BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

HIGH-LEVEL PROGRAMMING LANGUAGE TRANSPILERS

HIGH-LEVEL PROGRAMMING LANGUAGE TRANSPILERS

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **ANDREJ MOKRIŠ**
AUTOR PRÁCE

SUPERVISOR **prof. RNDr. ALEXANDER MEDUNA, CSc.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

153866

Institut:	Department of Information Systems (DIFS)
Student:	**Mokriš Andrej**
Programme:	Information Technology
Title:	**High-Level Programming Language Transpilers**
Category:	Compiler Construction
Academic year:	2023/24

Assignment:

1. Based on the advisor's guidance, study source-to-source compilers, which make translations betweeen high-level programming languages.
2. Based on the advisor's instructions, study intermediate codes used by these compilers.
3. Based upon the information obtained in parts 1 and 2, design a new transpiler that make translations between selected high-level programming languages.
4. Implement the designed compiler. Demonstrate its functionality by translating the code between selected languages, for example a translation from C++ into JavaScript.
5. Evaluate the results achieved and discuss potential future work on the project.

Literature:

- Meduna, A.: *Elements of Compiler Design*. Auerbach Publications; 1st edition (December 3, 2007).

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Meduna Alexandr, prof. RNDr., CSc.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	30.10.2023

Abstract

A rapid development of programming languages has caused practical problems, such as not granting backward compatibility. Transpilers offer a potential solution to some of these problems. This thesis presents concepts of formal languages, compilers, and more detailed description of parsing methods. The goal of this thesis is to design and implement a transpiler that converts a subset of PHP to JavaScript.

Abstrakt

Rýchly vývoj programovacích jazykov priniesol radu praktických problémov, napríklad obmedzenú spätnú kompatibilitu. Časť týchto problémov môže byť riešená transpilátormi. Táto práca predstavuje základné koncepty z teórie formálnych jazykov, prekladačov a prístupov k syntaktickej analýze. Cieľom práce je navrhnúť a implementovať transpilátor, ktorý prekladá podmnožinu jazyka PHP do jazyka JavaScript.

Keywords

Transpiler, Compiler, Syntax analysis, Static analysis, Abstract syntax tree, Recursive-descent parser, JavaScript, PHP

Klíčová slova

Transpilátor, Kompilátor, Syntaktická analýza, Statická analýza, Abstraktný syntaktický strom, Rekurzívny zostup, JavaScript, PHP

Reference

MOKRIŠ, Andrej. *High-Level Programming Language Transpilers.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. RNDr. Alexander Meduna, CSc.

Rozšířený abstrakt

Neustály vývoj programovacích jazykov priniesol radu praktických problémov, ako napríklad obmedzená spätná kompatibilita. Časť týchto problémov môže byť riešená automatickým prekladom do iného jazyka pomocou transpilátoru. Transpilátor prekladá zdrojový kód z jedného jazyka do druhého, no na rozdiel od tradičného prekladača sú dané jazyky na podobnej úrovni abstrakcie. Praktickým príkladom transpilátoru môžu byť Babel, ktorý prekladá JavaScript verzie ES6 do verzie ES5, zabezpečujúc tak spätnú kompatibilitu so staršími verziami webových prehliadačov, či 2to3, ktorý prekladal kód jazyka Python verzie 2 do verzie 3.

Hlavným cieľom práce je návrh, implementácia a testovanie transpilátoru z podmnožiny jazyka PHP do jazyka JavaScript. V úvode sa práca zaoberá motiváciu pre vývoj transpilátorov a popisuje štruktúru práce. Druhá kapitola definuje pojmy z teórie formálnych jazykov ako napríklad jazyk, abeceda, konečný automat, či bezkontextová gramatika. Sú tu pojmy, ktoré sú využívané pri výstavbe prekladačov aj v tejto práci.

Následne sa práca zaoberá teóriou prekladačov, kde sú popísané jednotlivé fázy prekladu a vzťahy medzi nimi. Práca popisuje základný princíp činnosti jednotlivých fáz prekladu, ako lexikálna či syntaktická analýza, až po generovanie cieľového kódu. Taktiež opisuje rozdiel medzi tradičným prekladačom a transpilátorom. Dva základné princípy syntaktickej analýze - zhora nadol (top-down) a z dola nahor (bottom-up) sú vysvetlené v kapitole 4, ktorá detailnejšie približuje a fungovanie, výhody a nevýhody využitia daných prístupov pri výstavbe prekladača. Dané prístupy sú následne využité aj pri realizácii praktickej časti práce.

Kapitola 5 obsahuje popis jazyka PHP, ktorý je zdrojový jazyk v rámci implementácie transpilátoru a jazyka JavaScript, ktorý je vybraný ako cieľový jazyk. Kapitola obsahuje popis podporovanej podmnožiny cieľového jazyka a vybrané syntaktické a sémantické rozdiely, ktoré ovplyvňujú výsledný návrh a implementáciu.

Práca opisuje proces návrhu transpilátora s dôrazom na možnosť budúcej rozšíriteľnosti ako podporovanej podmnožiny jazyka, tak vstavaných funkcií. Implementácia nevyužíva žiadne pomocné knižnice alebo nástroje na generovanie lexikálne alebo syntaktickej analýzy. Syntaktická analýza využíva kombináciu dvoch prístupov - rekurzívneho zostupu, teda metódy zhora-nadol, ktorá spracováva štruktúru programu ako podmienky či cykly a metódu precedenčnej syntaktickej analýzy, ktorá spracováva výrazy a pomocou precedenčnej tabuľky zabezpečuje vyhodnocovanie operácií v správnom poradí. Sémantická analýza overuje kompatibilitu typov pri operáciách, či existenciu premenných a funkcií. Zo zostaveného a analyzovaného stromu je následne vygenerovaný kód v cieľovom jazyku.

Výsledná implementácia podporuje základné štruktúry ako podmienky, či rôzne typy cyklov. Taktiež podporuje volanie a definovanie vlastných funkcií a malú podmnožinu vstavaných funkcií zdrojového jazyka primárne pre prácu s reťazcami, poľami, či matematické operácie, a vygenerovaný kód je funkčne ekvivalentný so zdrojovým. Funkčnosť riešenie je otestovaná na vzorke PHP programov, ktorých výstup je porovnaný s výstupom preloženého kódu. Funkčnosť je taktiež porovnaná s existujúcimi riešeniami poskytujúcimi preklad z PHP do jazyka JavaScript.

High-Level Programming Language Transpilers

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of prof. RNDr. Alexander Meduna CSc. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Andrej Mokriš
May 7, 2024

Acknowledgements

I would like to thank my supervisor, Prof. RNDr. Alexander Meduna, CSc., for his guidance and prompt responses. I would also like to express my gratitude to my family and my girlfriend for their support.

Contents

List of Figures

Chapter 1

Introduction

The process of rewriting the codebase from one programming language to another is very resource-intensive and error-prone. Transpilers, also known as source-to-source compilers, help automate parts of the process and make it more efficient. Languages that were a popular choice a while ago may have been replaced by newer alternatives that excel in many key factors, such as performance, speed of development, or popularity in the community. As programming languages are constantly being improved to support new features, they may no longer maintain backward compatibility with older platforms or runtimes, therefore programmers are limited to the subset of a language they can use. This practical problem can be solved using a transpiler. An example may be Babel, which transpiles the ES6+ version of JavaScript to older versions supported by legacy JavaScript environments, such as older browsers.

The principle and structure of transpilers is similar to that of traditional compilers. It consists of a sequence of phases described in 3.4, where each phase has its function in the chain. In comparison with traditional compilers, such as GCC, transpilers generate code at a similar level of abstraction, usually from a source high-level programming language to another high-level language, producing functionally equivalent source code.

The goal of this thesis is to design and implement a transpiler from PHP to JavaScript, which supports a subset of the source language and generates code in the target language so that programs are functionally equivalent. PHP and JavaScript are both dynamically typed scripting programming languages that are traditionally used in web development, with PHP running on the server and JavaScript on the client side. With the popularization of using JavaScript on the server side, mainly due to the arrival of the V8 JavaScript engine, it has proven beneficial to use the same language on the frontend as well as on the backend. Furthermore, JavaScript is the most popular language according to developer community surveys, such as Stack overflow developer survey 2023[1], and many codebases still written in PHP, with the big enough supported subset of the language, the tool may find a practical usage.

In chapter 2, the foundations of the theory of formal language that are referred to throughout the thesis are defined. That includes concepts such as language 2.1.3, grammar 2.3.1, or finite automata 2.2.1, providing their formal definitions and practical usage in the context of compiler development.

[1] Developer survey 2023 : https://survey.stackoverflow.co/2023

After the necessary theoretical introduction, the chapter 3 explains basic ideas of compiling, presents differences between different types of language processors, and describes individual phases of the compilation process.

