University of Southern Denmark

Department of Mathematics and Computer Science

Bachelor Thesis

The Giga Chad Compiler

The only GCC you will ever need

1st of June, 2024

Members

Jonas Bork Jørgensen	jonjo21
Malthe Hedelund Petersen	malpe21
Mikkel Nielsen	mikkn21
Sofus Hesseldahl Laubel	slaub21

Supervisor

Jakob Lykke Andersen, Associate Professor, Ph.D.

Abstract

To gain hands-on experience and insight into the intricacies of the compilation process, we have developed the Giga Chad Compiler (GCC) which compiles from the Giga Chad Language (GCL) down to x86-64 Linux assembly code in AT&T syntax. GCL is Java-like in syntax, and contains a unique representation for arrays that closely resembles that of coordinates in a typical coordinate system. GCC combines lexical analysis and syntax analysis into one parsing phase, using Parsing Expression Grammars and Boost.Spirit X3 to produce an Abstract Syntax Tree (AST). GCC's semantic analysis consists of the symbol collection and type checking phase. During symbol collection GCC finds all declared variables, functions, and classes and gathers them into scopes, wherein GCC maps them to corresponding symbols, after which it ensures proper usage based on scoping and usage of variables. The type checking phase makes use of a producer-consumer scheme, where vertices of the AST will produce information that is consumed by other vertices of the AST, together with a stack to ensure proper typing. The code generation phase of GCC uses the information from the previous phases to produce an intermediate representation that closely resembles the assembly code it compiles down to. The intermediate representation utilizes activation records and static linking to support deeply nested code structures. GCC uses liveness analysis together with peephole optimization to eliminate redundant instructions from its intermediate representation. GCC additionally uses the information from its liveness analysis, to carry out smart register allocation, using a colouring by simplification algorithm, to ensure an efficient register allocation. The final phase of GCC, called the emit phase, turns the optimised intermediate representation into actual assembly code.

Resume

For at få praktisk erfaring og indsigt i oversættelsesprocessens detaljer har vi udviklet Giga Chad Compiler (GCC), som oversætter fra Giga Chad Language (GCL) til x86-64 Linux assembly-kode i AT&T-syntaks. GCL har en Java-lignende syntaks og indeholder en unik repræsentation af arrays, der minder meget om koordinater i et typisk koordinatsystem. GCC kombinerer leksikal analyse og syntaksanalyse i én parsningsfase ved hjælp af Parsing Expression Grammars og Boost.Spirit X3 til at producere et Abstract Syntax Tree (AST). GCC's semantiske analyse består af symbol indsamlings- og type kontrollerings faserne. Under symbolindsamlingen finder GCC alle deklarerede variabler, funktioner og klasser og samler dem i scopes, efterfølgende afbilder GCC dem til tilsvarende symboler, hvorefter den sikrer korrekt brug baseret på scoping og brug af variabler. Typekontrolleringsfasen gør brug af et producer-consumer-skema, hvor knudepunkter i AST'et producerer information, der forbruges af andre knudepunkter i AST'et, sammen med en stak for at sikre korrekt typning. Kodegenereringsfasen i GCC bruger informationen fra de tidligere faser til at producere en mellemrepræsentation, der minder meget om den assembly-kode, den oversætter til. Mellemrepræsentationen udnytter aktiveringsoptegnelser og statisk sammenkædning til at understøtte dybt indlejrede kodestrukturer. GCC bruger livlighedsanalyse sammen med kighul-optimering til at eliminere overflødige instruktioner fra sin mellemrepræsentation. GCC bruger desuden informationen fra sin livlighedsanalyse til at udføre smart registerallokering ved hjælp af en farvning-ved-simplificering-algoritme for at sikre en effektiv registerallokering. Den sidste fase af GCC, kaldet emitteringsfasen, omdanner den optimerede mellemrepræsentation til faktisk assembly-kode.

Contents

1 Introduction

A compiler for programming languages is software that converts code written in one programming language, called the source language, to code in another, typically a lower-level language than the source, called the target language. The process of converting the code is called *compilation*. This can create a chain of compilers to achieve the lowest-level language that can be executed on a computer. For example, you might create a programming language, X, that compiles to Y that is then compiled to Z and so on. It is then possible to later create a new language that compiles to X to add to the chain of compilers. Each language in the chain can provide additional abstractions on top of the next language, such that abstractions can build on abstractions. These abstractions allow the source language to be easier and faster to write in than the target language. Additional benefits can be provided by the abstractions such as enhanced portability, for example, the compilation of C to assembly where assembly is dependent on the computer's CPU architecture but C code is architecture-independent and can then be compiled to assembly for multiple different architectures.

To guarantee the correctness of the compiled code, the abstractions of the source language need to be converted to the target language in some general way which often leads to unoptimised and slow code, compared to writing the code manually in the target language. Thus, it is very relevant for the compiler to also perform optimisations on the compiled code to further incentivise people to use the source language. If it is too slow, it would not be desirable to use.

Compilers play a key role in any software written in compiled programming languages, and thus, compilers are tremendously important in the modern world where almost everything is driven by software in some way. They do, however, also pose unique challenges when creating compilers as you need to both guarantee the correctness of the compiled code and have the compiled code run fast, while also keeping the running time of the compiler at a manageable level.

In this bachelor thesis, we will define a programming language called the Giga Chad Language (GCL) and create a compiler called the Giga Chad Compiler (GCC) that compiles GCL into assembly. Neither the language nor the compiler will be revolutionary nor significantly different from existing programming languages and compilers. In fact, the exact opposite will be the case, where the language will be very similar to existing languages and the compiler will follow a standardised approach. The point of this thesis is to provide the reader and ourselves insight into how compilers work in general by going through the design and implementation of one, and to a lesser degree, how to design a programming language. This insight can be applied when writing code in other popular and widely used programming languages, since the compilers for these perform similar work, which allows us to write more readable and fast code.

The standardised approach that our compiler will follow is the one described in [1]. The input to the compiler will be a file containing GCL code, after which the compiler will output corresponding assembly code by going through a list of predefined phases that each have a significant responsibility in the compilation.

2 The Giga Chad Language

The Giga Chad Language (GCL) is a statically typed programming language designed to provide a clear and unambiguous programming experience. Drawing syntactic influence from Java, GCL aims to reduce potential logical errors and improve code readability through explicit and intuitive syntax choices.

Key design principles of GCL include:

- Static typing: All variables must have a declared type, allowing the compiler to catch type-related errors early in the development process.
- Classes: GCL supports classes, enabling programmers to organize code into reusable and modular components.
- Explicit syntax: GCL's syntax is designed to be explicit and clear, reducing ambiguity and potential sources of bugs. For example, there are no implicit type conversions.
- Familiarity: By borrowing syntax and concepts from widely used languages like Java, GCL aims to be easily learnable for programmers already familiar with programming.
- Design pattern support: GCL's features are designed to facilitate the implementation of common design patterns, promoting code reuse and maintainability.

The subsequent sections will delve into the specifics of GCL's grammar, semantics, and unique language features. Through these design choices, GCL strives to provide a programming language that is both powerful and approachable, encouraging clear, robust, and maintainable code.

2.1 Grammar

The language is formally written as a Parsing Expression Grammar (PEG) on Figure 1. PEG is similar to context-free grammars (CFGs), but it has a different interpretation: the choice operator, denoted by "/". In section 5.1, "Parsing Expression Grammars", a deeper explanation of PEG will be given.

In GCL, a program consists of several declarations (decls) as seen in Figure 1. These declarations can then be expanded into usual programming structures like classes, functions and variable declarations. GCL demands that a program must consist of exactly one "main" function, so the program knows where to start. The main function consists of the code to be executed. This design choice exposes the execution flow of the program to the user, to minimize ambiguity around what the program is doing, thereby mitigating logic errors and making the language more readable. Only class, function, and variable declarations are allowed outside the "main" function, promoting good object-oriented practices such as encapsulation.

As seen in the grammar, GCL does not have structs, only classes. These classes only contain variables. That is, structs and classes are not differentiated, instead only classes are available. These classes are instantiated with a "`new`" keyword, like in Java.

Unlike many other languages, GCL does not use `&&` and `||` for the logical "and" and "or" in conditional statements, but rather use the singular `&` and `|` instead as seen in the grammar. This is because it is shorter, making it easier to read and faster to type. In other languages, `&` and `|` are used for bitwise operations, which is why these could not be used for the logical "and" and "or". These bitwise operations, however, have limited use cases in a high-level language like GCL and thus do not need to reserve short symbols.

Compared to other object-oriented programming languages, GCL has an unconventional syntax for arrays, that aims to be more akin to coordinates in mathematics such that you can index into an array, `arr`, with the variables `x` and `y` like `arr[x,y]`. Most other object-oriented languages would have you write `arr[x][y]` instead. For consistency, the syntax for the array type and array initialisation also has to match the array indexing so for defining a variable, `arr`, of type `int` array with 3 dimensions, you would write `int[3] arr`, and to simultaneously assign to it, a new array with 3 dimensions, you would write `int[3] arr = new int[10, 42, 21]`, where 10, 42 and 21 are the sizes of the 1st, 2nd and 3rd dimensions, respectively. This syntax is more concise as it avoids redundant square brackets for multidimensional arrays compared to other languages. The grammar enforces that the types of arrays is a "primitive type" i.e., we do not actually have "arrays of arrays" as other programming languages, instead do the array type expect a literal integer which indicates the size of the dimension of the array.

In GCL, parentheses are optional for `while`, `if` and `return` statements. This choice makes it easier to write code in GCL, as it allows for more flexibility and thereby also improves readability for the individual user.

Further, in GCL, null is implemented as "`beta`" since a variable is a beta if it has no value; an alpha variable has a value. This allows the user to clearly distinguish between if a value holds a meaningful value or not.

GCL allows a statement to become an expression followed by a semicolon. At first glance, this might seem unnecessary since an expression like `2+2` will never perform real work, so the code line `2+2;` will not do anything, and will thus be *dead code*. An expression can, however, also become a function call which can have side effects and thus `f();` might perform necessary work. This could have been solved by simply making a statement able to become a function call instead of an expression. This would not be a viable solution in the long run, however, since we would like to support operator overloading in the future with GCL. Operator overloading means that the user can define custom handling of built-in operators like `+` and `-` such that an expression like `a+b` can execute arbitrary code, meaning it is equivalent to a function call and thus can have side effects. Therefore, it was decided to allow a statement to become an expression. A solution to the dead code this might produce, like `2+2;`, is explored in Section 11, Peephole Optimization.

⟨*prog*⟩ = ⟨*decl*⟩*

⟨*decl*⟩ = ⟨*var_decl_assign*⟩ /
⟨*var_decl_statement*⟩ /
⟨*func_decl*⟩ / ⟨*class_decl*⟩

⟨*class_decl*⟩ = '`class`' ⟨*id*⟩ '`{`'
⟨*var_decl_statement*⟩* '`}`'

⟨*func_decl*⟩ = ⟨*type*⟩ ⟨*id*⟩ '`(`'
⟨*parameter_list*⟩ '`)`' ⟨*block*⟩

⟨*var_decl_assign*⟩ = ⟨*type*⟩ ⟨*id*⟩
'`=`' ⟨*expression*⟩ '`;`'

⟨*var_decl_statement*⟩ = ⟨*type*⟩
⟨*id*⟩ '`;`'

⟨*id*⟩ = [a-zA-Z_][a-zA-Z_0-9]*

⟨*id_access*⟩ = (⟨*id*⟩ '.')* ⟨*id*⟩

⟨*type*⟩ = ⟨*array_type*⟩ /
⟨*primitive_type*⟩ /
⟨*class_type*⟩

⟨*class_type*⟩ = ⟨*id*⟩

⟨*primitive_type*⟩ = '`int`' / '`bool`'

⟨*array_type*⟩ = ⟨*primitive_type*⟩
'`[`' ⟨*integer*⟩ '`]`'

⟨*parameter_list*⟩ = ((⟨*parameter*⟩
'`,`')* ⟨*parameter*⟩)?

⟨*parameter*⟩ = ⟨*type*⟩ ⟨*id*⟩

⟨*block*⟩ = '`{`' ⟨*block_line*⟩* '`}`'

⟨*block_line*⟩ = ⟨*statement*⟩ /
⟨*decl*⟩

⟨*statement*⟩ = ⟨*if_statement*⟩
/ ⟨*while_statement*⟩
/ ⟨*print_statement*⟩
/ ⟨*return_statement*⟩
/ ⟨*break_statement*⟩
/ ⟨*continue_statement*⟩
/ ⟨*var_assign*⟩
/ ⟨*expression*⟩ '`;`'

⟨*var_assign*⟩ = (⟨*id_access*⟩ /
⟨*array_index*⟩) '`=`'
⟨*expression*⟩ '`;`'

⟨*if_statement*⟩ = '`if`' ⟨*expression*⟩
⟨*block*⟩ ('`else`' '`if`'
⟨*expression*⟩ ⟨*block*⟩)* ('`else`'
⟨*block*⟩)?

⟨*while_statement*⟩ = '`while`'
⟨*expression*⟩ ⟨*block*⟩

⟨*break_statement*⟩ = '`break`' '`;`'

⟨*continue_statement*⟩ =
'`continue`' '`;`'

⟨*return_statement*⟩ = '`return`'
⟨*expression*⟩ '`;`'

⟨*print_statement*⟩ = '`print`' '`(`'
⟨*expression*⟩ '`)`' '`;`'

⟨*expression*⟩ = ⟨*binop_expression*⟩
/'`(`' ⟨*expression*⟩ '`)`'
/ ⟨*beta*⟩
/ ⟨*array_expression*⟩
/ ⟨*object_instantiation*⟩
/ ⟨*function_call*⟩
/ ⟨*array_index*⟩
/ ⟨*id_access*⟩ / ⟨*integer*⟩ /
⟨*boolean*⟩

Figure 1: The first part of the grammar for the Giga Chad Language (GCL). The grammar is continued on Figure 2.

⟨*beta*⟩ = 'beta'

⟨*integer*⟩ = [0-9]+

⟨*boolean*⟩ = '`true`' / '`false`'

⟨*function_call*⟩ = ⟨*id*⟩ '`(`'
    ⟨*argument_list*⟩ '`)`'

⟨*argument_list*⟩ =
    ⟨*expression_list*⟩?

⟨*expression_list*⟩ = (⟨*expression*⟩
    '`,`')* ⟨*expression*⟩

⟨*binop_expression*⟩ = ⟨*expression*⟩
    ('`+`' / '`-`' / '`*`' / '`/`' / '`%`' / '`==`'
    / '`!=`' / '`<`' / '`>`' / '`<=`' / '`>=`' /
    '`&`' / '`|`')
    ⟨*expression*⟩

⟨*array_expression*⟩ = '`new`'
    ⟨*primitive_type*⟩ '`[`'
    ⟨*expression_list*⟩ '`]`'

⟨*array_index*⟩ = ⟨*id_access*⟩ '`[`'
    ⟨*expression_list*⟩ '`]`'

⟨*object_instantiation*⟩ = '`new`' ⟨*id*⟩
    '`()`'

Figure 2: Grammar of the Giga Chad Language (GCL) (Part 2).

2.2 Semantics

Function-, variable- and class declarations all have an ID, as seen on Figure 1, i.e. their name. A significant detail about this ID is that it cannot be equal to any reserved keywords, such as `int`, `bool`, `if`, etc, because it could create problems for the parser and the readability of the code. For example, if you had a class named `int`, and you wrote `int x`, it would be unclear whether `int` referred to the class you created or the primitive type.

GCL does not enforce a strict top-down order on the user when it comes to class and function declarations. Instead, the user can define classes or functions at any point in the program, and then instantiate or call them, respectively, at any point in the program. This is not the case for variables though, as variables have to be declared earlier in the program before they are used. The reasoning behind this is that classes and functions are often used to abstract away parts of the code, and if they are forced to all be defined at the top of the file, then this abstraction is lost, as the user would have to look through all their functions and classes to get to the code that uses those functions and classes.

Another motivation behind allowing classes and functions to be defined in any order is that it avoids the issues of circular usage, i.e., function A wishes to call function B and vice versa. If functions and classes had to be declared in order, this functionality either simply would not be an option, or there would need to be some form of forward declaration system in place, that allows the user to declare a function or class ahead of time so that the compiler is aware that it should expect the forward declared function or class later on. However, such a system is quite cumbersome for the user to use, and we wished to avoid imposing such a burdensome system onto our user.

A significant detail about variables in GCL, is that they all have a default value. That is, simply by declaring a variable, it will already have been assigned a value. This is done to avoid the undefined behaviour of what should happen if

```
while (true) {
    int f() {
        break;
        return 0;
    }
    while (true) {
        f();
        i = i + 1;
    }
}
```

Figure 3: A while-loop with a function inside that has a break statement. The loop also has a nested while-loop that calls the function. This poses the question of what this code would do, since it is not immediately apparent which loop the function would break out of when it is called.

a user tries to use an uninitialized variable, as this now cannot happen because a variable will always have a value. This is discussed in detail in section 8.1.1.

Guards in `if` and `while` statements have to evaluate to a boolean value, to be a correct guard.

