Project 2 - Evochirp 🐦

CmpE 230, Systems Programming, Spring 2025

Instructor: Can Özturan
TAs: Gökçe Uludoğan
SA: Abdullah Rüzgar

Due: May 11, 2025, 23:59 (Strict)

1 Project Overview

In this project, you will implement a GNU assembly language program that **simulates bird song evolution**. Each bird species interprets and evolves songs differently, applying species-specific transformations to a sequence of notes using an evolution expression.
Birdsong plays a crucial role in communication, mating, and territory marking. Over generations, songs evolve due to learning, imitation, and environmental adaptation. In this simulation:

- Each **bird species defines unique transformation rules** for how songs evolve.
- The evolution process is governed by an **expression**, where each operator produces a new generation of the song.

Some operators apply to the most recent note(s) on the sequence, while others may transform the entire song. This distinction depends on the bird species and the operator used.

2 Bird Songs and Evolution

2.1 Song Elements

Each bird song begins as a sequence of basic sound symbols:

- `C` – **Chirp**: a soft, high-pitched, quick note.
- `T` – **Trill**: a moderately loud, oscillating call.
- `D` – **Deep Call**: a low, strong, resonant sound.

The song then evolves over generations as new notes emerge and operators are applied one-by-one.

2.2 Input Format

Your program will process a single input line structured as:

```
<Species> <Song Expression>
```

For example:

```
Sparrow C C + D T * H
```

2.3 Output Format

Your output should show the resulting song **after each generation**, in the following format:

```
<Species> Gen 0: <song after applying the first operator>
<Species> Gen 1: <song after applying the second operator>
...
```

Each generation corresponds to one operator from the expression.

3 Operators and Species Behavior

3.1 Operator Semantics

Each operator has a different interpretation depending on the bird species. The operator may apply to one or more recent notes, or in some species, to the entire sequence. The following table summarizes these behaviors:

Operator	Sparrow Interpretation	Warbler Interpretation	Nightingale Interpretation
`+` (Merge)	Merge two most recent notes: `X Y + → X-Y`	Combine two most recent notes into a melodic call: `X Y + → T-C`	Increase tempo: duplicate the most recent pair: `X Y + → X Y X Y`
`*` (Repeat)	Repeat the most recent note: `X * → X X`	Echo the two most recent notes: `X Y * → X Y X Y`	Repeat the entire song: `S * → S S` (entire sequence duplicated)
`-` (Reduce)	Remove the first occurrence of the softest note in the sequence	Remove the last note in the current sequence	Compress repeated notes by collapsing the most recent repetition (e.g., `X X - → X`)
`H` (Harmonize)	Apply global transformation: `C → T`, `T → C`, `D → D-T` on each note in the sequence	Extend the last sequence by appending a trill: `X H → X T`	Rearrange the last three notes into a nested form: `X Y Z H → X-Z Y-X` (e.g., `C T D H → C-D T-C`)

3.2 Softness Ranking of Notes

Some operations require comparing notes based on their softness. The following ranking is used, from softest to strongest: Chirp being softest, to Deep Call being strongest.

4 Evaluation Flow

1. Begin with an initial song (e.g., `C C D`).
2. Read the operators from left to right.
3. Apply each operator according to species-specific rules, producing one new generation per operator.
4. After each transformation, print the result in the format: `<Species> Gen N: ...`.

5 Example: Song Evolution

Input:

```
Sparrow C C + D T *
```

Output:

```
Sparrow Gen 0: C-C
Sparrow Gen 1: C-C D T T
```

Explanation:

- `+` merges two `C` into `C-C`.
- `D` and `T` are appended.
- `*` applies to `T → T T`.

Step-by-Step Song Evolution Across Species

The evolution of the same song across different species is provided in the table below, showing how each species interprets the same sequence of operators uniquely across generations.

