Problem Set 1

Bookmark this page

About the Problem Sets

Every problem set has two submission points: a beta submission and a final submission. The initial release of the problem set (e.g., this page about Problem Set 1) includes tests for the beta submission. After the beta deadline passes, additional tests will be released for you to download from edX. You can then run these final tests against your solution, fix any new problems that are found, and then submit your revised program for the final deadline.

For the problem sets, all the test cases are given to you, and you'll run the tests on your own machine before uploading your problem set, so you'll always know which tests are passing and which are failing before you submit. There are no hidden tests for the problem sets. You'll get checked off for the beta submission if it passes most of the beta tests (typically at least 80-85%), and you'll get checked off for the final submission if it

passes most of the beta + final tests.

Install Java

Java version 8 is required for this course. Don't just use whatever Java might already be installed on your system. Please install version 8.

1. Go to the Java JDK version 8 installation page, and download and install the JDK for your platform. First click on the download button for Java Platform (JDK):

Java Platform (JDK) 8u101 / 8u102

Find the first "Java SE Development Kit 8" section on the installation page. It doesn't matter whether you install 8u101, 8u102, or some other number, as long as it's 8. Then choose the right download for your computer's operating system:

Java SE Development Kit 8u101

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement **Decline License Agreement**

Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	77.77 MB	jdk-8u101-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	74.72 MB	jdk-8u101-linux-arm64-vfp-hflt.tar.gz
Linux x86	160.28 MB	jdk-8u101-linux-i586.rpm
Linux x86	174.96 MB	jdk-8u101-linux-i586.tar.gz
Linux x64	158.27 MB	jdk-8u101-linux-x64.rpm
Linux x64	172.95 MB	jdk-8u101-linux-x64.tar.gz
Mac OS X	227.36 MB	jdk-8u101-macosx-x64.dmg
Solaris SPARC 64-bit	139.66 MB	jdk-8u101-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	98.96 MB	jdk-8u101-solaris-sparcv9.tar.gz
Solaris x64	140.33 MB	jdk-8u101-solaris-x64.tar.Z
Solaris x64	96.78 MB	jdk-8u101-solaris-x64.tar.gz
Windows x86	188.32 MB	jdk-8u101-windows-i586.exe
Windows x64	193.68 MB	jdk-8u101-windows-x64.exe

For Windows, you have two choices: Windows x86 (which is for **32-bit** Windows) and Windows x64 (which means **64-bit**). If you aren't sure which one you have, see How Do I Know if I'm Running 32-bit or 64-bit Windows?. If that page doesn't help, then choose Windows x86, because it works on both 32-bit and 64-bit Windows, just a little slower.

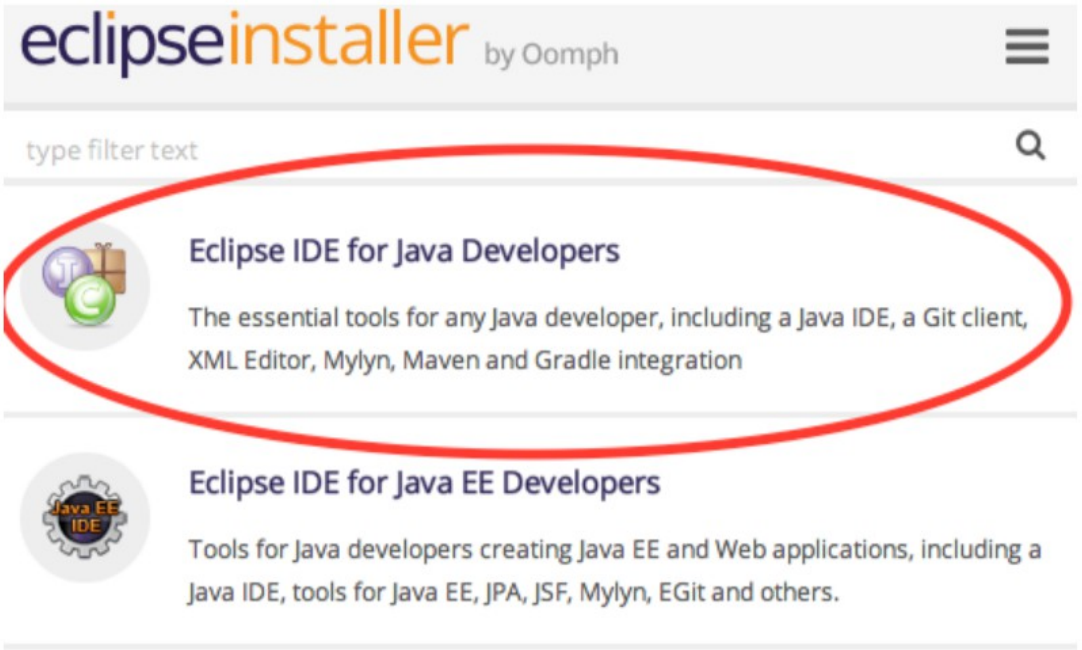
You don't need any of the other downloads on that page, like demos, samples, NetBeans, or Java EE. Just install Java SE Development Kit.