Inside while-loops, break and continue can be used to alter the flow of the loop by skipping the current iteration and continuing with the next iteration, respectively. GCL supports nested functions, so consider the example in Figure 3. It is not clear to a reader of the code which while-loop it would break out of, since it could either be decided at the time of the function declaration or at the time of the function call. Thus, break- and continue statements are not allowed to be inside a function even when the function is inside a while statement.

3 The Giga Chad Compiler Overview

The Giga Chad Compiler (GCC) will compile source code written in the Giga Chad language and output corresponding x86-64 Linux assembly code. The process that generates this code will follow the phases discussed in "Modern Compiler Implementation in C" [1] with some modifications to better fit the Giga Chad language and C++. An overview of the phases of GCC is shown in Figure 4, along with their responsibilities and how they are integrated.

The first phase of GCC is the parsing phase, which generates an abstract syntax tree (AST) from the code written in GCL. Traditionally, this involves separate parsing and lexing phases, but in GCC, these are combined into a single phase using Boost.Spirit X3.

The "Semantic Analysis" phase uses the AST to perform symbol collection and type checking. Symbol collection gathers and maps all variables, functions, and class declarations to unique symbols, ensuring proper scoping and identifier

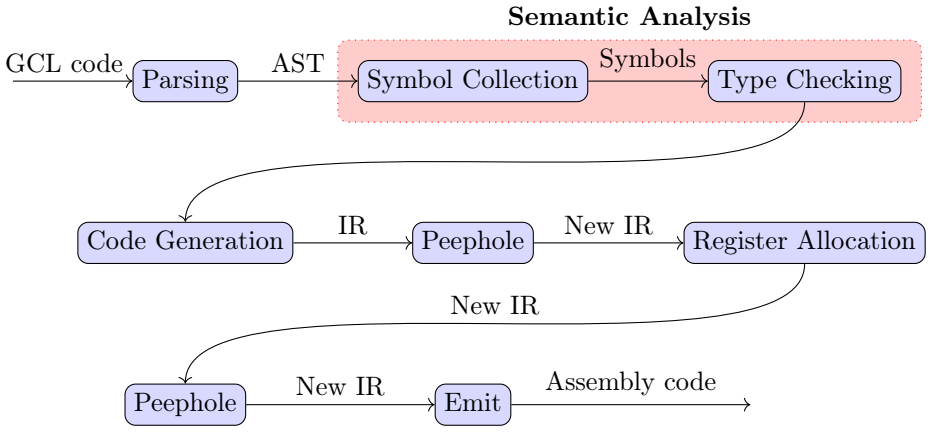


Figure 4: Overview of the Giga Chad compiler. The input is some GCL code and after going through all the phases, it outputs assembly code. The parsing phase and the lexing phase are both encapsulated in the "Parsing" block in the diagram. Type checking, unlike the other phases, does not output anything because it only ensures that the code is valid, and thus does not generate any information that the other phases need, apart from allowing the succeeding phases to assume that the types in the code are correct. The peephole optimisations performed by the compiler are referred to as "peephole" in the diagram.

resolution. It sends these symbols, along with their types and scope information, to the type checking phase for validation. Both symbol collection and type checking involve multiple passes through the AST and various checks.

After type checking has validated the AST, the code generation phase translates the validated AST into intermediate code, which is passed to the peephole optimization phase. This phase performs liveness analysis and applies peephole optimizations to enhance the intermediate code.

The optimized intermediate code is then passed to the register allocation phase, which assigns values and variables to a limited number of CPU registers, also utilizing liveness analysis. This phase ensures efficient use of the CPU's registers, minimizing memory access and improving the overall performance of the generated code.

Afterwards, the peephole optimization is run a second time to further enhance the intermediate code. Finally, the emit phase maps the intermediate code to actual x86-64 assembly code.

In the subsequent sections, each of the phases illustrated in Figure 4 will be explored in greater detail, providing a comprehensive explanation of their design and implementation.

```
giga [options] <input-file>
```

Figure 5: The format of how you would call the GCC program, i.e., `giga`. The `[options]` part of the format refers to the optional flags on Table 1 that can be given to the program. If you use multiple flags, the short names for the flags can be combined into a single `-`, for example, if you want to use both `-p`, and `-i`, then you can write `-pi` instead. The `<input-file>` is a required argument that is the file that should be compiled.

4 Using the Compiler

When the code for GCC has been downloaded, `cmake` should then be run with the root directory of the GCC files as input, followed by running `make`. This automatically uses the correct `make` flags based on whether you are on Linux or Mac. Both `cmake` and `make` commands are recommended to be run from the user-created directory, called "build" to avoid mixing the build files with the source files. These commands result in, among other files, the GCC compiler as an executable program called "giga". The "giga" executable can then be called from the command line using the format given in Figure 5 which supports the flags given on Table 1. These flags were implemented using the Boost library "Program_options" [11].

5 Lexing and Parsing

In theoretical compiler design, as outlined in [1], lexical analysis (lexing) and parsing are the first two phases. This structure is commonly seen when building a compiler with context-free grammars (CFGs) using Yacc/Bison and Lex/Flex. However, since we developed our compiler using parsing expression grammars (PEGs) and Boost.Spirit X3, which employs recursive descent parsing with expression templates and operator overloading, we were able to combine these two phases into a single 'parsing' phase.

In this section, we will explore the capabilities provided by PEG and the Spirit X3 library. We will discuss what a recursive descent parser is and how templates and operator overloading are utilized in this context. Additionally, we will highlight some differences between Spirit X3 and the more traditional approach of using Yacc/Bison.

5.1 Parsing Expression Grammars

Tools like Yacc/Bison use Chomsky's generative system of grammars, i.e., context-free grammars (CFGs) and regular expressions (REs) to express the syntax of programming languages. But these were originally designed as a formal tool for modelling and analysing natural (human) languages. The ability of a CFG to express ambiguous syntax is an important and powerful tool for natural

Short name	Long name	Description
`-h`	`--help`	Display a help message describing the compiler flags.
`-P`	`--parse-only`	Stop after parsing.
`-S`	`--symbol-collection-only`	Stop after symbol collection.
`-T`	`--type-check-only`	Stop after type checking.
`-C`	`--code-generation-only`	Stop after code generation.
`-R`	`--register-allocation-only`	Stop after register allocation.
`-p`	`--print-ast`	Prints the AST generated by the parsing.
`-i`	`--print-input`	Print the input before parsing.
`-c`	`--print-code-generation`	Print the Intermediate code after the code generation phase.
`-r`	`--print-register-allocation`	Print the Intermediate code after the register allocation phase.
`-e`	`--peephole-only`	Stop after peephole optimization.
`-E`	`--disable-peephole`	Disable peephole optimization.
`-n`	`--naive-register-allocation`	Use the naive register allocation.

Table 1: The flags that are available in the GCC compiler and a description of what they do. All flags have short and long names, both of which are case-sensitive.

```
A → a b | a

A → a | a b
```

(a) EBNF rules

```
A = a b / a

A = a / a b
```

(b) PEG rules

Figure 6: Comparison of EBNF and PEG rules. Looking at Figure 6a the rules are equivalent in a CFG, but the PEG rules Figure 6b are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'. Note that in PEG notation the equals ('=') is used instead of the right arrow ('→') this is a convention to distinguish PEG rules from CFG rules, emphasizing that PEGs use a different passing strategy.

Operator	Type	Precedence	Description
' '	primary	5	Literal string
" "	primary	5	Literal string
[ ]	primary	5	Character class
.	primary	5	Any character
(e)	primary	5	Grouping
$e?$	unary suffix	4	Optional
$e*$	unary suffix	4	Zero-or-more
$e+$	unary suffix	4	One-or-more
$\&e$	unary prefix	3	And-predicate
$!e$	unary prefix	3	Not-predicate
$e_1\ e_2$	binary	2	Sequence
$e_1\ /\ e_2$	binary	1	Prioritized Choice

Table 2: Operators for Constructing Parsing Expressions

languages. Unfortunately, this power gets in the way when we use CFGs for machine-oriented languages that are intended to be precise and unambiguous [8, 15] . That is, using CFGs makes it unnecessarily difficult both to express and to parse machine-oriented languages. Parsing Expression Grammars (PEGS) provide an alternative, recognition-based formal foundation for describing programming languages syntax, which solves the ambiguity problem by not introducing ambiguity in the first place [4]. PEGs are stylistically similar to CFGs with RE-like features added, much like Extended Backus-Naur Form (EBNF) notation. A key difference is that in place of the choice operator '|', PEGs use a *prioritized* choice operator '/', which can be seen in Figure 1. This operator lists alternative patterns to be tested in *order*, unconditionally using the first successful match. The EBNF rules shown on Figure 6 are equivalent in a CFG because CFGs are inherently *nondeterministic*. This means that the parser can try both alternatives and will successfully parse the input if any of the alternatives match. But the PEG rules are different because PEGs are inherently *deterministic* and uses *ordered choices*. This means the parser will try the alternatives in the given order and choose the first one that matches, for the example shown in Figure 6b this will result in the second alternative never succeeding if the input string begins with 'a'. The operators for constructing PEGs are summarized in Table 2.

PEGs may be viewed as a formal description of a top-down parser [4]. Combining the intuitive, top-down parsing approach with a powerful and expressive grammar formalism. They extend beyond the capabilities of traditional LL(k) parsers and encompass the full range of deterministic LR(l) languages, providing a robust tool for parsing complex languages.

5.2 LL-Parser

Spirit X3 employs a object-oriented recursive descent parser [7, Section introduction]. This type of top-down parser is built from a set of mutually recursive procedures (or a non-recursive equivalent), where each procedure implements one of the non-terminals of the grammar [3]. The result is an LL (left-to-right, leftmost derivation) parser.

An LL parser is called LL(∞) if it uses a parsing strategy that allows for arbitrary lookahead to make parsing decisions. LL(∞) parsers are capable of recognizing all LL(k) grammars, where k can be any number, effectively providing infinite lookahead. This capability makes LL(∞) parsers functionally closer to PEG parsers. They can optimally parse an arbitrary LL(k) grammar by efficiently managing the amount of lookahead and lookahead comparisons [2].

The LL(∞), is a top-down parser, that starts at the root of the grammar and works its way down to the leaves while consuming the input from left to right. In theory, an LL(∞) parser can parse any context-free grammar, though in practice the computational resource limits this.

In Spirit X3 the infinite lookahead is, in reality, recursive descent with backtracking, which is a technique that determines which production to use by trying each production in turn [16]. This means the parser will try all possible parsing paths of all possible lengths to determine the correct one. Despite trying all possible lengths, the parser will continually try to match the first shortest match it finds. This approach ensures that the parser efficiently processes the input. LL parsers are generally considered to be efficient and fast if designed well, while also being able to handle a wide variety of grammar, though with some drawbacks that you have to think about when designing the grammar. It cannot naturally handle left-recursive rules due to its deterministic nature[4], i.e., it needs to know which grammar rule to apply. When an LL parser encounters a left-recursive rule, it tries to expand the non-terminal with itself indefinitely. The LL parser will always look for a match starting at the left of the rule, and if it encounters the same non-terminal again, it will try to resolve it, which leads to an infinite recursion. The parser gets stuck in a recursive attempt to match the rule without ever progressing.

e.g., the Figure 7a is a CFG with left recursion, which represents a series of 'a's in a CFG and shows how left recursion is permissible for CFGs. The Figure 7b is a PEG with left recursion, this rule is degenerate because it indicates that to recognize nonterminal A, a parser must first recognize nonterminal A... This restriction applies not only to direct left recursion as in this example, but also to indirect or mutual left recursion involving several nonterminals [4]. While this can seem like a big problem, it is generally not considered so, since there exist techniques [1, p. 52] for turning left recursion grammars into using right recursion instead.

```
A → a A | a

A → A a | a
```

(a) CFG rules

```
A = A a / a
```

(b) PEG rules

Figure 7: Simplified example of left-recursion in both a CFG and PEG. Shows how left recursion is permissible in CFGs, but as with top-down parsing in general, left recursion is unavailable in PEGs because it represents a degenerate loop

```
1 auto parser = x3::int_ >> ',' >> x3::int_;
```

Figure 8: Example of an Spirit X3 inline grammar in C++

5.3 Boost.Spirit X3

Spirit X3 is an embedded (or Internal) Domain-Specific Language (EDCL), these are typically implemented within a host language as a library and tend to be limited to the syntax of the host language [5]. Spirit X3 is an EDCL implemented as a library, meaning it is limited to the syntax of C++, its host language. It allows you to write grammars using a format similar to EBNF directly in C++. These inline grammar specifications can mix freely with other C++ code and, thanks to the generative power of C++ templates, are immediately executable[7]. Expression templates are a C++ template metaprogramming technique that Spirit X3 leverages. This technique builds structures representing a computation at compile time, where expressions are evaluated only as needed to produce efficient code for the entire computation [9]. When you write a grammar rule in Spirit X3, it is not immediately executed, but rather transformed into a series of template instantiations. These templates represent the parsing expressions and are built at compile time.

For example, Figure 8 defines a parser that matches an integer, a comma, and another integer. The '>>' operator is overloaded to create a composite parser expression using expression templates. The actual evaluation of the parser expressions happens at runtime, but the structure of the parser is determined at compile time. Expression templates allow the combination of different parsing rules into more complex grammar. Each component parser, e.g., '`x3::int_`', '`','`' is combined using template expressions to form a compiled parser. The use of template metaprogramming ensures that these parsers are highly optimized, leveraging the full power of C++'s compile-time capabilities. Since the target input grammars are written entirely in C++ we do not need any separate tools to compile, preprocess or integrate into the build process. Spirit X3 allows seamless integration of the parsing process with other C++ code. This often allows for simpler and more efficient code. The created parser is fully attributed, which

Syntax	Explanation
`x >> y`	Match `x` followed by `y`.
`x > y`	After matching `x`, expect `y`.
`*x`	Match `x` repeated zero or more times
`x | y`	Match `x`. If `x` does not match, try to match `y`.
`+x`	Match a series of one or more occurrences of `x`.
`-x`	Match `x` zero or one time.
`x & y`	Match `x` and `y`.
`x - y`	Match `x` but not `y`.
`x ^ y`	Match `x`, or `y`, or both, in any order
`x || y`	Match `x`, or `y`, or `x` followed by `y`.
`x [ function_expression ]`	Execute the function/functor returned by `function_expression`, if `x` matched.
`( x )`	Match `x` (can be used for priority grouping)
`x % y`	Match one or more occurrences of `x`, separated by occurrences of `y`.
`∼x`	Match anything but `x` (only with character classes such as `ch_p` or `alnum_p`)

Table 3: Modified PEG notation from Spirit X3

allows for easily building and handling hierarchical data structures in memory. These data structures resemble the structure of the input data and can directly be used to generate arbitrarily formatted output [7].

The Spirit X3 library uses a modified version of the original EBNF syntax to conform to C++ syntax, as it is an EDCL and therefore is limited by C++ syntax. Most notably is the shift operator "`>>`" as seen in Table 3. Since there are no juxtaposition operators in C++, it is simply not possible to write, e.g., "`a b`" to mean sequencing (`a` should follow `b`). Spirit X3 uses the shift operator for this purpose [7]. As most operators in PEG already exist in C++ Spirit X3 has modified the standard PEG notation from Table 2 to conform with C++ syntax, by postfixing the operators, and additionally adding some more operators for common tasks e.g., `%` for comma-separated lists. This results in a modified PEG notation that Spirit X3 leverages.

5.4 Parsing

The purpose of the parsing phase is to convert a stream of tokens, i.e., the user's program written in GCL, into an Abstract Syntax Tree (AST). Because Spirit X3 utilises C++ metaprogramming techniques along with operator overloading, it can perform both lexical analysis and parsing in one step. This approach contrasts with the traditional method of using Yacc/Bison with Lex/Flex. Conventional compiler-compilers or parser-generators have to perform an additional translation step from the source code to C++ code, Spirit X3 eliminates this need [7]. This significantly reduces the build process's complexity, decreases

development time, and improves the codebase's maintainability.

Spirit X3 works by reading the abstract syntax, i.e. the source code written in Giga Chad, and then running the tokenization process where the source code is dissected into a sequential stream of tokens which represent the basic units of GCL. The stream of tokens is then analysed according to the inline grammar rules defined in C++ and passed to Spirit X3, which in the Giga Chad compiler corresponds to the grammar on Figure 1. During parsing, the stream of tokens will be converted into an AST.

In plain Spirit, you would for each grammar rule, define how it is translated into the corresponding AST vertex, but this is time-consuming and less readable, so in Giga Chad, we utilise the Boost.Fusion library [6] integration in Spirit, which allows Spirit to automatically create the instances of the correct AST vertices for a given grammar rule that it parses. This means that through the use of Boost.Fusion, we populate our AST's vertices with information from the stream of tokens using only the inline grammar rules. Each vertex in the AST is represented in C++ by a struct. The result is that the AST accurately represents the source code's content and structure.

Spirit X3 is fast because it utilises C++ templates and operator overloading, and it facilitates code as documentation since the grammar that is given to Spirit is written in a modified PEG notation as seen in Table 3, which is self-documenting.

