# Project Report - Evochirp🐦

Systems Programming – Spring 2025

Üstün Yılmaz, 2023400108
Department of Computer Engineering
Boğaziçi University

May 11, 2025

**Abstract**

This document is about a project given as an assignment in the CMPE230 course of Boğaziçi University, called Evochirp. It has an introduction about what the project entails and what problems are encountered and solved during the implementation process. It also delves into what methodology was used in the implementation of the project and the consequences of that methodology.

## 1 Introduction

The project is about implementing three different interpreters which have four common operations. Each interpreter handles these operations differently and has a unique way of expressing its output. These interpreters represent birds in real life, and they behave like these birds, having their own songs and the evolution patterns corresponding to each operation, hence the name Evochirp.

## 2 Problem Description

The main problem in this project is to implement it using Assembly language, x86-64 GAS AT&T syntax. Assembly language is a very low level language in which you have to comment each line, otherwise you may get lost. It involves communicating directly with the CPU of a computer, which makes the project that can be implemented in Python in say, 250 lines, be implemented in 800 lines in Assembly language.

The secondary problem is that after parsing and processing the input accordingly, printing it out. The syscall operation corrupts the registers up until r12, and this can really be a pain in the neck as if you don't know details like these, you don't really know how to deal with the continuous segmentation faults you get.

The third problem is handling the constraints, but after you get used to the flow of it, the rest is very easy. The constraints are:

- Input tokens are strictly separated by a single space.

- The input line will contain at most 256 characters.
- The first token (species) will be either `Sparrow`, `Warbler` or `Nightingale`.
- All subsequent tokens will be valid notes (`C`, `T`, `D`) or operators (`+`, `-`, `*`, `H`).
- There will be no leading or trailing spaces in the input line.
- If an operator requires more notes than are currently available (e.g., merging two notes when only one is present), it will have no effect.
- The expression is otherwise guaranteed to be well-formed.
- Merged notes (e.g., `C-T`) are not considered valid operands for any operation. You may safely assume such inputs will not occur.
- Consecutive operations are allowed (e.g., `C T * * *`) as long as they do not violate the above rules.

# 3 Methodology

The approach used in the problem was pretty simple. Since the task does not require hardcore algorithmic thinking (as the time complexity is unimportant), I just implemented classic array-manipulation techniques and some memory tricks like BSS having adjacent elements.

I solved the main problem by looking up online and asking ChatGPT how to get input output and how to add/remove elements to an array in Assembly language. Contrary to what these sources recommended, I then implemented my own methods strictly using no call stack nor call and ret statements since it was told in the lecture jumps are enough. The pseudocode on how I handled these is below, at the Sample Code section.

# 4 Implementation

The implementation is simple. Take the line as input, parse it into tokens, process the parsed tokens (if it's a note, add it to the array; else, process it as an operator), and after each processed operator (even if it does nothing), print out the current generation until a newline is reached.

## 4.1 Code Structure

The structure of the code is, I avoided the use of call/ret as a whole to make things simpler. Everything is handled by jumps and "functions & subfunctions" are merely labels to jump to. Down below are the implementations of reading input, processing an operator (Sparrow with + here), and printing the output.

