Compiler

Bachelor Project - Technical Appendix

Aarhus Institut for Elektro- og Computerteknologi

Authors:
Sune Andreas Dyrbye, 201205948
Morten Høgsberg, 201704542

Supervisor:

Anslag:

Afleveringsdato: Eksamineringsdato:

Resumé

Contents

1 Abbreviations

Abbreviations

Table 1

Abbreviation	Meaning
CLI	Command Line Interface
AST	Abstract Syntax tree
IR	Intermediate Representation

2 Introduction

3 Problem Statement

3.1 Background

As of 2023, the top four most popular programming languages: Python, JavaScript, TypeScript, and Java permit, to various degrees, a loose or optional type system[2]. This flexibility often results in ambiguities that may hinder the readability and maintainability of code, as type deductions occur either at compile time or runtime. With increasing complexities in software systems, there is an emerging need for a language and compiler that enforce discipline and explicitness in code, while offering robust control over system memory.

3.2 Problem

Developers currently can opt for dynamic or weak typing in popular programming languages, which may compromise code readability and maintainability due to implicit type deduction. This raises challenges in discerning the intentions of the developer solely based on the source code. Moreover, these languages often utilize garbage collectors, sacrificing control over system memory.

3.3 Objectives

To design and develop a custom programming language that mandates explicit typing for all data structures and variables. To implement a compiler for the language that performs zero type inference and provides clear error messages where ambiguities or mistakes are detected. To achieve a balance between the explicitness of data types and the conciseness of the language syntax, promoting readability. To offer precise control over system memory while ensuring safety and avoiding the performance drawbacks of garbage collection.

4 Requirements

The compiler will compile a custom programming language, that is inspired by Rust.
Requirements for the compiler, and accompanying language is listed below, using MoSCoW prioritisation:

4.1 Language requirements

- The language **must** be Turing complete
  - The language **must** contain loops
    - The loops **could** be recursive functions
  - The language **must** contain variables
    - The variables **could** be immutable, unless explicitly made mutable
  - The language **must** allow conditional code execution
  - The language **must** be able to do basic arithmetic[1]
- The language **must** contain functions
- The language **must** be strongly typed[2] and statically typed[3]
- The language **must** allow an output
- The language **must** support the following type primitives:
  - 32-bit Integer
  - 64-bit Floating point
  - 8-bit Character
  - Boolean
- The language syntax **must** follow the language specification in appendix A
- The language **should** have memory-management with Rust inspired borrow checker
- The language **could** allow a runtime input
- The language **could** have error handling
- The language **could** have native array support
- The language **could** allow access to system resources
- The language **wont** have classes
- The language **wont** have tooling/ecosystem/debugger

[1]Addition, subtraction, multiplication, division and modulus
[2]Strongly typed: Variable types does not change, except by explicit casting
[3]Statically typed: All variables must have an explicit type at initialisation

4.2 Compiler requirements

- Compiler **must** be able to compile cross-platform
- Compiler **must** crash on invalid inputs
- Compiler **must** pass all test cases
- Compiler **must** generate an intermediary language from an AST
- Compiler **should** have clear error messages
- Compiler **wont** have intuitive CLI
- Compiler **wont** support several national languages

5 Architecture

On the topic of architecture, the *Rusticque* compiler is comprised of five primary components: the *Lexical Analyzer*, the *Parser*, the *Static Code Analyzer*, the *Code Generator* and the *GCC Compiler*. These components form a pipeline, where the output of one component is the input of the next; in tandem, they constitute the complete compilation process.

Rusticque is a strong statically typed language with a focus on memory safety. It's explicit type system is designed to minimise confusion and obfuscation as each variable must be explicitly declared with a type. It's memory safety is heavily inspired by Rust, and is achieved through the use of a *Borrow Checker* which enforces a subset of Rust's borrow checking rules.

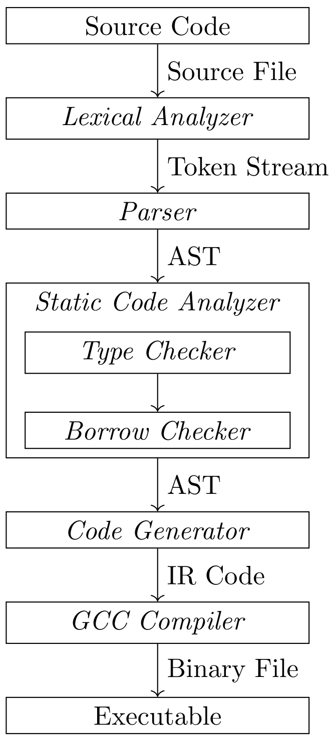

Figure 1: The compiler pipeline, transforming source code into an executable.

To facilitate the *Static Code Analyzer* and *Code Generator* components in way that is both efficient and modular, the *Rusticque* compiler is written using a visitor pattern. This pattern effiently enables the modification of the *Abstract Syntax Tree* without requiring extensive modifications to the implementation of the *Abstract Syntax Tree* itself.

5.1 Lexer

The *Lexical Analyzer* processes the source code as a stream of characters. Once it identifies a valid lexeme, it emits the corresponding token. Specifically, the *Lexical Analyzer* follows the *Maximal Munch* rule, recognizing the longest verified lexeme, ensuring that tokens represent the most appropriate grouping of characters. In fine, the *Lexical Analyzer* transforms a stream of characters into a stream of tokens.

Symbol	Token	Regular Expression
”fn”	FUNCTION	`"fn"`
”true”,	T_TRUE	`"true"`
”false”	T_FALSE	`"false"`
”return”	RETURN	`"return"`
”if”	IF_TOKEN	`"if"`
”else”	ELSE_TOKEN	`"else"`
”let”	KW_VAR	`"let"`
”mut”	KW_MUT	`"mut"`
”&int”	TYPE_REF	`"&int"`
”&bool”	TYPE_REF	`"&bool"`
”&float”	TYPE_REF	`"&float"`
”&char”	TYPE_REF	`"&char"`
”int”	TYPE	`"int"`
”bool”	TYPE	`"bool"`
”float”	TYPE	`"float"`
”char”	TYPE	`"char"`
”&”	KW_REF	`"&"`
”&mut”	KW_MUT_REF	`"&mut"`
”==”, ”!=”	EQ, NEQ	`"=="`, `"!="`
”<”, ”>”	LT, GT	`"<"`, `">"`
”*”, ”/”	MUL, DIV	`"*"`, `"/"`
”+”, ”-”	PLUS, MINUS	`"+"`, `"-"`
”%”	MOD	`"\%"`
Identifiers	IDENTIFIER	`[a-zA-Z][a-zA-Z0-9]*`
Character literals	TOKEN_CHAR	`[a-zA-Z]`
Floating point numbers	TOKEN_FLOAT	`[0-9]+\.[0-9]+`
Integer literals	TOKEN_INT	`[0-9]+`
”{” ”}”	LBRACE, RBRACE	`"{" "}"`
”;”	END_OF_LINE	`[;]`
”:”	COLON	`":"`
”,”	COMMA	`[,]`
”(” ”)” ”=”	LPAREN, RPAREN, ASSIGN	`"(" ")" "="`
tabs, spaces, and newlines	(Ignored)	`[ \\t\\n\\r]`
Anything else	Throws error: ”Unknown Token”	`[.]`

Figure 2: The *Lexical Analyzer* recognizes lexemes by using regular expressions. The table shows which lexemes are mapped to which tokens and which regular expression is used to identify them.

5.1.1 Flex

Flex is a tool for generating *lexers* or *scanners* and is often paired with *Bison* to produce *Parser*s and interpreters. It’s an open-source project and is highly valued for its speed and efficiency in text processing.

The use of *Flex* for the *Rusticque* project streamlines the process of lexical analysis and makes it simple to create or alter the definition of lexemes. Its compatibility with *Bison* ensures seamless integration between the *Lexical Analyzer* and the *Parser*. Just as with Bison, the choice of *Flex* is reinforced by its comprehensive documentation, active community support, and its proven track record in various software projects. This makes it a reliable and robust choice for the lexical analysis phase of the *Rusticque* compiler.

5.2 Parser

The *Rusticque Parser* utilizes a *bottom-up* parsing methodology. Specifically, it is a *GLR*[4] parser that is capable of parsing all deterministic context-free grammars (CFGs).
Given a grammar, the *Parser* synthesizes a *parse table* that is used to determine what action to take based on the current *state*[5] and the next token in the token stream.
The *Parser* can take one of four actions: *shift*, *reduce*, *accept*, or *error*.

- **Shift** - When the *Parser* reads a token that is not the last token in a production rule, it shifts. This means it places the current token on a stack, reads the next symbol, and transitions to the state specified by the parse table.
- **Reduce** - When the *Parser* reads a token that is the last token in a production rule, it reduces. This involves popping the symbols on the stack that correspond to the production rule, pushing the non-terminal symbol specified by the production rule onto the stack, and transitioning to the state specified by the parse table.
- **Accept** - The *Parser* enters this state when it has successfully parsed the entire token stream.
- **Error** - This state is entered when the *Parser* encounters an unexpected token that is not valid in any state reachable from the current state.

