**Background**

Even if have played [tenpin bowling](https://en.wikipedia.org/wiki/Ten-pin_bowling) (called bowling for simplicity hereafter), it is likely that you are not 100% clear how scores are calculated, as there are misconceptions and exceptions, as described below. A detailed guide on how scores are calculated can be found [here](https://www.breakdownbowling.com/how-are-bowling-scores-calculated/).

In a bowling game, there are 10 turns that are called **frames**. For now, let's focus on frames 1-9. In each frame, you are given 2 attempts to knock down all the pins. We will refer to these attempts as **rolls**.

* If you knock down all the pins on the first roll, it is called a **strike** and is marked with an X.
* If you knock down all the pins in two rolls, it is called a **spare** and is marked with /.
* If you knock down fewer than 10 pins in the frame, you simply note down the number of the pins knocked down in each roll. This is called an **open frame**.
* If no pins are knocked down, it is marked with - instead of 0.
* If a player steps on or over the fault line on the bowling lane, they receive 0 points for that roll. This is typically marked with an F.

Both *spares* and *strikes* are awarded 10 points initially. What is the benefit of rolling a strike or a spare?

* A **strike** earns 10 points + the points of the next **two consecutive rolls**, regardless of which frames they occur in.
* A **spare** earns 10 points + the points of the next **one consecutive roll**.

Calculating the scores for spares and strikes requires a look-ahead. Additionally, calculating the score for a spare may require looking back at the current frame's rolls (this will become clearer when we review an example).

Now, let us see how the 10th frame differs. It is important to note that there is no look-ahead in the 10th frame. A strike is worth 10 points, and a spare is also worth 10 points. However, there are additional benefits for achieving either.

* If you roll an **open frame** (i.e you didn't knock down all the pins in two attempts), the same scoring rules as for **open frames** in frames 1–9 apply.
* If you roll a **spare**, you are awarded one extra roll.
* If you roll a **strike**, you are awarded two extra rolls.

Thus, the 10th frame can have up to three rolls in total, depending on the player's performance.

Let's make an example. The following game is provided and we need to compute the scores frame by frame:

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| | | | | | | | | | |

+-----------------------------------------+

**Frame 1:** This is an **open frame** and the score is simply calculated as the sum of the two rolls, 9 in this case:

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| | | | | | | | | |

+-----------------------------------------+

**Frame 2:** This is another **open frame** and the score is again calculated as the sum of the two rolls, 7 in this case:

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| | | | | | | | |

+-----------------------------------------+

[⚠️] The scores in the scoreboard are reported in a cumulative sum, this means that the score of frame[i] is summed to the score of frame[i-1] (if any). I use here an array syntax for ease of explanation and do not take this as an implementation suggestion. Hence 7+9=16

**Frame 3**: We have a *spare*, the score is 10 plus the next roll (first roll on Frame 4), which is a 5. The score for this frame is 15.

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| | | | | | | |

+-----------------------------------------+

**Frame 4**: We have another *spare*, the score is 10 plus the next roll (first roll on Frame 5), which is a 8. The score for this frame is 18.

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| | | | | | |

+-----------------------------------------+

**Frame 5:** This is another **open frame**, 8

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| | | | | |

+-----------------------------------------+

**Frame 6**: we have a *strike*. The score is 10 plus the next two rolls. The first 'next' roll is 5 and the second 'next' roll is /. A spare is worth 10. Therefore, the score for this frame is 20.

[⚠️] The look-ahead is **NOT** recursive. Although the strike is followed by a spare (the second roll of Frame 7), we don't look ahead of that spare.

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| 77| | | | |

+-----------------------------------------+

**Frame 7**: We have a *spare*, the score is 10 plus the next roll (first roll on Frame 8), which is a X. As the strike is worth 10, the score for this frame is 20.

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| 77| 97| | | |

+-----------------------------------------+

**Frame 8**: we have a *strike*. The score is 10 plus the next two rolls. The first 'next' roll is X (first roll on Frame 9), and the second 'next' roll is 4 (first roll on Frame 10). The score for this frame is 24

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| 77| 97|121| | |

+-----------------------------------------+

**Frame 9**: we have a *strike*. The score is 10 plus the next two rolls. The first 'next' roll is 4 and the second 'next' roll is /. A spare is worth 10. Therefore, the score for this frame is 20.