## 4.2 Sample Code

```
_start: # main function

leaq merge_buffer(%rip), %rax
movq %rax, merge_buffer_pointer(%rip)
leaq harmonize_buffer(%rip), %rax
movq %rax, harmonize_buffer_pointer(%rip)


movq $0, %rax # read mode
movq $0, %rdi # open standard input
leaq input_buffer(%rip), %rsi # load address of input_buffer to source input
movq $256, %rdx # max number of characters to read
syscall # read characters and put them to rax

test %rax, %rax # invalid input, will never be reached
jle tokenize_done

movq %rax, %rcx # store number of bytes read in rcx
leaq input_buffer(%rip), %rsi # load address of input_buffer to source input
addq %rcx, %rsi # go to the end of read line
movb $0, (%rsi) # null terminate the line

leaq input_buffer(%rip), %rsi # load address of input_buffer to source input
addq %rcx, %rsi # go to the end of read line
decq %rsi # go before the null terminator we put
movzbq (%rsi), %rax # load that character to rax
cmpb $10, %al # is it newline?
je strip_newline # if so strip it
cmpb $13, %al # is it newline (unix systems)?
jne parse_input # if not, skip stripping



strip_newline: # function to strip newline from input line
movb $0, (%rsi) # change newline to null terminator (lazy strip)



parse_input: # input parser function
xorq %rbx, %rbx # empty the token count
leaq input_buffer(%rip), %rsi # load address of input_buffer to source input
leaq token_array(%rip), %rdi # load address of input_buffer to destination input


next_character: # subfunction to traverse the input line
movzbq (%rsi), %rax # load contents of source input into rax
cmpb $0, %al # end of line
je tokenize_done

cmpb $' ', %al # skip space before reading token
je skip_space
```

```
52
53  movq %rsi, (%rdi, %rbx, 8) # store current character pointer when a token
        starts
54  incq %rbx # go to the next character (hence the name of the subfunction)
55
56
57  read_token: # subfunction to read a token
58  incq %rsi
59  movzbq (%rsi), %rax # load next byte to rax
60  cmpb $' ', %al # is it space?
61  je terminate_token # if so terminate the token
62
63  cmpb $0, %al # is it null terminator?
64  je tokenize_done # if so we are done since we replaced newlines etc with
        null terminator
65
66  jmp read_token # lazy while loop
67
68
69  terminate_token: # subfunction to terminate a token
70  movb $0, (%rsi) # overwrite the space with null terminator
71  incq %rsi # go to next token
72  jmp next_character # parse next token
73
74
75  skip_space: # subfunction to skip spaces
76  incq %rsi # go to the next token
77  jmp next_character # parse next token
78
79
80  tokenize_done: # function to parse the species
81  leaq token_array(%rip), %r11
82  movq %rbx, %r10
83  shlq $3, %r10
84  addq %r10, %r11
85  movq $0, (%r11) # null terminate array
86
87  movq token_array(%rip), %rsi # load the token array to source input
88  movzbq (%rsi), %rax # get the first token into rax
89  cmpb $'S', %al # first letter of Sparrow
90  je case_sparrow # go to the specific marker function for that bird
91  cmpb $'N', %al # first letter of Nightingale
92  je case_nightingale # go to the specific marker function for that bird
93  cmpb $'W', %al # first letter of Warbler
94  je case_warbler # go to the specific marker function for that bird
95
96
97  case_sparrow: # marker function for Sparrow
98  movq $0, %r13 # the value for Sparrow is 0 (i picked it)
99  jmp species_done # species part has been handled
100
101
102 case_nightingale: # marker function for Nightingale
103 movq $1, %r13 # the value for Nightingale is 1
104 jmp species_done # species part has been handled
105
106
107 case_warbler: # marker function for Warbler
```

```
108 movq $2, %r13 # the value for Warbler is 2
109
110
111 species_done: # function to parse other tokens
112 movq $0, gen_count(%rip) # initialize generation counter
113 movq $0, song_length(%rip) # initialize song length
114 movq $1, %rbx # initialize token counter
115
116
117 process_token: # function to process remaining tokens
118 leaq token_array(%rip), %r11
119 movq (%r11 ,%rbx ,8), %rsi
120 cmpq $0, %rsi # is it a null terminator?
121 je exit_evochirp # if so, exit the program
122
123 movzbq (%rsi), %rax # get the token to rax to inspect it if its a note or
    an operator
124 cmpb $'C', %al # if its a chirp note
125 je add_note # process it
126 cmpb $'T', %al # if its a trill note
127 je add_note # process it
128 cmpb $'D', %al # if its a deep call note
129 je add_note # process it
130
131 movb %al, %r9b # move operator to r9b for easier handling and debugging
132 cmpb $'+', %r9b # if its a plus operator
133 je handle_plus # process it
134 cmpb $'-', %r9b # if its a minus operator
135 je handle_minus # process it
136 cmpb $'*', %r9b # if its a star operator
137 je handle_star # process it
138 cmpb $'H', %r9b # if its a harmony operator
139 je handle_harmony # process it
140
141
142 add_note: # function to append a note to the current sequence
143 movq song_length(%rip), %rax # load the song length counter to rax
144 leaq song_array(%rip), %rdx # load the address of next note to rdx
145 movq %rax, %rcx # load the current song length to rcx
146 shlq $3, %rcx # logical shift of 3 bits i.e. multiply by sizeof(pointer) =
    8 to get the current index
147 addq %rcx, %rdx # get the location in the song_array
148 movq %rsi, (%rdx) # add the token (note) to that index
149 incq song_length(%rip) # increment song length
150 incq %rbx # increment token index
