Constructing a Modern Programming Language for Learning Low-Level Programming

Bachelor's thesis

Samuel Bas Forsberg
Aisha Cabdulahi Mohamed
Daniel Cole
Emma Dahlqvist
Lucas dos Guaranys Olsen
Felix Tan

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026
www.chalmers.se | www.gu.se

Bachelor's thesis 2026

Constructing a Modern Programming Language for Learning Low-Level Programming

Bachelor's thesis

Samuel Bas Forsberg Aisha Cabdulahi Mohamed Daniel Cole
Emma Dahlqvist Lucas dos Guaranys Olsen Felix Tan

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Computer Science and Engineering
Computing Science
Group 17A
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden 2026

Constructing a Modern Programming Language for Learning Low-Level Programming

Samuel Bas Forsberg	Aisha Cabdulahi Mohamed	Daniel Cole
Emma Dahlqvist	Lucas dos Guaranys Olsen	Felix Tan

Supervisor: Magnus Myreen, Computer Science and Engineering
Co-Examiner: Aarne Ranta, Computer Science and Engineering
Examiner: Patrik Jansson, Computer Science and Engineering

Bachelor's Thesis 2026
Department of Computer Science and Engineering
Division of Computing Science
Group 17A
Chalmers University of Technology | University of Gothenburg
SE-412 96 Gothenburg

Cover: Made by Daniel Cole

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2026

Constructing a Modern Programming Language for Learning Low-Level Programming

Bachelor's thesis

Samuel Bas Forsberg, Aisha Cabdulahi Mohamed, Daniel Cole, Emma Dahlqvist, Lucas dos Guaranys Olsen, Felix Tan

Department of Computer Science and Engineering
Chalmers University of Technology | University of Gothenburg

Abstract

While modern high-level programming languages can vastly improve developer productivity and efficiency, they obscure the underlying machine code. For aspiring students or simply anyone who is interested in the field, understanding the output of compilers can deepen one's knowledge of how computer systems operate on a low level, and offer a new perspective on programming as a whole. Assembly is the human-readable representation of machine code, making it a practical entry point for anyone seeking to understand what happens at the hardware level. However, assembly is often challenging to learn, with architecture-specific commands, and logic that is fundamentally detached from the high-level syntax that most modern developers rely on. To address these challenges, this project presents `Fika`, a statically typed language, which focuses on showing the relationship between assembly and high-level programs. Writing code in Fika is meant to be simple and intuitive, which allows the user to focus their cognitive resources exclusively on mastering fundamental assembly concepts. The language is complemented by tooling, including a language server and a Visual Studio Code extension, which provides on-demand visualisation of the generated assembly code corresponding to the code written in Fika. A user study was conducted with participants ranging from novice to experienced programmers. The results indicate that Fika provides improved accessibility to assembly concepts and is perceived as a useful tool for learning low-level programming.

Sammandrag

Trots att moderna högnivåprogrammeringsspråk betydligt kan förbättra produktivitet och effektivitet hos programmerare, så är den underliggande maskinkoden ofta dold. För studenter och andra som är intresserade av ämnet kan förståelse för maskinkod ge djupare insikter i programmering som helhet, samt öka förståelsen för hur datorer fungerar på en låg nivå. Assembler är den mnemoniska representationen av maskinkod vilket gör det till en praktisk ingångspunkt för den som vill förstå vad som sker på hårdvarunivå. Däremot är assembler ofta utmanande att lära sig, då språket bygger på CPU-arkitekturspecifika instruktioner och logik som skiljer sig fundamentalt från den högnivåsyntax som majoriteten av utvecklare förlitar sig på idag. För att bemöta dessa utmaningar, presenterar detta projekt `Fika`, ett statiskt typat språk som fokuserar på att visa sambandet mellan assembler och högnivåprogrammering. Att skriva kod i Fika är avsett att vara enkelt och intuitivt, vilket tillåter användaren att koncentrera sig på bemästrandet av grundläggande assemblerkoncept. Språket kompletteras även av flera verktyg, såsom en tillhörande language server, samt ett Visual Studio Code extension som möjliggör direkt visualisering av assemblerkod till motsvarande kod skriven i Fika. En användarstudie genomfördes med både nybörjare och mer erfarna programmerare. Resultaten från denna studie indikerar

att Fika ökar tillgängligheten av assembler samt upplevs som ett användbart verktyg för att lära sig lågnivåprogrammering.

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Professor Magnus O. Myreen, for meeting with us regularly, answering our questions, and providing valuable guidance throughout the entire project.

We would also like to thank the testers of Fika for taking the time to try our language and for their invaluable feedback.

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
CPU	Central Processing Unit
IDE	Integrated Development Environment
JDK	Java Development Kit
JSON	JavaScript Object Notation
llc	LLVM Static Compiler
LLVM IR	LLVM Intermediate Representation
LSP	Language Server Protocol
stdout	Standard output
URI	Uniform Resource Identifier
VS Code	Visual Studio Code

Contents

1 Introduction

1.1 Background

Initially, before high-level programming languages, computer programs were written entirely in machine code. This rudimentary way of constructing software essentially boiled down to writing sequences of binary numbers, where each instruction corresponded directly to a Central Processing Unit (CPU) operation [1]. Due to the binary nature of these computer programs, they were extremely difficult to read and understand. Additionally, even simple programs required significant effort to produce, and programmers needed a deep understanding of the underlying hardware architecture, since instructions varied between different manufacturers [1]. To illustrate, Listing 1.1 shows how a simple addition of two numbers is expressed in machine operation code (opcode), represented in hexadecimal for human readability.

Listing 1.1: Example of simple addition in machine code.

```
B8 05 00 00 00   ; Opcode for 'Move 5 into register EAX'
BB 0A 00 00 00   ; Opcode for 'Move 10 into register EBX'
01 D8            ; Opcode for 'Add EBX to EAX'
```

As a natural successor, assembly emerged as the first abstraction over machine code [1]. It allowed for commands such as "MOV" and "ADD" rather than raw machine instructions. This improved readability, but assembly still had some of the limitations that machine code had, such as specific hardware dependencies and differences, with instructions varying between different manufacturers [1].

Listing 1.2: Example of simple addition in assembly code.

```
mov eax, 5  ; Instruction for 'Move 5 into register EAX'
mov ebx, 10 ; Instruction for 'Move 10 into register EBX'
add eax, ebx; Instruction for 'Add EBX to EAX'
```

Fortran, released in 1957, took this abstraction further, moving away from the hardware dependencies and providing a common, unified, programming language regardless of what computer the code was written on [2]. By including an optimising compiler, a program that translates high-level source code to lower-level code [3], code written in Fortran produced programs that ran nearly as fast as the equivalent hand-crafted machine code. As IBM puts it, "Fortran democratised computer programming" ([2] §3) by providing a framework for writing optimised code that did not require knowing machine code [2].

Listing 1.3: Example of simple addition in Fortran.

```
PROGRAM ADD_NUMS
  INTEGER :: A, B, C
  A = 10
  B = 5
  C = A + B
END PROGRAM ADD_NUMS
```

Since then, programming languages and their accompanying tools have evolved significantly. Modern languages such as Java and Python provide powerful abstractions that allow developers to write complex software without needing to understand hardware-level details. Features such as automatic memory management, advanced code editors, code completion, and more recently, development assisted by Artificial Intelligence (AI) have greatly improved the user experience and accessibility. These advancements have made programming more approachable than ever before [4].

Listing 1.4: Example of simple addition in Python.

```
a = 10
b = 5
c = a + b
```

Another advancement in providing a better user experience when writing code is the introduction of language servers. A language server is a tool that provides language-specific features, such as hover information, go-to definition, code completion, and real-time error diagnostics. The Language Server Protocol (LSP) was created to provide a common interface for both language server and development environments (also called language client within LSP). LSP then allows these features to be implemented once and reused for any development environments supporting the protocol [5]. An example of the hover information provided by a language server is shown in Figure 1.1.

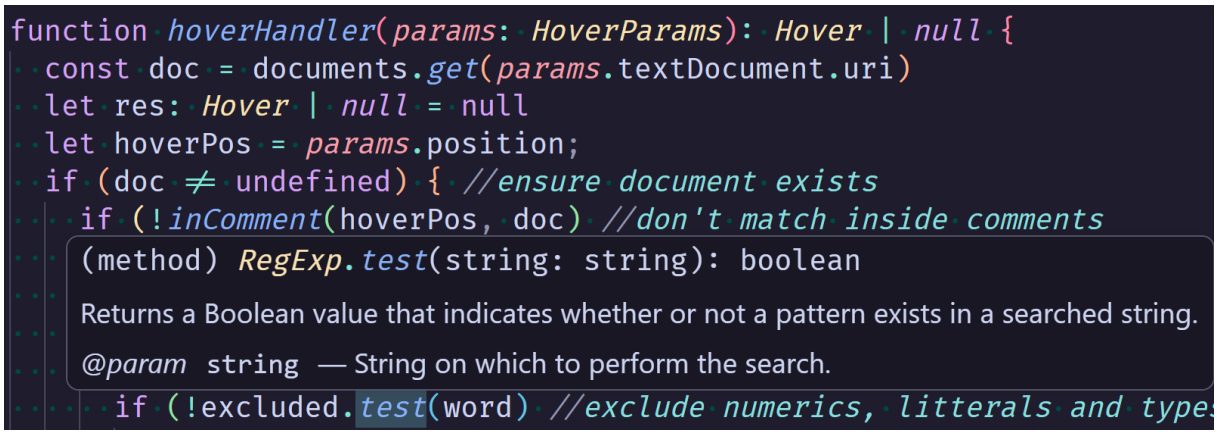

Figure 1.1: Hover information provided by the language server for TypeScript.

This separation of language intelligence from the editor has made it significantly easier to build advanced development tools, contributing to the modern programming experience

where immediate feedback and rich tooling are expected.

1.2 Purpose

The increased level of abstraction in programming languages also introduces a pedagogical challenge. While high-level languages improve usability, they often obscure the underlying machine code. Important low-level concepts such as memory management and register usage are hidden behind multiple layers of abstraction. As a result, students and programmers may find it difficult to understand how modern commands translate into actual machine instructions.

To address this challenge:

> This project presents a programming language and accompanying environment that explicitly visualises the relationship between high-level languages and their counterparts in assembly code, while still providing modern development features.

The project was guided by two main objectives:

A1 Support the teaching and learning of assembly language by providing a high-level abstraction while visualising the translation into low-level concepts on demand.

A2 Implement a programming language that includes characteristics and design considerations required for a programming language to be perceived as modern and usable today, while preserving transparency towards the underlying execution model.

With these objectives in mind, this project has explored the design and implementation of Fika, a high-level statically typed language that compiles to the low-level intermediate representation (IR), provided by the open source compiler infrastructure LLVM[1], with a Visual Studio Code extension providing live assembly visualisation. To evaluate whether Fika meets its objectives, a user study was conducted involving computer science students and a domain expert, combining think-aloud sessions with follow-up interviews to assess both the usability of the language and the effectiveness of the assembly visualiser as a learning tool.

1.3 Scope and Limitations

Designing and implementing a production-ready programming language typically spans years of development. Therefore, the scope and limitations of this project were defined in a way as to ensure that implementation remained feasible within the given time frame for this BSc project (18 weeks), whilst also maintaining a clear focus on the project's objectives A1 and A2 described above.

Thus, the implementation focused only on a handful of core language features. This meant that fundamental language constructs were prioritised over complex features such as polymorphism or support for multiple classes. Additionally, it does not include the development of a standard library beyond what is necessary to demonstrate the core language concepts.

[1] LLVM used to be an acronym, but now it's the full name of the project.

Furthermore, to support educational focus and reduce complexity, the project does not address manual or advanced compiler optimisations. Instead, existing optimisation mechanisms available through LLVM are utilised.

Finally, the project does not include the development of a custom integrated development environment (IDE). Instead, the aim has been to integrate the language with Visual Studio Code through extensions.

1.4 AI declaration

The project was carried out in accordance with Chalmers University guidelines for the use of AI tools [6]. The AI tools we used were Scopus AI[2], Claude[3], ChatGPT[4] and Github Copilot[5].

While we used Scopus AI to find relevant academic sources for the project, the generative AI tools were used primarily to help develop the language. We used them to generate ideas for solving implementation problems, to explain concepts such as code generation and assembly visualisation and to identify bugs more efficiently.

We also used the generative AI tools to generate some of the figures in the report and to assist with grammar, spelling, and phrasing.

[2]Scopus, `https://www.scopus.com/pages/home#scopus-ai`
[3]Anthropic, `https://www.anthropic.com/claude`
[4]OpenAI, `https://chatgpt.com`
[5]Github, `https://github.com/features/copilot`

2 Approach

This chapter describes the approach taken in the design and implementation of Fika as well as its associated tooling. The design decisions were driven by two overarching objectives A1 and A2, as described in Section 1.2. These objectives are closely related — the language needed to be simple and predictable enough that users could focus on understanding the underlying assembly output, rather than struggling with the syntax of the language itself. To inform these decisions, existing literature, widely used programming languages, and existing assembly visualisation tools were reviewed. Furthermore, two surveys were conducted to gather insights from the target user group regarding language expectations and syntactic preferences. Finally, the chapter describes the development workflow adopted by the team, as well as the testing and evaluation strategies employed throughout the project.

2.1 Language and Tooling Design Decisions

The language design and its associated tooling was guided by our two primary goals A1 and A2. The following subsections describe how existing tools, literature, and surveys informed the design decisions made to support these goals.

2.1.1 Design Principles and Influences

The design of the language was partly informed by a combination of established programming language concepts and practical observations from widely used languages. Rather than adopting a single paradigm, our goal was to identify and adopt key design principles, such as readability, simplicity, and type safety, with the specific requirement of making low-level execution transparent to the user.

Readability. A key principle in language design is readability, which has long been emphasised as essential for understanding and maintaining code [7]. This is particularly important in educational contexts, where reducing cognitive load can significantly improve the learning experience [8].

Due to its emphasis on simplicity and readability, we used Python as a primary source of inspiration. Python's design prioritises clear and intuitive syntax, making it especially suitable for beginners and educational settings [9]. This aligns with the goal of lowering the barrier to entry for new programmers (objective A2), which led us establishing the following decision:

Design Decision 1: Simple and Readable Syntax
The language syntax is to prioritise simplicity and readability, drawing primary inspiration from Python.

Type System. Python is dynamically typed, meaning that types are resolved at runtime rather than at compile time [9]. This contributes to its simplicity and flexibility, as the programmer does not need to declare types explicitly, and variables can be rebound to values of different types at any point. However, it also means that type errors may only surface during execution. For a language intended to expose the mapping between high-level code and assembly (objective A1), this is unsuitable for two reasons. First, dynamic typing obscures the relationship between values and their underlying memory representations, which is what the visualiser aims to make transparent. Second, a user should be able to identify and correct type-related issues before observing the generated assembly, rather than encountering unexpected behaviour at runtime.

In contrast, statically typed languages provide advantages in terms of program correctness and reliability. Statically typed languages enforce constraints on how values can be used, reducing the likelihood of runtime errors [10]. Languages such as Java exemplify this, offering structured and explicit type systems [11].

Furthermore, explicit and predictable types have a direct impact on the clarity of the generated assembly output. When generating the code, each type, such as an `int` versus a `double`, maps to a specific memory representation and register size. A statically typed language where the underlying type is made explicit therefore produces more consistent and predictable assembly code, making the mapping between high-level constructs and low-level instructions easier to follow.

The benefits that static typing provides in terms of type safety and direct mapping to assembly instructions, led us to the make the following decision:

Design Decision 2: Static Type System
The language is to employ a static type system to ensure type safety and predictable assembly output.

Type Inference. Requiring the programmer to write type annotations everywhere can introduce friction, particularly for beginners. Type inference offers a middle ground, where the compiler automatically deduces types from context without requiring explicit annotations. This concept has been fundamental in programming language design since being formalised by the Hindley-Milner type system in the 1980s [12], and has long been central to functional languages such as Haskell and OCaml [13, 14]. Several mainstream languages have since introduced forms of type inference. Java for instance, has a limited form of inference with the `var` keyword introduced in JDK 10 [15]. The keyword allows variables to be declared without explicit type annotations. For example `var x = 2;` is interpreted as an `int`, whereas `var x = 2.2;`, is interpreted as a `double`.

The convenience of type inference aligns well with objective A2, as it reduces boilerplate and lowers the barrier to entry for beginners without sacrificing the correctness guarantees of a statically typed language. However, it connects less naturally with objective A1,

when types are implicit, the relationship between a value and its underlying memory representation becomes less visible, which works against the goal of making the mapping between source code and assembly transparent. For this reason, we aimed to find a balance between the convenience of type inference and the transparency of explicit type annotations, by allowing the compiler to infer types while ensuring that the inferred types are always made visible to the user through editor annotations. This aim for balance led to the following design decision:

Design Decision 3: Annotated Type Inference
The language is to support type inference, and users should always have the option to have the inferred types be displayed or inserted automatically.

Language Tools. To further align with objective A2, we need to implement the supporting tools expected from a modern programming language today. Tools such as `syntax highlighting` and automatic indentation help with readability and source code structure. Meanwhile `code completion` can let developers work faster, whilst `hover` provides easy access to information about functions and variables. To establish which language tools this project should implement, we included questions about what supporting features our target demographic prefer. The answers and decisions are presented in Section 2.1.3.

2.1.2 Assembly Learning Tool Decisions

To inform the design of the visualisation tool (objective A1), we conducted a review of existing tools, focusing on how effectively they communicate the relationship between high-level code and assembly to users unfamiliar with low-level programming.

One visualising tool that we reviewed was the Visual Studio Code (VS Code) Compiler Explorer extension [16], which integrates the Compiler Explorer tool [17] directly into Visual Studio Code. This extension provides a real-time side-by-side view of high-level source code and its corresponding assembly output. While the Compiler Explorer extension is a well-functioning tool that supports translation into assembly for multiple existing programming languages, we found it better suited for experienced developers with pre-existing knowledge of assembly. This is due to the generated code being presented as raw, unannotated output, requiring the reader to already have a working knowledge of assembly language to interpret it. Several similar extensions were also identified, and like the Compiler Explorer tool, none of them included explanatory annotations or comments in the generated assembly.

However, certain aspects of the Compiler Explorer extension [16] also positively influenced our design decisions. We thought that the use of syntax highlighting and indentation in the assembly output, rather than presenting the code as a single block of uniform text, improved readability considerably. Additionally, we found the ability to highlight a section of source code and immediately see the corresponding assembly to be valuable for learners.

Both the gap and the strengths identified in existing tools were considered when designing the assembly visualiser for Fika. Rather than targeting developers who already understand assembly, we wanted to design the visualiser as an aid for students who are inexperienced with low-level code. Based on the drawbacks identified in existing tools, we

decided that the generated assembly output should be annotated with human-readable comments that explain what each instruction does and how it relates to the original high-level construct, establishing a clear and explicit mapping between the two levels of abstraction. At the same time, the syntax highlighting, indentation, and the ability to map source code sections directly to their corresponding assembly output were all identified as characteristics worth incorporating into the visualiser.

This approach is further supported by research in computing education, which suggests that visualising tools can play a significant role in helping students understand abstract and dynamic concepts by making otherwise invisible processes more concrete [18]. The visualiser is not intended as a stand-alone tool for learning assembly from scratch, but rather as a supplementary aid that contextualises assembly within a familiar programming environment. By writing code in a high-level language and immediately observing the annotated assembly it produced, students could develop an intuitive understanding of how high-level constructs map to low-level instructions, without needing prior assembly knowledge to get started.

Together, these findings motivated the following design decision:

Design Decision 4: Assembly Visualiser with Explanatory Annotations
The assembly visualiser is to annotate generated assembly with human-readable comments explaining each instruction. The tool is also to support interactive source-to-assembly mapping, where clicking a line in the source code highlights the corresponding assembly instructions.

