You are to implement a two-pass linker in C, C++, or Java and submit the **source** code on NYU Classes, which we compile and run.

The target machine is word addressable and has a memory of 200 words, each consisting of 4 decimal digits. The first (leftmost) digit is the opcode, which is unchanged by the linker. The remaining three digits (called the address field) form either

- An immediate operand, which is unchanged.
- An absolute address, which is unchanged.
- A relative address, which is relocated.
- An external address, which is resolved.

Relocating relative addresses and resolving external references were discussed in class and are in the notes.

The input consists of a series of object modules, each of which contains three parts: definition list, use list, and program text. Preceding all the object modules is a single integer giving the number of modules present.

The linker processes the input twice (that is why it is called two-pass). Pass one determines the base address for each module and the absolute address for each external symbol, storing the later in the symbol table it produces. The first module has base address zero; the base address for module $I+1$ is equal to the base address of module I plus the length of module I. The absolute address for a symbol S defined in module M is the base address of M plus the relative address of S within M. Pass two uses the base addresses and the symbol table computed in pass one to generate the actual output by relocating relative addresses and resolving external references.

The definition list is a count `ND` (Number of Definitions) followed by `ND` pairs (S, R) where S is the symbol being defined and R is the relative address to which the symbol refers. Pass one relocates R forming the absolute address A and stores the pair (S, A) in the symbol table.

The use list is a count `NU` (Number of Uses) followed by `NU` pairs (S, R), where S is an external symbol used in the module and R is a relative address where S is used. The (dummy) address initially in R is a pointer to the next use of S. This linked list of uses ends with a pointer of 777.

The program text consists of a count `NT` followed by `NT` 5-digit numbers, where the first 4 digits form an instruction as described above and the last digit gives the type of the address component: 1=immediate, 2=absolute, 3=relative, and 4=external. NT, the number of instructions, is thus the length of the module.

`NT` is also the length of the module.

Other requirements: Error detection, arbitrary limits, et al.

Your program must check the input for the errors listed below. All error messages produced must be informative, e.g., "Error: X21 was used but not defined. It has been given the value 111".

- If a symbol is defined but not used, print a warning message and continue.
- If a symbol is multiply defined, print an error message and use the value given in the first definition.
- If a symbol is used but not defined, print an error message and use the value zero.
- If multiple symbols are listed as used in the same instruction, print an error message and ignore all but the last usage given.
- If an address appearing in a definition exceeds the size of the module, print an error message and treat the address as 0 (relative).
- If an immediate address (i.e., type 1) appears on a use list, print an error message and treat the address as external (i.e., type 4).
- If an external address is not on a use list, print an error message and treat it as an immediate address.
- If an absolute address exceeds the size of the machine, print an error message and use the largest legal value.

You may need to set "arbitrary limits", for example you may wish to limit the number of characters in a symbol to (say) 8. Any such limits should be clearly documented in the program and if the input fails to meet your limits, your program must print an error message. Naturally, any such limits must be large enough for all the inputs on the web. Under no circumstances should your program reference an array out of bounds, etc.

Submit the **source** code for your lab, together with a *README* file (required) describing how to compile and run it. Your program must read an input set from standard input, i.e., directly from the keyboard. It is an error for you to require the input be in a file.

You may develop your lab on any machine you wish, but must insure that it compiles and runs on the NYU system assigned to the course.

There are several sample input sets on NYU classes. The first is shown below and the second is an re-formatted version of the first. If you use the java `Scanner` or C's `scanf()` (which I recommend you do), inputs 1 and 2 will look the same to your program. Some of the input sets contain errors that you are to detect as described above. We will run your lab on these (and other) input sets.

```
4
1 xy 2
1 z 4
5 10043 56781 27774 80023 70024
0
1 z 3
6 80013 17774 10014 30024 10023 10102
0
1 z 1
2 50013 47774
1 z 2
1 xy 2
3 80002 17774 20014
```

The following is output annotated for clarity and class discussion. Your output is not expected to be this fancy.

```
Symbol Table
 xy=2
 z=15
Memory Map
  +0
  0:       10043      1004+0 = 1004
  1:       56781                 5678
  2: xy:   27774 ->z             2015
  3:       80023      8002+0 = 8002
  4:       70024 ->z             7015
  +5
  0        80013      8001+5 = 8006
  1        17774 ->z             1015
  2        10014 ->z             1015
  3        30024 ->z             3015
  4        10023      1002+5 = 1007
  5        10102                 1010
  +11
  0        50013      5001+11= 5012
  1        47774 ->z             4015
  +13
  0        80002                 8000
  1        17774 ->xy            1002
  2 z:     20014 ->xy            2002
```