Com S 227
Spring 2019
Assignment 3
300 points

Due Date: Friday, April 5, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm April 4)

10% penalty for submitting 1 day late (by 11:59 pm April 6)

No submissions accepted after April 6, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: Our second exam is Monday, April 8, which is just a few days after the due date for this assignment. ***It will not be possible to have the assignments graded and returned to you prior to the exam****.*

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you some practice writing loops, using arrays and lists, and most importantly to get some experience putting together a working application involving several interacting Java classes.

For this assignment you will implement an *assembler* for the simple computer from Miniassignment 2. An assembler is a program that translates *assembly language* into machine code. An assembly language is similar in structure to machine code, but allows you to use symbolic names for opcodes, data locations (variables), and jump targets. There is a detailed example below.

There are two classes for you to implement. **`CS227Asm`** is the assembler itself, and **`AsmFileUtil`** is a utility containing some static methods for working with files. As always, your primary responsibility is to implement these classes according to the specification and test them carefully.

Some additional types you will need are provided in the **`api`** package, including **`Instruction`**, **`NVPair`**, and **`SymbolTable`**. These are described in more detail below. You will need to become familiar with the **`api`** package code in order to use it. These classes are not complex and you'll find that source code is pretty easy to read.

The two classes you write go in package **`hw3`**. You should not modify any of the code in the **`api`** package.

Specification

The specification for this asssignment includes

- this pdf,
- any "official" clarifications posted on Piazza, and
- the online javadoc

The assembly language

An assembly language program is a text file. However, in our implementation, the file-reading is done separately, so that the actual input to the assembler is an ArrayList of strings, one for each line of the file. The file is defined as follows:

- Programs have three sections, *data*, *labels*, and *instructions*, in that order.
- All three sections are required, even if empty.
- Each section starts with a semantic marker on a line by itself: "data:", "labels:", or "instructions:", respectively.
- The *data* section is a list of zero or more name/value pairs of variable names and their initial integer values. Initial values are required even if the program will write the variable before reading it. These name/value pairs occur one per line until the "labels:" semantic marker.

- The *labels* section consists of a series of jump target names. All jump instructions used in the instructions section must have a target that is named in the labels section. The jump target names are listed one per line until the "instructions:" semantic marker.
- The *instructions* section consists of a series of instructions and labels representing jump targets.
    - There are two I/O instructions, "read" and "write", which have a variable as a parameter (that is, they must be followed by a variable name previously defined in the data section).
    - There are two memory instruction, "load" and "store", which have a variable as a parameter. There are five arithmetic instructions, "add", "sub", "div", "mul",
    - and "mod", which have a variable as a parameter.
    - There are three jump instructions, "jump", "jumpn", and "jumpz", which have a jump target as a parameter (i.e., must be followed by a label that was previously defined in the labels section.)
    - There is a "halt" instruction which has no parameter.
    - Jump targets are given by a jump target name (per the labels section) appearing on a line in lieu of an instruction.
- Blank lines are not allowed.
- All lines may include comments that should be ignored. For lines that contain semantic markers, label declarations, labels (jump targets in the instruction section), and the "halt" instruction, comments begin with the *second* non-whitespace token. Variable declarations and instruction other than "halt" all require a parameter, so comments begin with the *third* non-whitespace token.

Here is an example of a CS227Asm program that prints out numbers from 10 down to 1. (The remarks on the right are comments.) Read through the code and see how it works.

```
data:
count 10          count = 10
const_one 1
labels:
point_a
point_b
instructions:
load count
point_a
jumpz point_b     while count != 0
write count           print the value of count
sub const_one         count = count - 1
store count
jump point_a          (go back to top of loop)
point_b
halt
```

Parsing and code generation

The sample program above is assembled into the following machine code. Note that the original instructions (though not the comments) are included in the result as descriptions following the machine instructions, and locations of jump targets are labeled (text in parentheses).

```
+3007 load count
+6306 jumpz point_b (label point_a)
+2007 write count
+5108 sub const_one
+4007 store count
+6001 jump point_a
+8000 halt (label point_b)
+0010 count
+0001 const_one
-99999
```

Here is how it works.

1. First the data section is parsed, creating a "symbol table" for the variables to be used in the program. The symbol table is just a list of name-value pairs that associates each variable with its initial value. In this example, the table would contain the string "count" with initial value 10 and the string "const_one" with initial value 1.

2. Next the label section is parsed, creating a symbol table for the labels. Initially, each label is associated with value 0. When we parse the instruction section, we'll adjust the label values to correspond to the actual memory address at which the label would occur when the instructions are correctly placed into memory.

