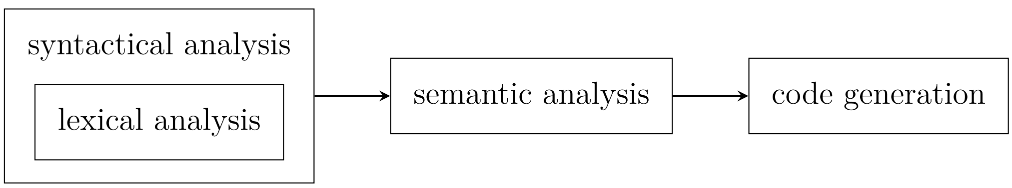


Figure 1.2 – Different Steps of Compilation (altered)

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 `3201cc3`, the entire rush project includes 17119 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 further 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? TODO: explain 1+ repetitions due to '-'**

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

Listing 2.1 – Grammar for Basic Arithmetic in EBNF Notation

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.

[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.

```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.

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. **TODO: mention backtracking**

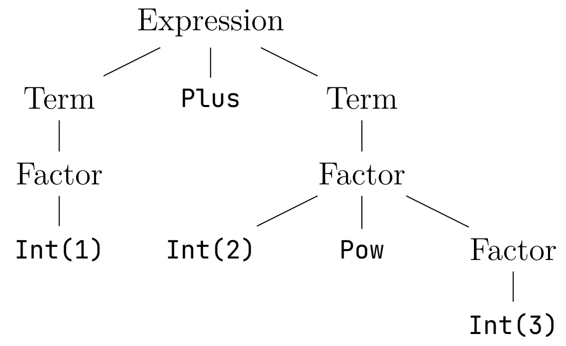


Figure 2.1 – Abstract Syntax Tree for '`1+2**3`'

An example for LR parsing is the *shift-reduce* parsing approach. ...**TODO: explain shift-reduce parsing?**...

LL parsers are usually much simpler to implement, but come with a limitation. By design, they must recognize a node by its first $n = k+1$ tokens, where n is the window size. However, due to that restriction, not every context-free language can be parsed by 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**

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

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) }`

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.

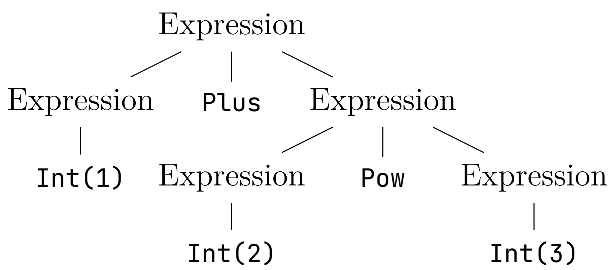


Figure 2.2 – Abstract Syntax Tree for ‘`1+2**3`’ Using Pratt-Parsing

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

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

the earlier they are evaluated. All unrelated tokens are simply assigned a precedence of 0 for left and right.

```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

The `expression` method on the `Parser` struct is then modified to take a parameter for the current precedence as seen in Listing 2.7. 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. 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 2.6. 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. The `infix_expr` method itself simply stores the right 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.

```rust
fn grouped_expr(&mut self) -> Result<'src, Spanned<'src, Box<Expression<'src>>>> {
    // ...
    let expr = self.expression(0)?;
    // ...
}
```

Listing 2.6 – Pratt-Parser: Call to `expression` With the Lowest Precedence

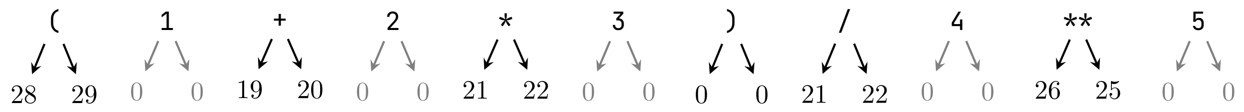


Figure 2.3 – Token Precedences for Input ‘`(1+2*3)/4**5`‘

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*

[4]Short for ‘left-hand side’

generators that generate parsers based on some specification of the desired syntax. Often parser generators define a domain specific grammar notation for the syntax specification. **TODO: some also allow traditional grammar notations, and e.g., 'nom' is simply a Rust crate** ...