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| 77| 97|121|141| |

+-----------------------------------------+

**Frame 10**: We have three rolls. Let's see roll by roll:

* Roll 1: We have a 4 and we will keep it as is.
* Roll 2: We have a spare. Now we have a / symbol, but how much is it worth? Mathematically, it's 6 (calculated as 10 - previous\_roll is an implementation trick that I recommend to use).
* Roll 3: We have a *strike*, so it's just 10. The total of this frame is 4+6+10=20

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |4 / X|

| 9| 16| 31| 49| 57| 77| 97|121|141| 161|

+-----------------------------------------+

As mentioned in the last frame example, calculating the spare as 10 - previous\_roll is an implementation trick that can be used for any circumstance shown above and may be useful to keep it in mind. I showed you in the last instance because it was better to focus on how scoring works first and then provide you some hints on the implementation later on.

Furthermore, you can use [this website](https://bowlinggenius.com/) to practise and make a few examples yourself.

For completeness, the [perfect game](https://en.wikipedia.org/wiki/Perfect_game_(bowling)) is achieved by rolling 12 strikes (a strike for each of frame between 1-9, and three more strikes in the last frame) and it's worth 300.

**Task 1**

**Overview**

Your task is to complete the implementation of the function bowling\_score\_parser and print\_scoreboard. The first function parses a game string and should return a linked list of frames. The second function prints a scoreboard (details below) given a linked list of frames as parameter.

**Details**

You are given the following files:

* linked\_list.c, linked\_list.h, queue.c, queue.h: These files are identical to those used in previous assignments.
* tpb.c: this is the file you should edit and implement your code. Further details about its contents are provided below. This files comes with a header file tpb.h required by the next file.
* score\_calculator.c: This is a pre-implemented main file. One or more games are provided as command-line arguments and passed to the bowling\_score\_parser function, implemented in tpb.c. The main file prints either the scoreboard (details below) or an error (details also below).
* Makefile to run tests on both tasks for this coursework and it's organised as seen in previous assignments.

You should not change any of the provided files, **except** tpb.c.

Your task is to **parse** a string representing a game and calculate the scores for each frame, as illustrated earlier. If we consider the game shown in the example, this will be represented with the following string: 547-9/5/8-X5/XX4/X. The major difference between this task and the Task 4 of the last coursework is that summations (referred to as **reductions**hereafter) are performed without operators. For example:

* For a strike, reduce immediately and look ahead (if applicable).
* For a spare, reduce immediately and look ahead (if applicable).
* For two rolls, reduce after parsing them.
* For the last frame, reduction may involve up to three rolls.

Most of the implementation will occur in the bowling\_score\_parser and print\_scoreboard functions:

* bowling\_score\_parser: This function takes a string (const char \*) and a pointer to int as input and returns a LinkedList \* containing parsed frames. The function must also handle incomplete games (more details below).
* print\_scoreboard: This function, also invoked in the main file, prints the game string followed by the scores for each frame. For example:

$ ./task1 547-9/5/8-X5/XX4/X

547-9/5/8-X5/XX4/X: 9 16 31 49 57 77 97 121 141 161

The print\_scoreboard function does not access the command line directly. It operates solely on the linked list produced by bowling\_score\_parser. This serves also as a way to validate your parsing.

If the program is called with multiple games, the following output is expected (note that the 3rd game is intentionally incomplete):

$ ./task1 7-35xxxxxx716- x7/xx637/x6/xxx7 xx8-6-

7-35xxxxxx716-: 7 15 45 75 105 135 162 180 188 194

x7/xx637/x6/xxx7: 20 40 66 85 94 114 134 154 184 211

xx8-6-: 28 46 54 60

You can also note that the parsing should be case insensitive, i.e. x and X, and f and F should be treated equally. A function called tokenization is already implemented for you and handles these cases too.