Spirit X3 provides the "expectation" operator denoted by, "`>`" as seen in Table 3. The expectation operator tells the parser not to backtrack, which allows us to enforce that certain rules must match at specific points in the parsing sequence; if the parser fails to match, it throws an exception instead of trying alternative parsing rules. Further, Spirit X3 also provides "special operators" i.e., operators not in standard PEG notation, for common tasks e.g., for lists separated by a given character, denoted by "`%`". The modified PEG notation of Spirit X3 is therefore a superset of PEG notation.

5.5 Spirit X3 vs Yacc/Bison

As mentioned Spirit X3 and traditional compiler-compilers like Yacc/Bison differ in several key aspects, even though they serve similar purposes of generating parsers. The integrated approach of Spirit X3 offers several advantages, like it eliminates the need for an additional build step, simplifying the overall compilation process. Additionally, it enables seamless integration with other C++ code, allowing for more flexible and efficient parsing logic. Further, the use of expression templates ensures that the resulting parser is, taking full advantage of C++'s compile-time capabilities, thus ensuring that the parser is optimized.

However, these differences also introduce some challenges and limitations compared to traditional compiler-compilers. In the following sections, we will explore these aspects in more detail, focusing on how Spirit X3 handles word boundaries and operator precedence differently from Yacc/Bison.

```
struct reservedkeywords : x3::symbols<std::string> {
    reservedkeywords() {
        add("if", "if")
            ("else", "else")
            ("while", "while")
            ("break", "break")
            ("continue", "continue")
            ("return", "return")
            ("int", "int")
            ("bool", "bool")
            ("true", "true")
            ("false", "false")
            ("new", "new")
            ("print", "print");
    }
} reservedkeywordsInstance;

const auto id_def = x3::raw[ x3::lexeme[( x3::char_("a
    ↪ -zA-Z_") >> * boost::spirit::x3::char_("a-zA-Z_0
    ↪ -9"))]] - (reservedkeywordsInstance >> ! x3::
    ↪ alnum) ;
```

Figure 9: Grammar: How the parser excludes reserved keywords as IDs, and fails if a reserved keyword is used as an ID

5.5.1 Word boundary

We store reserved keywords like "`if`", "`bool`" etc using a symbol table which Spirit X3 provides us. This simplifies the logic of checking that an ID is not a reserved keyword, i.e., valid, since we can use the symbol table directly in our inline rules with the modified PEG notation as shown in Figure 9. Notice how we parse using "`raw`" and "`lexeme`" [7, Section Parser Directives], the `raw` directive will capture the raw text of the input sequence matched by its argument, i.e., it will return the exact substring that matches the enclosed parser, without any transformation. The `lexeme` directive inside the `raw` directive ensures that no whitespace is skipped within the scope of its argument. This is crucial for IDs where any leading or trailing spaces must not be included in the token but do serve to define its boundaries.

Then using the reserved keyword along with the "except" operator denoted by "`-`" (as seen in Table 3) we exclude all reserved keywords from our IDs, but since Spirit X3 does not directly support a word boundary parser, we need to exclude reserved keywords that are not followed by alphanumeric characters, this is done by "`!x3::alnum`", such that we ensure that a reserved keyword is only recognized as such if it is not part of a larger ID.

For example, `int ifIAmCool = 42;` can still parse, but `int if = 42;` fails. This is only possible because we can use the "`!x3::alnum`" condition to enforce a word boundary, ensuring that the parser fails if a reserved keyword is used standalone. This allows us to fail quickly if an invalid ID has been used, such that we do not waste unnecessary computation.

As mentioned Spirit X3 does not have a built-in "word boundary" parser like regular expression engines do, this is yet another difference between Spirit X3 and Yacc. Yacc separates Parsing and tokenization (lexing), meaning that lexical analysis is handled by a lexer (like Lex) which uses regular expressions, and then a clear word boundary is defined by the lexer for the parsing phase. As mentioned Spirit X3 combines Parsing and lexing, thereby inlining the token definition within the grammar using C++ code, which is why the "`!x3::alnum`" operation of our ID rule is important, as it ensures a word boundary.

5.5.2 Precedence

Spirit X3 does not support any built-in directives to specify operator precedence and associativity like Yacc which has precedence directives [1]. Therefore, we need to structure our grammar hierarchically, such that the lowest precedence operators are parsed before the higher precedence ones. This is done by defining separate rules for different levels of precedence and then nesting them such that each rule builds upon the rule for the next lower precedence level, incorporating it to form increasingly complex expressions, as seen in Figure 10. This nested hierarchy ensures that the parser only expands into the ⟨`expression`⟩ rule at the appropriate level of the hierarchy, avoiding repeated evaluations of this rule for each token within arithmetic and logical operations. This both defines the precedence rules of GCL and optimizes the parsing process by reducing

the computational overhead involved in repeatedly parsing lower precedence expressions at each level.

The manual nesting of rules in Spirit X3 provides us with greater flexibility than Yacc provides with its built-in precedence directives, but it also comes with higher complexity in the grammar design. This increased flexibility allows for more nuanced control over how expressions are parsed, e.g., with Spirit X3, we can implement specific optimizations at each level of precedence by for example placing the most common operators earlier in the parsing process.

To model this structure in our AST, we have an AST vertex, that consists of a left-hand side and a list of all the right-hand sides, which are pairs of operators and expressions, i.e., we store the left-most expression and then a list of pairs:

$$\begin{aligned}&[\text{expression}_{L1}, (\text{operator}_{R1}, \text{expression}_{R1}),\\ &\ (\text{operator}_{R2}, \text{expression}_{R2}), \ldots,\\ &\ (\text{operator}_{Rn}, \text{expression}_{Rn})]\end{aligned}$$

in our AST vertex. This pattern is also apparent in the rules of Figure 10, in that each precedence level has one rule for the left-hand side and one for all the right-hand sides. Spirit X3 allows us to parse all of these rules into the same more general AST vertex instead of making separate AST vertices for each level, this enables us to encapsulate the precedence details in the parser phase of the GCC, thereby simplifying our AST for the later stages of the compiler.

Associativity among the operators is not implemented in the parser, instead, any arithmetic and logical operation is represented by the general AST vertex, consisting of a left-hand side and a list of all the right-hand sides, meaning associativity is determined by the order in which elements of the list are evaluated. This approach simplifies the parser phase and allows the code generation phase to encapsulate associative details, which is beneficial because the front-end of the compiler is not concerned with associativity, so by not implementing it until later, we simplify some of the logic in the front-end. The choice of the hierarchy of the precedence rules is meant to foremost mimic the precedence used in maths and secondly in other programming languages, i.e., mathematical operators have the highest precedence then comparing operators, equal operators and logical operators, such that `2 + 4 == 6` becomes `(2+4) == (6)`, as this is most natural.

6 Traversing the AST

The parsing phase outputs an abstract syntax tree (AST) which the next phases want to traverse to analyse the code. The traversal is done through a common system, based on the visitor pattern, so each phase does not have to implement it. The system consists of a "traveller" and a "visitor", where the traveller traverses the AST and then tells the visitor which vertex it has visited. The visitor will then perform phase-specific tasks based on which vertex is visited. The traveller performs a depth-first traversal of the AST, which also means that

⟨*logical_or*⟩ = ⟨*logical_and*⟩ ⟨*logical_or_rhs*⟩*

⟨*logical_or_rhs*⟩ = '|' ⟨*eq*⟩;

⟨*logical_and*⟩ = ⟨*eq*⟩ ⟨*logical_and_rhs*⟩*

⟨*logical_and_rhs*⟩ = '&' ⟨*eq*⟩

⟨*eq*⟩ = ⟨*comp*⟩ ⟨*eq_rhs*⟩*

⟨*eq_rhs*⟩ = ('==' | '!=') ⟨*comp*⟩

⟨*comp*⟩ = ⟨*term*⟩ ⟨*comp_rhs*⟩*

⟨*comp_rhs*⟩ = ('<=' / '>=' / '<' / '>') ⟨*term*⟩

⟨*term*⟩ = ⟨*factor*⟩ ⟨*term_rhs*⟩*

⟨*term_rhs*⟩ = ('+' / '-') ⟨*factor*⟩

⟨*factor*⟩ = ⟨*expression*⟩ ⟨*factor_rhs*⟩*

⟨*factor_rhs*⟩ = ('*' / '/' / '%') ⟨*expression*⟩;

Figure 10: Precedence directives going from lowest to highest precedence. This hierarchy is started by another rule that expands to <*logical_or*>.

```
while 3 > 5 {
    ...
}
```

Figure 11: A while-statement with condition `3 > 5` and unspecified code in the code block.

when it has visited one child of the vertex, it returns to the parent vertex so it can visit the next child and so on. When the traveller returns to the parent vertex, it tells the visitor that it has visited that vertex again, so the visitor can perform different actions on a vertex depending on which of its children have been visited already.

6.1 Visitor

The visitor is implemented by the `Visitor` interface, which is a collection of virtual "visiting" methods that a concrete `Visitor` implementation for each phase can override. Each of these methods has a default implementation that does nothing, such that a concrete implementation only has to override the ones it needs. `Visitor` has multiple such visiting methods for each AST vertex; one before any children have been visited, called `pre_visit`, and one after all children have been visited, called `post_visit`. When a vertex has more than one child, it also has a visiting method between each child, which all have context-specific names. For example, a while-statement vertex, as presented in Figure 11, has two children; the while-condition, i.e. `3 > 5`, and the code block. Therefore, it is visited before the "`while`", after the code block and before the code block, which corresponds to `pre_visit`, `post_visit` and between its two children, respectively. Thus, `Visitor` allows each phase to define when and what should be done in a traversal of the AST based on which vertex is being visited and which of its children have been visited. The general format of the `Visitor` interface is supplied in Figure 12.

6.2 Traveller

The traveller is implemented by the concrete `TreeTraveler` class. All phases that need to traverse the AST do it using `TreeTraveler` such that it is done invariably every time. The `TreeTraveler` class is cheap to instantiate, so each phase instantiates its own. Furthermore, since one phase should only perform a single traversal of the AST, it should also have exactly one `Visitor` implementation so `TreeTraveler` is instantiated with an unchangeable concrete `Visitor` implementation.

As seen in Figure 13, the traversal of the AST performed by `TreeTraveler` is invoked using the function operator, which has multiple overloads to handle AST vertices directly and to handle different wrappers that an AST vertex might

```
1  class Visitor {
2  public:
3      virtual void pre_visit(Prog &prog) { }
4      virtual void post_visit(Prog &prog) { }
5
6      virtual void pre_visit(WhileStatement &
       ↪ while_statement) { }
7      virtual void pre_block_visit(WhileStatement &
       ↪ while_statement) { }
8      virtual void post_visit(WhileStatement &
       ↪ while_statement) { }
9
10     // (...)
11 }
```

Figure 12: The `Visitor` interface, which consists of visiting methods for every AST vertex. The interface is rather large, so only a part of it is provided, but the remaining methods in the interface follow the same pattern as the few methods provided. Some namespaces were left out for readability.

be contained in. The templated function operator on line 18 on Figure 13, is responsible for handling each AST vertex. The method is then specialised for each AST vertex (the specialisations are not apparent in the figure) which works nicely because it throws a compile-time error if the method is called using an unsupported type, acting as a reminder to create a specialisation for the method when introducing a new AST vertex type into the compiler. Each specialisation of the method calls the appropriate methods in `Visitor`, while recursively calling the function operator on each child in the vertex. Since a specialisation exists for every vertex, if the function operator is called on the root of the AST, `TreeTraveler` will traverse through the entire AST. For example, the specialisation for the while-statement vertex is shown on Figure 14.

The other three function operators in `TreeTraveler` are used to handle common wrappers for the AST vertices, for example, if a vertex might not be there, it will be wrapped in a `boost::optional`. To gain access to the vertex, it needs to be unwrapped before it can be visited. This process could require a couple of lines of code, which is not ideal since the extra lines of code lower readability and these wrappers occur relatively often. The function operators for the wrappers alleviate this issue entirely, as the function operator can be called directly on any of these wrappers, and they will handle the unwrapping.

`TreeTraveler` entirely dictates how the AST is traversed, but is not customisable in how the traversal is done. When it starts its traversal on an AST vertex, it will always traverse the entirety of that vertex's sub-tree. This means that `TreeTraveler` essentially forces a `Visitor` to visit the entire AST when it is started on the root. This has the downside that, since each phase that

```
class TreeTraveler : boost::static_visitor<> {
private:
    Visitor &visitor;

public:
    TreeTraveler(Visitor &visitor) : visitor(visitor)
    ↪ { }

    template <typename T>
    void operator()(boost::spirit::x3::forward_ast<T>
    ↪ &ast);

    template <typename T>
    void operator()(boost::optional<T> &ast);

    template <typename T>
    void operator()(std::vector<T> &t);

    template <typename T>
    void operator()(T &t);
};
```

Figure 13: The declaration of `TreeTraveler`. It has a constructor that takes a `Visitor` that is saved in a private attribute. It has four overloads of the function operator, where each handles their special cases of parameters.

```
template <>
void TreeTraveler::operator()(WhileStatement &
    ↪ while_statement) {
    visitor.pre_visit(while_statement);
    (*this)(while_statement.exp);
    visitor.pre_block_visit(while_statement);
    (*this)(while_statement.block);
    visitor.post_visit(while_statement);
}

```

Figure 14: The specialisation of the `TreeTraveler` function operator for the while statement. It first calls the `pre_visit` function on the `Visitor` and recursively enters the while statement's expression (i.e. the while-condition). The same is done for `pre_block_visit`, and the code block inside the while statement, and finally `post_visit` is called on the `Visitor`.

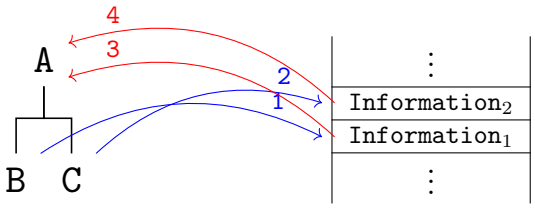


Figure 15: Schematic of the producer/consumer scheme. Displaying the AST and the stack containing information from the AST vertices. The blue arcs represent data being produced and stored on the stack, while the red arcs represent data being consumed from the stack. Each arc is numbered to indicate the order in which the `Visitor` visits these vertices and performs the action. The two lower level vertices `B` and `C` produce information that their parent `A` then consumes to perform some action.

traverses the AST uses `TreeTraveler` and `Visitor`, the phases do not have control over how the AST is traversed which limits the ways that some phases can be implemented which will be discussed later in detail when relevant.

Thus, `TreeTraveler` handles the traversal of the AST and makes the relevant calls to the `Visitor` that it is given at construction. This setup allows each phase to only implement a `Visitor` while still being able to traverse the entire AST and define custom code that should happen at each step of the traversal.

6.3 Propagating Data in the AST

In each phase of the compiler, the `Visitor` responsible for that pass through the AST will perform the specific actions necessary in that phase to ensure the code adheres to the semantics of GCL. To achieve this, the `Visitor` will need information from the previous `Visitor` or previously visited vertices in the AST. Therefore, access to prior context is necessary to perform these actions.

6.3.1 From Visitor to Visitor

To enable information sharing from `Visitor` to `Visitor`, i.e., from different passes through the AST, some vertices will hold additional metadata, e.g., a vertex representing a function could hold its scope additional metadata. The Spirit X3 library is not responsible for setting this metadata when parsing the code, as the `Visitor` will gather this information during its pass through the AST and then add it to the relevant vertex such that another `Visitor` in a different pass through the AST have this information available. Only the necessary AST vertices contain this additional metadata, since supplementary actions or more complex actions must be performed on these vertices.

6.3.2 From Visit to Visit

To enable a single `Visitor` to save information it gathers between its visits on the different vertices in the AST, a stack is used to hold the information. The `Visitor` will push the necessary information on the stack, that it needs in another visit. This action allows the `Visitor` to store information that it gathers from child vertices in the AST and then access it in the ancestor vertices. The `Visitor` will visit all vertices in the AST, but only the necessary vertices visits have an associated action, i.e., some visits will do nothing while others will aid in performing the necessary actions to ensure that the code is correct.

Utilizing the `Visitor`, and the stack data structure, we can set up a modularized and easily extendable information storing scheme, that we call a *producer/consumer* scheme. The schematic in Figure 15 demonstrates the structure of the scheme. It is quite simple, lower-level AST vertices, are responsible for pushing information onto the stack as they are visited, i.e., they produce type information to the stack. Then higher-level vertices subsequently pop this information from the stack to perform their actions, i.e., they consume information from the stack that they need.

Some vertices in this scheme act as both producers and consumers, which we call `producer-consumer` vertices. This means that a vertex, such as `A` in Figure 15, might consume information from its children to perform some intermediate action, which then produces information that its ancestors will need. Thus, it both consumes information and produces information. A side effect of the producer/consumer scheme is that some vertices might produce information that their ancestors did not need. This happens because the context of a vertex in the AST is not considered during a visit. For instance, a producer vertex might be a child of two different parent vertices in different contexts. For one parent, the child produces necessary information, but for the other parent, this information is not needed. This results in the stack being polluted with useless information. To address this, some special consumer vertices are visited to clean up the unnecessary information left by their children.