1. different strategies (LL, LR) and top-down / bottom-up
2. lookahead (to avoid backtracking)
3. pratt-parsing
4. parser generators

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

[5] Later referred to as "analyzer"

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

Listing 2.8 – 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.

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,
19     builtin_functions: HashMap<&'static str, BuiltinFunction>,
20     /// The names of all used builtin functions
21     used_builtins: HashSet<&'src str>,
22     /// Specifies whether there is at least one `break` statement inside the
   ↪ current loop.
23     current_loop_is_terminated: bool,
24     /// The source code of the program to be analyzed
25     source: &'src str,
26 }
```

Listing 2.9 – Attributes of the `analyzer` struct

Listing 2.9 displays the struct fields of the semantic analyzer. The 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. An example for diagnostic messages can be found in Listing 2.10. The field `curr_func_name` saves the name of the current function. Moreover, 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 `scopes` field in line 16 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 an error message is displayed in Listing 2.10, it occurs when another value is assigned to an immutable variable.

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 above again. 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 2.11 shows how one part of validating the `main` function's signature works.

```
404             // the main function must have 0 parameters
405             if !node.params.inner.is_empty() {
406                 self.error(
407                     ErrorKind::Semantic,
408                     format!(
409                         "the `main` function must have 0 parameters, however {} {} defined",
410                         node.params.inner.len(),
411                         match node.params.inner.len() {
412                             1 => "is",
413                             _ => "are",
414                         },
415                     ),
416                     vec!["remove the parameters: `fn main() { ... }`".into()],
417                     node.params.span,
418                 )
419             }
```

Listing 2.11 – Analyzer Validating the Signature of the '`main`' Function

This code is only executed if the `curr_func_name` field of the analyzer holds the value `main`. Therefore, these special checks are only performed for the `main` function. The if-clause in line 405 checks if the node's `params` vector contains any items. If this is the case, the analyzer generates an error message describing the issue. However, we have not yet explained how error handling in the analyzer works. When examining the signatures of some of the analyzer's methods, it becomes obvious that these methods do not return errors. Instead, the `self.error` method is invoked. This method requires the type of the error to be reported, its message, where it occurred and optional hints to display. The error is reported because this method pushes a new `Diagnostic` struct into the `diagnostics` vector.

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 statements, the `let_stmt` method is invoked.

```
    fn let_stmt(&mut self, node: LetStmt<'src>) -> AnalyzedStatement<'src> {
        // save the expression's span for later use
        let expr_span = node.expr.span();

        // analyze the right hand side first
        let expr = self.expression(node.expr);
```

Listing 2.12 – Beginning of the `let_stmt` Method

First, the initializing expression of the let-statement is analyzed 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 variable insertion is displayed in Listing 2.13.

```
644         if let Some(shadowed) = self.scope_mut().insert(
645             node.name.inner,
646             Variable {
647                 type_: match node.type_.map_or(expr.result_type(), |type_| type_.inner) {
648                     // map `!` to `{unknown}` to prevent misleading warnings
649                     Type::Never => Type::Unknown,
650                     type_ => type_,
651                 },
652                 span: node.name.span,
653                 used: false,
654                 mutable: node.mutable,
655                 mutated: false,
656             },
657         ) {
```

Listing 2.13 – Insertion of a Variable Into the Current Scope

As seen in line 652, the insertion includes the variable's span. Since the span includes the location of where the variable was declared, it can later be used in error messages like the one displayed in Listing 2.10. Furthermore, the inserted information 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 2.10 was generated as a result. By default, the variable is set to non-mutated (line 655). Most rush compiler backends use this information about actual mutation in order to implement some optimizations. 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 associated data has now been overwritten. This overwriting of variables 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 an error 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. 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.

