Universidad Nacional Autónoma de México

Ingeniería en Computación

Compiladores

Entrega de proyecto final (Compilador)

Alumno:
320198388
320051665
320298608
320244612
320054336

Grupo 5
Semestre 2025-2

México, CDMX, Junio 2025

Contents

1 Introduction

In this project, we will develop a compiler and its corresponding parser using the Hare programming language, both as the implementation language and as the language to be compiled. The main focus will be on the design and implementation of the syntax and semantic analysis stages, which are essential components of any compiler.

Syntax analysis verifies the structural correctness of the source code according to the language's grammar. This involves constructing a parser capable of identifying valid program constructs and producing an intermediate representation, typically in the form of a syntax tree. For this stage, we develop a recursive descent parser by applying theoretical concepts.

Semantic analysis complements syntax analysis by checking that the code adheres to the language's rules beyond its form, ensuring, for example, that operations are applied to compatible types and that identifiers are properly declared. It also involves building a symbol table to store relevant information about program elements.

Creating a compiler is necessary for understanding how programming languages are processed and executed. By completing this project, we aim to produce a working compiler for Hare and gain practical knowledge of compiler construction techniques, particularly those related to parsing and semantic validation.

2 Theorical Background

2.1 What is a Compiler?

A compiler is a program that reads a program written in one language, the source language, and translates it into an equivalent program in another language, the target language. In essence, a compiler maps a source program into a semantically equivalent target program. This mapping process consists of two main parts: analysis and synthesis.

The analysis phase breaks the source program into its constituent components and assigns a grammatical structure to them. Based on this structure, it generates an intermediate representation of the source program. During this phase, the compiler also collects information about the program and stores it in a data structure known as the symbol table. This table, along with the intermediate representation, is passed to the synthesis phase.

The synthesis phase uses this intermediate representation and the symbol table to construct the corresponding target program. Compilation is typically organized as a sequence of phases, with each phase transforming the program representation in preparation for the next.

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters from the source program and groups them into meaningful sequences called lexemes. For each lexeme, it produces a corresponding token, which is then passed to the next phase, syntax analysis.

The second phase, known as syntax analysis or parsing, uses the tokens generated by the lexical analyzer to build a tree-like intermediate structure that reflects the grammatical organization of the token stream. A common form of this structure is the syntax tree, where each internal node represents an operation, and the children represent its operands. The syntax tree visually illustrates the order in which operations are to be executed.

The semantic analyzer takes the syntax tree and symbol table to ensure the source program complies with the language's semantic rules. It also gathers type information,

storing it in either the syntax tree or the symbol table for later use during intermediate code generation. A key task during semantic analysis is type checking, where the compiler verifies that each operator is used with correctly typed operands.

As the compiler translates the source program into target code, it may generate one or more intermediate representations, which can take various forms. After completing syntax and semantic analysis, many compilers produce a low-level, machine-like intermediate representation, essentially a program for an abstract machine. This is often expressed as a sequence of three-address code instructions.

The code generation phase takes this intermediate representation and translates it into the target language. If the target is machine code, the compiler allocates registers or memory locations for each variable and translates the intermediate instructions into machine instructions that perform the equivalent tasks.

An important function of a compiler is to record the variable names used in the source program and track information about each one's attributes. This is managed through the symbol table, a data structure that contains an entry for each variable, with fields for its associated attributes.

For large programs, compilation is often done in parts. The resulting relocatable machine code may need to be linked with other object files and libraries to form a complete executable. The linker resolves references to external memory addresses where code in one file refers to locations in another. Finally, the loader places the fully linked executable into memory, ready for execution.

2.2 About the sintax analysis

Parsing is a fundamental process in the field of computer science, especially within the domains of compilers, interpreters, and language processing systems. It involves analyzing a sequence of tokens or symbols based on the rules defined by a formal grammar, with the purpose of constructing a syntactic structure, often in the form of a parse tree or abstract syntax tree. Parsing techniques are generally classified into top-down and bottom-up methods. Recursive descent parsing belongs to the former category and is one of the most widely understood and implemented parsing strategies due to its straightforward and modular approach.