3. Next the instruction section is parsed, creating a list of **`Instruction`** objects *and* completing the symbol table for labels using the locations of the jump targets. The **`Instruction`** type is defined for you in the **`api`** package. Each **`Instruction`** encapsulates an opcode, an operand, and a description. The way it works is that the instructions will be placed in memory starting at address 0. By keeping a count of how many instructions have been added to the list so far, whenever we find a label we know what the address associated with that label is. Note that the labels themselves do not increment the instruction count and are not included in the instruction list:

```
data:
count 10
const_one 1
labels:
point_a
point_b
instructions:
load count        <-- instruction 0
point_a           <-- this label gets value 1
jumpz point_b     <-- instruction 1
write count       <-- instruction 2
sub const_one     <-- instruction 3
store count       <-- instruction 4
jump point_a      <-- instruction 5
point_b           <-- this label gets value 6
halt              <-- instruction 6
```

4. The next thing to do is go through the instructions and set the numeric value of the operands. You'll notice that when in the constructor for an **`Instruction`** object, we record the symbolic name for the operand (either a label for jump instructions, or a variable name for the other instructions), but we don't yet have the numerical value for the operand. How we get that value will differ depending on whether it's a jump instruction or not. For a jump instruction, we find the label in the symbol table for labels and use the address we previously stored there in step 3. As an example, note the line

```
+6306 jumpz point_b
```

in the generated machine code. The operand part, 06, is the memory address of the instruction at the label **`point_b`** in the assembly code.

All the other instructions (except halt), refer to a variable, so we need the address for the variable. The variables will be laid out in memory immediately following the instructions, so all we need to do is find the index of the variable in the data symbol table, and add the size of the instruction list. As an example, note the line

```
+5108 sub const_one
```

in the generated machine code. The operand part, 08, is the memory address to be associated with the variable **`const_one`**. We know that because the index of **`const_one`** in the symbol table is 1 (i.e., it's the second data declaration), the length of the instruction list is 7, and 7 + 1 = 8.

5. Next, we want to include jump target locations in the instruction descriptions. For example, in the sample machine code above, the text in parentheses as seen in the line

```
+8000 halt (label point_b)
```

shows that the jump target represented by label **`point_b`** is memory location 6 (the seventh line of the machine code). This is not hard: in the symbol table for the labels, the value of each label is the index of the instruction whose description should have the label information appended. The **`Instruction`** class has a method **`addLabelToDescription`** to make this easy.

6. Finally, we need to write the machine code as a list of strings. The instructions come first, then the variables. The **`Instruction`** class has a built-in method **`toString()`** that converts the **`Instruction`** into the desired string format for the output. For the variables, just iterate over the symbol table for data. Getting the value to be represented as a four-digit string with the leading plus or minus sign can be done with **`String.format`**, which works the same as **`printf`** as used in Miniassignment 2.

The tasks above are all specified as public methods to simplify testing, so that your **`assemble()`** method can literally be implemented like this:

```
public ArrayList<String> assemble()
{
  parseData();
  parseLabels();
  parseInstructions();
  setOperandValues();
  addLabelAnnotations();
  return writeCode();
}
```

The `AsmFileUtil` class

The **`CS227Asm`** class does not read input or write output. The source program is provided to the constructor as a list of strings, and the machine language program is returned as a list of strings. In practice, an assembler would read its source from a file and write the machine code to another file. The necessary file operations are to be implemented by you as static methods in the utility class **`AsmFileUtil`**. For example, if the sample program above is in a file named "test1.asm227", then the following main method would produce the machine code as console output:

```
public static void main(String[] args) throws FileNotFoundException
{
  ArrayList<String> result = AsmFileUtil.assembleFromFile("test1.asm227");
  for (String s : result)
  {
    System.out.println(s);
  }
}
```

And if you have a working version of the machine simulator CS227Comp.java from Miniassignment 2 (you can download a sample solution), you could actually run the program too:

```
public static void main(String[] args) throws FileNotFoundException
{
  int[] code = AsmFileUtil.createMemoryImageFromFile("test1.asm227");
  CS227Comp comp = new CS227Comp(10);
  comp.loadMemoryImage(code);
  comp.runProgram();
}
```

The `NVPair` and `SymbolTable` classes

In addition to the **`Instruction`** class discussed above, the **`api`** package includes two types **`NVPair`** and **`SymbolTable`**. An **`NVPair`** is a simple type encapsulating a string name and an associated int value, such as a variable name and associated initial value or a jump target label and its associated address. A **`SymbolTable`** is basically a list of **`NVPair`** objects, along with some useful methods for looking things up. Take a look at the code to see what the methods actually do.

