CA$H

Vetle Mathiesen Knutsen[1,2] and Thea Jenny E. Kolnes[1,2]

[1] University of Bergen, Bergen, Norway
vkn006@uib.no tko036@uib.no
[2] Western Norway University of Applied Sciences, Bergen, Norway
189024@stud.hvl.no 189039@stud.hvl.no

Abstract. CA$H is a minimalistic, domain-specific programming language designed for modeling cashier workflows in an intuitive and accessible way. Tailored for users without traditional programming experience, CA$H offers a constrained set of operations such as input handling, arithmetic operations, conditional logic, printing, and discount application, mirroring real-world cash register tasks. The language is implemented with an ANTLR generated lexer and parser, and includes both an interpreter and a compiler that translates CA$H code into LLVM Intermediate Representation. To enhance usability, a custom Visual Studio Code extension provides syntax highlighting. This report presents the language's core syntax and semantics, implementation details, example programs, development insights, and a reflection on modern compiler tools.

Keywords: CA$H Programming Language · ANTLR · LLVM · DSL · Interpreter · Compiler.

1 Language Overview

1.1 Core Concepts

CA$H is a high-level, minimalistic domain-specific language (DSL) designed specifically for cash register operations. It aims to provide an intuitive programming model for domain experts who may not have traditional programming experience. Instead of offering general-purpose constructs, CA$H focuses on a small set of operations such as defining costs, requesting input, printing receipts, and applying discounts, all using keywords that reflect real-world cashier tasks. CA$H programs follow a clear, sequential flow, always starting with a `HELLO.` marker and ending with `BYE.`, structured like a cashier's thought process.

Variables are handled dynamically, supporting integers and floating-point numbers without the need for explicit type declarations. Arithmetic and comparison expressions allow basic calculations needed in everyday sales processes. Control structures are domain-specific: `SCAN` implements loops based on numeric conditions (e.g. number of items left in cart), `CONFIRM`, `CHECK_AGAIN`, and `FALLBACK` provide branching logic in a way that mirrors real-life customer interactions, and `ASK` asks for input. The `DISCOUNT` keyword allows applying percentage-based reductions to prices, supporting common sales scenarios such as seasonal promotions or bulk pricing.

By embedding common cashier operations directly into the language's core, CA$H minimizes the learning curve for non-programming users while still enabling workflows such as repeated item scanning, conditional discounts, and total calculations. Its design prioritizes simplicity, readability, and direct connection between everyday cashier tasks and program instructions.

1.2 Grammar & Syntax

The CA$H programming language is defined using an ANTLR-based grammar, separated into a lexer and a parser. The lexer is responsible for identifying the low-level tokens, as seen in Table 1, that make up the language. These include a set of fixed keywords such as `HELLO.` and `BYE.` to mark the start and end of a program, operational keywords like `COST`, `RECEIPT`, `DISCOUNT`, `ASK`, and `SCAN` to perform tasks related to computation and input/output, and flow control keywords like `CONFIRM`, `CHECK_AGAIN`, and `FALLBACK` for conditional branching. Special symbols like the period, comma, colon, dollar sign, and parentheses are also defined explicitly. Arithmetic operators for addition, subtraction, multiplication, and division are included, along with comparison operators such as equals, less than, less than or equals, greater than, and greater than or equals. CA$H supports basic types including integers, floats, strings, and identifiers composed of alphabetic characters possibly combined with digits or hyphens.

The parser uses these tokens to define the syntactic structure of valid programs. A CA$H program must begin with the `HELLO.` keyword and terminate with `BYE.`, containing a sequence of different types of statements between them.

Main statements handle variable assignment, output printing, discount application, user input requests, and task execution. CA$H replaces traditional branching keywords with domain-specific ones. The `SCAN` keyword is used to implement loops based on expressions, making `SCAN` blocks conceptually similar to while loops. Tasks are modular blocks that define reusable units of work, declared using `START TASK` and executed via the `TODO` command, supporting parameter passing. Expressions in CA$H allow arithmetic operations, including nested calculations, and comparisons enable conditional evaluation. Statements are typically terminated with a dollar sign ($) followed by a newline, ensuring clear separation and readability. Overall, the CA$H grammar combines simplicity with real-world expressiveness, enabling users to write scripts that are clear and intuitive.