Recursive descent parsing operates by associating each non-terminal in a grammar with a corresponding function in the parser. These functions call one another in a recursive manner to recognize structures in the input, hence the term "recursive descent". As the input string is processed from left to right, the parser attempts to apply production rules that correspond to the structure of the grammar, descending through the hierarchy of rules until it either accepts the input or fails due to a mismatch.

This method is particularly suitable for grammars that conform to the LL(1) class, meaning they can be parsed from left to right with Leftmost derivation using one lookahead token. However, for recursive descent to function correctly, the grammar must be free of left recursion. Left-recursive productions, which allow a non-terminal to appear as the leftmost symbol in one of its own derivations, can lead to infinite recursion and must therefore be transformed before the grammar can be parsed using this technique. Similarly, ambiguous or poorly factored grammars may require rewriting to ensure that decisions can be made deterministically based on the next token in the input stream.

Despite its advantages, recursive descent parsing does have limitations. Its reliance on grammars that are LL(1) restricts its applicability to a subset of possible languages.

Moreover, maintaining a recursive descent parser for a large and complex grammar can become cumbersome and error-prone, particularly when compared to automated parser generators or more powerful bottom-up parsing techniques. Nevertheless, for many use cases, particularly those involving smaller grammars or where control and transparency are valued over generality, recursive descent remains an effective and reliable method.

2.3 About the semantic analysis

Syntax-Directed Translation (SDT) is a fundamental concept in compiler design that integrates semantic processing with syntactic analysis. It relies on associating semantic rules or actions with the grammar productions of a language. These rules guide the compiler in performing translations such as type checking, intermediate code generation, or symbol table construction, depending on the stage of compilation.

In the context of a recursive descent parser, SDT is implemented through embedded semantic actions within the parser's recursive functions. Each non-terminal in the grammar corresponds to a function, and each production rule is handled by a conditional branch or sequence of statements. As parsing progresses, these functions not only verify syntactic correctness but also invoke code that performs semantic analysis or constructs intermediate representations.

In the compiler developed for this project, SDT serves as the mechanism through which semantic analysis is conducted during parsing. The approach integrates semantic checks directly into the recursive descent procedures, enabling the parser to enforce language rules and gather information necessary for later stages of compilation. Semantic actions are embedded within the parsing functions and executed in accordance with the structure of the grammar, which ensures that semantic correctness is validated as each construct is recognized.

A key component of the semantic analysis is the use of symbol tables, implemented as a set of mappings that correspond to each lexical scope in the program. These symbol tables are maintained and updated dynamically as the parser enters and exits scopes, such as functions or nested blocks. The SDT rules associated with variable declarations and references are responsible for updating or querying these tables, ensuring that variables are declared before use, preventing redefinitions within the same scope, and enabling correct resolution of identifiers across nested scopes.

In addition to managing declarations and references, the SDT implementation also supports stack-based memory layout by calculating and assigning stack offsets to variables at parse time. When a variable is declared, its offset is computed relative to the current frame, and this information is stored in the symbol table. This preparation enables the code generation phase to emit correct memory addressing instructions without requiring an additional traversal of the abstract syntax tree.

2.4 Code generation

Code generation is a critical phase in the compilation pipeline, responsible for translating an intermediate representation of a program into executable code in a target architecture. This process involves a careful mapping from high-level language constructs to low-level instructions, considering both syntactic structure and semantic information collected during earlier compilation stages.

Research in code generation focuses on several fundamental challenges, including instruction selection, register allocation, memory management, and adherence to target architecture conventions. For register-based architectures such as RISC-V, code generation must respect the calling conventions and efficiently manage the use of general-purpose registers and the stack. The translation process must also preserve the semantics of the original program, ensuring that variables are correctly stored and retrieved, control flow is maintained, and expressions are evaluated in the proper order.

A key element in the code generation process is the integration of information from the symbol table and the syntax-directed translation framework. Stack-based memory layout, for instance, is typically resolved during or after parsing by assigning memory offsets to variables based on their scope and lifetime. This allows code generators to emit instructions such as lw and sw with accurate address calculations, ensuring correct runtime behavior.

