Assignment of bachelor's thesis

Title:	TinyGo Language
Student:	Maksim Gusev
Supervisor:	Ing. Petr Máj, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science 2021
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Analyze the Go programming language with emphasis towards differences with the C language. Design a simplified variant of Go, TinyGo, that preserves the important design principles behind Go, but which can be used for educational purposes in compiler courses. Familiarize yourself with the Tiny86 virtual target used in the NI-GEN course and write a TinyGO compiler frontend and backend for it.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 13 February 2024 in Prague.

Bachelor's thesis

TinyGo Language

Maksim Gusev

Department of Theoretical Computer Science
Supervisor: Ing. Petr Maj, Ph.D.

May 16, 2024

Acknowledgements

In the first place, I would like to thank my parents, who gave me such an opportunity to study abroad and get a higher education. I appreciate their support and faith in me. I would also like to thank my supervisor, Ing. Petr Maj, Ph.D., for his guidance and support not only during this thesis but also during my applications for my masters.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Maksim Gusev. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Gusev, Maksim. *TinyGo Language.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Tato práce představuje jazyk TinyGo a jeho překladač. TinyGo je zjednodušená verze skutečného programovacího jazyka GoLang. Byl vytvořen pro vzdělávací účely studia konstrukce kompilátoru a zachování nejdůležitějších principů designu Go. Kompilátor pracuje pro jazyk TinyGo a generuje cílový kód pro malý x86 VM z kurzu NIE-GEN. Kompilátor je oddělený pro snazší pochopení jeho částí a otevřený pro jakékoli rozšíření a upgrade.

Klíčová slova Kompilátor, AST, IR, Generování Kódu, Programovací Jazyky

Abstract

This thesis presents the TinyGo language and its compiler. TinyGo is a simplified version of the real programming language GoLang. It was created for the educational purpose of studying compiler construction and preserving the most important design principles behind Go. The compiler works for a TinyGo language, generating the target code for a tiny x86 VM from a NIE-GEN course. The compiler is separated for an easier understanding of parts of it and open for any extension and upgrade.

Keywords Compiler, AST, IR, Code Generation, Programming Languages

Contents

1 Introduction **1**
- 1.1 Compiler 2
- 1.2 TinyGo 3
- 1.3 Tiny x86 4
- 1.4 The Goal of Thesis 4
- 1.5 Thesis Outline 4

2 Overview of a Modern Compiler **7**
- 2.1 Front End 7
  - 2.1.1 Lexical Analysis 8
  - 2.1.2 Syntax Analysis 9
  - 2.1.3 Semantic Analysis 9
  - 2.1.4 Intermediate Representation 10
  - 2.1.5 IR Generator 11
- 2.2 Middle End 11
- 2.3 Back End 13
  - 2.3.1 Instruction Selection 13
  - 2.3.2 Register Allocation and Assignment 15
- 2.4 Existing Compilers 15
  - 2.4.1 GCC 16
  - 2.4.2 LLVM 16

3 Compiling Pragramming Languages **19**
- 3.1 C 20
- 3.2 GoLang 21
- 3.3 Comparison of the C and GoLang 22
  - 3.3.1 Syntax 22
  - 3.3.2 Static Typing 24
  - 3.3.3 Concurrency 24
  - 3.3.4 Memory Management 24
  - 3.3.5 Error Handling 25
  - 3.3.6 Top-level Declarations 26
  - 3.3.7 Object-Oriented Programming 26

4 Design And Implementation **29**

4.1 TinyGo . . . 29
4.1.1 TinyGo Specifications . . . 29
4.1.2 Omitted features . . . 30
4.2 Design . . . 31
4.2.1 Front-End . . . 31
4.2.2 Middle end . . . 34
4.2.3 Back end . . . 34

5 Evaluation **37**
5.1 TinyGo Compiler . . . 37
5.2 Influence of the TinyGo in Compiler . . . 39
5.2.1 Front end . . . 40
5.2.2 Middle end . . . 40
5.2.3 Back end . . . 40
5.2.4 Other Compilers . . . 40

Conclusion **43**
Future Work . . . 43

Bibliography **45**

A TinyGo Grammar **49**
A.1 Notation . . . 49
A.2 Program . . . 49
A.3 Declarations . . . 49
A.4 Functions and methods . . . 49
A.5 Types . . . 50
A.6 Numbers and Identifier . . . 50
A.7 Numbers and Identifier . . . 50
A.8 Statements . . . 51
A.9 Expression . . . 51
A.10 List . . . 52

B IR Grammar **53**
B.1 Variables and Types . . . 53
B.2 Constants . . . 53
B.3 Functions . . . 53
B.4 Memory Access and Allocation Instructions . . . 54
B.5 Control Flow Instructions . . . 54
B.6 Other Instructions . . . 54

C Contents of attachments **55**

List of Figures

1.1 Compiler principle of work 2
1.2 Compiler scheme 3

2.1 Structure of the Front End 8
2.2 Shadow cast of the types by type checker 9
2.3 Structure of the Back End 13

3.1 Programming language as the input on the scheme 19

4.1 Look of the stack 34

List of Listings

1.1 Program that computes the modulo of the number, written on a machine code . . . 1
1.2 Example of the programs in different languages . . . 2
1.3 The program written in GoLang . . . 3
2.1 Lexer translation of the characters into tokens . . . 8
2.2 Example of elimination of the non-reachable code . . . 12
2.3 Example of the constant propagetion in the code . . . 12
2.4 Example of the common subexpression elimination . . . 13
2.5 Program written on LLVM IR . . . 17
3.1 Example of the program written on C . . . 20
3.2 Example of the program written on Go . . . 21
3.3 Example of the variable declarations in languages . . . 22
3.4 Example of functions and methods in languages . . . 23
3.5 Example of multiple return values . . . 23
3.6 Error handling in languages . . . 25
3.7 Example of GoLang top level declarations . . . 26
3.8 Example of the polymorphism in languages . . . 27
3.9 Example of the member inheritance in languages . . . 27
4.1 Function written on a TinyGo, that computes the nth fibonacci number . . . 29
4.2 Topological sort of the variables . . . 32
4.3 Program in IR for computing the sum of numbers . . . 32
4.4 Go function which returns more than one variable in IR . . . 33
5.1 Example of the wrong programs . . . 37
5.2 Different programs written to test the compiler . . . 38
5.3 Example of using functions and methods . . . 39
A.1 Grammar of the program . . . 49
A.2 Grammar of declarations . . . 49
A.3 Grammar of the function (method) declaration . . . 50
A.4 Grammar of types . . . 50
A.5 Grammar of numbers and identifier . . . 50
A.6 Grammar of numbers and identifier . . . 50
A.7 Grammar of statements . . . 51
A.8 Grammar of expressions . . . 51
A.9 Grammar of lists . . . 52

B.1 Variables and types grammar 53
B.2 Grammar of constanst in IR 53
B.3 Grammar of the functions 53
B.4 Memory instructions grammar 54
B.5 Grammar for control flow 54
B.6 Grammar of the other instructions 54

Chapter 1

Introduction

Computers do not understand programming languages. Machines operate with instructions, where each statement is a direct command to the computer's hardware. Such commands are very hard to read for people due to their extreme level of detail (Listing 1.1). Another problem with machine languages is portability. Different computers might accept different sets of instructions and commands from one computer will not be processed by another.

```
.data
    dividend dd 25
    divisor dd 4
    result dd 0

.text
    mov eax, dword [dividend]
    mov ebx, dword [divisor]
    xor edx, edx
    div ebx
    mov dword [result], edx

    mov eax, 4
    mov ebx, 1
    mov ecx, result
    mov edx, 10
    int 0x80

    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Listing 1.1: Program that computes the modulo of the number, written on a machine code

Because of that, programming languages were created. They provide a means for humans to communicate with computers in a structured and, at the same time, understandable way. But this creates a gap, where humans write programs on programming languages and computers understand instructions. A popular way to solve it with using a translation program named *compiler*, which takes a program written in a programming language and converts it into an instruction set with the same meaning.

```go
func div(a,b val)(int, error){
    if b == 0{
    return -1,errors.New("")
    }
    return a / b
}
```

(a) GoLang

```rust
fn main() {
    let x = String::from("");
    let y = x.clone();
    println!("{}, {}",x, y);
}

```

(b) Rust

```cpp
int* createInt() {
  int* resutl;
  result = new int;
  return resutl;
}
```

(c) C++

```lisp
(defun factorial (n)
  (if (or (= n 0) (= n 1))
      1
      (* n (factorial (- n 1))
    )))

```

(d) Lisp

Listing 1.2: Example of the programs in different languages

There are various of programming languages (Listing 1.2). Different languages have different syntax, different features, and different ways to handle them. As an example, Go can return from the function multiple values at the same time(1.2a). In rust, any object has its owner and cannot be simply passed to another variable (line 3 1.2b). C++ requires to manually allocate space for some variables (1.2c). In Lisp, everything is a list and this language does not have variables (1.2d).

It is therefore clear that different languages require different compilers to translate them. However, the significance of the differences between these compilers, who accept different languages, is not. It is possible, that compilers would be divergent from each other. Or source language might not affect at all and all compilers can be almost identical.

1.1 Compiler

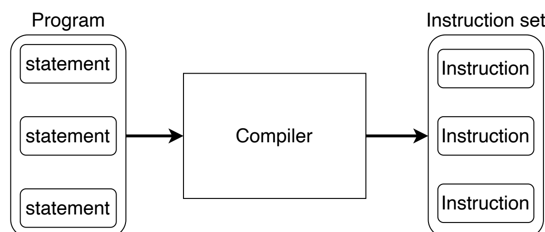


Figure 1.1: Compiler principle of work

Creation of the compiler is a substantial problem, that requires a lot of things to be learned. The NIE-GEN and BIE-PJP courses in the CTU teach about parsers, compiler construction, and different techniques for implementation.

To make the learning and developing process easier, mordern compilers splited into three main parts (Figure 1.2):

- **Front End :** It is responsible for reading the input program and analyzing it. Then it transltes the input into the IR that is then processed by the rest parts of the compiler.
- **Middle End :** The middle end is responsible for an IR optimization. It takes the result from the front end and creates a better-quality code, which then sends it to the next stage. However, this part is not mandatory and compiler can work even without it.
- **Back End :** The back end takes the optimized IR and generates the instruction set to a target machine, as the result of the compiler work.

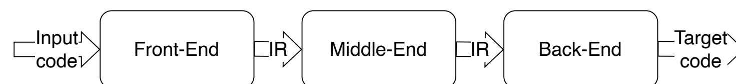


Figure 1.2: Compiler scheme

1.2 TinyGo

Go (or Golang) is a high-level compiled programming language designed by Google. It was chosen as a reference language for a compiler input because of its grammar. It has a lot of syntactic sugars, which complicates the compiler, and gives more opportunities to use different techniques to construct it.