2. After downloading the Java installer, run it and follow the prompts to install Java on your computer.

Install Eclipse

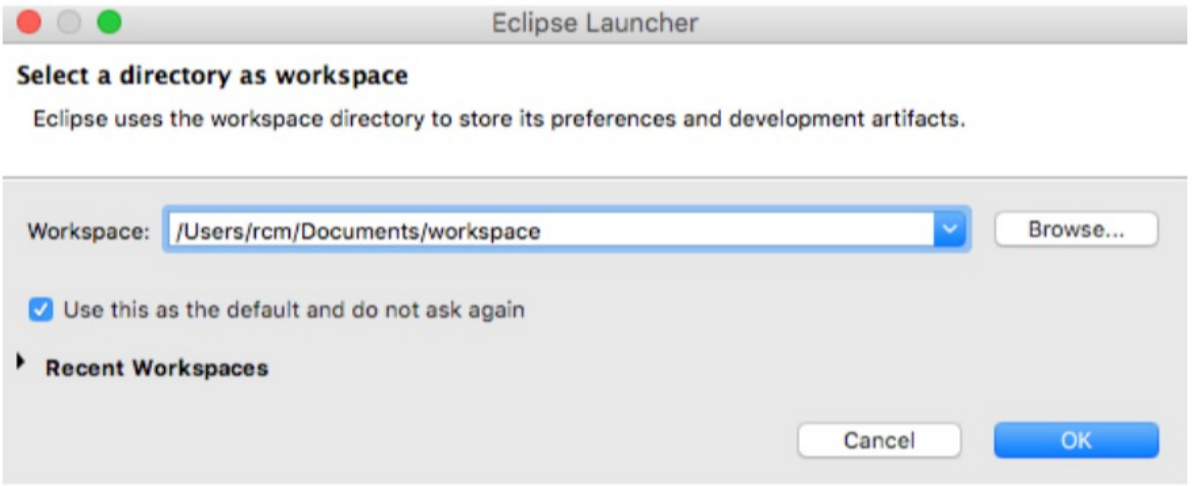
1. Go to the Eclipse Download page and click on the orange Download button to get the latest Eclipse release for your platform. The latest version in 2016 is called Eclipse Neon.