The chapter 4 provides a more detailed explanation of the syntax analysis and discusses two main approaches to syntax analysis; top-down 4.1 and bottom-up 4.2 parsing, their principles and limitations as well as construction of the abstract syntax tree.

The description of selected source and target programming languages is in the chapter 5. It provides a general description of languages, their usual use cases, and semantic differences, which influence the process of design and implementation of the transpiler.

The practical segment of the thesis, detailed in chapter 6 provides insight into design considerations before the implementation, the implementation process in the Python programming language.

Testing of the implementation and comparison of its capabilities with existing tools are described in chapter 7.

Chapter 2

Introduction to formal languages

Building a compiler requires a certain level of knowledge about formal languages. This chapter explains formal languages and mathematics tools that will be referred to throughout the thesis, such as alphabet 2.1.1, language 2.1.3, finite automata 2.2.1, or a context-free grammar 2.3.1. In this chapter, the information presented are adopted from [7], [1]. Formal definitions are adopted from [8].

2.1 Alphabets and languages

Definition 2.1.1 *An alphabet Σ is a finite, nonempty set of elements, which are called symbols.*

Definition 2.1.2 *Let Σ be an alphabet. ϵ denotes the empty string over Σ. If x is a string over Σ and $a \in \Sigma$, then xa is a string over Σ.*

Definition 2.1.3 *Let Σ^* denote the set of all strings over Σ. Every subset $L \subseteq \Sigma^*$ is a language over Σ.*

2.2 Finite automata

Finite automata are fundamental models of lexical analysis. Their task is to accept or reject an input. A finite automaton accepts an input string x if and only if there exists a path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x.

Definition 2.2.1 *A finite automaton (FA) is a 5-tuple:*

$$M = (Q, \Sigma, R, s, F)$$

where:

- *Q is a finite set of states.*
- *Σ is an input alphabet.*
- *R is a finite set of rules of the form: $pa \rightarrow q$, where $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$.*
- *$s \in Q$ is the start state.*

- *$F \subseteq Q$ is a set of final states.*

The general variants represent mathematically convenient models, which are difficult to apply in practice. In general, these automata work non-deterministically, meaning that from the current configuration of the automata, there are several possible moves with the same input symbol. Such behavior makes the implementation of the lexical analyzer more difficult. For those reasons, there are more restricted models, such as *ϵ-free automata* or *deterministic finite state automata.*

Definition 2.2.2 *Let $M = (Q, \Sigma, R, s, F)$ be an finite automaton. M is an ϵ-free finite automaton if for all rules $pa \to q \in R$, where $p, q \in Q$, holds $a \in \Sigma$ $(a \neq \epsilon)$.*

Definition 2.2.3 *Let $M = (Q, \Sigma, R, s, F)$ be an ϵ-free finite automaton. M is a deterministic finite automaton (DFA) if for each rule $pa \to q \in R$ it holds that $R - \{pa \to q\}$ contains no rule with the left-hand side equal to pa.*

All these variants are capable of recognizing the same languages, called the *regular languages*, so we can always use any of them without any loss of generality. A finite automaton can be represented either tabularly by a state table or graphically using a state diagram.

2.3 Grammar

A grammar is based upon finitely many rules, which contain terminal and nonterminal symbols. Terminals represent tokens and nonterminal symbols formalize more general syntactical entities, such as loops. A grammar derives string by beginning with the start symbol and repeatedly replacing a nonterminal with the body of the production for that nonterminal. If we find the derivation of the program, it is a syntactically well-formed program.

Grammars used in the field of formal languages and compilers are often called a *context-free grammar* 2.3.1 to point out that during any derivation step, a nonterminal is rewritten regardless of the surrounding context.

Definition 2.3.1 *A context-free grammar (CFG) is a quadruple*

$$G = (N, T, P, S)$$

where:

- *N is an alphabet of nonterminals (sometimes called „syntactic variables“).*
- *T is an alphabet of terminals (referred to as „tokens“), $N \cap T = \emptyset$.*
- *P is a finite set of rules of the form $A \to x$, where $A \in N$, $x \in (N \cup T)^*$.*
- *$S \in N$ is the start nonterminal.*

By convention, terminal symbols are written by lowercase letters, such as a, b, c, and nonterminals by uppercase letters A, B, C, D. As an example of a rewriting rule, consider string xAy and a rule $r : A \to u \in R$. By using this rule, G makes a derivation step from xAy to xuy by changing A to u, written as $xAy \to xuy$.

A language generated by the context-free grammar 2.3.1 is called a **context-free language** (CFL).

Definition 2.3.2 *Let L be a language. L is a context-free language (CFL) if there exists a context-free grammar that generates L.*

A **parse tree** is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. A grammar that produces more than one parse tree for some sentence is said to be **ambiguous**. Similar to non-deterministic finite automata, it causes practical problems in the process of implementation of the compiler, as the program has to determine, which parse tree is the correct one to continue with.

During the parsing, nonterminal symbols are rewritten by the body of the production. In the case of top-down parsing, it finds the **left-most derivation** of the program, meaning that during the derivation step, the leftmost nonterminal is rewritten.

Definition 2.3.3 *Let $G = (N, T, P, S)$ be a CFG, let $u \in T^*$, $v \in (N \cup T)^*$. Let $p = A \rightarrow x \in P$ be a rule. Then, uAv directly derives uxv in the leftmost way according to p in G, written as $uAv \Rightarrow_{lm} uxv[p]$.*

The construction of a top-down parser is aided by a construction of sets **First** and **Follow**, as they allow to choose the correct production rule based on the next input symbol. $First(x)$ is the set of all terminals that appear as the first symbol of a string derivable from x.

Definition 2.3.4 *Let $G = (N, T, P, S)$ be a CFG. For every $x \in (N \cup T)^*$, we define the set First(x) as First$(x) = \{a : a \in T, x \Rightarrow^* ay; y \in (N \cup T)^*\}$.*

$Follow(A)$, for non-terminal A is a set of all terminals, that can appear immediately to the right of that non-terminal in some sentential form derivable from the start symbol S.

Definition 2.3.5 *Let $G = (N, T, P, S)$ be a CFG. For every $A \in N$, we define the set Follow(A) as Follow$(A) = \{a : a \in T, S \Rightarrow^* xAay, x, y \in (N \cup T)^*\} \cup \{\$: S \Rightarrow^* xA, x \in (N \cup T)^*\}$.*

Chapter 3

Compilers

This chapter explains the terms which are relevant in the context of solving the thesis. It will explain the differences between compilers and transpilers and the steps in the process of transpiling. This chapter uses information from [7],[1], and [5].

3.1 Compiler

A compiler is a computer program, which reads a source program in a source language and translates it into a target program in a target language. A compiler is primarily used for translating between languages at different levels of abstraction, usually from a high-level programming language (C++, Java) to a low-level programming language (assembly language, or machine code), which is specific to the architecture of the platform. The source program and the target program are functionally equivalent. During the translation, the compiler first analyzes the source program to verify that the source program is correctly written in the source language. If so, the compiler generates the target program; otherwise, report the error. The steps in the compiling process are explained in chapter 3.4.

3.2 Transpiler

A transpiler, also known as a source-to-source compiler, is a special type of compiler that reads a source program in a source language and generates a functionally equivalent program in a different, in some cases even the same programming language. In comparison with the compiler 3.1, both source and target languages are at a similar level of abstraction. Transpilers are used to address performance, developer experience, or compatibility problems. The problem with transpilers is, that different programming languages have many semantic differences, that may not be obvious at first sight, which leads to a very complex development process and testing. As an example of usage of transpilers, TypeScript[1], which is a superset of JavaScript with static typing and has to be transpiled back to JavaScript to be executed, or Babel[2], which transpiles JavaScript to older versions to grant backward compatibility with older browsers.

[1] TypeScript: https://www.typescriptlang.org/

[2] Babel: https://babeljs.io/

3.3 Interpret

The last common type of language processor is called an *interpret*. Instead of translating the source program into the target program, it directly executes the source program, without requiring it to have been processed by a different program.

3.4 The structure of a compiler

The process of compilation is a complex problem, thus the process is segmented into distinct phases. Each phase operates with specific inputs and outputs, where the output of the previous layer is used as input for the next step in the chain.

The structure of the compiler is split into two main categories - the front end and the back end. The front end analyzes the source program and builds the intermediate representation of the source program, usually in the form of the three-address code or the abstract syntax tree. The front end consists of lexical analysis 3.4.1, syntax analysis 3.4.2, and semantic analysis 3.4.3. The role of the back end is target code generation. Such separation of concerns ensures the benefit of modularity of compilers.