Token Type	**Example**
Start / End Keywords	`HELLO.` / `BYE.`
Declaration Keyword	`COST`
Operation Keywords	`RECEIPT`, `DISCOUNT`, `ASK`, `SCAN`
Task Keywords	`START TASK`, `END`, `TODO`, `IN`
Condition Keywords	`CONFIRM`, `CHECK_AGAIN`, `FALLBACK`
Arithmetic Operators	`+ - * /`
Comparison Operators	`= < <= > >=`
Punctuation Symbols	`. , : $ ( )`
Literals	Integers, Floats, Strings
Identifiers	Pattern: [A-Za-z][A-Za-z0-9-]* (e.g., price, price-total)

Table 1. Summary of CA$H Tokens

1.3 Examples

To demonstrate the features and expressiveness of CA$H, the following examples showcase typical usage of the language. These include simple tasks such as printing output, performing calculations, and gathering user input, as well as more advanced constructs like conditional logic, loops, and task definitions. Each example highlights how CA$H balances clarity with functionality in a domain-specific context.

1.3.1 Hello World

The simplest example using CA$H is the classic "Hello World" program, as shown in Listing 1.1. The example prints a message to the user, demonstrating the basic structure and syntax of the language.

```
HELLO.

NOTE This is the code for Hello World in CA$H $
```

```
RECEIPT "Hello World!" $

BYE.
```

Listing 1.1. Hello World Example

1.3.2 Calculations

The next example, shown in Listing 1.2, demonstrates basic arithmetic and variable usage in CA$H. It calculates the total cost of items based on predefined values for price and quantity, then applies a discount before printing the result.

```
HELLO.

COST price = 15 $
COST quantity = 2 $
COST total = price * quantity $

DISCOUNT(10, total) $
RECEIPT "Total: ", total $

BYE.
```

Listing 1.2. Calculations Example

1.3.3 Conditional Logic

Listing 1.3 showcases conditional execution using `CONFIRM`, `CHECK_AGAIN`, and `FALLBACK` blocks. Based on the quantity entered, the program applies a discount, demonstrating CA$H's support for structured decision-making.

```
HELLO.

ASK price = "Enter the price of the item " $
ASK quantity = "Enter the quantity " $

COST total = price * quantity $

CONFIRM quantity >= 20:
    DISCOUNT(20, total) $
CHECK_AGAIN quantity >= 10:
    DISCOUNT(10, total) $
CHECK_AGAIN quantity >= 5:
    DISCOUNT(5, total) $
FALLBACK:
    DISCOUNT(0, total) $

RECEIPT "Discounted total: ", total $

BYE.
```

Listing 1.3. Conditional Logic

1.3.4 Boolean Expressions

Listing 1.4 shows the use of boolean expressions in CA$H. The example combines logical operators such as `NOT`, `OR`, and `AND` to construct compound conditions involving multiple variables. These expressions are used within `SCAN` and `CONFIRM` blocks to control the flow of execution based on runtime values. This highlights CA$H's support for expressive conditionals, enabling precise control over decision-making in interactive and iterative scripts.

```
HELLO.

COST counter = 3 $

COST total = 4 $
COST price = 5 $
COST discount = 10 $

NOTE Should loop 3 times $
SCAN (NOT (counter = 0)):
  RECEIPT "Counter: ", counter $
  COST counter = counter - 1 $
$

CONFIRM (total < 5 OR price > 5) AND discount > 0:
  RECEIPT "Total is low or price is high, and there's a
    discount!" $
  DISCOUNT(discount, total) $
  RECEIPT "Total after discount: ", total $

BYE.
```

Listing 1.4. Boolean Expression Example

1.3.5 Tasks

Listing 1.5 introduces reusable tasks in CA$H using the `START` TASK construct. This example defines a task that repeatedly prompts the user for item details, calculates subtotals, and accumulates a running total, illustrating function-like behavior and parameter passing.

```
HELLO.

START TASK t1 (IN: x):
      COST total = 0 $
      SCAN (x > 0):
          ASK price = "Enter the price of the item " $
          ASK quantity = "Enter the quantity " $
          COST itemTotal = price * quantity $
          COST total = total + itemTotal $
          COST x = x - 1 $
      $
      RECEIPT "Card Total: ", total $
END t1

TODO t1(x: 3) $

BYE.
```