Generation	Sparrow	Warbler	Nightingale
Gen 0	C-C	T-C	C C C C
Gen 1	C-C D T T	T-C D T D T	C C C C D T C C C C D T

6 Execution Instructions

Compile and run your assembly program with the following:

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

7 Testcases

Test Case 1: Moderate Complexity with All Operators

```
Warbler C T + D C * H -
```

Explanation Summary: Combines `C` and `T` into `T-C`, appends `D`, duplicates `D C`, harmonizes the last sequence by adding a trill, and removes the last note.

```
Warbler Gen 0: T-C
Warbler Gen 1: T-C D C D C
Warbler Gen 3: T-C D C D C T
Warbler Gen 4: T-C D C D C
```

Test Case 2: Heavy Repetition and Merging

```
Nightingale D C + T + * H
```

Explanation Summary: Two tempo-boosting merges expand the song; repeat operator duplicates the full sequence, and harmonization mirrors the last three notes.

```
Nightingale Gen 0: D C D C
Nightingale Gen 1: D C D C T C T
Nightingale Gen 2: D C D C T C T D C D C T C T
Nightingale Gen 3: D C D C T C T D C D C T-T C-T
```

Test Case 3: Softness-Aware Reduction

```
Sparrow D T C - * H
```

Explanation Summary: Reduce removes the softest note (C), * repeats the last note, H harmonizes: C→T, T→C, D→D-T.

```
Sparrow Gen 0: D T
Sparrow Gen 1: D T T
Sparrow Gen 2: D-T C C
```

Test Case 4: Long Sequential Harmony and Duplication

```
Warbler C D * T H H -
```

Explanation Summary:
Echo of last two, append trill twice (via H), then remove last.

```
Warbler Gen 0: C D C D
Warbler Gen 1: C D C D T T
Warbler Gen 2: C D C D T T T
Warbler Gen 3: C D C D T T
```

Test Case 5: Repeated Structure Expansion

```
Nightingale T C + C + * - H
```

Explanation Summary:
Two merges, song expands into long repeatable unit, then repeat entire sequence, reduce adjacent repeats, mirror last 3 notes.

```
Nightingale Gen 0: T C T C
Nightingale Gen 1: T C T C C C C
Nightingale Gen 2: T C T C C C C T C T C C C C
Nightingale Gen 3: T C T C C C C T C T C C C
Nightingale Gen 4: T C T C C C C T C T C-C C-C
```

8 Assumptions

- 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.

9 Submission

Your project will be tested automatically on the provided docker image. Thus, it's important that you carefully follow the submission instructions. Instructions for setting up the development environment can be found at this page. The root folder for the project should be named according to your student id numbers. If you submit individually, name your root folder with your student id. If you submit as a group of two, name your root folder with both student ids separated by an underscore. You will compress this root folder and submit it with the `.zip` file format. Other archive formats will not be accepted. The final submission file should be in the form of either `2020400144.zip` or `2020400144_2020400099.zip`.

You must create a Makefile in your root folder which creates a `evochirp` executable in the root folder, do not include this executable in your submission, it will be generated with the `make` command using the Makefile. The `make` command should not run the executable, it should only compile your program and create the executable.

Additionally, commit your progress and submit your project to the GitHub Classroom assignment to verify its structure and basic functionality. You can join the assignment using this invite link. Please commit your progress and submit your project to this assignment to verify its structure and basic functionality. For team formation, either create a team with your partner by combining your student IDs (e.g., 2022400XXX-2022400YY) or join the team that your partner has already set up.

Submission to GitHub allows you to automatically test your code within the provided Docker image environment. Each repository is allocated approximately **30 minutes** of GitHub Actions runtime. Although each individual workflow run is limited to a maximum of **10 minutes**, triggering the workflow more than **3 times** may consume your entire allowance.

Our organization has a total limit of **3000 GitHub Actions minutes**.
Please be cautious to avoid depleting your repository's minutes and preventing both yourself and others from testing their submissions.
To conserve minutes, you may manually trigger the grading workflow by navigating to the repository's `Actions` tab and selecting `Workflows → grade`.

