CIS*2750
Assignment 1
Deadline: Monday, January 29, 11:59pm
Weight: 12.5%

1. Description

In this assignment, you need to implement a library to parse the GEDCOM files. The link to the format description is posted in the Assignment 1 description and in course notes. Make sure you understand the format before doing the assignment.

Your assignment will be graded using an automated test harness, so you must follow all requirements exactly, or you will lose marks.

This assignment is individual work and is subject to the University Academic Misconduct Policy. See course outline for details.

According to the GEDCOM specification, a GEDCOM structure (which we will refer to as a GEDCOM object) contains:
- Exactly one header. The header must contain exactly one reference to a submitter record.
- 1 or more records of different types. Exactly one trailer record (file terminator)

The GEDCOM structure structure is represented by the `GEDCOMobject` type in `GEDCOMparser.h`.

There are seven different record types. Some may appear multiple times in a GEDCOM file, while others may appear only once per file. For the purposes of this assignment, we will focus on four records:
- Header
- Submitter
- Family record
- Individual record

These records are represented by `Header`, `Submitter`, `Family`, and `Individual` types. All of them are also defined in `GEDCOMparser.h`.

If your parser encounters any other record types, it should simply ignore those records.

You'll notice that each record has its own restrictions. Some are completely flexible, while others must include several parameters. Our types have the most relevant fields as struct members. Other fields will be saved as generic `Field` records in a list.

Multi-line content

The GEDCOM file consists of individual lines (gedcom_lines in the specification), which can indicate the start of a record, a field in a record, etc.. You'll notice that long individual line (field) values can be broken into multiple lines using CONT/CONC tags.

For example, the GEDCOM 5.5.1 format file contains the following address for the submitter record:

```
0 @5@ SUBM
1 NAME Reldon /Poulson/
1 ADDR 1900 43rd Street West
2 CONT Billings, MT 68051
1 PHON (406) 555-1232
```

You must store the complete value in the `address` field of the `Submitter` record. So for the above example, the `Submitter` record would be:
- `name = "Reldon Poulson"`
- `address = "1900 43rd Street West\nBillings, MT 68051"`

You will also notice that the format is flexible when it comes to line terminators. Your parser must be able to handle them as specified in the GEDCOM documentation.

2. Required Functions

2.1 GEDCOM functions

```
GEDCOMerror createGEDCOM(char* fileName, GEDCOMobject** obj);
```

This function does the parsing and allocates a GEDCOM object. It accepts a filename and an address of a GEDCOMobject pointer (i.e. a double pointer). If the file has been parsed successfully, the calendar object is allocated and the information is stored in it.

The return type is an `GEDCOMerror` struct. The `type` field indicates the error type using the `ErrorCode` enum, while the `line` filed indicates the line in the file containing the error (if applicable).

If `createGEDCOM` successfully creates a GEDCOM object, it returns `GEDCOMerror` with type `OK` and line `-1`. However, the parsing can fail for various reasons. In that case, the `obj` argument is set to NULL and the function returns a struct with the appropriate error code (type) and line number:
- `INV_FILE` is returned if there's a problem with `file` argument - its null, it's an empty string, file doesn't exist or cannot be opened, file doesn't have the `.ged` extension, etc. The `line` field must be -1.

  If the error is in the file contents, you must use an appropriate error code - see below.

- `INV_GEDCOM` is returned if the **GEDCOM object itself** is invalid. The `line` field must be -1. For example, you would return this code if :
  - the file is missing the header
  - the file is missing the closing tag
  - the file contains no records
  - etc.

  If the error is in one of the records contained within the GEDCOM object, return a component-specific error code instead (see below).

- `INV_HEADER` is returned if the header is present, but is somehow invalid, e.g.
  - Grammar error in header (`line` = first error that contains invalid grammar). This includes misspelled tags, invalid tags, tags with missing values, missing/incorrect numbers at the start of the line, etc.
  - Header missing a reference to submitter record or any other required filed (`line` = last line of header)
  - etc.