```rust
1007     fn expression(&mut self, node: Expression<'src>) -> AnalyzedExpression<'src> {
1008         let res = match node {
1009             Expression::Block(node) => self.block_expr(*node),
1010             Expression::If(node) => self.if_expr(*node),
1011             Expression::Int(node) => AnalyzedExpression::Int(node.inner),
1012             Expression::Float(node) => AnalyzedExpression::Float(node.inner),
1013             Expression::Bool(node) => AnalyzedExpression::Bool(node.inner),
```

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. In this function, the recursive tree-traversal algorithm used in the analyzer is clearly visible. For instance, if the current expression is a block-expression (line 1009), the responsible method is called. Since rush allows blocks which contain expressions, a block containing another block is a legal construct. 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 AST nodes may contain themselves. For simple types of expressions like integers or floats, further analysis is omitted. Since these types of expressions are constant, the method can directly return an analyzed version of the expression. 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.

```rust
impl AnalyzedExpression<'_> {
    pub fn result_type(&self) -> Type {
        match self {
            Self::Block(expr) => expr.result_type,
            Self::Int(_) => Type::Int(0),
            Self::Float(_) => Type::Float(0),
            Self::Bool(_) => Type::Bool(0),
            Self::Char(_) => Type::Char(0),
            Self::Ident(expr) => expr.result_type,
            Self::If(expr) => expr.result_type,
            Self::Prefix(expr) => expr.result_type,
            Self::Infix(expr) => expr.result_type,
            Self::Assign(expr) => expr.result_type,
            Self::Call(expr) => expr.result_type,
            Self::Cast(expr) => expr.result_type,
            Self::Grouped(expr) => expr.result_type(),
        }
    }
```

Listing 2.15 – Obtaining the Type of Expressions

The code in Listing 2.15 shows how the type of any analyzed expression can be obtained. For constant expressions like `AnalyzedExpression::Int(_)`, the determination of its type is straight-forward. 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. This way, 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, we have implemented the `Unknown` type. 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 was caused solely by a previous error. Therefore, errors do not cascade, meaning that an undeclared variable will not cause another type error.

After the let-statements, the `exit` function is called. Here, the analyzer calls the `call_expr` function which is responsible for analyzing the validity of function calls. First, the call parameter expressions are analyzed. Therefore, the expression `two + three` is traversed before further analysis can proceed. 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 since it was not declared, the `Unknown` type would be used here in order to prevent cascading errors. Because the values of the two variables should be added, the method `infix_expr` is called. This method validates several constraints. For instance, the operands must both be of the same type. Here, both operands are integers, thus complying with the specification. 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 a classical example for tree nodes which save their result type as a struct field on their own. Therefore, the type of infix-expressions can be accessed like it is displayed in line 140 of the Listing 2.15. Now that the analysis of the argument expression has completed, its compatibility with the declared parameter must be validated.

```rust
            (arg_type, param_type) if arg_type != param_type => self.error(
                ErrorKind::Type,
                format!("mismatched types: expected `{param_type}`, found `{arg_type}`"),
                vec![],
                arg_span,
            ),
```

Listing 2.16 – Validation of Argument Type Compatibility in the Analyzer

The listing 2.16 shows a part of the `arg` function which is responsible for validating that a function call argument is compatible with the declared parameter. 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. Since the example from above presents a syntactically and semantically valid rush program, the analyzer accepts this program and returns its annotated tree.

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].

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

Listing 2.17 – 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
}
```

textnormal color blackRedundant texttt while Loop Inside a rush Program

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

textnormal color blackFaster Loop Algorithm Implemented in rush

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 2.18.

```
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 2.18 – 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
}
```

textnormal color blacktexttt while Loop Inside a rush Program Which Never Iterates

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 2.18. 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.

```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(
                format!("expected an expression, found `{invalid}`"),
                self.curr_tok.span,
                self.lexer.source(),
            ));
        }
    };

    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)
}
// ...
fn infix_expr(
    &mut self,
    start_loc: Location<'src>,
    lhs: Expression<'src>,
    op: InfixOp,
) -> Result<'src, Expression<'src>> {
    let right_prec = self.curr_tok.kind.prec().1;
    self.next()?;
    let rhs = self.expression(right_prec)?;
    // ...
}
```