Modern code generation techniques often involve multiple levels of intermediate representations, which can simplify target code emission and enable architecture-specific optimizations. However, in certain cases, a single-pass approach that emits target code directly during parsing may be appropriate, especially in compilers targeting constrained environments or educational settings.

Although sophisticated code generators incorporate optimizations like instruction scheduling or loop unrolling, even minimal implementations must carefully handle fundamental tasks such as translating expressions, managing scope transitions, and preserving the control flow of the source program.

Targeting the RISC-V instruction set presents particular opportunities and challenges. As an open, modular architecture with a relatively clean instruction set, RISC-V offers a compelling platform for compiler research and experimentation. It allows researchers to explore code generation strategies without the complexity of legacy architectures, while still addressing real-world constraints such as register pressure, alignment, and function call conventions.

3 Development

3.1 Parser

The parser is a fundamental component of the compiler, responsible for analyzing the syntactic structure of the source code and transforming it into an internal representation that the rest of the compiler can work with. In this project, the parser was built specifically to process code written in the Hare programming language.

The development of the parser began with the formal definition of the grammar rules that describe valid Hare constructs. These rules, grounded in theoretical models such as context-free grammars, guided the structure of the parser and ensured that the syntax recognized was consistent with the language specification.

We based on the grammar that was already described in Hare's language especification, we didn't have any trouble since this grammar is LL(1). This makes our parser design a lot simpler, which was a deliberate goal of the language design. Some of the sections implemented are:

```
- Expressions
   - literal:
      - integer-literal
```

```
      - rune-literal
      - string-literal
      - struct-literal
      - true
      - false
      - void
      - null
      - done
   - unary-operator:      one of:
      - - ~ ! * &
   - comparison-expression:
      - inclusive-or-expression
      - comparison-expression < inclusive-or-expression
      - comparison-expression > inclusive-or-expression
      - comparison-expression <= inclusive-or-expression
      - comparison-expression >= inclusive-or-expression
   - equality-expression:
      - comparison-expression
      - equality-expression == comparison-expression
      - equality-expression != comparison-expression
   - logical-and-expression:
      - equality-expression
      - logical-and-expression && equality-expression
   - logical-or-expression:
      - logical-xor-expression
      - logical-or-expression || logical-xor-expression
```

Here are some simple examples:

```
Code:

int f(){

return 999;

}

Grammar

<program>

<function>

 "int" <id> "(" ")" "{" <statement> "}"

 "int" f "(" ")" "{" <statement> "}"

 "int" f "(" ")" "{" "return" <exp> ";" "}"
```

```
 "int" f "(" ")" "{" "return" 999 ";" "}"
```

```
Code:

int suma(int a, int b) {

  return a + b;

}

Grammar

<program> ::= <function>

<function> ::= "int" <identifier> "(" <identifier> ")" "{" <statement> "}"

<id> ::= "identidad"

<id> ::= "x"

<statement> ::= "return" <exp> ";"

<exp> ::= <identifier>

<identifier> ::= <name>

<name> ::= "x"
```

We also implemented for loops and if statements

The parser was designed to process a sequence of tokens —produced by a lexer— and validate whether those tokens form valid syntactic structures like variable declarations, expressions, or function definitions. During this stage, the parser also constructs an abstract syntax tree (AST), which represents the hierarchical structure of the program in a form that can be traversed and analyzed by later compilation stages.

Key to the development was the identification and separation of concerns. Each syntactic construct was handled by a specific part of the parser, allowing for modularity and clarity. The design aimed to closely follow the grammar rules, making the parser easier to extend and debug.

By the end of this phase, the parser was capable of transforming raw Hare source code into a structured and meaningful representation, ready for semantic analysis and code generation. This work highlights the importance of applying formal language theory and grammar design in the practical construction of programming tools.

3.2 Compiler construction

The integration of code generation in the compiler, is inspired in the logic and structure proposed by Nora Sandler in her blog post "Let's Write a Compiler". The original article outlines a minimalistic approach to compiler construction, targeting the x86-64 architecture.