2.1.3 Survey 1: Modern Programming Languages

To better understand the target group's expectations of a modern programming language, we conducted a survey early in the project. The survey targeted university students in the computer science field, as they are well versed in programming and may contribute with valuable feedback. The goal of this initial survey was to collect opinions that could guide the design decisions for the language and its features.

The questionnaire had a total of nine questions, organised around three themes. The first theme concerned which languages respondents had used and found easiest to learn, in order to identify which languages were most relevant to study and draw inspiration from. The second theme focused on which tools and language features respondents valued the most, to guide the prioritisation of tools and features during development. The third theme asked about preferred development environments, to determine which IDE to target for language support. The full questionnaire is to be provided in Appendix B along with the key results.

A total of 76 participants completed the survey. The results that influenced the language design the most are presented below, highlighting the key findings that influenced the design decisions for the language and its development tools.

Easiest to Learn. Python was considered the easiest language to learn by the majority of respondents, as shown in Figure B.1.

This result, in addition to the result shown in Figure B.3, indicates that students tend to

favour languages with simple and readable syntax when learning programming. Based on these results, the language was designed with a strong focus on simplicity and readability, aiming to reduce the learning curve and make the language as accessible as possible from the start. These results further strengthen the choice of Design Decision 1.

Most Appreciated Characteristics. When asked which programming languages they used most, respondents frequently mentioned Java and C-family alongside Python. This was taken into consideration in the design process, particularly with regard to the type system. As shown in Figure B.3, 42.1% of respondents identified strong typing as one of the most appreciated characteristics of a programming language. Consequently, the type system was inspired by languages such as Java, which provide stronger static type guarantees compared to dynamically typed languages like Python. This further supports Design Decision 2, that concluded the language will employ a static type system.

Tools and Features. The majority of respondents identified clear error messages, good documentation, and syntax highlighting as the most important tools to support a modern programming language, as shown in Figure B.2.

These findings directly influenced the prioritisation of tooling features during development, leading to the following decision:

> **Design Decision 5: Language Tooling Priorities**
> The development of language tools is to prioritise clear error messages, documentation, and syntax highlighting.

Preferred IDE. The most commonly preferred IDE, with a majority vote of 59.2%, was Visual Studio Code, as shown in Figure B.4.

Based on this result, the project focused on integrating language support primarily within VS Code, rather than targeting multiple IDEs, as summarised in the following decision:

> **Design Decision 6: Primary IDE Support**
> Language support is to be focused on Visual Studio Code as the primary development environment.

2.1.4 Survey 2: Keyword Preference

With simplicity and readability established as core design principles (Design Decision 1), we conducted a keyword preference survey early in the design phase to guide specific syntax decisions. The goal was to determine which syntactic choices felt most intuitive to our target audience, in particular whether English keywords or traditional symbolic operators would better support the readability goal. To maintain consistency with the previous survey, this survey targeted the same audience group: university students in computer science and related fields. The final decisions are summarised in the end of this section.

The questionnaire had a total of seven questions that compared traditional symbolic operators, such as those found in Java and C, against English keywords. The goal was

to test whether English keywords would feel more intuitive and easier to learn, as they resemble natural language. The questions were focused on four areas: logical operators, comparison operators, postfix operators, and boolean literal casing. The full questionnaire can be found in Appendix C.

A total of 92 people completed the questionnaire[1]. As shown in Figure C.1, the respondents represented a balanced distribution across experience levels, with the largest groups having 6-10 years (31.5%), 4-5 years (29.3%), and 2-3 years (28.3%) of programming experience. This diversity provided valuable insight into what users with different backgrounds find most readable.

Below we present the results that most influenced our syntax decisions.

Logical Operators. Figures C.2 and C.3 show a slight preference for English keywords `and` and `or`. However, Figure C.4 shows a clear preference for the symbol `!` over the keyword `not`. As `and` and `or` were in the majority, and due to Design Decision 1, we chose to adopt these keywords. Despite `not` being in the clear minority, we still chose `not` for consistency across all three logical operators.

Postfix Operators. Figure C.5 shows a strong preference for postfix increment and decrement `++`, `--` over standard function calls. Postfix operators were adopted accordingly.

Comparison Operators. One representative question used `>=` as an example, assuming preferences would generalise across all comparison operators. As shown in Figure C.6, there was nearly a unanimous preference for the symbolic version over the English keyword. Symbolic operators are to be used based on this result.

Boolean Literals. Figure C.7 shows that lowercase boolean literals `true`, `false` were strongly preferred over the other alternatives. Lowercase boolean literals were adopted based on this result.

Together, these results informed the following syntax decisions:

Design Decision 7: Keyword and Operator Syntax
Based on survey results, the following syntax decisions are to be used:

- `and`, `or`, `not` for logical operators
- Postfix increment and decrement (`++`, `--`)
- Symbolic comparison operators (`<`, `>`, `==`, etc.)
- Lowercase boolean literals (`true`, `false`)

2.1.5 Language Tools

With the core syntax decisions established, the following subsection outlines the implementation approach taken for the tools prioritised in 5. In addition to documentation, error messages, and syntax highlighting, warnings and language configurations were also identified as valuable additions. For each feature, two approaches were considered: implementation via the language server or via the VS Code API.

[1]Note that two questions were revised, reducing the response count to 87 for those questions.

In general, implementing functionality via the language server is preferred over the VS Code API due to its modularity, extensibility and support for multiple IDEs. In contrast, the VS Code API can sometimes be better suited for functionalities inside of VS Code, but with the disadvantage of being limited to the VS Code editor.

Documentation and Diagnostics. Documentation is a central aspect of the project, as defined in Design Decision 5. Effective documentation not only requires well written descriptions of language constructs, but also convenient access to this information during development. This can be achieved through a hover functionality, which can be provided by the language server.

Similarly, error messages can be implemented through language server `diagnostics`. Diagnostics also allow for several different levels of severity and lets us also provide warnings for less severe cases. Warnings can be used to highlight cases where the code is still valid and will compile, but may lead to unintended behaviour or suboptimal design. For instance, `while(true)` can trigger a warning, as it results in a non-terminal loop.

Since both diagnostics and hover information are supported by the language server protocol, it was decided to implement both of these features within the language server, leading to the following decisions:

Design Decision 8: Documentation Accessible using Hover
Easy access to documentation is to be provided by the language server using hover capabilities.

Design Decision 9: Error and Warning Messages through Diagnostics Support
In order to provide clear and easy to understand error and warning messages, the language server must provide for diagnostics.

Source Code Highlighting. For the implementation of source code highlighting, there is one approach supported by the LSP called semantic highlighting, which is closely related to syntax highlighting in the sense that both colour-code the source file [19]. Semantic highlighting is not only capable of producing the same highlighting, but also more context dependent results.

However, semantic highlighting also requires the language server to repeatedly analyse the source document as well as constantly communicate between the client (editor) and server. The language server will need to wait for the processes gathering the highlighting information before being able to provide it to the client. The gathering process can take noticeable time and lead to delays where the highlighting is not updated in time while the user types. These delays make the semantic highlighting better suited at being an extra, additional, feature that only provides highlighting for the tokens that require deeper context awareness, on top of a separate syntax highlighting [19].

With this in mind, we decided to develop the `Fika language server` to be able to provide diagnostics and hover information. Meanwhile source code highlighting is implemented specifically for VS Code and extending the language server to support additional semantic highlighting is left as a point for potential expansion.

Design Decision 10: Syntax Highlighting over Semantic Highlighting
Simple syntax highlighting is to be prioritised over semantic highlighting, to ensure a responsive editing experience.

Language Configurations. Within the VS Code extension API there exists a collection of relatively small and simple tools called Language Configurations. These are written in a JSON file and when defined, enables, amongst others the following features:

- Comment toggling
  Allows keyboard-shortcuts to toggle line and block comments.
- Auto closing
  Writing an opening bracket automatically inserts a closing bracket after the cursor.
- Auto surrounding
  Enables the automatic surrounding of highlighted regions with the specified characters.
- Indentation rules
  Automatic indentation for the specified patterns

Due to the relatively small amount of effort required to implement these features, we decided to include all the features in the list above as part of the VS Code extension, leading to the following decision:

Design Decision 11: Editor Support through Language Configurations
Comment toggling, brackets definition, auto closing, auto surrounding, folding, word pattern and indentation rules are be implemented using VS Code Language Configurations.

2.2 Development Process

With the design decisions established, the following subsections describe how the workflow was structured to develop Fika. The development was organised around three key considerations:

1. The overall workflow and team structure adopted by the team.
2. The testing strategy employed throughout the implementation.
3. The evaluation strategy used to assess the final system.

2.2.1 Development Workflow

The development workflow of the programming language and its supporting tools followed an iterative and incremental approach, meaning that features were developed and refined in cycles rather than all at once.

The core compiler components required sequential implementation, as each stage depends directly on the output of the previous one, as described in detail in Section 3. This

sequential dependency was one of the core reasons we chose an incremental approach, which allowed us to define a subset of the grammar and implement the corresponding pipeline stages for that subset, before gradually extending the language with additional constructs. This way, we could test each part of the pipeline incrementally rather than attempting to implement the full language at once.

Some of the language tooling however, such as the syntax highlighting, could be developed in parallel with the core compiler, as it depended only on the language design/syntax decisions rather than on the internal compiler stages. For this reason, two of the six team members focused on these features from an early stage, while the remaining four concentrated on the core language implementation.

2.2.2 Testing Strategy

The incremental development process described above meant that testing was an essential part of each development cycle rather than an afterthought. Since each stage of the compiler pipeline depends directly on the correctness of the previous one, it was crucial that each component was thoroughly tested before the next stage could begin. For example, before code generation for a subset of the language could start, the earlier stages responsible for reading and structuring that code had to produce the correct output. To ensure correct functionality and expected behaviour, tests were written alongside the code that was currently in development

Primarily, unit tests were written to validate the correctness of individual components, such as the type checker and code generator. Unit tests were chosen as the primary testing method as they allow individual components to be verified in isolation, making it straightforward to pinpoint the source of a failure without interference from other parts of the pipeline.

The unit tests consisted of "valid" and "invalid" tests. The "valid" tests were expected to pass to ensure correct input produced the expected output. Conversely, the "invalid" tests were made to ensure that erroneous input was handled and produced appropriate error messages rather than silent failures or crashes. Each stage of the compilation pipeline was tested in this manner before development progressed to the next.

While this approach ensured that individual components behaved correctly in isolation, test coverage was not formally measured, and some edge cases were discovered later in development as components were integrated.

2.2.3 Evaluation Strategy

The evaluation strategy of the final design was designed around the two primary objectives of the project. The first was to assess whether the language and its accompanying assembly visualiser can support the understanding of assembly language A1, and the second was to evaluate the usability and intuitiveness of the language itself A2.

To address these objectives, a think-aloud evaluation method was selected, in combination with a follow-up interview. Nielsen and Landauer [20] describes think-aloud as a valuable observational technique that provides insight into the user's decision-making process and reveals usability issues that may not be apparent through expert analysis alone. The

follow-up interview complements the think-aloud session by giving participants the opportunity to reflect on their experience and elaborate on aspects that may not have been fully expressed during the task itself.

The evaluation was primarily conducted with participants representing the target user group: computer science students with at least one year of high-level programming experience but limited exposure to assembly language. The target number of participants was aimed to be around five to ten. This range is supported by research showing that a single test participant discovers approximately 31% of usability problems on average, and that testing with five users is sufficient to uncover around 85% of usability issues [20].

Participants were given a set of focused tasks of increasing complexity, designed to naturally engage both the language syntax and the assembly visualiser. By observing how users interacted with the system and examining whether they were able to interpret the generated assembly output, the evaluation aimed to provide qualitative insight into both the usability and the educational value of the system

Prior to the main evaluation sessions, a pilot test was conducted with a single participant. According to Dix et al, this strategy allows for task design and interview questions to be refined before the primary sessions take place [21]. Usability issues uncovered during the pilot were reviewed, and where there was sufficient confidence that they represented genuine problems rather than isolated incidents, they were addressed before the remaining sessions.

To complement the primary evaluation, an additional think-aloud session was also conducted with the project supervisor, a Computer Science professor at Chalmers University of Technology, who acted as a domain expert. The session followed the same formal structure as the student evaluations, but the tasks and follow-up interview questions were adapted to better match the supervisor's level of expertise. Rather than focusing on whether the language syntax felt intuitive, the tasks were directed towards the generated assembly output, and the interview questions were centred around the technical correctness and clarity of the assembly annotations. This provided a complementary perspective to the student evaluations. While the student sessions assessed usability and the learning experience, the expert session assessed the technical accuracy of the visualiser's output.

3 Implementation

This chapter describes the implementation of Fika and its associated tooling. It covers the full compiler pipeline, from lexing and parsing to LLVM Intermediate Representation (LLVM IR) code generation, as well as the Visual Studio Code extension that provides syntax highlighting, error diagnostics, type inference visualisations or insertions, and the assembly visualiser. The reasoning behind key implementation choices is discussed alongside the technical details.

3.1 Language Implementation

The language implementation follows a classical compiler pipeline, where source code written by the user is progressively transformed through a series of stages before producing the final assembly output. Each stage has its own responsibility and produces output that serves as input for the next stage. An overview of the pipeline is illustrated in Figure 3.1.

The pipeline begins with a lexical analysis, and parsing. The lexer is the first part of the pipeline. Its responsibility is to read the raw source code as a sequence of characters and group them into meaningful units called tokens, such as keywords, identifiers and operators. The parser then takes this token sequence produced by the lexer and verifies that it conforms to the grammatical rules of the language, producing a structured representation of the program, called a parse tree.

This structured parse tree is then transformed into an AST, which is a simplified representation of the program, that captures the hierarchical structure. For example, an if-statement is represented as a node with a condition and two branches (`then`- and `else`-branch), in a form that is independent of the original source text and easier to analyse in the following stages.

The type checker then traverses the AST and verifies that all operations in the program are type-safe. The type checking stage thus includes resolving inferred types, checking that operands are compatible, and ensuring that functions are called with the correct argument types. Any violations are reported as type errors, which later on are shown to the user directly in the editor. During the type checking, the AST nodes are supplied with its corresponding types, for instance the `x < 2` expression will be given a `boolean` type, as it produces a response that is either `true` or `false`. After the type checker has processed the AST, it will have produced a type-annotated AST.

Finally, the code generator traverses the type-annotated AST and produces LLVM IR code. Each high-level construct is translated into one or more LLVM IR instructions, which represents the program in a low-level, platform-independent form. The LLVM

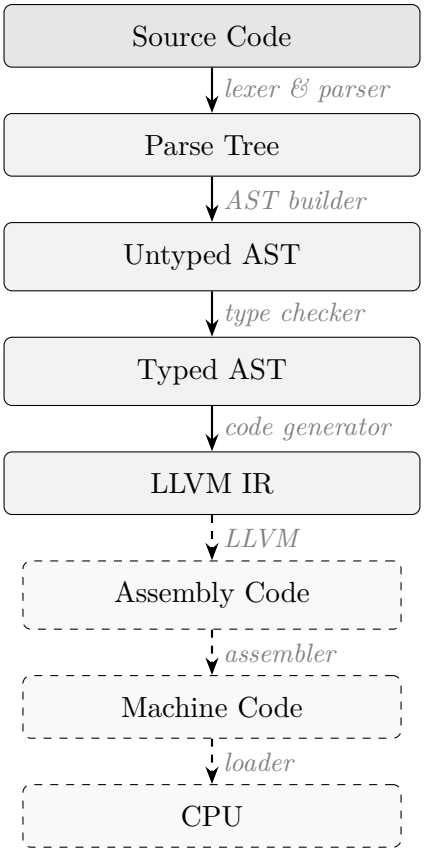

Figure 3.1: The Fika compiler pipeline. The dashed box and arrow indicate steps not handled by the Fika compiler itself.

toolchain then compiles this IR into native assembly code, which is subsequently assembled into executable machine code that is executed by the CPU. The transformations into assembly and machine code are handled directly by the LLVM toolchain without any additional tooling. This approach avoids the need to implement a custom assembly backend, instead leveraging LLVM's well-tested compilation infrastructure [22].

The following subsections describe the implementation and design decisions behind each of the stages implemented as part of the Fika compiler in further detail, meaning the lexer and parser, the AST, the type checker, and the code generator.

3.1.1 Lexer and Parser

The lexer and parser were implemented using ANTLR4 (ANother Tool for Language Recognition version 4). The choice of tooling was driven primarily by the built-in support for error recovery that ANTLR4 provides, which allows the parser to continue processing input even when syntax errors are encountered. The ability to continue processing was essential for the language server integration. It is desirable for partially written or erroneous programs to still produce a meaningful parse tree for the rest of the compiler pipeline to process, so that the language server can continue to provide diagnostics and error messages.

The grammar of the language was therefore defined using ANTLR4, from which both the lexer and parser were automatically generated. This was achieved by defining the

language in a `.g4` file, which specifies both the syntactic structure of valid programs (parser rules) and the lexical tokens of the language (lexer rules). This is the file where the syntax decisions 1 and 7 are realised, for instance boolean literals were defined in all lower case, like `true` and `false`.

The grammar for Fika is structured around a top-level `program` rule, with separate rules for definitions, statements, and expressions. Expressions are defined using a precedence-based hierarchy to ensure correct evaluation order.

An example of the grammar is shown below:

```
program : separator* (def (separator+ def)*)? separator*
   EOF;

def
     : func  #DFunc
     | stmt    #DStm
     ;

stmt
     : simpleStmt
     | blockStmt
     ;
```

This excerpt illustrates how a program consists of one or more definitions, where each definition is either a function or a statement. This division reflects a distinction in the language, where functions are named, reusable blocks of code that can accept parameters and return values, whereas statements are the building block that make up the function's or programs body, executed in order.

As illustrated in the example, statements are further divided into simple statements and block statements. Simple statements are single-line constructs that produce or manipulate a value, such as variable assignments or expressions. Block statements, on the other hand, are control flow constructs such as `if`-statements and `while` loops, which contain a body of further statements.

Expressions are a subset of simple statements and represent constructs that evaluate to a value, such as arithmetic operations, function calls, or comparisons. The distinction between statements and expressions is important because statements do not produce any values that can be used further. For example, a `while` loop executes a block of code repeatedly, but does not itself evaluate to a value, meaning it would not make sense to pass a `while` loop as an argument or use it in a condition. Expressions on the other hand, always produce a value that can be used further, such as being passed as an argument or used in a condition.

Given a grammar, ANTLR4 produces a parse tree representing the full syntactic structure of the input program. The next step is to transform this into an `Abstract Syntax Tree`.

3.1.2 Abstract Syntax Tree

The Abstract Syntax Tree serves as a simplified and semantically meaningful representation of the program, derived from the parse tree generated by ANTLR4. While the parse

tree reflects the full syntactic structure of the input, including syntactic details such as delimiters and punctuation, the AST abstracts away these elements and retains only the constructs relevant for further analysis and code generation.

We implemented the AST in Java as a sealed interface hierarchy, where each node type corresponds to a specific language construct, such as a program, a function, a statement, an expression, or a type. The structure of the AST closely mirrors the grammar defined in the lexer and parser stage, where constructs such as `Program`, `Stmt` and `Exp` correspond directly to the top-level grammar rules described in Section 3.1.1. However, where the parse tree preserves every syntactic detail of the grammar, the AST retains only the semantically meaningful elements.

An excerpt of the AST structure is shown below:

```
public record Program(List<Stmt> stmts, List<Func>
    functions) {}

public sealed interface Stmt extends HasPos permits
    BlockStmt, SimpleStmt  {}

public sealed interface Exp extends HasPos, HasType
    permits EInt, EDouble, EString, EBool, EId, ECall,
    EOpp, ...

public record EOpp(Exp left, Exp right, Op op, Type type,
     Pos pos) implements Exp {}

public sealed interface Type permits TInt, TDouble,
    TString, TBool, TUnknown, TArray {}
...
```

This excerpt illustrates the top-level structure of the AST interface, for instance how a program consists of a list of statements and a list of functions. It also shows how all nodes implement `HasPos` for precise source position tracking, and how expressions implement `HasType`, allowing types to be attached and refined incrementally during type checking.