Listing 1.5. Task Example

1.3.6 Fibonacci

Listing 1.6 demonstrates a task-based implementation of the Fibonacci sequence. The fibonacci task handles input validation using conditional blocks and iteratively generates the sequence using a SCAN loop. This example highlights CA$H's ability to model reusable logic with input parameters, conditionals, and iterative computation.

```
HELLO.

START TASK fibonacci (IN: n):
    COST a = 0 $
    COST b = 1 $
    COST count = 0 $

    CONFIRM n < 1:
        RECEIPT "Invalid" $
    CHECK_AGAIN n = 1:
        RECEIPT "Fibonacci when n=1: ", a $
    FALLBACK:
        RECEIPT "Fibonacci series up to: ", a $
        SCAN (count < n):
            RECEIPT a $
            COST nth = a + b $
            COST a = b $
            COST b = nth $
            COST count = count + 1 $
        $
END fibonacci

TODO fibonacci(n: 10) $

BYE.
```

Listing 1.6. Fibonacci Example

2 From Idea to Product

Following the decision on syntax and semantics for the language, the implementation of the interpreter and compiler began. The creation of the interpreter and compiler was heavily based on the course by Patrick Stünkel and the book, *The Definitive ANTLR 4 Reference* [Parr, 2013]. The ANTLR tool translates grammars into parsers that can build parse trees. Parse trees are data structures that represent how grammar matches a given input. Additionally, ANTLR generates tree walkers that can be used to visit tree nodes and execute application-specific code [Parr, 2013].

For the interpreter, we built an ANTLR lexer grammar to recognize input and an ANTLR grammar to specify language syntax. The grammar contained a set of rules, each rule expressing the structure of a phrase.

```
lexer grammar CASHTokens;

START_KEYWORD : 'HELLO.';
END_KEYWORD : 'BYE.';

PERIOD : '.';
DOLLAR : '$';
COMMA : ',';
COLON : ':';

COST_KEYWORD : 'COST';
PRINT_KEYWORD : 'RECEIPT';
...
```

Listing 1.7. Lexer grammar – CASHTokens.g4

```
grammar CASH;
import CASHTokens;

program : START_KEYWORD NEWLINE (main_stmt | cond_mod |
    scan_mod | task_mod)* END_KEYWORD NEWLINE?;

main_stmt : statement DOLLAR NEWLINE;

statement : COST_KEYWORD IDENTIFIER COMPARE_EQ expression #
    cost
        | PRINT_KEYWORD (STRING (COMMA expression)? |
    expression) # print
...
```

Listing 1.8. Grammar – CASH.g4

After the lexical analysis, the next step was the parser. The parser received the tokens to recognize the sentence structures and generated a parse tree. To walk the parse tree, ANTLR provides two tree-walking mechanisms, Listeners and Visitors [Parr, 2013].

In order to control the walk, we used a Visitor, a generated interface from a grammar that provides a visit method per rule. Due to Python being our target language, we made ANTLR generate Python files. The visitor methods follow a structured approach in which each `visitX()` method corresponds to a specific grammar rule `X` from the parser.

For the interpreter, the methods directly evaluate or execute statements during runtime. Arithmetic expressions recursively compute values, while constructs such as conditionals or loops dynamically control the flow and use a symbol table to keep track of the state. In contrast, the compiler translates the same grammar-based logic into LLVM Intermediate Representation (IR) using `llvmlite`. Each visit method constructs IR instructions through `IRBuilder`. The compiler generates low-level code that is a reflection of the structure of the source program.

3 Setup instructions

The project source code is available on GitHub (see Section Source Code). To use CA$H, it is necessary to download the repository for the project. It can be downloaded by navigating to the terminal and entering the following command:

```
git clone https://github.com/tjekol/DAT259-CASH.git
```

3.1 Installation guide of ANTLR

The CA$H programming language requires certain tools for execution. The installation section of this setup guide is specific to Mac OS; however, guides for other operating systems can be found on the tool's websites if needed.

