Evochirp Project Report

Systems Programming – Spring 2025

Salih Can Erer, 2022400174

Beyza Nur Deniz, 2021400285

May 11, 2025

Contents

1 Introduction

In this project, we implement a simulation of bird song evolution in GNU Assembly language, following the species-specific transformation rules outlined in the assignment specification. Birdsong evolution is driven by learning, imitation, and environmental adaptation, and our goal is to model this process by reading an initial sequence of notes and a series of operators, then applying each operator in turn to produce a new generation of the song. Through this exercise, we deepen our understanding of low-level programming constructs—such as system calls, buffer manipulation, and control flow in x86-64 assembly—while rigorously handling string parsing and in-place transformations. Successfully completing this project will demonstrate our ability to manage raw memory buffers, implement conditional logic without high-level abstractions, and produce correctly formatted output under strict size and performance constraints.

2 Problem Description

Our assembly program must read a single line of input from standard input, structured as:

$$\langle\text{Species}\rangle\ \langle\text{Song Expression}\rangle$$

where ⟨Species⟩ is one of `Sparrow`, `Warbler`, or `Nightingale`, and ⟨Song Expression⟩ is a sequence of tokens (notes and operators) separated by single spaces. Notes are the symbols `C`, `T`, and `D`, and operators are `+`, `-`, `*`, and `H`. The input line will contain at most 256 characters, and tokens will not include merged notes as operands.

For each operator in the expression, the program must apply the corresponding species-specific transformation to the current song buffer and then print one output line of the form:

$$\langle\text{Species}\rangle\ \text{Gen}\ N:\ \langle\text{current song state}\rangle$$

where N is the zero-based generation index (i.e., the first operator produces Gen 0, the second produces Gen 1, etc.) and the song state is truncated or padded to fit within a 1024-byte output limit. If an operator cannot be applied due to insufficient operands (e.g., merging when fewer than two notes are available), it has no effect on the song, but a generation line must still be printed. All output is sent to standard output, with each generation on its own line, and no extra blank lines or trailing spaces are permitted.

3 Methodology

In this section we describe at a high level how the program ingests its input, dispatches per-species logic, and iterates through operator-driven song generations. We then delve into each step in turn.

Our approach consists of the following major steps:

1. **Input reading and tokenization.** We issue a single `read` syscall to fetch up to 256 bytes into `input_buf`. We then scan byte-by-byte, splitting on spaces to extract tokens (species name, notes, and operators) into a temporary buffer (`temp_buf`) and record pointers or counts to delimit each token.

2. **Species dispatch.** After the species token (`"Sparrow"`, `"Warbler"`, or `"Nightingale"`) is identified, we jump to the corresponding handler code. Each handler shares the same core loop structure but applies slightly different rules for the operators `+`, `-`, `*`, and `H`.

3. **Generation loop.** We maintain a zero-based generation counter in `r13`. For each operator token:

   - Parse the current song stored in `song_buf` (initially copied from the note sequence in `temp_buf`).
   - Perform the species-specific transformation (e.g. for Sparrow "merge" the last two notes into `X-Y` , for Warbler replace `X-Y` with `T-C` , for Nightingale append a duplicate of the last segment, etc.).
   - Update the end-pointer register (`r15`) to reflect the new buffer length.
   - Invoke our `print_gen_song` function to emit "<Species> Gen N: <song_buf>" via `write` syscalls.
   - Increment `r13` and proceed to the next operator.

4. **Edge cases and buffer management.** All transformations are performed in-place on a 1024-byte `song_buf` using pointer arithmetic (AT&T syntax `lea`, `movb`, `sub`, `add`). If not enough notes exist when an operation is requested, we simply skip the transform but still print the generation line. We also ensure no trailing spaces remain by explicitly writing a space (`0x20`) when we shrink the buffer.

5. **Termination.** After consuming all operator tokens, we exit via `exit` syscall.

4 Implementation

Here we detail the layout of our buffers, the role of each register, the design of the helper function, and the species-specific loops that perform merges, repeats, and trills. We start with an overview of register usage before stepping through the code.

The program is written entirely in AT&T-syntax x86_64 GNU Assembly. It is organized into three major segments—`.bss`, `.data`, and `.text`—and a single helper function print_gen_song. Below we describe each part in detail and show the critical code snippets.

4.1 Register Usage

Each register has a dedicated job throughout our assembly interpreter:

Register	**Role**
%r13	Generation counter (–1→0 on first inc)
%r14	"Recent-note" counter (guards merges/repeats)
%r12	Input-scanner pointer (walks through input_buf)
%r15	Write pointer into temp_buf
%r10, %r11	Scratch pointers for copying between temp_buf and song_buf
%rax, %rdi, %rsi, %rdx,%rcx	Syscall arguments and loop counters

4.2 Memory Layout (.bss and .data)

- .bss (uninitialized):
  - input_buf (1024 bytes): holds the entire input line.
  - song_buf (1024 bytes): stores the current generation's song before printing.
  - temp_buf (1024 bytes): staging area for in-place transformations.
- .data (constants and strings):
  - Prompts and separators: prompt, newline, gen, colonsp.
  - numbuf (3 bytes): holds up to three ASCII digits for the generation counter.
  - Species names (with trailing space): sparrow, warbler, nightingale.

