# CMPE 230 - Project 2 Report

Evochirp: Birdsong Evolution Simulator in Assembly

May 12, 2025

## 1 Introduction

This report presents the implementation and evaluation of the Evochirp project, which simulates the evolutionary behavior of bird songs across generations using GNU Assembly. The project demonstrates how low-level programming techniques can be applied to model structured, rule-based transformations in data. Our focus is not only on the correctness of functionality, but also on structuring the Assembly code in a modular and comprehensible way.

## 2 Problem Description

In Evochirp, each bird species interprets a sequence of notes and operators in a species-specific manner. The song starts with a series of notes (C for Chirp, T for Trill, D for Deep Call), followed by operators (+, -, *, H) that transform the song differently depending on the species. The problem involves parsing this input and evolving the song step-by-step according to the transformation rules, outputting the state of the song after each operator is applied.

## 3 Methodology

The methodology underlying the Evochirp simulation is designed to reflect a clear, low-level control flow while maintaining strict adherence to species-specific song evolution semantics. The process begins with input acquisition via a syscall read, capturing a line of text into the .bss-allocated raw_input data buffer (256 bytes). The species name is parsed by iterating over the buffer until the first whitespace character is encountered; this segment is copied into a separate buffer bird_species_name using byte-wise register operations. Once the species is identified by comparing its initial character ('S', 'W', or 'N'), control is transferred to the corresponding logic block using direct jumps (jmp) to labels such as route_to_sparrow_processor. Following the species identification, the rest of the input is tokenized: each character token (representing either a note or an operator) is stored sequentially in a 4-byte-aligned memory array parsed_song_elements. The tokenization loop carefully skips spaces and uses register-indexed memory access to ensure proper alignment and termination conditions.

During processing, the core execution loop iterates over the parsed elements using an index register (%r14), comparing each element with operator symbols (+, *, -, H)

to determine the appropriate transformation logic. Notes are directly copied to the output_song_elements buffer, while operators trigger complex species-dependent routines. Each species implements these operators differently—requiring local stack manipulation, register shuffling, and conditional memory shifts. For instance, Sparrow's + operation merges the last two notes into a hyphenated string by modifying memory in-place, while Nightingale's * operator duplicates the entire output buffer using a nested loop with memory copy instructions. Stateful tracking of the current generation is maintained using the generation tracker variable, incremented after each operation and printed via a syscall-assisted render_integer_as_string routine. This pipeline, though procedurally linear, is deeply sensitive to branching logic and manual memory control, ensuring deterministic and isolated evolution behavior for each song sequence, fully consistent with the project specification.

# 4 Design and Implementation

The implementation of Evochirp is centered around careful memory layout, efficient use of processor registers, and clear division of control flow based on species type. All data structures are statically allocated in the .bss segment to avoid dynamic memory allocation and associated complexity. Each token (note or operator) is stored as a 4-byte aligned word to simplify indexing and manipulation. Key buffers include parsed_song_elements for the raw token stream and output_song_elements for the evolving result.

Control flow is divided using explicit labels and jump instructions, which redirect execution based on species identification. Species-specific handlers implement the semantic transformations for each operator. These handlers modify the output buffer in place or copy values using nested loops and address arithmetic. Register usage is disciplined: %r14 handles input indexing, %r10 tracks output size, %r15 is often used as a base pointer, and temporary values use %r12--%r13. Procedure calls like call display_current_song_state encapsulate repeated logic and help preserve modularity.

One of the most critical routines is the print function, which uses syscall write to output both metadata (e.g., species name, generation count) and the evolving song. We implemented number-to-string conversion manually in render_integer_as_string, respecting stack safety and endianness. These implementation choices reflect not only a concern for correctness but also performance and system-level efficiency.

# 5 Results

The implementation was tested using the official test suite. Below is a screenshot from our terminal showing full correctness with 100/100 grade across all provided test cases:

**The image related to result is deleted due to privacy concerns.

This confirms that our song evolution logic matches the expected results for all species and transformation scenarios. The song updates after each operator match the exact format and content specified in the project documentation, including edge cases such as insufficient operands or redundant reductions. Furthermore, the runtime behavior adheres to the expected input/output structure, demonstrating consistency across test conditions and species.

# 6 Discussion

Throughout development, several key challenges emerged due to the low-level nature of assembly programming. First, memory management had to be handled manually. Since the use of dynamic memory (malloc, heap access) was prohibited, all buffers were statically allocated and carefully indexed. Buffer overflow prevention and correct alignment of 4-byte entries required precise control over indexing logic.

Additionally, the operator implementations posed architectural challenges. For instance, implementing the Nightingale species' harmony logic required accessing the last three tokens in reverse and reconstructing them into a two-part pattern, all while preserving output buffer structure. This demanded controlled pointer arithmetic and awareness of multi-character note formatting (like T-C or D-T).

Print operations added another layer of complexity, as the output format combined both string literals and dynamically generated numeric strings (generation index). This required writing a full base-10 integer renderer using syscall-compatible routines and string reversal via stack operations.

Despite these challenges, the final product is robust and adheres to the evolution rules for all species. However, the solution is limited by static buffer sizing, which places a hard cap on input complexity. There is also limited fault tolerance—invalid inputs are not gracefully rejected but ignored. Nevertheless, the architecture is highly modular and could be extended to support new species or rules in future iterations.

# 7 Conclusion

The Evochirp project has successfully demonstrated how a system with strict evolution rules can be simulated using Assembly. By dividing logic by species and using modular operator handling, the solution is both extensible and efficient. Future improvements may include support for additional species or custom operator rules.

# 8 Usage Instructions

To compile and run the program, use the following commands in the root directory:

```
$ make
$ ./evochirp
Sparrow C C + D T * H
Sparrow Gen 0: C-C
Sparrow Gen 1: C-C D T T
---
```

The Makefile compiles the program and outputs an executable named evochirp. Input is typed in manually after execution.

# AI Assistants

We used ChatGPT for reviewing documentation structure, clarifying Assembly syntax, and getting help during debugging (e.g., syscall behavior). All algorithmic design, parsing, and implementation were written and validated by the authors.