Transforming the parse tree into an AST was achieved by traversing the nodes of the parse tree using a visitor-based approach. In this approach, each node type in the parse tree has a corresponding visitor function that defines how that node should be transformed. This makes the transformation logic easy to follow, as the handling of each syntactic construct is defined in one place. During the traversal, syntactic structures are mapped to corresponding AST nodes. For example, an expression consisting of two operands and an operator in the parse tree can be recognised and transformed into an `EOpp` node, which explicitly represents an operator together with its two operand expressions.

Figure 3.2 illustrates the resulting AST structure for the statement `return a + b`. The program decomposes into an `SReturn` node, representing a return statement. Unlike the nodes beneath it, `SReturn` is a statement, as it executes an action rather than evaluating to a value that can be used in further computation. The `SReturn` node contains its return value, in this case the `EOpp` node. The `EOpp` node and its two `EId` children are expressions, each evaluating to a value that are passed up the AST and finally used by

the return statement.

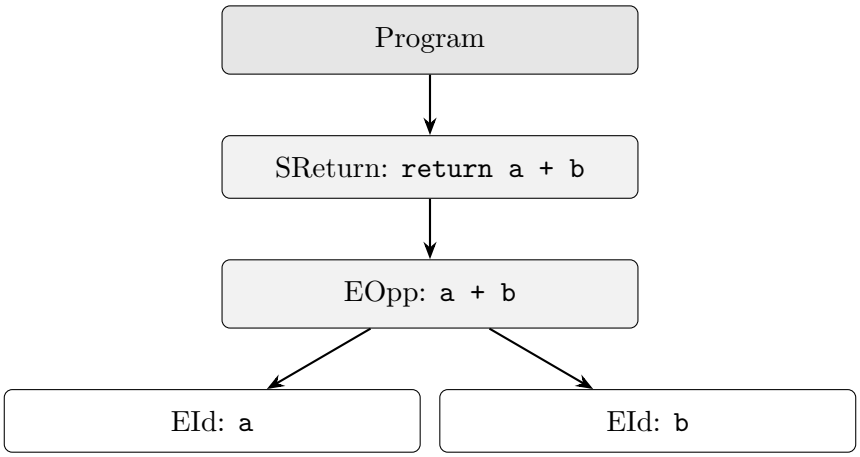

Figure 3.2: AST structure of the return statement `return a + b`, showing how a program decomposes into statements and expressions.

Since the language supports type inference, not all types can be resolved during this traversal. Variables and functions whose types cannot yet be determined are assigned a temporary placeholder type, `TUnknown`, allowing the AST to be constructed in a single pass without requiring full type information upfront. The resolution of `TUnknown` nodes is handled subsequently by the type checker, as described in the following section.

3.1.3 Type Checker

Once the AST is created, it can be passed into the type checker, whose responsibility is to ensure that all operations are type-safe and to resolve any remaining inferred types. By performing these checks at compile time, type errors are caught before the program runs, directly realising Design Decision 2.

To be able to type check the program, the AST is traversed once again, this time using a recursive pattern matching approach, where each expression node type is handled by its own dedicated `typeCheck` method. This recursive approach was chosen as it leverages Java's switch expressions with pattern matching [23], allowing each AST node type to be dispatched to its corresponding type checking logic in a concise, modular and readable way.

As shown in the following example, the top-level `typeCheck` method acts as a dispatcher, delegating the responsibility of each type check to its specific handler.

```
public Ast.Exp typeCheck(Ast.Exp exp) {
        return switch(exp) {
            case Ast.EInt eInt -> typeCheck(eInt);
            case Ast.EDouble eDouble -> typeCheck(eDouble
               );
            case Ast.ECall eCall -> typeCheck(eCall);
            case Ast.EOpp eOpp -> typeCheck(eOpp);
            ...
        };
}
```

Each handler is then responsible for verifying the types of its specific node. A simplified example of the handler for the `EOpp` expression (consisting of two operand expressions and one operator) could look as follows:

```
public Ast.Exp typeCheck(Ast.EOpp eOpp) {
    Ast.Exp left  = typeCheck(eOpp.left());
    Ast.Exp right = typeCheck(eOpp.right());
    ...
    // Booleans are not valid operands for arithmetic
       operators
    if (left.type() instanceof TBool || right.type()
       instanceof TBool) {
        throw new TypeException(
            "Arithmetic operators require numeric
               operands",
            eOpp.pos()
        );
    }
    return eOpp.withType(left.type());
    ...
}
```

In this example, the type checker recursively checks both operands before verifying that their types are compatible, meaning that the operator can be applied to both operand types. For instance, adding two `int` values is valid, whereas applying `+` to a `bool` is not, as boolean values do not support arithmetic operations in Fika. If the types are incompatible, a type error is recorded and later displayed to the user in the editor. Note that in practice, the full implementation also handles additional cases such as implicit widening from `int` to `double`, string concatenation coercions, and `TUnknown` types arising from type inference.

Type Checker Context

During type checking, the type checker needs to keep track of which variables are currently in scope and what types they have been assigned. To clarify, a variable being in scope means that it is accessible within the part of the code that is currently running. A scope in Fika is anything within curly braces, including functions, control flow constructs, or blocks. For instance, a variable declared within a function will not be accessible from outside of that function, since it does not belong to that scope. Listing 3.1 illustrates this example.

Listing 3.1: Example scopes in Fika.

```
1 func foo() {
2     int x = 2
3 }
4
5 print(x) // This will not compile, x is not declared
     outside of foo's scope.
```

To keep track of which variables are declared within a scope, we are using a shared `context`. The `context` keeps track of the variable name/ID, mapping each ID to its type. For instance, after writing `int x = 1`, the context will be updated with a mapping from `x` to `int`.

Type Checker Traversal

An example of how the type checker traverses the expression `a + b`, where both operands are of type `int`, is shown in Figure 3.3. The process begins at the `SExp` node, representing a statement consisting of a single expression. The type checker then evaluates the enclosed expression, which is an `EOpp` node representing a binary operation.

To type check this operation node, the type checker recursively evaluates its two sub-expressions. This results in two independent traversals, where the types of `a` and `b` are retrieved from the typing `context`. Once both operand types have been determined, control returns to the `EOpp` node, where the operator is validated against the operand types.

Since both operands are of type `int` in Figure 3.3, the operation is valid, and the resulting type of the expression is inferred as `int`. Note that this example implies that both `a` and `b` have been previously assigned type `int`. If the variables had not yet been declared, that would produce a type error.

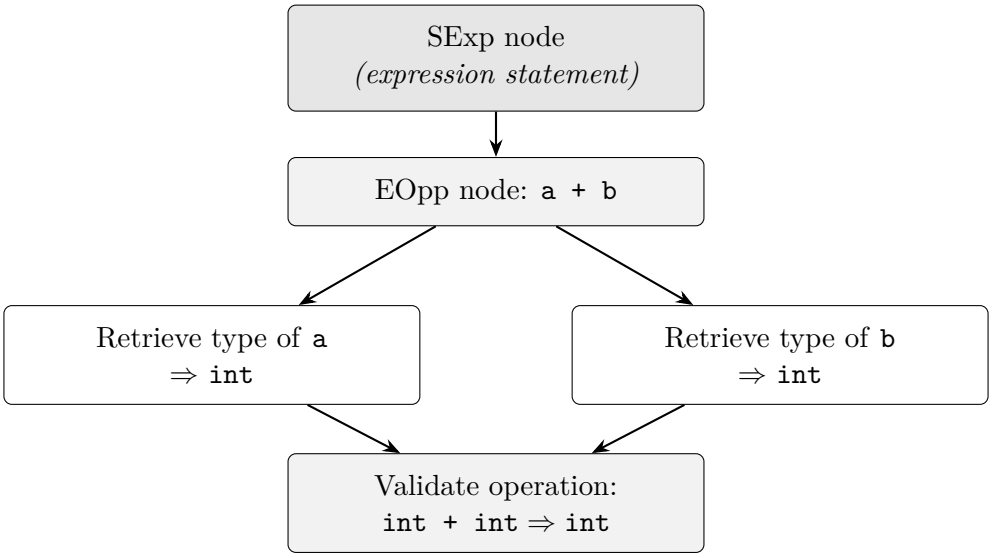

Figure 3.3: Type checking of the expression `a + b`.

.

Type Inference

As described in Section 3.1.2, the language supports type inference, meaning some nodes may still carry a `TUnknown` type after the initial AST construction. The type checker needs to populate the expressions with `TUnknown` types with explicit types, for the code generator to be able to generate code for them.

The inference was initially handled in a simple and straightforward manner. For instance, when it encountered an initialisation statement (`SInit node`) without an explicitly annotated type, meaning its recorded type was `TUnknown`, it directly inferred it to the

initialisation value type. An example of this is `x = "hello"`, where the type checker will be able to infer that `x` is of type `string`, as its value `"hello"` is of type `string`. The mapping of `x` to `string` can thus also be stored in the `context`.

While the simple method described above works in some cases, it is not sufficient in general. For instance, an SInit contains both the variable, as well as the value expression, which means it contains all information needed for inferring its type. However, more complex cases exist where the type needs to be inferred from an expression that is not contained inside of node itself. For instance, a declaration node for `x;` with no explicitly annotated type contains no information about the value of x. The value might be assigned much later, in an assignment expression `x = "hello"`. This creates difficulties, since the necessary type information is not yet available at the point where the declaration is visited in the type checking phase.

A natural solution to the problem described above is to allow declaration nodes to retain their `TUnknown` type during an initial traversal, and then perform a second pass over the AST, as visualised in Figure 3.4. This enables the nodes to be revisited once additional type information has been collected.

During the first pass, the type checker traverses the program and updates the `context` with any type information that can be determined. In the example `x = "hello"`, the assignment provides sufficient information to infer that `x` is of type `string`, and this mapping is stored in the context. The inferred type must remain consistent across all subsequent uses of the variable. For example,

```
x           // declaration
x = "hello" // attempt to assign string value to x
x = 1       // attempt to assign int value to x
```

results in a type error, since `x` cannot simultaneously be of type `int` and `string`. Such inconsistencies are caught and reported as type errors.

In the second pass, the AST is traversed again, this time with a more complete context. When encountering the declaration `x;`, the type checker can now retrieve the previously inferred type of `x` from the context and update the node accordingly. In this way, types that could not be resolved during the initial traversal can be consistently inferred in the second pass.

Similarly to variable declarations, the return type of a function can be resolved during the second pass. For instance, the following example of a function named `foo` has no explicit return type declared, but it returns the value 2. During the first pass, the return value will remain as `TUnknown`, and the `return 2` statement will update the `signature` context, a separate context for functions, that maps each function name to its return types and parameter types. During the second pass, the function can thus correctly retrieve the return type `int` by making a lookup in the `signature` context.

```
func foo() {
    return 2
}
```

However, function parameters are not as straightforward. In the case of a function call, the parameters can simply retain their types in the second pass. However, if the function

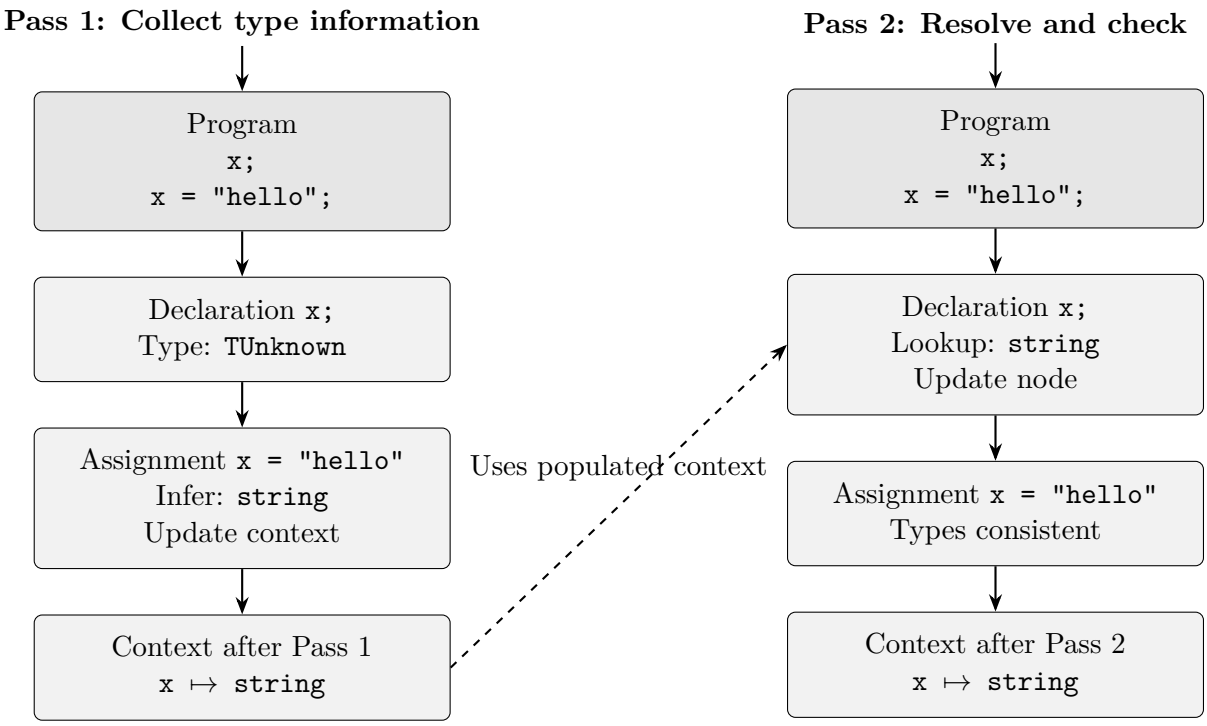

Figure 3.4: Two-pass type inference and checking for `x; x = "hello";`

is never called, but the parameters are used within the function, type checking is still necessary to prevent the user from retrieving runtime errors. Consider the following example, where `bar` has one parameter called `x`, and contains three statement expressions.

```
1 func bar(x) {
2     x > 10
3     x++
4     x = "hello"
5 }
```

In Fika, `x > 10` is only possible with numeric types, meaning `double` or `int`. The second expression `x++`, is only permitted for type `int`, no problem yet. However, the third expression is only permitted if `x` is of type `string`. Thus, there is a contradiction that should lead to a type exception, even if the function is not yet called.

To solve this, parameters that are not explicitly typed are initialised with a special `TUnresolved` type which accumulates type conditions as the AST is traversed. These conditions must remain mutually consistent, where a contradiction results in a type error.

When visiting the first statement of `bar`, `x` gathers two conditions, `TInt` and `TDouble`, as it can still be of any numeric type. The second statement, `x++`, is only permitted for `int`. Since the conditions for `x` already include `TInt`, it will eliminate the `TDouble` as a condition and infer `x` as `TInt`, and `TInt` only. However, the third statement attempts to gather the type `TString` as a condition, which is not possible as the conditions currently do not include `TString`. This will result in a contradiction, and therefore a type error. This process is summarised below:

```
x: TUnresolved
after x > 10: conditions = {TInt, TDouble}
after x++: conditions = {TInt}
after x = "hello": contradiction -> Type Error
```

Discussion of Type Inference System

While the type inference system we implemented was sufficient for the needs of our language, the design tended to grow incrementally and evolve as increasingly complex inference cases were encountered during development. The implementation relies on multiple AST traversals and specialised handling of unresolved parameter types through `TUnknown` and `TUnresoved` nodes.

A more principled and formally established approach to type inference is the Hindley–Milner type system [12]. Rather than resolving types through multiple specialised passes, Hindley–Milner represents unknown types as type variables and generates typing constraints while traversing the program. These constraints are then solved through a process known as unification, allowing the system to infer the most general valid type for each expression.

Compared to Hindley–Milner, the implemented solution is less general and more ad hoc, but was considered appropriate for the scope of the project due to its relative simplicity and ease of integration with the existing compiler pipeline.

3.1.4 Type Error Handling

The type checker implements error handling through an exception system which preserve source location information. When the type checker encounters an invalid operation it collects the `TypeExceptions` in a list. These exceptions are then serialised to JSON and printed to standard output (stdout), where the language server reads and convert them into editor diagnostics. More about how the language server works can be found in Section 3.2.

Error Source Code Positions

ANTLR4 provides the ability to retrieve line and column information in the parse tree, corresponding to its position in the source code. When traversing the parse tree to create the AST, the position of each node in the parse tree is appended to the corresponding AST node. This position is saved in a `Pos` object within the AST node, which contains two fields, `line` and `column`. This allows the type checker to know precisely where each expression or statement originated in the original source file when traversing the AST.

```
public record Pos(int line, int column) {}
```

The TypeException Class

The `TypeException` class provides three constructors:

- `TypeException(String message)`
  Used for errors that cannot be attributed to a specific location
- `TypeException(String message, Pos pos)`
  Used when the error has a known source location
- `TypeException(String message, String expected, String actual, Pos pos)`
  Specifically designed for type mismatches. It captures both the expected type and the actual type found.

Furthermore, to ensure user-friendly error messages, the overridden method `getMessage()` was implemented to produce a structured output.

```
@Override
public String getMessage() {
    if (pos == null) {
        return "Type Error: " + errorMessage;
    }

    StringBuilder sb = new StringBuilder();
    sb.append("Type Error: "). append((errorMessage))
       ;

    if (expected != null && actual != null) {
        sb.append("\nExpected: ").append(expected);
        sb.append("\nActual: ").append(actual);
    }
    sb.append("\nAt line ").append(pos.line).append("
       , column ").append(pos.column);
    return sb.toString();
}
```

If the error does not have position information, then it simply returns the error message prefixed with `"Type Error:"`. Otherwise, it returns a formatted string that contains the error message, followed by the expected and actual types when applicable, as well as the exact line and column numbers where the error occurred.

3.1.5 Code Generation

In order to run the program the type annotated AST must first be translated into an existing programming language. This is known as an intermediate language, and for Fika we chose LLVM IR as the intermediate language.

There were two main reasons why we chose LLVM IR. Firstly, when compiling LLVM IR, it is possible to generate, and in turn, view the assembly code. Since one of our main objectives (A1) is to be able to show the user their source code translated into assembly, this was a clear advantage.

Secondly, LLVM IR is a low-level language, thus allowing the assembly to map more directly to the original source code (see Figure 3.5). A higher-level intermediate would introduce an additional layer of abstraction between the source and the assembly, making the relationship between the two harder to observe and understand.

```
int func add(a, b) {
    a = a + b
    return a
}
```

Fika Source Code	LLVM IR
`func add(a, b) {`	`define i32 @add(i32 %a, i32 %b) {` entry: %a.addr = alloca i32 store i32 %a, i32* %a.addr %b.addr = alloca i32 store i32 %b, i32* %b.addr
`a = a + b`	`%r0 = load i32, i32* %a.addr` %r1 = load i32, i32* %b.addr %r2 = add i32 %r0, %r1 store i32 %r2, i32* %a.addr
`return a`	`%r3 = load i32, i32* %a.addr` ret i32 %r3
`}`	`}`

Figure 3.5: Mapping from Fika source code to LLVM IR.

Another benefit of choosing LLVM IR is that when compiling to assembly, it allows us to choose how optimised the assembly should be. Although optimised assembly produces faster code, the optimisation may remove variables, reorder instructions or modify the assembly in other ways that make it more difficult to interpret. To facilitate the learning of assembly, we chose to show the unoptimised code by default. However, in case the user wishes to experiment with different levels of optimisation, we also added the possibility for them to do so.

The process of code generation follows the structure of the AST in a recursive manner, just as the type checker does. As each parent node in the AST is defined in terms of its child nodes, a recursive generator is a natural fit. This approach ensures that dependencies are resolved bottom-up, as child expressions are generated before their parent nodes.

