Project Report
EvoChirp: Birdsong Evolution Simulator
Systems Programming – Spring 2025

Ali Ayhan Gunder, 2021400219
Yunus Emre Ozturk, 2022400204

May 11, 2025

Contents

1 Introduction

This project implements a GNU Assembly-based simulator called **EvoChirp**, which models the evolution of bird songs across generations. Inspired by natural avian communication systems, the program interprets a structured input expression composed of species, notes, and operators to simulate species-specific song transformations. Each bird species—Sparrow, Warbler, and Nightingale—responds to the same input in uniquely defined evolutionary ways. This program emphasizes control flow, pointer arithmetic, string processing, and low-level system programming.

1.1 Problem Statement

Birdsong plays a vital role in mating, signaling, and communication. In this simulation, the input represents a bird species and an evolutionary song expression. The program applies each operator to the current song based on the species rules and outputs the song's state after every transformation.

1.2 Objectives

- Develop a GNU assembly program to simulate song evolution.
- Implement per-species behavior for four operators: + (merge), - (reduce), * (repeat), H (harmonize).
- Print the song state after each generation of transformation.
- Ensure correctness with input parsing, string manipulation, and memory-safe operations.

1.3 Motivation

The EvoChirp project challenges students to handle real-world string processing in assembly, using architecture-specific conventions while simulating natural phenomena. It encourages clean, modular design even in a low-level language and reinforces understanding of memory management and control logic.

2 Problem Description

The input format is a single line string structured as:

```
<Species> <Note> <Note> ...  <Operator> <Operator> ...
```

For each valid line, the program must:

- Identify the species (Sparrow, Warbler, or Nightingale).
- Parse the song expression (notes + operators).
- Apply each operator to the song according to species-specific rules.
- Output the resulting song after each operation.

The simulator enforces a strict grammar and assumes no malformed input.

3 Methodology

3.1 Data Structures

The notes array is implemented as a statically allocated block in the stack frame, with a custom format that includes an index counter and 128-entry note buffer. Each entry is 8 bytes (to support string operations).

3.2 Key Functions

- `get_species`: Maps species name to species code (0–2).
- `op_plus`, `op_minus`, `op_star`, `op_h`: Species-specific implementations for each operator.
- `softness`: Determines note softness for Sparrow's reduction rule.
- `process_input_line`: Tokenizes input, dispatches operations, prints each generation.

3.3 Operator Semantics

The operator meanings vary by species. For instance:

- **Sparrow**: Merges with hyphen, transforms notes globally on H.
- **Warbler**: Repeats last notes or appends trill.
- **Nightingale**: Expands entire song or mirrors last three notes.

4 Implementation

4.1 Code Structure

The code is modular and consists of:

- Global constants (.rodata) for strings and formats.
- Dedicated functions for each operator.
- Main loop that reads input and handles exit conditions.
- Tokenized processing with register-safe logic and pointer manipulation.