Remember: do not modify any code in the **`api`** package.

Notes on error checking

Although our CS227Comp program had a configurable amount of memory, it could only address up to 100 words. This is because of the two-digit operand. Any program + data that ends up greater than 100 words long is invalid. Of course, there are many other things that can invalidate a program that we haven't discussed here as well; for instance, using a variable or label which hasn't been defined, using a label more than once, etc. Handling all possible errors adds a not-insignificant extra level of complexity on top of the assembler. Suffice to say that we (the teaching staff) will not perform tests using deliberately broken programs on your assembler. You may assume that all input programs will be valid. This is not to say that you may not inadvertently (or on purpose) test your program using invalid programs of your own!

In most cases, some type of exception will be thrown whether or not you explicitly check for invalid code. You are not expected to perform error checking, but you may notice exceptions being "thrown" at some points in the **`api`** code. For example, in the in the **`SymbolTable`** method **`findByName`**, if there is no symbol with the given name, it will reach the line

```
throw new IllegalArgumentException(...)
```

which will, not surprisingly, cause your program to exit and display the stack track on the console. You'll find that this is useful for tracking down bugs in your code. (We will study the exception mechanism in some detail at the end of the semester.(

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specfications, ask questions when things require clarification, and write your own unit tests. Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up the two required classes. Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

> See the document "SpecChecker HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links" if don't remember how to import and run a SpecChecker.

Importing the sample code

1. Download the zip file. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.

If you have an older version of Java (below 8) or if for some reason you have problems with this process, or if the project does not build correctly, you can construct the project manually as follows:

1. Unzip the zip file containing the sample code.
2. In Windows Explorer or Finder, browse to the src directory of the zip file contents
3. Create a new empty project in Eclipse
4. In the Package Explorer, navigate to the src folder of the new project.
5. Drag the **`api`** folder from Explorer/Finder into the **`src`** folder in Eclipse.

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on functional tests that we run and partly on the grader's assessment of the quality of your code. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.
- Avoid code duplication.
- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. Use internal comments where appropriate to explain how your code works. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)

See the "Style and documentation" section below for additional guidelines.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.

    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
    - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Try not to embed numeric literals in your code. Use the defined constants wherever appropriate.
- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own "profile" for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`assignment3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`assignment3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be

placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification. *In particular, for this assignment we are not providing a specchecker that will perform any functional tests of your code.* It is up to you to test your own code (though you are welcome to share test cases on Piazza). Here are some basic operations:

- Be sure you are familiar with the basic machine operations as implemented in Miniassignment 2. In the Piazza homework post, you can find source code for a sample Miniassignment 2 solution that you can use to try out your machine code.
- Be sure you have done lab 8, which covers reading and writing files as well as the use of ArrayLists.
- Familiarize yourself with the code in the **`api`** package, since you will need to use all of it in one way or another.
- All the operations in **`AsmFileUtil`** depend on having a working **`CS227Asm`**, so you need to start with **`CS227Asm`**.

Each of the six public methods described previously (in "Parsing and code generation") is dependent on the previous one, so it makes sense to begin implementing them in the order given. As always start by writing simple test cases. For example, here is an easy way to create an ArrayList of strings for the sample program from the "Parsing and code generation" section, so you can start testing the assembler without worrying about reading and writing files:

```
String[] temp = {
  "data:",
  "count 10",
  "const_one 1",
  "labels:",
  "point_a",
  "point_b",
  "instructions:",
  "load count",
  "point_a",
  "jumpz point_b",
  "write count",
  "sub const_one",
  "store count",
  "jump point_a",
  "point_b",
  "halt"
};
```

```
ArrayList<String> program = new ArrayList<>();
for (String s : temp)
{
  program.add(s);
}
```

Then, you could write a simple test that calls **parseData** and then checks the symbol table to be sure it worked:

```
CS227Asm asm = new CS227Asm(program);
asm.parseData();
SymbolTable dataTable = asm.getData();
for (int i = 0; i < dataTable.size(); i += 1)
{
  NVPair p = dataTable.getByIndex(i);
  System.out.println(p);
}
```

which should produce the output:

```
count:10
const_one:1
```

See the sample program SimpleTests.java for some more ideas along these lines.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **SUBMIT_THIS_hw3.zip**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **hw3**, which in turn contains two files, **CS227Asm.java** and **AsmFileUtil.java.** Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 3 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw3**, which in turn should contain the two files **CS227Asm.java** and **AsmFileUtil.java**. You can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.