The initial step involves downloading the JAR file and executing it using the Java virtual machine. The JAR file location is specified as `/usr/local/bin` in this guide, and subsequent commands will utilize this path. It is imperative that ANTLR is executed following the setup of the following aliases and environment variables (inspired by the book [Parr, 2013]):

```
export CLASSPATH=".:/usr/local/bin/antlr-4.13.2-complete.jar:
    $CLASSPATH"
alias antlr='java -jar /usr/local/bin/antlr-4.13.2-complete.
    jar'
alias antdbg='java org.antlr.v4.gui.TestRig'
```

3.2 Write a program

To write a program, navigate to the root folder of the project and create a file with the suffix `.cash` and take inspiration from the example codes in Section 1.3 and check the CA$H Tokens Table 1. A CA$H program needs to start with `HELLO.` and end with `BYE.` The interpreter works with all the operations; however, the compiler only works with `RECEIPT`, `COST` and `DISCOUNT` operations as for now. The example below 1.9 shows some general operations that you can try out!

```
HELLO.
NOTE Code has to be in between the HELLO. and the BYE. $
RECEIPT "Hello, I'm a string output." $
```

```
NOTE Next line is variable assigmnet $
COST var = 20 $
ASK userInput = "Ask the user for a number" $
BYE.
```

Listing 1.9. Example of operations

3.3 Execute interpreter

To execute the interpreter, the following tools are needed:

– **antlr4-tools**, installed with `pip install antlr4-tools`
– **runtime library**, installed with `pip install antlr4-python3-runtime` (for python in this project)

Following the installation of both tools, the next step is to generate the Python files and the visitor class for both the interpreter and compiler. These files can be generated using the given command:

```
 antlr -Dlanguage=Python3 CASH.g4 -o
 interpreter/cash -visitor && antlr -Dlanguage=Python3
 CASH.g4 -o compiler/cash -visitor
```

At this stage, all the tools and files needed to run the interpreter have been installed and generated. There are multiple example files in the repository that can be tested, but in order to execute the interpreter with your file, use the command (`/path` is to be replaced with your file path):

```
python interpreter/interpreter.py /path/<fileName>.cash
```

Listing 1.10. Command to run interpreter

3.4 Execute compiler

In order to compile a file, the `llvmlite` package is needed. The installation of the package is initiated with the command `pip install llvmlite==0.44.0`. Subsequent to the installation of `llvmlite`, the following command compiles a file:

```
python compiler/compiler.py /path/<fileName>.cash
```

Listing 1.11. Command to run compiler

3.5 Syntax highlighting (optional)

This VSCode extension provides syntax highlighting for the CA$H programming language by using a TextMate grammar [VSCode Documentation, 2025] that defines patterns to colorize different language elements. The extension is installed via the VSCode command line and works exclusively in VSCode. For additional customization, you can change the file icon theme by opening the command palette, searching for "File Icon Theme", and selecting "CA$H File Icons" to display a dollar sign icon for `.cash` files. Although, note that this currently will show blank icons for all other file types.

```
code --install-extension syntax-highlighting/cash-0.0.1.vsix
```

Listing 1.12. Command to add syntax highlighting

4 Reflection

4.1 Reflection on Language Development with Modern Tools

Throughout the course, we have been instructed in the creation of our own programming language with the help of modern tools like ANTLR and LLVM. These tools have changed the way compilers are built by taking care of many of the low-level details for us. As described in the "Dragon Book" [Aho et al., 1986], tasks like syntax analysis once required significant intellectual effort, but are now considered relatively easy thanks to powerful parser generators, enabling us to prioritize the design of the language itself. Traditionally, the development of a parser necessitated the manual composition of tokens and a comprehensive understanding of parsing theory and code generation methodologies.

ANTLR, for example, makes it really easy to define a grammar and automatically get a working lexer and parser. ANTLR handles the complex parsing work for us, so we can define a grammar and get a lexer and parser without diving into the technical details. A similar approach is employed by LLVM, which facilitates the generation of low-level code without the need to write assembly or deal with the low-level details of processor operations. LLVMlite is a tool that enables the writing of intermediate representation code in Python, thereby allowing the user to have control over the language's functionality while avoiding the necessity of diving into the details of hardware-level complexity. These kinds of tools match what the "Dragon Book" describes as "successful" compiler-construction tools: they hide complex generation logic and produce components that are easy to integrate [Aho et al., 1986].

That said, there are still important things we need to understand. The capacity of a tool to facilitate parsing does not negate the necessity of comprehending the fundamental principles of grammar. If an error occurs in our grammar or we want to design a particular feature, it's important to know how the parser processes such elements. In addition, we also have to understand semantic analysis, including the scoping of variables, the verification of data types, and the behavior of functions. These components are not automated for us, and are essential to creating a language that works effectively.