2. After downloading the installer, run the installer. It will give you a choice like this:

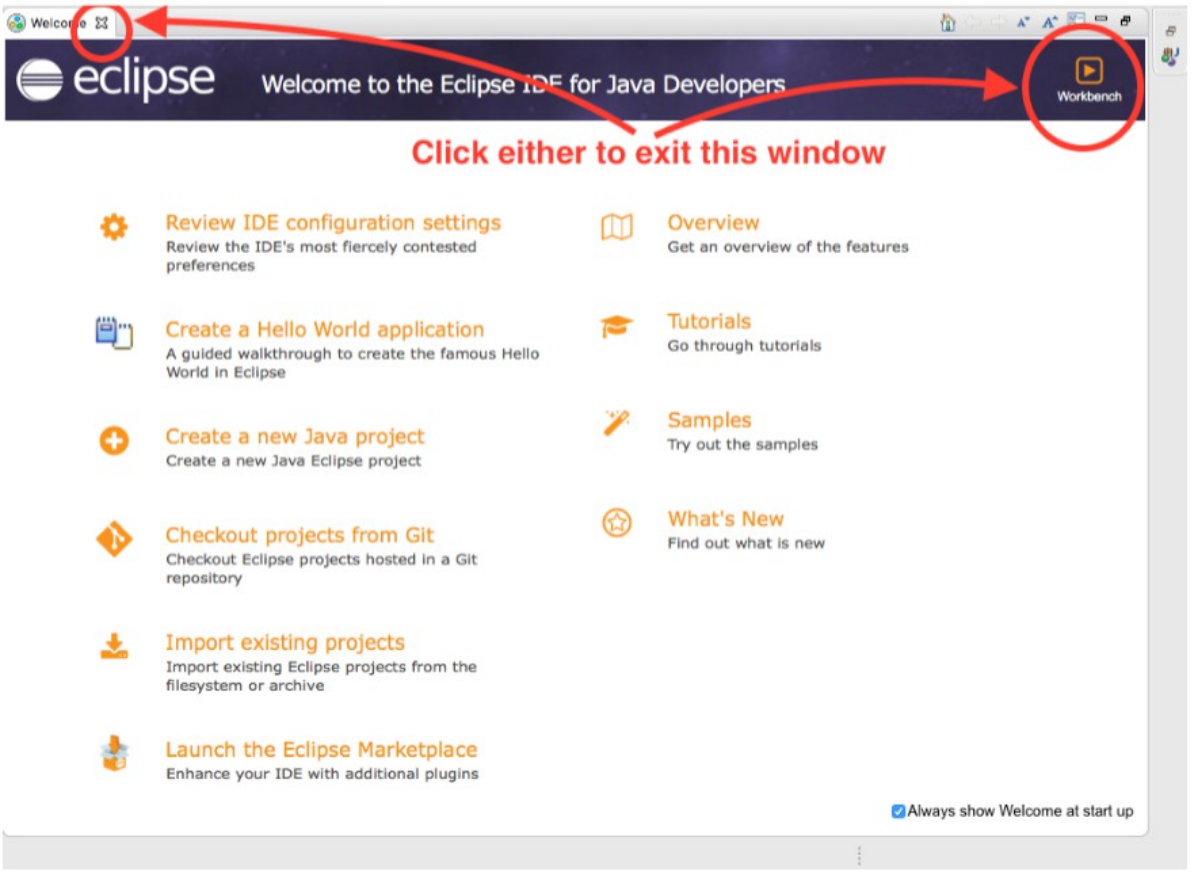


Choose the first option, Eclipse IDE for Java Developers, and follow the prompts to install Eclipse on your computer.

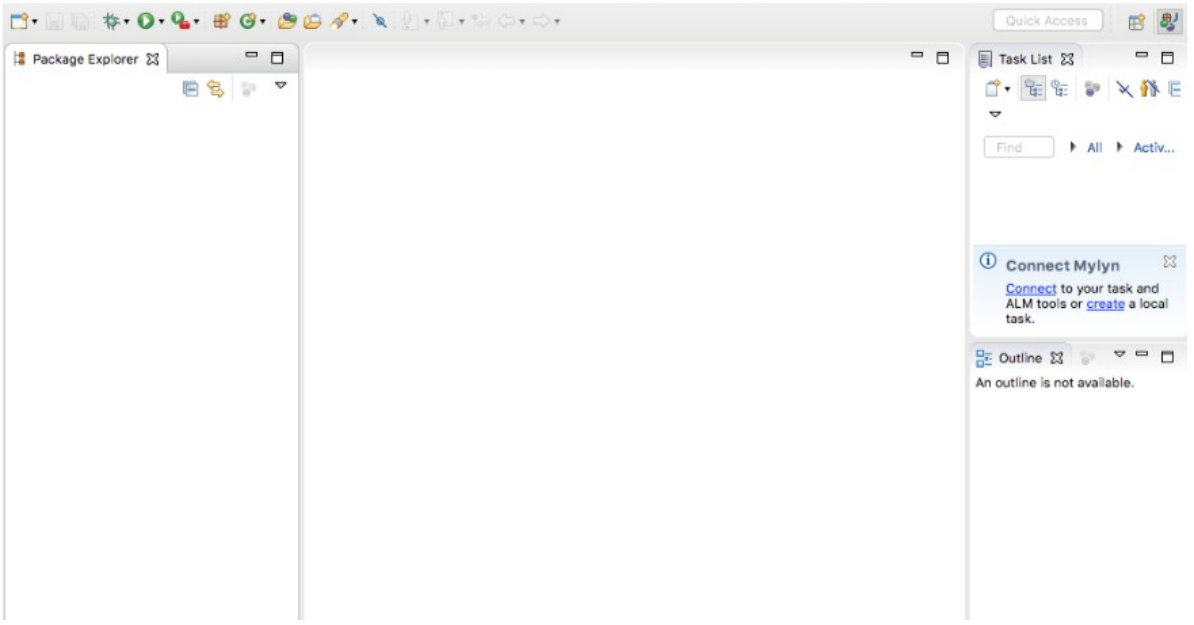
3. Run Eclipse. The first time you run it, Eclipse will ask where you want to put your workspace, which is where all your Java files will be stored. You can just accept its suggestion, and check "Use this as the default and do not ask again":