In addition to phases, compilers use symbol tables, which are data structures used to hold information about the constructs of the source program. The entries in the table are collected during the analysis phases of a compiler and contain information about identifiers, such as name, position, and any relevant information [1].

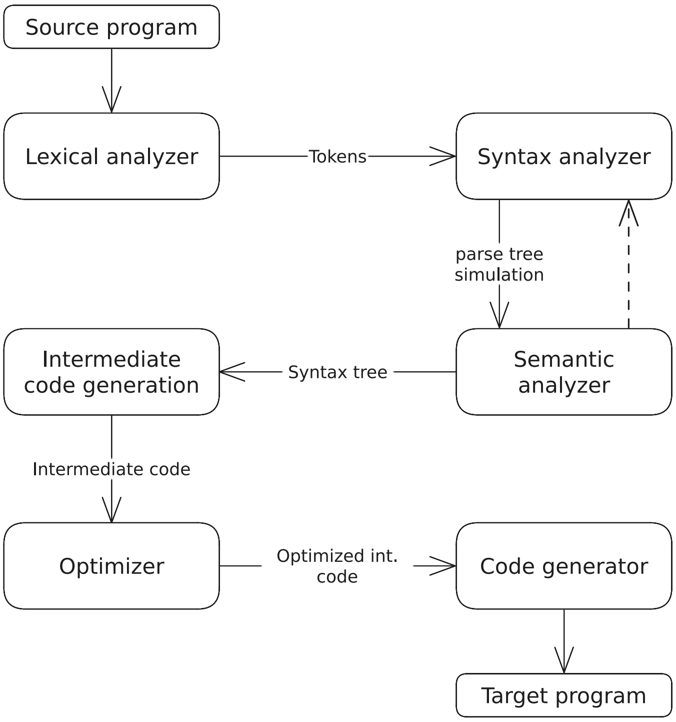

Figure 3.1: Compiler phases diagram

3.4.1 Lexical analysis

The initial phase of a compiler is called *lexical analysis* or *scanning*. In this phase, the lexer reads the stream of characters that make up the source program and groups the characters into meaningful sequences called *lexemes*.

The main task of lexical analysis is the recognition and classification of lexemes and their representation of them by their tokens. Its task is also to remove parts of the source code that are not necessary for compiling, such as comments or whitespaces. For each lexeme, the lexical analyzer produces a token of the form

<token-name, attribute-value>

that it passes on to the next phase, syntax analysis. The *token-name* attribute possesses the information about the type of the token such as a variable name or a floating-point number. This information is important for the process of syntax analysis. The *attribute-value* points to an entry in the symbol table for this token. During the lexical analysis, new entries are inserted into the symbol table. Information from the symbol table is used primarily during the semantic analysis phase.

The lexical analyzer can be expressed using a *deterministic finite state machine* or a set of *regular expressions*:

- In the case of finite state machines, transitions between states are determined by the incoming character and the current state. If the state is declared as a *terminal state*, and can not accept another character, a new token is returned, and the state machine returns to the initial state. If there is no possible transition from the current configuration with the incoming character, the lexer will report a lexical error.

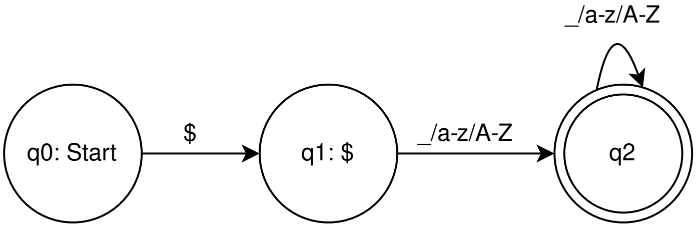

Figure 3.2: Finite state machine accepting a PHP variable name

- Lexems can be described using *regular expressions*, which use three operations: *concatenation*, *union*, and *iteration*.

In practice, lexical analyzers are built using lexical analyzer generators, such as **Lex** [6]. It uses user-defined regular expressions to describe patterns for tokens using Lex language. The Lex compiler translates the source code into a program in the C programming language, which can be used as a lexical analyzer module within the compiler.

3.4.2 Syntax analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The *syntax* of a programming language describes the proper form of its programs. To specify syntax, the forms used are called *context-free grammars* or *BNF* (Backus Naur form). Grammar consists of finitely many rules that are used for derivation. An example of rules may be:

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \mathit{expr}; \\
 & | & \texttt{if (} \;\; \mathit{expr} \;\texttt{)}\; \mathit{stmt}_1 \\
 & | & \mathit{assignment} \\
 & & \\
\mathit{assignment} & \rightarrow & \mathit{identifier} = \mathit{expr}
\end{array}
$$

The syntax analyzer uses tokens generated by the lexical analyzer to verify that the string of tokens represents a syntactically well-formed program and is in compliance with the grammar of the source language. Incoming tokens are compared with the expected values, according to grammar rules. When the program is syntactically correct, the parser generates a structured representation of the token stream, referred to as a *parse tree*, which is used during subsequent phases of the translation. A parse tree is a graphical representation of derivation steps that show the order in which productions are applied to replace nonterminals. Otherwise, the parser will report a syntax error.

There are two main approaches to parsing; *top-down* and *bottom-up*, which will be discussed further in chapter 4.

Syntax analyzer closely cooperates with the semantic analysis and drives the entire process of translation - referred to as *syntax-directed translation.* The syntax-directed translation is done by attaching rules or program fragments to productions in grammar, which are called *semantic actions.* The result of syntax-directed translation is the intermediate representation of the program, usually a *abstract syntax tree.* It is derived from the parse tree, however, it abstracts from any information, that is not essential for the compilation. It may also perform various checks of the semantic analysis, such as type checking.

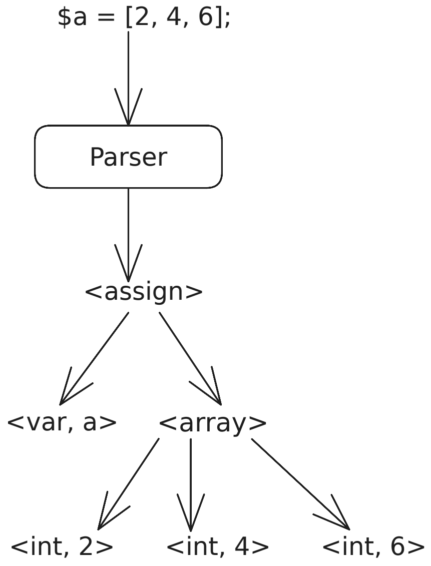

Figure 3.3: Parsing of the expression to the derivation tree

3.4.3 Semantic analysis

The *semantic analyzer* uses the parse tree, which is generated by the syntax analyzer, and the information in the symbol table to check the program for semantic correctness. This process detects errors that were not discovered during the lexical and syntax analysis, as they cannot be defined formally using regular expressions or grammar. The semantic analyzer checks the compatibility of types; for example, operation addition (+) is not allowed between the integer and string types.

Using the symbol table verifies the existence of variables or functions used in the program. It handles implicit type conversions. An example may be the operation multiplication (*) between an integer and a floating-point number. In this case, the operator of type *integer* will be converted to a floating-point number, without the programmer having to explicitly make it so, as on a lower level, computers use different instructions to work with integers and floating-point numbers. The result of the semantic analysis is an *abstract syntax tree.*

3.4.4 Intermediate code generation

After the analysis phase of the compilation, the compiler generates an *intermediate code.* It is the internal version of the target program, which is similar to the machine code, also called the *three-address code.* In a three-address code, there is at most one operator on the right side of an instruction, so it does not allow for built-up arithmetic expressions. This intermediate representation should have two important properties: It should be easy to produce and easy to translate into the target machine.

The intermediate code serves as a bridge between the front end and the back end of a compiler, where the front end analyzes a source program and creates an intermediate representation from which the back end generates the target code. The intermediate code is also used during the optimization phase of the compiling, which will be discussed in chapter 3.4.5.

3.4.5 Optimization

Optimization is an optional phase in the compilation process. The syntax-direct translation produces intermediate code that can be made more efficient so that it runs faster or takes less space. The code is improved by various transformations, such as the removal of unnecessary instructions or their replacement. However, the optimized program and the original program have to be functionally equivalent.

The optimizer first breaks the program into basic blocks that are made up of sequences of three-address instructions that are executed sequentially. Then it analyzes the use of variables within these blocks and between them.

Optimization can be split into two different types - *machine dependent* and *machine independent* optimization.