4.3 print_gen_song Function

This routine prints "Gen X: " followed by the contents of song_buf, then a newline. It supports generation numbers 0–999 stored in %r13. Key steps:

- **Write "Gen "** via syscall.
- **Convert %r13 to ASCII digits:**
  - ¡10: one digit (.Lpgs_one_digit)
  - 10–99: two digits (.Lpgs_two_digit)
  - 100–999: three digits (divide by 100, then by 10).
- **Write ": "** and then song_buf (up to 1024 bytes).

- **Write newline.**

```
print_gen_song:
# --- print "Gen␣" ---
mov   $1, %rax
mov   $1, %rdi
lea   gen(%rip), %rsi
mov   $4, %rdx
syscall

# --- convert %r13d to ASCII in numbuf ---
mov   %r13d, %eax
cmp   $10, %eax
jl    .Lpgs_one_digit
cmp   $100, %eax
jl    .Lpgs_two_digit
# three-digit case: divide by 100 then 10...
[...]
.Lpgs_one_digit:
mov   %r13b, %al
add   $'0', %al
movb  %al, numbuf(%rip)
mov   $1, %rdx
jmp   .Lpgs_print_number
.Lpgs_two_digit:
xor   %edx, %edx
mov   $10, %ecx
div   %ecx          # EAX=gen/10, EDX=gen%10
add   $'0', %al
movb  %al, numbuf(%rip)
add   $'0', %dl
movb  %dl, numbuf+1(%rip)
mov   $2, %rdx
.Lpgs_print_number:
mov   $1, %rax
mov   $1, %rdi
lea   numbuf(%rip), %rsi
syscall

.Lpgs_print_colon:
# --- print ":␣" ---
mov   $1, %rax
mov   $1, %rdi
lea   colonsp(%rip), %rsi
mov   $2, %rdx
syscall

```

```
# --- print song_buf ---
mov    $1, %rax
mov    $1, %rdi
lea    song_buf(%rip), %rsi
mov    $1024, %rdx
syscall

# --- print newline ---
mov    $1, %rax
mov    $1, %rdi
lea    newline(%rip), %rsi
mov    $1, %rdx
syscall
ret
```

4.4 Entry Point and Species Dispatch (_start)

1. **Print prompt:** `syscall` to write `prompt` (28 bytes).
2. **Read input:** `syscall` to read up to 1024 bytes into `input_buf`.
3. **Initialize registers:** $\%r13 \leftarrow -1$ (so first `inc %r13` yields 0), $\%r14 \leftarrow 0$ (note counter).
4. **Dispatch on first character:** `cmp $'S',(%rsi)` → `.case_sparrow`, `cmp $'W'` → `.case_warbler`, `cmp $'N'` → `.case_nightingale`, else exit.

4.5 Species Handlers

Each species loop shares the same high-level pattern:

1. Load the next token character from `(%r12)` into %cl.
2. **Operator dispatch:** `'+'`, `'-'`, `'*'`, `'H'` jump to species-specific routines.
3. **Note handling:** Otherwise treat %cl as a note character:
   - Store at `(%r15)`, append space, advance %r15 and %r12.
   - Update %r14 (count up to 2 notes for merge/repeat guards).
4. **On operator:**
   - Adjust `temp_buf` in place (e.g. remove, merge, duplicate notes).
   - Increment %r13 (generation), skip operator+space in input (%r12+=2).
   - Copy full `temp_buf` → `song_buf` (up to 1024 bytes).

- Jump to `.species_loop_end` to print species name + call `print_gen_song`.

5. Loop until input byte = 0, then exit.

Below is the skeleton of the Sparrow merge and loop-end logic as an example; Warbler and Nightingale follow the same structure with different transformations:

```
.case_sparrow:
lea  input_buf+8(%rip), %r12   # skip "Sparrow␣"
lea  temp_buf(%rip),    %r15
jmp  .sparrow_loop

.sparrow_loop:
mov  (%r12), %cl
cmp  $'+', %cl
je   .sparrow_merge
# ... other ops ...
# note case:
movb %cl, (%r15)
inc  %r15
movb $'␣', (%r15)
inc  %r15
inc  %r12
# manage %r14...
jmp  .sparrow_loop

.sparrow_merge:
cmp  $2, %r14          # need at least two notes
jl   .merge_skip
movb $'-', -3(%r15)   # overwrite space before last note
.merge_skip:
add  $2, %r12          # skip "+␣"
inc  %r13              # gen++
# copy temp_buf -> song_buf:
lea  temp_buf(%rip), %r10
lea  song_buf(%rip), %r11
mov  $1024, %rcx
.copy_loop:
movb (%r10), %al
movb %al,    (%r11)
inc  %r10; inc %r11
dec  %rcx; jne .copy_loop
jmp  .sparrow_loop_end
.sparrow_loop_end:
mov  $1, %rax; mov $1, %rdi
lea  sparrow(%rip), %rsi; mov $8, %rdx; syscall
call print_gen_song
jmp  .sparrow_loop
```

The Warbler and Nightingale loops replace the merge, reduce, repeat and harm cases with their species-specific transformations.