In the case of an error, the second parameter of bowling\_score\_parser (a pointer to int) should be set to the **zero-based position** in the string where the error occurred. The error message is already implemented for you in score\_calculator.c but relies on the position your implementation provides. Examples of incorrect input and their output are as follows:

$ ./task1 7-35xxxxxx716-/ x7/xx637xx6/xxx7 x/x8-6-

Invalid token near 15 (token found /)

Invalid token near 9 (token found x)

Invalid token near 2 (token found /)

Explanation of the errors:

* **Game 1:** The 3rd roll in the 10th frame is invalid because that frame can only have two rolls.
* **Game 2:** A strike appears as the second roll in a frame (frames 1–9). This is not allowed.
* **Game 3:** A spare follows a strike, which is not permitted.

To handle all the cases and exception, you can help yourself with the following parsing grammar:

<game> ::= <frames> | <frames> <tenth-frame>

<frames> ::= <frame> | <frame> <frames>

<frame> ::= <strike> | <spare-frame> | <open-frame>

<strike> ::= "X"

<spare-frame> ::= <roll> "/"

<open-frame> ::= <roll> <roll>

<tenth-frame> ::= <strike> <bonus-roll> <bonus-roll>

| <spare-frame> <bonus-roll>

| <strike> <spare-frame>

| <open-frame>

<roll> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "-" | "F"

<bonus-roll> ::= <strike> | <roll>

Your implementation does not need to specify the nature of the error, only its position. The provided code handles error reporting for you.

Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. As discussed in lectures, a good tool for assessing if a program has handled dynamic memory allocation correctly is valgrind. To check your implementation using valgrind you can type:

$ make task1\_test\_memcheck

If your program has correctly handled dynamic memory allocation, the last line of output should read:

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

**Implementation hints**

This Task may be challenging, so it's important to carefully follow the suggested implementation steps. The core function, bowling\_score\_parser, should be structured into the following key sections. If you follow these guidelines, your implementation will align with the provided framework.

* **Section 1 - Initialisation:** Initialise all the data structures required for the function. For example:
  1. Define variables to track the state of all the sections you'll see below
  2. Allocate and initialise:
     + A queue to store tokens generated during lexical analysis (next section).
     + An array of linked lists (frame\_list), one for each frame. Use FRAME\_NUMBER to define the size of this array. This array can be statically allocated. This array of lists will become handy in *Section 3*.
     + A linked list called scoreboard to hold the final parsed and validated frames to be returned.
* **Section 2 - Lexer:** Perform lexical analysis on the input string game\_characters:
  1. Iterate over the characters in the string:
  2. For each character, use the provided tokenization function to classify it as a valid token.
  3. If tokenization fails (returns false), record the error position inside the 2nd parameter err\_position and stop further processing. Keep in mind this second parameter is a pointer.
  4. Otherwise, assign the token's position (pos property of the struct Lex) and add it to the queue of tokens (use push\_queue function).
* **Section 3 - Parser:** Parse the tokens into an intermediate representation use:
  1. Pop tokens from the queue one by one.
     + For each token, append it to the linked list corresponding to the current frame in frame\_list.
  2. Handle look-ahead tokens:
     + For each token, check its look-ahead value (unless it's the last frame).
     + Copy the required number of look-ahead tokens from the queue into the linked list for the current frame.
  3. Transition frames:
     + After processing the rolls for a frame, determine whether to move to the next frame:
     + Advance to the next frame if the current frame contains enough rolls (accounting for strikes and spares).

[ℹ️] The array of linked lists frame\_list aims to implement a data structure resembling an **Abstract Syntax Tree** ([AST](https://en.wikipedia.org/wiki/Abstract_syntax_tree)). An AST is a typical data structue in parsing. However, we don't need its complexity for this task, because we know we have 10 frames and we also know that we have elements of the same type all times (i.e., we don't need to differentiate between literals or operators). If we consider the example 547-9/5/8-X5/XX4/X, this data structure will look like this at the end of this section:

Array of Lists

+---------+

| List #0 | -> [5] -> [4]

+---------+

| List #1 | -> [7] -> [-]

+---------+

| List #2 | -> [9] -> [/] -> (5)

+---------+

| List #3 | -> [5] -> [/] -> (8)

+---------+

| List #4 | -> [8] -> [-]

+---------+

| List #5 | -> [X] -> (5) -> (/)

+---------+

| List #6 | -> [5] -> [/] -> (X)

+---------+

| List #7 | -> [X] -> (X) -> (4)

+---------+

| List #8 | -> [X] -> (4) -> (/)

+---------+

| List #9 | -> [4] -> [/] -> [X]

+---------+

Square brackets were used to indicate rolls in the frame, parenthesis to indicate look-ahead rolls. If you follow these instructions, the next section will be implemented easily.