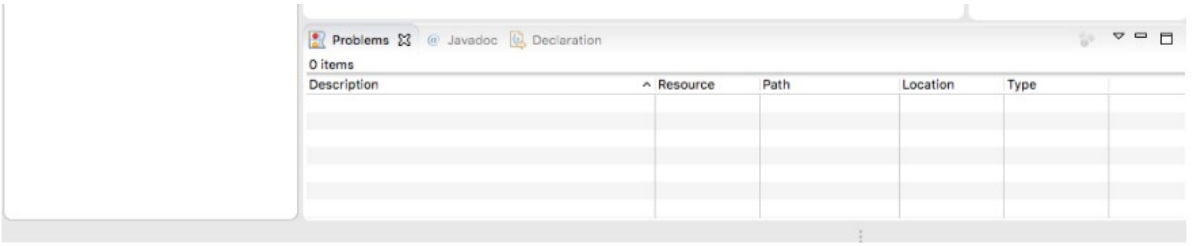


4. Finally the Eclipse window will appear, with a Welcome tab showing:



Close this Welcome tab, either using the X next to Welcome, or using the Workbench button in the upper right, so that you get to the Eclipse workbench:





Configure Eclipse

Eclipse is a powerful, flexible, and occasionally frustrating set of tools for developing and debugging programs, especially in Java.

> *Note: if you prefer, you can do the problem sets in another Java development environment, or using command-line tools. The problem sets require compiling Java, running JUnit, and running Ant, so if you know how to do those things some other way, go for it. The rest of these instructions will talk only about using Eclipse.*

When you run Eclipse, you will be prompted for a "workspace" directory, where Eclipse will store its configuration and metadata. The default location is a directory called `workspace` in your home directory. You should not run more than one copy of Eclipse at the same time with the same workspace.

On the left side of your Eclipse window is the Package Explorer, which shows you all the projects in your workspace.

1. Open Eclipse preferences.

   Windows & Linux: go to *Window → Preferences*.

   OS X: go to *Eclipse → Preferences*.

2. Make sure Eclipse is configured to use **Java 8**.

   a. In preferences, go to *Java → Installed JREs*. Ensure that "Java SE 8" or "JDK 1.8.0_71" is the only one checked. If it's not listed, click Search.

b. Go to *Java → Compiler* and set "Compiler compliance level" to 1.8. Click OK and Yes on any prompts.

3. Make sure **assertions are always on**. Assertions are a great tool for keeping your code safe from bugs, but Java has them off by default.

   In preferences, go to *Java → Installed JREs*. Click "Java SE 8", click "Edit…", and in the "Default VM arguments" box enter: `-ea` (which stands for *enable assertions*).

4. Make sure your **Eclipse workspace refreshes automatically** whenever files are changed on the filesystem. This will be important when you run the grader script, so that you see the output files it generates.

   In preferences, go to *General → Workspace*. Turn on both checkboxes that talk about refresh: "Refresh using native hooks or polling" and "Refresh on access."

5. **Tabs**. If you configure the editor to use spaces instead of tabs, then your code will look the same in all editors regardless of how that editor displays tab characters.

   In preferences, go to *Java → Code Style → Formatter*. Click the "Edit…" button next to the active profile. In the new window, change the Tab policy to "Spaces only." Keep the Indentation size and Tab size at 4. To save your changes, enter a new "Profile name" at the top of the window and click OK.

Download and Import the Problem Set Code

1. Download the starting code for this problem set:

   ps1.zip

2. Import the code into Eclipse:

   a. In the Eclipse menubar, choose File → Import...

   b. In the Select dialog, open General and choose Existing Projects Into Workspace.

c. In the Import Projects dialog, choose Select archive file, then click Browse and use the file dialog to find where you downloaded ps1.zip.

d. Still in the Import Projects dialog, make sure ps1-expressivo is listed in the Projects, and that the checkbox next to it is checked

e. On "Import projects," make sure ps1-expressivo is checked.