151 jmp process_token # return to while loop
152
153 ------------------------------------------------------------------------
154
155 sparrow_plus: # plus operator handler for Sparrow interpreter
156 movq song_length(%rip), %rax # load the song length counter to rax
157 cmpq $2, %rax # are there enough notes?
158 jl skip_operation # if not, do nothing
159
160 movq song_length(%rip), %rcx # load the current song length to rcx
161 decq %rcx # since we are merging two notes - 1st note
162 leaq song_array(%rip), %rdx # load the song array into rdx
163 movq (%rdx, %rcx, 8), %rdi # load the last note into destination input
```

```
decq %rcx # since we are merging two notes - 2nd note
movq (%rdx, %rcx, 8), %rsi # load the second last note into source input

subq $2, song_length(%rip) # delete the last two notes since we are merging them

movq merge_buffer_pointer(%rip), %rcx # load the merge buffer array into rcx
movb (%rsi), %al # load contents of first character into al
movb %al, (%rcx) # first character is the second last note
movb $'-', 1(%rcx) # second character is the dash
movb (%rdi), %al # load contents of last character into al
movb %al, 2(%rcx) # last character is the last note
movb $0, 3(%rcx) # null terminate the buffer

movq song_length(%rip), %rax # load the song length counter to rax
leaq song_array(%rip), %rdx # load the song array into rdx
shlq $3, %rax # multiply by 8
addq %rax, %rdx # go to the index of last note before the merged ones
movq %rcx, (%rdx) # store the merged note at that index
incq song_length(%rip) # increment the song length
addq $4, merge_buffer_pointer(%rip) # increment merge buffer pointer
jmp operation_done

------------------------------------------------------------------------

print_gen: # function to print generations
cmpq $0, %r13 # if its a Sparrow
je print_sparrow # process it
cmpq $1, %r13 # if its a Nightingale
je print_nightingale # process it
cmpq $2, %r13 # if its a Warbler
je print_warbler # process it

print_sparrow:
leaq bird_sparrow(%rip), %rsi # load "Sparrow" to source input
movq $7, %rdx # length of "Sparrow"
jmp print_common

print_nightingale:
leaq bird_nightingale(%rip), %rsi # load "Nightingale" to source input
movq $11, %rdx # length of "Nightingale"
jmp print_common

print_warbler:
leaq bird_warbler(%rip), %rsi # load "Warbler" to source input
movq $7, %rdx # length of "Warbler"
jmp print_common

print_common:
movq $1, %rax # write mode
movq $1, %rdi # standard output
syscall # print

leaq space(%rip), %rsi # put space to source input
movq $1, %rdx # length of space
movq $1, %rax
movq $1, %rdi
```

```
220 syscall # print it
221
222 leaq gen_string(%rip), %rsi # put "Gen " to source input
223 movq $4, %rdx # length of "Gen "
224 movq $1, %rax
225 movq $1, %rdi
226 syscall # print it
227
228
229 movq gen_count(%rip), %rax # load generation count to rax
230 leaq number_buffer_end(%rip), %rcx # prepare for itoa operation
231 xorq %r9, %r9 # empty r9 (we used it for operators before, now we use it
       for digit count)
232 cmpq $0, %rax # gen 0 is a special case, we don't want DIV:0 errors
233 jne itoa # go on with processing itoa operation
234 decq %rcx # go to the last letter of number_buffer
235 movb $'0', (%rcx) # make it '0'
236 movq $1, %r9 # '0' has 1 digit
237 jmp after_itoa
238
239 itoa: # integer to ascii
240 xorq %rdx, %rdx # empty rdx
241 movq $10, %r10 # divisor is 10 for decimal numbers
242 divq %r10 # now our quotient will go to rax and remainder to rdx
243 addb $'0', %dl # convert remainder to ascii
244 decq %rcx # go to the last letter of number_buffer
245 movb %dl, (%rcx) # make it that digit
246 incq %r9 # increment digit count
247 cmpq $0, %rax # is the whole number processed?
248 jne itoa
249
250 after_itoa: # finish integer to ascii and print the gen number
251 movq $1, %rax
252 movq $1, %rdi
253 movq %rcx, %rsi # pointer to first digit
254 movq %r9, %rdx # string length is the number of digits
255 syscall # print
256
257 leaq colon_and_space(%rip), %rsi
258 movq $2, %rdx
259 movq $1, %rax
260 movq $1, %rdi
261 syscall # print ": "
262
263 incq gen_count(%rip) # increment gen_count
264
265 movq song_length(%rip), %r12 # move number of notes to r12
266 xorq %r14, %r14 # empty the r14 register, will use as for loop index
267
268 note_loop: # print the notes one by one
269 cmpq %r12, %r14 # have we reached the end?
270 jge after_notes # if so, terminate loop
271 leaq song_array(%rip), %rdx # load song array to rdx
272 movq (%rdx, %r14, 8), %rsi # move current note to source input
273
274 xorq %r9, %r9 # empty the r9 register, will use as strlen
275 strlen: # compute length of note to be printed
276 movzbq (%rsi, %r9, 1), %rax # get the character at the nth index
```

```
277 cmpb $0, %al # is it a null terminator?
278 je after_strlen
279 incq %r9 # keep traversing
280 jmp strlen # loop
281
282 after_strlen: # print the note
283 movq $1, %rax
284 movq $1, %rdi
285 # implicit movq rsi to rsi here!
286 movq %r9, %rdx # length of note
287 syscall # print note
288
289 leaq space(%rip), %rsi
290 movq $1, %rdx
291 movq $1, %rax
292 movq $1, %rdi
293 syscall # print space
294
295 incq %r14 # increment loop index
296 jmp note_loop # loop
297
298 after_notes: # print newline and continue
299 leaq new_line(%rip), %rsi
300 movq $1, %rdx
301 movq $1, %rax
302 movq $1, %rdi
303 syscall # print newline
304
305 incq %rbx # increment token index for operators
306 jmp process_token # loop back to process the next note / operator
```