However, the implementation presented here opts for a more modern and RISC-oriented alternative: the RISC-V 64-bit architecture (riscv64). This transition reflects both pedagogical and practical motivations, including the simplicity and regularity of RISC-V's instruction set, which is well-suited for educational compilers and facilitates understanding of low-level code generation principles.

The compiler backend is structured around generating RISC-V assembly code from an abstract syntax tree (AST), following a recursive descent through expressions and statements. The central function, gen, initiates this process by traversing a list of top-level declarations and dispatching function declarations (ast::$\text{decl}_{\text{func}}$) to gen_{fn}. Each function is compiled into a global RISC-V symbol, with appropriate prologue code for stack manipulation (addi sp, sp, -4 and sw ra, 0(sp)) to preserve return addresses. Parameter handling is simplified by supporting a single argument, moved into the a registers and stored on the stack.

Variable management is facilitated through a custom var_{map}, which serves as a hash table mapping variable names to stack offsets. This design ensures proper variable scoping and reuse across expressions, and enables the generation of correct load (lw) and store (sw) instructions for identifiers. The compiler maintains stateful information such as stack depth and control structure counters (if_{cntr}, for_{cntr}) to support block-structured code generation and proper label generation for branches and loops.

Each expression type defined in the AST, such as binary operations, conditionals, loops, function calls, and variable assignments, is compiled via a corresponding gen_* function. These functions emit appropriate RISC-V assembly instructions. For example, binary operations like logical OR ($\text{gen}_{\text{lorexpr}}$) and AND ($\text{gen}_{\text{landexpr}}$) follow a push/pop stack discipline to maintain intermediate values, while arithmetic operations are compiled into instructions such as add, sub, and mul. Conditional constructs ($\text{gen}_{\text{ifexpr}}$) and looping structures ($\text{gen}_{\text{forexpr}}$) leverage label-based branching and maintain correct stack discipline by restoring the previous stack state after control-flow divergence.

Return statements are compiled with stack cleanup and either a jump to ecall for the main function or a ret for others, following conventional calling conventions. Additionally, the code handles compound blocks by creating new variable scopes and carefully managing stack unwinding, especially in the presence of non-returning branches.

4 Results

In this repository, you will find the `compile.ha` source file, which can be compiled and run using the usual Harelang toolchain. The resulting program will emit an assembly program to `stdout` which can be redirected in to a file for linking and finally execution.

```
$ cat testfiles/basic.ha
fn main() int = {
        return 3 + 1;
};
$ hare run compile.ha -f testfiles/basic.ha
.global _start
_start:
        addi sp, sp, -4
        sw ra, 0(sp)
        li t0, 3
```

```
        addi sp, sp, -4
        sw t0, 0(sp)
        li t0, 1
        lw t1, 0(sp)
        addi sp, sp, 4
        add t0, t1, t0
        addi sp, sp, 0
        mv a0, t0
        sw ra, 0(sp)
        addi sp, sp, 4

        addi a7, x0, 93
        ecall
```

We created a set of source files in order to test the compiler we built. In general, these files utilise the basic constructs of the language, like `for` loops, conditional blocks (`if-else`), variable bindings, assignments and scope management, function definitions and calls.

Consider this program:

```
fn foo(x: int, y: int) int = {
        if (x < 10) {
                y *= 2;
        };
        return x * y;
};

fn main() int = {
        let x = foo(4, 5);
        return x + 5;
};
```

After compilation and execution, the status code of the program should be the output:

```
$ qemu-riscv32 program
$ echo $?
45
```

The results are the following, note that the `nix-shell` command should be replaced by `make`. Nix is being used to provide an immutable development shell. That is not needed for the compiler to work:

It is clear that execution is correct, although not optimized. There is no functioning *IO* system, which limits our output to program status codes. Still that is enough to showcase that compilation is being done.