During the code generation traversal, each AST node in the program is processed, and the LLVM IR instructions corresponding to the current AST node are appended to a string builder. When the entire AST has been traversed, the generated code is saved into a `.ll` file before being processed by the LLVM static compiler into a `.s` file, containing the assembly code. Finally, the `.s` file is compiled into machine code. Below, Figure 3.6 shows an example of the code generation process.

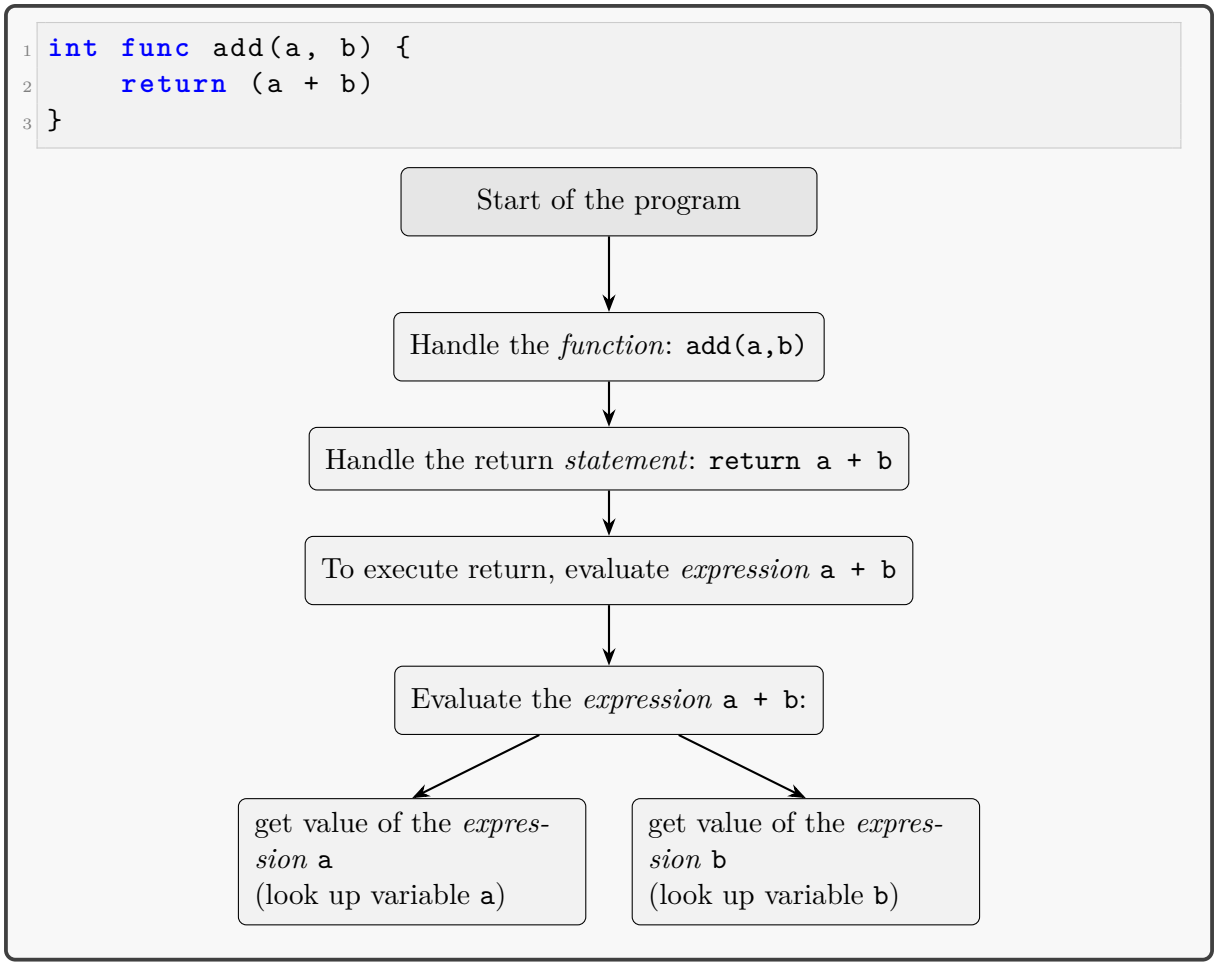

Figure 3.6: Example illustrating the code generation process of a simple function.

3.2 Visual Studio Code Extensions

The language tools we have developed during this project fall under two categories: editor specific client features and language server capabilities.

As established in Design Decision 6, language support for Fika is focused on the VS Code IDE. The client features are implemented specifically for VS Code using the VS Code Extension Application Programming Interface (API) and includes syntax highlighting, language configurations, and assembly visualisation.

The language server capabilities are implemented through a language server following the Language Server Protocol (LSP). These features include; diagnostics (showing type and syntax error messages in client), visualising or inserting inferred types and hover information.

3.2.1 TypeScript Language Server

The language server is implemented in TypeScript, building upon Microsoft's LSP sample server [24]. It consists of a language server and a language client that communicate using the Microsoft `vscode-languageserver` and `vscode-languageclient` libraries.

The client and server communicate using `messages`, of which there are three types: `RequestMessage`, `ResponseMessage` and `NotificationMessage` [25]. As the name suggests, a `ResponseMessage` is sent in response to another message, where a `RequestMessage` always expects a response, but for a `NotificationMessage` it is optional. The request and notification messages contain a string detailing what method the recipient should run as well as a field for what parameters to provide the method. The response then return the result of the method back to the request-sender. Even if the requested method does not produce any result values, a request must still receive a response. A typical chain of events for client-server request and response is shown in Figure 3.7.

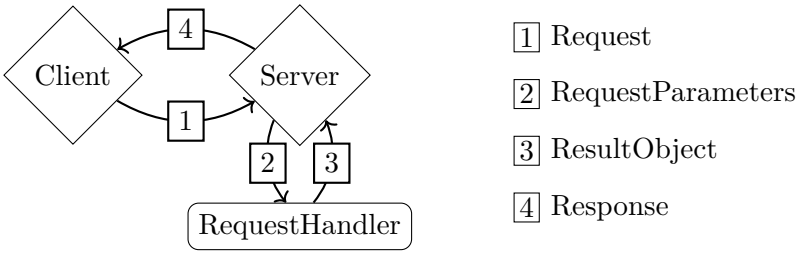

Figure 3.7: Diagram of the client-server communication for handling requests.
(1) The client sends the server a request.
(2) The language server runs it's handler function and providing the relevant parameters.
(3) Once finished, the handler returns a response object.
(4) The language server returns the results to the client.

3.2.2 Syntax Highlighting

As established in Decision 10, the syntax highlighting is implemented using the VS Code extension API, where the highlighting is handled through a `TextMate` language grammar file [19]. The grammar file contains a JSON object with root-level key/value pairs, including a unique name for the grammar (`scopeName`), an array of language rules used to parse the document (`patterns`) and a dictionary of language rules (`repository`) [26]. Details of how a TextMate language grammar and the language rules are built can be found on the documentation-manual [26], and a brief description of some of the possible fields for a language rule are listed below:

- `name`: A name tag/label used to identify and categorise the tokens.
- `match`: A regular expression (regex) pattern describing the token to be highlighted.
- `begin/end`: Can be used instead of `match` but the pattern is split into two sections describing the beginning and the end of a token. Optionally combined with `patterns` to select what rules are allowed to occur in between.
- `captures`: assigns names for regex-groups inside the `match`. There also exists equivalent `begincaptures` and `endcaptures`.

For example, in `Fika`, the syntax highlighting for function declaration is defined as the following:

```
"FunctionDeclaration": {
  "name": "meta.statement.definition.function",
  "begin": "(\\b[a-zA-Z_][a-zA-Z0-9]*\\b)? *(\\bfunc\\b)
     *(\\b[a-zA-Z_][a-zA-Z0-9]*\\b) *\\(",
  "end": "\\)",
  "patterns": [
    {"include": "#ArgumentDeclaration"},
    {"include": "#ArgumentDeclarationArray"}
  ],
  "beginCaptures": {
    "1": {"name": "entity.name.type"},
    "2": {"name": "entity.name.tag"},
    "3": {"name": "entity.name.function.function-id"}
  }
}
```

with Figure 3.8 showing an example of the resulting highlighting when using the `Catppuccin Mocha`-theme in VS Code

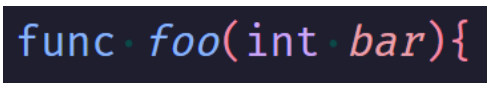

Figure 3.8: Syntax highlighting for function declaration in Fika.

3.2.3 Language Configurations

As established in Design Decision 11, we decided to include a few language configurations as they were relatively simple to implement, and provide meaningful improvements to the editing experience. These configurations are defined within a JSON file, where each feature corresponds to a key whose values contain a regex pattern describing the symbol(s)/tokens that should be associated with the feature.

For instance, defining `lineComment` and `blockComment` enables their respective keyboard shortcuts for comment toggling. The JSON file therefore contains the comment keyword and the comment tokens, such as `//`, which are then interpreted by VS Code to provide the corresponding functionality.

```
"comments": {
    "lineComment": "//",
    "blockComment": ["/*", "*/"]
}
```

```
"brackets": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"]
]
```

```
"autoClosingPairs": [
    { "open": "{",  "close": "}" },
    { "open": "[",  "close": "]" },
    { "open": "(",  "close": ")" },
    { "open": "\"", "close": "\"",  "notIn": ["string"]
       },
    { "open": "/*", "close": " */", "notIn": ["string", "
       comment"] }
```

3.2.4 Hover

As mentioned in Design Decision 8, `hover`-capabilities are used to display documentation. When the server is declared as a `hoverProvider`, the client will send a `HoverRequest` to the server whenever the user places their cursor above any character, causing the server to run the hover handler. The hover handler is a function that implements the `ServerRequestHandler` interface, which is a part of Microsoft's `vscode-languageserver` library. The handler function receives hover parameters and returns either a hover object or `null`.

In order to display information about function and variable we developed an algorithm to find and gather relevant text in the source code.

The first step of the algorithm used for finding hover information is to ensure that the hovered symbol: is not inside a comment, is not inside a string, and is a character which is allowed inside variable or function identifiers. The valid characters include alphabetic (`A-z`) and numeric characters (`0-9`) as well as underscore (`_`).

The second step involves finding the hovered word's range (position of first and final characters) inside the document and ensuring the hovered word is not any of the excluded words: numeric literals, type keywords, or any of the language keywords (e.g. `func`, `if`, `do`).

Now knowing the word to search for, the entire document is scanned for the word's first occurrence whose position is recorded. Using both positions, the region in between is analysed to ensure both occurrences are within the same scope. If this is the case, the entire line where the first occurrence was found is returned as a description for the hover information. Otherwise, this process is repeated but with the search area reduced to exclude the first occurrence.

The text description and the range of the hovered word is packaged into a `Hover`-object and sent to the client to be displayed.

The algorithm for retrieving hover information is displayed as a flowchart in Figure 3.9.

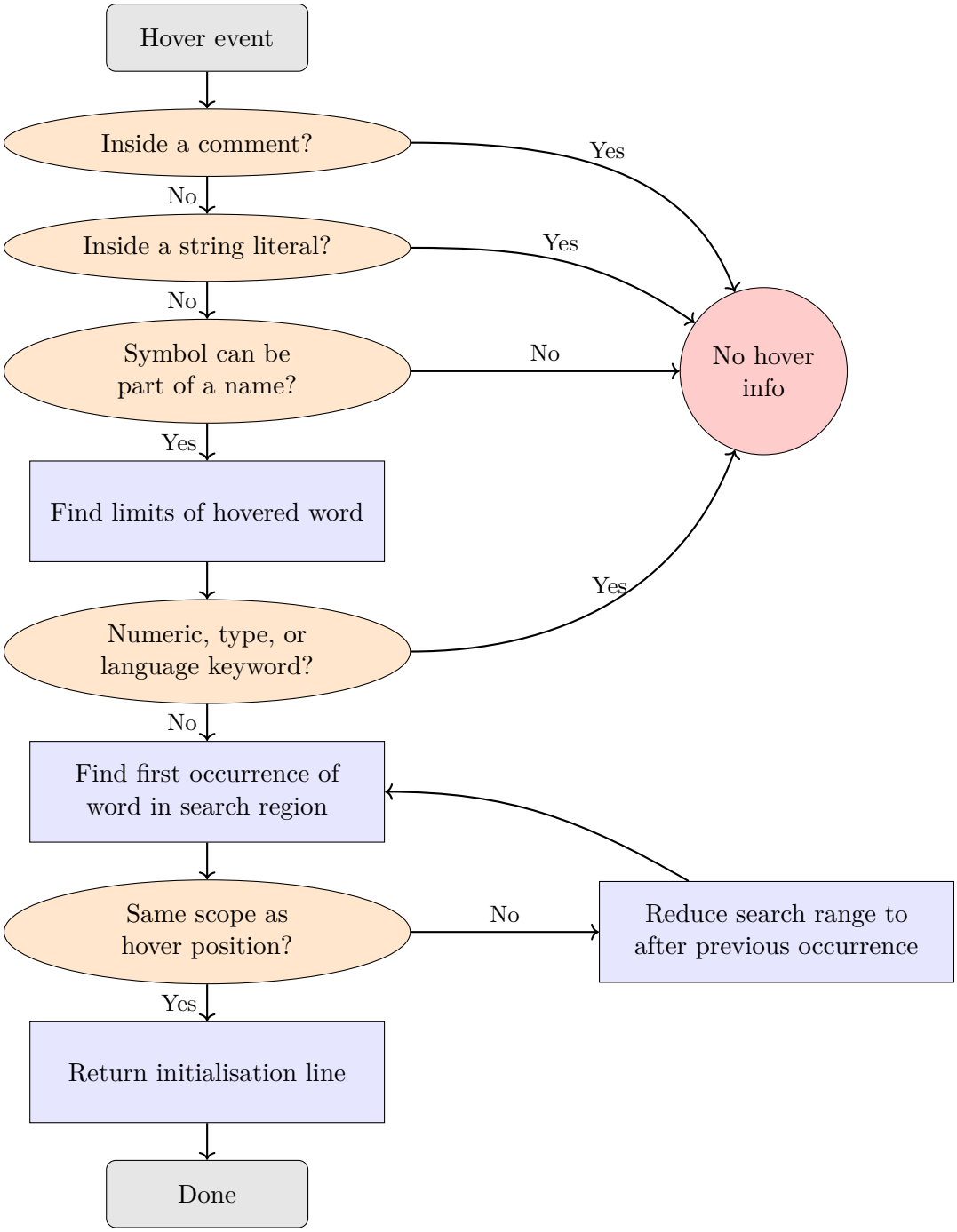

Figure 3.9: Process for retrieving hover information.

3.2.5 Error and Warning Diagnostics

In line with Design Decision 9, the language server communicates with the editor through the LSP, which provides real-time diagnostics as the user types. A diagnostic is defined as a structure that contains all the information needed to display an error or warning in the editor [5]. The display includes the range of text where the error or warning occurred, a severity level, and a descriptive message. The protocol also supports optional fields such as a source label, error code, and related information which points to other relevant locations in the code [5].

For example, if a user makes a type error such as `int x = "hello"`, the language server might produce the following diagnostic:

```
{
    "range": {
        "start": {"line": 1, "character": 8},
        "end": {"line": 1, "character": 15}
    },
    "severity": 1,
    "message": "Cannot assign string literal to variable of type int"
}
```

When the language server is initialised with `diagnosticProvider` capabilities, it will scan the document for diagnostics. This is achieved using a diagnostics-handler function that validates the source code document.

Error Handling

The type checker detects errors, but the language server is responsible for communicating these errors to the editor through diagnostics. This distinction is important, because the latter is what the user actually sees when typing in the language.

Error handling is implemented in two distinct phases: syntax error detection followed by type checking. This approach ensures that type checking only runs on syntactically correct code, which prevents confusing error messages that would otherwise arise from invalid syntax.

Syntax Error Detection. When gathering diagnostics, the server feeds the source code through a parser generated by ANTLR4, which is also responsible for generating the lexer (see Section 3.1.1). The parser then generates a parse tree while also collecting any syntax error it encounters. Each error includes the line number, column position, and a descriptive message provided by ANTLR4.

To capture these errors, a custom syntax error collector implementing ANTLR4's error listener interface was created. It stores each error internally and makes them available once parsing is complete.

The server then wraps each syntax error in an LSP diagnostic object, which includes the error severity, the exact range of text where the error occurred, a source label that identifies the error as coming from the syntax checker, and an error message. These diagnostics are then collected inside a `validateTextDocument` function. This function is triggered whenever the client editor requests document diagnostics, which activates the diagnostics event handler provided by the LSP library. The editor then renders the errors as red squiggly underlines beneath the offending code.

Type Error Detection. Type error detection follows a different architecture due to the type checker being implemented in Java whilst the language server runs in TypeScript. Therefore, to bridge these two environments, a Java bridge server was created for LSP integration.

The `validateDocument` function first performs syntax checking, as previously described. If no syntax errors are found, it then calls the `checkTypes` function from the custom `TypeCheckerBridge` module. This function spawns the Java type checker as a child process, passes the source code as a command-line argument, and captures the JSON output from standard output. The JSON output contains a list of the type errors captured by the type checker, described in Section 3.1.4.

When the Java process returns type errors as a JSON array, the server converts each JSON error object into an LSP diagnostic. Each diagnostic includes the error message, severity, source label, and calculated range. The server then sends all these diagnostics to the editor together, which allows multiple type errors to be displayed at once.

If the Java process fails to start, returns invalid JSON, or exits with a non-zero code, the server handles the failure by sending a fallback diagnostic indicating that type checking is unavailable. Syntax checking continues to function normally in this scenario, ensuring that the server never crashes and always provides at least basic feedback to the developer.

Warnings and Information

To be able to detect warnings, a simple diagnostics function was implemented using regular expression patterns. These patterns describe some predefined cases that the server detect and label with the relevant severity before being sent to the client. The three regular expressions are listed below:

```
const notRecVarName: RegExp = /\b(if|else|while|do|true|
   false|return|and|or)(?= *\=)/g
const whileTrue: RegExp = /\bwhile *(true|\(true\))/g
const todo: RegExp = /\bTODO.*/g
```

Where `notRecVarName` and `whileTrue` are labelled as warnings (severity 2), displayed in the editor as orange squiggly lines, and `todo` as information (severity 3), displayed with blue squiggly lines.

3.2.6 Inference Insertions

As concluded in Design Decision 3, inferred types should always remain visible to the user. To achieve this, the language server implements two different display modes. The first mode is inlay hints, shadowed text rendered directly in the editor at inferred variable declarations, showing the resolved type without modifying the source file. For example, a declaration such as `x = 3` would display `int x = 3` visually, with the `int` annotation rendered as non-editable hint text.

The second mode performs an explicit code insertion, writing the inferred type directly into the source file. The user can configure which mode is active according to their preference. Together these features ensure that the inferred types are visible to the user, and the user always has a clear indication of how the compiler interpreted their code. Figure 3.10 illustrates the difference between the two modes.

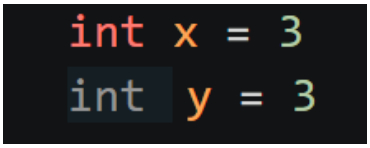

Figure 3.10: Difference between direct source-code injection and inlay hint.

However, one problem with modifying the source code directly is the possibility for users changing their minds later about an inferred variable. A variable they originally intended to be an `int`, but later wanted to be a `double` cannot directly be changed in the code due to the back-end insisting that the variable is an `int`. To resolve this issue, when a value or a type of a variable is changed, we implemented a pop-up that asks if the user wants to change the type. The user is offered three options: ignore the change, accept it, or accept with cascade. The cascade option propagates the type change transitively through all dependent variables. This makes code like `x = 3, y = x` change the type of y if the type of `x` is changed.