- **Machine independent** - transforms the intermediate code independently of the target machine.
- **Machine dependent** - optimization for a specific type of hardware and architecture. Focuses on effective register allocation so it results into a reduction of the number of moves between the registers and the memory.

```
t1 = int2float(60)
t2 = id2 * t1                  t1 = id3 * 60.0
t3 = id2 + t2      ------>     id = id2 + t1
id = t3
```

Figure 3.4: An example of the optimization of the three-address code

3.4.6 Code generation

The code generation phase is the last and it translates the intermediate representation of a source program, which can either be an abstract syntax tree or a three-address code, to a functionally equivalent target program and, thereby, completes the compilation process.

The code generation process is highly dependent on the target language. In the case of traditional compilers, it is a low-level machine code, which means that the intermediate representation of the program is translated into sequences of machine instructions. The generated code is built for a specific type of machine. Transpilers, on the other hand, usually generate code at a level of abstraction similar to that of the source code, which means that in the case of high-level language transpilers, there is no need for machine-dependent code generation.

3.5 The structure of a transpiler

Transpiler is simply a special type of traditional compiler that produces functionally equivalent source code in different programming languages at a similar level of abstraction, which means that the principle and structure are almost identical to that of a traditional compiler, described in 3.4, which generates code in a lower level language. Phases, which belong to the front end of a transpiler, work identically to the transpiler, as the source code has to be split into tokens and parsed to construct an intermediate code representation, such as an abstract syntax tree. Due to potentially different semantics of the source and target programming languages, the abstract syntax tree is statically analyzed and transformed to match the structure of the program in the target language, for example moving function definitions to the start of the program in case of languages, that do not allow call before definition, or moving variable declarations to match scopes in the target language. The transformed tree is then used for the generation of the source code in the target language. In addition to transpiling the source code, the transpiler needs to ensure support for the built-in functions of the source language for the tool to be practically usable.

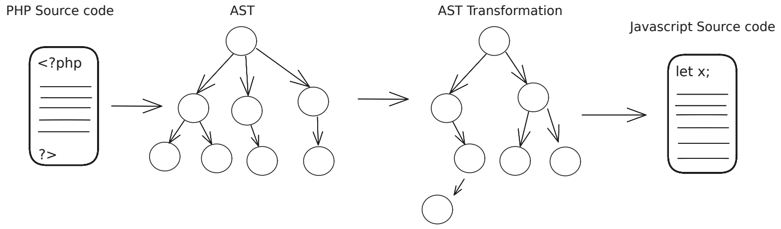

Figure 3.5: Simplified principle of the transpiler

Chapter 4

Syntax analysis approaches

Syntax analysis 3.4.2 is the key part of every compiler. This chapter explains two common approaches to syntax analysis: top-down parsing and bottom-up parsing. It will show its strengths and limitations. Informations are adopted from [7], [1], and [11].

4.1 Top-down parsing

The top-down parser constructs its parse tree starting from the root of the tree and proceeding down towards the leaves of the tree, which represent terminal symbols. It reads the token stream from left to right and tries to find the leftmost derivation 2.3.3 of the input stream, meaning that at each step the leftmost nonterminal symbol is replaced by its production rule. Top-down parsers are based on the **LL(k) grammars**, where the first *L* stands from left-to-right scan and the second *L* stands from left-most derivation. The *k* indicates how many lookahead tokens it uses during parsing. However, in practice, LL(1) grammar is preferred, as it simplifies the construction of the parser.

At each step of the parsing process, the key problem is, which production to pick. When the production is chosen, the role of the parser is to verify that the incoming token matches with the expected terminal symbols in the body of the production. When the incoming token does not match the expected token, the parser reports a syntax error. As an example, the rule which stands from **if-statement**:

$$\text{statement} \rightarrow \textit{if}\ (\,\text{expression}\,)\,\text{statementList}$$

has been chosen. After the `if` keyword, the parser expects left parentheses, otherwise reports a syntax error due to an unexpected token. When the incorrect production has been chosen, the parser backtracks to find an alternative production, which will match the incoming token stream. Such an approach makes the practical implementation more complex; however, it may be solved by **predictive parsing**.

The predictive parser relies on the parsing table, which is a two-dimensional array. The parsing table is constructed using sets FIRST(X) 2.3.4 and FOLLOW(X) 2.3.5. The LL(1) parse table is used to determine which rule should be applied for any combination of the nonterminal symbol on the stack and the next token on the input stream. Each table entry identifies which production to pick and continues the derivation using the production, otherwise reports an error. By definition, an LL(1) grammar has exactly one rule to be applied for each combination[11].

4.1.1 Recursive descent parsing

Recursive descent parsing is a top-down method of syntax analysis. It is a system of mutually recursive procedures that parse the input program. For every nonterminal symbol in the grammar, there typically exists a specific method, usually a Boolean function, that determines whether the incoming token stream matches the expected program structure defined by the grammar, comparing expected and the actual value of terminals and calling corresponding procedures for expected nonterminals. The parsing algorithm starts with a procedure corresponding to the start symbol of the grammar, which chooses the next step in the derivation process based on the look-ahead symbol.

It runs until a complete parse tree is constructed, meaning the procedure corresponding to the start symbols successfully finishes, otherwise reports an error. This method is relatively simple to understand, as the implementation copies the grammar.

```
def IfRule() -> bool:
    if match("if") and ExpressionRule() and match(":") and StatementRule():
        return True
    else:
        report("syntax error")
        return False
```

Listing 4.1: Nonterminal procedure matching if statement in Python

The recursive descent parser works with grammar that is not **left-recursive**. In context-free grammar, a production rule is said to be left-recursive if the leftmost symbol of the body is the same as the nonterminal at the head of the production.

$$expr \quad \rightarrow \quad expr + term$$

This creates a problem for recursive descent parsers, as it leads to infinite recursion when attempting to expand the non-terminal symbol.

4.2 Bottom-up parsing

In contrast with the top-down parsing approach, bottom-up builds a parse tree starting at the leaves and working up to the root. Each step in the derivation process represents a *shift* or a *reduction*. At each *reduction* step, a specific substring that matches the body of a production is replaced by the nonterminal at the head of that production [1]. Similar to top-down parsing, the key problem is which production to choose and whether to *shift* or *reduce*.

4.2.1 Operator-precedence parsing

Operator-precedence parsing is a popular deterministic bottom-up parsing method for expressions whose operators and their priorities control the parsing process, meaning operators with higher priorities are evaluated before lower-priority operators. It uses a stack, the bottom of which is marked by a convention with the `$` sign and a *precedence table*, which shows the corresponding action for every combination of the top terminal and the incoming symbol. At every step of the parsing process, the decision is made on whether an action of *shift* or a *reduction* should take place. The decision is based on the symbol in the precedence table.

- **Shift** - When the operation marks *shift* (usually marked with < symbol), the handle symbol is added to the right side of the topmost terminal and current symbol is pushed to the operator stack. In case of the = action, the current symbol is pushed to stack without inserting the handle.
- **Reduction** - When the operation marks *reduction* (usually marked with > symbol), the parser verifies, whether there exists any production with the body matching the top string at the pushdown. The end of the string at the stack is marked with a *handle* symbol. If such a rule exists, the string is replaced with the nonterminal in the head of the production. If there is not any rule matching production, the parser reports an error.
- **Error** - The empty entry in the precedence table for the configuration of the topmost terminal and an incoming character signals an error and ends the parsing process.

Algorithm 1: Operator-Precedence Parsing Algorithm [8]

```
Push($);
repeat
    Let $a$ be the topmost terminal on the stack;
    Let $b$ be the current token from the input string;
    switch Table[a][b] do
        case $=$ do
            Push($b$);
            NextToken();
        case $<$ do
            Replace $a$ with $a <$ on the stack;
            Push($b$);
            NextToken();
        case $>$ do
            if $< y$ is the top string on the stack and $A \rightarrow y \in P$ then
                Replace $< y$ with $A$ on the stack;
            end
            else
                Error;
            end
        case blank do
            if $a = \$$ and $b = \$$ then
                Success;
            end
            else
                Error;
            end
    end
until Success or Error;
```

In practice, the operator precedence parsers are frequently used in combination with other parsers, including predictive parsers 4.1.1. In such configuration, the precedence parser handles the syntax of expression, while the predictive parsers handle the general structure of programming constructs, such as conditionals and loops [7].

4.2.2 LR parsing

LR parsers are table-driven bottom-up shift-reduce parsers, where L stands for left-to-right scan of tokens and R for the rightmost derivation in reverse. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods. This means that the power of the LR parser is greater than that of other parsing methods and can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written [1]. It relies on the stack, which stores the states of the parser, whereas the other shift-reduce types of parsers use stack to shift symbols representing terminals and nonterminals. Each state summarizes the information contained in the stack below it. The LR table on which the parser is built consists of two parts:

- **Action part** - Determines which action the parser should take, mainly what the next state for the current configuration is, based on the current state and the next input symbol. Common actions include Shift, Reduce, Accept, and Error action.
- **Goto part** - After the parser reduces a portion of the stack based on a production rule, it consults the goto section to determine the next state to transition to based on the non-terminal symbol at the head of the production applied.

Algorithm 2: LR Parsing Algorithm [1]

```
Input  : Input string $w$ with end marker '$'
Output: Parse tree or error message
Let $a$ be the first symbol of $w\$$;
while true do
    Let $S$ be the state on top of the stack;
    if $ACTION[S, a] = shift\ t$ then
        Push $t$ onto the stack;
        Let $a$ be the next input symbol;
    else if $ACTION[S, a] = reduce\ A \rightarrow \alpha$ then
        Pop $|\alpha|$ symbols off the stack;
        Let state $t$ now be on top of the stack;
        Push GOTO[$t$, $A$] onto the stack;
        Output the production $A \rightarrow \alpha$;
    else if $ACTION[S, a] = accept$ then
        Break;
        // Success
    else
        Call error-recovery routine;
        // Handle syntax error
```

The main disadvantage of the LR parser is the difficulty of implementing the LR parser for the grammar of a programming language by hand, as it is possible in the case of the LL parsers. However, similar to a lexical analysis generator, there are tools for generating LR parser, such as Yacc or Bison, which read a description of a context-free grammar and generate a LALR(1) parser in the C programming language.

Chapter 5

Source and target languages

This chapter describes the PHP 5.1 and JavaScript 5.2 languages, which are used as the source language and the target language for the transpilation, respectively. It describes the syntax of source languages, which is important during the syntax analysis phases as well as comparing semantic differences, which may lead to functionally non-equivalent programs or difficulties in the process of target code generation. Information about languages is adapted from [10] and [2].

5.1 PHP

PHP (Hypertext Preprocessor) is a widely-used open-source general-purpose scripting language that is especially suited for web development and can be embedded into HTML. PHP is a dynamically typed language, which means that the data type of a variable is determined at runtime rather than at compile time. It is an imperative, procedural, and object-oriented language. PHP is commonly used as a server-side scripting language, where it is used in combination with HTML to create web pages with dynamic content based on various factors such as user input, or populating the webpage with data from the database. However, it can also be used as a traditional scripting language, which will be the focus of the selected subset of the target language, not its usability in the web environment.

5.1.1 Syntax

PHP program typically starts with the opening tag `<?php` and may end with the closing tag `?>`. The tags tell the PHP interpret, which part of the source code is code to be interpreted, the rest is ignored. The syntax of the language is similar to that of the C programming language, as it uses semicolons to terminate statements or curly braces to enclose code blocks. Variables are denoted by the `$` symbol followed by the name. It supports various data types such as integers, floats, strings, booleans, null, arrays, objects, and resources. The flow of the program is controlled by traditional structures such as conditions or loops such as `for`, `foreach`, or `while`. It supports arithmetic and logical operations, assignments, as well as function calls and definitions of custom functions.

5.1.2 Supported subset of the language

PHP is a versatile language with a wide range of built-in features and approaches to software design. The design and implementation of a transpiler to support an entire language that would produce functionally equivalent code would be very complex. Ignoring the object-oriented approach, the transpiler supports basic programming constructs such as assignment to a variable or an array, including assignment to a specific index in the string or an array. It supports most of the control structures of the language:

- **Conditional statement** - Decision, whether the code is executed based on the specified condition. Examples of such structures are `if`, `elseif`, or `else`. The ternary operator is also supported.
- **Looping constructs** - Allows iterating over data or executing the specified code repeatedly. Example of looping constructs are while, do-while, for, or foreach.
- **Control flow statements** - Additional built-in constructs to control the flow of the program including `switch-case`, `break`, `continue`, or a `return` statement.

Definition of custom functions and function calls are possible as well, including function calls within mathematical expressions. In addition to user-defined functions, a small subset of built-in functions is also supported:

- **Array functions** - Small subset of functions which allow manipulation and creation of arrays, such as `array()`, `count()`, `array_pop()`, `array_push()`, `explore()`, or `implode()`. Function `range()`, which creates an array containing a range of elements is supported as well.
- **String functions** - Functions that allow operations with string literals, such as `strlen()`, `strpos()`, or `substr()`.
- **Math functions** - Implements commonly used mathematics functions such as `sqrt()`, `floor()` or a `pow()`, which raises number to a certain power.
- **Type conversions** - Adds support for explicit type conversions such as `intval()`, `strval()`, and conversion of characters to and from their ASCII values; `chr()` and `ord()`.

Within expressions, in addition to traditional mathematics operations, it supports logical operations, comparisons, and working with array items at concrete indexes as well as ternary operators.

5.2 JavaScript

JavaScript is an interpreted scripting language for web pages, many non-browser environments also use it, such as Node.js[1], which is a JavaScript runtime environment. It is a single-threaded, dynamic language supporting object-oriented, imperative, and declarative styles [2]. The language is specified by the ECMA-262 standard [4]. This thesis assumes the usage of Node.js as a JavaScript runtime. As already mentioned in the introduction, JavaScript has been the most popular language in the developer community for many years. Due to its popularity, the language has a large environment of open-source libraries and tools, such as Bun[2], the performance of which drastically overcomes that of Node.js.

[1] Node.js: `https://nodejs.org/en`

[2] Bun: `https://bun.sh/`

5.2.1 Syntax and semantics

The syntax of JavaScript is not very different from that of PHP, as it is part of C-family programming languages, including code blocks delimited by curly braces (`{}`), or semicolon (`;`) terminating expressions. Variables are declared using `let`, `const`, or `var` keywords. Both `let` and `const` were introduced with the ES6 version of JavaScript and declare block-scoped local variables, where the value of `const` cannot be reassigned using the assignment operator. The `var` statement declares function-scoped or globally-scoped variables [2]. Similarly to PHP, a function can be declared using a `function` keyword, however, the ES6 version introduced the use of arrow functions using the `=>` syntax. It is a dynamically and weakly typed language, meaning that variables can change their type over time and allows for implicit type conversions when the operation involves mismatched types. The fundamental model of JavaScript is objects, including primitives such as Number, String, Boolean, Null, or Undefined are wrapped into their corresponding object wrappers [2], allowing the use of built-in functions specific to the object. Objects are created using a pair of curly braces (`{}`) and can store key-value pairs. Even when assigned to a variable declared using the `const` keyword, fields of objects can be mutated. Similar to PHP, it supports different types of loops or conditional statements. The JavaScript code can be organized into modules, where objects can be imported or exported using `import` and `export` keywords introduced in ES6. It supports an object-oriented programming approach, allowing the creation of classes, objects, or inheritance using the `extend` keyword. JavaScript offers asynchronous programming using `async/await` syntax or `promise`. It enables the non-blocking execution of time-consuming tasks such as database queries, or network request processing

5.2.2 Language differences

Languages are syntactically and semantically similar, however, certain differences influence the functionality of the generated code and require better semantic analysis in order to mitigate the differences in the generated code. The difference in operator priority is solved by the precedence-operator parser and using brackets to explicitly declare the priority of operators in the generated code. Strings in PHP are mutable, therefore, you can explicitly change the string, for example, assign a character to a certain index, which JavaScript does not allow, and requires a sequence of steps to be generated to simulate the same behavior. Implicit type conversions, such as using `echo`, which converts the expression to string, behave differently than using `console.log()`, which in addition adds a new line character after the output. The boolean value `true` is assigned to string `'1'`, therefore, the code generator has to simulate the behavior of PHP. Arrays in PHP can be indexed either traditionally starting from 0, or including key-value pairs, where each item in the array can be indexed using the corresponding key, and the behavior partly corresponds to that of Objects from JavaScript.

Chapter 6

Design, implementation, and testing

This chapter explains the design process and implementation of the transpiler. The program is implemented in Python programming language. The transpiler is divided into phases that are explained in 3.4.

6.1 Design of the transpiler

Before the start of the implementation process itself, it was necessary to create a maintainable and easily extensible design of the transpiler, which may support a larger subset of the language in the future.

The first design concern was the usage of existing standardized tools to build compilers, such as Lex for lexical analysis or Bison for generating a parser. After a discussion with the supervisor, a decision was made not to use any of the tools due to the size of the subset of the source language, which can be processed using approaches such as recursive descent parsing, which can be easily written by hand without the need for a parser generator. Designing the entire workflow without relying on standardized tools may offer greater flexibility and enhance the learning experience.

The second problem was the approach to code generation. Although it is possible to generate the code directly during the syntax analysis process, this approach would not be ideal when translating the source code to other high-level programming languages, as it requires more context, than in the case of generating the three-address code or other similar low-level language. Similar to other transpilers, the ideal solution is to construct an abstract syntax tree, which can be then statically analyzed, transformed, and used for the generation of the target code with enough context to generate a functionally equivalent program.