```
1  package main
2
3  func init_values() (int,int) {
4    return 1,2
5  }
6
7  func sum(a,b int) int {
8    return a + b
9  }
10
11 func main() {
12   var a,b int = init_values()
13   c := sum(a,b)
14   if c == 3 {
15     Print("Value equals to 3!")
16   }
17 }
```

Listing 1.3: The program written in GoLang

TinyGo is a simplified subset of the Go. It has been simplified as much as possible while keeping the most interesting components of the compiler to make it more appropriate for an educational purpose to study compiler construction and code generation. It supports functions, variables, pointers, named types[1], methods. Inside functions, language supports usual control flow statements

[1] In GoLang each type can have a shortcut on how to call it - a name. Types which is used by its name called *named type*.

(*if*, *for*, *switch*) and local variables. Basic data types have been stripped and only floats, integers, structures, and pointers are kept.

1.3 Tiny x86

Different computers might accept different instructions. Covering all possible instruction architectures will require an enormous amount of work. Luckely, for the NIE-GEN course special instruction architectures was designed by Ivo Strejc [1]. It is a virtual machine with custom ISA, created for an educational purpous of studying code generation. Thanks to it, students can focus more on the compiler itself instead of studying complex and hard architectures, such as x86 or RISC-V.

Tiny x86 originally worked as a C++ library. It allowed through its interface create instructions and build them into a target program. However, Filip Gregor has extended it [2] with a CLI, which allowed not to use an interface, create a target program, and only after that execute it. Such method is preferred and in this thesis, we will use it to execute the program created by compiler.

1.4 The Goal of Thesis

This thesis aims to look into the creation of the compiler. For it, we will design a new language – TinyGo, a simplified version of the already existing language. Next, it will be compared to a similar language, that has been created for the same purpose – TinyC. Afterward, we will create a compiler for a TinyGo language compare it with already existing compilers for a TinyC, and look into how does introduction of the new language changes the structure of the translation process.

1.5 Thesis Outline

In the following chapters, we will describe the TinyGo language and how the compiler is constructed.

In **Chapter 2** we will talk about various of different techniques and methods which is used in compiler construction. Will talk about their pros and cons, which are better, and in which case any of them could be better to be used.

Chapter 3 will describe the design of the TinyGo: what syntax does it have, which features from the original language have been omitted, and how the language differs from the similar one, TinyC, from the NIE-GEN course. We will also talk about how the compiler is designed which techniques have been used from chapter 2 and why.

Chapter 4 speaks about how methods from Chapter 3 are implemented. It focused on implementation and how challenges have been overcome.

In the last **chapters 5 and 6**, we will talk about the result of the work. and how the compiler works compared to a standard one of the original language. Also, it will compare how long it takes and what is the quality of the code. The last chapter will speak about the whole result, and what has been done.

Also, it will outline what hasn't been done yet and what can be done in future works, which will improve the current one.

Chapter 2

Overview of a Modern Compiler

Compiler is a fundamental tool in computer science and software engineering, serving as a bridge between human-readable source code and machine-executable instructions. Essentially, it's a specialized program that translates high-level programming languages like Java, C++, or Python into low-level machine code that a computer can understand and execute directly.

Such ways have changed the programmer's life by allowing them to express their ideas in high-level languages without the burden of dealing with low-level instructions. With a compiler, programmers can focus on the logic and structure of their code rather than dealing with the machine code's direct instructions. This abstraction allows developers to write code that is more readable, maintainable, and portable across different platforms. By translating high-level code into machine-readable instructions, compilers bridge the gap between human-readable languages and the binary language understood by computers, facilitating the development of complex software systems with efficiency and ease.

Modern compilers consist of several stages (Figure 1.2). However, to make the process of development even easier, all of the stages are usually divided into substages, where each of them plays their role. Such modularity gives a lot of space for reuse – to create a compiler for a different machine, there is no need to create a new compiler – Front and Middle ends can be used the same. Only the Back end will be replaced with a different one, generating the target code for a different ISA.

This chapter describes techniques currently used in compiler construction. It will observe the usual structure of modern compilers, what each stage is doing, and which techniques are used.

2.1 Front End

The front end is the first part of the compiler. It is responsible for processing the input, the *source language*, and making it into a form of IR for the rest compiler. This conversion process may occur straightforwardly if each language keyword or construct corresponds directly to a form in the IR. However, if the IR instruction is absent, it requires transforming into a series of IR instructions that produce the equivalent outcome.

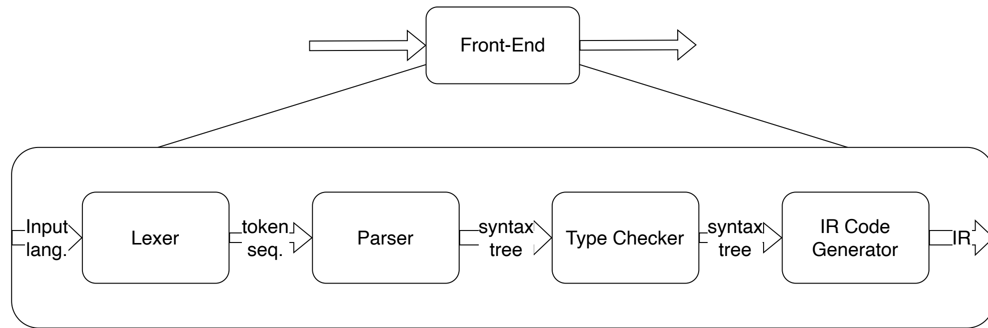


Figure 2.1: Structure of the Front End

Front-end with its task to translate the source program into IR usually separated into parts:

- **Lexical Analysis :** It reads input and transforms it into a sequence of the tokens, defined by the input language.
- **Syntax Analysis :** Takes tokens created before and represents the program in the form of an AST.
- **Semantic Analysis :** The task is to check the correctness of the AST.
- **IR Generator :** After the check, correct AST is translated into an IR, which will be used by other compiler modules.

2.1.1 Lexical Analysis

Input for the compiler is a stream of characters. *Lexer* (part of the compiler, which is doing lexical analysis) reads all characters one by one and groups them into *tokens*. *Token* is a basic lexical unit, representing one semantic item of the input language. It might be a keyword (while, if), number (5, 765), identifier (i, vectorOfNumbers), etc. A token consists of two parts: name, a list of which is predefined in the compiler, and an optional attribute value.

After processing and grouping, the lexer outputs the sequence of the tokens. This sequence is sent forward to the next stage(Figure 2.1).

```
1 while i < 5 {
2   i++
3 }
```

(a) Input program

```
1 tok_while
2 tok_identifier(name = i)
3 tok_less
4 tok_intNumber(value = 5)
5 tok_figureBrOpen
6 tok_identifier(name = i)
7 tok_doublePluss
8 tok_figureBrClose
9
```

(b) Sequence of tokens

Listing 2.1: Lexer translation of the characters into tokens

On this step, there is no possible errors and compiler cannot fail. Lexer does not know anything about syntax. And if tokens going to be in the wrong order, Lexer will generate them in given order and sent on for the next stage.

2.1.2 Syntax Analysis

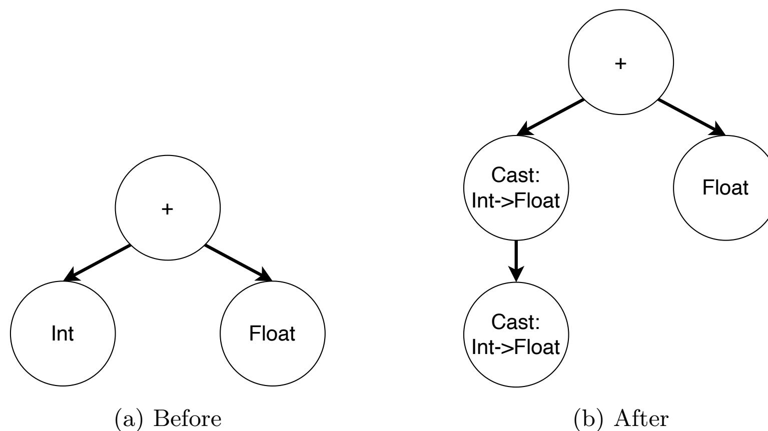
The second stage of the compiler's front-end is syntax analysis. Block named *Parser* is responsible for this. The parser uses a token stream from the lexer and creates a tree-like intermediate representation, usually keeping the grammar structure of the language. Such tree representation is called *syntax tree* in which a tree node is an operation in an original language and children of the node are his arguments. Such structure called an Abstract Syntax Tree (AST).

Also, this is the first and one of the two stages, where the compiler can throw an error due to a mistake made in an input language. Parser verifies the syntactical structure of a token sequence and if it is possible which such order to create a correct syntax tree. If it is not possible, the compiler cannot process it forward and therefore drops compilation at this stage, notifies about the error, and terminates.

2.1.3 Semantic Analysis

Semantic analysis is a crucial phase within the compilation process, serving as the checker in the compiler. Even grammatically correct input might not make any sense. It verifies the correctness of the input program, analyzes the source code, and checks for semantic errors: if the operations support types of arguments, does variables were declared before their use, etc.

But not all mistakes should raise errors. There are some types of mistakes, which can handled by the compiler. Usually, they are called *warnings*. Such mistakes will not cause any trouble to execute them but might lead to unexpected or undefined results.



(a) Before (b) After

Figure 2.2: Shadow cast of the types by type checker

After this phase compiler must not raise any error. Therefore, semantic analysis must ensure, that the program is correct and can be processed by the rest of the compiler. If any critical error is detected, the semantic analysis must

fail the compilation process and notify that about it. However, for non-fatal errors, there might be many approaches. The way to handle such errors is left to the discretion of the compiler. They could be completely ignored, due to their non-critical nature or fail the compilation process, forcing the programmer to create a correct input. Some compiler tries to solve such issues by the pre-build solutions (Figure 2.2).

2.1.4 Intermediate Representation

In a modern compiler, the translation between the programming languages and machine code is not direct. Inside of it, it uses *intermediate representation* (IR). Such representation of the source code is easier to analyze and optimize. Also, it makes the translation process much smoother: instead of transforming high-level languages into a set of instructions, the first translation happens from a complex language to a simpler one, and only after that it generates instructions.

2.1.4.1 Types of IR

IR is not defined by the programming language. Different compilers have different middle languages. All of these IRs also can classified into a different groups. Ussualy, they divided by the form of the representations: *flat*, *hierarchical* (tree or graph) or *stack based*.

Flat IR Flat IR is a low-level variant of the IRs. It is represented as the list of instructions, where they are executed sequentially. Each instruction performs one operation and returns the result to a temporary variable, which others will later use. Labels and branches implement the control flow in such a variant.

Such representations are often favored for analyses and optimizations. Flated IR more straightforward compared to a hierarchical representation. It is much easier to traverse and analyze linear structures, than graphs. Also, memory efficiency is higher. In

Tree-Based IR Another variant of the IR is a *tree-based*, where all instructions are represented as tree nodes and all arguments stored as subnodes. Such representation is similar to an AST, rather than machine code, but the operations are much simpler.

The advantage of such representation is its similarity to the source language. Tree-based IR can express most of the semantics from the input language. It also allows the reuse of the type system and variables from a source language.

Stack-Based IR As the name implies, stack-based IR uses the stack for data manipulation. In such a paradigm, the instruction does not have arguments as children, like in tree IR, or links to another instruction, like in flat IR. They take values from a stack, operating *pop* on a stack, which gives the most recent value from it. When the result is calculated, it *pushes* it to a stack, from which it later can be taken by another instruction.

Such an approach allows us to avoid the use of the complex addressing system, straightforwardly picking and storing all values on a stack, using only

two operations. Also, this decreases the instruction set of the IR, which also can be called the advantage of such representation.

2.1.5 IR Generator

The last step of the front end – is to generate the IR from an AST and pass it to the rest part of the compiler. This step can be performed only after a semantic analysis, which will guarantee, that the program is safe. However, it also can be performed simultaneously. Semantic analysis recursively goes through the whole AST and verifies each node. After the node is verified, it can be translated into a corresponding IR instruction.

Also, the process of generating the IR can be named as *lowering*. During it, the compiler changes the program from the form of the syntax tree into the IR. But not only the form is changed. With the switch of the level, the compiler also loses some information, regarding the program. For example, a type behind the pointers. Instructions do not know what are they dereferencing and what type they are getting. However, the correctness of any operation has been guaranteed by the type checker. Therefore, even without knowing the type explicitly, we can be sure that all instructions will work correctly.

2.2 Middle End

The middle-end part (also referred to as an *optimizer*) is an inner part of the compiler. Its primary purpous is to improve the efficiency and performance of the generated code. Optimization techiques applied at this stage can range from simple transformations of the single instructions to complex analyses of the whole program and its flow.

One of the key challenges in developing a middle end is finding the right balance between the aggressiveness of optimizations and the time taken to perform them. While more complex optimizations can lead to substantial performance gains, they also might increase compilation time, which may not be acceptable in all scenarios. Thus, compiler designers often need to make trade-offs to ensure that the optimizations applied in the middle end strike an optimal balance between compilation speed and code quality.

The most popular optimizations, which used in modern compilers are:

- **Dead Code Elimination :** Dead code elimination is a compiler optimization technique aimed at removing code that is never executed or whose results are never used. This includes variables, functions, or entire code blocks that have become unreachable or redundant due to program transformations or conditional branches. The process of dead code elimination typically involves analyzing the program's control flow graph to identify code paths that cannot be reached during execution. This can occur due to conditional statements that are always true or false, unreachable code after return statements or exceptions, or variables that are assigned values but never used[3, 4].

```
...
  %a = alloca i32
  br label %end
  %add = add nsw i32 7, 6
  store i32 %add, ptr %a
end
  %a.val = load i32, ptr %a
...
```

(a) Before

```
...
  %a = alloca i32
  %a.val = load i32, ptr %a
...

```

(b) After

Listing 2.2: Example of elimination of the non-reachable code

- **Constant Propagetion :** Some variables do not depend on the input and are already pre-defined with the constant values before the compilation. In such a case, it is possible to eliminate the variable call and replace it with the constant, reducing the memory use and streamlining the program's execution. The process typically begins by analyzing the program's control flow and data flow to determine which variables hold constant values at specific points in the code and if so, replace all occurences of it with a constant value[3, 4].

```
%a = alloca i32
%add = add nsw i32 7, 6
store i32 %add, ptr %a
%a.val = load i32, ptr %a
```

(a) Before

```
%a = alloca i32
store i32 13, ptr %a
%a.val = load i32, ptr %a

```

(b) After

Listing 2.3: Example of the constant propagetion in the code

- **Function Inlining :** A function call is an expensive process, requiring a pass of the parameters, stack manipulation, instruction pointer manipulation, etc. Inserting the function code directly into the calling function can result in faster execution and smaller code size. Inlining is particularly effective for short and simple functions, as the overhead of function call setup and termination can outweigh the actual computation performed by the function. However, inlining larger functions can increase code size and potentially degrade performance[5].

- **Common Subexpression Elimination :** Common subexpression elimination is aimed at reducing redundant computations by identifying and eliminating duplicate expressions within a program. In many programs, the same computation might occur multiple times, resulting in unnecessary overhead. Eliminator works by analyzing the code to identify expressions that produce the same result and computing them only once, storing the result in a temporary variable. Then, wherever the expression occurs subsequently, the compiler replaces it with a reference to the temporary variable. This optimization reduces both computation time and memory usage[6, 4].

```
1 %a = alloca i32
2 %b = alloca i32
3 %add1 = add nsw i32 7, 6
4 store i32 %add1, ptr %a
5 %add2 = add nsw i32 7, 6
6 store i32 %add2, ptr %b
```

(a) Before

```
1 %a = alloca i32
2 %b = alloca i32
3 %add1 = add nsw i32 7, 6
4 store i32 %add1, ptr %a
5 store i32 %add1, ptr %b
6
```

(b) After

Listing 2.4: Example of the common subexpression elimination

2.3 Back End

The back end is the last part of the compiler. It is responsible for producing the result of the compilation – the target code. As an input, it takes an IR produced by the previous parts. Mainly, the last part can be divided into two big blocks:

- **Instruction Selection :** IR program should be translated into the target code. To do so, for each instruction (or sequence of them) compiler should find the set of machine code instructions, with the same semantics.
- **Register Allocation and Assignment :** While the Instruction Selection translates an IR program into a machine code, it makes it with the assumption, that it has an unlimited number of registers. Physical machines are limited by the universe and cannot have infinite registers. The register allocator decided, which registers should be in work and which should be spilled into the MM.

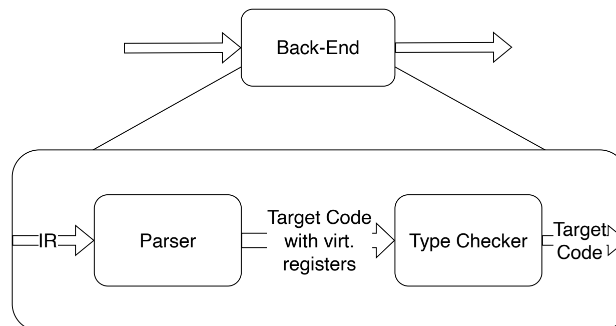


Figure 2.3: Structure of the Back End

2.3.1 Instruction Selection

After the creation and optimization of an IR code, it needs to be translated into a target code, which will be executed on a machine. This task requires creating a mapping from an IR instruction into a target language. Implementation of it is not a trivial task, because not all IR instructions can be mapped one-to-one with target instruction. Also, it could be that different sets of target instructions have the same behavior and correspond to one IR line(Figure later).

The instruction selection problem can be considered as a pattern-matching problem, since each pattern of IR instructions will correspond to a target code with the same semantics. The procedure of selection can be divided into two subproblems[7]:

- **pattern matching** – find all possible blocks of a target code with the same semantics, as the input IR code.
- **patern selection** – choose the most optimal block from others.

To solve it, there are various of different approaches. The most straightforward way: each IR instruction has a pre-defined target code. It is very easy to implement, but it will create a lot of redundant loads and store instructions. Another way is to use advantage, that instruction selection defined as a matching problem, and try to solve it in this way. Formalizing these two methods, two main approaches were created:

- **macro expansion** – selector takes the IR code and list of pre-defined skeletons of the target code and compares them until they match.
- **covering (tree, DAG or graph)** – selector covers IR code with patterns with different shapes, trying to find the optimal one.

2.3.1.1 Macro Expansion

Due to its straightforward nature, this method was the first one, which was used. Even nowadays some compilers, such as GCC, still use this approach, but combined with powerful heuristics.

The macro expansion deals with instruction selection as a separate problem: it goes through all IR instructions and matches each with an already predefined set of macros. Whenever it finds a match – it uses matched target code (macros). Therefore it is a greedy algorithm – it produces solutions locally and does not see the whole picture. Such an approach makes generation fast and easy to implement.

However, making it in such a way will cause a lot of problems. While a greedy algorithm finds a solution locally, globally it might be not optimal at all, and the general quality of the code will be low. It will produce a lot of redundant loads and store instructions and will be inefficient. Such issues could be solved by additional optimizations. Or, to avoid them at all, another approach could be used.

2.3.1.2 Graph Covering

Assuming, that IR code can be represented as a graph, the graph covering method can be used. The idea of this method is to "cover" a graph with patterns, such that every node is under exactly one pattern. For that, the compiler is required to have a built-in set of patterns, which will cover the graph. Not hard to see, that macro expansion can be viewed as a special case of the covering – all patterns is a one node size and each node translated separately.

Also, covering processes might be simpler for a certain types of graphs. One such case - the tree. Thanks to the shape, each node has exactly one parent,

which omits potential overlaps. In addition, it will be much easier to process it, due to the possibility of using dynamic programming to traverse the graph and covering with the local-optimal solutions.

2.3.2 Register Allocation and Assignment

During instruction selection, registers were out of scope. Assignments, store, and load operations were executed above abstract registers. An approach not to do anything and keep everything in a MM, loading before each operation and storing it back after, could be used. However, it would significantly affect the speed of a program since such operations are very costly[4]. Therefore, registers must be used. They are the fastest computational units. But the number of registers is limited and not all values can be stored in them. So, correct and efficient usage of them is an important task.

To simplify the task, it can be subdivided into two problems:

- **Register allocation** – find out, which variables temporarily will be stored in registers.
- **Register assignment** – which register will be used for a variable at the exact time.

Unfortunately, the optimal solution for this task is classified as a **NP**-problem and cannot be solved in a reasonable time[4]. Therefore, all solutions, that are going to be described lately, are heuristic methods, which create reasonable solutions in a reasonable time.

2.3.2.1 Linear Scan Register Allocation

Linear scan register allocation is a linear algorithm, that greedily allocates registers. It uses a *liveness analysis*[8], computing the live interval for each register and computing the local optimal solution for such intervals[9]. Such an algorithm works very fast, in linear time, not requiring a lot of pre-work to be done. Price for it is the less optimal solution, than other approaches.

2.3.2.2 Register Allocation by Graph Coloring

Another way to solve this problem is to rephrase it to another one: each virtual register is a node and they somehow need to be painted into a limited number of colours[10, 11]. Because K graph coloring is a **NP**-problem, an optimal solution cannot be created at an appropriate time. Therefore, the algorithm for it is based on various of different heuristics. For them to work correctly, different preparations and pre-analyses are required. One such is a *liveness analysis*[8], which collects information about each register life span.

2.4 Existing Compilers

In terms of studying compiler construction, it's appropriate to talk about existing modern compilers. There are plenty of them, for different languages and for different machines. The two main compilers, that would be worth talking, are the GCC (C Compiler) , and the LLVM. These two compilers have a modular

construction [12, 13], which makes it easier to study each stage of it, because of their independence.

2.4.1 GCC

The GCC (GNU Compiler Collection) is a suite of compilers for various programming languages, primarily C, C++, and Fortran. It is has been developed in 1987 as a free compiler under the public[14]. GCC compiler is one of the most widely used compiler toolchains in the world, known for its high performance and extensive optimization capabilities.

IR

For better optimization, the GCC compiler uses several different IR, and executes different optimizations on a different representation, that is better suited for it. In total, GCC uses three IR:

- **GENERIC**, which plays a role as a high-level representation for a front-end. It is a tree-based IR and has the same form, as the AST of the source language.
- **GIMPLE** is created from a GENERIC tree. It serves as a true internal language, being in the middle between the front end and back end. Most of the optimization is applied to this representation, due to its flat nature. Generates the RTL after itself for a back end.
- **RTL** serves as a low-level representation. It represents the program in terms of register-level operations and is closer to the machine language level compared to others IRs.

Back End

The back end of the GCC, as has been stated before, uses RTL. Before it is converted into a machine code, the compiler runs its final operations to make the output better. After the RTL generation from the GIMPLE, the compiler performs the last optimizations, such as common sub-expression elimination, peephole optimization, tail call elimination, etc. Next, it starts allocation registers. RTL originally uses unlimited virtual registers, which cannot be represented on a virtual machine. In the end, it changes the RTL instruction with the target machine one and outputs the final result into a binary file, which can be executed later.

2.4.2 LLVM

The LLVM (Low-Level Virtual Machine) is a compiler infrastructure project designed for the optimization and compilation of the programming languages. Originally developed at the Unniversity of Illinois, LLVM has evolved into a into a comprehensive system that includes various components.

Such components is a key feature of LLVM. it has a modular design, which allows developers to easily add support for new programming languages and target architectures. It's widely used in both academia and industry, serving

as the foundation for various programming language implementations (such as Swift, Rust, GoLang, etc.) and tools (compiler frameworks, static analyzers, etc.).

IR

LLVM IR is designed to be platform-independent and expresses code in a form that's close to machine code but still retains some high-level language constructs. It is a very powerful tool and one of the biggest advantages of the compiler – it represents the semantics of the source program in a way that's easy for optimization and code generation passes to manipulate.

```
define i32 @main() #0 {
  %a = alloca i32
  %b = alloca i32
  %c = alloca i32
  %a.val = load i32, ptr %a
  %b.val = load i32, ptr %b
  %add = add i32 %a.val, %b.val
  store i32 %add, ptr %c
  %c.val = load i32, ptr %c
  ret i32 %c.val
}
```

Listing 2.5: Program written on LLVM IR

It has a flat representation in a SSA form, which gives plenty of room for various optimization and analyses to produce a better output. It applies such algorithms as loop optimizations, function inlinings, strength reductions, dead code eliminations, and many others.

Back End

Just like GCC, LLVM also has several layers of IR. The lowest one is a *LLVM bitcode*, which is a target-independent machine code representation. LLVM bitcode is a portable binary format that represents the program's semantics and control flow in a platform-independent manner. This bitcode is then used as input to the target-specific code generation phase, where it creates a machine code for a specific target architecture.

Chapter 3

Compiling Pragramming Languages

Programming languages are formal languages designed to communicate instructions to a computer. They provide a means for humans to write code that computers can understand and execute. Each programming language comes with its own syntax, semantics, and set of rules, which govern how instructions are written and interpreted. These languages serve as a bridge between human thought processes and machine operations, allowing programmers to develop software and algorithms to solve various problems.

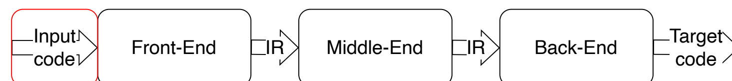


Figure 3.1: Programming language as the input on the scheme

The evolution of programming languages has been influenced by factors such as technological advancements, user requirements, and programming paradigms. As a result, new languages continue to emerge, each aiming to address specific needs or improve upon existing languages. Moreover, programming languages play a crucial role in shaping the way software is developed, maintained, and scaled in various domains, including web development, artificial intelligence, and scientific computing.

One fundamental aspect of programming languages is their ability to express algorithms and manipulate data structures. This capability enables programmers to create complex software systems ranging from simple scripts to large-scale applications. Additionally, programming languages can be classified into different paradigms, such as imperative, functional, and object-oriented, each offering unique approaches to solving computational problems. Another way to classify languages is by how they are translated. One such class is languages, processed by the compiler. This class of languages is called *compiled programming languages*.

Compiled programming languages are those where the programming language is translated into instructions before execution. This translation typically made using a compiler, which converts a *source language*, into a *machine code* that the computer's processor can directly understand and execute.

One of the primary advantages of compiled languages is their efficiency, as the compiled code usually runs faster than interpreted code since it's already

translated into machine language. Examples of compiled languages include C, C++, and Rust, which are known for their performance and versatility. Despite the initial overhead of compilation, compiled languages are favored for system-level programming, high-performance computing, and applications where speed is crucial. Another key benefit of compiled languages is their ability to produce executable files, which can be run on any compatible system without requiring the source code or an interpreter. Additionally, compiled languages often offer strong static typing and compile-time error checking, which can help catch bugs early in the development process and improve code robustness. Also, compiling languages

3.1 C

The C programming language stands as one of the most foundational and enduring languages in the realm of computer science and software engineering. Developed by Dennis Ritchie at Bell Labs in the early 1970s, C was born out of a necessity to create a flexible and efficient tool for system programming. Its design principles emphasize portability, efficiency, and a clear, concise syntax, making it a versatile choice for a wide range of applications, from operating systems to embedded systems and everything in between.

```
1 int arr[100];
2 ...
3 for (int i = 0; i < 100; ++i)
4   for (int j = 0; j < 99; ++j)
5     if (arr[j + 1] < arr[j]){
6       int tmp = arr[j + 1];
7       arr[j + 1] = arr[j];
8       arr[j] = tmp;
9     }
```

Listing 3.1: Example of the program written on C

At its core, C is a structured, procedural language, meaning that it follows a logical flow of control through functions and code blocks. It eschews the more abstract and complex features found in later languages in favor of simplicity and directness. This simplicity, however, belies its power; C provides low-level access to memory through pointers, enabling developers to manipulate data and resources at a granular level, a capability crucial for tasks like memory management and hardware interaction.

One of the defining characteristics of C is its close relationship with the underlying hardware of a computer system. Unlike higher-level languages that abstract away many details of the machine, C offers a level of control that allows programmers to optimize their code for performance and efficiency. This closeness to the hardware, coupled with its minimalistic syntax and powerful features, has made C the language of choice for building many of the foundational components of modern computing, including operating systems, compilers, and device drivers.

Meanwhile, it is a key feature of the language, but it is also a big disadvantage of it. Being a low-level, C does not have much pre-build functionality. Lack of abstraction in many operations requires for programmer to manually control them, while a compiler could easily handle it. For example, in doing

concurrent programming in C programmer must do everything by himself, using POSIX library *pthreads*, where each move must be done by the programmer manually.

3.2 GoLang

Go, often referred to as Golang, is a statically typed, compiled programming language designed by Google. It is influenced by languages like C and Pascal but aims to provide a simpler and more efficient way to build software, especially for large-scale systems. It was developed as an addition to C/C++, improving the weak sides of these languages, which Google found critical for itself.

```
var arr [8]int = {1, 4, 6, 9, 11, 12, 17, 22}
var l, r, resutl = 0, 7, -1
for ; r > l ; {
  m := (l+r) / 2
  if arr[m] == n {
    result = m
    break
  }

  if arr[m] > m {
    r--
  } else {
    l++
  }
}
```

Listing 3.2: Example of the program written on Go

The philosophy behind the GoLang revolves around **simplicity** and **efficiency**. For simplicity, Go has a very minimalistic and intuitive syntax, making it easy to understand programs written on it and to learn it. The language avoids unnecessary complexity and features, that could lead to ambiguity or confusion. Efficiency of the Go is manifested in its high level. Being a not true OOP language, it still supports it and allows programmers to avoid the reinvention of the wheel.

Even Go was inspired by C, the only thing it got from it is syntax. These two languages look very familiar when they are written side by side **??**. It allows programmers in C/C++ to easily understand programs written in Go and to learn it. On the other hand, Go is a high-level programming language with all the ensuing consequences.

The main feature of the language is concurrent programming. The whole language was built around the idea of making use of multi-threading programming as easy as possible. To call a *gorutine*, a thread in Go, the programmer just needs to write a keyword *go* before the function call. For communication between gorutines, *channels* is used. It is a mechanism to synchronize thread execution. It provides a way for one goroutine to send data to another goroutine in a safe way.

3.3 Comparison of the C and GoLang

GoLang was created with an eye to C and looks very familiar. However, it is a different language and even if it looks the same, it has a lot of differences, which makes them not as similar as it may seem at first glance.

3.3.1 Syntax

These two langauges have a very similar syntax. This was done specifically, to facilitate learning a GoLang for people, who already knew how to program on C/C++. However, these are still two different languages, created in a different time.

3.3.1.1 Basic Grammar

The most basic statements, such as for loop and if-*else* branching, are almost identical. In both languages, for loops consist of 4 main parts: *init-expression*, usually containing the declaration of the iterated variable, condition expression, lopp-expression, and the body. The first real differences can be noticed in the variable declaration. In Go, each variable declaration (Listing A.2) starts with the keyword *var* (Listing 4.2a). Also, if the user provides an initial value for a variable, the specification of a type can be omitted. In that case, the variable will take the type of value, which is assigned to it. At the same time, in C variable declaration starts with its type (Listing 4.2b) and it cannot be omitted, even if an initial value is provided.

```
1 // value of a is 0
2 var a int
3
4 var b = 1.2
```

(a) GoLang

```
1 // value of a is unknown
2 int a;
3
4 float b = 1.2;
5
```

(b) C

Listing 3.3: Example of the variable declarations in languages

Another noticeable difference is a semicolon. While C requires them after each statement, in Go they are optional: a new line plays the role of a separator. But if multiple statements have to be written in one line – semicolon should be put between them.

3.3.1.2 Functions and Methods

Another major difference is in methods and functions. In C++ [2] and in the Go, functions can be declared only on the top-level and it is not possible to create them inside another function [3]. Functions in a C++ start with the type, which it returns. After it is followed by the name of the function. Next, arguments of the function are declared in brackets (Listing 3.4b). At the same time, GoLang

[2]Because in C there are no methods, in this comparison, we will use C++, which is an extension of the C language

[3]We do not count lambda functions in this context

has a slightly different order of function signature. It starts with the keyword *func* and is followed by the name of it. The return type is declared at the end, right before the body of the function. Also, it is possible not to specify the type for each argument and group them up (Listing 5.3a,A.3).

Methods in both languages are also slightly different. The main distinction is in the place, where methods are declared. In Go, methods are the same, as regular functions – they are declared on the top level. The distinguishing feature of the method is a special signature right before the name. In the brackets programmer specifies the inner name of the structure, which will be used inside the method to refer members of it, and for which type the method provides new behavior. Such type can be either the structure itself, or a pointer to it (Listing 5.3a). In C++, methods are located inside the structure, for which it creates a new behavior. In that case, no additional signatures are needed and methods look identical to the functions (Listing **??**).

```go
//function
func sum(a,b int) int {
  return a + b
}

// structure with methods
type student struct{
  age, grade int
}

func (stud *student) setAge(
    new_age int){
  stud.age = new_age
}

func (stud student) getAge()
    int {
  return stud.age
}
```

(a) GoLang

```c
//function
int sum(int a, int b){
  return a + b;
}

//structure with methods
struct student {
  int age, grade;

  void setAge(int new_age){
    age = new_age;
  }

  int getAge(int new_age){
    return age;
  }
}

```

(b) C

Listing 3.4: Example of functions and methods in languages

In addition, Go allows to return of multiple values from a function (Listing 3.5). To do it, instead of providing a function with just one return type, the programmer must write multiple types in brackets. In that case, the function will be able to provide a value for multiple variables on the left side of the assignment. However, the return type is not a tuple – such functions cannot be assigned to a single variable or being used in an expression. The only way to use it is in a direct assignment, where a number of variables on the left side is the same, as the number of return values.

```go
func f() (int,int,int){
    return 1,2,3
}
...
var a,b,c = f()

//ERROR. Not allowed
```

```
8 var d = f()
```

Listing 3.5: Example of multiple return values

3.3.2 Static Typing

Typing in programming languages refers to the system used to define and enforce the data types of variables and expressions within the language. It determines how the programming language treats different types of data, such as integers, strings, floating-point numbers, etc. Primarily, there are two types of typing systems: *dynamic* and *static*.

In dynamically-typed languages, variables are not explicitly declared with a data type. Instead, the type of a variable is determined at runtime based on the value assigned to it. Dynamic typing allows for more flexibility but can also lead to errors if types are not handled carefully. Examples of dynamically-typed languages include Python, JavaScript, and Ruby.

A statically typed language is one in which variable types are explicitly declared at compile time and checked for consistency before the program is executed. This means that you must declare the data type of each variable when you define it, and once defined, the variable cannot change its data type.

GoLang and C are both classified as statically typed. In both languages, type checking typically occurs during the compilation phase, where the compiler analyzes the code to ensure that operations are performed only on variables of compatible types. This helps catch many potential errors before the program is run, reducing the likelihood of type-related bugs.

3.3.3 Concurrency

Concurrent programming is a paradigm in computer science that deals with the execution of multiple tasks (or processes) at the same time. Unlike sequential programming, where tasks are executed one after the other, concurrent programming allows tasks to overlap in time, potentially running simultaneously.

Both languages support it. GoLang has built-in tools for it, such as *goroutines*, lightweight threads, channels, and synchronization primitives like mutexes and wait groups. This makes it easier for developers to write concurrent programs without dealing with low-level details. At the same time, C does not have such things. For concurrency, it uses a POSIX library *pthreads*, which deals with threads on the low level, where the programmer controls everything by himself, giving room for errors and memory leaks.

3.3.4 Memory Management

Memory management in C and Go presents a contrast reflecting their respective language design philosophies. C, being a low-level language, demands manual memory management, where everything is controlled by a programmer and he is responsible for making everything safe. It gives control over memory allocation and deallocation but increases the risk of memory-related bugs. On the other hand, GoLang, being a high-level language, manages the memory automatically. It decides on its own, whether variables can be allocated on a stack, or should be placed in a heap. Also, it contains a garbage collector in its compiler, to handle memory deallocation seamlessly.

3.3.5 Error Handling

Error handling is a critical aspect of programming languages, allowing developers to manage unexpected situations that may occur during program execution. Different programming languages offer various mechanisms for handling errors. Here are some common approaches:

- **Exceptions :** When an error occurs, an exception is raised, and the program looks for an exception handler to catch and deal with it. This mechanism separates the error-handling logic from the regular code flow, making the code cleaner and more manageable.
- **Return Values :** Particularly popular around functional languages. Functions return either a result or an error code, and the caller must check the return value to determine if an error occurred.
- **Error Codes :** Functions return a special value (typically -1 or NULL) to indicate an error condition, and the caller must check this value and handle the error accordingly.

In C programming, error handling is commonly done using error codes and return values. In the first case, there are global variables. If during the function execution error appears, such a variable is set to an error value. After the function, the caller checks if any error occurs and behaves on the value of a variable. In the second case, the function may return specific values to indicate error conditions (Listing 3.6b).

By the guideline, in Go, programmers should handle errors using return values. It is not hard, especially using the Go feature, which allows to return of multiple values at the same time (Listing 3.5). Any function along with a result, if it exists, returns a special variable of a type *error*, which can be checked after the call of the function (Listing 3.6a).

```go
func div(a,b int) (int,error){
    if b == 0 {
    return 0, errors.New("div by zero")
    }
    return a / b, nil
}
...
result, err := divide(a,b)
if err != nil{
  ...
}
```

(a) GoLang

```c
//0-OK, 1-division by 0..
int error = 0;

int div1(int a, int b){
  if (b == 0){
    error = 1;
    return 0;
  }
  return a / b;
}
int div2(int a,int b,int* res)
     {
  if (b == 0)
    return -1; // ERROR.
  *res = a / b;
  return 0; // SUCCESS
}

```

(b) C

Listing 3.6: Error handling in languages

3.3.6 Top-level Declarations

As stands on the official GoLang webpage language specification[15]:

> The scope of an identifier denoting a constant, type, variable, or function (but not method) declared at top level (outside any function) is the package block.

This means, that all variables, types, constants, and functions, declared on the top level (globally) are declared for the scope of the whole file and can be accessed anywhere, even before its place of the declaration in the file (Listing 3.7).

However, it creates a problem of loop dependencies, where one variable might depend on another, which depends on the first one. Language compiler can detect it and programs with such a problem will not compile.

```
...
// OK
int a = b
int b = 12

// ERROR
int c = d
int d = c
...
```

Listing 3.7: Example of GoLang top level declarations

3.3.7 Object-Oriented Programming

3.3.7.1 Polymorphism

Polymorphism refers to the ability of different types to be treated as the same type in certain contexts. This allows code to be written in a more generic way, where different types can be used interchangeably as long as they adhere to a common interface.

C++ and GoLang have radically different approaches to achive the polymorphism[4]. C++ achive it through the *function overriding* (Listing 3.8b). This occurs when a derived class provides a specific implementation of a method that is already defined in its base class. The function in the derived class ”overrides” the function in the base class.

In Go polymorphism is achived through interfaces. In such case, different types can be treated as the same interface type, allowing for code flexibility and reuse (Listing 3.8a). An interface defines a set of methods, and any type that implements those methods satisfies the interface implicitly. This allows different types to be treated uniformly if they fulfill the requirements of the interface.

[4] C does not support OOP. Because of that, comparison is going to be with a C++, which is a C language with an extra features and extensions

```go
type human interface {
  getAgeStatus() string
}

type student struct {
}
type teacher struct {
}

func (s student) getAge()
    string{
  return "Young"
}

func (t techer) getAge()
    string{
  return "Old"
}
```

(a) GoLang

```c
struct human {
  virtual string getAge();
}

struct student : human{
  string getAge(){
    return "Young";
  }
}

struct teacher : human{
  string getAge(){
    return "Old";
  }
}

```

(b) C

Listing 3.8: Example of the polymorphism in languages

3.3.7.2 Inheritance

C is not an OOP language and in this scope, GoLang is much ahead. However, both languages do not support inheritance in the full sense of this word. In C it is only possible to have an "inherent" structure as a member (Listing 3.9b). In Go, such a thing is called *composition*. It allows not to specify the name to an "inherent" member. But the use of it will still require access through the name of the type (*student.human.age* (Listing 3.9a)).

```go
type human struct {
  age int
}

type student struct {
  human
  grade int
}
```

(a) GoLang

```c
typedef struct {
  int age;
} human;

typedef struct {
  human human;
  int grade;
} student;

```

(b) C++

Listing 3.9: Example of the member inheritance in languages

Chapter 4

Design And Implementation

This chapter first will talk about the TinyGo and its specifications: how it looks like and the differences with a GoLang. Then, we will discuss the compiler creation: its design, what choices have been made, and how they are implemented.

4.1 TinyGo

TinyGo was created for an educational purpose to study the compiler construction. Modern high-level languages are overwhelmed with various of different features and syntactic sugars, that might be not too important in terms of studying code generation. TinyGo solves it, preserving the most important design principles behind Go and, at the same time, omitting not-so-valuable features.

```
1  func fib(n int) int {
2      if n < 1 {
3          return -1
4      }
5      if n == 1 || n == 2 {
6          return n
7      }
8      var a = fib(n - 1)
9      var b = fib(n - 2)
10     return a + b
11 }
```

Listing 4.1: Function written on a TinyGo, that computes the nth fibonacci number

4.1.1 TinyGo Specifications

To preserve a TinyGo to be a subset of the Go and not just a similar language, it was very important to save the main design principles of the GoLang (4.1.2.1). One such principle is simplicity. This is expressed in the grammar of the language. TinyGo fully repeats the syntax of the original language and utilizes the same grammar. The grammar of the TinyGo is included in appendix B.

Alongside simplicity, another major principle is efficiency. TinyGo preserves it as much as possible, giving the program a variety of supported syntactic

sugars and high-level language features, such as OOP features. Unfortunately, the time is limited and it was not possible to implement everything. Some aspects of the language have been removed, even the crucial ones, such as concurrent programming.

4.1.2 Omitted features

TinyGo was created for educational purposes to study languages and compiler creation. Original GoLang is a big language, created over the years by many people. It has a huge amount of different features, created through time. Although a lot of them are useful, not all of them are too important in the case of studying compiler construction.

Some of such features are just syntactic sugars and do not mean a lot. Some of them might be important, but the time it would cost to implement them does not correspond to the value they will bring. List of them is presented below.

4.1.2.1 Concurrent programming

It is a big and important part of a GoLang (Section). But this feature depends mostly on the target machine rather than on the compiler. Unfortunately, Tiny x86 does not support multi-threading and it is not possible to make it work.

4.1.2.2 Interfaces

Interfaces is a special mechanism, that allows classes to hide implementation and internal fields, leaving only relevant information to the outside world (Section 3.3.7.1). Such feature is an important part of any Object-Oriented language, such as GoLang. But it is very hard to make the compiler support it. It would require an enormous amount of time to make it. Because of the lack of time and complexity of it, TinyGo does not support interfaces.

4.1.2.3 Generic programming

Generic programming is a programming paradigm, which allows to write code without specifying the type of the variables and leaving it for a compiler to deal with. Addition of the generics into the language would not change the compiler substantially. It would be nullified on the stage of IR by the front end. But it would take a lot of time implementing it, not just changing the parser, but adding the block to generate a new code for genetic functions. Therefore, generics is not supported by the compiler.

4.1.2.4 Garbage collector

It is a memory management unit. It reclaims memory, that has been allocated by the program and is no longer referenced by any pointer or value in a program. Such a feature is a key one in languages without a manual memory control because it makes sure, that all memory allocated by a user is going to be cleared and the user does not need to think about it. Implementation of such a feature is too complicated and would require to creation of whole new units, which must to taken care of. This thesis focuses on the basic compiler construction

and due to its nature, does not go into the most deep insides of the code generation. Also because of the lack of time, the garbage collector does not implemented in the compiler.

4.2 Design

The 2 chapters have provided all necessary information, to make a compiler design. In this section, we will talk that the design solution being used to create a compiler and why. To ensure the modularity of the compiler, just like in modern ones, all parts were designed separately and do not depend on each other.

4.2.1 Front-End

4.2.1.1 Lexer And Parser

The first two blocks of the compiler are merged and working together. This was done in order not to store the result of the lexer in the memory. Working simultaneously, the lexer generates a new token only if it is asked by the parser. Such a design allows a decrease in the memory and time complexity and creates an AST right after reading the input, skipping part of generating the sequence of tokens.

Lexer block works only with the file. After receiving the signal, it greedily starts reading the character stream from the file, until it will not form a token. Next, it returns it to a caller. Meanwhile, the parser does the more global task, parsing the language grammar with the tokens. With getting new tokens, the parser matches them with grammar, expanding the rules or doing the comparison operation.

With the expansion, the parser creates a new node for an AST. It is a child of the node, in which rule the expansion happened and it's an optional child going to be any expansions made from the current rule.

4.2.1.2 Type Checker

In the AST, the type checker recursively visits and verifies the correctness of the built tree. It starts from the root of the tree and traverses all of its children. It also has its own space for internal information, called $IRContext$. This space has all the information on the current situation, while it passes over the AST. It stores the declared variables, named types, return type, loops stats, etc.

For the program level (in the root node), the checker runs twice for each declaration. It is not possible without it, because the top-level declarations in the TinyGo work without the order (Section) and to process them, we must sort. On the first run, it collects information about the declarations and their dependencies. For the type, dependence is a custom type, which is allocated inside. It is not possible to allocate size for the type, without knowing the precise size of its member (obviously, this does not apply to a pointer type). For the global variables, dependence is a variable, that stands in the r-value. It would not be possible to determine the value if some of them would form a cycle.

After the information is gained, it is sorted. It uses an algorithm of a topological sort (Listing 4.2). If any variable/type does not depend on anything, it can be declared and should be removed from the right side of other variables/types. This continues until all variables/types are declared or there are no more items, that have no dependencies. The second case means, that there is a loop declaration and, therefore, such a program terminates and reports the fail.

```
1 var a = b + e
2 var b = e
3 var c = 12
4 var d = e * 2
5 var e = c + 17
```

(a) Original language

```
1 var c = 12
2 var e = c + 17
3 var d = e * 2
4 var b = e
5 var a = b + e
6
```

(b) After sorting

Listing 4.2: Topological sort of the variables

Then initial preparation is finished, it can start checking the program. First, it starts declaring types in the order, issued by the previous step. After types, variables, and constants are declared. In the end, it starts to declare the functions and checks the function's body.

The process of verification of the function's body is much easier. Every statement must be executed by order in which it is written and, therefore, there are no forward declarations and the type checker is fairly straightforward here. It visits every AST node and checks for its correctness. Usually, it synthesizes type of the node from its children and then propagates it to be the child for the rest nodes.

4.2.1.3 IR

From the different possible types of IR listed in the section 2.1.4.1, we have decided to choose the flat IR. Such a choice was made because it was easier to optimize. It is going to be crucial, because the instruction selection algorithm, which is going to be discussed later, was chosen the not optimal, and middle-end optimizations can save the quality of the result.

```
1  function %main, arguments: , return: {
2      %a.addr = alloca i32
3      %b.addr = alloca i32
4      %0 = call: scan arguments: %a.addr
5      %1 = call: scan arguments: %b.addr
6      %a = load from: %a.addr
7      %b = load from: %b.addr
8      %sum = binop add first: %a, second: %b
9      %2 = call: print arguments: %sum
10     ret
11 }
```

Listing 4.3: Program in IR for computing the sum of numbers

A whole IR program is a list of functions and each of them consists of a list of instructions inside. Each instruction is a 3-address code because all of them have up to three operands and an argument. The grammar of the IR is presented in the Appendix B.

This IR is inspired by the LLVM one and completely copies its structure. All instructions have their analogs in LLVM IR and are almost identical. This gives a big advantage to an optimizer. There are already implemented plenty of different optimization algorithms for such IR and it is possible to reuse them, without big changes in the input. Also, IR is a low-level, which makes it easier to implement the back end of the compiler.

4.2.1.4 IR Generation

The IR is low-level and cannot represent all possible features of the source language. In the process of converting the TinyGo in the intermediate language, it lowers all such possible features and creates a plaint IR.

```
package main

func init() (int,int) {
    return 1,2
}

func main() {
    var a,b = init()

}
```

```
function %init; arguments: ('ptr' %1), return: 'nothing' {
    %3 = alloca 'ptr'
    store what: %1 where: %3
    %6 = load from: %3
    %7 = get member from: %6; which: 0
    %9 = create int constant 1
    store what: %9 where: %7
    %11 = get member from: %6; which: 1
    %12 = create int constant 2
    store what: %12 where: %11
    ret 'nothing'
}

function %main; arguments: , return: {
    %1 = alloca {i32, i32}
    %2 = alloca i32
    %3 = alloca i32
    %4 = call %init arguments: (%1)
    %5 = get member from: %1; which: 0
    %6 = load from: %5
    store what: %6 where: %5
    %8 = get member from: %1; which: 1
    %9 = load from: %5
    store what: %9 where: %8
    ret 'nothing'
}

```

Listing 4.4: Go function which returns more than one variable in IR

The generator goes recursively through the whole tree and performs the translation. Each node expands to a set of the IR instructions and saves it to a corresponding function. But IR is flated and

4.2.2 Middle end

The middle end has been not included in this work. The lack of time prevents from creating it. However, the TinyGo IR is helping here. It has a similar structure to one of the most bride used intermediate languages, LLVM IR, and it is possible to make an optimization from it. The grammar of both languages is similar and modular construction of the compiler would allow to just put it in the middle. This fact makes the implementation of the middle end not so valuable.

4.2.3 Back end

4.2.3.1 Instruction Selection

For the instruction selection, the macro expansion has been used as an algorithm. It has been chosen as the selector, due to the low-level nature of the TinyGo IR. It is already very near to the target code and generating the target instructions from an intermediate one is a trivial task. Also, being a low-level, IR instructions is almost a target instruction, allowing to describe each of it with the small number of machine ones.

Generation of the instructions is going sequentially, by the same order from the IR. Each instruction has a pre-defined target code, to which it is translated. It's only inserting the register numbers and pointers to a stack in the right place and outputting the result.

4.2.3.1.1 Calling Convetion For the calling convention, we decided to make everything through the stack. Before the function is called, it reserves the place for the return values. After that, it pushes the arguments of the function to it. With the call, it places the return pointer for the return instruction. Also with the return pointer, it stores the previous value of the basic pointer, which will be picked up with the return (Figure 4.1).

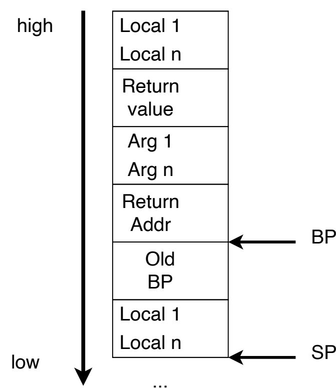


Figure 4.1: Look of the stack

4.2.3.2 Register Allocation and Assignment

Tiny x86 being the VM, unlike the physical ones, supports the unlimited number of registers in itself. These features allow for the compiler to not care about

the spiling of the registers and always carry values in them.

TinyGo compiler uses it and does not implement any register allocation algorithms and uses all of them. It has a module, that is responsible for giving registers to the instructions. But right now the work of it is straightforward – for each new result, that is stored in a register, the allocator gives a new register, which was not used before, and saves information about it. If other instructions ask for a value, it will return the already allocated register with the value in it.

Chapter 5

Evaluation

After the implementation, we have got a working compiler for a TinyGo language. This chapter first will overview the result of the work and what it is doing. Next, we will compare it to other and already-created TinyC compilers to see their differences.

5.1 TinyGo Compiler

The main objective of this thesis was to create a working compiler for TinyGo. Since the TinyGo is a subset of the original Go language, it must support programs written on it, except for some features (4.1.2). It can be tested on the basic examples, taken from the official language specifications.

```go
package main

func main() {
  var a = 0
  if a == 0
    print(a)
}
```

(a) Wrong syntax of the if branch

```go
package main

func main() {
  var a time
}

```

(b) Wrong type

```go
package main

func main() {
  var a int
  var b = &a
  if a == b {
    print(1)
  }
}
```

(c) Unmatched types

```go
package main

func main() {
  var a,a int
}

```

(d) Creates 2 variables with the same name

Listing 5.1: Example of the wrong programs

```go
package main

func main() {
  var a,b int
  scan(&a)
  scan(&b)

  var add, sub, mul, div = a + b, a - b, a * b, a / b

  print(add)
  print(sub)
  print(mul)
  print(div)
}
```

(a) Basic arithmetics

```go
package main

func main() {
  var a, b int
  scan(&a)
  scan(&b)

  if (a == b){
    print(1)
  } else{
    print(2)
  }
}

```

(b) If/else branching

```go
package main

func main() {
  var a = 0
  var b int
  scan(&b)

  for i := 0; i < b; ++i {
    a += 2
  }

  print(a)
}
```

(c) For loops

```go
package main

type student struct{
  grade,age int
}

func main() {
  var a student
  scan(&a.grade)
  scan(&a.age)
}

```

(d) Structures

```go
package main

func main() {
  var a int
  var b *int
  b = &a
  scan(b)
  print(a)
}
```

(e) Pointers

```go
package main

func main() {
  var a [10]int
  for i := 0; i < 10; ++i {
    scan(&a[i])
  }
}

```

(f) Arrays

Listing 5.2: Different programs written to test the compiler

```
package main

func fib(n int){
  if n < 1 {
    return -1
  }
  if n == 1 || n == 2 {
    return n
  }

  var a = fib(n - 1)
  var b = fib(n - 2)
  return a + b
}

func main() {
  var a int
  scan(&a)
  print(fib(a))
}
```

(a) Functions

```
package main

type human struct {
  age int
}

func (h *human) setAge(val int
    ) {
  h.age = val
}

func (h human) getDoubleAge()
    int {
  h.age *= 2
  return h.age
}

func main() {
  var a human
  a.setAge(12)
  print(a.getDoubleAge())
}

```

(b) Methods

Listing 5.3: Example of using functions and methods

First, the compiler must not tolerate the wrong input and fail it without finishing the process. TinyGo meets it, failing the programs, that are written with mistakes, and not compiling them. The parser can detect, if the input does not meet the grammar of the language and the compiler will fail (Listing 5.1a). Even if the input matches with the grammar, it is not the guarantee of the correct program. After that, the type checker takes the job and fails all other problems, such as the unknown type (Listing 5.1b), not comparable types (Listing 5.1c) or declaration of the two variables with the same names (Listing 5.1d).

But the compiler should not only fail the wrong input but process the correct ones. TinyGo compiler also copes with it, accepting the correct inputs, processing them, and outputting the corresponding programs, written in machine code. Examples of the correct inputs can be programs, presented in Listing 5.2. It contains a lot of different programs, which test various of different statements and situations. Also, TinyGo is an Object-Oriented language and, therefore, supports methods (Listing 5.3b).

All of these, and more, tests are located in the same place with the compiler, in the subdirectory called *tests/*, and can be executed before its usage, to test the correctness of the translator.

5.2 Influence of the TinyGo in Compiler

In the first chapter, we were not sure, how the input languages affect the compiler and its structure. In this chapter, we can evaluate the influence of the source language on the compiler and which parts depend on it. We will

discuss, how much the source language affects different parts of the compiler and will compare it to others.

5.2.1 Front end

The part, that accepts and processes the input should be affected by it. Most compilers have the same steps to accepting the language. The only thing that changes is an implementation. Different languages have different grammars and rules. The parser goes through the language grammar and an AST is a language representation in the form of the tree. But all of them are done by the same structure, accepting the tokens, expanding the rules, and creating the tree representation of it. At the end of the part, the front end generates the program in a new language, which does not connect to the source one, and passes it to the next part.

5.2.2 Middle end

After the language is lowered to the IR, it loses information about itself. The IR has the same behavior, but it had nothing to do with the original input, because it already fully translated to another, absolutely different language. The middle end analyzes and optimizes such representation. Therefore, because the IR does not connect to the original language, the middle end also does not depend on it.

5.2.3 Back end

By analogy with the middle end, the back end also is not influenced by the source language. It accepts the IR code and translates it to the target. Because it takes as the input only the middle representation, which does not connect to the input, this part also does not depend on the programming language.

5.2.4 Other Compilers

The degree of influence can be not only analyzed but also compared to already existing solutions. We can try to find the similar by their nature compiler with the different source languages and compare their structure to this one.

We will look to the very similar compilers, that have been written by other students as their master theses [16, 17]. They have the same nature, being the educational projects, processing the languages, that were created also to be an educational stand for studying the compiler construction.

In both works, the authors describe the same components, as have been mentioned in this work. Moreover, both compilers have the same structure, as a TinyGo one. The only difference is in the implementation of the front end. For different languages, different parsers and checkers are needed. But after that, compilers utilize the same structure and have the same components. Having the different IRs does not change the whole picture, they process them identically. Also, theoretically, all IRs can be converted to the one, which will make them identical.

This finding corresponds with the previous sections – the source language does not affect the compiler construction much. All compilers have the same

structure and translate the code in the same way. They require different implementations of the parser, to process the different rules, but all of them have the same stages and do the same steps.

Conclusion

In this thesis, we have designed a TinyGo language – a simplified version of the GoLang. Go as any language has its important design principles. TinyGo mostly preserves them, being the true sub-set language of the original one and at the same time omitting other things, being a not overwhelmed language, that is a good fit for studying the compiler creation.

Also, we have overviewed the most popular design solutions during the compiler creation and designed a compiler for TinyGo. The compiler is composed only of two main parts (front end and back end). The modularity and simplicity made it a great tool for an education purpose.

Comparison with the other compilers has shown, that source language does not significantly affect the modular compilers. With the modular design, the only part that depends on the input language is the front end. After that, it lowers the program into an IR, where the features of the original language are not presented and divided into a set of simpler instructions with the same behavior. Although different compilers have different IR, all of such representations have the same nature to be a middle language in the compiler and do not depend on the input. All other parts, such as the middle and back end, depend only on IR, meaning they are not affected by the source language.

Future Work

The creation of the compiler is a substantial problem. Despite the fact, that the TinyGo compiler is done, there are plenty of different places available for improvements and updates in the future.

The first place for updates is the front end. TinyGo does not support a lot of different things for an original language. To make a language support new features, an extension of the front-end is required, as the module, processes the language and converts it for the rest of the compiler. The possible extensions of it are with interfaces, generic programming, etc.

The middle end currently does not exist at all. Plenty of optimizations have been described and that exists. They are all can be applied. And thanks to modularity, the addition of them would not cause any significant code changes. The list of possible optimizations has been described in the second chapter 2.2.

The last place for potential improvements is the back end. We have described different algorithms during the code generation. TinyGo utilizes the

not optimal ones and it can be changed with the better ones. Another approach to the back-end improvements is expansion. Now, it supports only the Tiny x86 architecture. In the future, it can be converted into a list of different machine codes, making the compiler more cross-platform.

Bibliography

[1] Strejc, I. *Tiny x86 - Architecture Simulator for Educational Purposes*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2021, available at `https://dspace.cvut.cz/handle/10467/94644`.

[2] Filip, G. *Tiny86 Debugger*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2023, available at `https://dspace.cvut.cz/handle/10467/108916`.

[3] Schneck, P. B. A survey of compiler optimization techniques. In *Proceedings of the annual conference on - ACM'73*, New York, New York, USA: ACM Press, 1973.

[4] Aho, A. V.; Lam, M. S.; et al. *Compilers*. Upper Saddle River, NJ: Pearson, second edition, Aug. 2006.

[5] Liu, Y. A.; Stoller, S. D. From recursion to iteration. In *Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, New York, NY, USA: ACM, Nov. 1999.

[6] Potkonjak, M.; Srivastava, M.; et al. Multiple constant multiplications: efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 15, no. 2, 1996: pp. 151–165, doi:10.1109/43.486662.

[7] Blindell, G. H. *Instruction selection*. Cham, Switzerland: Springer International Publishing, first edition, June 2016.

[8] Muth, R. Register liveness analysis of executable code. *Manuscript, Dept. of Computer Science, The University of Arizona, Dec*, 1998.

[9] Poletto, M.; Sarkar, V. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, volume 21, no. 5, sep 1999: p. 895–913, ISSN 0164-0925, doi:10.1145/330249.330250. Available from: `https://doi.org/10.1145/330249.330250`

[10] Chaitin, G. J.; Auslander, M. A.; et al. Register allocation via coloring. *Computer Languages*, volume 6, no. 1, 1981: pp. 47–57, ISSN 0096-0551, doi:https://doi.org/10.1016/0096-0551(81)90048-5. Available from: https://www.sciencedirect.com/science/article/pii/0096055181900485

[11] Briggs, P. *Register allocation via graph coloring*. Rice University, 1992.

[12] Zakowski, Y.; Beck, C.; et al. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, volume 5, no. ICFP, Aug. 2021: pp. 1–30.

[13] Carroll, S.; Ko, W.; et al. Optimizing compiler design for modularity and extensibility. In *Languages and Compilers for Parallel Computing: 14th International Workshop, LCPC 2001, Cumberland Falls, KY, USA, August 1–3, 2001 Revised Papers 14*, Springer, 2003, pp. 1–17.

[14] History of the GCC compiler. https://gcc.gnu.org/wiki/History, accessed: 2024-04-01.

[15] The Go Programming Language Specification. https://go.dev/ref/spec, accessed: 2023-10-15.

[16] Prokopič, M. *Modular Compiler for the TinyC Language*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2023, available at https://dspace.cvut.cz/handle/10467/108868.

[17] Slávik, M. *TinyC Optimizing Compiler*. Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2023, available at https://dspace.cvut.cz/handle/10467/108848.

Acronyms

AST Abstract Syntax Tree.

BIE-PJP Parsers and Compilers.

CLI Command Line Input.

CTU Czech Technical University in Prague.

DAG Directed acyclic graph.

IR Intermediate Represintation.

ISA Instruction Set Architecture.

MM Main Memmory.

NIE-GEN Compiler Constrcution.

OOP Object-Oriented Programming.

SSA Single Static Assignment.

VM Virtual Machine.

Appendix A

TinyGo Grammar

A.1 Notation

```
[] - option (0 or 1 time)
{} - repetition (0 to n times)
() - grouping
|  - alternation
identifier with capital letters -- non terminal symbol
"..." - terminal symbols inside quotation marks
```

A.2 Program

```
SOURCEFILE := PACKAGE ";" { (DECLARATION | FUNCTION) ";" }
PACKAGE := "package" IDENTIFIER
```

Listing A.1: Grammar of the program

A.3 Declarations

```
DECLARATION := CONSTDECL | TYPEDECL | VARDECL

CONSTDECL := "const" (CONSTPRIME | "("  {CONSTPRIME ";"} ")" )
CONSTPRIME := IDENTIFIER_LIST [TYPE] "=" EXPR_LIST

TYPEDECL := "type" (TYPEPRIME | "(" {TYPEPRIME ";"} ")")
TYPEPRIME := IDENTIFIER ["="] TypeOfAssign

VARDECL := "var" (VARPRIME | "(" {VARPRIME ";"} ")")
VARPRIME := IDENTIFIER_LIST ( (TYPE [ "=" EXPR_LIST]) | "="
    EXPR_LIST )

SHORT_DECL := IDENTIFIER_LIST ":=" EXPR_LIST
```

Listing A.2: Grammar of declarations

A.4 Functions and methods

```
FUNCTION := "func" [METHOD_SIGNATURE] IDENTIFIER SIGNATURE BLOCK

METHOD_SIGNATURE := "(" IDENTIFIER TYPE ")"
SIGNATURE := PARAMETERS [RESULT]
PARAMETERS := "(" [ PARAMETER_LIST ] ")"
PARAMETER_LIST := PARAMDECL {"," PARAMDECL}
PARAMDECL := IDENTIFIER_LIST TYPE
RESULT := PARAMETERS | TYPE
```

Listing A.3: Grammar of the function (method) declaration

A.5 Types

```
TYPE := TYPE_NAME | TYPE_LIT | "(" TYPE ")"
TYPE_NAME := IDENTIFIER
TYPE_LIT := BASE_TYPES | STRUCT_TYPE | POINTER_TYPE
BASE_TYPES := "int8" | "int32" | "int" | "int64" | "float"

STRUCT_TYPE := "struct" "{" {FIELD_OF_STRUCT ";"} "}"
FIELD_OF_STRUCT := IDENTIFIER_LIST TYPE

POINTER_TYPE := "*" TYPE
```

Listing A.4: Grammar of types

A.6 Numbers and Identifier

```
IDENTIFIER := LETTER {LETTER | DIGIT}

NUM := INT_NUM | DEC_FLOAT_NUM

INT_NUM := DEC_NUM | HEX_NUM | OCT_NUM | BIN_NUM
DEC_NUM := "0" | ("1" .. "9") [ ["_"] DEC_DIGS ]
HEX_NUM := "0" ("x" | "X") ["_"] HEX_DIGS
OCT_NUM := "0" ("o" | "O") ["_"] OCT_DIGS
BIN_NUM := "0" ("b" | "B") ["_"] BIN_DIGS

DEC_DIGS := ("0" .. "9") {["_"] ("0" .. "9")}
HEX_DIGS := ("0" .. "f") {["_"] ("0" .. "f")}
OCT_DIGS := ("0" .. "7") {["_"] ("0" .. "7")}
BIN_DIGS := ("0" | "1") {["_"] ("0" | "1")}

DEC_FLOAT_NUM := DEC_DIGS "." [DEC_DIGS]
```

Listing A.5: Grammar of numbers and identifier

A.7 Numbers and Identifier

```
IDENTIFIER := LETTER {LETTER | DIGIT}

NUM := INT_NUM | DEC_FLOAT_NUM

INT_NUM := DEC_NUM | HEX_NUM | OCT_NUM | BIN_NUM
DEC_NUM := "0" | ("1" .. "9") [ ["_"] DEC_DIGS ]
HEX_NUM := "0" ("x" | "X") ["_"] HEX_DIGS
OCT_NUM := "0" ("o" | "O") ["_"] OCT_DIGS
```

```
9  BIN_NUM := "0" ("b" | "B") ["_"] BIN_DIGS
10
11 DEC_DIGS := ("0" .. "9") {["_"] ("0" .. "9")}
12 HEX_DIGS := ("0" .. "f") {["_"] ("0" .. "f")}
13 OCT_DIGS := ("0" .. "7") {["_"] ("0" .. "7")}
14 BIN_DIGS := ("0" | "1") {["_"] ("0" | "1")}
15
16 DEC_FLOAT_NUM := DEC_DIGS "." [DEC_DIGS]
```

Listing A.6: Grammar of numbers and identifier

A.8 Statements

```
1  STATEMENT := DECLARATION | IF_STMT | SWITCH_STMT | FOR_STMT | "
       break" | "contunie" | RETURN_STMT | SIMPLE_STMT
2  SIMPLE_STMT := SHORT_DECL | EMPTY | ASSIGMENT | EXPR
3
4  IF_STMT := "if" EXPR_STMT BLOCK ["else" (BLOCK | IF_STMT)]
5
6  SWITCH_STMT := "switch" [ EXPR ] "{" { CASE_CLAUSE } "}"
7  CASE_CLAUSE := SWITCH_CASE ":" STATEMENT_LIST
8  SWITCH_CASE := "case" EXPR_STMT | "default"
9
10 FOR_STMT := "for" FOR_CLAUSE BLOCK
11 FOR_CLAUSE := [SIMPLE_STMT] ";" [EXPR_STMT] ";" [SIMPLE_STMT]
12
13 RETURN_STMT := "return" EXPR_STMT
14
15 ASSIGMENT := IDENTIFIER_LIST OP EXPR_LIST
16 OP := ['+' | '-' | '*' | '/' | '%'] =
```

Listing A.7: Grammar of statements

A.9 Expression

```
1  EPSILON :=
2
3  EXPR_STMT := E11
4
5  E11 := E10 E11_PRIME
6  E11_PRIME := "||" E10 E11_PRIME | ϵ
7
8  E10 := E9 E10_PRIME
9  E10_PRIME := "&&" E9 E10_PRIME | ϵ
10
11 E9 := E8 E9_PRIME
12 E9_PRIME := "|" E8 E9_PRIME | ϵ
13
14 E8 := E7 E8_PRIME
15 E8_PRIME := "&" E7 E8_PRIME | ϵ
16
17 E7 := E6 E7_PRIME
18 E7_PRIME := "==" E6 E7_PRIME | "!=" E6 E7_PRIME | ϵ
19
20 E6 := E5 E6_PRIME
21 E6_PRIME := "<" E5 E6_PRIME | ">" E5 E6_PRIME | "=>" E5 E6_PRIME |
       "<=" E5 E6_PRIME | ϵ
22
```

```
23 E5 := E4 E5_PRIME
24 E5_PRIME := "+" E4 E5_PRIME | "-" E4 E5_PRIME | ϵ
25
26 E4 := E2 E4_PRIME
27 E4_PRIME := "*" E2 E4_PRIME | "/" E2 E4_PRIME | "%" E2 E4_PRIME | ϵ
28
29 E2 := "+" E2 | "-" E2 | "++" E2 | "--" E2 | "!" E2 | E1
30
31 E1 := E0 E1_PRIME
32 E1_PRIME := "++" E1_PRIME | "--" E1_PRIME | "(" EXPR_LIST ")"
       E1_PRIME | "." E0 E1_PRIME | ϵ
33
34 E0 := "(" E11 ")" | NUM |  IDENTIFIER
```

Listing A.8: Grammar of expressions

A.10 List

```
1 IDENTIFIER_LIST := IDENTIFIER {"," IDENTIFIER}
2 EXPR_LIST := EXPR_STMT {"," EXPR_STMT}
3 STATEMENT_LIST := {STATEMENT ";"}
4
5 BLOCK:= "{" STATEMENT_LIST "}"
```

Listing A.9: Grammar of lists

Appendix B

IR Grammar

This chapter describes the grammar of the Intermediate Language used in a TinyGo language.

B.1 Variables and Types

```
1 VARIABLE := '%' ('a' - 'z' | 'A' - 'Z' | 0 - 9) {'a' - 'z' | 'A' -
      'Z' | 0 - 9}
2
3 TYPE := ('int' | 'float' | 'ptr' | ('array[' INT ']' TYPE) | ('{' {
      TYPE ','} '}'))
```

Listing B.1: Variables and types grammar

B.2 Constants

```
1 INT := 1 - 9 {0 - 9}
2 FLOAT := 1 - 9 {0 - 9} '.' {0 - 9}
3 PTR := INT | 'nullptr'
4 STRUCT := '{' {(INT | FLOAT | PTR | VARIABLE) ','} '}'
5
6 CONST := INT | FLOAT | PTR | STRCUT
7
8 CONSTINST = VARIABLE '= create' TYPE CONST
```

Listing B.2: Grammar of constanst in IR

B.3 Functions

```
1
2 FUNCTION := 'function' VARIABLE 'arguments:' {TYPE VARIABLE ','} '
      return:' TYPE '{' FUNCTIONBLOCK
3
4 FUNCTIONBLOCK := {INST}
5 INST := CALLINST | ALLOCAINST | STOREINST | LOADINST | MEMCPYINST |
       MEMACCINST | LABELINST | CONDJUMPINST | JUMPINST | ARITHOPINST
       | SCANINST | PRINTINST | CASTINST | RETINST
6
```

```
7 CALLINST := VARIABLE '= call:' VARIABLE 'arguments:' {VARIABLE ','}
```

Listing B.3: Grammar of the functions

B.4 Memory Access and Allocation Instructions

```
1
2 ALLOCAINST := VARIABLE '= alloca' TYPE [CONST]
3 STOREINST := 'store what :' VARIABLE ' where:' VARIABLE
4 LOADINST := VARIABLE '= load from:' VARIABLE
5
6 MEMCPYINST := 'copy content from:' VARIABLE 'to:' VARIABLE 'size:'
    INT
7 MEMACCINST := VARIABLE '= get member from:' VARIABLE 'which:' INT
```

Listing B.4: Memory instructions grammar

B.5 Control Flow Instructions

```
1 LABELINST := VARIABLE
2 CONDJUMPINST := 'cond jump condition:' VARIABLE 'if true:' VARIABLE
     'if false:' VARIABLE
3 JUMPINST := 'jump' VARIABLE
```

Listing B.5: Grammar for control flow

B.6 Other Instructions

```
1
2 AIRTHOP := 'add' | 'sub' | 'mul' | 'div' | 'mod' | 'fadd' | 'fsub'
    | 'fmul' | 'fdiv' | 'EQ' | 'NQ' | 'LT' | 'LE' | 'GT' | 'GE'
3
4 ARITHOPINST := VARIABLE '= binop' AIRTHOP 'first:' VARIABLE 'second
    :' VARIABLE
5
6 SCANINST := VARIABLE '= scan'
7
8 PRINTINST := 'print:' VARIABLE
9
10 CASTINST := VARIABLE '= cast' VARIABLE 'to:' TYPE
11
12 RETINST := 'ret' [VARIABLE]
```

Listing B.6: Grammar of the other instructions

Appendix C

Contents of attachments

```
readme.txt ........................ the file with the content description
src....................................the directory of source codes
  CompilerForTinyGo ................. the source code of the compiler
  thesis...............the directory of LaTeX source codes of the thesis
text .......................................the thesis text directory
  thesis.pdf...........................the thesis text in PDF format
```