4.6 Exit

When the null terminator is reached or an unrecognized species is given, the code jumps to:

```
.exit:
mov   $60, %rax    # exit syscall
xor   %rdi, %rdi   # status 0
syscall
```

This completes the detailed implementation of our species-driven bird song evolution simulator in GNU Assembly.

5 Usage Instructions

To compile and run the program, from your project root do:

```
$ make              % assembles and links evochirp
$ ./evochirp        % launches the simulator
```

You can then type (or pipe) your song expression on stdin.

Example

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

6 Results

We exercised the simulator on a variety of hand-crafted and random song expressions for each species. All tests were run on an Ubuntu 22.04 VM (x86_64), with GCC's assembler and linker, measuring only functional correctness (no timing). Representative cases are shown below.

Test Case 1: Sparrow merge & repeat

```
Input:  Sparrow C C + C *
Output:
Sparrow Gen 0: C-C C
Sparrow Gen 1: C-C C C
```

Test Case 2: Warbler reduce & trill

```
Input:  Warbler D T C - H
Output:
Warbler Gen 0: D T
Warbler Gen 1: D T T
```

Test Case 3: Nightingale full duplication

```
Input:  Nightingale C D + * -
Output:
Nightingale Gen 0: C D C D
Nightingale Gen 1: C D C D C D C D
Nightingale Gen 2: C D C D C D C D
```

7 Discussion

We now reflect on our solution's correctness, performance characteristics, and limitations, and outline possible improvements that could further enhance robustness and maintainability.

Correctness & Robustness Our in-place transformation approach—using `temp_buf` for staging and full copies into `song_buf` on each generation—ensures that partial operations never corrupt the next iteration. We guard all merge/reduce/repeat routines with note-count checks (%r14), so invalid operations simply yield an unchanged generation, preserving stability.

Performance Each operator incurs a full 1024-byte memory copy. In the worst case of 50 operators, that is $50 \times 1024 = 51\ 200$ byte moves, which on modern hardware is sub-millisecond. The overall time complexity is $O(G \cdot B)$ where G is the number of generations (operators) and B the buffer length. For our target assignment sizes (< 256 tokens), runtime is negligible.

Limitations

- **Fixed buffer sizes:** All buffers are statically 1024 bytes. Extremely long inputs (over 1023 characters) are silently truncated.
- **No invalid-token detection:** Any character other than `C,T,D,+,-,*,H,space` is treated as a note or ignored, with no error message.
- **Code duplication:** Sparrow, Warbler, and Nightingale loops share large sections of copy logic. Maintenance would benefit from unified macros or a parameterized routine.

Possible Improvements

- Replace the repeated copy-and-print sequences with a single macro or shared subroutine that takes the species name as an argument.
- Dynamically calculate the exact length to `write` instead of always issuing 1024 bytes, saving a few syscalls.
- Add input validation and a fallback `.default_loop` that emits a "`Invalid token`" warning.

8 Conclusion

We have built a fully-functional GNU Assembly simulator of species-specific bird song evolution, satisfying all assignment requirements. Our solution cleanly parses a single input line, correctly applies merge, reduce, repeat, and harm/trill rules for Sparrow, Warbler, and Nightingale, and produces properly formatted "`<Species> Gen N: ...`" output for each generation. While performance and correctness are excellent for classroom-scale inputs, future work could focus on reducing code duplication, improving input validation, and supporting dynamic note sets. This project has strengthened our understanding of low-level buffer management, system call usage, and control-flow construction in x86_64 assembly, laying a solid foundation for more advanced systems programming tasks.

AI Assistants

AI assistants were used during this project strictly within the boundaries defined by the course policy. The use of such tools was limited to support and clarification purposes and did not replace independent implementation or understanding.

Throughout the development process, ChatGPT was employed in the following ways:

- **Debugging Assistance:** The assistant provided help in interpreting complex error messages and understanding subtle issues related to dynamic memory management and string manipulation in C. These insights were used to guide manual debugging efforts and code refinement.
- **Documentation Support:** ChatGPT was extensively used to enhance the clarity, coherence, and academic tone of the project's written report. While all content was originally authored by us, the assistant helped rephrase and formalize sections to ensure the language met academic standards.
- **Concept Clarification:** For reinforcing understanding of C programming concepts—such as pointer handling, and token parsing—ChatGPT was consulted as a learning aid. This helped solidify implementation decisions through clearer conceptual grounding.