5 Conclusion

This project demonstrates the practical application of theoretical concepts in compiler construction, highlighting the structured progression from abstract syntax rules to concrete

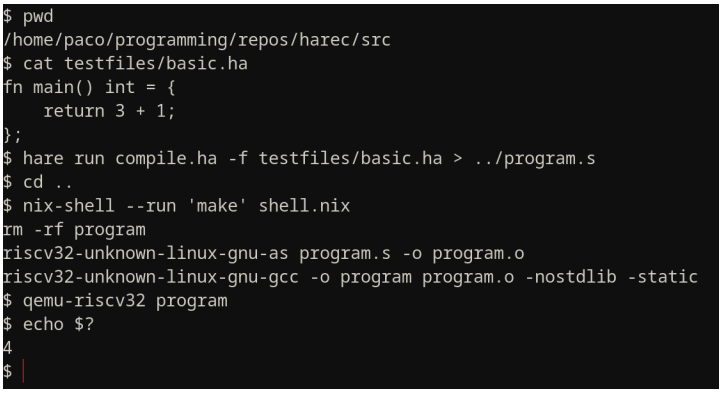

Figure 1: Probando un programa básico con una operación aritmética.

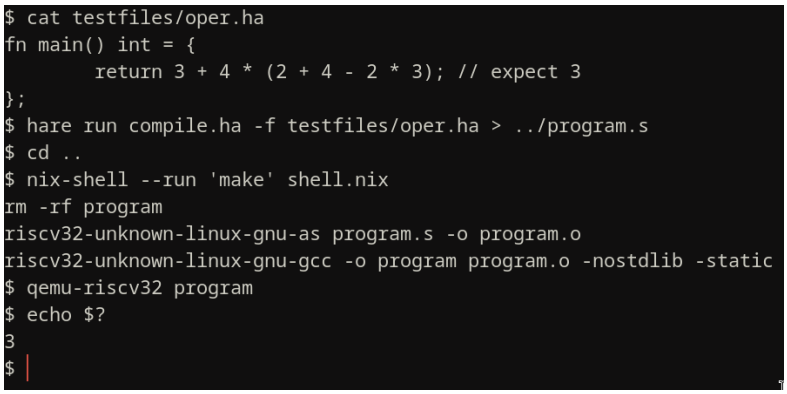

Figure 2: Probando un programa con operaciones aritméticas más complejas.

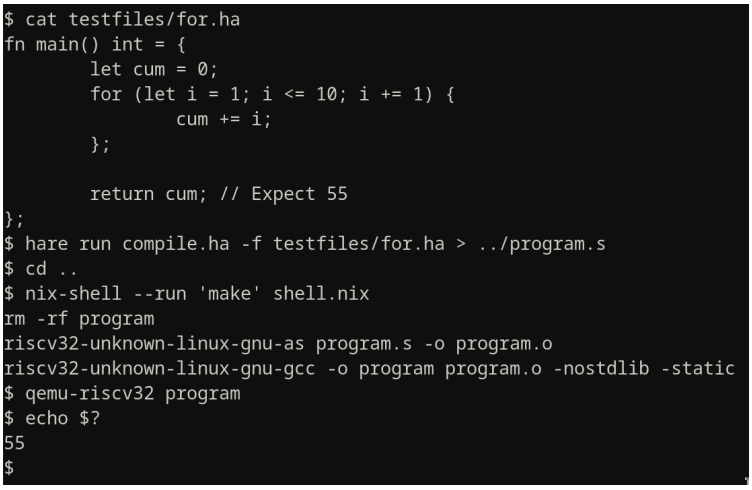

Figure 3: Probando un programa con un ciclo `for`.