Listing 2.7 – Pratt-Parser Implementation **TODO: split into 2 listings?**

```
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 2.10 – Output When Compiling an Invalid rush Program

3. Interpreting the Program

TODO: write short introduction on interpreters

3.1. Tree-Walking Interpreters

TODO: @RubixDev: write this section

3.2. Using a Virtual Machine

Just like a tree-walking interpreter, a virtual machine presents a way of implementing an interpreter for a programming language. However, the way a virtual machine operates 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*[1] is a software entity which emulates how a computer interprets instructions. Just like a real computer, a virtual machine executes low-level instructions directly. Therefore, the VM is unable to traverse the AST and therefore relies on a compiler to generate 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[2], 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.

- (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.

[1] May later be shortened to "VM"
[2] Short for "arithmetic logic unit"

- (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 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 constrains 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, implementation of 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.

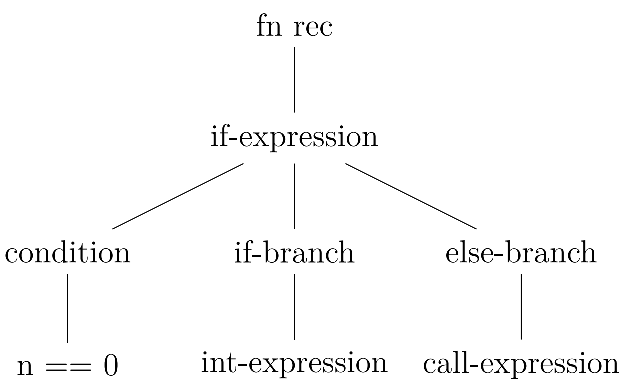
3.2.3. Comparing the VM to the Tree-Walking Interpreter

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 3.1 displays a recursive function implemented in rush.

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

Listing 3.1 – 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 3.1 – Abstract Syntax Tree and VM Instructions of a Recursive rush Program

Figure 3.1 displays a heavily simplified syntax tree and rush VM instructions representing the function displayed in Listing 3.1. 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 3.1 represent the program in Listing 3.1. 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[3]. 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 is 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 3.2 – Struct Definition of the Vm

Listing 3.2 displays the struct definition of the rush VM. In line 18, a field called `stack` is defined. Like explained above, the rush VM uses a stack for storing temporary values.

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

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, the `call_stack` is declared. This field also behaves like a stack and is responsible for managing function calls and returns. The instructions in Figure 3.1 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 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 3.2 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 3.2, 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.

In order to get a deeper understanding of the addressing modes, a practical example can be considered. The code in Listing 3.3 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.

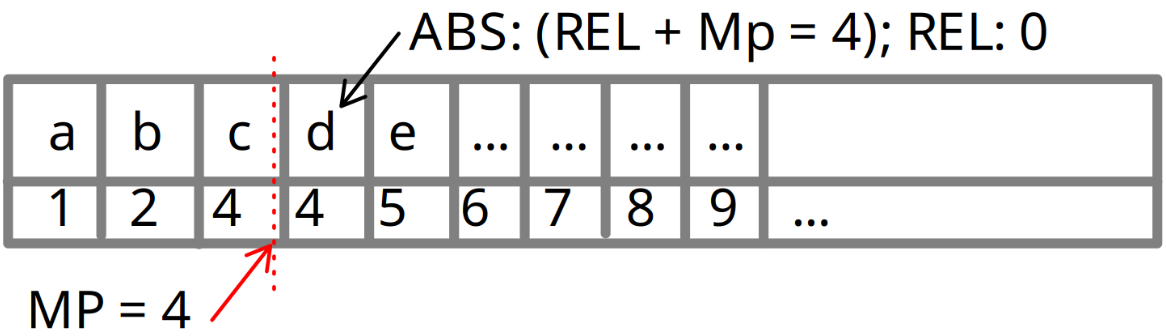


Figure 3.2 – DRAFT: Linear Memory of the rush VM

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

Listing 3.3 – 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 3.4 shows the VM instructions generated from the rush program in Listing 3.3.

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

Listing 3.4 – 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.

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 3.1. For this, we should consider the instructions in Figure 3.1 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 3.3 displays the state of the VMs call stack after the `call 1` instruction has been executed. During execution of a call-instruction, the VM pushes a new stack frame onto its call stack. Listing 3.5 shows how a call frame is implemented.