This means the *Parser* effectively becomes an automaton, cycling through states shifting and reducing until it either accepts or errors. In this process it builds an *abstract syntax tree* (AST) that represents the structure of the program.

5.2.1 Bison

Bison is a prominent parser generator, part of the GNU project, capable of producing parsers for a variety of languages by supporting multiple parsing algorithms[3].
For the *Rusticque* compiler, *Bison* was employed because of its flexibility, efficiency, and user-friendliness. It facilitates rapid iterations and adjustments to the grammar. Moreover, its robust and modular parser implementation can be easily extended. Notably, *Bison* integrates natively with *Flex*, ensuring a cohesive interaction between the *Lexical Analyzer* and *Parser*.
When configured appropriately, *Bison* can generate a *GLR* parser using LALR(1), IELR(1), or CLR(1) parsing tables. Its comprehensive documentation and active community support further cemented its selection as the parser generator of choice for the *Rusticque* compiler.

[4]Look-Ahead LR Parser with 1 symbol of lookahead
[5]The tokens read so far

5.3 Static Code Analyser

Following the syntactical validation by the *Lexical Analyzer* and *Parser*, which results in the formation of an *Abstract Syntax Tree*, the next critical phase involves the *Static Code Analyzer*. This component is tasked with conducting a semantic analysis of the *Abstract Syntax Tree*, a process essential for ensuring the semantic correctness of the program.

The *Type Checker* is responsible for scrutinizing the use of types within the program. It systematically verifies that each type is correctly declared and applied, ensuring type consistency across the program. This step is vital in preventing type-related errors and maintaining the integrity of data handling within the program.

On the other hand, the *Borrow Checker* focuses on memory safety. It analyses the borrowing and lifespan of variables, ensuring compliance with stringent rules[6] for memory, and concurrency safety[1]. The *Borrow Checker* plays a crucial role in avoiding common pitfalls related to managing memory, such as memory leaks, unauthorized access and dangling pointers, thereby enhancing the program's reliability and safety.

In tandem, these two stages of semantic analysis form a comprehensive approach to validating the program beyond its syntax, ensuring that the logic is sound and the memory usage is safe. The following sections provide a detailed exploration of the *Type Checker* and *Borrow Checker*.

5.3.1 Type Checker

The *Type Checker*

[6]These are the rules also used by Rust

5.4 Borrow Checker

5.5 LLVM and Visitor Pattern

The *Code Generator* component of the compiler generates LLVM IR and and exetable binary from the AST. It implements the visitor pattern to traverse the AST in an efficient and extensible manner.

References

[1] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2023. URL: `https://doc.rust-lang.org/book/` (visited on Nov. 13, 2023).

[2] Unknown. *2023 Developer Survey*. 2023. URL: `https://survey.stackoverflow.co/2023/#overview` (visited on Sept. 20, 2023).

[3] Unknown. *GNU Bison*. 2023. URL: `https://www.gnu.org/software/bison/` (visited on Sept. 20, 2023).

Appendix

A Language Syntax

NOTE: things as expressions?

A.1 Variables

let mut Identifier: type = value

Identifier ::= StartChar SubsequentChars*
StartChar ::= [a-zA-Z_]
SubsequentChars::= [a-zA-Z0-9_]

type: [int, float, char, bool]

A.2 Expressions

A.3 Conditionals

IfExpression ::= "if" expression Block ?ElseExpression?
ElseExpression ::= "else" (Block — IfExpression)
let a:int = if(bool) return 5
fn fun():void
let a:int = 1
if(!test()) return
do something else
let a = fun
else if block
else block

A.4 Functions

FunctionExpr ::= "fn" FunctionName ParamList ":" type Block
FunctionName ::= Identifier ParamList ::= "(" Param ("," Param)* ")"
Param ::= Identifier ":" Type
Block ::= "" Statement* ""
Statement ::= /* definition of what constitutes a statement in your language /
Type ::= / definition of what constitutes a type in your language /
Identifier ::= StartChar SubsequentChars*
StartChar ::= [a-zA-Z_]
SubsequentChars::= [a-zA-Z0-9_]

fn func(param:bool):int code

fn test(param_ fun:(name:int, var:int)- int):(int,int)- int
param_ fun(1,1)
return param_ fun

let a:(int,int)- int = test(func)

A.5 Types

int: 32-bit integer
float: 64-bit floating point
char: 8-bit integer
bool: 8-bit
void