Contents

List of Figures

List of Tables

List of Listings

1. Introduction

See Chapter 2, Section 2.1, Section 2.1.1, Figure 2.1, Table 1.3, and Listing 1.1.

1.1. Stages of Compilation

a

Figure 1.1: Steps of compilation.

a

Figure 1.2: Steps of compilation. (altered)

1.2. The rush Programming Language

```
fn fib(n: int) -> int {}
```

Listing 1.1: Generating Fibonacci numbers using rush.

1.2.1. Features

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

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

Table 1.3: Data types in the rush programming language.

2. Analyzing the Source

2.1. Lexical and Syntactical Analysis

2.1.1. Formal Syntactical Definition by a Grammer

```
fn main() {}
```

Listing 2.1: Grammar for basic arithmetic in EBNF notation.

2.1.2. Grouping Characters Into Tokens

```
fn main() {}
```

Listing 2.2: The rush ‘`Lexer`’ struct definition.

Table 2.1: Advancing window of a lexer.

```
fn main() {}
```

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

2.1.3. Constructing a Tree

a

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

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

Operator Precedence

```rust
fn main() {}
```

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

a

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

```rust
fn main() {}
```

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

```rust
fn main() {}
```

Listing 2.6: Pratt-parser: Implementation for expressions.

```rust
fn main() {}
```

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

```rust
fn main() {}
```

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

a

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

Parser Generators

2.2. Semantic Analysis

2.2.1. Defining the Semantics of a Programming Language

2.2.2. The Semantic Analyzer

```
fn main() {}
```

Listing 2.9: A rush program which adds two integers.

Implementation

```
fn main() {}
```

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

```
fn main() {}
```

Listing 2.11: Output when compiling an invalid rush program.

```
fn main() {}
```

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

```
fn main() {}
```

Listing 2.13: Analyzer: The ‘`let_stmt`’ method.

```
fn main() {}
```

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

```
fn main() {}
```

Listing 2.15: Analyzer: Obtaining the type of expressions.

```
fn main() {}
```
Listing 2.16: Analyzer: Validation of argument type compatibility.

```
fn main() {}
```

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

Early Optimizations

```
fn main() {}
```

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

```
fn main() {}
```

Listing 2.19: Analyzer: Loop optimization.

a

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

3. Interpreting the Program

3.1. Tree-Walking Interpreters

```rust
fn main() {}
```

Listing 3.1: Tree-walking interpreter: Type definitions.

3.1.1. Implementation

```
fn main() {}
```

Listing 3.2: Tree-walking interpreter: ‘`Value`’ and ‘`InterruptKind`’ definitions.

3.1.2. How the Interpreter Executes a Program

a

Figure 3.1: Call stack at the point of processing the ‘`return`’ statement.

```
fn main() {}
```
Listing 3.3: Tree-walking interpreter: Beginning of execution.

```
fn main() {}
```
Listing 3.4: Tree-walking interpreter: Calling of functions.

```
fn main() {}
```
Listing 3.5: Example rush program.

3.1.3. Supporting Pointers

3.2. Using a Virtual Machine

3.2.1. Defining a Virtual Machine

```
fn main() {}
```

Listing 3.6: Struct definition of the VM.

3.2.2. Register-Based and Stack-Based Machines

3.2.3. The rush Virtual Machine

```
fn main() {}
```

Listing 3.7: Minimal pointer example in rush.

a

Figure 3.2: Linear memory of the rush VM.

```
fn main() {}
```

Listing 3.8: VM instructions for the minimal pointer example.

3.2.4. How the Virtual Machine Executes a rush Program

```
fn main() {}
```

Listing 3.9: A recursive rush program.

```
fn main() {}
```

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

a

Figure 3.3: Example call stack of the rush VM.

```
fn main() {}
```

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

3.2.5. Fetch-Decode-Execute Cycle of the VM

```
fn main() {}
```

Listing 3.12: The ‘run’ method of the rush VM.

```
fn main() {}
```

Listing 3.13: Parts of the ‘`run_instruction`’ method of the rush VM.

3.2.6. Comparing the VM to the Tree-Walking Interpreter

a

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

4. Compiling to High-Level Targets

4.1. How a Compiler Translates the AST

a

Figure 4.1: Abstract syntax tree for ‘`1 + 2 < 4`’.

4.2. The Compiler Targeting the rush VM

4.3. Compilation to WebAssembly

4.3.1. WebAssembly Modules

4.3.2. The WebAssembly System Interface

4.3.3. Implementation

Function Calls

Logical Operators

4.3.4. Considering an Example rush Program

4.4. Using LLVM for Code Generation

4.4.1. The Role of LLVM in a Compiler

a

Figure 4.2: Steps of compilation when using LLVM.

4.4.2. The LLVM Intermediate Representation

Structure of a Compiled rush Program

4.4.3. The rush Compiler Using LLVM

4.4.4. Final Code Generation: The Linker

a

Figure 4.3: The linking process.

4.4.5. Conclusions

4.5. Transpilers

5. Compiling to Low-Level Targets

5.1. Low-Level Programming Concepts

5.1.1. Sections of an ELF File

5.1.2. Assemblers and Assembly Language

a

Figure 5.1: Level of abstraction provided by assembly.

5.1.3. Registers

a

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

5.1.4. Using Memory: The Stack

Alignment

a

Figure 5.3: Examples of memory alignment.

5.1.5. Calling Conventions

5.1.6. Referencing Variables Using Pointers

5.2. RISC-V: Compiling to a RISC Architecture

Table 5.1: Registers of the RISC-V architecture.

5.2.1. Register Layout

5.2.2. Memory Access Through the Stack

a

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

5.2.3. Calling Convention

5.2.4. The Core Library

5.2.5. RISC-V Assembly

5.2.6. Supporting Pointers

5.2.7. Implementation

Struct Fields

Data Flow and Register Allocation

a

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

Functions

Let Statements

Function Calls and Returns

Loops

5.3. x86_64: Compiling to a CISC Architecture

5.3.1. x64 Assembly

5.3.2. Registers

Table 5.2: General purpose registers of the x64 architecture.

5.3.3. Stack Layout and Calling Convention

a

Figure 5.6: Stack layout of the x64 architecture.

5.3.4. Implementation

Struct Fields

Memory Management

Register Allocation

Functions

Function Calls

Control Flow

a

Figure 5.7: Structure of if-expressions in assembly.

Integer Division and Float Comparisons

Pointers

5.4. Conclusion: RISC vs. CISC Architectures

6. Final Thoughts and Conclusions

List of Figures

List of Tables

List of Listings

Bibliography