```rust
26 }
27
28 #[derive(Debug, Default)]
29 struct CallFrame {
30     /// Specifies the instruction pointer relative to the function
31     ip: usize,
```

Listing 3.5 – Struct Definition of a `CallFrame`

In this implementation, each call frame holds two important pieces of information. In line 28 of Listing 3.5, 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 30. 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 3.3 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`[4] instruction, it should leave the current function. However, it should also know where

[4] Short for "return"

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

Figure 3.3 – Call Stack of the rush VM

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 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.

Now that we have explained the semantic meaning of the instructions in Figure 3.1, we will explain how the fetch-decode-execute cycle works in the VM. The code in Listing 3.6 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 3.6 – 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 fundamental role in the VM. Since the fetch-decode-execute cycle executes instructions repeatedly, the main construct in the function is a while-loop. 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 168, 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 can return a runtime error, such as an integer-overflow error. Furthermore, this method may return an integer representing the exit code of the program. However, if the method returns none of these two possible types, the fetch-decode-execute cycle continues. In this code however, one cannot observe the instruction pointer being incremented. 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 3.7.

```rust
            Instruction::BitAnd => {
                let rhs = self.pop();
                let lhs = self.pop();
                self.push(lhs.bit_and(rhs))?;
            }
            Instruction::BitXor => {
                let rhs = self.pop();
                let lhs = self.pop();
                self.push(lhs.bit_xor(rhs))?;
            }
        }
        self.call_frame_mut().ip += 1;
        Ok(None)
```

Listing 3.7 – Parts of the `run_instruction` Method of the rush VM

The code in Listing 3.7 displays the last part of the `run_instruction` method. This method mainly consists of an algorithm mathing the current instruction in order to execute specific code representing the instruction's semantic meaning. In this example, the implementations of the `bitand` and `bitxor` instructions are visible. Both instructions first pop two elements from the stack since they represent the operands of the underlying logical computation. Then, a corresponding helper function is invoked on the left hand side operand. The helper function then performs the actual computation of the logical operation. Most of the 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 and nothing is returned. For some special instructions, such as the jump-instructions, the instruction pointer should not be incremented since it would interfere with the jump.

```rust
            Instruction::Jmp(idx) => {
                self.call_frame_mut().ip = *idx;
                return Ok(None);
            }
```

Listing 3.8 – Execution of the `jmp` Instruction in the rush VM

As seen in Listing 3.8, the `jmp` instruction only sets the instruction pointer to the target index specified in the instruction operand. Then, the code returns from the method so that the instruction pointer is not incremented later. This method represents both the *decode* and *execute* step since it first matches (*decode*) and then interprets (*execute*) the current instruction. 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 next 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 and reduced memory usage at runtime. The downsides include the need for a compiler targeting the VM, thus making its implementation more demanding compared to a tree-walking interpreter.

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

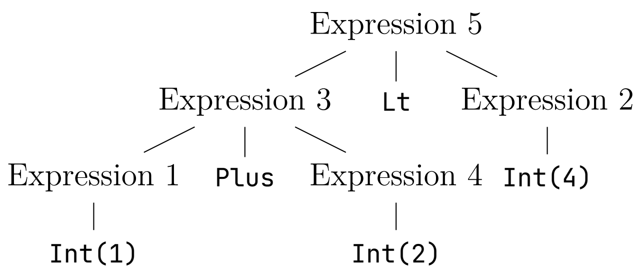


Figure 4.1 – Abstract Syntax Tree for ‘`1+2 < 4`’

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.

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 some AST-nodes. For instance, the compiler translates infix-expressions, such as $n + m$, into instructions using the `infix_expr` method. A part of this method is displayed in Listing 4.2.

```
                self.expression(node.lhs);
                self.expression(node.rhs);
                self.insert(Instruction::from(op));
```

Listing 4.2 – Compilation of Infix-Expressions Targeting the VM

Here, the left hand side expression is compiled first. Next, the right-hand side is compiled too. Finally, the appropriate arithmetic instruction is inserted. The final instruction is generated by a helper function which converts an infix-operator into a matching instruction. Most of the other compilers we have implemented for the rush project required significantly more code in order to implement the translation of infix-expressions.