* **Section 4 - Reducer:** Build the final scoreboard from the parsed data:
  1. Validate frames:
  2. Use helper functions (e.g. validate\_frame and validate\_last\_frame) to verify the structure of each frame in the linked list.
  3. If validation fails, set the error position and stop processing (this is why the Lex struct has the pos property).
  4. Calculate scores:
  5. For each frame, initialise a Frame struct to store its rolls, score, and cumulative score.
  6. Traverse the linked list for the frame and compute:
     + The score for the frame, including look-ahead rolls if applicable.
     + The cumulative score, by adding the current frame's score to the previous frame's cumulative score (if any).
  7. Update scoreboard:
  8. Append the Frame struct to the scoreboard linked list.
* **Section 5 - Epilogue:**
  1. Free resources:
  2. Deallocate memory for the frame\_list linked lists.
  3. If an error occurred, also deallocate the scoreboard linked list.
  4. Return:
  5. If there were no errors, return the scoreboard.
  6. In the case of errors, return NULL.

[⚠️] In the case of incomplete games where the last played frame requires a look-ahead, simply consider that frame as 10 (it's either a *strike* or a *spare*). If *Section 4* is implemented correctly, your code will already handle this feature.

An example of an allowed incomplete games:

#1 #2 #3 #4 #5 #6 #7 #8 #9 #10

+-----------------------------------------+

|5 4|7 -|9 /|5 /| | | | | | |

| 9| 16| 31| 41| | | | | | |

+-----------------------------------------+

^

|\_\_\_\_\_\_\_ We don't know what comes next, but we do know that Frame 4 is at least worth 10 points.

You can add to the file tpb.c any helper functions you need, as long as you don't edit tpb.h.

**File tpb.c structure**

The tpb.c file contains essential code and utilities needed for your implementation. Below is a breakdown of its components:

1. **Includes and Macros**: The file begins with a series of #include directives. You may add more as needed, e.g., string.h, as long as they belong to the Standard library of C (except the maths library, which you are not allowed to use). There is also a custom definition of a bool type using typedef, along with true and false macros (this was shown in a lecture):

#define true 1

#define false 0

typedef int bool;

This approach ensures compatibility with the ANSI C standard.

1. **Constants and Enums**: A #define statement for FRAME\_NUMBER (set to 10) is included to represent the fixed number of frames in a bowling game. An enum called Token defines the possible types of tokens used in the game:

typedef enum {

TOKEN\_LITERAL, /\* Number 0-9 \*/

TOKEN\_SPARE, /\* '/' symbol \*/

TOKEN\_STRIKE, /\* 'X' or 'x' symbol \*/

TOKEN\_GUTTER, /\* '-' symbol \*/

TOKEN\_FAULT /\* 'F' or 'f' symbol \*/

} Token;

[ℹ️] An enumeration contains constant integer values (the first one is typically internally initialised with 0) and also defines a **type**. Although internally everything is treated as int, enumerations are a type-safe alternative to #define statements. You can read more [here](https://www.geeksforgeeks.org/enumeration-enum-c/).

1. **Structures**: The Lex structure is used to represent each token in the game string. Its fields are:

* token: The type of token (e.g., TOKEN\_STRIKE). The type is the Token enum shown above.
* character: The actual character from the input.
* look-ahead: Number of rolls to consider for scoring look-ahead (e.g., 2 for a strike).
* pos: The position of the character in the input string. Example:

typedef struct {

Token token;

char character;

int lookahead;

int pos;

} Lex;

1. **Functions:** The file provides several functions to assist with the parsing and scoring process:

* free\_scoreboard: Frees all memory allocated for the linked list of frames (scoreboard). *Recommended usage:*Call this function whenever an error occurs. The main function also invokes free\_scoreboard.
* get\_token\_value: Computes the numeric value of a given token. Special handling for *spares*: If the token is a spare /, it calculates the score based on the previous roll. Example usage (assuming both parameters are pointers):

int value = get\_token\_value(current\_token, previous\_token);