# 5 Results

The results were surprisingly accurate after fixing the truckload of segmentation fault errors :). Here are some challenging sample inputs and outputs I have tested my program with:

- Input: ”Warbler C D C T D C C T D D * * - H +”
  Output:
  Warbler Gen 0: C D C T D C C T D D D D
  Warbler Gen 1: C D C T D C C T D D D D D D
  Warbler Gen 2: C D C T D C C T D D D D D
  Warbler Gen 3: C D C T D C C T D D D D D T
  Warbler Gen 4: C D C T D C C T D D D D T-C

- Input: ”Nightingale C D T T D D C C C - - + * H”
  Output:
  Nightingale Gen 0: C D T T D D C C
  Nightingale Gen 1: C D T T D D C
  Nightingale Gen 2: C D T T D D C D C
  Nightingale Gen 3: C D T T D D C D C C D T T D D C D C
  Nightingale Gen 4: C D T T D D C D C C D T T D D C-C D-C

- Input: ”Sparrow C C C C C C D D D D D D - * * H D T +”
  Output:
  Sparrow Gen 0: C C C C C D D D D D D
  Sparrow Gen 1: C C C C C D D D D D D D
  Sparrow Gen 2: C C C C C D D D D D D D D
  Sparrow Gen 3: T T T T T D-T D-T D-T D-T D-T D-T D-T D-T
  Sparrow Gen 4: T T T T T D-T D-T D-T D-T D-T D-T D-T D-T D-T

# 6 Discussion

Performance was excellent compared to any other project I've done in my academic life, thanks to Assembly language. I didn't really experience any limitation other than the number of registers not being enough sometimes when syscalling (because of the clobbering of registers <r12 caused by the syscall).

A possible improvement would be to stick with one type of implementation when manipulating arrays, as I used shlq and movq to do pointer arithmetic in some of my functions, but I used addq and offset in others. As I documented clearly what I've done in each line, I used this approach merely to improve my own Assembly language skills and it does not really cause any drawbacks readability-wise.

# 7 Conclusion

The project was an excellent way of learning Assembly language, especially AT&T syntax. The task was really easy (as it can be implemented in Python sub 2 hours I would bet), the hard part was to do it using Assembly language.

As I've said, a future enhancement would be to improve the algorithm of the program and maybe make it accept multiple lines as input.

Who knows, maybe I can create an R2-D2 in real life by combining some scrap parts, an AI assistant, and Evochirp. :)

# AI Assistants

The two AI assistants I've used in this project were ChatGPT and DeepSeek R1. Note that I did not copy and paste any AI-generated code anywhere in my implementation and implemented the whole project myself (hence the inconsistencies :)). How I utilized them can be listed as:

- I asked for ChatGPT's help for taking and parsing an input line in Assembly language, with also some insights on how to use syscalls for input and output.
- I asked for DeepSeek R1's help when debugging segmentation faults and learned that syscalling may corrupt the contents of the registers <r12. This saved literally hours of my work.

- I asked for ChatGPT's help on how to do pointer arithmetic on Assembly language and how to add/remove certain elements of an array doing that.
- I used ChatGPT to help write a listings package that colors the sample code on Overleaf with respect to Assembly language.