```
    fn expression(&mut self, node: AnalyzedExpression<'src>) {
        match node {
            AnalyzedExpression::Int(value) => self.insert(Instruction::Push(Value::Int(value))),
            AnalyzedExpression::Float(value) => self.insert(Instruction::Push(Value::Float(value))),
            AnalyzedExpression::Bool(value) => self.insert(Instruction::Push(Value::Bool(value))),
            AnalyzedExpression::Char(value) => self.insert(Instruction::Push(Value::Char(value))),
```

Listing 4.3 – Compilation of Expressions Targeting the VM

The code in Listing 4.3 shows the top of the `expression` method in the VM compiler. When we examine the method's signature, it becomes apparent that it only consumes an `AnalyzedExpresssion`. However, the method does not return anything which represents the runtime value of the expression. This is possible because the types of expression displayed in the snipped are pushed onto the stack directly. By pushing the values of atomic 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 usually returns the register which contains the value of the compiled expression at runtime. This way, other parts of the compiled program can still use the runtime values of compiled expressions.

The rush VM includes a special instruction for the mathematical power operation (`**`). Since many real architectures lack such a power instruction, implementing a rush compiler targeting the VM has proven to be less demanding in this way. On the opposite, many other rush compilers demanded implementation of special edge-cases in order to make compiling power-expressions feasible. Furthermore, the VM also includes an `exit` instruction which terminates the fetch-decode-execute cycle instantly. Here, the VM would come to a halt instantly, returning the top value on the stack as its exit code. These examples showed how a carefully chosen target architecture simplifies the implementation of its compiler by a great deal.

However, there is also one aspect of the VM which made implementation of the compiler targeting the VM more demanding than usual. For instance, in most Assembly dialects, *labels* can be used to allow jumps between blocks of code. However, the VM intentionally does not support the use of such labels. Since the VM would have to look up the exact instruction index of a label at runtime, each jump targeting a label would involve some additional overhead. This overhead is eliminated by the assembler during assembly of a program. Since the assembler performs these lookups during translation, the CPU does not have to deal with label lookups at runtime. Like seen in the previous examples, jumping VM instruction require the exact index of the target instruction as their operands. Therefore, the exact target index to which the instruction should jump must be known. To illustrate this issue, we will consider how loops are implemented in the VM. The rush code in Figure 4.2 presents a program containing a loop. In the loop's body, the variable `n` is incremented by 1. Next, the `break` keyword is used to terminate loop execution. Therefore, the total amount of iterations is 1.

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

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

Figure 4.2 – How Loops are Interpreted by 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 the variable `n`. The instructions in the lines 4–9 are used to increment the variable `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` is 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. If this occurs, 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.

```
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);
```

Listing 4.4 – Implementation of Loops in the rush VM Compiler

The statement in line 337 inserts the instruction responsible for jumping back to the start of the loop's body. In line 340, the top loop is popped from the `loops` 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 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 while the second lists saves instructions generated by `continue` statements. For instance, if the compiler encounters a `break` statement, the code in Listing 4.5 is executed.