Type inference logic is primarily handled by the type checker. The type checker holds a list of `InferenceSuggestions`, which is a custom Java object that contains all necessary information required to display the the inferred type through the language server. As shown below, this information includes the name of the variable which type has been inferred, the type it has been inferred to, and the position of the variable in the source code, which indicates where the type annotation should be appended.

```
public record InferenceSuggestion(
        String name,
        String inferredType,
        int line,
        int column,
        int endLine,
        int endColumn
        )
{}
```

The list of InferenceSuggestions will be populated during the second pass of the AST, explained in Section 3.1.3. At this point, some variable nodes may still be of type `TUnkown`, but the `context` will hold a mapping from their name to their type. Thus, an InferenceSuggestion can be created and appended to the list of suggestions stored in the Type Checker class. ´´ Similar to InferenceSuggestions, a list of TypeReplacementSuggestions is also populated during the type checker AST traversals. Whenever an incompatible type error is detected in a variable initialisation node, a TypeReplacementSuggestion object is created and appended to the list of replacement suggestions, with both the current type and the type inferred from the expression. As an example, `int x = true` would create an object with `currentType int` and `inferredType true`. The pop-up displayed would then ask if the user wants to change the type of the variable.

After the two passes are done in the type checker, the InferenceSuggestion- and TypeReplacementSuggestion lists will be populated. Both are bundled together into a JSON object, and this JSON object is then sent via standard output (stdout) to the language server, where it is read and processed. The server iterates through the JSON object,

handling each case differently.

Visualising Inferred Types Once received by the language server, the inference suggestions are surfaced to IDE either through the LSP's inlay hint capability, or through direct source code insertions depending on the current extension settings. If inlay hints are enabled, the language server advertises inlay hint support to the IDE during initialisation, after which it knows to requests hints from the server. The IDE then requests hints whenever the document is in view, and the server triggers a refresh after each analysis completes, prompting the editor to re-request and re-render the updated hints. The server then responds to the request with each suggestion mapped to a position and a type label. The editor then renders the suggestions as shadowed, non-editable text directly in the editor at the inferred variable declarations, and the hints are refreshed automatically as the user continues to edit the document.

If source code insertion is enabled instead, the suggestions are applied as real text edits directly to the source file. The user has two options, either to apply these edits on every document change, or on save. Which mode is enabled is also decided in the extension settings. If the `onChange` setting is enabled, the server applies the edit immediately after each analysis completes. If the `onSave` setting is enabled instead, the server returns the edits through the LSP's `willSaveWaitUntil` hook, which allows edits to be injected into the document just before it is written to disk.

Applying Type Replacements. Type replacements are tied to source code insertion modes, enabled automatically alongside it, and disabled when inlay hints are used. When the server receives and extracts the replacement suggestions, if replacements are enabled, it uses LSP's window message capability to request the code editor to display a modal pop-up for any initialisation mismatches. When the user clicks a button on the pop-up, the server receives the response and requests the IDE to either to modify the source code or ignore. Since these replacements edit the source code, another Java analysis will run. This repeats the whole process, and thus causing the server to re-analyse and show pop-ups for dependent variables until no more type mismatches are found.

3.2.7 Assembly Extension

As established in Design Decision 4, the assembly visualiser should annotate generated assembly with human-readable explanations and support interactive source-to-assembly mapping. The assembly extension realises both of these goals as part of the VS Code extension. When running the extension, it compiles the source code and displays the corresponding assembly. It is shown side by side as a file, to easily see how the code maps to assembly instructions.

To be able to start the visualisation, the command `Show Assembly` is run from the command palette, with a `Fika` source file open in the editor. This triggers the compilation pipeline and opens the assembly in a virtual read only document. The `TextDocumentContentProvider` API [27] is used to open this document. Rather than writing the assembly output to disk, the extension registers a custom URI scheme `(asm-preview://)` and provides the content dynamically when VS Code requests it, allowing the assembly to be displayed as a read-only document without creating an actual file.

Assembly pipeline

The pipeline starts with a `.fika` source file, which is passed to the Java backend packaged as a JAR file and bundled with the extension. The compiler processes the source code through the compilation stages as described in Section 3, emitting LLVM IR into a `.ll` file. The `.ll` file is then passed to llc (LLVM Static Compiler), a tool that compiles LLVM IR to native assembly for a specific target architecture and optimisation level.

$$.fika \rightarrow \text{java -jar} \rightarrow .ll \rightarrow llc \rightarrow .s$$

Compilation options

The extension supports different architectures and optimisation levels. Since different CPU architectures produce different assembly, letting users switch between them allows for more variety and the ability to learn different conventions. Configurable optimisation levels allow users to observe how the compiler generates different assembly for the same source code, and to see what the compiler does to improve performance.

The extension provides four architectures, x86-64, x86, AArch64 (ARM64) and AArch32 (ARM32). Including x86-64 (also known as AMD64 or Intel 64) and AArch64 covers the majority of users, as x86-64 is used in modern Windows and Linux computers, while AArch64 is used in Macs with Apple Silicon. To illustrate how the same source code compiles differently across 32-bit and 64-bit variants of an architecture family, both x86 and ARM32 are included

The LLVM static compiler has four flags corresponding to different optimisation levels [28]. Our extension provides options for using the flags during the visualisation, which allows users to observe how the compiler transforms code at different levels and to compare the resulting assembly.

- O0 – no optimisation, disable as many optimisations as possible
- O1 – basic optimisation, optimise quickly without destroying debuggability
- O2 – more optimisation, optimise for fast execution without significant compile time or code size increase
- O3 – max optimisation, optimise for maximum performance

A challenge with supporting multiple architectures is that each architecture has different assembly conventions. The prologue and epilogue instructions, which handle stack setup and teardown, differ between architectures. For example, on x86-64 the stack is set up with `subq \$X, \%rsp`, while on ARM64 it uses `stp x29, x30`. The raw output from llc also differs between architectures, meaning the filtering of directives, labels and comments had to work correctly across all four. When implementing this, it led to bugs where instructions were incorrectly mapped to source lines or filtered. These bugs were resolved by introducing architecture specific profiles, where each architecture defines its own set of filtering rules. It made it possible to handle the differences between architectures in a structured way, rather than relying on a single set of rules that would fail across targets.

Assembly display

The raw output from llc includes directives, debug information, comments and local labels that are not actual assembly instructions. These are filtered out by removing lines starting

with `.` (directives), `;` or `@` (comments), `l\_` (local symbols) and lines that start with L that is not a label. The filtered content is then passed to a local class that fires a change event, notifying VS Code that the document has changed, which updates the displayed content.

A custom syntax highlighting was also created for the assembly preview, using the same implementation method as syntax highlighting described in Section 3.2.2. Each instruction line is padded to the length of the longest line so that the hints appear aligned in a column. The padding is implemented using a built in Typescript method "padEnd", which pads a string with spaces to the right until it reaches a given length.

Source and assembly highlighting

To realise the interactive mapping described in Design Decision 4, we have implemented the ability to click on a line in the source code which in turn highlights the corresponding assembly instructions and vice versa.

The line mapping is built using debug metadata, which is additional information the compiler embeds in the LLVM IR alongside the generated code. The metadata is not executed but allows tools to map instructions back to the original source code. When compiling to LLVM IR, the compiler annotates every instruction with `!dbg !N`, where N references a `!DILocation` entry that stores the original source line. In the example below, **!dbg !3** is the debug metadata appended to an LLVM IR instruction generated by the code generator.

```
%register2 = icmp slt i32 3, 4, !dbg !3
```

At the end of .ll file, the compiler emits the debug metadata definitions. An example of this is shown below:

```
!1 = !DIFile(filename: "good-5.fika", directory: ".")
!2 = distinct !DISubprogram(name: "main", scope: !1, file
   : !1)
!3 = !DILocation(line: 1, scope: !2)
```

Here, `!DIFile` defines the source file the code originated from, `!DISubprogram` defines a function within that file, in this case `main`, and references `!DIFile`, linking the function to its file. `!DILocation` stores the source line number of a specific instruction and links its back to the function by referencing `!DISubprogram`, which in turn links to `!DIFile`. This chain allows the visualiser to trace any instructions back to its exact location in the original source file.

This metadata is then used by `llc` when compiling to assembly. For each `!DILocation`, `llc` translates the metadata into `.loc` directives in the `.s` assembly file, for example `.loc 1 3 0` meaning file 1, source line 3. These directives appear before the assembly instructions they belong to, preserving the source mapping in the output. Two maps are then built from these directives: one mapping source lines to assembly lines, and one mapping assembly lines to source lines, which together enable the bidirectional highlighting between source and assembly.

The extension walks through the assembly file line by line, updating the current source line each time a `.loc` directive is encountered and recording the mapping for each real

instruction.

Clicking on a line is handled using the VS Code API [29]. It is an event that fires whenever the cursor position or selection changes in an editor. When the user clicks a line, this event fires and looks up the clicked line in the corresponding map, then highlights the matching lines in the other panel.

Note that at every optimisation level above O0, the highlighting may be inaccurate. This is because the compiler can remove, merge, and reorder instructions, which can result in an inaccurate mapping to the original source lines.

Instruction hints

To make it easier to understand what the assembly instructions mean, aligning with Design Decision 4, a small description explaining what each assembly instruction does is displayed next to it. There is also an extended description including more information about the instruction, and both the short and the extended descriptions are hard-coded in a separate file, using a record that maps each operation to its different descriptions depending on the operands. Operands are the value or register that an instruction operates on.

The short descriptions are implemented using VS Code inlay hints, which are small text labels displayed inline in the editor without being part of the actual file. The extension loops through the assembly file and for each instruction, check if it exists in the description file, then use the corresponding description. The extended description functionality was first implemented to be displayed on hover, using VS Code tooltips, which are small pop-up boxes that appear when the user hovers over something. After the user evaluation, the hover solution proved to be problematic. Users accidentally trigged tooltips, and when these tooltips were trigged they were difficult to dismiss. To solve this, the tooltip functionality was replaced with a side panel provided by the VS Code API [30], which instead displays the extended description when the user clicks an instruction line.

To further help the user understand the assembly instructions, the inlay hints also display variables names in place of raw memory addresses. For example, the address `-4(%rsp)` is shown as `x` instead, if that address corresponds to the location where the value of `x` is stored. To access variable names, an extra debug metadata entry is emitted when a variable is declared in the LLVM IR. The debug metadata can look something like the example below:

```
#dbg_declare(ptr %register0, !5, ...)
!5 = !DILocalVariable(name: "hej", ...)
```

In this example, `%register0` is the memory address and `!5` is a reference to the `!DILocalVariable` entry that stores the variable name. In this way, each memory address is linked to its corresponding variable name. When displaying the inlay hints, memory addresses are replaced with their variable names if a mapping exists.

4 Results

This chapter presents the results of the project across two areas. First, the Visual Studio Code extension is presented, encompassing syntax highlighting, the language server, and the assembly visualiser. Second, the results of the user evaluations are presented, including observations from the think-aloud sessions.

The syntax and language design of Fika, covering variables, operators, control flows, functions, and arrays, are presented in Appendix A, which is the documentation for the language.

4.1 The Fika Extension

This section presents the Visual Studio Code extension developed for Fika. The extension provides multiple language tools designed to support both objective A1 and A2, described in Section 1.2. The tools include syntax highlighting, language configurations, hover information, error diagnostics, type inference visualisation through inlay hints and code insertion, and an assembly visualiser with interactive source-to-assembly mapping.

4.1.1 Syntax Highlighting

For `.fika` files, VS Code is configured to highlight the source code in accordance to the language rules. An example of the highlighting is shown in Figure 4.1, where it is illustrated that different keywords get different colours, such as types and variables. Note that different themes in VS Code may change the colours and appearance of the highlighting.

demo.fika

```
1  func demoFunction(string a, string b, int c, int d) {
2      print(a + b);
3      bool doRun = true // do is not highlighted in variable here!
4      if (c ≤ d and d < 10) {
5          do 5 {
6              c++
7              while false {c--}
8          }
9      }
10     print(c)
11 }
12
13 /* TODO:
14  the todo tag is highlighted in a different colour
15  */
16 int func fibbonacci(int n) {
17     // TODO non implemented function ...
18     return -1
19 }
```

Figure 4.1: Example of syntax highlighting from using the Fika extension.

4.1.2 Language Configurations

Amongst the language configurations, Fika supports: automatic indentation after "`{`" and de-indentation after "`}`", comment toggling with keyboard shortcuts, automatic insert of the corresponding closing symbol after the cursor when typing an opening symbol (e.g. quotation marks, ", appear in pairs and any opening parenthesis inserts the respective closing parenthesis), and marking a section of code and typing a quotation mark or opening parenthesis will insert an opening-closing pair on each side of the selected area.

4.1.3 Hover

The hover function allows for users to gather information about certain elements of the code. When an element is hovered, information is provided about variable- and function declarations. As an example, hover can find when the first declaration of a variable is made. As shown in Figure 4.2, hovering the local variable `number` inside the function shows where it was declared.

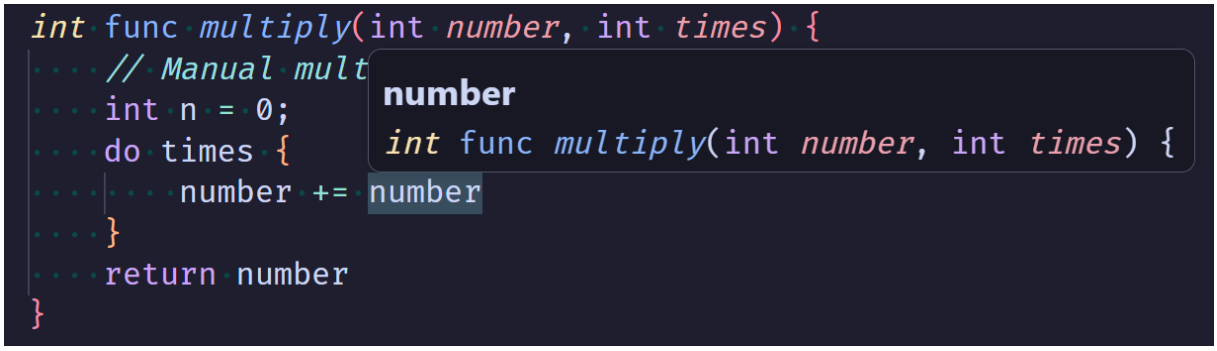

Figure 4.2: Example of hover using the Fika extension.

4.1.4 Document Diagnostics

The extension provides real-time diagnostics as the user types, surfacing errors and warnings directly in the editor without requiring the program to be run. Diagnostics are produced in two phases where syntax errors are detected first, followed by type errors, ensuring that syntactically incorrect code does not produce type errors. In addition to these, the language server produces warnings simultaneously for a number of common issues that, while not errors, may indicate unintended behaviour.

Syntax Errors. When a syntax error is encountered, the unexpected token that caused the error is underlined and a descriptive message is displayed indicating what was expected at that position.

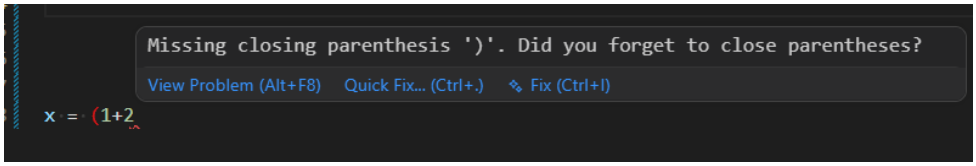

Figure 4.3: A syntax error surfaced in the editor. The offending token is underlined and a descriptive message shows a hint of what might be wrong.

Type Errors. The language employs a static type system, meaning all types are resolved at compile time. The VS Code extension enables type errors to be surfaced directly in the editor as red underline markers, allowing mistakes to be caught and corrected early.

As shown in Figure 4.4, when a type mismatch is detected, the offending expression is underlined in red and a descriptive error message is displayed indicating the expected and actual types. This immediate feedback is intended to support the user in identifying and resolving type errors without needing to run the program.

Other diagnostics. In addition to type and syntax errors, the language server also produces diagnostics for warnings and `TODO` comments. These are shown in Figure 4.5: an explicit `while-true` statement producing a warning, and the underlining of a `TODO-comment`.

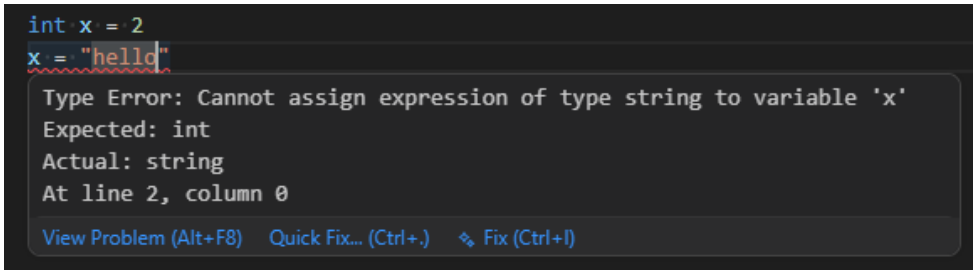

Figure 4.4: A type error surfaced in the editor. The offending expression is underlined in red and a descriptive error message indicates the expected and actual types.

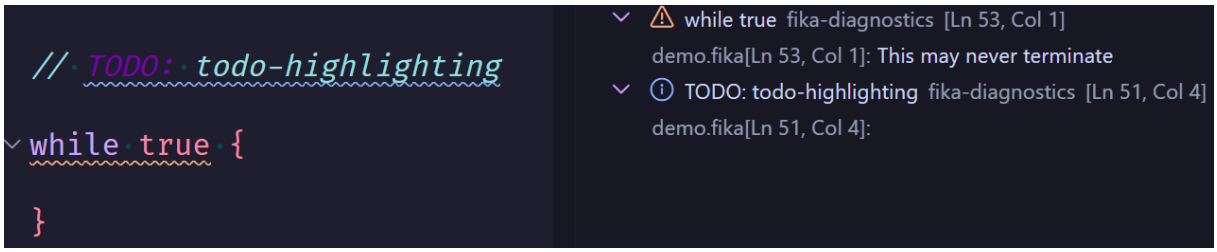

Figure 4.5: The language server diagnostics provides warnings for explicit while-true expressions and underlines TODO-comments.

4.1.5 Inline Hints and Code Insertion

Rather than requiring explicit type annotations in all cases, the language employs type inference at the point of first use. This applies to variable initialisations, declarations, function parameters, and return types. Once inferred, the type is locked, meaning subsequent uses with a different type will produce a type error. This distinguishes the inference system from generics, as functions are bound to a single concrete type rather than remaining polymorphic.

To make type inference visible and accessible to the user, the VS Code extension provides two different modes. The user can configure which mode is active in the extension settings.

Inlay hints. The first mode displays inferred types as inlay hints, which are non-editable annotations rendered directly inside the editor. For example, writing `x = 2` causes the extension to display `int` inline as shown in Figure 4.6, without modifying the source file.

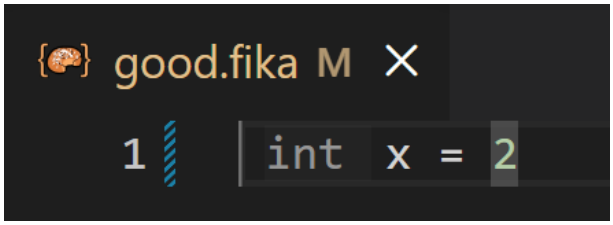

Figure 4.6: Type inference displayed as an inlay hint. Writing `x = 2` causes the extension to display the inferred type `int` inline.

The inlay hints are updated dynamically. For instance, if the user later on changes `x = 2` to `x = 2.2`, it will automatically update the type to a `double`.

The user is also able to input the hint directly into the source code by double clicking the shadowed text, as shown in Figure 4.7

Figure 4.7: Double click the inference hint to insert the type into the code.

Although the inlay hint updates dynamically, the type is still locked on first use. For instance, Figure 4.8 shows that `x = 2` will lock `x` to type `int`, meaning a subsequent usage of `x` with any other type than `int` will produce a type error.