To make the transpiler extensible and maintainable, it is split into modules, which communicate with each other. The modules are as follows:

- **Symbol table** - Data structure to store and retrieve constructs such as identifiers, or functions.

- **Lexer** - Performs lexical analysis, splitting the source code into tokens. Communicates with the syntax analyzer to provide tokens.

- **Parser** - Controls the entire compilation process and contains an implementation of the LL parser.

- **Precedence parser stack** - Data structure used during expression analysis. Contains methods for push operation and decides what the next operation will be based on the configuration of the stack.

- **Expression parser** - Parsing of expressions. Called by the *Parser* module. Implementation of the operation precedence parser.

- **AST** - Definition of nodes used in the construction of the abstract syntax tree. Each node is derived from the common `Node` parent node and implements a `generate_code()` and `generate_dot()` method.

- **Code generator** - The data structure used to store generated code and provides utilities such as unique variable name generation.

6.2 Lexical analysis

The lexical analyzer is located in the *src/lexer* of the project folder structure in the class `Lexer`. It is implemented as a deterministic finite state machine 2.2.1. At initialization, the lexical analyzer reads the source code and splits it into tokens, which are stored in the memory of the lexer.

During the syntax analysis phase of the compilation, the parser communicates with the lexical analyzer using the function `get_next_token()`, which returns the next token object and the function `unget_token()`, which decreases the index of the current token, so the token can be returned by next call of the `get_next_token()` function. It is used in such cases when the parser needs to look for more tokens ahead to parse the source program.

The token object contains information that is necessary for the compilation such as token type and value. In addition, it also provides additional information such as the location of the token in the source code (line and column), that are used for debugging and error logging purposes. It ignores tokens that are not essential for the compilation process such as comments, or white spaces. Lexer has to distinguish between identifiers and built-in keywords. In PHP, keywords and function names are lexically identical, so it is distinguished by the keyword table.

Lexer reads the source code character by character and based on the current state and the incoming character, it decides which state to transition into. When the state is marked as terminal and current character does not belong to the currently processed token, it returns the new token object and makes a transition to the initial state. The lexer contains other private methods, that are used during the lexical analysis including `get_next_char()`, or `add_token()`, which adds the created token to the list of tokens, that are further used during the compilation.

6.3 Symbol table

The symbol table is a data structure used during syntax and semantic analysis to store information about identifiers, such as functions and variables. It is located in the *src/utils* in the project structure and is divided into two modules - a variable table and a function table.

The purpose of the symbol table is to store the names of variables used in the program in different scopes. It is implemented as a stack of scopes, so when the parser enters the body of a function, the previously used scope is stored in the stack, so it can be used later. Available methods are `get_var(name: str)` that look for a variable in the symbol table and returns the object if it is found, methods for working with scopes such as `pop_scope()` or `push_scope()`, or methods for adding new variables. It uses Python's built-in dictionary data structure, which stores key-value pairs.

The function table is used to store the functions defined within the program and the built-in functions of the standard library. When the function call is found during the syntax analysis, the parser uses the `get_function()` method to verify, whether the function has been defined or exists in the standard library. If it exists, the node corresponding to the function is returned so it can be used during the abstract syntax tree construction. During the definition of a new function, the `add_function()` method is used, which, in addition to adding the new function object to the table, also verifies whether the function had not been defined before. It also includes a list of built-in functions, where for every built-in function exists a specific type of abstract syntax tree node, which includes methods for generating the code in the target language.

6.4 Syntax analysis

A second part of the compilation process, syntax analysis, verifies the syntactic correctness of the source program. As already mentioned in 3.4, syntax-directed translation controls the entire compilation process. The syntax analyzer requests the next token from the lexer or rewinds to the previously obtained token. Every rule in the parser includes semantic actions, which not only generate nodes of the **abstract syntax tree** but also perform various semantic analyses, such as controlling the existence of a variable in the symbol table or verifying the type compatibility in operations. The syntax analyzer uses two different approaches: **top-down parsing** 6.4.2 and **bottom-up parsing** 6.4.3. These approaches are used in different situations where the top-down approach is used for parsing the general structure of the program, such as loops and conditions, and the bottom-up parsing verifies expressions.

6.4.1 Abstract syntax tree construction

Abstract syntax tree represents the structure of the source program and is the result of the syntax analysis phase. This representation is later used for the generation of the target code. It consists of nodes, which are defined in the `src/ast` in the project structure. The *Node* class serves as the base for all AST nodes, defining the `kind` attribute and two common methods:

- `generate_dot()` - Generates DOT language code for visualization of the node. This method recursively generates DOT code for each of its children, allowing for the visualization of the entire abstract syntax tree.

- `generate_code()` - Using the context around the node generates the code in the target language - JavaScript. It uses a similar recursive approach to the `generate_dot()` function, meaning it calls the function for all of its child nodes.

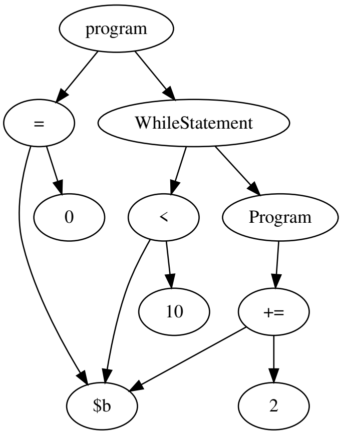

Figure 6.1: Visualized AST of a simple PHP program

Other subclasses extend the `Node` base class by providing custom attributes, their implementation of `generate_dot()` and `generate_code()` methods but also custom methods. The root of the tree consists of the node of type `Program`:

```
class Program(Node):
    def __init__(self, kind: str = "Program"):
        super().__init__(kind)
        self.children: List[Node] = []
```

Listing 6.1: Root of the AST

The parser then adds child nodes to the current root of the current code block, meaning that if the parser enters a different code block, for example, the body of the while statement, the node representing the body of the construct will become the current root of the AST, and the previous root will be saved in the stack.

6.4.2 Parsing of the program structure

The main part of the syntax analyzer is implemented in the class `Parser`. It is a recursive descent parser that analyzes most of the syntactic structures of the code, except for expressions. The principle of a recursive descent parser is explained in 4.1.1 The process starts with the call to the method, which corresponds to the starting symbol of the grammar. It uses a look-ahead token to choose the appropriate action. As an example, when the token represents the `if` keyword, a method corresponding to the `if` nonterminal is called. If the method is completed successfully, the function corresponding to the starting symbol of the grammar calls itself recursively until the end of the program is reached. As the goal of syntax analysis is to build an abstract syntax tree, every function creates a node of a type, which corresponds to the current nonterminal. Each type of node has different attributes, which are necessary for creating the intermediate code representation. When the procedure matching the nonterminal finishes successfully, the newly created AST node is added as a child to the parent node.

```python
class WhileStatement(Node):
    def __init__(self, node_type: str = "WhileStatement"):
        super().__init__(node_type)
        self.test: Node = Node()
        self.body: Program = Program()
```

Listing 6.2: AST node corresponding to while-loop construct

When the expression construct is expected, the parser has to cooperate with the operator precedence parser. It calls the function `analyze_exp(term_tokens)`, which takes the list of terminating tokens as an argument to know where the expressions end. If expression parsing succeeds, the abstract syntax tree of the expression is returned. In the while-loop example, the returned expression may be used as a `test` attribute, which represents the condition of the loop. The analysis is performed in two passes because PHP allows function calls before the function has been defined. In the first pass, the parser checks the function definitions and stores them in the function table. In the second pass, the remaining source code is checked. This means that if a function call is encountered and the function has been defined, even later in the program, it has already been added to the function table.

6.4.3 Parsing of expressions

Due to the different priorities of the operator in the expressions, the recursive descent parser cannot be used for this purpose. The ideal approach may be to use an operator-precedence parser explained in section 4.2.1. It is implemented in `src/expression_parser` in the project structure and consists of three main parts - precedence table, stack, and the parser itself. The table contains a corresponding operation for every combination of the top-most terminal and the incoming terminal. At the initialization of the stack, the starting symbol *$* is inserted. The stack stores terminal and nonterminal symbols, which will be used in the reduction process. It also provides helper functions such as `insert_handle()`, `stack_push()`, or `get_operation()`.

The `PrecParser` class is provided with a set of tokens, which signal the end of the expression. The `analyze_exp()` function is implemented as an infinite loop, which gets a new token from the lexical analyzer and, using the previously mentioned *get_operation*(), performs the corresponding action. If the operation signals reduction, items from the stack are popped and saved to the `reduced_items` variable until the handle symbol is hit. The parser then tries to find which rule corresponds to the item for reduction. If such a rule exists, the reduction operation is performed and the result is pushed back to the stack; otherwise, an error is reported. During the reduction, various semantic checks are also performed, meaning that it cooperates with the symbol table, as well as analyzing the type of the outcome of the operation.

