In this lab you will be implementing a two-pass linker. In general, a linker takes individually compiled code/object modules and creates a single executable by resolving external symbol references (e.g. variables and functions) and module relative addressing by assigning global addresses after placing the modules' object code at global addresses.

Rather than dealing with complex x86 tool chains, we assume a target machine with the following properties: (a) word addressable, (b) addressable memory of 512 words, and (c) each valid word is represented by an integer (<10000).
[*I know that is a really strange machine, but I once saw an UFO too*].

The input to the linker is a file containing a **sequence of tokens** (symbols and integers and instruction type characters). Don't assume tokens that make up a section to be on one line, don't make assumptions about how much space separates tokens or that lines are non-empty for that matter or that each input conforms syntactically. Symbols always begin with alpha characters followed by optional alphanumerical characters, i.e.[a-Z][a-Z0-9]*. Valid symbols can be up to 16 characters. Integers are decimal based. Instruction type characters are (I, A, R, E). Token delimiters are ' ', '\t' or '\n'.

The input file to the linker is structured as a series of "object module" definitions.
Each "object module" definition contains three parts (in fixed order): definition list, use list, and program text.

- ***definition list*** consists of a count *defcount* followed by *defcount* pairs (S, R) where S is the symbol being defined and R is the relative word address (offset) to which the symbol refers in the module (0-based counting).
- ***use list*** consists of a count *usecount* followed by *usecount* symbols that are referred to in this module. These could include symbols defined in the *definition list* of any module (prior or subsequent or not at all).
- ***program text*** consists of a count *codecount* followed by *codecount* pairs (**type, instr**), where *type* is a single character indicating the addressing mode as **R**elative, **E**xternal. **I**mmediate or **A**bsolute and *instr* is the instruction (integer) Note that *codecount* defines the length of the module.

An instruction is composed of an integer that is comprised of an opcode (op/1000) and an operand (op mod 1000). The opcode always remains unchanged by the linker. For the instruction value read an integer and ensure opcode < 10, see errorcodes below. The operand is modified/retained based on the instruction type in the *program text* as follows:
(**R**) operand is a relative address in the module which is relocated by replacing the relative address with the absolute address of that relative address after the module's global address has been determined (absolute_addr = module_base+relative_addr).
(**E**) operand is an external address which is represented as an index into the uselist. For example, a reference in the program text with operand K represents the Kth symbol in the use list, using 0-based counting, e.g., if the use list is ''2 f g'', then an instruction ''E 7000'' refers to f, and an instruction ''E 5001'' refers to g. You must identify to which global address the symbol is assigned and then replace the operand with that global address.
(**I**) an immediate operand is unchanged.
(**A**) operand is an absolute address which will never be changed in pass2; however it can't be ">=" the machine size (512);

The linker must process the input twice (that is why it is called two-pass) (to preempt the favored question: "Can I do it in one pass?" → NO, because storing tokens makes your program more complex). **Pass One** parses the input and verifies the correct syntax and determines the base address for each module and the absolute address for each defined symbol, storing the latter in a symbol table. The first module has base address zero; the base address for module X+1 is equal to the base address of module X plus the length of module X (defined as the number of instructions in a module). The absolute address for symbol S defined in module M is the base address of M plus the relative address of S within M. After pass one print the symbol table (including errors related to it (see rule2 later)). Do not store parsed tokens, the only data you should and need to store between passes is the symboltable.
Pass Two again parses the input and uses the base addresses and the symbol table entries created in pass one to generate the actual output by relocating relative addresses and resolving external references. You should reuse pass-1 parser code just with different actions. You must clearly mark your two passes in the code through comments and/or proper function naming.

Other requirements: error detection, limits, and space used.

To receive full credit, you must check the input for various errors (test inputs will have lots of errors). All errors/warnings should follow the message catalog provided below. We will do a textual difference against a reference implementation to grade your program. Any reported difference will indicate a non-compliance with the instructions provided and is reported as an error and results in deductions.

You should continue processing after encountering an error/warning (other than a syntax error) and you should be able to detect multiple errors in the same run.

1. You should stop processing if a syntax error is detected in the input, print a syntax error message with the line number and the character offset in the input file where observed. A syntax error is defined as a missing token (e.g. 4 used symbols are defined but only 3 are given) or an unexpected token. Stop processing and exit.
2. If a symbol is defined multiple times, print an error message and use the value given in the first definition. The error message is to appear as part of printing the symbol table (following symbol=value printout on the same line).
3. If a symbol is used in an E-instruction but not defined anywhere, print an error message and use the value absolute zero.
4. If a symbol is defined but not used, print a warning message and continue.
5. If an address appearing in a definition exceeds the size of the module, print a warning message and treat the address given as 0 (relative to the module).
6. If an external address is too large to reference an entry in the use list, print an error message and treat the address as immediate.
7. If a symbol appears in a use list but is not actually used in the module (i.e., not referred to in an E-type address), print a warning message and continue.
8. If an absolute address exceeds the size of the machine, print an error message and use the absolute value zero.
9. If a relative address exceeds the size of the module, print an error message and use the module relative value zero (that means you still need to remap "0" that to the correct absolute address).
10. If an illegal immediate value (I) is encountered (i.e. >= 10000), print an error and convert the value to 9999.
11. If an illegal opcode is encountered (i.e. op >= 10), print an error and convert the <opcode,operand> to 9999.