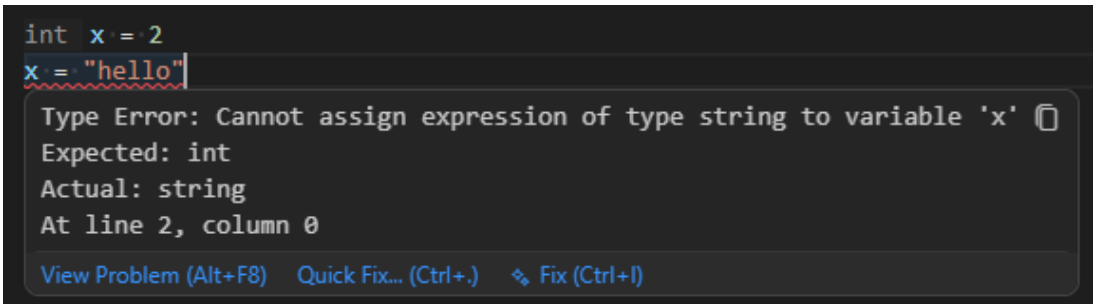

Figure 4.8: Type error when attempting to assign `x` a `string` value, after it has been assigned an `int` value previously.

Source Code Insertion. The second inference mode performs an explicit insertion, writing the inferred type directly into the source code automatically. For instance, writing `x = "hello"` will explicitly insert `string` before `x`, resulting in `string x = "hello"`.

The extension includes built-in support for changing the type of already-declared variables. When the value is changed to a different type, a pop-up, as shown in Figure 4.9, prompts the user to accept the type change, ignore it, or accept and propagate it to all dependent variables. An example of cascading type changes is illustrated in Listing 4.1. An additional extension setting allows this behaviour to trigger automatically without the prompt.

Listing 4.1: Example illustrating type propagation when accepting cascade.

```
1 int x = 3
2 int y = x //y depends on x
3
4 //When x is changed to a double with "accept and cascade".
5
6 double x = 3.3
7 double y = x //y also changes type
```

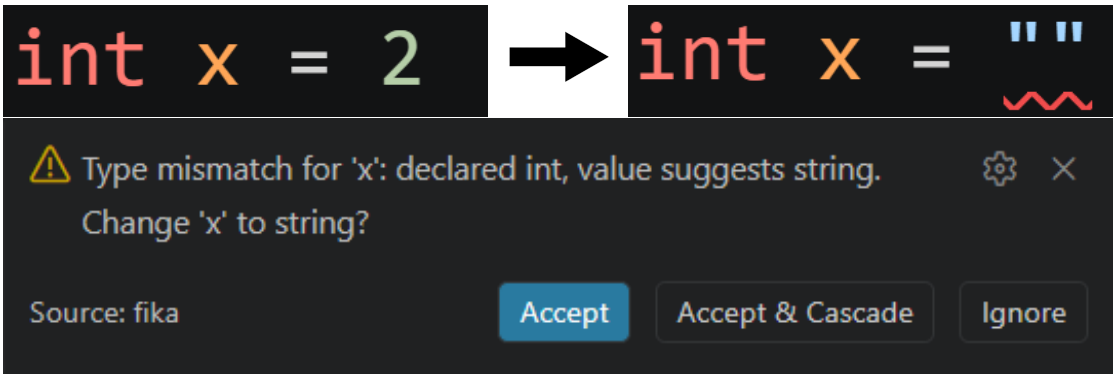

Figure 4.9: Changing the value of the expression from `int` to `string` prompts a type replacement pop-up

4.1.6 Assembly Extension

The assembly extension displays the corresponding assembly instructions for the source code. As shown in Figure 4.10, the source code and assembly are split into two panels. Clicking on a source line, in this case `x++`, highlights the corresponding assembly instructions in light blue. The instructions have syntax highlighting applied to improve readability.

The extension also provides descriptive text next to each assembly instruction. Clicking on an description, in this case `x = register`, shows an extended explanation in a side panel. Variable names are displayed in place of raw memory addresses, for example `-4(%rsp)` is shown as `x`. The extension supports four architectures (x86-64, x86, AArch64 and AArch32) and four optimisation levels (O0–O3).

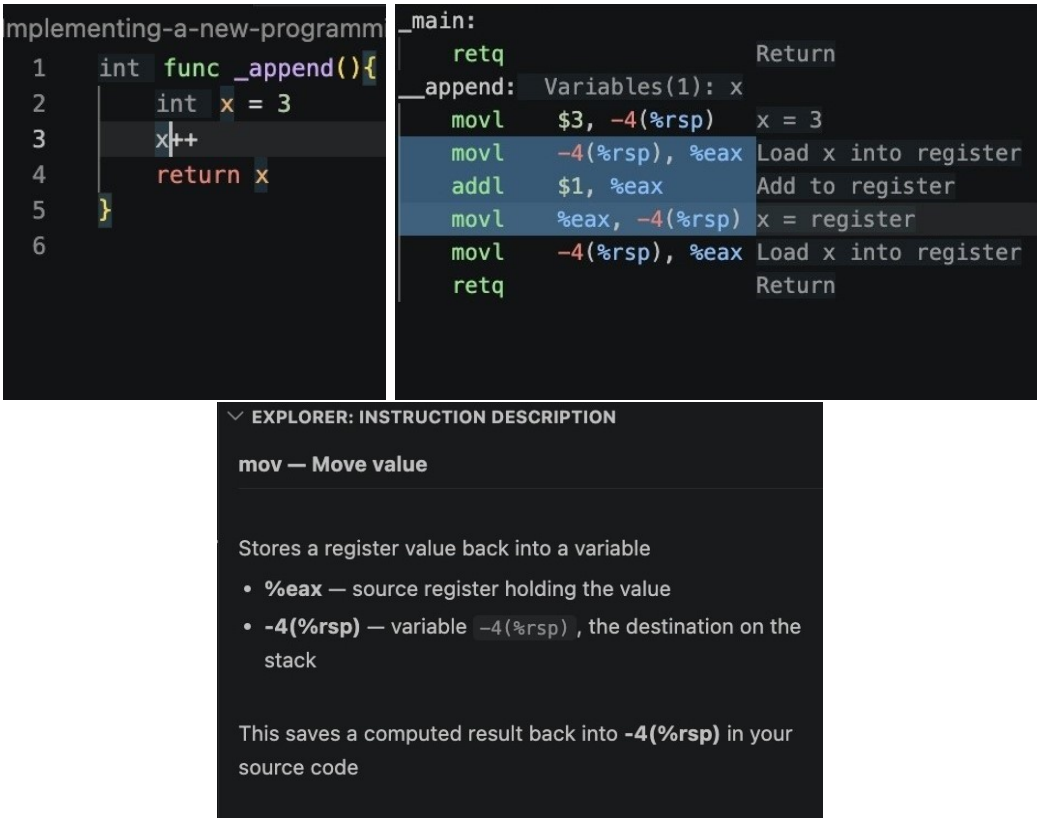

Figure 4.10: The assembly extension highlights the mapping between source code and assembly instructions, including instruction descriptions and an extended description.

4.1.7 Settings Menu

The extension offers various options for the type inference system which can be chosen in the extension settings page in VS Code. It also offers the ability to toggle whether or not the assembly visualisation should run every time the document is saved. These settings are shown in Figure 4.11.

The type replacement system has two options:

1. **Always ask**
   The user is always prompted about changing initialised types.
2. **Always auto-accept**
   Initialised variable type changes are performed automatically without prompt when differing types are detected, and propagated automatically to all dependencies.

And the type insertion has three options:

1. **None**:
   No type insertions will be performed.
2. **On save**:
   Types will be inserted as the document is saved.
3. **Immediate**:
   Types will be inserted as soon as the source file can successfully compile. This option will disable inlay-hints.

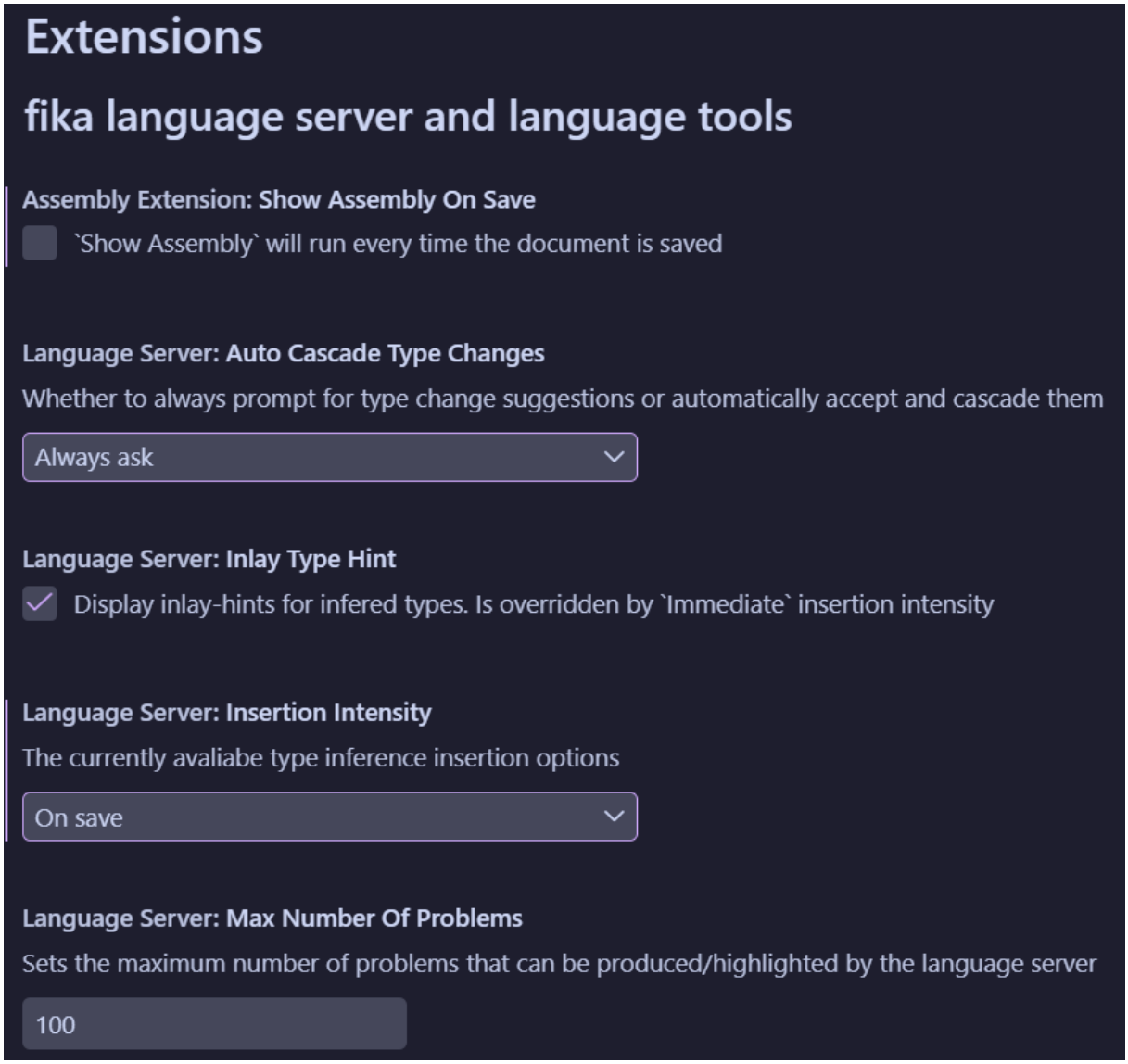

Figure 4.11: The Fika page in the VS Code settings menu.

4.2 Evaluation Results

This section presents the results of the user evaluation to assess Fika's effectiveness in achieving its two main objectives: supporting assembly language learning (A1) and providing a modern, usable high-level language (A2).

4.2.1 Participants

A total of eight participants took part in the evaluation: six students, one pilot tester, and one domain expert. Among the six students, all had completed introductory courses in computer engineering and machine-oriented programming, giving them some familiarity with low-level concepts. While their current proficiency was limited, this prior exposure allowed them to reflect on how the tool could have supported their learning when first encountering these topics.

4.2.2 Task Results

Prior to conducting the evaluations, we gave participants 5-10 minutes to review the documentation (see Appendix A) for Fika. We recommended them to focus on the following sections: types, variables, control flow, functions, type system inference and the standard library. They were allowed to ask questions about the documentation if anything was unclear. After reviewing the documentation, the participants were given tasks to perform, which took about 20 minutes to complete. The tasks and their observations are summarised below:

Task 1: Variables and Addition
The goal of this task was to observe if participants could perform a simple variable addition and print the result. While they were performing their tasks, we noted their tendencies to make syntax errors, the helpfulness of error messages, and their use of explicit type annotations.

All participants successfully completed the task. Most participants did not use explicit type annotations or semicolons. The inlay hints were noticed and appreciated. Several participants discovered the double click to insert functionality independently. One participant noted minor performance sluggishness.

Task 2: Assembly Interpretation
The goal of this task was to assess if the generated assembly from the code written in Task 1 was sufficiently commented, if any constructs were unclear, and if participants could understand registers, memory, and stack operations.

All participants could map variables to assembly locations and identified where the print operation occurred. However, some assembly instructions were unclear, such as `movl` due to its lack of comments. The hover feature was helpful once discovered, but many participants did not know it was available at first. One participant noted that comments made the assembly easier to understand. Participants responded positively to the feature that allows highlighting the source code to highlight the corresponding assembly.

Task 3: Conditional Logic (If statement)
The goal of this task was to observe if participants could write a simple `if` statement and if the syntax felt natural.

All participants successfully wrote an `if-else` statement to compare a sum against 10. The syntax felt natural, although some participants attempted Python-style syntax before correcting themselves.

Task 4: Assembly Conditionals
The goal of this task was to see if participants' could identify the if and else branches in the generated assembly, from the `if` statement written in Task 3, and if they understood conditional jumps.

Approximately half of the participants could identify both the true and false branches. A common point of confusion was that conditional jumps were inverted, which several

participants missed or misinterpreted. However, as soon as they were told they could select parts of the source code to highlight the corresponding assembly, they could easily map where the true branch was. However, this feature was not easy to find.

Task 5: Functions
The goal of this task was to review whether the participants found the function syntax intuitive or not.

All participants successfully created a function with two parameters, moved the `if` statement inside it, and called the function with new parameters. The function syntax was positively received, and one participant described it as "quite simple." Some participants relied on type inference for function parameters rather than writing explicit types. A minor issue with inlay hint offset was noted by two participants. The double-click-to-insert functionality was discovered by participants and appreciated, and one participant expressed a desire for a key-binding to insert inferred types.

Task 6: Function Assembly
The goal of this task was to see if participants could find the function body and the function call in the assembly output, and whether the supporting comments were helpful enough.

All participants could successfully identify both the function body and where the function was called. However, several suggestions for improvement were made. One participant found that `nop` and `ret` lacked comments. Another participant suggested that jump labels should be indented for better readability. Furthermore, a few participants suggested that it would be helpful to click on function names in the source code to see the entire function highlighted in the assembly view.

Task 7: Do Loop
The goal of this task was to assess if participants could successfully use the `do` loop and if the syntax was intuitive.

All participants managed to complete the task successfully. Most remembered the syntax from the documentation without needing to refer back to it. Many participants expressed enthusiasm while completing the task, and one participant explicitly stated that they preferred the `do` loop to a traditional `for` loop.

Task 8: Loop Assembly
The goal of this task was to assess if participants could identify where the loop starts and ends in the generated assembly, and where the condition is calculated.

Most participants could successfully identify the loop start and end, as well as the conditional jump. One participant found the comparison with zero confusing, but understood it after some reasoning. Some missed that the loop variable was initialised before the loop body. The code selection feature was used effectively and helped participants understand loop feature.

4.2.3 Student Evaluation

After the participant had completed the tasks, we performed an additional interview to have them reflect on their experience. The interview took about 5 minutes in total. Their responses will be summarised below.

Helpfulness for Assembly Learning. One of the questions we asked during the evaluation was how helpful Fika would be for someone with limited or no prior knowledge in assembly. The responses were overwhelmingly positive. Most participants agreed that Fika would be very useful, and several participants noted that they would not have been able to write the assembly themselves without it. One participant said that being able to write in a high-level language and then translate it to assembly was valuable. Additionally, multiple people said that the ability to click on source code and see the corresponding assembly highlighted was particularly helpful. Another participant with some assembly background described it as a good refresher, but also acknowledged that a complete beginner might initially find it confusing.

Syntax Highlighting. The second question we asked was about the helpfulness of the syntax highlighting, which all participants generally found intuitive and discreet.

Implicit Types. The third question we asked was about the implicit types. The participants appreciated that the hints were subtle and did not force one to write types explicitly. One participant explicitly said that the hints served as a useful reminder, especially when coming from a dynamically typed language such as Python. Another participant appreciated that the hints could help them if they forgot what type a variable was supposed to be.

The Do Loop. Lastly, we asked them about the do loop which is unique to our language. The opinions were generally positive. Several participants expressed that it was fun and simple, appreciating that it requires less writing compared to a traditional `for` or `while` loop. One participant noted that the loop would be helpful for beginners who do not want to manage a counter manually. However, some questioned its practical utility, and noted that a `for` loop provides more control and that it is uncommon to know exactly how many times a loop would execute at compile time.

4.2.4 Domain Expert Evaluation

Our domain expert, Professor at Chalmers University of Technology in Computer Science, participated in a separate think-aloud session that followed the same task structure. After the session, we asked similar questions to those we asked the students, but with some tweaks to suit the expert perspective. More specifically, we asked about the following areas: assembly learning support for beginners, comment quality, documentation clarity, fundamental language design issues, comparison to similar tools, and potential improvements. Furthermore, we also asked general questions about the language. The provided feedback that included multiple areas of improvement.

Task Performance. The expert managed to complete all tasks almost entirely without syntax errors.

Assembly Learning Support. The expert was positive that the language could facilitate assembly learning for beginners.

Assembly Comments. The comments were found to be sufficient in most cases, and the expert specifically appreciated that they show how assembly maps to source code. However, some were considered insufficient, for instance the `imull` instruction lacked a comment entirely. In general, the comments could be more detailed and context dependent, for example by clarifying the purpose of each instruction in the function call setup, such as which argument is being loaded into a register. String literals were also difficult to identify in the output, as the comments did not show the actual string value.

Language and Tooling. The hover functionality for extended assembly instructions felt unnatural to trigger and difficult to dismiss. Certain error messages were also occasionally confusing. They also noted that the language currently lacked a way to check the size of an array. There were no special remarks on the inlay hints.

Comparison to Similar Tools. The expert had not used a tool exactly like Fika before, but had experience looking at raw assembly output from compilers. They concluded that Fika is better in this regard, as it makes more of the underlying behaviour visible to the user and adds the ability to directly map source code to assembly code.

Suggested Future Improvements. For future work, the expert suggested adding a debugger that steps through the generated assembly side-by-side with the source code, and proposed features such as polymorphism.

5 Discussion and Future Work

This chapter reflects on the outcomes of the project and discusses the findings in relation to the two primary objectives A1 and A2, established in Section 1.2. It covers the interpretation of the evaluation results, improvements made to the language and tooling following the evaluation, limitations identified in both the user testing and the language itself, and directions for future work.

5.1 Interpretation of Evaluation Results

The purpose of this project was to design and implement a high-level programming language, Fika, together with an integrated assembly visualisation tool. The project had two primary objectives which were to support the teaching and learning of assembly language, while maintaining a language that feels modern and usable. The work was motivated by the observation that existing tools are typically designed for experienced developers rather than students, and that many educational languages sacrifices modern features for simplicity and flexibility. Fika aimed to bridge this gap, and the evaluation sessions were conducted to assess how well it succeeded in doing so.

The evaluation was conducted as a user study following the strategy described in Section 2.2.3. The think-aloud method proved effective at revealing how participants reasoned about both the language syntax and the assembly output in real time, surfacing issues that might not have appeared in a purely questionnaire-based evaluation. The follow-up interviews provided additional depth, allowing participants to reflect on their experience and clarify observations made during the tasks. Overall, the combination of think-aloud observation and structured interviews provided a rich source of qualitative insight into both the usability of the language and its effectiveness as a learning tool.