* tokenization: Converts a character into a Lex structure with its corresponding token type, character, and lookahead value. Returns true if the character is valid, false otherwise. Example usage:

Lex lexchar;

if (tokenization('X', &lexchar)) {

/\* Token is valid and lexchar is initialised \*/

}

else

{

/\* handle error \*/

}

* bowling\_score\_parser: The main function you will implement, responsible for parsing the game string into a linked list of frames (scoreboard). It follows five main sections: **Initialisation**, **Lexer**, **Parser**, **Reducer**, and **Epilogue**. Returns NULL in case of an error, or the constructed LinkedList for the scoreboard if successful.
* print\_scoreboard: A function for printing the scoreboard (to be implemented). It should traverse the linked list and display frame information.

**Additional information**

You are free to choose whichever method you want to implement your program. You can add any helper functions you need and may even edit the structs in tpb.c (as you think more appropriate) with the following restrictions:

* you are not allowed to use global variables
* all dynamically allocated memory should be freed
* the function bowling\_score\_parser must return a linked list of Frame structs.
* the function print\_scoreboard must take a linked list of Frame structs as parameter. The last two constraints derive from the fact you are not allowed to change tpb.h.

As stated above, you can use any standard library functions (except the maths library).

**Task 2**

**Overview**

In this task, you will extend the print\_scoreboard function to print a visually formatted scoreboard. The scoreboard should produce output the same as the provided examples *below*, with rows displaying frame rolls and their cumulative scores, enclosed within a nicely formatted table. The function must handle both complete and incomplete games, as well as cases with multiple games passed as input. An an example is below:

$ ./task2 547-9/5/8-X5/XX5/X

+-----------------------------------------+

|5 4|7 -|9 /|5 /|8 -|X |5 /|X |X |5 / X|

| 9| 16| 31| 49| 57| 77| 97|122|142| 162|

+-----------------------------------------+

The first row and last row are border of a table made by mostly - (minus) characters. Just the first and last one are + (plus) characters. The second line should print each roll for each frame (if a roll is missing, a should be used instead). The third row prints the score for each frame aligned to the right: this can be simply achieved by using  %3d (or %5d for the last frame) inside your printf format string. This will ensure the printed integer occupies 3 (or 5) characters with a right alignment using spaces to pad if needed.

The following example show you what happens with multiple or incomplete games:

$ ./task2 7-5/xxxxxx71xxx xxxxxxxxxxxx xx8-6-7/

+-----------------------------------------+

|7 -|5 /|x |x |x |x |x |x |7 1|x x x|

| 7| 27| 57| 87|117|147|174|192|200| 230|

+-----------------------------------------+

+-----------------------------------------+

|x |x |x |x |x |x |x |x |x |x x x|

| 30| 60| 90|120|150|180|210|240|270| 300|

+-----------------------------------------+

+-----------------------------------------+

|x |x |8 -|6 -|7 /| | | | | |

| 28| 46| 54| 60| 70| | | | | |

+-----------------------------------------+

**Details**

The main function, that must not be changed, invokes the function  print\_scoreboard for each game parsed from the command line. Therefore you have one function but need it to behave in two different ways, depending if your code is compiled for Task 1 or Task 2. If you have a look at the provided Makefile, you will notice that, for Task 2, the file tpb.cis compiled with an extra flag -DSCOREBOARD. This is a hint to use conditional compilation (as seen in [Week 6](https://moodle.nottingham.ac.uk/mod/page/view.php?id=7664819)). If the code is compiled without -DSCOREBOARD, the function print\_scoreboard should behave as in Task 1. If the code is compiled with -DSCOREBOARD, the function print\_scoreboard should print a table with the scoreboard as shown in this task.

If you don't remember what the command-line argument -DSCOREBOARD for gcc is, the lecture on Week 6 shows a live example and you can rewatch the video.

Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. As discussed in lectures, a good tool for assessing if a program has handled dynamic memory allocation correctly is valgrind. To check your implementation using valgrind you can type:

$ make task2\_test\_memcheck

If your program has correctly handled dynamic memory allocation, the last line of output should read:

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

The same rules as those for Task 1 apply for this task.