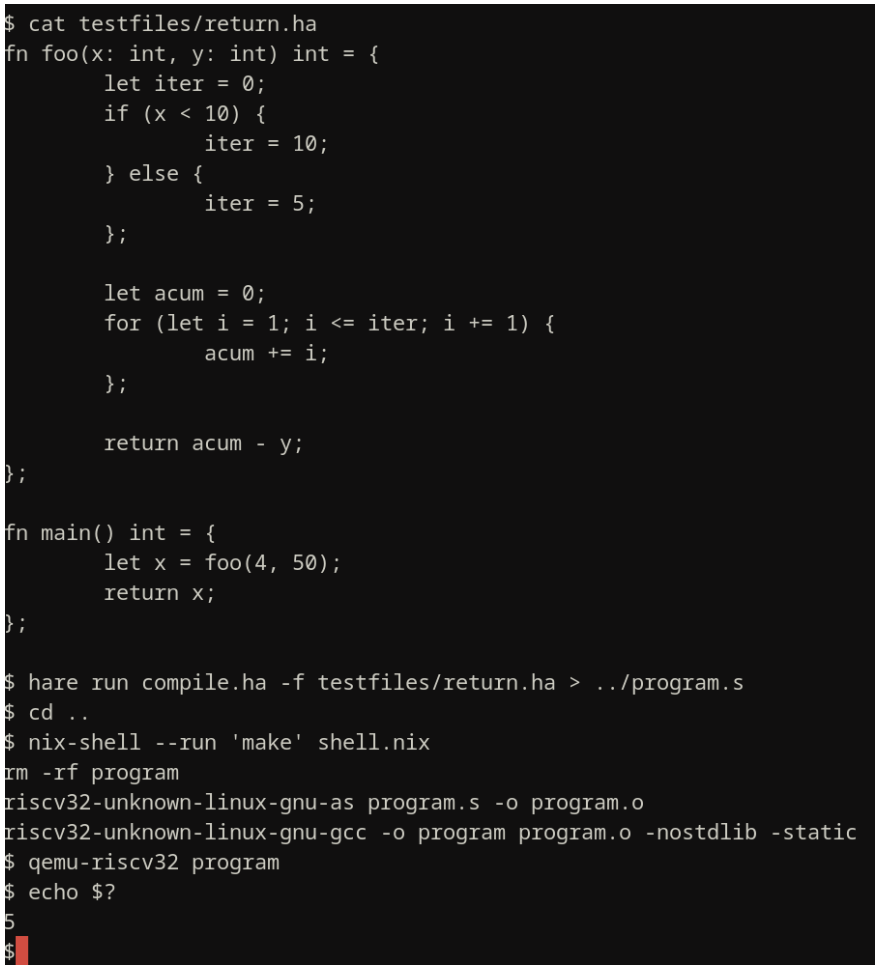

Figure 4: Probando un programa con una función, condicionales y ciclos.

code generation. By implementing a compiler and parser for the Hare programming language, written in Hare itself, we reinforced foundational principles such as formal grammars, lexical analysis, syntax analysis, and semantic analysis.

Instead of relying on external libraries during the syntax analysis phase, we chose to build a recursive descent parser based on theoretical concepts covered in class and supported by bibliographic references. The semantic analysis phase was implemented through a symbol table that validates identifiers by checking their presence within a defined scope. This approach aligns with the theoretical model of symbol resolution and scope management, illustrating how simple data structures can effectively enforce semantic rules.

Furthermore, by targeting the RISC-V architecture for code generation, the project required precise translation from high-level constructs to a register-based, low-level instruction set. This provided a concrete application of the theoretical mapping between intermediate representations and target machine code.

Our compiler, although basic in comparison to a Java compiler, supports block-structured code such as if statements, for loops, and functions, allowing simple programs to be parsed, compiled, and executed successfully.

Overall, the project exemplifies how theoretical knowledge, such as grammar definitions, parsing strategies, scope rules, and target architecture modeling, can be systematically applied to develop a working compiler. The results validate the importance of a solid theoretical foundation for solving complex systems-level programming problems.

6 References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Jan. 1986. [Online]. Available: `https://doi.acm.org/10.1145/6448`.

[2] *Hare specification*. [Online]. Available: `https://harelang.org/specification`.

[3] A. W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*. Cambridge University Press, Jul. 2004.

[4] Rswinkle, *RISC-V Assembly Programming*. [Online]. Available: `https://github.com/rswinkle/riscv_book`.

[5] N. Sandler, *Writing a C Compiler, Part 1*, Nov. 2017. [Online]. Available: `https://norasandler.com/2017/11/29/Write-a-Compiler.html`.