The evaluation results suggest that Fika met its two primary objectives, though with some limitations. The following paragraphs reflect on the findings in relation to each objective.

Reflection on Objective 1: Learning Assembly. Regarding the first objective, the participants were able to successfully map source code variables to assembly instructions. Additionally, they found the interactive feature that highlights the corresponding assembly code when clicking on source code to be particularly helpful and engaging. This interactively supports learning, which is the core goal of the project. However, the inverted conditional jumps caused confusion, and some descriptions were insufficient, which indicates that assembly comments could be further improved for beginners. All participants showed a positive attitude towards the assembly visualisation, which suggests Fika could be used as a supporting tool for learning.

Reflection on Objective 2: Modern Language. As for the second objective, participants found the syntax intuitive and appreciated type inference. The optional semicolons worked as intended. The `do` loop generated enthusiasm since it is an uncommon feature and has a simple syntax, although there were mixed opinions about its practical utility, as some argued a `for` loop could do the same job.

Furthermore, participants made almost no syntax errors after being given only 5-10 minutes to view the documentation, which strongly suggests our syntax is simple and aligns with the survey results that guided our design decisions. Specifically the Keyword Preference Survey (see Appendix 2.1.4) indicated strong support for symbolic comparison operators, and postfix operators, while the Modern Language Survey (see Appendix 2.1.3) confirmed that Python-inspired simplicity was preferred by the majority of respondents.

5.1.1 Language Improvements After Evaluation

During the evaluations, some unintended behaviours were observed, and certain parts could be improved by increased clarity. A majority of the unintended behaviours and their solutions are presented below:

One such behaviour was the missing support for negative numbers, which was identified during the pilot test. This was caused by a lexical error, as we had forgotten to include the possibility to write a `-` sign before numbers. Once identified, the issue was easily resolved for the subsequent evaluation sessions.

Additionally, most assembly comments were identified as helpful, but there were a few special cases where the participants found them confusing. Especially the `ret`, `mov` and `nop` comments lacked a proper description initially. This was resolved by updating these comments and verifying with our supervisor that they were precise enough.

Another common frustration during the evaluation was the hover functionality for the extended descriptions in the assembly code. While the descriptions themselves were appreciated, the hover was sometimes unintentionally triggered and covered part of the assembly. To resolve this, the hover was swapped out for a select functionality, meaning the extended comments were made visible when clicking on the short description instead of hovering it.

There were also some suggested functionalities that were not supported by the language, but that would have been appreciated. However, we only had time to implement one, namely the `size` function for retrieving the size of an array, following the suggestion of the expert tester. There was also a desire from several participants to implement key binding as an option to inserting the inferred types, which currently works only by double clicking. Due to time constraints there was not enough time to implement this functionality.

The documentation also lacked proper explanations for the `do` loop, and the exponentiation operator, which were both improved after the evaluation.

5.1.2 Limitations in User Testing

The participants who took part of the evaluation were all university students who had completed introductory courses in machine programming and computer engineering, except for one domain expert. This made the participants suitable for evaluating whether

the language and visualiser could support students with some foundational knowledge, as well as providing valuable reflections on whether the extension could have offered them support early in their assembly journey. However, it also introduced a limitation. We cannot fully determine how effective Fika would be for students with absolutely no prior knowledge of low-level concepts. A student encountering assembly for the first time, without any foundational understanding of low-level concepts, might have a different experience compared to our participants. Future work should evaluate Fika with students who have no prior exposure to assembly.

Another limitation concerns how we measured the learning objective (A1). Our evaluation captured participants' perception of Fika's helpfulness in assembly learning, but not whether they retained any assembly knowledge after using Fika. As a result, we cannot make strong claims about long-term learning of assembly. It would require more extensive research, which was outside the time limits of this project. In particular, we would have liked to test two different groups, one using Fika and one without, and compare their exam results.

5.2 Language and Tooling Limitations

The following outlines the limitations of the language and of the hover functionality.

5.2.1 Language limitations

Due to the narrow time frame for this project (18 weeks), the scope of Fika is quite limited. Ideally, we would have liked to implement a larger standard library and possibly also a generic type to allow for more advanced usages of the language. On the other hand, an overly feature-rich language could risk overwhelming the user and diverting focus from the assembly concepts that Fika is designed to teach.

5.2.2 Hover limitations

Whilst the foundations for the hover provider is there, currently no part of the algorithm differentiates between variables or functions, meaning it does not gather any user written documentation nor search for standard library function descriptions in other documents. In the future, we would like to include additional features for the hover functionality to provide more accurate information to the user.

5.3 Future Work

We believe several promising directions for future work with Fika still remain. A few examples are reflected upon below, including improvements to existing tooling, extensions to the language server and integration with additional development environments.

5.3.1 Inference Cascades

A natural extension of the current cascade system, as shown in Section 4.1.5, would be to propagate type changes through function parameters and return types. This means that for any variable, function or parameter, changing and cascading the variable that all other

depend on would propagate throughout the whole program. This would vastly increase efficiency in major refactoring of code, and would also allow any users to effortlessly test how different types affect the assembly throughout a program.

As illustrated in Listing 5.1, changing the type of a variable passed to a function could automatically update the corresponding parameter and return type of that function. This would make the cascade feature significantly more powerful, allowing type changes to propagate consistently across both variable and function boundaries.

Listing 5.1: Example illustrating potential cascade behaviour for function parameters and return types.

```
int x = 3
foo(x)
int func foo(int a){
    return a
}
// After cascading a type change on x to bool:
bool x = true
foo(x)
bool func foo(bool a){
    return a
}
```

5.3.2 Extending the Language Server

The language server protocol supports a lot more features than the ones implemented in this project. For example `code completion`, which in our first survey (Section 2.1.3) received the fourth-highest number of votes when asked about what was important for a programming language to feel modern (Figure B.2).

Another feature supported by the LSP is `code actions` which include: `quickfix` actions, several different kinds of `refactoring` actions and `source` actions. Figure 5.1 shows source actions supported by the Eclipse JDT Language Server for Java.

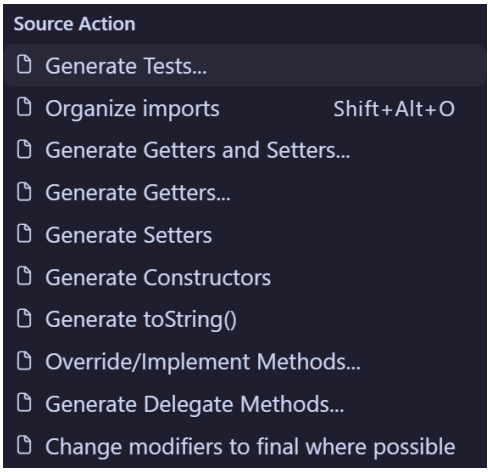

Figure 5.1: Source actions available for Java when using the Eclipse JDT Language Server.

5.3.3 Integration with other IDEs

A core principle of the LSP is that any language server adhering to the principle can communicate with any IDE that supports LSP. Therefore, it should be possible to connect our language server to any of the many available IDEs. However, since the syntax highlighting and assembly visualisation were developed specifically for VS Code, they would need to be reimplemented for other IDEs. Future work could therefore focus on integrating Fika, especially the visualisation tool, with other popular IDEs, such as JetBrains or Eclipse.

Bibliography

[1] R. A. Saputro. "Languages: From binary code to artificial intelligence. "[Online]. Available: https://rianaditro.medium.com/the-history-of-programming-languages-from-binary-code-to-artificial-intelligence-63784adbdd00. Accessed: 10/02/2026.

[2] IBM. "Fortran. "[Online]. Available: https://www.ibm.com/history/fortran. Accessed: 11/02/2026.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson, 2006.

[4] T. Mucci. "Ai isn't just making it easier to code. it makes coding more fun." IBM Think Insights, accessed 2026-03-27. [Online]. Available: https://www.ibm.com/think/insights/ai-improving-developer-experience.

[5] Microsoft, *Language server protocol specification*, Accessed: 2026-04-15, 2023. [Online]. Available: https://microsoft.github.io/language-server-protocol/.

[6] Chalmers University of Technology, *Regler för användning av ai-verktyg*, https://www.chalmers.se/utbildning/dina-studier/kandidat-och-examensarbete/regler-for-anvandning-av-ai-verktyg/, accessed May 2026, 2024.

[7] B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style*. McGraw-Hill, 1978.

[8] M. Kölling, "The greenfoot programming environment," in *ACM SIGCSE Bulletin*, vol. 42, ACM, 2010, pp. 139–143.

[9] Python Software Foundation, *The python language reference*, https://docs.python.org/3/reference/, Accessed: 2026-03-17, 2024.

[10] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, Java SE 8 Edition. Oracle, 2014.

[12] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[13] Haskell Wiki Contributors, *Type inference*, https://wiki.haskell.org/Type_inference, Haskell Wiki, 2007.

[14] M. R. Clarkson et al., *OCaml Programming: Correct + Efficient + Beautiful*. Spring, 2026, Spring 2026 Edition. Accessed: 2026-05-10. [Online]. Available: https://cs3110.github.io/textbook/chapters/interp/inference.html.

[15] Oracle, *Local variable type inference*, https://docs.oracle.com/en/java/javase/17/language/local-variable-type-inference.html, Java SE 17 Language Updates, 2021.

[16] M. Evers, *Vs code compiler explorer*, https://github.com/mattgodbolt/compiler-explorer, Visual Studio Code Extension, 2024.

[17] M. G. et. al, *Godbolt compiler explorer*, https://github.com/compiler-explorer/compiler-explorer.

[18] E. Fouh, M. Akbar, and C. A. Shaffer, "The role of visualization in computer science education," *Computers in the Schools*, vol. 29, no. 1-2, pp. 95–117, 2012. DOI: 10.1080/07380569.2012.651422.

[19] M. S. Code. "Syntax highlight guide. "[Online]. Available: http://code.visualstudio.com/api/language-extensions/syntax-highlight-guide. Accessed: 29/04/2026.

[20] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of ACM INTERCHI'93 Conference*, Amsterdam, The Netherlands, 1993, pp. 206–213.

[21] A. Dix, J. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction*, 3rd. Pearson Education Limited, 2004, ISBN: 978-0-13-046109-4.

[22] LLVM Project, *The llvm compiler infrastructure*, https://llvm.org, accessed April 2025, 2026.

[23] Oracle, *Pattern matching for switch*, https://docs.oracle.com/en/java/javase/17/language/pattern-matching-switch.html, Java SE 17 Language Updates, accessed May 2026, 2021.

[24] V. S. C. Microsoft. "Vscode-extension-samples/lsp-sample. "[Online]. Available: https://github.com/microsoft/vscode-extension-samples/tree/main/lsp-sample. Accessed: 01/02/2026.

[25] Microsoft. "Language server protocol specification - 3.17. "[Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#message. Accessed: 12/05/2026.

[26] MacroMates. "Language grammars. "[Online]. Available: https://macromates.com/manual/en/language_grammars. Accessed: 16/04/2026.

[27] Microsoft. "Textdocumentcontentprovider - vs code api." accessed May 2026. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api#TextDocumentContentProvider.

[28] LLVM Project, *Llvm::optimizationlevel class reference*, https://llvm.org/doxygen/classllvm_1_1OptimizationLevel.html, Accessed: 2026-05-16, 2026.

[29] Microsoft. "Ondidchangetexteditorselection - vs code api." accessed May 2026. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api#window.onDidChangeTextEditorSelection.

[30] Microsoft. "Webviewviewprovider - vs code api." accessed May 2026. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api#WebviewViewProvider.

A Appendix: Documentation

A.1 Introduction

Fika is a modern programming language designed to make low-level concepts more accessible through a simple and readable syntax. It combines a clean, minimal design with a strong type system and interactive tooling such as live compilation and type inference feedback.

The language is particularly focused on education, allowing users to understand how high-level code maps to lower-level representations.

A.2 Basic Code Example

The following example demonstrates the basic syntax of Fika, including variable declaration, function definition, and the use of a `do` loop to repeat an action multiple times.

```
x = 5

func printThreeTimes(value) {
  do 3 {
      print(value)
  }
}

printThreeTimes(x + 10)
```

Output:

```
15
15
15
```

A.3 Types

Fika has four primitive types:

- `int`
- `double`
- `bool`
- `string`

Dynamic arrays are also supported, and can be declared putting a primitive type inside of squared brackets. For example, `[int]` represents an array of integers, and `[string]` represents an array of strings. The size of the array does not need to be specified when declaring the array, as Fika will automatically manage the memory for you.

A.4 Basic Syntax

In the following section, the basic syntax of Fika is explained, including variable declarations, control flow constructs, function definitions and calls, array usage, and comments. Additionally, the powerful type inference system of Fika is discussed in more detail.

A.4.1 Variables

Variables in Fika are declared using a simple assignment syntax. The type of the variable is inferred from the assigned value, but can also be explicitly specified if desired.

For instance, both of the following examples are valid variable initialisations of an integer variable named x:

```
x = 2
int x = 2
```

Semi-colons are optional, and can be used to separate multiple statements on the same line if desired. Example:

```
x = 2; y = 3;
```

Declaration works in a similar manner, explicit type declaration is optional, but can be used for clarity or to specify a different type than the inferred one.

Example of variable declarations (both are valid):

```
x;
int x;
```

A.4.2 Arithmetic Operations

Fika supports a full range of arithmetic operators for numeric types `int` and `double`.

Basic Arithmetic

Operators in Fika follow a well-defined precedence hierarchy, meaning that expressions involving multiple operators are evaluated in a predictable order without requiring explicit parentheses. The precedence from lowest to highest is shown below, where operators on the same level are evaluated left to right:

Precedence	Operators	Description
1 (lowest)	`=, +=, -=, *=, /=`	Assignment
2	`or`	Logical disjunction
3	`and`	Logical conjunction
4	`not`	Logical negation
5	`==, !=, <, >, <=, >=`	Comparison and equality
6	`+, -`	Addition, subtraction
7	`*, /, %`	Multiplication, division, modulo
8	`**`	Exponentiation (right associative)
9	`++, -`	Postfix increment/decrement
10 (highest)	`( exp )`	Parenthesis around an expression.

For example, `2 + 3 * 4` evaluates to `14` and not `20`, since multiplication has higher precedence than addition. However, parentheses can always be used to override the default precedence. As an example, `(2 + 3) * 4` evaluates to `20` instead of `14`, since the parentheses forces the addition to be evaluated first.

Increment and Decrement

Fika provides increment and decrement operators for increasing or decreasing a variable by 1.

Example:

```
x = 3
x++
print(x)

y = 4
y -
print(y)
```

Output:

```
4
3
```

A.4.3 Logical Operators

Fika supports the logical operators `and` (logical AND) and `or` (logical OR).

Example:

```
print(true and true)
print(true and false)
```

will output:

```
true
false
```

Fika's logical operators do not use lazy evaluation; both operands are always evaluated.

A.4.4 Control Flow

Fika supports standard control flow constructs such as `if`, `else`, `while`, and a unique `do` loop that executes a block of code a specified number of times.

If Statements

If statements are used for conditional execution of code blocks. The syntax is straightforward, with the condition enclosed in parentheses and the code block enclosed in curly braces.

Example:

```
x = 5
if (x > 0) {
    print("x is positive")
} else {
    print("x is non-positive")
}
```

will output:

```
x is positive
```

While Loops

While loops allow for repeated execution of a block of code as long as a specified condition is true. The syntax is similar to if statements, with the condition enclosed in parentheses and the code block enclosed in curly braces.

Example:

```
x = 0
while (x < 3) {
    print(x)
    x += 1
}
```

will output:

```
0
1
2
```

Do Loops

The `do` loop is a unique control flow construct in Fika that allows for executing a block of code a specified number of times. The syntax is `do <number> ...`, where `<number>` is the number of times to execute the block. The `<number>` is evaluated once when the loop is first encountered. Changing the variable inside the loop body does not affect the number of iterations.

Example:

```
n = 3

do n {
    n++
    print("Hello, Fika!")
}
```

will output:

```
Hello, Fika!
Hello, Fika!
Hello, Fika!
```

A.5 Functions

A.5.1 Function Definitions

Functions are defined by using the `func` keyword, followed by the function name, a list of comma-separated parameters in parentheses, and the function body enclosed in curly braces. The return type can be specified before the func keyword, but is optional if it can be inferred.

Example:

```
func add(a, b) {
    return a + b
}
```

This defines a function named `add` that takes two parameters `a` and `b`, and returns their sum. The types of a and b are inferred from the context in which the function is called.

The same function can also be defined with explicit types:

```
int func add(int a, int b) {
    return a + b
}
```

Functions can also be defined without a return value. In that case, no explicit return type should be specified, and the function body can simply perform actions without returning a value:

```
func printMessage(string message) {
    print(message)
}
```

A.5.2 Function Calls

Functions are called by using the function name followed by a list of comma-separated arguments in parentheses.

Example of calling the `add` function defined earlier:

```
result = add(5, 10)
print(result)
```

A.6 Arrays

The syntax for declaring and using arrays is defined by enclosing the type of the array inside square brackets, eg `[double]`. All arrays are dynamic, so it is not needed to specify the size of the array when declaring it.

Example of declaring and initialising an array of integers:

```
[int] numbers = [1, 2, 3, 4, 5]
```

Appending to an array can be done using the `append` function:

```
numbers.append(6)
```

will result in numbers being [1, 2, 3, 4, 5, 6].

Note that the same array can also be declared without an explicit type, and the type will be inferred from the assigned value:

```
numbers = [1, 2, 3, 4, 5]
```

Elements are accessed using index notation with square brackets. For example, `numbers[0]` will access the first element of the array, which is 1.

The size of the array can be accessed with the `size` function. Example:

```
numbers = [1, 2, 3, 4, 5]
print(numbers.size())
```

will output:

```
5
```

A.7 Comments

Comments in Fika are denoted by `//` for single-line comments and `/* ... */` for multi-line comments. Comments are ignored by the compiler and are used to provide explanations or annotations within the code.

```
// Single-line comment

/*
Multi-line comment
*/
```

A.8 Type System and Inference

The following section explains the type system of Fika in more detail, with a focus on its powerful type inference capabilities. The language is a statically typed language, which means that the types of variables and expressions are determined at compile time. However, Fika also has a powerful type inference system that allows for a more concise and readable code, while still maintaining the benefits of a strongly typed language.

A.8.1 Simple Inference Example

To understand how type inference works in Fika, consider the following example:

```
x
x = 5
```

In this example, we declare a variable `x` without specifying its type. When we assign the value 5 to `x`, the language automatically infers that `x` is of type `int`. This means that we can use `x` in any context where an integer is expected, and the language will treat it as an integer.

It is important to note that once a variable as been assigned a type, it cannot change type automatically. If `x` is later on assigned a value of a different type, such as a string, the language will raise a static type error inside the IDE, since `x` has already been inferred to be of type `int`. This is one of the key features of Fika's type system, which combines the benefits of static typing with the flexibility of dynamic typing.

A.8.2 Int to Double Promotion

Although the language is statically typed, it provides a lot of flexibility. For instance, ints are automatically promoted to doubles when used in an expression with a double, and the result will be inferred as a double. This allows for seamless integration of different types without the need for explicit type conversions. An example of this is:

```
x = 5
y = 2.5
result = x + y
print(result)
```

will output:

```
7.5
```

Where result will automatically be inferred as a double, since it is the result of adding an int and a double.

A.8.3 String Concatenation

Similarly to int to double promotion, when using the `+` operator with strings, the language will automatically concatenate them into a result of type `string`. For example:

```
greeting = "Hello, "
name = "Fika"
message = greeting + name
print(message)
```

will output:

```
Hello, Fika
```

When using the + operator with a string and a non-string type, the non-string type will be converted to a string before concatenation. For example:

```
number = 42
message = "The answer is: " + number
print(message)
```

will output:

```
The answer is: 42
```

with message being inferred as a `string` automatically.