This producer/consumer scheme is inherently scalable when introducing new AST vertices into GCC. For instance, if a new vertex consists of child vertices that are already handled by existing logic, the new vertex needs only to manage the popping (consuming) of this information from the stack, thereby minimizing the need to reimplement or alter underlying logic. Instead, adjustments are localized to the specific behaviours of the new or modified vertices with this scheme.

7 Semantic Analysis

The semantics of a language refers to the meaning of the sentences it produces, therefore semantic analysis in a compiler aims to check that the meaning of the code it's given is valid. This is a process that can only be done after the parsing phase so it knows that the code is given in a valid syntax, and should

particularly be done immediately following the parsing phase because the next phases rely on the semantics of the code being correct. Semantic analysis is split into two processes; symbol collection and type checking. Symbol collection is the process of collecting the names of all the declared variables, functions and classes, as well as their types and location, while type checking makes sure that these are not used with invalid types, for example, that you attempt to assign a boolean to an integer variable. This section will cover how symbol collection and type checking work in GCL.

7.1 Symbol Collection

A core concept within symbol collection is that of a symbol. A symbol is a unique representation of a given variable, function, or class. It is important to keep track of symbols because there can be multiple variables, functions, or classes that have the same identifier (when they are not in the same scope) and thus it can be non-trivial to deduce what an identifier exactly references. For example, suppose you have a variable with the identifier `x` in two scopes, and you encounter `x` again somewhere else. You need some way to know which variable it references without much effort, since it needs to be done many times, especially when the reference to the variable is in a different scope than the scope it was declared in. Since symbols are unique representations, they can aid in the process of finding exactly what is being referenced by an identifier. Symbol collection is responsible for linking an identifier to the symbol it references. A symbol also contains additional information based on what it represents; for a variable, it has a type, for a class it has the name of the class and for a function, it has the return type and the types of the parameters, as well as the number of parameters. The succeeding phases could obtain this information through other means, but since it is commonly used information, it is stored in the symbol for convenience.

7.1.1 Retrieving the Symbols

The other phases need some way to obtain the symbols when they encounter an identifier. One way is to store a symbol on the `Id` AST vertex, but sometimes; however, a phase needs a symbol without having an `Id` vertex for it, which is where symbol tables are useful. A symbol table is a mapping from an identifier to a symbol, such that there can only be one corresponding symbol in the symbol table for any given identifier. Every scope has its own symbol table where a scope is a block of code for a function, in addition to the outermost set of declarations, i.e., the ones directly inside the `Prog` vertex, called the global scope. The symbols for all variables, functions, and classes that are declared in a scope are put into that scope's symbol table, mapping their identifier to their symbol. This means that you can reuse the same identifiers in nested functions, since these will have separate symbol tables. The symbol tables are set up in a tree-like structure such that if a scope is within another scope (for example with nested functions), then the inner-scope's symbol table has a reference to

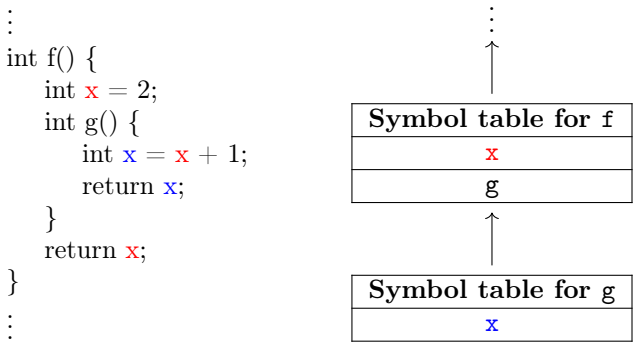


Figure 16: A code example with two functions with arbitrary code around them and the two corresponding symbol tables. The `f` function's symbol table has an entry for `x` and one for `g`. The symbol table for `g` only has an `x` entry. In the code, the `x`'s that reference the `x` in the symbol table for `f` are coloured red, and the ones referencing the one in the symbol table for `g` are coloured blue. The symbol table for `g` has a reference to the symbol table for `f` since `g` is inside of `f`.

the outer-scope's symbol table, also called the parent symbol table. This is necessary because GCL has static linking such that if a symbol table does not contain a symbol for an identifier, it needs to look in its parent symbol table for it which is then done recursively until the symbol is found, or you reach the global scope symbol table, and it doesn't have it either.

For example, in Figure 16, if a lookup for "x" is done in the symbol table for `g`, it will find `x`. If a look of "g" is made instead, then it will find that it does not have a "g" itself so it goes to its parent symbol table, which is `f`'s symbol table, and then it will find the `g` in there. If a lookup of "x" is made in `f`'s symbol table instead, it will find x. Another example of a use case for symbol tables is if a phase wants to find the symbol for the `main` function, it can look up in the global scope's symbol table with "main" and then find the corresponding symbol.

7.1.2 Collecting and Checking for Symbols

A symbol should be collected, i.e., put in a symbol table, when a declaration for the corresponding variable, function, or class is encountered. When an identifier for a variable, function or class is encountered, it is checked whether it has been collected yet by looking it up in the symbol table for the current scope, and it is checked that it is the correct type of symbol so that you, for example, cannot use a variable like a function call. If it has not been collected, it is assumed that the identifier does not have a symbol in the code, and thus is an error in the code, for example, if a variable is used but has not been declared.

The declarations and usages of identifiers are encountered by symbol collection through a `Visitor` using `TreeTraveler`. If a declaration is in the GCL code before its usage, this is allowed since the `TreeTraveler` traverses the AST vertices in a top-down order in the code. If a variable declaration is instead found in the GCL code after its usage, this will result in an error, since the `Visitor` first will visit the usage, before the declaration and will thus find that it has not been collected yet. This is desirable according to the semantics of GCL when it comes to variables but with functions and classes, we need their declarations to also be able to come after the usage of them. This is achieved by creating a second `Visitor` that will make a second pass over the AST after the first `Visitor` in symbol collection. With this scheme, the first `Visitor` will collect all symbols by visiting the declarations of variables, functions, and classes. Additionally, it will also visit the usages of variables and verify that they have been collected such that using them before declaring them is disallowed. The second `Visitor` will then use the symbols collected from the first `Visitor` and will visit the usages of functions and classes to verify that they have been collected such that they just have to be declared at some point in the code (and a reachable scope) to be used.

7.2 Break and Continue

Recall that in GCL, break and continue statements are only allowed to occur within while-loops, where you also need to consider nested while-loops and that there can be nested functions where a break- or continue statement is not allowed to reference outside the function. The compiler checks this by keeping track of a stack of integers, where there is one integer for every function we are currently inside of, to support nested functions, and the integer represents the number of nested while-loops we are inside of. Since a while-loop is a statement, we know that in GCL it can only occur within a function, so having an integer for every function is sufficient. Therefore, when pre-visiting a function declaration, an integer is pushed to the stack and when pre-visiting a while-loop, the integer at the top of the stack is incremented. When post-visiting a while-loop, the integer is decremented again and when post-visiting a function declaration, the integer is popped from the stack. When encountering a break- or continue statement, it can be checked whether it is inside a while-loop by retrieving the top integer in the stack and then checking whether it is equal to or greater than 1.

7.3 Type Checking

The type-checking phase of compiler design is where the compatibility of data types used throughout the language is verified. It ensures that the usage of the language is semantically meaningful according to the programming language's rules and constraints. At its core, type checking involves analysing the AST, ensuring that the types of parent AST vertices are compatible with the types of their children. This check establishes the integrity and correctness of the

```
int main() {
    bool f() {
        ...
        return true;
    }
    return 0;
}
```

Figure 17: Example demonstrating the limitations of the two-attribute solution compared to the stack solution for nested functions.

program, while also preventing type errors from propagating into the runtime environment.

Our approach to type checking leverages the producer-consumer scheme described earlier in, involving a systematic traversal of the AST using the `Visitor`. The `Visitor` visits all vertices of the AST primarily in a post-order manner, allowing type information to flow from the children to their parents, effectively traversing the AST from the bottom up. For some special vertices, the `Visitor` may use a different visiting order to accommodate specific type checking needs. Utilizing the producer-consumer scheme it allows for flexible handling of type checking with both scopes, nested structures, etc. In the following sections, we will describe the traversal order of the `Visitor`, for special traversal orders and its purpose, detail the rules and constraints of GCL, and explain the strategies employed to ensure reliable type-checking.

7.3.1 Why a Stack?

Utilizing the producer-consumer scheme for type checking raises the question: "why use a stack instead of simpler data structures like attributes?" Two attributes might seem easier to manage and more space-efficient.

Consider using two attributes in the following way: attribute A stores the last evaluated expression type, and attribute B stores an expression type for a longer duration. These attributes are then checked against each other during type checking. However, this approach encounters significant issues, as illustrated by the example in Figure 17: In this example, using two attributes means we must store the return type of `main` in attribute B, since this value is needed when reaching the return statement of `main`. However, attribute B would be overwritten with the return type of `f`. By the time we reach `return 0;` in `main`, attribute B holds a boolean type, causing a type-checking error despite the program being valid.

This highlights a major flaw in using attributes: each nested function requires a new attribute to hold its information. Since we cannot know in advance how many attributes are needed, a dynamic data structure like a stack is preferable. A stack can handle arbitrarily complex and deeply nested ex-

pressions, preserving context without overwriting information. This scalability makes it well-suited for the recursive nature of the visitor pattern and AST traversal.

7.3.2 Handling Different Types of Vertices

Vertices such as those representing an expression in the AST can assimilate various types of constructs. This abstraction enables us to represent a wide range of AST vertices within the AST in a single AST vertex. During compilation, these vertices assimilating various vertices are evaluated in their specific context and set to the appropriate AST vertex.

These vertices will not contain any actions because they are "pseudo" vertices in the sense that once evaluated they take the type of other AST vertices, e.g, an expression can become an integer, so the action needs to be on the visit of an integer and not an expression. The `Visitor` will never perform any actions on these "pseudo" vertices, only on the vertices that they can assimilate, such that the correct actions are performed in the correct context.

Because of this, conceptually in the AST, a "pseudo" vertex will perform the producer/consumer actions of the vertex it will eventually embody. For instance, as shown in Figure 19, an `Expression` vertex is schematized as producing information for its parent and having multiple unknown children. Conceptually, it should just be thought of as an expression, which could have children but will always produce information. During compilation, the `Expression` is evaluated in its context and will take the type of the correct AST vertex. However, in the general schematic of the AST, the specific context is unknown.

7.3.3 Root of the AST

The root vertex of the AST is special, as it is the first and last vertex the `Visitor` visits. Therefore, special actions are taken on this particular vertex to enforce the semantics of the GCL and to confirm that the AST is processed correctly. One such action is a pre-visit on the `Prog` vertex, which represents the program written in GCL and serves as the root of the AST. During this pre-visit, the `Visitor` checks that the program contains a `main` method with a return type of integer. This check ensures the presence of a `main` method early in the compilation process, allowing the compiler to fail fast if this requirement is not met. If this requirement of the GCL is not met, failing early is beneficial, as it prevents the compiler from wasting resources on further processing of an invalid program.

To reduce errors during development, we decided to assert in the post-visit of the `Prog` vertex that the stack should be empty. This assertion ensures that the relationship between the producer/consumer vertices is correct. If the scheme is set up properly, the stack will be empty once the `Visitor` has finished visiting all vertices in the AST. One advantage of this assertion is that it simplifies reasoning about the correctness of our approach. It ensures that all type of information pushed onto the stack has been collected.

7.3.4 Producer Vertices

For type checking, the producer vertices will push their types onto the stack, acting as data sources for their ancestor vertices to use for performing type checking actions.

- The `Id` vertex has a reference to its symbol, which contains its type
- The Spirit X3 library converts integer and boolean values into AST vertices. This allows the `Visitor` to visit integer and boolean types, even though no specific struct exists for them.

Looking at Figure 19a, the `Id` vertex produces information directly for `VarDeclAssign`, since no type checking action is performed on its parent. This is a common pattern, as vertices with no action can be skipped, and can thus let type information propagate further up the AST.

7.3.5 Producer-Consumer Vertices

The producer-consumer vertices will consume information from the descendants to do some intermediate type checking and then produce new type information based on these checks that they will send to their ancestors in the AST. These vertices can produce type checking errors during the intermediate type checking process.

- A return statement is represented in the AST as `ReturnStatement` vertex. This vertex will check that a function returns the type stated in the function header. Since multiple computational paths could exist, and thus numerous return statements, the `ReturnStatement` will lastly push the type of the function back onto the stack. This action ensures that return statements in other computational branches can also access and verify the return type against the same function declaration.
- the return statement produces type information, therefore the last visited return statement will always pollute the stack, thus the AST vertex `FuncDecl` representing a function declaration in GCL e.g., `int f(int x, bool y) return 0;` will clean up the type information from the last return statement in a post visit. Additionally, the `Visitor` will perform a pre parameter list visit, which acts as a mid-visit, where it will clean up the stack before visiting the vertex representing the function's parameters. Furthermore, the `FuncDecl` vertex will be pre visited by the `Visitor` to add its own type to the stack for its ancestors
- Function calls are repressed in the AST by the `FunctionCall` vertex. The `Visitor` will post-visit the `FunctionCall` vertex and perform two checks. First, it will verify that the object being called is a valid function. Second, it will check that the number of arguments passed matches the number of parameters expected by the function. The descendants of the `FunctionCall` vertex have pushed all the necessary type information

onto the stack, but this information is in reverse order. Therefore, an alignment is necessary to ensure that each argument is checked against the corresponding parameter. For example, a function `int func(int x, int y, bool z)` can be called like this: `func(2, 4, true)`. The types of the arguments *bool, int, int* (in reverse order) will be checked against the types of the parameters `2, 4, true`, starting from position $i = 0, 1, \ldots, n$ in arguments and position $n - i$ in parameters for alignment. Lastly, the function call vertex will push its return type to the stack.

- For instantiating a new object in GCL, i.e., `new T();` for instantiating a new object named `T`, the `ObjInst` vertex represents this construct in the AST. It checks that the class that is being attempted to be instantiated is, in fact, a class, if this is the case, it will push the type representing the instantiated class onto the stack.

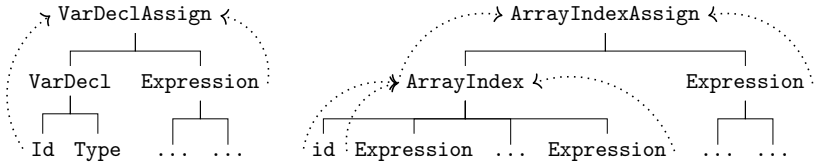
- To represent array instantiating in the AST a struct `ArrayInitExp` is used. This vertex will check that only integers have been used to specify how much space should be allocated for each dimension of the array. If this check passes, it will push a type representing the instantiated array onto the stack.

- The struct `ArrayIndex` in the AST represents indexing into an array, i.e., `arr[2]` in GCL. This vertex checks whether the user is indexing on a compatible type, i.e., using the '`[]`' operator on a variable of type array and the type of the index is integer. When indexing into an array the user will need to explicitly give the index for each dimension, thus the number of indices corresponds to the number of dimension if it is a correct indexing of an array. On a valid array index, the vertex will push the type of the element on the index onto the stack.

- Arithmetic expressions are represented as an expression (that could be either integer or boolean) that corresponds to the left-hand side and then by a list of right-hand side which corresponds to the AST vertex `Rhs` (right-hand side) in the AST. This vertex contains both the operators and the right-hand expression. The `Visitor` will post visit the `Rhs` to check that the types of the left and right-hand side of the arithmetic expression match, i.e., `5 + false` would not be a valid arithmetic expression in GCL. An additional check is done to check that the operators between the left-hand side and right-hand side are used correctly, i.e., "`true == 2`" or "`4 & 2`" will throw a type-check error, since it does not comply with the semantics of GCL. Lastly, this vertex will produce a type based on the evaluation of the left and right-hand side and their operator, i.e., "`4 != 2`" will produce a boolean type that will be pushed to the stack, but "`4+2`" will produce an integer type. This is done to recursively handle arbitrarily long arithmetic expressions, e.g., "`4 > 2 | true`" by evaluating the expression like "`((4 > 2) | true)`".

```
Obj1.Obj2.····.ObjN.x
```

Figure 18: Example of `IdAccess` chain being arbitrarily long. The right-most link in the chain can have all types, this would be the variable `x`, while all objects left of `x` have to be of a class type.



(a) Schematic of AST vertex `VarDeclAssign` (b) Schematic of AST vertex `ArrayIndexAssign`

Figure 19: Figure 19a and Figure 19b shows schematics of two vertices in the AST. The dotted lines show type information being produced and sent upwards the AST, by producer vertices. The vertices that the dotted lines point to are either producer/consumer or consumer vertices, i.e., they consume the type information to make perform type checking. `Expression` is a "psuedo" vertex as it will on compile time be assigned the correct type based on the context, and thus this vertex might have none or multiple children.