> If ps1-expressivo is grayed and uncheckable, then the likely reason is that ps1-expressivo already exists in your Package Explorer (perhaps because you are following these directions a second time). You need to cancel the import, delete or rename ps1-expressivo in your Package Explorer, and try the import again.

f. Click Finish. ps1-expressivo should now appear in your Package Explorer.

Problem Set 1: Expressivo

Overview

Wouldn't it be nice not to have to differentiate all our calculus homework by hand every time? Wouldn't it be just lovely to type it into a computer and have the computer do it for us instead? For example, we could interact with it like this (user input in green):

```
> x * x * x
x*x*x

> !simplify x=2
8

> !d/dx
(x*x)*1+(x*1+1*x)*x

> !simplify x=0.5000
.75
```

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal

places.

```
> x * y
x*y

> !d/dy
0*y+x*1
```

In this system, a user can enter either an **expression** or a **command**.

An *expression* is a polynomial consisting of:

- + and * (for addition and multiplication)
- nonnegative numbers in decimal representation, which consist of digits and an optional decimal point (e.g. `7` and `4.2`)
- variables, which are case-sensitive nonempty sequences of letters (e.g. `y` and `Foo`)
- parentheses (for grouping)

Order of operations uses the standard PEMDAS rules.

Space characters around symbols are irrelevant and ignored, so `2.3*pi` means the same as `2.3 * pi`. Spaces may not occur within numbers or variables, so `2 . 3 * p i` is not a valid expression.

When the user enters an expression, that expression becomes the **current expression** and is echoed back to the user (user input in green):

```
> x * x * x
x*x*x

> x + x * x * y + x
x + x*x*y + x
```

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

A *command* starts with `!`. The command operates on the current expression, and may also update the current expression. Valid commands:

> **`d/dvar`**
> produces the derivative of the current expression with respect to the variable *var*, and updates the current expression to that derivative
>
> **`simplify var1=num1 ... varn=numn`**
> substitutes num_i for var_i in the current expression, and evaluates it to a single number if the expression contains no other variables; does **not** update the current expression

Entering an invalid expression or command prints an error but does not update the current expression. The error should include a human-readable message but is not otherwise specified.

More examples:

```
> x * x * x
x*x*x

> 3xy
ParseError: unknown expression

> !d/dx
(x*x)*1+(x*1+1*x)*x

> !d/dx
(x*x)*0+(x*1+1*x)*1+(x*1+1*x)*1+
(x*0+1*1+x*0+1*1)*x

> !d/d
ParseError: missing variable in
derivative command

> !simplify
(x*x)*0+(x*1+1*x)*1+(x*1+1*x)*1+
(x*0+1*1+x*0+1*1)*x

> !simplify x=1
6.0

> !simplify x=0 y=5
0
```

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

The three things that a user can do at the console correspond to three provided method specifications in the code for this problem set:

- `Expression.parse()`
- `Commands.differentiate()`
- `Commands.simplify()`

These methods are used by `Main` to provide the console user interface described above.

Problem 1: we will create the `Expression` data type to represent expressions in the program.

Problem 2: we will create the parser that turns a string into an `Expression`, and implement `Expression.parse()`.

Problems 3-4: we will add new `Expression` operations for differentiation and simplification, and implement `Commands.differentiate()` and `Commands.simplify()`.

Problem 1: Representing Expressions

Define an immutable, recursive abstract data type to represent expressions as abstract syntax trees.

Your AST should be defined in the provided `Expression` interface (in `Expression.java`) and implemented by several concrete variants, one for each kind of expression. Each variant should be defined in its own appropriately-named `.java` file.

Concrete syntax in the input, such as parentheses and whitespace, should not be represented at all in your AST. Parentheses should *not* have corresponding `Expression` variants, and should *not* be stored explicitly in the reps of any `Expression` variant. Instead, the structure of the AST should represent the order of operations directly.

1.1 `Expression`

To repeat, your data type must be **immutable** and **recursive**. Follow the recipe for creating an ADT:

- **Spec**. Choose and specify operations. For this part of the problem set, the only operations `Expression` needs are creators and producers for building up an expression, plus the standard observers `toString()`, `equals()`, and `hashCode()`. We are strengthening the specs for these standard methods; see below.

- **Test**. Partition and test your operations in `ExpressionTest.java`, including tests for `toString()`, `equals()`, and `hashCode()`. Note that we will not run your test cases on other implementations, just on yours.
- **Code**. Write the rep for your `Expression` as a data type definition in a comment inside `Expression`. Implement the variant classes of your data type.

Remember to include a Javadoc comment above every class and every method you write; define abstraction functions and rep invariants, and write checkRep; and document safety from rep exposure.

1.2 `toString()`

Define the `toString()` operation on `Expression` so it can output itself as a string. This string must be a valid expression as defined above. You have the freedom to decide how to format the output with whitespace and parentheses for readability, but the expression must have the same mathematical meaning.

Your `toString()` implementation must be recursive, and must not use `instanceof`.

Use the `@Override` annotation to ensure you are overriding the `toString()` inherited from `Object`.

Remember that your tests must obey the spec. If your `toString()` tests expect a certain formatting of whitespace and parentheses, you must specify this formatting in your spec.

1.3 `equals()` and `hashCode()`

Define the `equals()` and `hashCode()` operations on your AST to implement *structural equality*.

Structural equality defines two expressions to be equal if:

1. the expressions contain the same variables, numbers, and operators;
2. those variables, numbers, and operators are in the same order, read left-to-right;

3. and they are grouped in the same way.

For example, the AST for `1 + x` is *not* equal to the AST for `x + 1`, but it is equal to the ASTs for `1+x`, `(1+x)`, and `(1)+(x)`.

For *n*-ary groupings where *n* is greater than 2:

- Such expressions must be equal to themselves. For example, the ASTs for `3 + 4 + 5` and `(3 + 4 + 5)` must be equal.
- However, whether they are equal or not to different groupings with the same mathematical meaning is *not specified*, and you should choose an appropriate specification and implementation for your AST. For example, you must determine whether the ASTs for `(3 + 4) + 5` and `3 + (4 + 5)` are equal.

For equality of numbers, you have the freedom to choose a reasonable definition. Integers that can be represented exactly as a `double` should be considered equal. For example, the ASTs for `x + 1` and `x + 1.000` must be equal.

Remember: concrete syntax, including parentheses, should not be represented in your AST. Grouping, for example, should be reflected in the AST's structure.

Be sure that AST instances which are considered equal according to this definition and according to `equals()` also satisfy observational equality.

Your `equals()` and `hashCode()` implementations must be recursive. Only `equals()` can use `instanceof`, and `hashCode()` must not.

Remember to use the `@Override` annotation.

Problem 2: Parsing Expressions

Now we will create the parser that takes a string and produces an `Expression` value from it. The entry point for your parser should be `Expression.parse()`, whose spec is provided in the starting code.

Examples of valid inputs:

```
3 + 2.4
3 * x + 2.4
3 * (x + 2.4)
((3 + 4) * x * x)
foo + bar+baz
(2*x    )+    (    y*x    )
4 + 3 * x + 2 * x * x + 1 * x * x * (((x)))
```

Examples of invalid inputs:

```
3 *
( 3
3 x
```

Examples of optional inputs:

```
2 - 3
(3 * x) ^ 2
6.02e23
```

You may consider these inputs invalid, or you may choose to support additional features (new operators or number representations) in the input. However, *your system may not produce an output with a new feature unless that feature appeared in its input*. This way, a client who knows about your extensions can trigger them, but clients who don't know won't encounter them unexpectedly.

2.1 Write a grammar

Write a ParserLib grammar for polynomial expressions as described in the overview. A starting ParserLib grammar file can be found in `src/expressivo/Expression.g`. This starting grammar recognizes sums of integers, and ignores spaces.

See the reading on parser generators for more information about ParserLib.

The API documentation for ParserLib may also be helpful.

2.2 Implement `Expression.parse()`

Implement `Expression.parse()` by following the recipe:

- **Spec**. The spec for this method is given, but you may strengthen it if you want to make it easier to test.
- **Test**. Write tests for `Expression.parse()` and put them in `ExpressionTest.java`. Note that we will not run your tests on any implementations other than yours.
- **Code**. Implement `Expression.parse()` so that it calls the parser generated by your ParserLib grammar. The reading on parser generators discusses how to call the parser and construct an abstract syntax tree from it, including code examples.

A general note on precision: you are only required to handle nonnegative decimal numbers in the range of the `double` type.

2.3 Run the console interface

Now that `Expression` values can be both parsed from strings with `parse()`, and converted back to strings with `toString()`, you can try entering expressions into the console interface.

Run `Main`. In Eclipse, the Console view will allow you to type expressions and see the result. Try some of the expressions from the top of this handout.

Problem 3: Differentiation

The symbolic differentiation operation takes an expression and a variable, and produces an expression with the derivative of the input with respect to the variable. The result does not need to be simplified.

For example, the following are correct derivatives :

x*x*x with respect to **x**
`3 * x * x`

x*x*x with respect to **x**
`x*x + (x + x)*x`

x*x*x with respect to **x**
`( ( x*x )*1 )+( ( ( x*1 )+( 1*x ) )*x )+( 0 )`

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

Incorrect derivatives:

y*y*y with respect to **y**
`3*y^2` *uses unexpected operator*

y*y*y with respect to **y**
`0` *d/dx, should be d/dy*

To implement your recursive differentiation operation, use these rules:

$\frac{dc}{dx} = 0$	$\frac{dx}{dx} = 1$	$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$	$\frac{d(u \cdot v)}{dx} = u\left(\frac{dv}{dx}\right) + v\left(\frac{du}{dx}\right)$

where *c* is a constant or variable other than the variable we are differentiating with respect to (in this case *x*), and *u* and *v* can be anything, including *x*.

3.1. Add an operation to `Expression`

You should implement differentiation as a method on your `Expression` datatype, defined recursively. The signature and specification of the method are up to you to design. Follow the recipe:

- **Spec**. Define your operation in `Expression` and write a spec.
- **Test**. Put your tests in `ExpressionTest.java`. Note that we will not run your test cases on other implementations, just on yours.
- **Code**. The implementation must be recursive. It must not use `instanceof`, nor any equivalent operation you have defined that checks the type of a variant.

3.2 Implement `Commands.differentiate()`

In order to connect your differentiation operation to the user interface, we need to implement the `Commands.differentiate()` method.

- **Spec**. The spec for this operation is given, but you may strengthen it if you want to make it easier to test.
- **Test**. Write tests for `differentiate()` and put them in `CommandsTest.java`. These tests will likely be very similar to the tests you used for your lower-level differentiation operation, but they must use `Strings` instead of `Expression` objects. Note that we will not run your tests on any implementations other than yours.
- **Code**. Implement `differentiate()`. This should be straightforward: simply parsing the expression, calling your differentation operation, and converting it back to a string.

3.3 Run the console interface

We've now implemented the `!d/d` command in the console interface. Run `Main` and try some derivatives in the Console view.

Problem 4: Simplification

The simplification operation takes an expression and an environment (a mapping of variables to values). It substitutes the values for those variables into the expression, and attempts to simplify the substituted polynomial as much as it can.

The set of variables in the environment is allowed to be different than the set of variables actually found in the expression. Any variables in the expression but not the environment remain as variables in the substituted polynomial. Any variables in the environment but

not the expression are simply ignored.

The only required simplification is that if the substituted polynomial is a constant expression, with no variables remaining, then simplification must reduce it to a single number, with no operators remaining either. Simplification for substituted polynomials that still contain variables is underdetermined, left to the implementer's discretion. You may strengthen this spec if you wish to require particular simplifications in other cases.

For example, the following are correct output for simplified expressions:

> **`x*x*x`** with environment **`x=5, y=10, z=20`**
> `125`
>
> **`x*x*x + y*y*y`** with environment **`y=10`**
> `x*x*x+10*10*10`
>
> **`x*x*x + y*y*y`** with environment **`y=10`**
> `x*x*x+1000`
>
> **`1+2*3+8*0.5`** with empty environment
> `11.000`

If the output is an expression, your system may output an equivalent expression, including variations in spacing, parentheses, simplification, and number representation.

If a number, your system may output an equivalent number, accurate to at least 4 decimal places.

Incorrect simplified expressions:

> **`x*x*y`** with environment **`x=1, y=3`**
> `1*1*3` *not a single number*
>
> **`x*x*x`** with environment **`x=2`**
> `(8)` *includes parentheses*
>
> **`x*x*x`** with empty environment
> `x^3` *uses unexpected operator*

Optional simplified expressions:

> **`x*x*y + y*(1+x)`** with environment **`x=2`**
> `7*y`
>
> **`x*x*x + x*x*x`** with empty environment
> `2*x*x*x`

4.1 Add an operation to `Expression`

You should implement simplification as a method on your `Expression` datatype, defined recursively. The signature and specification of the method are up to you to design. Follow the recipe:

- **Spec**. Define your operation in `Expression` and write a spec.
- **Test**. Put your tests in `ExpressionTest.java`. Note that we will not run your test cases on other implementations, just on yours.
- **Code**. The implementation must be recursive (perhaps by calling recursive helper methods). It must not use `instanceof`, nor any equivalent operation you have defined that checks the type of a variant class.

You may find it useful to add more operations to `Expression` to help you implement the simplify operation. Spec/test/code them using the same recipe, and make them recursive as well where appropriate. Your helper operations should not simply be a variation on using `instanceof` to test for a variant class.

4.2 `Commands.simplify()`

In order to connect your simplify operation to the user interface, we need to implement the `Commands.simplify()` method.

- **Spec**. The spec for this operation is given, but you may strengthen it if you want to make it easier to test.

- **Test**. Write tests for `simplify()` and put them in `CommandsTest.java`. These tests will likely be very similar to the tests you used for your lower-level simplify operation, but they should use `Strings` instead of `Expression` objects. Note that we will not run your tests on any implementations other than yours.
- **Code**. Implement `simplify()`. This should be straightforward: simply parsing the expression, calling your simplify operation, and converting it back to a string.

4.3 Run the console interface

We've now implemented the `!simplify` command in the console interface. Run `Main` and try using it in the Console view.

Before you're done

- Make sure you have **documented specifications**, in the form of properly-formatted Javadoc comments, for all your types and operations.

- Make sure you have **documented abstraction functions and representation invariants**, in the form of a comment near the field declarations, for all your implementations.

  With the rep invariant, also say how the type prevents rep exposure.

  Make sure all types use `checkRep()` to check the rep invariant and implement `toString()` with a useful representation of the abstract value.

- Make sure you have satisfied the **Object contract** for all types. In particular, you will need to specify, test, and implement `equals()` and `hashCode()` for all immutable types.

- Use `@Override` when you override `toString()`, `equals()`, and `hashCode()`, to gain static checking of the correct signature.

Also use `@Override` when a class implements an interface method, to remind readers where they can find the spec.

- Make sure you have a thorough, principled **test suite** for every type. Note that `Expression`'s variant classes are considered part of its rep, so a single good test suite for `Expression` covers the variants too.

Run the Autograder

Once your implementation is passing all your own tests, you need to run the automatic grader. The autograder is the file `grader.xml` in your project. It is written as an Ant script, where Ant is an example of a *build management tool*, a kind of tool commonly used in software development to compile code, run tests, and package up code for deployment. Our autograder script does all three of these things. Support for running Ant scripts is built into Eclipse.

To run the automatic grader, right click on `grader.xml` in the Package Explorer, and choose *Run As → Ant Build*. Choose the plain *Ant Build* rather than *Ant Build...*, because you don't need all the options that the Ant Build... dialog box will offer you.

The grading script will run and display output in the Console tab. At the end, it should tell you that it has created two new files: `my-grader-report.xml`, which is the result of running the grading tests, and `my-submission.zip`, which is your problem set code and grader output packaged up and ready for submission to edX.

> *Note: if automatic workspace refreshing is not working for some reason, then `my-grader-report.xml` and `my-submission.zip` may not immediately appear in the Eclipse Package Explorer. Try refreshing the project manually by right-clicking on `ps1-warmup` and choosing Refresh. Then you should see the files appear. If they still aren't there, check the Console tab for errors.*

To view your autograder results, double-click on `my-grader-report.xml`, and you will see the results in your JUnit pane. The `my-grader-report.xml` file is simply the output of running JUnit on the autograder's tests. For now (Problem Set Beta 1), the

autograder tests are identical to the tests you've already been using in QuadraticTest.java, so you should see exactly the same tests passing or failing that you see when you ran JUnit yourself as in the previous section. On future problem sets, the autograder will run different tests.

You can change your code and rerun the autograder as often as you want. You have to double-click on `my-grader-report.xml` each time to see the newest results. You don't need to Refresh the project each time, however. Once `my-grader-report.xml` is visible in the Package Explorer, Eclipse will load the latest version whenever you click on it.

Submit the Beta version of your Problem Set

After the autograder runs, it produces `my-submission.zip` in your Eclipse project. This zip file contains your code and your grading report. To get credit for the problem set, you need to upload this zip file to edX before the deadline.

The Problem Set Beta 1 submission page is the last section of this handout. To find it, click the rightmost button on the section bar at the top of this page:

Previous Next