Assignment Description

TeX (or it's later evolution LaTeX) is a program used often in academia to write technical papers and documents. Users define macros in text files that also contain the contents of the document, and TeX processes macros and string expands them based on their definitions.

For this assignment you will implement a TeX-like macro processor in C. This macro processor will perform a transform on a set of input files **(or the standard input when no files are specified)** and output the result to the standard output. As the input is read, your program will replace macro strings according to the macro's value mapping and the macro expansion rules. Any input file(s) are provided as command line arguments. The execution of your program on the command line should have this form:

```
proj1 [file]*
```

Macros always start with an unescaped backslash followed by a name string. Macros have optional arguments. Each argument is placed in curly braces immediately after the macro name. For example:

```
\NAME{ARGUMENT 1}{ARGUMENT 2}{ARGUMENT 3}
```

Here's a brief example of an input/output execution:

Input	**Output**
`A list of values:`	`A list of values:`
`\def{MACRO}{VALUE = #}`	
`\MACRO{1}`	`VALUE = 1`
`\MACRO{2}`	`VALUE = 2`
`\MACRO{3}`	`VALUE = 3`
`\MACRO{4}`	`VALUE = 4`
`\MACRO{5}`	`VALUE = 5`
`\MACRO{6}`	`VALUE = 6`

`\MACRO{7}`	`VALUE = 7`

The `\def` macro defines a new macro called `\MACRO`, Future occurrences of `\MACRO` will be replaced with `VALUE = #` where the `#` is replaced with the argument to `\MACRO`. We will go into more detail on `\def` in the section below.

Some notes about the general macro grammar:

- Macro names must only contain a string of letters, numbers.
- No white space is allowed between a macro name and the arguments or between the arguments.
- With a few exceptions, macro arguments can contain arbitrary text, including macro expressions or fragments of macro expressions. The exceptions are for built-in macros, see the section below.
- Macro arguments must be brace balanced (i.e., the number of left braces is greater than or equal to the number of right braces in every prefix and equal in the entire string). For example:
  `\valid{this arg {{ is }} balanced}`
- With a few exceptions (see the built-in macros below), macro arguments can contain escape characters (backslashes). The details on how escape characters should be handled is given in a section below.

Built-In Macros

You will need to implement a set of built-in macros in the macro processor you are building. The programmer can use these special macros in the source files to define/undefine new macros, include text from other files, do comparisons,etc.These are listed below:

- **`\def`** allows a programmer to define a new macro mapping:
  `\def{NAME}{VALUE}`
  The entire \def macro and arguments are always replaced by the empty string. The argument `NAME` must be a nonempty alphanumeric string (can be arbitrarily long). As usual, the `VALUE` argument must be brace balanced, but can contain arbitrary text. After processing a `\def` macro and its arguments, the `NAME` argument is now mapped to the `VALUE` argument. In the future, macros with that name are valid: the `\NAME{ARG}` macro should be replaced by `VALUE`—with each occurrence of the unescaped

character `#` replaced by the argument string (`ARG`). Note: custom defined macros must always have exactly one argument, if the `VALUE` doesn't have any unescaped `#` characters, the argument is ignored.

- **`\undef`** undefines previously defined macro. `\undef` is replaced by the empty string:
  `\undef{NAME}`

- **`\if`** allows text to be processed conditionally (like and if-then-else block). Your implementation should consider false as the empty string and true for any non-empty string. The `\if` macro should have the following form:
  `\if{COND}{THEN}{ELSE}`
  Like all macros, all three arguments can contain arbitrary text, but must be braced balanced. You should not expand `COND`. The functionality should be this: the entire `\if` macro including arguments should be replaced with either the `THEN` or `ELSE` depending on the size of `COND`. Then, after expansion, processing resumes at the beginning of the replacement string.

- **`\ifdef`** is similar to `\if`; it expands to either `THEN` or `ELSE`:
  `\ifdef{NAME}{THEN}{ELSE}`
  The main difference is with the condition argument, `NAME`, which is restricted to alphanumeric characters. If `NAME` matches a currently defined macro name then the condition is true, otherwise it is false.