4.2 Sample Code Snippet

```
# Constants and format strings
.section .rodata
bird_species_sparrow:       .string "Sparrow"
# Name of Sparrow species
bird_species_warbler:       .string "Warbler"
# Name of Warbler species
bird_species_nightingale:   .string "Nightingale"
# Name of Nightingale species
format_string_combine:      .string "%s-%s"
# Format for combining notes
note_c:                     .string "C"
# Chirp note
note_t:                     .string "T"
# Trill note
note_d:                     .string "D"
# Deep call note
space_char:                 .string "␣"
# Space character for tokenizing
error_unknown_species:      .string "Unknown␣species:␣%s\n"
# Error message for invalid species
format_generation:          .string "%s␣Gen␣%d:"
# Format for generation output
format_note:                .string "␣%s"
# Format for individual notes
newline:                    .string "\n"
# New line character
cmd_exit:                   .string "exit"
# Exit command
cmd_quit:                   .string "quit"
# Quit command

# Function: op_plus - Merge notes according to species rules
.globl  op_plus
op_plus:
        pushq   %rbp                    # Set stack frame
        movq    %rsp, %rbp
        subq    $48, %rsp               # Allocate space for local variables
        movq    %rdi, -40(%rbp)         # Store pointer to notes array
        movl    %esi, -44(%rbp)         # Store species code
```

```
        # Check if there are at least two notes
        movq    -40(%rbp), %rax
        movl    1024(%rax), %eax      # Load current note count
        cmpl    $1, %eax              # If notes <= 1, skip operation
        jle     .op_plus_insufficient_notes

        # Get pointer to last note
        movq    -40(%rbp), %rax
        movl    1024(%rax), %eax
        subl    $1, %eax
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -40(%rbp), %rax
        addq    %rdx, %rax
        movq    %rax, -8(%rbp)        # Pointer to last note

        # Get pointer to second last note
        movq    -40(%rbp), %rax
        movl    1024(%rax), %eax
        subl    $2, %eax
        cltq
        leaq    0(,%rax,8), %rdx
        movq    -40(%rbp), %rax
        addq    %rdx, %rax
        movq    %rax, -16(%rbp)       # Pointer to second last note
        # ... (continues with merge logic)

# Function: op_star - Repeat notes based on species rules
.globl  op_star
op_star:
        pushq   %rbp                  # Set stack frame
        movq    %rsp, %rbp
        subq    $32, %rsp             # Allocate space for local variables
        movq    %rdi, -24(%rbp)       # Store pointer to notes array
        movl    %esi, -28(%rbp)       # Store species code

        # Check if notes array is empty
        movq    -24(%rbp), %rax
        movl    1024(%rax), %eax      # Load note count
        testl   %eax, %eax            # Check if zero notes
        je      .op_star_empty_array  # Skip if no notes

        # Select operation based on species
        cmpl    $2, -28(%rbp)         # Nightingale species?
        je      .op_star_nightingale
        cmpl    $2, -28(%rbp)
        ja      .op_star_default
        cmpl    $0, -28(%rbp)         # Sparrow species?
        je      .op_star_sparrow
        cmpl    $1, -28(%rbp)         # Warbler species?
        je      .op_star_warbler
        jmp     .op_star_default
        # ... (continues with repeat logic)

# Function: process_input_line - Process and tokenize input line
.globl  process_input_line
process_input_line:
        pushq   %rbp                      # Set stack frame
        movq    %rsp, %rbp
        subq    $1872, %rsp               # Allocate buffer space
        movq    %rdi, -1864(%rbp)         # Store input line pointer
```

```
        # Copy input safely to buffer
        movq    -1864(%rbp), %rcx
        leaq    -304(%rbp), %rax
        movl    $256, %edx              # Max length 256 chars
        movq    %rcx, %rsi
        movq    %rax, %rdi
        call    strncpy
        movb    $0, -49(%rbp)           # Add null terminator

        # Split input line into tokens
        movl    $0, -4(%rbp)            # Token counter start at 0
        leaq    -304(%rbp), %rax
        movl    $space_char, %esi       # Space char as separator
        movq    %rax, %rdi
        call    strtok
        movq    %rax, -16(%rbp)         # First token pointer
        jmp     .process_tokens
        # ... (continues with token processing logic)

# Function: main - Entry point and main loop
.globl  main
main:
        pushq   %rbp                    # Set stack frame
        movq    %rsp, %rbp
        subq    $256, %rsp              # Buffer for input line

command_loop:
        # Read input line from user
        movq    stdin(%rip), %rdx       # stdin stream
        leaq    -256(%rbp), %rax        # Input buffer address
        movl    $256, %esi              # Max chars to read
        movq    %rax, %rdi              # Input buffer
        call    fgets                   # Read user input line
        testq   %rax, %rax              # Check EOF or error
        je      finish_program          # Exit if EOF or error detected
        # ... (continues with checking for exit and calling processing functions)
```

5 Results

The program was tested using all five official test cases provided in the project description. After addressing minor bugs in memory indexing and species identification, the program passed **100% of the grader tests**. It produces the correct output format and transformations for all operators across all species.

6 Discussion

The primary challenge was emulating high-level string logic using assembly language. Managing memory manually for variable-length strings, ensuring register correctness, and validating grammar compliance added significant complexity. The project highlights the tradeoff between control and convenience in low-level languages.

Future improvements could include:

- Dynamic memory allocation for note arrays.
- Additional operator support or species customization.
- Full input grammar checking and recovery.

7 Conclusion

EvoChirp successfully implements a text-based bird song evolution simulator in assembly. It reinforces foundational systems programming concepts including memory management, stack-based computation, and control flow. The program adheres strictly to specification and output format, and demonstrates a deep understanding of x86_64 GNU Assembly practices.

References

- CMPE230 Assignment 2 PDF
- Piazza Q/A Discussions
- Lecture Slides and Notes
- GNU x86_64 Assembly Documentation

AI Assistants

AI tools such as ChatGPT were used for the following:

- Clarifying assembly syntax and debugging segmentation faults
- Understanding x86_64 calling conventions
- Explaining GNU toolchain behavior and register usage
- Formatting and grammar-checking LaTeX documentation

All implementation decisions and coding were performed by the project authors.