Function calls can be part of expressions, which makes the implementation of the expression parser more complex. First, it has to verify the existence of the function in the symbol table and report an error if it has not been defined. Then, it has to switch to the recursive descent approach to parse arguments of the function call and then continue in the process of expression parsing as the function call can be considered a constant, similar to the occurrence of a variable in expressions.

6.5 Semantic analysis

In addition to checking the syntax of the source program, the parser has to validate the semantic correctness of the program as well. It is not implemented as a stand-alone module but rather performed during the syntax analysis and code generation phases.

The first thing to verify is the existence of identifiers and functions. At every occurrence of a variable in expressions, the parser cooperates with the symbol table, which contains information about all identifiers, to verify whether the variable has been declared and can be used in the current scope. The symbol table implements a method $get\_var(name : str)$, which either returns the object corresponding to the variable or none if it has not been found. New variables are added to the symbol table during the syntax analysis phase, more precisely when the `variable_assignment` rule is being executed. Similar to checking the existence of variables, when the function call within an expression is processed, the parser needs to verify the existence of the function in the table of functions. Functions can be defined within the program or be a part of the supported subset of built-in functions of PHP. If the function exists, the parser needs to verify the correct number of arguments that are passed to the function during its call. Additionally, parameters and the return type of the function may optionally have declared types, which have to be semantically analyzed as well.

The second task of the semantic analysis is to verify the compatibility of operators and their types. Most of the type compatibility analysis occurs in the expression parser during the reduction operation. First, the parser has to confirm that the given operator is compatible with its operands; for example, operation addition (+) can not have a string as its operand.

6.5.1 Type inference

The goal of type inference is to assign a type to each expression that occurs in a program [3]. In addition to verifying the type compatibility of operators, the type of the result has to be inferred, which is most cases affected by the operator, where, for example, result of the comparison operation will be of type boolean. In general, type inference rules specify, for each operator, the mapping between the operand types and the result type [3].

The inferred type is propagated to other parts of the program and may later influence the function of the generated code. The simplest rule is variable assignment, where the type on right side is assigned as the type to variable on left side of the assignment. The type is assigned to the variable object found in the symbol table and can be used when the variable occurs in expressions. In contrast with statically-typed languages, the declaration of functions return and parameters types in PHP is only optional, and has to be inferred by a semantic analyzer. The type declarations, however, enable better type checking when the function call is a part of an expressions and can potentially provide more accurate code generation. The type is returned as a result of the `analyze_exp()` function and can be assigned to variable.

6.6 Code generation

The last step in the chain is code generation. It uses the abstract syntax tree representation of the source code to generate the code in the target language. To generate functionally equivalent code, every node in the tree has to know the context surrounding it, due to different semantics of operations in source and target languages. It uses the `CodeGenerator` class located in `/src/generator` in the project structure. In addition to storing the generated source code, it provides methods for generating unique variable names or changing the order of generated code. The generation of code uses the `generate_code()` method, which is defined for every type of node that is used in the construction of the abstract syntax tree.

The target code generation phase uses the recursive approach, where the `generate_code()` function is initially called for the root of the abstract syntax tree, which represents the `Program` type of node, meaning it includes a list of children nodes. The `generate_code()` is then called for every child node, which has its own implementation of the method, and generating the code from the entire abstract syntax tree.

6.6.1 Variable generation

When generating a new variable in JavaScript, it can be declared in 3 ways - using `const`, `var`, `let`, or automatically, with `var` used mostly in older browsers. Therefore during the syntax analysis, we have to keep track, of whether the current occurrence of the variable is the first one and should declare a new variable using `let` or `const`. To decide between `let` and `const`, we have to keep track of whether the variable has been reassigned.

Another problem is, that in JavaScript, function has access and can modify variables outside the scope of the function. This is solved by a symbol table that keeps track of the existence of variables in certain scopes.

The major problem with a variable is keeping track of its type, which influences the functionality of generated code. The type of variable is determined by the type of expression that is assigned to the result. However, when the code is generated after the parsing and the variable could have been reassigned many times, we need to find the type of it at a certain point in time retrospectively. To solve this, every variable object has a queue, which includes the history of the assignment operation so we can find the type during the code generation.

6.6.2 Control structure generation

Control structure such as conditions or loops have almost identical syntax and semantics in the source and target languages, so the generation itself is straightforward, since it is using the recursive approach.

```python
def generate_code(self, parent: Node) -> str:
    gen_code = "while ("
    if self.test:
        gen_code += f"{self.test.generate_code(parent=self)} ){{"
    gen_code += f"{self.body.generate_code(parent=self)}}}\n"
    return gen_code
```

Listing 6.3: Generation of the while-loop construct

6.6.3 Expression generation

The precedence of operators has been solved by the operator-precedence parser, so in code generation, the code can be generated recursively from the abstract tree, only using brackets to split the left and right subtrees to generate code for binary operations such as addition or subtraction.

However, in order to generate functionally equivalent code, we need to use the context, as there is difference in type conversion, mutability of strings between languages, and many other differences not obvious at the first sight. Strings in PHP are mutable, which means what in order to change the character at certain index, it is possible to assign a new character to the desired index. To simulate this behavior in JavaScript, the string has to transformed to a type, which allows mutation, such as an array, and mutate it. It can be then transformed back to string, and assigned back to the original variable. When generating `Assign` nonterminal, left side of which contains specific index and the right side is of type string, the generator provides the code corresponding to the series of steps mentioned before. Another example may be casting a boolean value to a string. In PHP, the boolean value `true` is cast to '1' and `false` to '', whereas in JavaScript, it's 'true' and 'false'. The context has to identified by the generator and generate the code functionally equivalent with that in PHP. Therefore, each node needs access to its parent, which provides the context in which the node is used.

6.6.4 Functions generation

The syntax of a function definition in PHP is very similar to that of JavaScript. Each function is identified by its name, which is then used for the call of the function and has to be unique within the program. It also includes list of parameters, where each parameter may optionally have a type declared. Optionally, a return type of function may be declared, simplifying the semantic analysis of expressions calling the function. The problem with function generation in the target language is, that variables and functions name may collide, which is not a problem in PHP. Therefore, during syntax and semantic analysis, when a new function is declared, a new unique name is generated for the function, which is then stored with the function in the table of functions. This ensures that during the code generation process, a function is generated with the newly generated name. Function calls have to be changed, so they match the newly generated name of function, as the function can be found in the function table using the original name.

A supported subset of built-in functions is generated in two ways. The first approach is that function calls are directly replaced by functionally equivalent code in the target language, such as `pow($number,$exponent)` being replaced by `Math.pow(number, exponent)` for the exponent function, or by a relatively simple sequence of steps to simulate identical behavior of the original function. The second approach is used, when the code needed to replace the original built-in function is more than a one-line function, such as in the case of the `range()` function. In this approach, the code in the target language is stored in the `CodeGenerator`, so at the start of the code generation process, saved built-in functions are inserted at the start of the generated code, making them accessible for use.

Chapter 7

Testing and comparison with alternative tools

This chapter describes the testing process of the implemented transpiler and compares the results with existing tools. The comparison will focus on the size of the subset of the source language, support of built-in functions, and whether the generated code is functionally equivalent to the source program.

7.1 Testing

A set of automatic tests has been developed to verify the correctness of the implementation. Test scripts are located in `/tests` folder containing a set of `.php` programs. Each of the programs is transpiled into the target language and the results of running the original and the transpiled program are compared. As the transpiler should produce functionally equivalent code, the outputs must match for the test to pass successfully. The testing is done using `Node.js v20.12` and `PHP 8.1.2`.

The set of test cases includes tests at different levels of complexity, starting with testing simple language constructs such as assignment to variable, operations with implicit type conversion, testing of supported built-in function, or recursive function call. However, it also includes more complex program examples, such as calculation of factorials or different types of sorting algorithms. A subset of tests has been adopted from RosettaCode[1], which is a website that presents solutions to programming tasks in as many different languages as possible, including PHP. An example may be Euler method, or Sierpinski carpet. The source code of the solution in PHP from the RosettaCode is transpiled into JavaScript and outputs of original and transpiled program are compared. Even though the transpiler detects error, most of the test programs are syntactically and semantically correct, as the goal of testing is not to verify error detection, but rather code generation in the target language.

Regression testing using the developed test set helped to ensure functionality of previously implemented modules during refactoring of the code or adding support for new features. The testing ability is limited by the supported subset of the PHP, as commonly used object oriented approach is not supported, as well as many commonly used built-in functions.