- `INV_RECORD` is returned if some record (other than header) is present, but is somehow invalid. In future assignments, we may expand this to include different error codes for different records - this is normal in iterative development. However, for now we will keep things simple. For example, you would return this code if the file has:
  - Grammar error in record (`line` = first error that contains invalid grammar). This includes misspelled tags, invalid tags, tags with missing values, missing/incorrect numbers at the start of the line, etc.
  - Record missing a required filed (`line` = last line of record)

- etc.

- `OTHER_ERROR` is returned if some other, non-GEDCOM error happens (e.g. malloc returns NULL).

Note: it is possible that a record in a file is valid, but does not contain one or more fields that we have in our record structs. This is NOT an error. For example, an individual record may be missing the given name. If you encounter this, simply use the empty string as the value for that field. The returned error code should be `OK`. If a family is missing the husband, set the husband reference to NULL. The returned error code should be `OK`.

If the missing field is actually a collection of things - e.g. children filed in the family record - initialize the list, but keep it empty. The returned error code should be `OK`.

```
char* printGEDCOM(const GEDCOMobject* obj);
```

This function returns a humanly readable string representation of the entire GEDCOM object. Sample output will be added to the assignment description. It must not modify the calendar object in any way. The function must allocate the string dynamically.

```
void deleteGEDCOM(GEDCOMobject* obj);
```

This function deallocates the object, including all of its records, lists, etc..

```
char* printError(GEDCOMerror err);
```

This function returns a string based on the `GEDCOMerror` value to make the output of your program easier to understand - i.e. “OK” if `err.type` is `OK`, “INV_FILE” or “invalid file” is `INV_FILE` was passed, "invalid record (line 20)" if `err.type=INV_RECORD` and `err.line = 20`, etc.. The function must allocate the string dynamically.

```
Individual* findPerson(
                    const GEDCOMobject* failyRecord,
                    bool (*compare)(const void* first, const void* second),
                    const void* person);
```

This function is designed to find an individual in a GEDCOM object. We might wish to search by surname (list name), given name+surname, given name+surname+spouse's name, etc.. The function should modify the GEDCOM object in any way. To allow for this customization, the function takes a custom comparator function and a custom search records. See the function description in `GEDCOMparser.h` for more details. Return a pointer to the record if found, NULL otherwise.

```
List getDescendants(const GEDCOMobject* failyRecord,
                    const Individual* person);
```

This function returns a newly created List of Individual records of all descendants of a given individual. Make sure the list contains **copies** of the Individual records, rather than the references to actual records. This way, if the user clears the list, he or she does not blow away a portion of our GEDCOM object! Return an empty list of no descendants found.

2.2 List helper functions

In addition, you must provide implementations for delete/compare/print functions for every type that we store in a list, i.e.
- `Event`
- `Individual`
- `Family`
- `Field`

The headers for these functions are provided in `GEDCOMobject.h`. These functions are necessary for the test harness to work correctly.

The the `print…` functions return a newly allocated string with a humanly readable representation of each record.

The `delete…` functions free the records passed to them - `freeField()` frees a `Field` argument, `freeFamily()` - Family argument, etc..

The compare… functions must return an int that indicates the relative ordering of the arguments, similar to `strcmp()`:
- `compareEvents()` compared by event date. It returns -1 if first argument has an earlier date than second, 0 if dates are the same, 1 if first argument has a later date then second,. If one of the arguments is missing a date, perform lexicographical comparison using the `type` field.
- `compareIndividuals()` performs lexicographical comparison on concatenated `givenName` and `surname` fields. Use `","` to separate the fields when concatenating them. For example, if the first argument is the Individual record for Bugs Bunny and the second one is the Individual record for Elmer Fudd, we're comparing "`Bunny,Bugs`" to `"Fudd,Elmer"`.
- `compareFields()` performs lexicographical comparison on concatenated `tag` and `value` fields, using " " (a single whitespace) as a separator.
- `compareFamilies()` compares by the number of family members. It returns -1 if first argument has fewer family members than second, 0 the number is the same, and 1 if first family has mome members than second.