A.8.4 Function Type Inference

Fika also supports inference of function return types. If a function does not have an explicitly specified return type, the language will infer the return type based on the types of the values returned by the function. For example:

```
func add(a, b) {
    return a + b
}
```

In this example, the return type of the add function is inferred based on the types of `a` and `b`. If both `a` and `b` are integers, the return type will be inferred as int. If one of them is a double, the return type will be inferred as double.

Likewise, the types of the parameters a and b can also be inferred based on the context in which the function is called. For example, if we call `add(5, 10)`, the language will infer that both `a` and `b` are of type `int`. If we call `add(5.0, 10)`, the language will infer that `a` is of type `double` and `b` is of type `int`, and the return type will be inferred as `double`.

For instance, in the following example:

```
func mixOfTypes(a, b, c) {
    return c
}

mixOfTypes(5, 2.5, "hello")
```

The types of the parameters a, b, and c will be inferred as `int`, `double`, and `string` respectively, based on the types of the arguments passed to the function. The return

type of the function will be inferred as `string`, since it returns the value of `c`, which is a `string`. The same function could also be defined with explicit types:

```
string func mixOfTypes(int a, double b, string c) {
    return c
}
```

A.9 Standard Library

The language has a limited standard library, that includes one basic function for output, `print`, which can be used to display values to the console. The `print` function can take any type of argument and will convert it to a `string` representation before outputting it.

Example:

```
print("Hello, Fika!")
print(42)
print([1, 2, 3])
print("1 + 2 = " + (1 + 2))
```

will output:

```
Hello, Fika!
42
[1, 2, 3]
1 + 2 = 3
```

B Appendix: Modern Language Survey

This appendix presents the full questionnaire used in the Modern Programming Languages Survey. The survey was conducted among university students with programming experience and consisted of nine questions, including both multiple-choice and open-ended questions.

Some of the key results are also presented at the corresponding question.

Question 1

Which imperative programming languages have you used before?

- Python
- Rust
- Java
- C#
- C++
- C
- Other

Question 2

Which of the languages did you enjoy using the most, and why?

Respondents provided free-text answers.

Question 3

Which language was the easiest to learn?

- Python
- Rust
- Java

- C#
- C++
- C
- Other

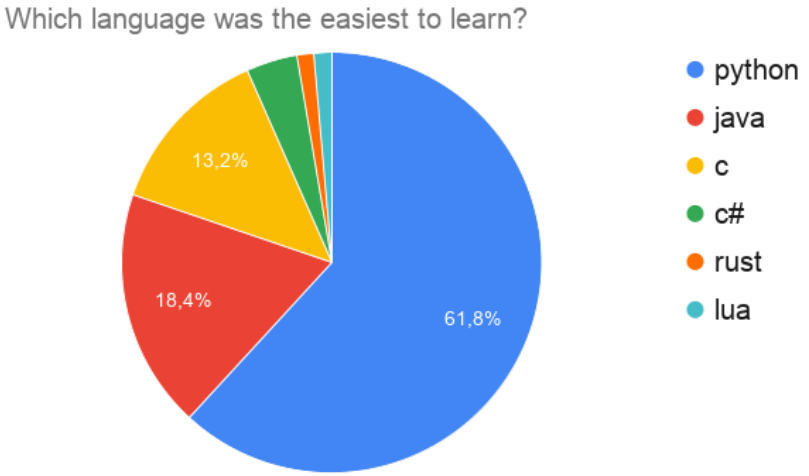

Figure B.1: Distribution of responses to the question "Which language was the easiest to learn?" (n = 76).

Question 4

Why was that language easiest to learn?

Respondents provided free-text answers.

Question 5

Which of the following tools do you feel are important for a programming language to feel modern and easy to use? Choose at most three.

- Good error messages
- Good documentation
- Syntax highlighting
- AI completion (predicts block/lines of code)
- Code completion (word prediction)
- AI-assistance (like Copilot)
- Other

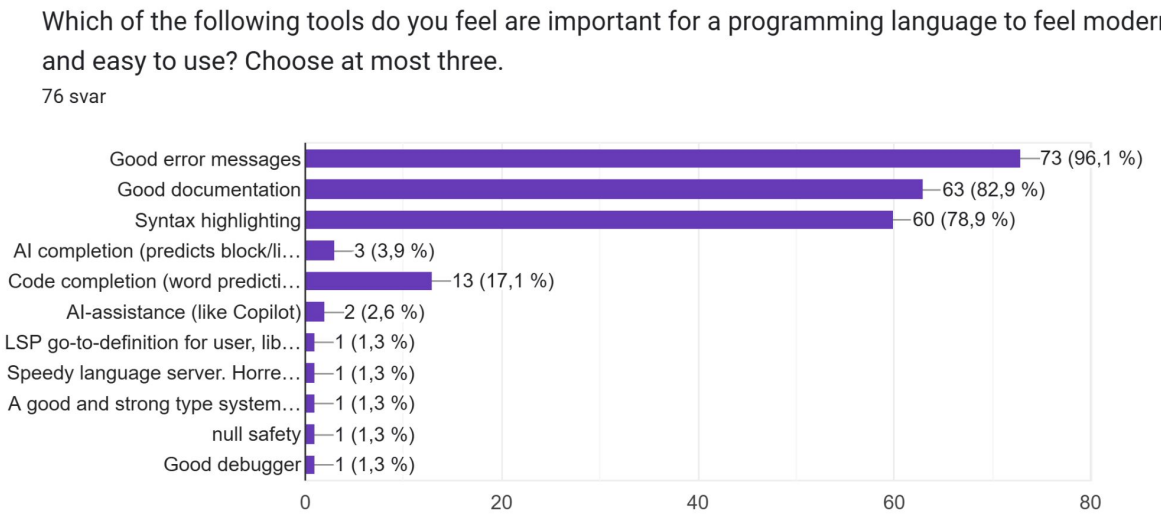

Figure B.2: Distribution of responses to the question "Which of the following tools do you feel are important for a programming language to feel modern and easy to use?" (n = 76).

Question 6

What do you appreciate most in a programming language?

- Distinct and simple syntax
- Strongly Typed
- Syntax highlighting
- Fast feedback when compiling/running
- Other

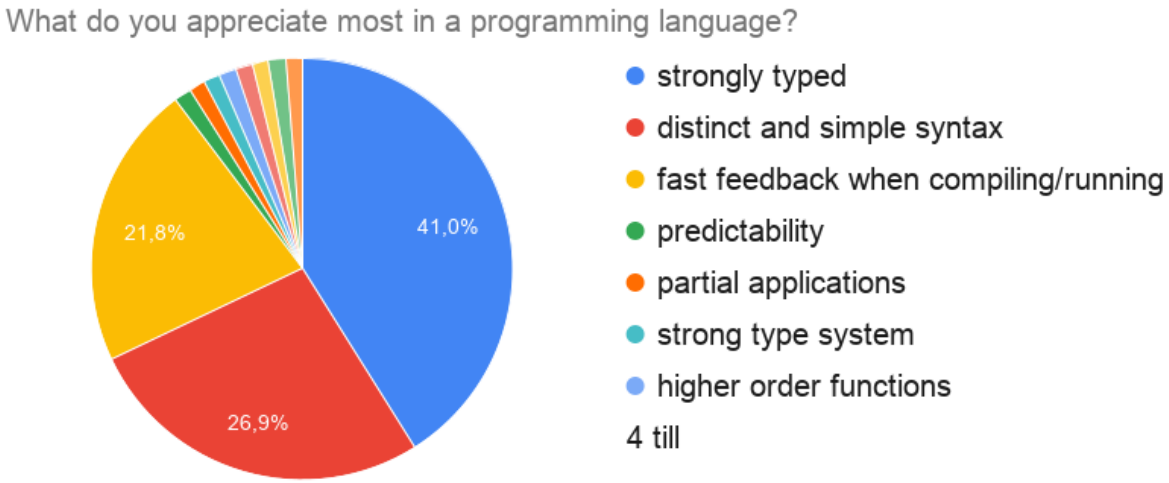

Figure B.3: Distribution of responses to the question "What do you appreciate most in a programming language?" (n = 76).

Question 7

What do you think modern languages lack? Please be specific

Respondents provided free-text answers.

Question 8

Which Integrated Development Environment (IDE) do you prefer to use?

- Visual Studio Code
- JetBrains IDE (IntelliJ/PyCharm/etc)
- Eclipse
- Spyder
- Other

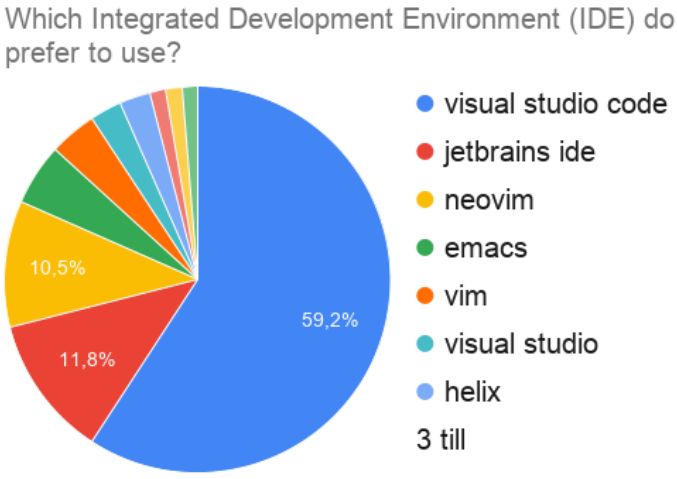

Figure B.4: Distribution of responses to the question "Which Integrated Development Environment (IDE) do you prefer to use?" (n = 76).

Question 9

What in particular do you prefer with this IDE? Does it have something that others don't?

Respondents provided free-text answers.

C Appendix: Keyword Preference Survey

This appendix presents the full questionnaire used in the Keyword Preference Survey. The survey was conducted among university students with varying programming experience and consisted of seven questions. The results are also presented at the corresponding question.

Question 1

How many years of programming experience do you have?

- 0-1
- 2-3
- 4-5
- 6-10
- 10+

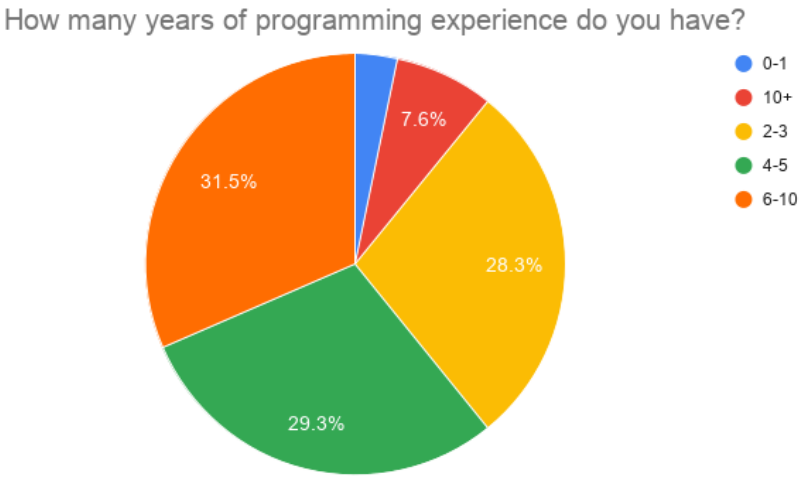

Figure C.1: Distribution of responses to the question “How many years of programming experience do you have?” (n = 92).

Question 2

For the sake of readability and ease of use, which would you prefer?

- if (x && y)
- if (x and y)

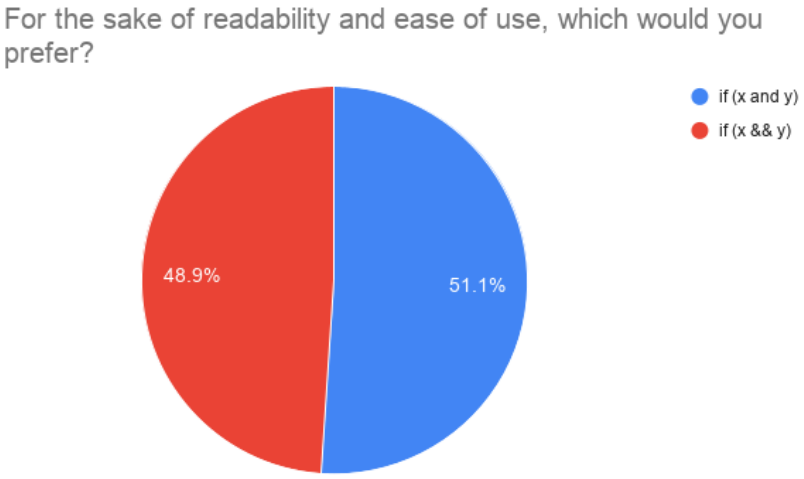

Figure C.2: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 92).

Question 3

For the sake of readability and ease of use, which would you prefer?

- if (x || y)
- if (x or y)

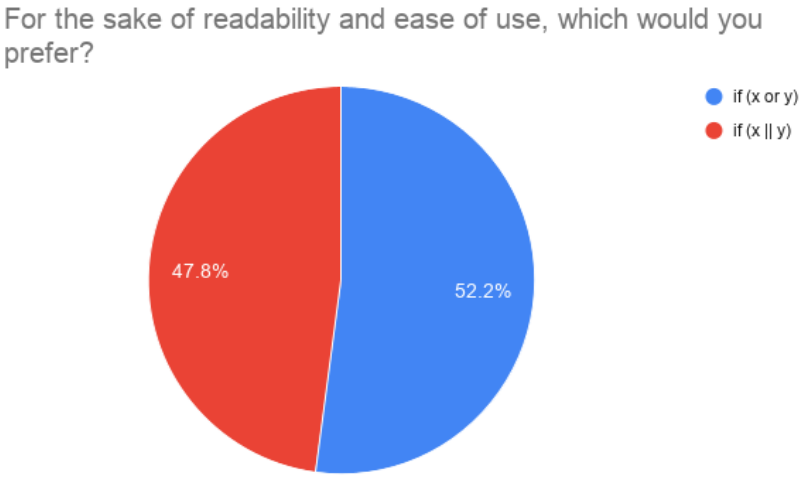

Figure C.3: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 92).

Question 4

For the sake of readability and ease of use, which would you prefer?

- if (!x)
- if (not x)

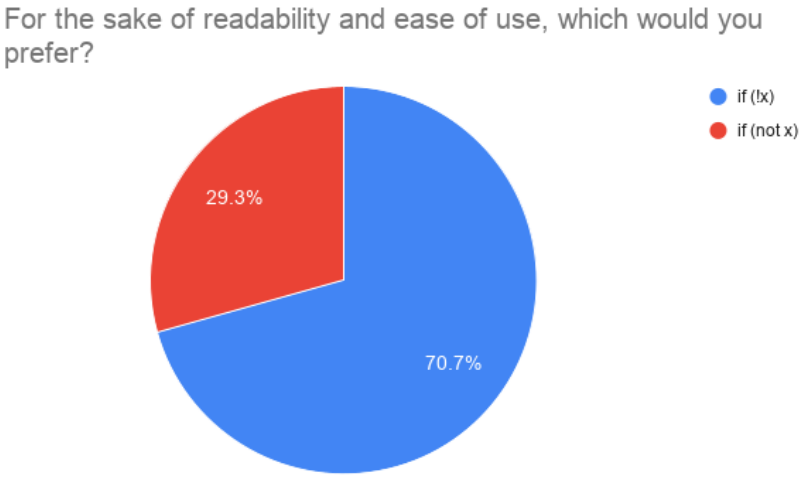

Figure C.4: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 92).

Question 5

For the sake of readability and ease of use, which would you prefer?

- x++ and x–
- inc(x) and dec(x)

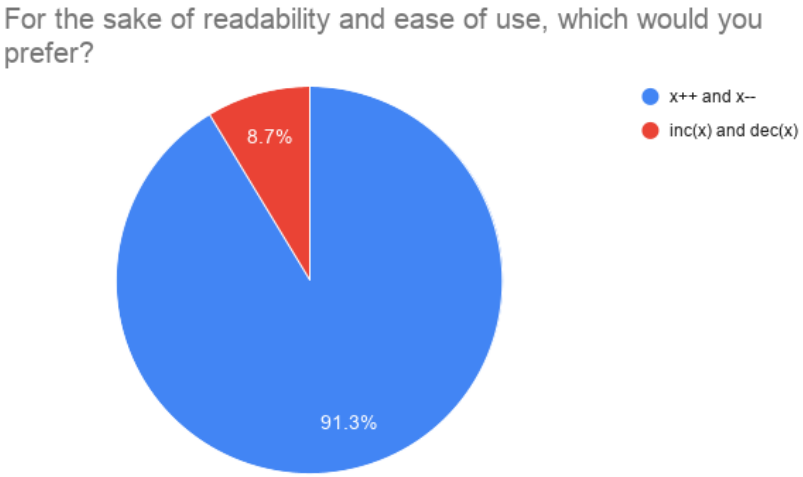

Figure C.5: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 92).

Question 6

For the sake of readability and ease of use, which would you prefer?

- if (x >= y)
- if (x greater or equal y)
- if (x is greater than or equal to y)

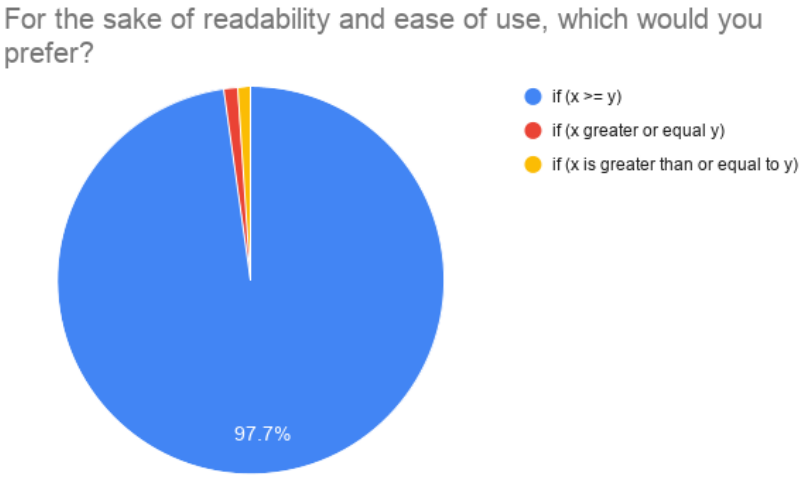

Figure C.6: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 87).

Question 7

For the sake of readability and ease of use, which would you prefer?

- true/false
- True/False
- TRUE/FALSE
- T/F

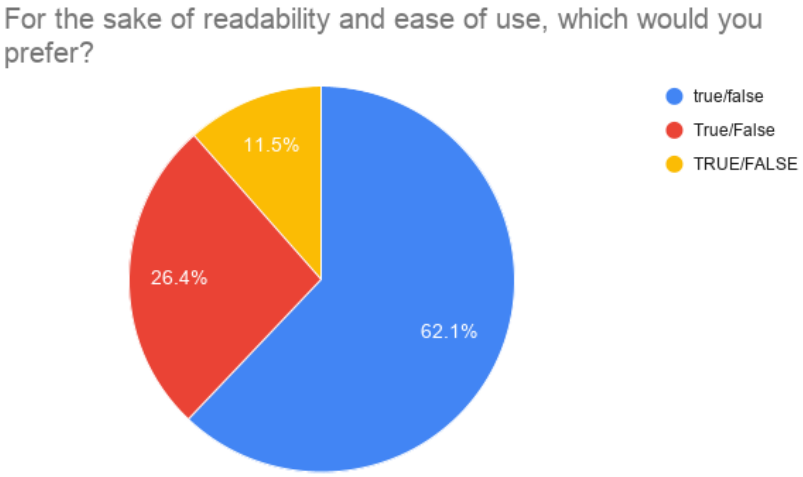

Figure C.7: Distribution of responses to the question "For the sake of readability and ease of use, which would you prefer?" (n = 87).

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026
www.chalmers.se | www.gu.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY