Com S 227
Spring 2021
Assignment 4
300 points

Due Date: Friday, April 30, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm April 29)

NO LATE SUBMISSIONS - EVERYTHING MUST BE IN FRIDAY NIGHT

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.

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 a chance to think about implementing a class hierarchy using inheritance. There are a lot of pieces to implement, but individually all of them are short and simple. When it's done, you will actually have implemented an interpreter for a simple programming language that we call Junebug[1]. It is huge fun to get this working and it will also start to give you some insight into how programming languages work.

[1] Because, of course, it is so delightful to work with.

For the purposes of this assignment, we will be implementing a hierarchy of *program nodes* representing *expressions* and *instructions*. Expressions are values, and instructions are statements that can be executed.

Classes intended to represent expressions implement the `Expression` interface, which has one method `eval()` that returns an integer. Classes intended to represent instructions implement the `Instruction` interface, which has one method `execute()`. Both of these types are subinterfaces of the interface `parser.ProgramNode`.

One of your main objectives in thinking about the implementations should be to minimize the amount of duplicated code throughout the classes.

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by how effectively you have been able to use inheritance to minimize duplicated code in the ProgramNode hierarchy.

Summary of tasks

You'll implement a *minimum* of eighteen classes. Whoa, you say, that is too much work! *Au contraire*, *mon cher ami*. If you take advantage of the opportunities for code reuse that inheritance provides, there is not necessarily much work to do for each class. That is why we say "minimum": you will definitely need to define one or more abstract classes containing common code, in order to reduce duplication.

Here is a list of the eighteen concrete types you'll complete:

Ten arithmetic, relational, and logical expressions (All these classes implement the interface `Expression)`:

```
AopAdd
AopDivide
AopMultiply
AopNegation
AopSubtract
LopAnd
LopNot
LopOr
RopEqual
RopLessThan
```

Two expression types for values (also implements the `Expression` interface):

```
Identifier
Literal
```

One expression representing function calls (also implements the `Expression` interface):

```
CallExpression
```

Five classes that implement the `Instruction` interface:

```
AssignmentInstruction
BlockInstruction
IfInstruction
OutputInstruction
WhileInstruction
```

All of your code goes in the package **`hw4`**.

Expressions

An *expression* is a program element that has a value. Expressions are formed by combining other expressions using operators and parentheses. Expressions in this project can be *arithmetic, relational,* or *logical,* though they all behave essentially the same way. Every type representing an expression implements the `Expression` interface, which includes the method **`eval()`**.

Arithmetic expressions

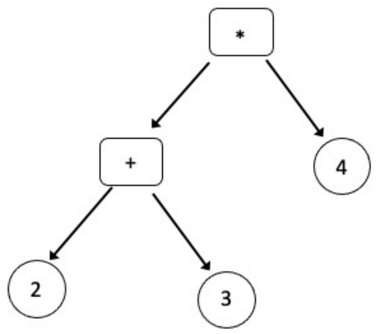
The simplest arithmetic expressions are *literals* and *identifiers* (variables). For example, in programming you might write **`2 + 3`**.

This is an expression with value 5. It is composed of two literals and the binary operator +. Likewise an expression such as

```
(2 + 3) * 4
```

is composed from the *child expression* 2 + 3 and the literal 4. To find its value, you first evaluate the left-hand side (5) and then the right-hand side (4) and then apply the operator * to get the value 20. *Every binary operator works in essentially this same way.*

An expression such as `(2 + 3) * 4` can be conveniently represented by *nodes* arranged in a tree-like structure:



From this you can see that evaluating an expression is like a recursive process. So if you have a tree-like structure as above representing an expression, to return the value of the expression at the root of the tree you would:

> *call eval() on the left child to get its value*
> *call eval() on the right child to get its value*
> *multiply the results together*

Of course, calling eval() on the left child would in turn cause eval() to be called on *its* two children, which is why it looks like a recursive process.

For arithmetic expressions, there are four binary operators: +, -, *, and /. Expressions built from these operators are represented by the four classes

```
AopAdd
AopDivide
AopMultiply
AopSubtract
```

An arithmetic operator always has two child nodes to represent the left side and the right side of the operation. There is also one *unary* arithmetic operator, referred to as the "unary minus" which is like the minus sign you see in something like -7 or -(2 + 3) * 4. An expression made from a unary minus is represented by the class **`AopNegation`**. It has only one child expression, and its **`eval()`** method returns the negative of that expression's value.

A literal number in such an expression is represented by the class **`Literal`**. Evaluating a **`Literal`** node just returns its numeric value.

Example

Here is a little test case illustrating the expected behavior of the classes mentioned above. (You can also find this example in the default package of the sample code. We are using two add nodes to make the expression (2 + 3) + 4, rather than. (2 + 3) * 4 as in the example above, only because the sample code includes a working version of **`AopAdd`**.) Note that the eval() method requires an argument of type **`Scope`**, which is explained later. (It is not actually being used here; as long as the expression only uses literals (no variables), the scope does not matter.)

```
// A literal value.
Expression e0 = new Literal(2);
Expression e1 = new Literal(3);

// a literal always evaluates to itself
// (Note the scope is not actually being used here)
Scope env = new Scope();
System.out.println(e0.eval(env));   // expected 2
System.out.println(e1.eval(env));   // expected 3

// create the expression 2 + 3
Expression aSum = new AopAdd(e0, e1);
System.out.println(aSum.eval(env)); // evaluates to 5

// create the expression (2 + 3) + 4
Expression e2 = new Literal(4);
Expression anotherSum = new AopAdd(aSum, e2);
System.out.println(anotherSum.eval(env)); // expected 9
```

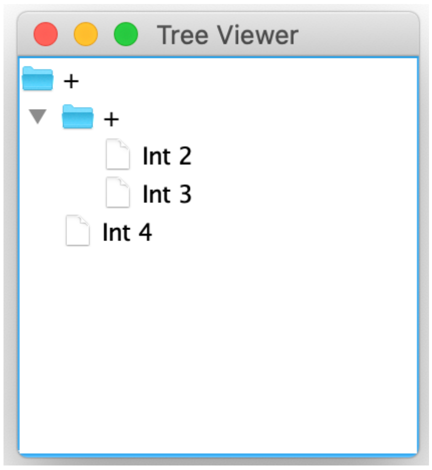
We will refer to the sub-expressions **`e0`** and **`e1`** as the *children* of the sum expression **`aSum`**. Likewise **`aSum`** and **`e2`** are the children of the expression **`anotherSum`**. The order will be important: the left side is child 0, and the right side is child 1.

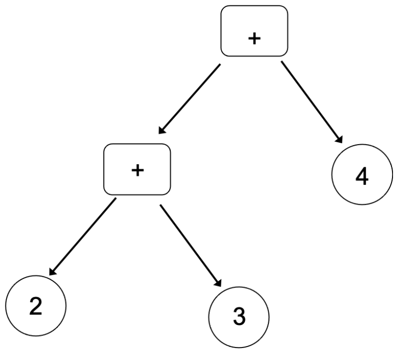
Creating tests and examples by hand, as in the example above, can be extremely tedious. Fortunately, we are also providing you with some utilities, a *parser* and a *viewer*, to make this part much easier.

The viewer works like this: if we add the line,

```
TreeViewer.start(anotherSum);
```

a window like this should pop up. You can click on the little triangles to expand a node and see its children. Children are listed top-to-bottom rather than left-to-right, but the structure is the same.





Relational expressions

There are two relational operators: **==** and **<** . These are represented by the classes **`RopEqual`** and **`RopLessThan`**, respectively. Any two arithmetic expressions can be combined by a relational operator into a logical expression (an expression with a value of true or false). In our code, there is no separate boolean type as in Java. Instead the value 1 is used for "true" and the value 0 is used for "false". A relational operator always has two children, where child 0 is the left side and child 1 is the right side.

Logical expressions

Expressions with a true or false value can be combined using logical operators denoted by the symbols &&, ||, and ! as in Java. The first two are binary operators and are represented by the types **`LopAnd`** and **`LopOr`**, respectively. Logical "not" is represented by the class **`LopNot`**. Evaluation is similar to the arithmetic or relational expressions, and the result is always 0 or 1.

Note: even though we mentally distinguish arithmetic expressions and logical expressions, the language always treats a nonzero value as "true" and a zero value as "false". Hence a (silly) expression such as "2 && 3" would evaluate to 1.

Variables and the Scope

Expressions can also contain *variables*. For example, think about the value of an expression such as `x + 4`. No doubt you have written programs in which the variable "x" occurred in several places. So what is the value of `x + 4` ? Its value depends on the context in which it occurs. We will call that context the *scope*. A scope is just a list of variable names and their corresponding values. In order to determine the value of an expression containing a variable, the scope needs to be provided. That is why the `eval()` method of `Expression` includes an argument representing the scope.

In our code, the scope will be represented by the class `api.Scope`. It is pretty simple; it has just three methods (see the javadoc for details.)

```
void put(String name, int value)
int get(String name)
void remove(String name)
```

Once you have the idea of a scope, it makes sense to talk about expressions containing variables. A variable is represented by the class `Identifier`. An `Identifier` is just a name. Its `eval()` method returns the value associated with that name, *in the current scope*.

Example

```
// create an expression with a variable, say x + 4
Identifier id = new Identifier("x");
Expression sumWithVar = new AopAdd(id, new Literal(4));

// to evaluate it, we need to provide a scope in which x has a value
Scope env = new Scope();
env.put("x", 42);
System.out.println(sumWithVar.eval(env));  // expected 46
```

Instructions

An *expression* is a value, whereas an *instruction* is a statement intended to perform some action. In our code, instructions are represented by the interface `Instruction`, which includes the method `execute()`. We have two simple instruction types, one for output and one for assignment, plus three compound instruction types for conditionals, loops, and blocks.

An instance of `OutputInstruction` has one child, which is an instance of

`Expression`. Invoking the **`execute()`** method first evaluates the expression and then prints the result to the console using **`System.out.println()`**. Note that the expression may contain variables, so a scope is provided as an argument to **`execute()`**.

An instance of **`AssignmentInstruction`** has two children: the first is an **`Identifier`**, and the second is an **`Expression`**. Invoking **`execute()`** first evaluates the expression, and then updates the given scope such that the name of the identifier is associated with the expression's value. (The name does not have to be in the scope to start with.)

In the previous example, we could have set a value for x in the scope by executing an assignment:

```
// we could also give x a value by executing an assignment statement
// (recall that anotherSum was the expression (2 + 3) + 4).
AssignmentInstruction a = new AssignmentInstruction(id, anotherSum);

// executing the assignment will cause the expression to be evaluated
// and stored as the value of x
a.execute(env);
System.out.println(sumWithVar.eval(env));  // 9 + 4 = 13
```

There are also three *compound* instruction types. An **`IfInstruction`** has three children: an **`Expression`** and two instances of **`Instruction`**. First the expression is evaluated. If it is "true" (nonzero) then **`execute()`** is invoked on the first instruction. If "false" (zero), then **`execute()`** is invoked on the second instruction. Note that the second instruction (the "else" branch) is always required; to make an "if" without an "else", use an empty **`BlockInstruction`** for the second part (i.e., put a pair of empty braces **`{}`** in the code).

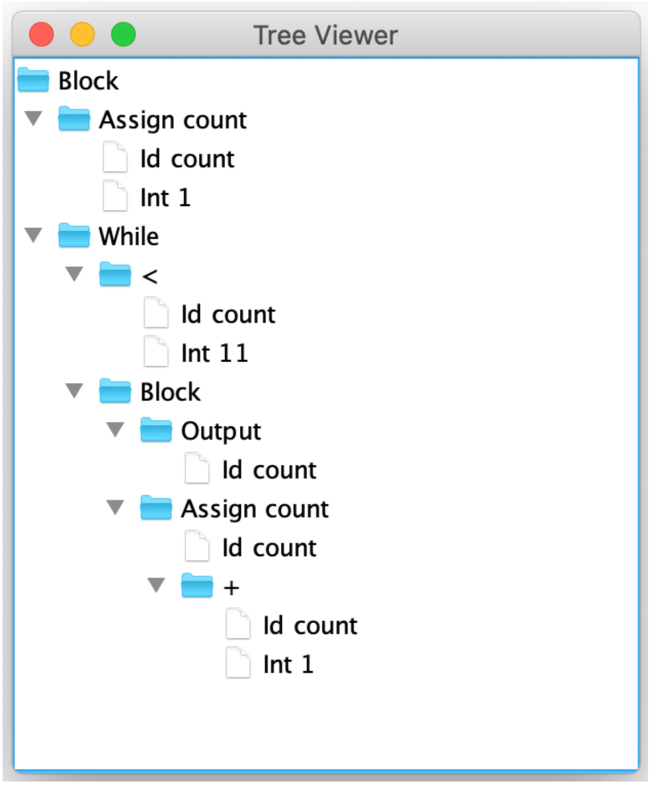
A **`LoopStatement`** has two children, an **`Expression`** and an **`Instruction`**, in that order. (Typically the instruction is a **`BlockInstruction`**.) Not surprisingly, the behavior of its **`execute()`** method is

```
evaluate the expression
while the expression is true
      execute the instruction
      evaluate the expression again
```

Finally, a **`BlockInstruction`** is a list of zero or more **`Instruction`** objects. The number of children is not fixed. New instructions are added using the public **`addStatement()`** method. The behavior of **`execute()`** for a block is to invoke **`execute()`** on each child instruction, in the order they were added. Here is an example of making a loop instruction that prints numbers 1 through 10:

```
// count = 1
Identifier indexVar = new Identifier("count");
Instruction initialize =
    new AssignmentInstruction(indexVar, new Literal(1));

// count < 11
Expression test = new RopLessThan(indexVar, new Literal(11));

// output(count)
Instruction display = new OutputInstruction(indexVar);

// count = count + 1
Expression sum = new AopAdd(indexVar, new Literal(1));
Instruction increment = new AssignmentInstruction(indexVar, sum);

// create the loop statement using a block for the body
BlockInstruction block = new BlockInstruction();
block.addStatement(display);
block.addStatement(increment);
Instruction loop = new WhileInstruction(test, block);

// combine initialization and loop into a block
BlockInstruction main = new BlockInstruction();
main.addStatement(initialize);
main.addStatement(loop);

// now run it
Scope env = new Scope();
main.execute(env);
```

The example above would look like this in the tree viewer:



```
Block
  Assign count
    Id count
    Int 1
  While
    <
      Id count
      Int 11
    Block
      Output
        Id count
      Assign count
        Id count
        +
          Id count
          Int 1
```

The language *Junebug*

The tree of instructions and expressions we have created above is essentially a computer program, one that you can execute by calling the **`execute()`** method on the root node. It would be a lot more sensible if we had a *programming language* that could be used to create a tree like that to execute. And we do! It is a simple language called *Junebug*, and in this language the program above could be written (see the file count.txt in the sample code):

```
count = 1;
while (count < 11)
{
  output(count);
  count = count + 1;
}
```

You can see that the Junebug syntax is superficially like Java or C, where curly braces delimit blocks of code. The language has only one type (int). There are five arithmetic operators (**`+`**, **`-`**, **`*`**, **`/`** and *unary minus*). The value of a relational or logical expression is always 0 for "false" and 1 for "true". Any int value can be treated as a boolean, where nonzero values are considered "true" and zero is considered "false". There are only two relational operators, **`==`** and **`<`**, but since we do have boolean operators **`(&&`**, **`||`**, **`!`**) as in Java, we can fake the rest. Variables come into existence when assigned a value; there are no separate variable declarations. The keyword **`output`** causes its argument value to be printed on the console.

There are **`while`** loops, but no for-loops. The braces are always required for a **`while`** instruction. There are conditional statements using keywords **`if`** and **`else`**. Unlike Java, the **`else`** block is always required, and the braces are always required.

Finally, can define functions using the **`function`** keyword, followed by an identifier, a parameter list, and a statement block. A function is called using its name, supplying a matching number of expressions as arguments. A function must contain *exactly* one return statement, and it must occur at the end.

Below is a more involved example illustrating most of the features described above (see primes.txt in the sample code). This prints out all prime numbers less than **`max`**.

Notice that the variable **`max`** is being used before being assigned a value. In order for this to work, we have to make sure that the Scope we pass in has that variable already defined. (This is the only way we can provide *input* to a Junebug program.)

```
n = 2;

while (n < max)
{
  if (isPrime(n))
  {
    output(n);
  }
  else {} // else block always required, whether you want it or not
  n = n + 1;
}

function isPrime(n)
{
  d = 2;
  possible = 1;  // we have boolean expressions, but not boolean variables

  while (!(n < d * d) && possible == 1)
  {
    remainder = n - d * (n / d);  // we don't have a mod operator :(
    if (remainder == 0)
    {
      possible = 0; // can't be prime
    }
    else {}
    d = d + 1;
  }
  return possible; // 1 if prime, 0 if not
}
```

The example above also illustrates the general structure of a Junebug program: a sequence of one or more instructions, followed by zero or more function definitions.

The parser

Please note: The parser is useful for creating examples, but is still in an experimental state. The reporting of syntax errors can be particularly confusing. There may be bugs that we don't know about yet. If you suspect a parser bug, post a question on Piazza (with an example to reproduce it) and we'll try to figure it out. *Parser bugs will not prevent your own code from working correctly!*

The language syntax is defined by a *parser*, that is, a utility that takes a stream of text such as the sample programs above, and figures out how to construct the tree of nodes that represents its structure. It can also be used to parse individual expressions. The text of the program or expression can come from a string or from a file.

The three basic methods are static methods of the class **`util.ParserUtil`**.

```
public static Expression parseExpression(String expr)
public static Junebug parseProgramFromString(String programText)
public static Junebug parseProgramFromFile(String filename)
```

The **`Junebug`** object returned by the latter two methods is just a node whose children consist of a **`BlockInstruction`** and a sequence of **`Function`** objects. It has a method called **`run()`** that just extracts the block and invokes **`execute()`** on it.

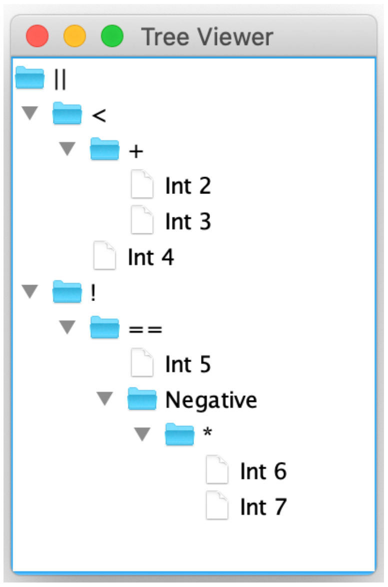
For example, just using what we have available in the sample code, we could write

```
String s = "(fred + 3) + (george + (4 + 5))";
Expression e = ParserUtil.parseExpression(s);
```

The expression could be viewed in the tree viewer, or evaluated (using an appropriate scope in which variables **`fred`** and **`george`** are defined).

When you get all the arithmetic, relational, and logical expression types working (but not necessarily the **`CallExpression`** or instruction types), you could try an example like this:

```
String t = "(2 + 3) < 4 || !(5 == -(6 * 7))";
Expression e = ParserUtil.parseExpression(t);
// optionally, start the tree viewer
TreeViewer.start(e);
System.out.println("Value: " + e.eval(new Scope()));
```



```
||
  <
    +
      Int 2
      Int 3
    Int 4
  !
    ==
      Int 5
      Negative
        *
          Int 6
          Int 7
```

Once you have everything implemented and working in the hw3 package, you could try out something like the primes.txt program (see primes.txt in the sample code).

```
p = ParserUtil.parseProgramFromFile("primes.txt");

// optionally, look at the syntax tree
TreeViewer.start(p);

// this program needs a variable named 'max' in the environment
Scope env = new Scope();
env.put("max", 100);
p.run(env);
```

"Raw" mode

For the three **`ParserUtil`** methods above, the parser will attempt to construct the tree using the node types that you are implementing in the **`hw4`** package. If your program or expression requires a node type that you don't have implemented yet, the parser will fail when it tries to instantiate and/or use the type.

To experiment with the language before you have all the node types implemented, it's also possible to run the parser in "raw" mode. What this means is that it will construct a tree using a very simplistic node type that displays basic information about the node, but has no capability to evaluate or execute it. This is useful because a) running the parser will check the syntax of your program or expression, and b) you can still view it in the tree viewer.

The three relevant methods are

```
public static ProgramNode parseExpressionRaw(String expr)
public static ProgramNode parseProgramFromStringRaw(String programText)
public static ProgramNode parseProgramFromFileRaw(String filename)
```

The sample code includes a file **`TryParserOnly.java`** with some examples.

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 as in homework 3, **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 specifications, 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 everything in your hw4 package. **Please check this carefully. In this assignment you will be including one or more abstract classes of your own, in addition to the 18 required classes, so make sure they have been included in the zip file.**

About the sample code

The sample code **`hw4_skeleton.zip`** includes a complete skeleton of the eighteen classes you are writing. It is distributed as a complete Eclipse project that you can import. It should compile without errors out of the box.

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 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. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
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 Eclipse Package Explorer, navigate to the src folder of the new project.
5. Drag the **`hw4`**, and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.
6. Drag the two sample files from the default package into the **`src`** folder.
7. Drag the three jar files: antlr3.jar, parser.jar, and util.jar, into the *project directory,* and add them to the build path
8. Optionally, drag the sample files count.txt and primes.txt into the project directory

What's in the project

The `hw4` directory includes a basic skeleton for all eighteen class you are to implement. Each includes a constructor that is declared, but unimplemented. Note:

- Do not change the constructor declarations, they will be called by the parser.
- `BlockInstruction` includes a declaration for a public method `addStatement`. Don't change the declaration, it will be called by the parser.
- `CallInstruction` includes a declaration for a public method `setFunction`. Don't change the declaration, it will be called by the parser.
- Do not change anything in the `api` package.

What's in the three jar files (fyi)

- **util.jar** includes just the class `util.ParserUtil`.
- **parser.jar** includes the code for the parser itself, the declaration of the `parser.ProgramNode` interface, and the `viewer.TreeViewer` code.
- **antlr3.jar** is the tool that was used to generate the parser code from the language grammar, and the parser still depends on some of its types.

The interface ProgramNode

The `ProgramNode` interface is the subinterface of `Expression` and `Instruction`. You can see the Javadoc online, but it just specifies four methods for accessing child nodes, and also for obtaining textual information needed by the tree viewer

```
String getLabel();
String getText();
ProgramNode getChild(int i);
int getNumChildren();
```

It also includes a "default method" called `makeString` that combines the label and text information into a string. This is useful, because any class that implements the interface can easily override `toString()` (i.e. from java.lang.Object) with the declaration:

```
@Override
public String toString()
{
  return makeString();
}
```

*It is a **requirement** that you include or inherit this toString method in all eighteen classes. (This is where the labels in the tree viewer come from.)*

Special documentation and style requirements

- You may not use `protected`, `public` , or package-private instance variables. Normally, instance variables in a superclass should be initialized by an appropriately defined superclass constructor. You can create additional `protected` getter/setter methods if you really need them.

Style and documentation

See the special requirements above.

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

- 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**.
- 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.
    - **When a class implements or overrides a method that is already documented in the supertype (interface or class) you normally do not need to provide additional Javadoc,** unless you are significantly changing the behavior from the description in the supertype. You should include the `@Override` annotation to make it clear that the method was specified in the supertype.

- 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.

- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (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.)
    - Internal comments always *precede* the code they describe and are indented to the same level.

- 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 **`assignment4`**. 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 **`assignment4`**. *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

0. Be sure you have done lab 8, especially checkpoint 2.

1. There are working implementations of four of the classes in the **`hw4`** package that are included in the sample code as examples:

```
AopAdd
Literal
Identifier
AssignmentInstruction
```

These are good examples to study, and they allow us to run the simple test cases included in the default package, but ***you will eventually need to reimplement them to reduce the amount of duplicated code***. These examples "directly" implement the interface, that is, adding the declaration "**`implements Expression`**" and then adding implementations for the required methods. You will more likely implement the interface "indirectly" by *extending* an abstract class that already implements it, or that at least implements the **`ProgramNode`** subinterface.

2. Try implementing something similar to **`AopAdd`**, such as **`LopAnd`**. You can do this at first by copying and pasting. Make up a test case: the "and" of any two "true" (nonzero) children is 1, but if either of the children is "false" (zero), the result should be 0. For example,

```
    Expression e0 = new Literal(0);
    Expression e1 = new Literal(42);
    Expression e = new LopAnd(e1, e1);
    System.out.println(e.eval(new Scope()));  // expected 1
    e = new LopAnd(e0, e1);
    System.out.println(e.eval(new Scope()));  // expected 0
```

3. Before you go hog wild implementing more of the operators by cutting and pasting code, as above, think about the common code you see in **`AopAdd`**, **`Literal`**, **`Identifier`**, and **`AssignmentInstruction`**. How can some, or all, of it be moved into an abstract superclass? Remember that you can define the superclass to have whatever constructors it needs to initialize itself. Does it seem to make sense to have just one superclass for all eighteen concrete types, or are there other similarities to exploit? (For example, there are eight binary operators – do they have common aspects of their implementation that would be different from, say, **`BlockInstruction`** ?)

4. There are a few node types already implemented in the **`api`** package: **`Junebug`**, **`ArgList`**, **`ParameterList`**, and **`Function`**. Read the code for those too (fortunately it's pretty simple). Also take a look at the **`Scope`** class (also short). The type **`DefaultNode`** is used as a return value when **`getChild`** is called with an out of range index (e.g. see **`AopAdd`**).

5. As you ponder step 3, you can work in parallel on thinking about the instruction types. (We would suggest leaving **`CallExpression`** until the last, since it is likely the trickiest.). Again, a reasonable strategy to start out is to do what we did in step 2 for **`LopAnd`**, i.e., pick an instruction type such as **`OutputInstruction`**, add the declaration "**`implements Instruction`**", and make it work as specified. There will be a lot of duplication that you'll have to remove later, but this is a good way to a) understand what the code is supposed to do in the first place, and b) identify what the common code is. Of course, you'll need a simple test case, maybe like this:

```
Expression e = new AopAdd(new Literal(2), new Literal(3));
OutputInstruction out = new OutputInstruction(e);
ProgramNode test = out.getChild(0);  // child 0 should be e
System.out.println(e == test);       // expected true
out.execute(new Scope());            // expected: 5 printed to console
```

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_hw4.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains a) the eighteen files you started with in the hw4 directory, and b) whatever abstract classes you added in your design.

Please LOOK at the file you upload and make sure it is the right one and contains everything it should!

Submit the zip file to Canvas using the Assignment 4 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 **hw4**, which in turn should contain the required .java files. You can accomplish this by zipping up the **src** directory within 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.