3. Additional guidelines and requirements

While the above functions are required, you will need to write a number of helper functions. For example, it is strongly recommended that you write additional helpers functions for parsing the file - e.g. parsing each record type.

All functions (required and not) **must** be in a file called `GEDCOMparser.c`. For your own test purposes, you will also want to code a main program in another `.c` file that calls your functions with a variety of test cases, but you won't submit that program. **Do not put your main() function in `GEDCOMparser.c`, or else the test program will fail due to multiple definitions of main(); you will lose marks for that.**

Applications will #include `GEDCOMparser.h` in their main program. A basic `GEDCOMparser.h` has been provided for you. You must customize the file header with your student name and number. However, **do not** change anything else, or else your utilities will not compile with the test program. Also, do not add any other functions, types, or headers. `GEDCOMparser.h` is the public "face" of our calendar API. All the helper functions are internal implementation details, and should not be visible to the end user.

You are free to create your additional "helper functions" in `GEDCOMparser.c`, each with its proper function header, if you find some recurring processing steps that you would like to factor out into a single place. They must be in a separate header file, since they are internal to your implementation and not for

public users of the utility package. You may put these functions in a separate file called `GEDCOMutilities.c`

Your functions are supposed to be robust. They will be tested with various kinds of invalid data and must detect problems without crashing. However, be sure to follow the specification regarding return values. If your helper functions encounter a problem, they must free all memory and return the appropriate `error` value.

Libraries are written with very specific requirements. They must have a clear public API, and they must communicate errors to the higher-level modules, which will then handle these errors. They should definitely **not** print their own error messages!

Important points

- **Do not** put any internal helper function headers into `GEDCOMparser.h`
- **Do not** not change the given typedefs or function prototypes `GEDCOMparser.h`
- **Do not** submit any `main()` functions
- **Do not** exit the program from one of the parser functions if a problem is encountered, return an error value.
- **Do not** print anything to the command line.
- **Do not** assume that your pointers are valid. Always check for NULL pointers before dereferencing them.

4. Submission structure.

The submission must have the following directory structure:
- `assign1/` contains the README file and the `Makefile`
- `assign1/bin` should be empty, but this is where the `Makefile` will place the static lib files
- `assign1/src` contains `GEDCOMparser.c`, `LinkedListAPI.c`, and `GEDCOMutilities.c` (if you need it)
- `assign1/include` - contains `GEDCOMparser.h`, `LinkedListAPI.h`, and any additional header files that you might have.

Makefile

You will need to provide a `Makefile` with the following functionality:
- `make list` creates a static library `liblist.a` in `assign1/bin`
- `make tree` creates a static library `libtree.a` in `assign1/bin`
- `make parser` creates a static library `libparse.a` in `assign1/bin`
- `make` or `make all` creates `liblist.a`, `libtree.a`, and `libparse.a` in `assign1/bin`
- `make clean` removes all `.o` and `.a` files

5. Evaluation

Your code must compile, run, and have all of the specified functionality implemented. Any compiler errors will result in the automatic grade of **zero** (0) for the assignment. Make sure you compile and run the assignment on the SoCS Linux server before submitting it - this is where it will be graded.

Marks will be deducted for:
- Incorrect and missing functionality
- Deviations from the requirements
- Run-time errors

- Compiler warnings
- Memory leaks
- Bad / inconsistent indentation
- Bad variable names
- Insufficient comments
- Failure to follow submission instructions

I will post a rough marking scheme with grade breakdowns later.

6. Submission

Submit your files as a Zip archive using CourseLink. File name must be be `A1FirstnameLastname.zip`.

Late submissions: see course outline for late submission policies.