7.3.6 Consumer Vertices

The consumer vertices ensure that operations are type-safe and adhere to the GCL's semantics. This is done by performing type checking actions using the information gained from their descendants. Additionally, consumer vertices are responsible for throwing type checking errors if any issues arise.

- Both the `IfStatement`, which represents if and else if statements, and the `WhileStatement`, which represents a while statement, check that their guards are of type boolean. This is done in a pre-block-visit, meaning that the `Visitor` will perform this action before visiting the body of the statements.
- A variable declaration assignment, represented by the `VarDeclAssign` vertex, checks that the variable type matches the type of the expression it is being assigned. The schematic in Figure 19a illustrates how this works. For example, in `int x = 2;`, the variable `x` has the type `int`, and the expression `2` is evaluated to the type `integer`. The `VarDeclAssign` vertex is then responsible for checking if these types match.
- The `IdAccess` vertex, which represents a list of `Id`s in a chain of '`.`' operators. This vertex will check the types of all the `Id` vertices in the list besides the last one. Because a valid '`.`' operator is used on a class type only, the right-most objects type can be of any type, hence it is not

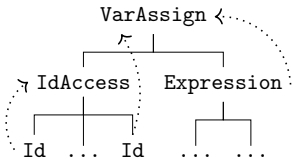


Figure 20: Schematic of the AST vertex `VarAssign`, showing its descendants. The dotted arcs show produced type information propagating upwards in the AST to the vertex that consume it. The `IdAccess` vertex only consumes type information from all its children besides the last, which sends it type information to `IdAccess` parent, as shown on the schematic. `Expression` is a "psuedo" vertex as it will on compile time be assigned the correct type based on the context, and thus this vertex might have none or multiple children.

checked. The Figure 18 shows an example of such a chain being arbitrarily long, only at the last '`.`' operator between `ObjN` and `x` can we be sure that another type besides class is allowed, i.e., we do not check `x` since it is allowed to be all types, but all previous variables must be of type class.

- The `VarAssign` vertex represents a variable assignment, and therefore it checks that the type of the variable matches the type of the expression it is being assigned. A valid variable assignment for a class object `t` with a member `x` could be the following `t.x = 42;`. Looking at the schematic Figure 20 the `IdAccess` vertex would have two children namely `t` and `x`, where it performs type checking on `t` and let the type information from `x` propagate upwards to its parent `VarAssign`. The `Expression` vertex would be assigned the type of the `integer` vertex in this context, and therefore send type information upwards the AST also, enabling `VarAssign` to perform type checking.

- The `ArrayIndexAssign` vertex represents assigning a value to a specific element in an array, e.g., `a[2] = 3;`. This vertex must check that both the type of the index and the type of the assigned value match. This construction is schematized in Figure 19b, where an intermediate type checking action is performed by the producer-consumer vertex `ArrayIndex`. The information passed by this vertex ensures that the indexing is semantically correct, so only the type of the assigned value needs to be checked.

- Print statements are repressed in the AST by the `PrintStatement` vertex. This vertex checks that the expression to be printed is of type boolean or integer. The programming construction that represents the type and dimensions of an array, such as "`int[1]`" for GCL, is represented in the AST as `ArrayType`. This vertex contains a member for the number of dimensions, which is assigned during the parsing phase as an integer. The parser enforces the constraint that the dimension must be an integer.

This vertex is one of the special clean-up vertices mentioned earlier, so it is visited to clean up the stack. Additionally, this vertex checks that the dimension is a positive integer of size one or more. Note that as the dimension can only be a literal integer, it is known on compile time thus this information is saved in the AST vertex and not on the type stack as it is gathered in an early phase.

- To keep the stack clean during the producer/consumer scheme, additional vertices in the AST also act as special clean-up vertices. The vertices `Parameter`, `VarDeclStatement`, `StatementExpression`, and `ClassDecl` are all clean up vertices. They represent the following programming constructs respectively: a parameter in a function declaration, a variable declaration with an ending semicolon, an expression with an ending semicolon, and a class declaration.

7.3.7 Enforcing Return

The semantics of GCL dictate that a function should always return a value, i.e., all computational paths must lead to a return value. To enforce this, the type checking phase will trigger a second pass through the AST with a specialized `Visitor` that will recursively traverse the code to evaluate if all computational paths lead to a return value. This specialized `Visitor` will pre visit a function declaration and examine the code in its body. The `Visitor` guarantees that a function containing a return statement will always return, as shown in Figure 21b. Similarly, an if-else statement that returns in all branches will also always return, as illustrated in Figure 21a. For any other cases, it cannot guarantee this and will, therefore, respond with false. Using this specialized `Visitor`, we can determine whether all possible computational paths will lead to a return statement. If all paths lead to a return statement, we can conclude the program is correct because the type checking phase has been completed, and thus each return statement in each computational path is semantically correct.

8 Code Generation

The goal of our code generation is to translate the AST into an intermediate representation (IR) of the code that closely resembles the AT&T syntax assembly language. The IR serves as a midpoint between the AST and the final assembly code, capturing the essential operation and control flow of the program while still abstracting away some low-level details that should be in a later phase, such as specific register assignments. This approach delegates many of the responsibilities to later phases, preventing the code generation phase from becoming too complex and monolithic.

During the code generation, all vertices in the AST are visited and converted into the equivalent IR instructions, while still taking into account relevant factors such as the control flow and data types.

```
int main(){
    int f() {
        int x = 2;
        if (x == 1) {
            return 2;
        } else if (x ==
    ↪ 4) {
            return 2;
        } else {
            return 42;
        }
    }
    return f();
}
```

(a) Guarantee of if-else returning

```
int main(){
    int f() {
        int x = 2;
        if (x == 1) {
            return 2;
        } else if (x ==
    ↪ 4) {
            x = 42;
        } else {
            x = 42;
        }
    return 42;
    }
    return f();
}
```

(b) Guarantee of function body returning

Figure 21: Example of how return statements are forced by a specialized `Visitor`, that only guarantees that a function body containing a return will return and if-else statement returning in all blocks will return.

In this section, we will delve into the specifics of the conversions and how the conversion between the AST and the intermediate code representation is done

8.1 Intermediate Code Representation

An intermediary instruction consists of an operation and its arguments, along with a specification for direct, indirect and indirect relative memory access. The arguments have been carefully designed to eliminate the need for explicit register specification at this stage. We introduce the generic register which serves as a temporary holder for the data, which will in a later phase be converted into actual registers or stack locations. This will allow for later optimisations, like a smart register allocation scheme, to be applied independently of the code generation phase.

While this is useful, some instructions need specific registers for operating, like `%RAX`, but also specific values like a label or an immediate value, and therefore our IR also has the ability to specify concrete register of values when necessary. The use of generic registers means that not every machine instruction can be directly generated from the AST. To handle this, we use what we call a procedure as an operation, such as caller-save operations, which push the registers used by the callee onto the stack for later restoration. However, since the actual registers used are not known at this stage, naively pushing all caller-save registers onto the stack would be suboptimal. Instead, we delegate

the responsibility of this to a later phase, where the necessary information is available.

8.1.1 Default Initialization

In our language, we decided we wanted to use default values over undefined behaviour regarding uninitialized variables. To facilitate this, we decided to implement default values for all of our variable types, and to initialize any variable to these default values when declared. Thus, using a variable before assigning a value to it, no longer has any undefined behaviour, as a variable will always have a value assigned to it. For integers the default value is `0`, for booleans it is `false`, for classes it is `beta`, and for arrays the default value is also `beta`. The value `beta` being our implementation of what would typically be called `null` in other programming languages.

Whilst simply declaring a new variable that is of type class will only default initialize the variable itself to be `beta`, and not actually default initialize any members as there is no object yet present. Should the variable be assigned an actual value by instantiating a new object and assigning the object as the value of the variable, the members of this new object instance, will then be default initialized.

Default initialization for arrays functions similarly, simply declaring a new array, will only set its value to be the default value of `beta`. When the array is then eventually initialized by creating a new instance of an array, then all the elements inside the array will be initialized to their respective default values. Of course, this only goes for when an array is initialized by only giving the size of the array, not when an array is initialized by giving it actual values to hold. Variables are default initialized only when the scope they are declared in is entered, and before any other code in the scope is executed, ensuring that any variables declared in the scope cannot be accessed before they have been initialized.

8.1.2 Unique Labels

In the assembly code that we generate, we use labels for functions, if statements, and while statements. These labels are generated automatically, to ensure that they are unique and that a user of our language cannot accidentally create a function with the name of an existing label.

Labels for functions are generated by taking the name of the function, and prefixing it with `Lid_`, as can be seen here for a function named 'f': '`L0_f:`' The '`id`', is then an integer ID that is unique across all labels. Because '`id`' is unique across all labels, we ensure that a user cannot end up naming a function the same thing as a label we generate, and thus we avoid duplicate labels. Should a user of our language try to create a function name with the prefix we generate, the label that would be generated for said function, would effectively have our label prefix twice, like in this example: '`L0_L0_functionName`'. For `if` and `while` statements we use the same approach, but instead of using the

name of a function, we simply prefix `if` and `while` respectively with the `Lid_` prefix, as can be seen in the following examples: '`L1_if_statement:`', '`L0_-while_statement:`' `If` and `while` cannot be used as function names to break this scheme, as they are both reserved keywords in our language, and thus attempting to use them as function names would cause a compile error.

Another approach we could have taken to creating unique labels, would be to reserve some prefix, and simply throw a compile error if the user tried to include the reserved prefix in a function name. But we decided not to go with such an approach, as we would rather not limit the user in how they are allowed to name their functions, such a limitation would also be quite unintuitive for the user. By simply prefixing all function names, we ensure that any information about how labels are kept unique is abstracted away from the user, whilst also ensuring freedom for how the user chooses to name their functions. Unique labels cannot be generated during the code generation phase, as in GCL we support calling a function before declaring it. This means that we must have generated the label for a function before we perform code generation, either a functional call or a function declaration. And since all labels must share a unique ID, this means that all labels must be generated during the same phase. Ideally, this would be done in its own separate phase right before code generation, but we decided to instead include it in the first symbol collection phase, to save a pass over the entire AST.

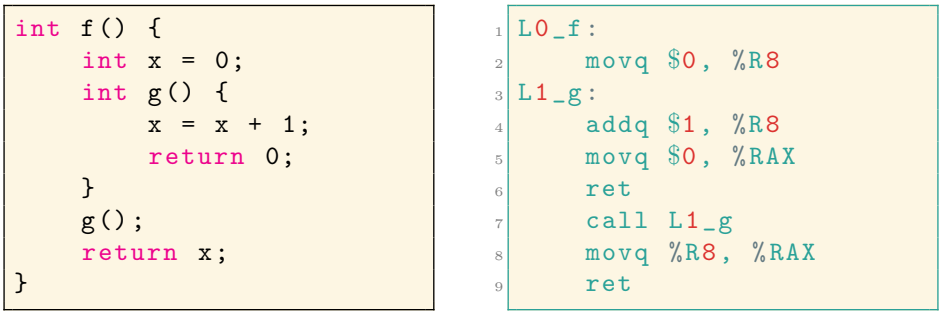
8.1.3 Global Scope

In GCL, we consider the global scope as a function for the purpose of keeping track of the current scope, and the order in which different scopes are translated into their intermediate representation. But global scope is not considered an actual function when it comes to emitting assembly code, instead we regard the standard entry point, `_start`, section of an assembly file, as the global scope of GCL. So any code written in the global scope, would be performed in the `_start` section of assembly, and therefore always executed before the `main()` function in GCL would.

8.1.4 Global Variables

In our language we allow for variables to be declared and initialized in the global scope, we even allow for variables declared in the global scope, to be initialized by the return value of functions. As all variables are default initialized when the scope they are declared in is entered, attempting to initialize a variable in the global scope with the return value of a function is allowed, and in such cases the functions used to initialize a global variable, are run before the main function is run. The order these functions are run in, is the order in which the global variables they are used to initialize, are declared in.

Functions can be called like this because the labels for functions are generated before any code is generated. So even though the code for calling a function might be generated before the code for the declaration of the same



```
int f() {
    int x = 0;
    int g() {
        x = x + 1;
        return 0;
    }
    g();
    return x;
}
```

(a)

```
L0_f:
    movq $0, %R8
L1_g:
    addq $1, %R8
    movq $0, %RAX
    ret
    call L1_g
    movq %R8, %RAX
    ret
```

(b)

Figure 22: The code in (a) is written in a language that supports nested functions. The function `f` should return 1. However, a compiler without proper function ordering may generate incorrect assembly code, as shown in (b), which would return 0 instead of 1, assuming that `f` is called by some other function.

function is generated, the function call and declaration will use the same label, thus ensuring the correct function can be used to initialize global variables.

8.1.5 Standard Functions

Currently in GCL we have seven "Standard functions", that is, functions that are always included in the emitted assembly file. One of these is for allocating memory, the other six are various functions for printing. Currently, the way these are being generated, is by appending intermediate representations to the list of intermediate representation code generated during the code generation phase of the compiler for GCL. But, as these files are always added to any code we generate an intermediate representation for, and given that we also know exactly what instructions need to be performed for these two functions to perform their intended functionality, we had the option to instead emit them straight to the end of the final assembly code emitted by the emitter phase. We chose not to do this, as generating an intermediate representation instead, will let us optimize this intermediate representation in the future, when we make any general optimizations that affect the entirety of the intermediate code generated in the code generation phase.

8.2 Function Manager

When generating code, the compiler typically processes the user's code in the order it is written. However, this approach can lead to problems when dealing with nested functions. Consider the example shown in Figure 22.

To address this issue, we implemented a function order manager using a stack-based approach. Every time a function is entered, an entry is added to the stack for that function. When a function is then exited, the top entry is

popped from the stack. Thus, the top entry is always the function that we are currently inside of and the stack in its entirety represents all the nested functions that we are currently inside of. Therefore, new code that is generated is always added to the function at the top of the stack. At the end of code generation, the functions that were added to the stack are returned in a linear fashion such that the nested nature of the code is not reflected in the returned code. This approach ensures that functions are not defined within each other, and it also offers the benefit of reflecting the same code structure in the assembly code order as the code was written in GCL. This makes it easier to navigate and understand the assembly code, as the function definition order in the original code is preserved in the generated assembly.

8.3 Activation Record

Activation records are a key concept in compiler design, they are essential to managing the scopes of functions. They ensure each function has its own space on the stack to store its local variables, arguments, and other necessary information. This also includes a link to a parent activation record, so not only the local variables can be accessed but also outer variables. Activation records are also a key part of managing the flow of execution by keeping track of the caller's context, when calling a new function, so the program can resume execution in the caller's function after the callee completes its execution. See Figure 23 for an illustration of the layout of our activation record.

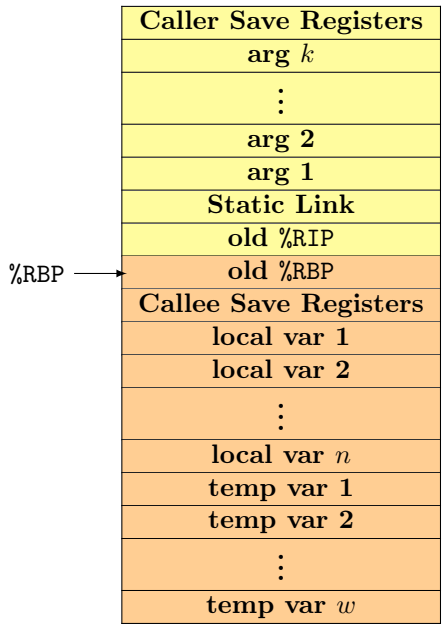


Figure 23: An illustration of the layout of our activation record on the stack. The stack grows downwards. The yellow part is created by the caller, and the orange part is created by the callee. The current base pointer is pointing at the location of old base pointer.

8.3.1 Managing Activation Records

The activation record resides on the stack as a block of memory and is jointly created by the caller and the callee, each contributing their part to the activation record. An activation record is created on the stack when a function is called and then later destroyed when the function returns. Here is an overview of what the caller does when calling a function:

1. The caller and callee have a contract between them, which ensures they do not overwrite registers which are in use by the other. Caller-saved registers are registers that the caller is responsible for saving before making a function call, which is all other registers than `RBP, RBX, R12-15`. Here the caller save registers procedure is called, pushing the caller save registers to the stack.

2. All arguments are pushed onto the stack in reverse order. The reverse order is chosen because when accessing the arguments in the stack, it is done relative to the location of the `%RBP`, so when accessing the arguments it is natural to read them in the opposite order in which they were pushed to the stack.

3. A static link pointing to the base pointer of the static parent's scope (i.e., the scope in which the current function is defined) is pushed onto the stack. This "link" is the parent's base pointer, which enables the current activation record to access the parent's activation record. These static links can be followed recursively until the desired ancestor scope is reached. Since we support nested functions and global variables, one will need access to the variables defined in ancestor scopes, hence why the static link is part of the activation records.

4. The old instruction pointer `%RIP` is saved so we can later return to the old location in the caller's function.

Now the callee has control of the stack, and then the following happens:

1. Pushing the old base pointer `%RBP` to the stack and making the new base pointer point to that location on the stack. This is done so when the function returns, the old base pointer can be restored.