- **`\include`** macros are replaced by the contents of the file `PATH`. Typical brace balancing rules apply here:
  `\include{PATH}`

- **`\expandafter`** has the form:

`\expandafter{BEFORE}{AFTER}`
The point of this macro is to delay expanding the before argument until the after argument has been expanded. The output of this macro expansion is simply `BEFORE` immediately followed by the expanded `AFTER`. Note that this changes the recursive evaluation rule, i.e. you should eagerly expand the all macros in the `AFTER` string before touching `BEFORE`. This means that any new macros defined in `AFTER` should be in scope in for the `BEFORE`. You may not use additional processes/threads to

accomplish these actions. Here's an example program:

Input	Output
`\def{B}{buffalo}` `\expandafter{\B{}}{\undef{B}\def{B}{bison}}`	`bison`

-
- Why is this the case? It is because `\B{}` is expanded after it has be redefined in the after argument. Here's the steps to process these macros:
    1. `AFTER` should be fully expanded by running your expansion algorithm recursively (including the removal of certain escape characters in normal text, see the section below).
    2. the result of the above expansion should be appended to the (unexpanded) `BEFORE` argument
    3. the `\expandafter` macro and arguments are now replaced with the above concatenation.
    4. standard expansion processing should continue, starting from the start of `BEFORE`.

Comments

Your program should support comments. The comment character, `%`, should cause your program to ignore it and all subsequent characters up to the first non-blank, non-tab character following the next newline or the end of the current file, whichever comes first. This convention applies only when reading characters from the file(s) specified on the command line (or the standard input if none is specified) or from an included file. Comments should be removed only once from each file or from standard input. After all inputs are read and comments are removed, then you should start expanding. Note: the comment character can be escaped, see the section below.

Escape Characters

Besides being used as the “start” character for a macro, a `\` character can also be used to escape one of the following special characters `\`, `#`, `%`, `{`, `}` so that it is not treated as a special character. For these characters the effect of the \ is preserved until it is about to be output, at which point it is suppressed and the `\`, `#`, `%`, `{`, `}` is output instead. In effect, the \ is ignored and the following character is treated as a non-special character thereafter. That is, in effect the pair of characters can be treated as a macro with no arguments until it is expanded and output. We then have the following cases:

- Escape character followed by `\`, `#`, `%`, `{`, `}`: for this case use the rule above,i.e.when it is time to output only print the second character.
- Escape character followed by an alphanumeric character: in this case we must be reading a macro,so all the macro parsing rules apply.
- Escape character followed by non-alphanumeric and not `\`, `#`, `%`, `{`, `}`: in this case these characters have no special meaning to your parser.

Error Detection

The following two kinds of errors should be detected:

- Parsing Errors. For example, in a `\def` macro, if `NAME` is not a nonempty alphanumeric string.Another example would be if a macro name is not immediately followed by an argument in wrapped in balanced curly braces, or more generally: if a macro has too few arguments.
- Semantic Errors. For example, a macro name is not defined. Another example, would be an attempt to redefine a macro before undefining it, or an attempt to undefine a nonexistent macro.

For both of these kinds of errors your program should write a one-line message to stderr and exit. If you detect an error, you should not output a partial evaluation of any input. The following is a list of errors or scenarios you should ignore:

- Cyclical macro definitions
- Cyclical file includes

- Infinite expandafter loops
- Attempts to redefine a built-in macro

You may assume malloc, calloc, and realloc never fail (Note: when you are writing robust software outside the classroom, this is not a safe assumption).

Performance

The number of macros will never be large enough to require more than a linear search of the list of macros. Your program should run in time and space proportional to the number of characters processed (= the sum of the lengths of the file(s) specified on the command line (or the standard input if none is specified) and the lengths of all macro expansions). If your program fails any test by exceeding the time or space limit, the burden of proof that this is not an error is on you. The key to good performance here will be on how you handle strings (i.e. your expansion algorithm/data structure), try to avoid doing string shifts and insertions.

Misc. Requirements

- The input files should be thought of as one long string (after removing comments). Hence, macros defined in the first file should be accessible in the second, and macros can span two or more files.
- When your program exits, all allocated storage must be reachable. In short, you don't have tofree all memory that you have dynamically allocated using malloc, calloc, etc., but you do have tofree any memory before it goes out of scope. You have to keep an in-scope pointer to all unfreed dynamic memory.
- Your program should have no warnings when compiled. We will use the C11 standard when com-piling your program with gcc, and check for warnings with these compiler options:
  `-std=c11 -pedantic -Wall`
- For this assignment, we require that you use make to build your code. You will submit your Makefile with your C code. See below in the submission instructions on how to setup this Makefile.

- Do not use any external libraries other than the C Standard Libraries (i.e.do not link other libraries in your compilation step).
- You may create temporary files, but the files must be deleted when the program exits.