Late Submission

If the project is submitted late, the following penalties will be applied:

Hours late	Penalty
0 < hours late <= 24	25%
24 < hours late <= 48	50%
hours late > 48	100%

10 Grading

Your project will be graded according to the following criteria:

- **Code Comments (10%):** Write code comments for discrete code behavior and method comments. This subgrading evaluates the quality and quantity of comments included in the code. Well-written comments are essential for understanding the purpose, structure, and logic of the code, especially when dealing with complex algorithms or data structures. Key expectations for comments:
  - **Method-level comments:** Provide a brief description of each function/method, including its purpose, parameters, return value, and any side effects.
  - **Inline comments:** Explain discrete code behavior, such as non-trivial logic, important variable assignments, or specific algorithm steps.
  - **Code structure clarity:** Use comments to mark logical sections of the code to improve readability.

  Proper documentation ensures that the code is easily readable and maintainable for future developers.
- **Documentation (15%):** A written document describing how you implemented your project. This subgrading assesses the quality and completeness of the written documentation accompanying the project. Good documentation should effectively communicate the project's purpose, design, and implementation details, ensuring that others can understand and build upon your work. Key expectations for documentation:
  - **Project Overview:** Clearly state the problem definition, objectives, and motivation behind the project.
  - **Design and Implementation:** Provide a structured explanation of the methodology, including any algorithms, flowcharts, or pseudocode used. Explain key implementation details with code snippets, where appropriate.

  - **Challenges and Solutions:** Describe any difficulties encountered during the project and how they were resolved.
  - **Usage Guide:** Include examples of input and output, along with instructions on how to compile, run, and test the program.
  - **Code Structure:** Explain the organization of the code, including the purpose of key functions, modules, or classes.

  Students should aim for concise, well-organized, and readable documentation. For formatting, follow the provided project report template in LaTeX, which demonstrates the expected structure.

- **Implementation and Tests (75%):** The submitted project should be implemented following the guidelines in this description and should pass the testing. This subgrading assesses the quality and correctness of the implementation.

  Grading the test cases will be straightforward: your score from the test cases will be determined by the formula:

  Total Correct Answers to Lines / Total Lines Given

11 Warnings

- You can submit this project individually or as a group of two.
- All source codes are checked automatically for similarity with other submissions and exercises from previous years. Make sure you write and submit your own code. Any sign of cheating will be penalized with an F grade.
- Project documentation should be structured in a proper manner. The documentation must be submitted as a pdf file and in raw text format (i.e. as a tex file).
- Make sure you document your code with necessary inline comments and use meaningful variable names. Do not over-comment or make your variable names unnecessarily long. This is very important for partial grading.
- Do not start coding right away. Think about the structure of your program and the possible complications resulting from it before coding.
- Questions about the project should be sent through the discussion forum on Piazza.
- There will be a 30 second time limit for your program execution. The execution of the program consists of the total time required for all queries. This is not a strict time limit, and the execution times may vary for each run. Thus, objections regarding time limits will be considered if necessary.

12 AI Assistants

If AI assistants were used during this project, the report must explicitly describe their usage. Clearly specify the extent of AI involvement to ensure transparency and maintain academic integrity.
The use of AI assistants (such as ChatGPT, Copilot, or similar tools) is permitted for limited purposes, including:

- Grammar and spelling correction in the documentation.
- Debugging assistance (e.g., understanding error messages, suggesting fixes).
- Clarifications on C programming concepts and best practices.

However, students must adhere to the following restrictions:

- **Do not use AI-generated content directly in your code or documentation.** Doing so will be viewed as plagiarism and thoroughly investigated.

- Ensure that all submitted work is your own. AI tools should be used as an aid, not as a replacement for understanding or implementation.
- Clearly document any AI usage in your report.

Failure to comply with these guidelines can result in academic penalties.