[1] RosettaCode: https://rosettacode.org/

7.2 Comparison with alternative tools

The capabilities of the implementation were compared with the set of existing tools, which differ in approach to transpiling and supported subset of the language.

7.2.1 Uniter

Uniter [9] is an open-sourced transpiler from PHP to JavaScript written in JavaScript. The tool enables PHP to run in the browser, using a custom transpiler. However, the focus will be on its capabilities within the Node.js environment, where it can be installed using the NPM package manager. It includes a command line interface, which offers three main functions; directly running the PHP program using Node.js, transpiling the PHP to JavaScript and returning the generated code, and returning the abstract syntax tree of PHP code in the JSON format. It supports a larger subset of PHP, including object-oriented programming capabilities, `goto` statement, magic constants, and a small subset of built-in functions. The main limitation is the generated code, which is generated in a format, which depends on using the `phpruntime` runtime package to execute the code. Even though the approach allows better control over type conversions and semantic differences, it makes the transpiler dependent on the custom runtime packages, therefore not generating universally usable code not only within the Node.js environment but across different runtimes, such as Deno or Zod. When tested in the tests mentioned in 7.1, the support for the built-in function is more limited, causing many tests to fail.

```
require('phpruntime').compile(function(core) { // echo 23;
  var createInteger = core.createInteger,
    echo = core.echo,
    instrument = core.instrument,
    line;
  instrument(function() {
    return line;
  });
  line = 3;
  echo((line = 3, createInteger(23)));
});
```

Listing 7.1: Generated JavaScript code for the PHP program: `echo 23;`

7.2.2 PHP-to-JavaScript

The PHP-to-JavaScript[2] provides a simple web interface playground to test its capabilities. In addition to traditional constructs such as loops or conditions, the transpiler supports namespaces, classes, or constants, which is a bigger set of the supported language than is supported by the transpiler developed for this thesis. The practical usage is however very limited by the fact, that it does not support any built-in functions, such as functions to work with arrays. Unknown functions are transpiled to JavaScript with an equivalent name, which produces non-functioning code, as the name of the built-in function is not defined in the target language. Another problem is the absence of type inference and almost entire semantic analysis. The transpiler does not report errors on the use of undefined variables,

[2] PHP-to-JavaScript: https://github.com/tito10047/PHP-to-Javascript

functions, or incompatible type operations. It is therefore generating a functionally different program. As an example, we may take a simple `echo true;` PHP program, which outputs the expression with no additional newlines or spaces. The `echo` construct is replaced with `console.log()`, which adds a new line after the output. It also does not perform implicit type conversion from boolean to string, which in PHP would output `1`, as the echo transforms the boolean value into string, which in PHP is represented by '1'. Overall, the tool supports a larger subset of the language, but the absence of semantic analysis and naive approach to code generation, which generates functionally non-equivalent code makes the usage of the tool problematic.

7.2.3 LLM-based tools

With the rise in popularity of large language models, several tools use AI for converting code from one language to another. Example of these tools are Codeconverter[3], Docuwriter[4], or prompting any large language model, like ChatGPT, to perform the code transpilation task. Due to non-determinism in these tools, it is hard to benchmark their capabilities and the subset of supported language. The practical usage is limited by the „hallucination“, where models generate a confidently-sounding answer, which is however fundamentally incorrect. That is, for any given PHP source code, it will generate the JavaScript source code, even though it may be functionally totally different. The advantage of using LLM is that the generated code is styled, like it has been written by a human programmer, not generated by a machine, therefore it's more readable and easier to debug and fix errors, so the ideal use case of LLM-based tools would be the acceleration of the migration of code base to different programming languages when the code maintainability is preferred over exact function equivalence. Using the set of tests from 7.1, the LLM-based tools perform better than the 2 tools mentioned above.

[3] CodeConverter: `https://www.codeconvert.ai/free-converter`

[4] Docuwriter: `https://www.docuwriter.ai/php-to-js-code-converter`

Chapter 8

Conclusion

The goal of the thesis was to design and implement a high-level language transpiler from PHP to JavaScript, and it has been achieved. The transpiler supports a usable subset of the source programming language, allowing usability for PHP programs with traditional syntactic structures such as loops, conditions, or function calls. It even supports a small subset of PHP's built-in functions, such as functions for working with arrays, strings, or math operations.

The transpiler is built without the use of any tools, such as lexical analysis or parser generators, using approaches mentioned in Chaps 3 and 4. Using top-down and bottom-up parsing, the abstract syntax tree is built that represents the structure of the source code. It is analyzed and used for the generation of the code in the target language. The abstract syntax tree can be visualized using Graphviz. To verify the correctness of the implementation, a set of automatic sets has been developed, including tests of different complexity. The capabilities of the developed transpiler were compared with existing solutions, such as Uniter or LLM-based tools, showing both advantages and disadvantages compared to these solutions. Even though the developed tool supports a smaller subset of the language, the output is deterministic, and its main focus is on function equivalence of the generated source code, which was not always the case in the previously mentioned tools. The transpiler performs static analysis and type inference of the source code, which is necessary for the generation of the functionally equivalent code in the target language, which was not the case in most of the tested alternative solutions.

The main limiting factor of practical usage is the size of the supported subset of PHP. To be usable, support for an object-oriented approach is necessary, as it is commonly used in PHP code bases. The support for built-in functions is a very important factor in terms of practical usability. The transpiler supports most commonly used functions, mostly to work with arrays and strings, however, the greater subset would increase the usability of the transpiler. The current design allows for a simple extension of the set of built-in functions. It also does not support `include` or `require`, allowing usage of multiple files. To reach the functional equivalence, the generated code's structure may not be easily readable and maintainable, therefore, it is more useful as a tool for running the PHP codebase in JavaScript, rather than accelerating the migration of code base from one language to another.

Bibliography

[1] Aho, A. V. *Compilers : principles, techniques & tools.* 2nd ed. Boston: Addison Wesley, 2007. ISBN 0-321-48681-1.

[2] Contributors, M. *JavaScript* [online]. 2024. [Accessed: April 19, 2024]. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

[3] Cooper, K. D. and Torczon, L. Chapter 4 - Context-Sensitive Analysis. In: Cooper, K. D. and Torczon, L., ed. *Engineering a Compiler (Second Edition).* Second Editionth ed. Boston: Morgan Kaufmann, 2012, p. 161–219. DOI: https://doi.org/10.1016/B978-0-12-088478-0.00004-9. ISBN 978-0-12-088478-0. Available at: https://www.sciencedirect.com/science/article/pii/B9780120884780000049.

[4] Ecma, E. 262: Ecmascript language specification. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,.* 14th ed. 2023. [Online; Accessed: April 22, 2024]. Available at: https://262.ecma-international.org/.

[5] Fischer, C. N., Cytron, R. K. and LeBlanc, R. J. *Crafting A Compiler.* 1stth ed. USA: Addison-Wesley Publishing Company, 2009. ISBN 0136067050.

[6] Lesk, M. E. and Schmidt, E. E. Lex—a lexical analyzer generator. 1990. Available at: https://api.semanticscholar.org/CorpusID:7900881.

[7] Meduna, A. *Elements of Compiler Design.* 1st ed. Milton: Auerbach Publishers, Incorporated, 2007. ISBN 1420063235.

[8] Meduna, A. and Křivka, Z. *Materials for IFJ* [Brno University of Technology]. 2017. [Online; Accessed: March 23, 2024]. Available at: https://www.fit.vutbr.cz/study/courses/IFJ/public/materials/.

[9] Phillimore, D. *Uniter* [online]. 2023. Accessed: April 22, 2024. Available at: https://phptojs.com/.

[10] Team, P. D. *PHP Manual.* 2024. Online; Accessed: 13-April-2024. Available at: https://www.php.net/manual/en/.

[11] Thain, D. *Introduction to Compilers and Language Design.* 1st ed. Lulu.com, 2016. ISBN 0-359-13804-7.

Appendix A

Contents of the included storage media

```
/
├── implementation ........................ Implementation of the transpiler
│   ├── src ................................ Source files of the transpiler
│   │   ├── transpiler.py .................. Entry point of the transpiler
│   │   ├── ast ............................ Nodes of the AST
│   │   ├── lexer .......................... Implementation of the lexical analyzer
│   │   ├── parser ......................... Implementation of the recursive-descent parser
│   │   └── expression_parser .............. Implementation of the operator-precedence parser
│   ├── tests .............................. Folder containing test PHP programs
│   │   └── *.php .......................... Set of test programs
│   ├── test.py ............................ Test script
│   └── README.md .......................... Transpiler manual
├── thesis ................................. Thesis LaTeX source codes
├── README.md .............................. Thesis manual
└── xmokri01.pdf ........................... Thesis text
```