The following exact limits are in place.

a) Accepted symbols should be upto 16 characters long (not including terminations e.g. '\0'), any longer symbol names are erroneous.
b) a uselist or deflist should support 16 definitions, but not more and an error should be raised.
c) number of instructions are unlimited (hence the two pass system), but in reality they are limited to the machine size.
d) Symbol table should support at least 256 symbols (reference program supports exactly 256 symbols).

There are several sample inputs and outputs provided as part of the sample input files / output files (see NYU Classes).
The first (*input-1)* is shown below and the second (*input-2)* is a re-formatted version of the first. They both produce the same output as **the input is token-based** and hence present the same content to the linker. Many of the input sets contain errors that you are to detect as described above. Note that when you have questions regarding errors, please first make sure the structure of the input is not messing with your mind. We will run your lab on these (and other) input sets.

```
1 xy 2
2 z xy
5 R 1004  I 5678  E 2000  R 8002  E 7001
0
1 z
6 R 8001  E 1000  E 1000  E 3000  R 1002  A 1010
0
1 z
2 R 5001  E 4000
1 z 2
2 xy z
3 A 8000  E 1001  E 2000
```

Your output is expected to strictly follow this format (with exception of empty lines):

```
Symbol Table
xy=2
z=15

Memory Map
000: 1004
001: 5678
002: 2015
```

```
003: 8002
004: 7002
005: 8006
006: 1015
007: 1015
008: 3015
009: 1007
010: 1010
011: 5012
012: 4015
013: 8000
014: 1015
015: 2002
```

The following output is heavily annotated for clarity and class discussion. You are not creating this output nor do we have means to check it. However, it should help you understand the operation and mapping of symbols etc.

```
Symbol Table
xy=2
z=15
Memory Map
+0
0:      R 1004        1004+0 =  1004
1:      I 5678                  5678
2: xy:  E 2000 ->z              2015
3:      R 8002        8002+0 =  8002
4:      E 7001 ->xy             7002
+5
0:      R 8001        8001+5 =  8006
1:      E 1000 ->z              1015
2:      E 1000 ->z              1015
3:      E 3000 ->z              3015
4:      R 1002        1002+5 =  1007
5:      A 1010                  1010
+11
0:      R 5001        5001+11=  5012
1:      E 4000 ->z              4015
+13
0:      A 8000                  8000
1:      E 1001 ->z              1015
2 z:    E 2000 ->xy             2002
```

Note that even an empty program should have the “Symbol Table” and “Memory Map” line.

For a test case to pass you must catch ALL warning/errors and generate the correct output for a given input file.

Example:

```
Symbol Table
X21=3
X31=4

Memory Map
000: 1003
001: 1003
002: 1003
003: 2000  Error: Absolute address exceeds machine size; zero used
004: 3000  Error: Relative address exceeds module size; zero used

Warning: Module 3: X31 was defined but never used
```

Parse errors should abort processing.
Error messages must be following the instruction as shown above.
Warnings message locations are defined further down.
Module counting starts at 1.

I provide in C the code to print parse errors, which also gives you an indication what is considered a parse error.

```
void __parseerror(int errcode) {
    static char* errstr[] = {
        "NUM_EXPECTED",              // Number expect, anything >= 2^30 is not a number either
        "SYM_EXPECTED",              // Symbol Expected
        "ADDR_EXPECTED",             // Addressing Expected which is A/E/I/R
        "SYM_TOO_LONG",              // Symbol Name is too long
        "TOO_MANY_DEF_IN_MODULE",    // > 16
        "TOO_MANY_USE_IN_MODULE",     // > 16
        "TOO_MANY_INSTR",             // total num_instr exceeds memory size (512)
    };
    printf("Parse Error line %d offset %d: %s\n", linenum, lineoffset, errstr[errcode]);
}
```

(Note: line numbers start with one and offsets in the line start with one, offsets should indicate the first character offset of the token that is wrong, not the last). Tabs (‘\t’) count as one character.
Error messages have the following text and should appear right at the end of the line you are printing out

"Error: Absolute address exceeds machine size; zero used"	(see rule 8)
"Error: Relative address exceeds module size; zero used"	(see rule 9)
"Error: External address exceeds length of uselist; treated as immediate"	(see rule 6)
"Error: %s is not defined; zero used" (insert the symbol name for %s)	(see rule 3)
"Error: This variable is multiple times defined; first value used"	(see rule 2)
"Error: Illegal immediate value; treated as 9999"	(see rule 10)
"Error: Illegal opcode; treated as 9999"	(see rule 11)

Warning messages have the following text and are on a separate line.