```
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             }
```

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, the `jmp` instruction is inserted containing a placeholder target index. Therefore, at the end of each loop's compilation, 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

TODO: @RubixDev must write this section

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 where 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 also use 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 using LLVM, it is responsible for generating target-specific code. Furthermore, LLVM 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 must only analyze the source program to create an AST. Therefore, LLVM represents the *back end* of a compiler system. Then, the AST is traversed and the API of LLVM is used to construct an intermediate representation of the program so that the system can understand it. Next, LLVM compiles the input program 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.

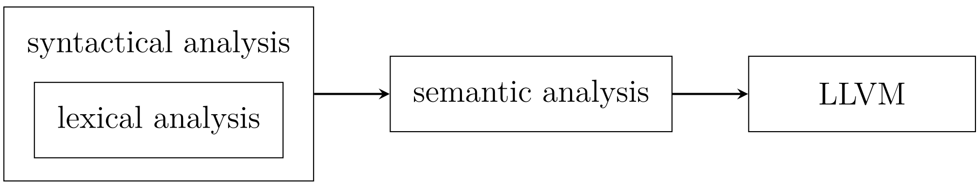


Figure 4.3 – Steps of Compilation When Using LLVM

4.3.2. The LLVM Intermediate Representation

The intermediate representation (*IR*) represents the source program in a low-level but target-independent way. Through the use of the LLVM intermediate representation, 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 at compile time compared to other compiler solutions. Furthermore, LLVM communicates a lot of information to the linker. As a result of this, many so-called *link time optimizations* can be achieved which are not present in most other compilers [Lat02, p. 5]. Therefore, programs compiled using LLVM as the backend will often run significantly faster due to the many aggressive optimizations introduced by the system.

LLVM provides many APIs for interacting with the IR in memory, so that it can be created by a compiler frontend without it being written onto a file. The official API for LLVM is for C++ and C. 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++. A program represented using the IR always obeys 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 LLVM IR.
- Each function contains several *basic blocks*. Such a basic block contains a sequence of instructions. Blocks always have to be terminated using a jumping or returning instruction. However, a block must never be terminated twice.
- As mentioned above, each basic block contains a sequence of *instructions*. Each instruction holds a semantic meaning and represents a part of the source program.

[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 a program represented using the IR, LLVM is able to perform many aggressive optimizations on the IR during later stages of compilation. This way, LLVM 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 represents a virtual architecture which is able to represent most common types processors. However, the IR avoids machine specific constraints like register-count or low-level calling conventions. The virtual architecture provides an infinite set of virtual registers which can hold the value of primitives like *integers*, *floating-point numbers*, and *pointer*. All registers in the IR use the *SSA*[1] form in order to allow more optimizations. In order to enforce the correctness of the type information included in the IR, the operands of an instruction all obey LLVM's type rules [Lat02, p. 14-17].

In order to understand how the LLVM IR represents a program, we now consider the calculation of 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 4.6 displays LLVM IR representing this rush program. The IR was generated by the LLVM targeting rush compiler[2].

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

[2]Generated in Git commit `3201cc3`, automatically built with this document

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

The code displayed in the snipped is part of a LLVM module. In the lines 5 and 29, 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. Furthermore, if we examine the signature of the `fib` function in line 5 of the IR, it becomes apparent that the function returns an `i64`. In rush, each `int` can hold 64-bit signed numbers, therefore the `i64` LLVM type represents the rush `int` type. Furthermore, we can observe that the function takes an i64 parameter named `%0`. This parameter represents the `n` parameter in our 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, out compiler uses the `entry` block solely for variable declarations. In line 7, the block is terminated using the `br`[3] instruction. This instruction jumps to the beginning of the block specified in its operand. In this case, the target of the jump is the beginning of the `body` basic block in the same function. Due to constraints introduced by its internal optimizations, LLVM only allows targeting blocks contained in the same function.

In line 10, the `icmp slt`[4] is used in order to compare the runtime value of the parameter `%0` to a constant 2. The boolean result is then saved in the virtual register `%i_lt`. Here, it becomes apparent that LLVM's virtual registers can be given arbitrary names. In places where

[3] Short for "branch"
[4] Short for "integer compare (signed less than)"

this is possible, our compiler will use names which will make reading the IR easier for humans. In line 11, another branch-instruction is used. However, this time, the jump is placed under the condition that the value of `%i_lt` is true. Here, we can see that LLVM instructions are able to operand on different type of operands depending on what the instruction should do. Furthermore, the `else` label is also an operand of the branch-instruction. This is because conditional jumps in LLVM always require an alternative jump to perform if the condition is false at runtime. As the names `then` 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 `then` block. However, this block only contains one instruction jumping to the `merge` block.

In line 17, the `phi` instruction is used. These so called ϕ-nodes are necessary due to the SSA form used in the LLVM 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 `then` 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 `then` 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 21 with the `sub` instruction. In this case, the instruction subtracts 2 from the parameter `%0` and saves the result in `\%i_sum`. 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 25 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 17 so that the result of the recursive calls is used as the result of the if-expression. Finally, the `ret` instruction in line 18 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 36, 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.

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 4.7 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>,
```

Listing 4.7 – 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 LLVM which allows generation of IR whilst only using in-memory structures. 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.