Therefore, the tools enable us to skip over mechanical work to a certain extent and let us focus on the creative and logical parts of language design. However, to use them properly, a good understanding of the theory behind them is substantial. There's no need for compiler experts to build a language, but a more profound understanding of the tools leads to better utilization of them and increases the likelihood of resolving issues when things don't work as expected.

In essence, modern tools do not replace the need to comprehend compiler theory, but they make it easier to apply the theory in practice. The tools enable people to engage in experiential learning with language development without needing a comprehensive foundation of the compiler theory.

4.2 Comparison with Mainstream Languages

To understand the role and design of the CA$H programming language, it is beneficial to compare it with well-known general-purpose languages such as Python and Java. Although these languages are designed for a wide range of software development tasks, CA$H takes a different approach by focusing on simplicity, clarity, and domain-specific functionality. Additionally, there is a difference in how each language is processed and executed through its toolchain. A toolchain is defined as the sequence of tools that collaborate to transform the source code into executable behavior. This typically includes lexers and parsers to analyze syntax, interpreters or compilers to convert code into machine-executable instructions, and possibly linkers or runtime environments to manage execution.

Both Python and Java offer robust standard libraries and powerful runtime environments. They support a wide variety of programming paradigms, including object-oriented and functional programming. Furthermore, they are well-suited for building large-scale applications, handling complex data structures, and integrating with a wide range of technologies.

In contrast, CA$H is designed as a lightweight, domain-specific language with a much narrower scope. Its syntax is intentionally minimalistic, using high-level constructs like `COST`, `SCAN`, and `RECEIPT` to express common control flow and state operations in a readable and concise way.

The Python toolchain is highly dependent on interpretation. The source code is parsed and executed directly through a virtual machine, making it suitable for rapid development and scripting. Java compiles the source code into an intermediate bytecode format that is executed through the Java Virtual Machine (JVM). As a result, Java achieves both portability and performance through compilation and runtime optimization.

CA$H's pipeline is similar in concept but more lightweight. The language uses ANTLR for lexing and parsing the source code into a parse/syntax tree. The execution can then be performed using a custom-built interpreter or compiled into an LLVM IR. CA$H lacks advanced features, such as proper memory management or a runtime environment, and its toolchain resembles the fundamental structure of more mature programming languages. The design allows CA$H to serve not only as a practical language for modeling small programs but also as an instructive example of how high-level code is transformed into executable logic in modern programming environments.

Overall, CA$H cannot match the breadth or optimization performance of languages such as Python or Java, but it offers a focused approach to explore language design, compiler construction, and domain-specific abstraction.

4.3 Future Work

For future work, extending the compiler would have been the next step. The compiler is currently only compatible with the `RECEIPT`, `COST`, and `DISCOUNT` operations. Developing it to work with conditions and loops would have been a priority if the project had allowed for that. Currently, CA$H only supports `integer`, `float` and `string` data types. CA$H does not support sequence types such as `lists`, nor loops that iterate over sequences. Allowing the language to work with sequence types would have been a part of the future work as well.

Another feature that would have improved the developer experience is a build tool. Currently, compiling a CA$H program involves several manual steps, including parsing, visiting the parse tree, generating LLVM IR, and compiling it with Clang. A build tool could automate these tasks, manage dependencies, and provide support for optimization and debugging. This would make the language more user-friendly and scalable, similar to how tools like `make`, `cargo`, or `gradle` serve other languages.

When developing CA$H, we looked into an Language Server Protocol (LSP) tool that could provide language features like auto-complete and mark warnings and errors. However, setting up the LSP proved to be difficult. Due to time constraints, we prioritized compiler development, code quality, and other features.

All in all, we are satisfied with the outcome of the project and the results of how CA$H evolved to be. Although the language lacks full coverage, improved tooling, and deeper integration with modern development environments, the current version of CA$H provides a solid foundation upon which future improvements can be built.

Source Code

The project is available at: github.com/tjekol/DAT259-CASH

References

Aho et al., 1986. Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Massachusetts, USA, 1st edition.

Parr, 2013. Parr, T. (2013). *The Definitive ANTLR 4 Reference.* The Pragmatic Bookshelf.

VSCode Documentation, 2025. VSCode Documentation (2025). Syntax highlight guide.