2. The callee-saved registers are registers that the callee is responsible for saving before using them, which is `RBP, RBX, R12-15`. Here, the rest of the callee save registers are pushed onto the stack.

3. All local variables are known at this point, so a default value is pushed onto the stack for each variable, as described in section 8.1.1

4. The rest of the activation record is used for temporary variables. These temporary variables are used for intermediate operations. For example, `int x = (2+2)+(3+3)+(4+4)`, we need to store $2+2=4$ somewhere, `int x = 4+(3+3)+(4+4)`, before we can calculate the remaining `(3+3)+(4+4)`. Having temporary values on the stack, we ensure there is always a place to store these values.

After the callee is done, the result of the callee function is put in the register `%RAX`. The callee is then responsible for cleaning up its part of the activation record before returning control to the caller.

1. We can safely remove the temporary and local variables since they will not be used anymore, so the stack pointer is moved to the callee save registers, essentially removing the variables from the stack.
2. The callee save registers are then restored by popping them from the stack in the reverse order of how they were saved.
3. The old base pointer is restored by popping it from the stack, which moves the stack pointer to the old `%RIP`.
4. Control is then given to the caller, by restoring the old instruction pointer.
5. The static link and arguments to the function are no longer relevant, so they can be safely removed.
6. The caller save registers are restored.

Now the activation record is fully removed from the stack and the caller can safely resume execution. The caller can access the return value from the `%RAX` register.

This process of creating and removing the activation records ensures proper function calling, nesting and recursion. When a new function is called, this process is repeated, creating a new activation record on top of the existing one.

8.3.2 Static linking

Static linking is a technique used for accessing variables from outer scopes, that are visible to the current scope. Each scope is associated with a depth, which represents the nesting of the scope. When a variable is encountered that has a lower scope depth than the current one, we can apply the static linking technique. Here we can use the scope depth difference between the current scope depth, and the variable's own scope depth, to calculate how many static links should be traversed through the activation records to reach the variable's activation record.

The process of traversing static links goes as follows:

1. Start from the current activation record.
2. Follow the static link to the parent activation record.

```
int main() {
    int x = 1;
    int f() {
        int g() {
            int y = 2;
            print(x+y);
            return 0;
        }
        g();
        return 0;
    }
    f();
    return 0;
}
```

Figure 24: Example of how static linking works. In this case, when the `g()` function references the variable `x`, the compiler calculates the scope depth difference between `g()`'s scope depth (3) and `x`'s depth (1). In this case, it is 2, meaning the compiler has to follow two static links before reaching the correct activation record where `x` is defined.

3. Repeat step 2 until the desired scope is reached.
4. Access the variable from the activation record.

See Figure 24 for an example of how the static linking would work.

There is an overhead of accessing variables outside of one's own scope, since you have to follow the static link each time you want to access a variable.

8.4 Conversion of AST to IR

Whilst the AST is a high-level representation of the structure of the compiled code, the intermediary representation of the code is much lower level, and therefore there needs to be performed a translation process of the individual vertices the AST consists of. In this section, we will go over exactly how various parts of the AST are converted into an appropriate intermediary representation.

8.4.1 Classes

Classes, in the intermediate representation, are translated into two parts. A block of memory allocated for the class, which is where all the members of the class will be located, and a variable containing the pointer to this block of memory. When allocating the amount of memory that needs to be allocated for an instance of a class, it is found by allocating eight bytes of memory for each member of the class. This is done because all variables in the Giga Chad

Language are considered as eight byte variables, thus eliminating the need for padding the memory for the members. As there is no need for padding, the members can be stored in the block of memory, in the order they are declared in the class declaration. The variable that represents the instance of the class, then, simply contains the pointer to the block of memory after allocation.

As each member of a class is a variable, it contains an id, representing its order in the class declaration. This id can then be used to calculate an offset, which is then used to access the variable through indirect relative access to the variable containing the pointer to the block of memory the desired member is contained in. A member of a class, can be another class, as a variable containing an instance of a class simply contains a pointer to the block of memory allocated for the instance of a class. Accessing members of classes of classes, works like accessing a member of a class, the intermediate representation just follows multiple pointers before it finds the desired member.

8.4.2 Preliminary Intermediary Code Generation

The first intermediary code that is generated, is to set up an activation record to mimic the activation records of all other functions in our intermediate representation. What this entails can be seen in the callee generated segment of 23. This includes saving the callee save registers and making space on the activation record for the global variables. This is done to have a consistent manner of offsetting the base pointer when accessing variables in any scope. As the offset generated for accessing a variable, expects there to have been performed a callee save, which takes up a few spaces on the stack, that has to be accounted for when generating offsets. This way, any offset can be created in the same way, without worrying about how many spaces on the stack have been taken.

8.4.3 Function Calling

When a call to a function is encountered, it is translated into intermediate representation. The first thing that happens is a caller-save, after which the arguments to the function are pushed to the stack for the current scope, from an internal stack of intermediary values. These values are pushed to the stack in the opposite order they were passed to the function, to easily access them on the stack in the order they are expected to be accessed, as the first argument passed to the function will end up on the top of the stack.
The next step is to generate a static link between the current scope and the scope where the declaration of the called function is located.
After which the function is actually called, and the resulting value of the function will be found in `%RAX`, which is moved into a generic register that is then stored in an internal stack.

8.4.4 Function Declarations

When a function declaration is encountered, it is translated into intermediate code representation. First, the function's label is added, so it can be called later.

y	x	1	2	...	n

Table 4: How a 2D array is stored in memory by GCC. The numbers represent distinct elements of the array. The size of the first dimension is represented by x and the size of the second dimension is y. A value representing this array in the code would point to the element marked 1.

Then a new activation record is started, and a callee-save is performed. This is because the declaration of a function acts as the body of a function, containing all the code that needs to be executed.

8.4.5 Binary Operations

Binary operations are translated into their respective assembly instructions, and the result is kept in the intermediate storage for temporary safekeeping, so it can be used when translating code for the parent vertices of the binary operation vertex in the AST. How values that result from one of these operations that exceed the 64-bit limit of the assembly instructions we compile to, are handled, can be seen in section 14.1.

8.4.6 Arrays

Arrays, similar to instances of classes, are stored on the heap such that all elements are stored beside each other where a value representing an array in GCL is a pointer to the heap location of the first element in the array. The size of an array in GCL is set by an expression which can only be evaluated at runtime, meaning the size is not known at compile time and has to be stored alongside the array in the heap such that it can be fetched at runtime. The size of the array is stored behind the first element in the array, such that the memory looks like the example in Table 4.

For multi-dimensional arrays, there are multiple orders to store the elements in an array, in which are row major and column major. An example of 2D-array is used to illustrate the difference:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Using row-major, you would store the rows side-by-side such that the elements are stored like $1, 2, 3, 4, 5, 6, 7, 8, 9$ but with column-major, the columns would be side-by-side such that it would be stored like $1, 4, 7, 2, 5, 8, 3, 6, 9$. Picking between these does not directly change the meaning of the GCL code written, but rather affects what order an array should be looped through in GCL. You would always want to access an array in a linear order in the heap if accessing multiple elements, since this will decrease the number of cache misses in the CPU caches. So, if row-major is used, code written in GCL should index through the elements as $1, 2, \ldots, 9$ but if it is column major, it would be $1, 4, 7, 2, \ldots, 9$. In

```
for x in 0..n:
    for y in 0..m:
        arr[x,y]
```

Figure 25: Pseudocode for the optimal way to write loops in GCL.

GCL, the way you would dictate whether it is row-major or column-major is by deciding what an index into an array means, since this is the only time elements inside an array are accessed or modified. For example, does `arr[1,2]` mean that you access the element at row 1 and column 2, or vice versa?

In GCL, we opted for storing arrays as row major because this is what most other popular programming languages do, so our language will be consistent with what the programmer expects. When you look at a matrix, this is also what you would expect, since we are used to reading from left-to-right and top-to-bottom. With row-major, `arr[1,2]` means that you access the element at column 1 and row 2. Thus, when creating loops in GCL, you would prefer to have the outer loop be the first dimension and the inner loop be the second dimension, as shown on Figure 25.

When indexing into an array, we assume that it has at least one element (i.e. its size is greater than 0) since the pointer for an array points to the first element in the array. Furthermore, this is a valid assumption because if it does not have an element, the array is empty and equivalent to beta.

8.4.7 Other Implementation Details

In this section, we will discuss how several minor elements of the AST are translated into an intermediate representation.

- **If statements:** We need to handle the control flow of the `if`, `else if` and `else` statements. We can implement this in assembly by adding a label to each block of code for each of these statements. We can then check if their guard is false; if so, we jump to the next label. If the guard is true instead, it will ignore the jump and execute the code in the block. Once a block of code inside an if statement finishes executing, it jumps to a label representing the end of all the if statements to avoid also executing the next block of code.
- **Variable Expression:** Every time a variable appears in an expression, we need to check whether static linking should be performed, to retrieve the variable from another activation record. Thereafter, we simply add it to the intermediary result stack as described in Section 6.3.2, From Visit to Visit, as a `generic register` so the next vertex in the AST can use it.
- **Variable Assign:** When a variable needs to be assigned, we also need to check which scope the variable is defined in, to determine whether static

linking should be performed. It will then be assigned the value that is at the top of the stack in the intermediary storage, said value was calculated in a previous vertex in the AST.

- **Return Statement:** First, the result is put into the register `%RAX` to be saved. This puts some restrictions on how our register allocation scheme works, as discussed later in Section 10, Register Allocation. When a function returns, as all functions do in our language, the activation record needs to be cleaned up, which will be done here, as described in Section 8.3, Activation Record. This is done by removing the local variables and temporary variables, popping the callee saved registers from the stack and restoring the old base pointer and old stack pointer from the previous activation record, so control is returned to the caller.

- **While Loops:** While loops are constructed similarly to if statements, using two labels. One label is placed at the beginning of the while loop, and the other at the end of the while loop. After the start label, a comparison is performed to determine whether the expression in the guard condition is true so the code block inside the loop should be executed or if the program should jump to the end of the while loop, skipping the code block. However, every time the end of the code block is reached, the program jumps back to the start label of the while loop. This process is repeated until the guard condition becomes false.

  - **Break and Continue Statements:** To support break statements in the while loop, we simply jump to the end label of the while loop. This immediately terminates the loop and continues with the next statement after the loop. For continue statements, we jump to the start label of the while loop. This skips the remaining code in the current iteration and proceeds to the next iteration of the loop.
  - **Nested While Loops:** To support break and continue statements in nested while loops, we maintain a stack in C++ to keep track of the current loop. When entering a while loop in code generation, we push the loop onto the stack, making it the current loop. When leaving a while loop, we pop it from the stack, so the previous while loop becomes the current loop. When a break statement or continue statement is encountered, it is linked to the top while loop in the stack such that it knows which it should break out of or continue in. This stack-based approach ensures that break and continue statements always affect the innermost enclosing loop, even in the presence of nested loops.

- **Memory deallocation:** In GCL, the memory the user allocates, is never deallocated again. Neither the user has the option of deallocating the memory themselves, nor is there any form of garbage collection in place to automatically deallocate any allocated memory.

- **Print:** Printing in GCL supports different printing for all the types in GCL such that if you, for example, print an integer, it will print the number but if you print a boolean, it will print "true" or "false". This is implemented by checking the type of the expression in the print statement which was found in type checking and then calling the corresponding standard function that we have implemented for that type. Thus, if the expression is of the type array, it calls the array print function. For arrays, it prints "array" if it has a value, otherwise it prints "beta". For objects, it instead prints "object" if it has a value.

9 Liveness Analysis

In our compiler, liveness analysis is the backbone for optimizing the code. Liveness analysis determines which variables are live at each instruction, meaning that we can know for each instruction which values may be needed in the future. By identifying these variables, the compiler can make informed decisions about register allocation and peephole optimizations, which can improve the performance of the generated code.

9.1 Understanding Liveness

Liveness analysis is a fundamental concept in compiler design, utilized to optimize code by understanding the lifecycle of variables. A variable is considered *live* at a particular point in the program if its value will be used in the future. Conversely, a variable is *dead* if its value is no longer required.

9.1.1 How Liveness Analysis Works

A basic implementation of liveness analysis involves examining the intermediate representation (IR) of the program to determine the set of live variables at each instruction. The process typically follows these steps:

1. **Control Flow Graph (CFG) Construction**: A control flow graph of the program is constructed, where vertices represent instructions, and edges represent the flow of control. Each vertex will point to the other possible vertices, that comes afterwards, i.e. the next instruction that follows. The only instructions that can have multiple edges are the conditional jump statements

2. **Initialization**: Each node in the CFG is initialized with an empty set of live variables. More specifically, each vertex contains an in, out, use and def set:

$$in[n] = use[n] \cup (out[n] - def[n])$$
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

Dataflow equations for liveness analysis.

Where $in[n]$ is the set of variables that are live at the entry of a vertex n, $out[n]$ is the set of variables that are live at the exit of a vertex n, and succ$[n]$ are the successor vertices of n.

We define use and def as the following:

- $use[n]$: The set of variables used (read) in vertex n.
- $def[n]$: The set of variables defined (assigned) in vertex n.

By looking at each instruction, we can identify which variables are used and are defined, enabling us to calculate in and out.

3. **Iterative Analysis**: The algorithm iteratively traverses the CFG in reverse order, updating the sets of live variables based on data flow equations [1, p. 220] until a fixed point is reached (i.e., no further changes occur).

The liveness should only be done for each function, and not between them, as the variables' scope and lifecycle are confined within the function itself. Therefore, variables are not accessible outside unless they are statically linked or are given as function parameters since these reside on the stack. By understanding which variables are live at each point in a program, the compiler can make more efficient decisions, however, the process of liveness analysis, especially in large and complex control flow graphs, can be computationally intensive. The iterative approach to reaching a fixed point in large CFGs can significantly impact the compile time.

10 Register Allocation

During the code generation phase, transforming the AST into our intermediate representation (IR) involves strategic register allocation to enhance program performance. We prioritize the use of generic registers over specific registers, allowing for flexible and efficient mapping to concrete registers later. This approach delegates the register allocation problem from code generation to another phase, which we call register allocation. We have solved this problem in two ways, with a naive and a smart register allocation algorithm.

10.1 Naive Register Allocation

This approach is fully stack-based, meaning that all generic registers are stored on the stack and are only mapped to concrete registers for a short time when they occur in an instruction. When an instruction is found that uses a generic register, the following is done:

1. A new instruction is added before the original instruction that moves the value from the generic register's stack position to an available concrete register.
2. The generic register in the original instruction is replaced by the concrete register.
3. A new instruction is added after the original instruction that moves the value in the concrete register to the generic register's stack position.

The process above can be repeated multiple times if an instruction uses multiple generic registers. It is guaranteed that there are available concrete registers to use since an instruction at a maximum uses three registers and the registers become available for use again after the new instructions above have been executed. The frequent loading and storing to the stack gives a performance overhead due to the increased memory operations. Despite this, the naive register allocation algorithm can be desired when low compilation time is desired, as its simplicity makes it fast.

10.2 Smart Register Allocation

Our smart register allocation is leveraging the colouring by simplification algorithm. It is an approximation algorithm of the graph colouring problem, which is known to be NP-complete when it has three or more vertices. It actually turns out that we can make a polynomial time reduction from the graph colouring problem to the register allocation problem by constructing it as a graph, that is:

$$\text{Graph colouring} \leq_p \text{Register allocation}$$

We call this graph the interference graph, as it tells us which variables are alive at the same time, which is denoted by the edges.

10.2.1 Interference Graph Construction

We can construct an interference graph for each function by doing liveness analysis on our code. It tells us which variables are live at the same time. We add a vertex to the graph for every generic register and concrete register, and we add an edge if two registers are live at the same time. We can now run the graph colouring algorithm on it. However, this is not necessarily solvable, since an arbitrary number of registers could be live at the same time, meaning it could be more than we have available, and the algorithm would fail. Hence, we use the colouring by simplification algorithm to work around this problem.

10.2.2 Colouring by simplification

Colouring by simplification[1, p. 236] is an approximation of the graph colouring problem and thus, in the context of register allocation, attempts to find the smallest number of concrete registers required to simulate the generic registers

used in the code, i.e., with the smallest number of concrete registers, it attempts to satisfy the constraints of liveness and thus the constraints in the interference graph. Since this is not always possible considering there is a finite number of registers that can be used, it sometimes needs to assign some generic registers to be on the stack.

Let K be the number of concrete registers, i.e., colours, the algorithm has available and the *select stack* be a stack of generic registers that should be mapped to concrete registers. Lastly, the *spill stack* is the stack of generic registers that could not be given a colour and should thus be put on the stack. Using K and the two stacks, the colouring by simplification works as follows:

1. **Build**: Construct the interference graph using liveness analysis.

2. **Simplify**: Next we try to colour the graph, where the number of colours we use corresponds to the available registers. We use a simple heuristic where we look for all vertices with fewer neighbours than the available registers. When we find such a vertex, we can remove it temporarily and push it onto the select stack since we know it can be colourable, so we want to focus on the rest. The idea is that removing the vertices simplifies the graph, making it easier to colour the remaining vertices. Every time we remove such a vertex, the number of edges for the remaining vertices can potentially decrease, making it possible for further simplifications.

3. **Spill**: If at some point the simplification fails, i.e., there are some vertices that cannot be simplified (all remaining nodes have K or more neighbours), a node is selected for spilling. This node is then removed and pushed onto the spill stack with the assumption it will be placed in memory rather than a register. This reduces the graph and allows simplification to continue.

4. **Select**: In this phase, we assign colours, representing registers, to the vertices reintroduced from the stack. We start with an empty graph and add back each vertex, ensuring it can be coloured based on our earlier simplifications. If all potential registers are occupied by its neighbours, the vertex cannot be coloured and is marked for potential spilling. This process continues until all vertices are either assigned a register or confirmed as spills.

5. **Start Over**: If any vertex remains uncolourable after the **Select** step, we modify the program by managing these variables in memory. These variables are then represented as new temporaries with shorter lifespans, intended to reduce interference in the interference graph. We reapply the colouring algorithm starting from the **Build** step, iterating until all vertices can be coloured successfully without spills.

When a generic register has been assigned a concrete register, i.e., a colour, by the algorithm, it is as simple as replacing that generic register with the concrete register. When a generic register is assigned to be on the stack, i.e., it "spilt" in

the algorithm, it is required that it is fetched into a concrete register before using it in an instruction and put the value back to the stack when an instruction sets its value, very similar to the way the naive register allocation does it. There is a difference, however, since we here have access to the liveness analysis of the instruction, so we only fetch it from the stack if the instruction uses it and only store it in the stack again if the instruction defines it.

10.2.3 Advantages of Colouring by Simplification

This approach offers several advantages:

- **Efficiency**: By minimizing stack usage and effectively utilizing available registers, this method reduces the performance overhead associated with memory operations.
- **Flexibility**: The use of generic registers during code generation allows for a flexible and efficient mapping to concrete registers, deferring complex decisions to the register allocation phase.
- **Optimized Performance**: This method produces more optimized code, as it minimizes the number of memory accesses required, leveraging the faster access times of registers.

While the smart register allocation is more complex than the naive allocation, the colouring by simplification approach provides a significant performance boost, especially in applications where execution speed is critical. By strategically managing register usage, it ensures that the compiled code runs efficiently on the target hardware.

11 Peephole Optimization

Peephole optimization is an optimization technique, that involves looking at smaller snippets of the instructions in the intermediate code representation, often called peepholes, and then through these peepholes, looking for optimizations to make by selecting small peepholes of instructions and replacing them with logically equivalent, but more performant instructions [10].

Peephole optimization addresses common patterns of inefficiency, such as redundant instructions, unnecessary loads and stores, and suboptimal use of registers. it is run both before and after each modification of the intermediate code. This two-phase application ensures that any new inefficient instructions introduced during another phase are promptly identified and optimized. By repeatedly refining the intermediate code we obtain more efficient code.

In the GCC compiler, this technique is applied both to the instructions generated by the code generation phase, and also to the instructions produced by the register allocation phase. This means that whilst peephole optimization is its own phase like all the other phases in the GCC compiler, unlike the other phases, peephole optimization is a phase that is entered multiple times. Since it needs to be run each time, the intermediate code is modified.

```
...
call L0_f
popq %rcx
popq %rax
push %rax
push %rcx
call L0_f
...
```

(a) Before modification

```
...
call L0_f
call L0_f
...
```

(b) After modification

Figure 26: Code examples before and after applying the POP PUSH pattern. These are merely code snippets, not a full program. They show how the assembly code generated by the GCC for calling two functions in a row is optimized by applying pattern 2 from Table 5. In this example, it is assumed that the two registers in question, `%rax` and `%rcx` are not used later in the code before being defined

11.1 Patterns

The patterns for peephole optimization are defined by, a pattern of intermediate code instructions, some rules that the pattern must comply with, and a replacement. The replacement is either an equivalent set of instructions or nothing, which will be applied if the pattern complies with the rules. Peephole is a "local" optimization, meaning that it optimizes instructions within each scope. The patterns defined in GCC, and their corresponding rules, i.e., the conditions for applying the replacement and replacements, can be seen in Table 5.

Peephole optimization uses the information gathered from liveness analysis to determine whether some rules should be applied, e.g., pattern 4 in Table 5 will remove an instruction of the intermediate code if the instruction defines a register, but it is not used in later instructions. That is, the pattern essentially scans the intermediate code for "dead code" and removes these instructions. Other patterns do not rely on the information from liveness analysis, e.g., pattern 5 uses the argument of the instruction to determine whether it should apply the replacement.

As an example of how applying a pattern affects the final assembly code, see Figure 26, how applying pattern number 2, the POP and PUSH elimination pattern, eliminates several redundant POP and PUSH instructions from the code. This optimization not only reduces the size of the code by removing extra instructions but also improves the runtime efficiency of the generated assembly code.

	Pattern	Condition	Replacement

1	`MOV A, B`
	`WILDCARD C, D`
		`(MOV A,B).def` $\in$ `(WILDCARD C,D).use`
		`(MOV A,B).def` $\notin$ `(WILDCARD C,D).def`
		`(WILDCARD C,D).use` $\notin$ `(MOV A,B).use`
			`MOV A,B`
			`WILDCARD A,D` or
			`MOV A,B`
			`WILDCARD C,A`

2	`PUSH A` `POP A`		$\emptyset$
3	`POP A` `PUSH A`	`A` $\notin$ `PUSH.out`	$\emptyset$
4	`WILDCARD`	$\forall$ `A` $\in$ `WILDCARD.def` : `A` $\notin$ `WILDCARD.out`	$\emptyset$
5	`JMP A` `LABEL(A)`		`LABEL(A)`
6	`MOV A, A`		$\emptyset$

Table 5: Peephole optimization patterns implemented in GCC. Each pattern represents a sequence of x86-64 assembly instructions identified in the intermediate code. If the pattern is found and its condition is met, the corresponding replacement is applied, which may reduce the number of instructions or eliminate redundant code. The `WILDCARD` pattern is used to identify unused instructions as it matches all possible instructions. The $\emptyset$ symbol indicates that the matched instructions should be removed.

11.2 Applying Peephole Optimization Patterns

To apply peephole optimization, liveness analysis is performed for each scope. Each defined peephole pattern (as seen in Table 5) is tested against all instructions in the intermediate code representation of that scope. The rules of these patterns are evaluated using the information gathered from liveness analysis or the arguments of the instructions. If any changes to the intermediate code have been made by peephole, liveness analysis is run again before continuing with peephole, as the relationship between the instructions might have changed. Peephole is run on the instructions of the current scope until the state of the instructions is stable, i.e., no more replacements could be applied. This is necessary as peephole patterns modify the intermediate code and therefore might introduce a new pattern, e.g., would pattern 1 result in some dead code namely "`MOV A, B`", which pattern 4 can handle.

Peephole will look at the intermediate code of a given function and call liveness analysis to receive liveness information about said function. This liveness information appears as a collection of data, where each entry corresponds to an instruction in IR of the function currently being peephole optimized. This is important as peephole will search for patterns in the liveness information, and use the indexes of the matching patterns from the liveness information, to find the corresponding instructions to apply the replacement of the pattern to. Since a replacement will never result in adding more instructions than it is replacing, we

are guaranteed that modifying the intermediate code starting from the index of the match in the liveness information will never replace instructions, not part of the matched pattern. However, one problem is that modifying the length of the intermediate code list will break the duality between the liveness information and the IR, for that reason, we pad the intermediate code with "dummy" instructions, such that we can continue applying the patterns through all instructions and once all patterns have been searched for, in the liveness information for the current function being examined, the intermediate representation will be pruned for "dummy" values before running liveness analysis again. This approach allows us to get away with only calling liveness analysis once per pass over the function being optimised. Every function is examined at least once for potential optimisations, and any functions that are found to have instructions that match a peephole pattern, will be examined at least twice. This is done to ensure that applying a pattern, will not create instructions that result in matching a new pattern, or the same pattern that created the instructions.

12 Emit

The responsibility of this phase is taking each instruction generated during the code generation phase, and converting it into AT&T x86-64 syntax assembly instructions, then writing it to a file.

This conversion is almost implicit, as our immediate representation is designed to be very close to the assembly syntax we compile down to. Each instruction is converted into a string, which can be done easily since each intermediate representation instruction already contains all the information that would be needed to write its corresponding assembly instruction. This information is also stored in a manner, that is reminiscent of how the information would be used in its corresponding assembly instruction.

As an example, see below how a move instruction is represented in our immediate representation:

```
Instruction(Op::MOVQ, Arg(Register::RSP, DIR()),
                      Arg(Register::RBP, DIR()));
```

in comparison to its corresponding assembly instruction:

```
movq %rsp, %rbp
```

13 Benchmark

To evaluate the performance improvements of our peephole optimizations and smart register allocation, we conducted benchmarks on three different programs: Fibonacci, matrix-matrix multiplication, and merge sort. All implemented in GCL, the code can be found in Appendix A, Appendix B and Appendix C respectively.

The benchmarks plot the input size against each program's run time. The time is excluding compilation time of GCC, so solely the run time of the emitted assembly code of GCC. Each time stamp was registered using the Python library "os" [12] and "time" [13] to execute shell commands and time them:

```
import time
import os
...
now_time = time.time()
os.system(run_command)
end_time = time.time()
...
```

The test programs were chosen because they represent well-known computational problems, each with distinct traits that make them ideal for evaluating the effectiveness of our optimizations. Fibonacci's recursive nature results in numerous function calls, making it an excellent test for evaluating how our optimizations handle recursion and function call overhead. Merge Sort involves significant data manipulation and memory access patterns. It tests the optimization's ability to handle sorting operations. Matrix-Matrix multiplication is computationally intensive, as it requires numerous arithmetic operations. This test clearly shows the optimization's effectiveness in improving the performance of the program. This diversity ensures a comprehensive evaluation of our optimizations across different types of workloads, highlighting their impact on various aspects of program performance.

The benchmarks were performed on the same PC with the following specs: Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz with 16 GB RAM.

The primary goal was to compare the execution times of these programs using different combinations of register allocation strategies and optimization techniques. Specifically, we tested "Naive" (naive register allocation) versus "Smart" (smart register allocation) approaches, and the impact of enabling and disabling "Peephole" optimizations. Each test shown was run three times and the average run time is plotted. This comprehensive analysis helps in understanding the trade-offs and benefits of each optimization.

13.1 Fibonacci

The Fibonacci program written in GCL was benchmarked on the input sizes 43, 44, 45 and 46 since each increase in input size will essentially double the execution time. The program was run with all different combinations of register allocation strategies and peephole, to clearly see the impact of these optimizations. The benchmarks can be seen in Figure 27.

As the input size increases from 43 to 46, the run time for all tests increases, as expected. Noticeably, "Smart" consistently shows the lowest run time across all input sizes, demonstrating the effectiveness of the smart register allocation. Both "Smart + No Peephole" and "Smart + Peephole" perform the same, indicating that peephole optimization has no tangible effect on the code when used

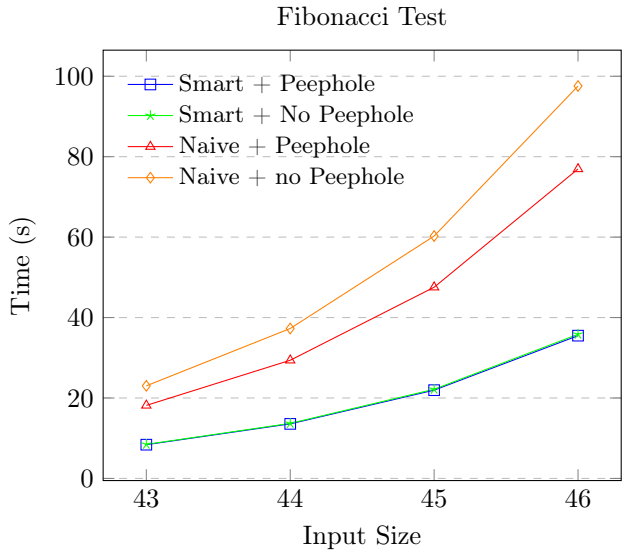


Figure 27: Benchmarks for performance tests for Fibonacci implementation. For the GCL code, see Appendix A. The benchmarks are conducted using four different input sizes for each test, which is represented by their own symbols on the X-axis. Each test at each input size is run three times, and the average time is plotted. The chart includes all combinations of optimizations, i.e., "Naive" (naive register allocation), "Smart" (smart register allocation), "Peephole" (peephole enabled), and "No Peephole" (peephole disabled), each plotted with unique colours and symbols. The X-axis represents the input size, while the Y-axis shows the time in seconds.

in combination with smart register allocation. "Naive + Peephole" shows moderate performance, better than "Naive + No Peephole" but not as good as when using smart register allocation, highlighting that while peephole optimization improves performance, it does not do much when combined with smart register allocation in this case.

This observation clearly indicates that we might be missing some patterns for peephole optimization, which is anticipated given our limited number of defined patterns. For future work, analysing the emitted assembly code from our Fibonacci test would provide insight into what types of patterns, could be added to the already defined set of patterns. This analysis could help us reduce the run time of this test by leveraging peephole optimization more effectively.

"Naive + No Peephole" consistently shows the highest execution times, illustrating the lack of performance enhancements without optimizations. It is clear that smart register allocation is a significant improvement compared to the naive approach, but peephole optimisation currently does not improve much on top of smart register allocation in this case. For smaller input sizes (43 and 44), the difference in execution time between the optimization strategies is less pronounced. For larger input sizes (45 and 46), the differences become more significant, especially highlighting the superiority of the smart optimizations over the naive ones.

13.2 Matrix-Matrix Multiplication

The Matrix-matrix Multiplication program written in GCL was benchmarked on the input sizes 400, 600, 800, and 1000, to have a constant span between the input sizes while keeping them small enough to keep the benchmark time reasonable. The program was run with all different combinations of register allocation strategies and peephole. The benchmarks can be seen on Figure 28.

As the input size increases from 400 to 1000, the execution time for all tests increases, which is expected given the computational intensity of matrix-matrix multiplication. Notably, peephole has little impact on the run across all input sizes. As each test is only run three times, and the average of these are plotted, it is expected to see some fluctuation in the graph. However, looking at the assembly code reveals that peephole affects the instructions in the matrix-matrix multiplication test, but not enough to show a significant reduction in run time.

For smaller input sizes (400 and 600), the difference in run time between the optimization strategies is less pronounced. For larger input sizes (800 and 1000), the differences become more significant, especially highlighting the superiority of the smart register allocation over the naive ones, as the run time is significantly reduced. The performance benefits of using smart register allocations are apparent from the graph mentioned initially.

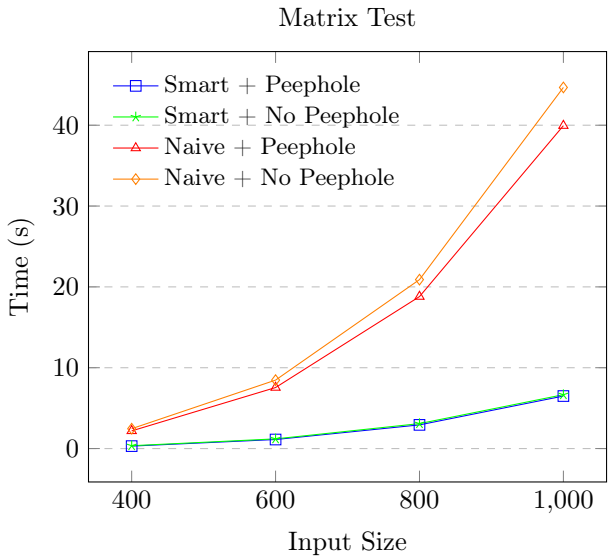


Figure 28: Benchmarks for performance tests for Matrix-Matrix multiplication implementation in GCL. For the GCL code, see Appendix B. The benchmarks are conducted using four different input sizes for each test, which is represented by their own symbols on the X-axis. Each test at each input size is run three times, and the average time is plotted. The chart includes all combinations of optimizations, i.e., "Naive" (naive register allocation), "Smart" (smart register allocation), "Peephole" (peephole enabled), and "No Peephole" (peephole disabled), each plotted with unique colours and symbols. The X-axis represents the input size, while the Y-axis shows the time in seconds.

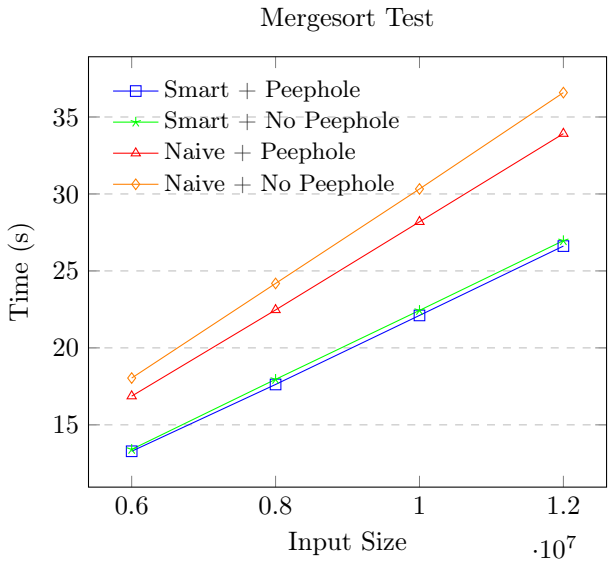


Figure 29: Benchmarks for performance tests for Merge Sort implementation in GCL. For the GCL code, see Appendix C. The benchmarks are conducted using four different input sizes for each test, which is represented by their own symbols on the X-axis. Each test at each input size is run three times, and the average time is plotted. The chart includes all combinations of optimizations, i.e., "Naive" (naive register allocation), "Smart" (smart register allocation), "Peephole" (peephole enabled), and "No Peephole" (peephole disabled), each plotted with unique colours and symbols. The X-axis represents the input size, while the Y-axis shows the time in seconds.

13.3 Merge Sort

The Merge Sort program written in GCL was benchmarked on input sizes of $0.6 \cdot 10^7$, $0.8 \cdot 10^7$, $1 \cdot 10^7$ and $1.2 \cdot 10^7$. This range was chosen to provide a significant run time, to see the impact of the optimizations. The benchmarks can be seen in Figure 29.

As the input size increases from $0.6 \cdot 10^7$ to $0.8 \cdot 10^7$, the run time for all tests increases. The graph demonstrates that both smart register allocation and peephole optimizations have a significant impact on the run time of Merge Sort.

"Smart + Peephole" consistently shows the lowest run time across all input sizes, demonstrating the combined effectiveness of both smart register allocation and peephole optimizations in improving performance. "Smart + No Peephole" also performs well but slightly worse than "Smart + Peephole", indicating the additional benefit of peephole optimization. This is also clear from looking at "Naive + Peephole" and "Naive + No Peephole" where Peephole significantly reduces the run time.

This analysis indicates that both optimizations contribute to reducing the run time of the Merge Sort program. For smaller input sizes ($0.6 \cdot 10^7$ and $0.8 \cdot 10^7$), the difference in run time between the optimization strategies is less pronounced. However, for larger input sizes ($1 \cdot 10^7$ and $1.2 \cdot 10^7$, the differences become more significant, especially highlighting the superiority of the smart register allocation over the naive ones, as the run time is significantly reduced. The performance benefits of using smart register allocations along with peephole optimization are clearly apparent from the graph initially mentioned.

14 Runtime Error Handling

Some errors can only be detected at runtime but checking for errors at runtime will always have a running time cost of the compiled code. If such an error is not checked and the error case is reached, it often results in undefined behaviour and thus will in the best-case result in a segmentation fault which does not provide the user with any information about why the error happened, making debugging of the GCL code more difficult. Thus, runtime error handling introduces an interesting tradeoff between the running time of the compiled program and giving sufficient error messages to the user of the language. In this section, we will describe the considerations we have had on runtime error handling.

14.1 64-bit Overflow

When it comes to values that exceed the range representable by a 64-bit integer, we do not actually do anything to handle this overflow. What this means is that what happens to cases of integer overflow depends on what the architecture, we compile down to, does. Since we compile to AT&T assembly x86-64, any integer overflow will be truncated, meaning the 64 least significant bits will be kept and stored as the result of the operation that exceeded the 64-bit limitation, whilst any bits beyond the 64 least significant, will be discarded. This way of handling

```
# Code above that defines %A
CMPQ $0, %A    # Compare if beta
JNE B          # If not beta, jump
MOVQ $C, %RDI # The line number
CALL print_is_beta # Print beta and quit
B:
# Code after that uses %A
```

Figure 30: The assembly instructions for checking whether the value in register `%A` is beta. In this example, `B` is the label that marks the start position of the code that uses `%A`. Thus, this label is jumped to if the value in `%A` is not beta. If it, however, is beta, the `print_is_beta` function is called using the line number in GCL where the access of `%A` was attempted. The line number is a value known at compile time and is represented by `$C` in the figure.

integer overflows can lead to some potentially unexpected behaviour, but it has particularly dangerous implications for negative numbers, as when a negative number is truncated, the result can be a positive number. Currently, we do not have a specified way of handling integer overflows but in the future we should, even if we end up simply agreeing with the way AT&T assembly x86-64 handles integer overflows.

14.2 Beta Check

Checking whether a variable is beta is only applicable to objects and arrays since these are pointers to memory so they are followed when accessing members in them, such that if they have their default value of 0, it will give a segmentation fault. Other types, i.e., integers and booleans, always have a value and thus don't require such a check. For arrays, the beta check is performed before indexing into an array such that before indexing into it, it checks whether the array pointer is 0. If it is 0, the compiled code prints "Attempted to access a beta value on line " followed by the line number that the index occurred on in the GCL code. Writing the line number is important to allow the user to debug their GCL code. The code is given on Figure 30. The beta check has not been implemented for objects because it is so similar to doing it for arrays.

15 Testing

To verify the validity of our compiler and the various phases comprising it, we have written several functionalities to test each phase of the compiler, and the final compiler itself. It is important to always have a version of the software that works so that you can develop new features on it without being distracted by discovering and having to resolve issues with the other features. Furthermore,

tests increase your confidence in the program, which enables quicker development since you do not have to manually try the program with different inputs but instead can trust that the tests cover all the necessary cases. Each test verifies that a single feature works as expected, so it is easy to identify exactly what broke if new code was added to the compiler. Moreover, each test is associated with the phase that the feature it tests is a part of so that the feature is aware whether to accept a failure or not. For example, if a symbol collection test is expecting an error during the symbol collection phase, but runs into an error during the parsing phase, it will not succeed. Additionally, each test is executed in isolation such that the execution of one test doesn't affect the others. As we are already using Boost for large parts of the compiler itself, we decided to use the test suite from Boost called Boost.Test [14]. This allows us to use the `BOOST_AUTO_TEST_CASE` macro, which automates large parts of the testing for us, we only need to provide the macro with a function that runs our code and then Boost.Test runs the test, and generates a test report and log detailing which tests succeed, fail, and why the failing tests fail. In the CMake file, we have created the custom target "tests" that compiles all the tests, which allows us to use `ctest` to execute all the tests at once. The tests can easily be run in parallel using `ctest`, which can be utilised since the tests have been made to run in isolation. The implementation of the tests is often rather straightforward as you simply attempt to run the compiler on the GCL code in the test, but for some of the phases, extra considerations and care had to be made to create good tests. The testing of these phases will be described in the following subsections.

Additionally, since we are developing on different machines and operating systems, we added a Dockerfile to our project. This Dockerfile sets up a consistent development and testing environment using an Ubuntu operating system, the same system used by GitHub Actions. The Dockerfile runs an interactive shell, allowing the user to mount the local project directory within the interactive shell and execute tests on the Ubuntu image. This setup ensures that we can all locally test on the same operating system, regardless of individual setups.

15.1 Testing Parser

To test the parser phase, we have created test cases for almost every possible rule used in the parser phase, to ensure that all rules succeed and fail when we expect them to.

When we test rules from the parser phase, we perform two parses on the same input. That is, after performing the first parse, we save the resulting AST, translate it into its textual representation and perform the second parse on that. The reason for doing this is to determine whether the parser is stable, that is, it does not modify the input in any way. If it did, this would become apparent during the second parse, and the output of the first and second parse would not match.

15.2 Testing Assembly

To verify that the final phases of the compiler, code generation, register allocation, and emit, all work as intended, we have written several tests that each contain code written in GCL. These tests aim to have the compiler compile them down to assembly, and then run these assembly files to see if they give the expected output. The "output", of these GCL files is the values that they print. This means that to test the assembly code we generate, the assembly code we generate for print statements needs to work. This output is then captured and compared against the expected output, the result of this comparison is then the overall result of the test.

15.3 Testing Main

We have created several flags for the compiler, that allow us to perform a partial compilation of a file, and then decide how far the compiler gets before it stops compiling. We have then created a testing file and several test cases to ascertain that all of these flags cause the compiler to behave as expected.

15.4 Automated Testing

Tests are only useful if you make sure to execute them. Automated testing is the process of automatically executing the tests without human intervention when it makes sense, such that you cannot forget to execute them. Since GitHub is used to store the code for GCC, it was convenient to use GitHub Actions for automated testing for our compiler since it integrates well with GitHub. It has been set up such that when a pull request for a new feature is created, the tests are executed and the new feature is not added without passing the tests, providing the security that the new feature does not break already established features, assuming the feature is tested sufficiently.

16 Future Work

Multiple features and optimizations did not make it into the first iteration of GCC. Looking at the grammar (Figure 1), we decided to use the term 'class' instead of 'struct'. This decision was driven by our goal to make GCC fully object-oriented, and therefore the vision was not to have structs but instead fully functional classes. But as other features and improvements took precedence, this goal was never realized, and therefore would be interesting to implement in the future.

As seen in Section 13, Benchmark, peephole did play a major role in optimizing the run time of the programs written in GCL. Therefore, in the future, analysing the emitted assembly code to discover even more peephole patterns would aid in significantly improving the overall runtime of programs written in GCL.

In Section 2.1, Grammar, we mentioned operator overloading as a feature we plan to support in the future iterations of GCL. It would be interesting to explore the full potential of operator overloading, as this will involve designing a robust mechanism for defining and resolving overloaded operators, ensuring type safety and maintaining the overall performance of the compiler.

The compile time of GCC is slow as our implementation of liveness analysis is computationally heavy and our implementations of register allocation and peephole optimization utilize liveness analysis. Tweaking and optimizing our implementations of these features to decrease the overall compile time of GCC would be interesting, as it involves improving the efficiency of liveness analysis and exploring more efficient algorithms for register allocation and peephole optimization.

17 Conclusion

In this bachelor thesis, we have presented the design and implementation of the Giga Chad Compiler (GCC) for the Giga Chad Language (GCL). Our goal was to gain hands-on experience and a deeper understanding of the compilation process by developing a compiler that translates GCL code into x86-64 assembly code in AT&T syntax. We have, throughout this project, implemented various phases of the compilations process following the approach described in [1].

We combined lexical and syntax analysis into a single parsing phase using Parsing Expression Grammars (PEG) and the Boost.Spirit X3 library. This approach allowed us to efficiently generate an Abstract Syntax Tree (AST) from the GCL code. The semantic analysis phase, consisting of symbol collection and type checking, ensured for proper scoping, identifier resolution and type safety. We deployed a producer-consumer scheme to propagate type information through the AST, enabling type checking and error detection. In the code generation phase, we translated the AST into an intermediate representation (IR) that closely resembled the assembly code. We utilized the activation record scheme to order the stack, making us able to use static linking and support deeply nested code structures.

To optimize the generated code, we applied liveness analysis and peephole optimization techniques to eliminate redundant instructions and improve overall performance. Additionally, we implemented smart register allocation based on graph colouring to ensure efficient utilization of CPU registers. Performance benchmarks, such as matrix-matrix multiplication and merge sort, demonstrated the effectiveness of our optimizations. The results showed significant improvements in execution time and overall efficiency when applying smart register allocation and peephole optimizations.

In conclusion, the development of GCC has given valuable insights into the inner workings of compilers. By implementing each phase, we gained a deeper understanding of semantic analysis, code generation and optimization techniques.

```
int fibonacci(int n) {
    if n <= 1 {
        return n;
    } else {
        return fibonacci(n-1)+fibonacci(n-2);
    }
}

int main() {
    fibonacci(41);
    return 0;
}
```

Figure 31: The Fibonacci implementation in GCL for input size 41.

Appendices

A Fibonacci GCL Code

The implementation of Fibonacci in GCL can be seen on Figure 31.

B Matrix-Matrix Multiplication GCL Code

The implementation of matrix-matrix multiplication in GCL can be seen on Figure 32 and Figure 33.

C Merge Sort GCL Code

The implementation of merge sort in GCL can be seen on Figure 34 and Figure 35.

```
int matrixMultiplication(int[2] A, int A_rows, int
    ↪ A_cols, int[2] B, int B_rows, int B_cols) {
    if (A_cols != B_rows) {
        print(false);
        return 1;
    }

    int[2] C = new int[A_rows, B_cols];

    int i = 0;
    int j = 0;
    int k = 0;

    while i < A_rows {
        j = 0;
        while j < B_cols {
            C[i, j] = 0;
            k = 0;
            while k < A_cols {
                C[i, j] = C[i, j] + A[i, k] * B[k, j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }

    return 0;
}

```

Figure 32: The matrix-matrix implementation in GCL for input size 400. The code is continued on Figure 33.

```
int main() {
    int row = 400;
    int col = 400;
    int[2] A = new int[row, col];
    int i = 0;
    int j = 0;

    while i < row {
        j = 0;
        while j < col {
            A[i, j] = i % 10 + 1;
            j = j + 1;
        }
        i = i + 1;
    }

    int[2] B = new int[col, row];
    i = 0;
    while i < col {
        j = 0;
        while j < row {
            B[i, j] = i % 10 + 1;
            j = j + 1;
        }
        i = i + 1;
    }

    matrixMultiplication(A, row, col, B, col, row);

    return 0;
}
```

Figure 33: Part two of the matrix-matrix implementation in GCL for input size 400. The first part is on Figure 32.

```
int merge(int[1] arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int[1] L = new int[n1];
    int[1] R = new int[n2];
    int i = 0;
    while (i < n1) {
        L[i] = arr[l + i];
        i = i + 1;
    }
    int j = 0;
    while (j < n2) {
        R[j] = arr[m + 1 + j];
        j = j + 1;
    }
    i = 0;
    j = 0;
    int k = l;
    while (i < n1 & j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i = i + 1;
        } else {
            arr[k] = R[j];
            j = j + 1;
        }
        k = k + 1;
    }
    while (i < n1) {
        arr[k] = L[i];
        i = i + 1;
        k = k + 1;
    }
    while (j < n2) {
        arr[k] = R[j];
        j = j + 1;
        k = k + 1;
    }
    return 0;
}
```

Figure 34: The merge sort implementation in GCL for input size 4 million. The code is continued on Figure 35.

```
int mergeSortHelper(int[1] arr, int l, int r) {
    while (l < r) {
        int m = l + (r - l) / 2;

        mergeSortHelper(arr, l, m);
        mergeSortHelper(arr, m + 1, r);

        merge(arr, l, m, r);

        return 0;
    }

    return 0;
}

int mergeSort(int[1] arr, int n) {
    return mergeSortHelper(arr, 0, n - 1);
}

int main() {
    int arr_size = 4000000;
    int[1] arr = new int[arr_size];

   int j = 0;
    while (j < arr_size) {
      arr[j] = arr_size - j;
        j = j + 1;
    }
    mergeSort(arr, arr_size);
    return 0;
}
```

Figure 35: The second part of the merge sort implementation in GCL for input size 4 million. The first part of the code is on Figure 34.

Bibliography

[1] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2004. ISBN: 0521607655.

[2] Peter Belcak. "The LL (finite) strategy for optimal LL (k) parsing". In: *arXiv preprint arXiv:2010.07874* (2020).

[3] William H Burge. "Recursive programming techniques". In: (1975).

[4] Bryan Ford. "Parsing expression grammars: a recognition-based syntactic foundation". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2004, pp. 111–122.

[5] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[6] Dan Marsden Joel de Guzman and Tobias Schwinger. *Boost Fusion*. URL: `https://www.boost.org/doc/libs/1_79_0/libs/fusion/doc/html/index.html`. (accessed: 31-05-2024).

[7] Joel de Guzman and Hartmut Kaiser. *Boost Spirit X3*. URL: `https://www.boost.org/doc/libs/1_72_0/libs/spirit/doc/x3/html/`. (accessed: 31-05-2024).

[8] Lillian Lee. "Fast context-free grammar parsing requires fast boolean matrix multiplication". In: *Journal of the ACM (JACM)* 49.1 (2002), pp. 1–15.

[9] Kiminori Matsuzaki and Kento Emoto. "Implementing fusion-equipped parallel skeletons by expression templates". In: *Implementation and Application of Functional Languages: 21st International Symposium, IFL 2009, South Orange, NJ, USA, September 23-25, 2009, Revised Selected Papers 21*. Springer. 2010, pp. 72–89.

[10] Steven Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[11] Vladimir Prus. *Boost Program Options*. URL: `https://www.boost.org/doc/libs/1_85_0/doc/html/program_options.html`. (accessed: 31-05-2024).

[12] Python Software Foundation. *os - Miscellaneous operating system interfaces*. Python documentation. URL: `https://docs.python.org/3/library/os.html`.

[13] Python Software Foundation. *time - Time access and conversions*. Python documentation. URL: `https://docs.python.org/3/library/time.html`. (accessed: 31-05-2024).

[14] Gennadiy Rozental. *Boost Test*. URL: `https://www.boost.org/doc/libs/1_85_0/libs/test/doc/html/index.html`. (accessed: 31/05).

[15] Amir Shpilka. "Lower bounds for matrix product". In: *SIAM Journal on Computing* 32.5 (2003), pp. 1185–1200.

[16] Des Watson. *A practical approach to compiler construction*. Vol. 254. Springer, 2017.