"Warning: Module %d: %s too big %d (max=%d) assume zero relative\n"	(see rule 5)
"Warning: Module %d: %s appeared in the uselist but was not actually used\n"	(see rule 7)
"Warning: Module %d: %s was defined but never used\n”	(see rule 4)

Locations for these warnings are:

Rule 5: to be printed after each module in pass1
Rule 7: to be printed after each module in pass2 (so actually interspersed in the memory map (see out-9)
Rule 4: to be printed after pass 2 (i.e. all modules have been processed) (see out-3).

Parse Error Location:

Parse errors are to be located at the first character of the wrong token, or if end-of-file is reached at the end of file location. There is one special case when the eof ends with `\n’. My expectation is that the line number reported actually exists in the file and that an editor (e.g. vi) can jump to it. In this particular case the linenumber to be reported is the last line read and the last position of that line, not the next line and offset 1 (see *input-12* for an example). The error is at the very last position of the line. Reason is when one does a linecount on the file (“wc –l input-12”) it shows 3 not 4.

Hint: for each parser error and warning error mentioned above you should have code checking for that.

Program Specification

Only C/C++ are allowed for this lab. The program should take a single command line argument for input file. The output of the program should go to **standard output.**
The program **should run** on the **linserv1.cims.nyu.edu** machine of NYU, which is where it will be graded, so please make sure your program runs there. We realize you code on your own machine and transferring to *linserv1* machine exposes occasionally some linker errors. In that case use static linking (such as not finding the appropriate libraries at runtime), though hopefully that is now resolved with the "module" approach discussed at the end.

Testing your program before submission

As part of the *lab1_assign.tar.Z* in NYU Brightspace you will see a ***runit.sh*** and a ***gradeit.sh*** script (the same we will use for grading albeit with more inputs). Use "tar -xzvf lab1_ssign.tar.Z" to decompress the file on a Unix machine. Understand the script and what it does. If you can't pass this script, things will be flagged during the grading process as well. Don't assume that just because you pass for these inputs, it will pass for all inputs. Carefully go through the rules above and ensure covered each rule with some code (if/then/else) and it is at the right location.

Execute as follows:
Create yourself a directory <your-outdir> where your outputs will be created.
`> cd lab1_assign`
`> ./runit.sh <your-outdir> <your-executable and optional arguments>` # make your output directory different
The above will create all the outputs for the available inputs (1-20)
`> ./gradeit.sh . <your-outdir>` # note 2nd argument is a DOT for the local directory where reference output are.

The above will compare the reference outputs (out-[1-20]) with the ones created with your program and tell you how many you got right and which ones are wrong. There will be a file called <your-outdir>/LOG that contains which cases you got wrong and where the differences are. If you want to analyze further run "`diff -b -B -E`" by hand on a particular output pair. It will create a file LOG.txt which shows you the diff and where things are wrong.

The reference program used during grading is located on */home/frankeh/Public/linker* on cims systems. So feel free to try it in order to answer any questions you might have on what is expected for a particular input. An input generator is provided under /home/frankeh/Public/lab1gen.py

What to submit

Submit a zip archive containing (i) source code (ii) makefile to compile your code (iii) ReadMe (If running the program is not straightforward, otherwise not required). Please make sure the zip doesn't have any input and/or output files nor contains other zip or tar files or backup files.

Grading

The lab is graded using a "diff –b –B -E" against the reference output created by the test program (aka reference program) using a grading harness described above (*runit.sh + gradeit.sh)*
Inputs will be the ones provided in NYU Classes as well as other ones and will be checked for all of the error conditions.
It is imperative that you match the output as generated by the ref program to allow for the automated testing for yourself as well. Use the harness instead of manual checking when you have the basics running, it tells you where things are wrong.
We score this lab as 100pts. You will receive 40 pts for a submission that attempts to solve the problem. The rest you get 60/N points for each successful test that passes the "*diff*". In order to institute a certain software engineering discipline, i.e. following a specification and avoiding unintended releases of code and data in real life, we account for the following additional deductions:

Reason	Deduction	How to avoid
Makefile not working on CIMS or missing.	1pts	Make sure your source compiles with your Makefile
Storing Tokens instead of reparsing the file. All you should store is the Symboltable between passes.	2pts	Follow instructions on parsing. After that, copy the shell of the parser for the 2nd pass. Close the file, reopen the file and parse again, just change the actions taken between 1st (error checking, module & symbol table creation) and 2nd pass (instruction transformation).
Late submission	2pts/day	Upto 7 days late allowed.
Inputs/Outputs or *.o files in the submission	1pt	Go through your intended submission and clean it up.
Output not going to the screen but to a file	1pt	We utilize the output to <stdout> during the runit.sh and gradeit.sh so just use printf or cout. You will have to fix this

Useful Stuff

How to approach this lab:

Step 1: Write a tokenizer
Write a tokenizer that simply parses the input, prints the token and the position in the file found, at the end prints the final position in the file (as you will need that for error reporting). Verify it correctly recognizes tokens, lines and line offsets and also print the final error location as the last line read with last character in the line (not the next empty line).
Historically, getting this first step right and then layering <step 2> over it is the main headache in this assignment.
The functions you need to study for this are `getline()` or `fgets()` (reads a full line into a buffer) or C++ equivalent and `strtok()`, which tokenizes input lines. `Please` use Linux's build-in help : "man strtok" and understand how a new line is seeded and continued in subsequent calls. My tokenizer executable is available on cims under /home/frankeh/Public/tokenizer (see what it produces on an input). You don't have to match, its just a good way to start. Read the "man page"; it will tell you all you need to know how to use strtok (including examples and what to watch out for).

Step 2: Write basic token functions
Once you have the `getToken()` function written above and you verified that your token locations are properly reported, extract it out of the program above and start writing the linker program. Layer the `readInt(), readSymbol(), readIAER()` functions on top of it by checking that the token has the correct sequence of characters and length (e.g. integers are all numbers) and test via a simple program. Macros like `isdigit(), isalpha(), isalnum()` can proof useful.

Step 3: Write the parser
Here are some hints on writing a parser. In general one could write this parser using lex/yacc, however this is so simple that I suggest writing a simple recursive decent parser, in particular since you have to parse the input twice. In lex and yacc you would need to handle the error locations to conform to the specification. It would have a structure similar to the following (all pseudoCode). This structure is simply copied twice and the actions taken in each path are different.
Once you have the first pass written, reset the input file, copy the pass1() to pass2() and rewrite. Note, in pass2 certain error checking doesn't have be done anymore. Instead you are now rewriting the instructions.

```
Pass1() {
      while (!eof) {
            createModule();
            int defcount = readInt();
            for (int i=0;i<defcount;i++) {
                 Symbol sym = readSym();
                 int val = readInt();
                 createSymbol(sym,val);          ← this would change in pass2
            }
            int usecount = readInt();
            for (int i=0;i<usecount;i++) {
```

```
                Symbol sym = readSym();
                // we don't do anything here    ← this would change in pass2
            }
            int instcount = readInt();
            for (int i=0;i<instcount;i++) {
                char addressmode = ReadIEAR();
                int  operand = ReadInt();
                :  // various checks                ← this would change in pass2
                :  //  - " -
            }
        }
}
```

Of course you will have to deal with the errors etc. The first pass does syntax and early error checking and creates the symbol table, the second pass performs additional error checking and instruction transformation. In order to go over the file a second time reset the file pointer or close and reopen the file. You should not store tokens or deflist or uselist or instructions during the first pass in order to pass to pass2. Only the symbol table needs to be stored between passes.

Writing a Makefile

A makefile should be named either "makefile" or "Makefile". *make* will look for either of these files during building. Here is a good introduction: *http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/* with more sophisticated ones. The simplest Makefile you can think of contains how it builds the executable and how it cleans up (here it assumes one source file only and builds the *linker* executable):

```
linker: linker.cpp
    g++ -g linker.cpp -o linker

clean:
    rm -f linker *~
```

Here is a more sophisticated one that assumes multiple inputfiles: file1.c .. file3.c and allows to specify the compiler version

```
CFLAGS=-g       # -g = debug,  -O2 for optimized code    CXXFLAGS for g++
CC=gcc-your-favorite-version

linker: file1.c file2.c file3.c
    $(CC)  $(CFLAGS) –o linker file1.c file2.c file3.c
```

There are certainly more sophisticated once to write.

Switching compiler versions on linserv1 it is strongly suggested you use gcc-9.2, the default 4.8.5 is old.

```
gcc –v or g++ -v        # will tell you the currently active gcc version
module avail gcc        # This will show you the list of all available modules
module load gcc-9.2.0   # loads indicated gcc version (must be in avail list)
module unload gcc-9.2.0 # reverts back to previous one (a "stack" is maintained)

# If you want to load a different version - first unload otherwise they stack
```

Please test you code on linserv1 and run (runit.sh and gradeit.sh). If there are errors go to the LOG.txt file created in the output directory and see what's wrong and fix it. Note that code that runs fine on windows or MAC might not run correctly on Linux machines. The burden is on you.