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

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

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

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

The source program on the left side contains the `foo` and the `main` functions. These functions are declared in the lines 5 and 17 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 18 of the IR, the parameters `n` and `m` are added together. The result of this addition is then used in order to call the `exit` function. This function call takes place in line 11 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 4.8 displays the top part of the method responsible for translating the `main` function.

```
328         let fn_type = if self.compile_main_fn {
329             self.context.i32_type().fn_type(&[], false)
330         } else {
331             self.context.void_type().fn_type(&[], false)
332         };
333 
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);
```

Listing 4.8 – Compilation of the '`main`' Function Using LLVM

In the lines 334–336, the `main` function is added to the current LLVM module. Here, the name of the function is specific by the `fn_name` variable. 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*. 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 the Inkwell `FunctionValue`. How the `entry_block` field in line 346 is used every time a pointer is declared 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 arguments 2 and 3. In order to understand how this compiler translates function calls, we will now consider Listing 4.9.

```
        let res = self
            .builder
            .build_call(func, &args, format!("ret_{}", node.func).as_str())
            .try_as_basic_value();
```

Listing 4.9 – Compilation of Call-Expressions Using LLVM

The code in Listing 4.9 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 4.10.

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

Listing 4.10 – Compilation of Expressions Using LLVM

The code in Listing 4.10 shows the top part 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 *value* at runtime of the program. Therefore, the '`BasicValueEnum`' returned by the function represents the virtual register holding 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 of the compiler, using the current AST node 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 4.11 – Compilation of Integer Infix-Expressions Using LLVM

The code in Listing 4.11 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`' integer 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 righthand 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 a 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 4.12.

```
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                 self.scope_mut()
623                     .insert(node.name, Variable::new_mut(ptr,
    ↪   node.expr.result_type()));
624             }
```

Listing 4.12 – Compilation of Let-Statements Using LLVM

The code in Listing 4.12 displays the top part 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 after the line 613 is only executed if the variable was declared as mutable. Therefore, in order to present this code in action, 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 4.13 shows the `alloc_ptr` method of the compiler.

```rust
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 4.13 – 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

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*[5] 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*. Furthermore, 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. An essential part of the liker's actions is presented by relocation and code modification. The object file generated by an assembler or a compiler uses unrelocated addresses. Furthermore, any data or code defined outside the file is represented by zeros. Therefore, the linker needs to modify the code in order to update the actual addresses

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

assigned. 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 4.14 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 `/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.

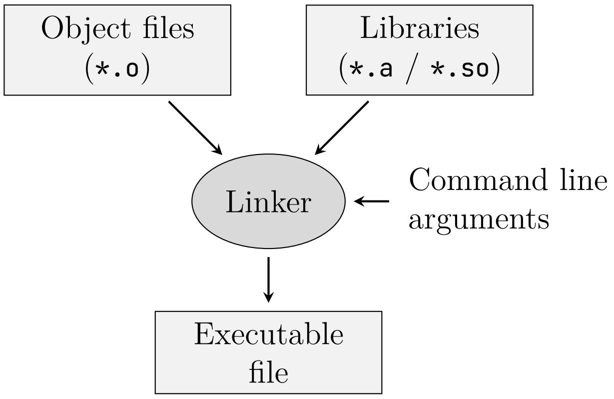


Figure 4.5 – How a Linker Works

```
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 4.14 – Using LD to link the LLVM output

TODO: Elaborate further

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

TODO: Difficult: blocks need to be terminated, loops and alloca

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

5. Compiling to Low-Level Targets

5.1. Low-Level Programming Concepts

5.2. RISC-V: A RISC Architecture

5.3. x86_64: A CISC Architecture

6. Final Thoughts and Conclusions

Acknowledgements

We would like to express out sincere gratitude towards our supervisor - Sonja Soković 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`.

[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.

[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.

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

[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.

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

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