CS61B, Fall 2002 **Project #1** **P. N. Hilfinger**

Due: Friday, 4 October 2002 at 2400

This first project involves writing a calculator program that can perform *polynomial arithmetic*. We'll do only a very limited set of operations—nothing like production symbolic algebra programs—but it should give you plenty of exercise in Java. You might take a look at §2.5.3 of *Structure and Interpretation of Computer Programs* from CS61A; it's not a section that is regularly used in that class, but you may find it useful.

1 Background

Officially, a *polynomial of degree n in x over D* is an expression of the form

$$\kappa_n x^n + \kappa_{n-1} x^{n-1} + \ldots + \kappa_1 x + \kappa_0, \quad \text{where } \kappa_i \in D \text{ and } \kappa_n \neq 0 \text{ if } n > 0. \tag{1}$$

We call n the *degree* of the polynomial, κ_i the *coefficients,* and x the *indeterminate.* In general, D can be any integral domain[1]. In this project, D will either be $\mathbf{Z}$, the set of integers, or it will be another set of polynomials that don't contain x. So, a polynomial in two indeterminates—say x and y—is a polynomial in x over polynomials in y. That is, we can think of

$$3y^3x^2 - 7y^3x + 2y^2x^2 + 3y + x$$

as the second-degree polynomial in x

$$\begin{aligned} & (y^3 + 2y^2 + 0y + 0)x^2 + (-7y^3 + 0y^2 + 0y + 1)x + (3y + 0) \\ = \quad & (3y^3 + 2y^2)x^2 + (-7y^3 + 1)x + 3y \end{aligned}$$

However, for our simple purposes, we'll just think of a polynomial in $m \geq 0$ indeterminates as being a *sum of terms,* where each term, $T_{i_1 \cdots i_m}$, has the form

$$T_{i_1 i_2 \cdots i_m} = \kappa_{i_1 i_2 \cdots i_m} x_1^{i_1} x_2^{i_2} \cdots x_m^{i_m} \tag{2}$$

for some distinct variables $x_1, \ldots x_m$, some non-negative exponents $i_1 \cdots i_m$, and some constant integer $\kappa_{i_1 i_2 \cdots i_m}$. If $m = 0$ or if all of the $i_j = 0$, then we have simply a constant polynomial.

The operations that we are going to handle in this project are restricted to addition, subtraction, multiplication, and substitution of polynomials. By substitution, I simply mean simultaneous replacement of the indeterminates of a polynomial by other polynomials. Thus, if we have defined

$$p(x, y) \stackrel{\text{def}}{=} 2x^2y + 3xy^2 + 1$$

then

$$\begin{aligned} p(z + 1, q^2) &= 2(z + 1)^2q^2 + 3(z + 1)(q^2)^2 + 1 \\ &= 3q^4z + 3q^4 + 2q^2z^2 + 4q^2z + 2q^2 + 1 \end{aligned}$$

We'll also allow substitutions like

$$P(y, x) = 2y^2x + 3yx^2 + 1$$

where the variables substituted happen to include the ones substituted for.

[1] A set D is an *integral domain,* if it is a commutative ring with cancellation. That is, if it's a set with two operators, $+$ and $\cdot$, such that (1) it is closed under $+$ and $\cdot$, (2) the two operators are commutative and associative, (3) $\cdot$ distributes over $+$, (4) there is a $0 \in D$ that is the identity over $+$ and a $1 \in D$ that is the identity over $\cdot$, (5) there are inverses under $+$ for all members of D, and (6) there is multiplicative cancellation: if $c \neq 0$ and $ca = cb$, then $a = b$. The integers, reals, rationals, and complex numbers are integral domains. But you knew all that, right?

2 Commands

The inputs to your calculator consist mostly of a sequence of polynomial expressions and definitions, with a few other special-purpose commands thrown in. The program basically prints canonicalized versions of the expressions you type, and internally records the definitions for use in later expressions.

Input is *free-form,* which means roughly that blanks, tab characters, and end-of-line characters don't matter except to mark the end of a multi-character symbol, such as an integer. As in Java, we'll use semicolons to mark the ends of commands.

In the command descriptions that follow, we'll use a little notation to keep descriptions short:

- $\mathcal{P}$ denotes a *polynomial symbol,* which is a single capital letter. Polynomial symbols can be defined to denote polynomials.
- $\mathcal{V}$ denotes a *variable,* which is a single lower-case letter.
- $\mathcal{N}$ denotes an *integer constant.* To simplify our life a little, all our integers will be in the range -2^{63} to $2^{63}-1$, the range of a **long** in Java. Furthermore, you may limit any integers in user input to the range $-2^{53}+1$ to $2^{53}-1$, which just happens to be the range of integer values that `StreamTokenizer` can read exactly. Yes, if you are really ambitious, you can get your calculator to handle arbitrarily large integers, just like Scheme, but you don't have to.
- $\mathcal{E}$ denotes a *polynomial,* whose syntax is described below.

Here are the commands:

Comments. A comment starts with a `#` and continues to the end of the line. The calculator simply throws comments away.

Definitions. A command of the form

$$\texttt{DEF } \mathcal{P}\texttt{(}\mathcal{V}_1\texttt{,...,}\mathcal{V}_m\texttt{) = }\mathcal{E}\texttt{;}$$

defines $\mathcal{P}$ to be a polynomial in the $m \geq 0$ distinct variables $\mathcal{V}_1$, ...,$\mathcal{V}_m$. The expression $\mathcal{E}$ may only contain those variables. This replaces any previous definition of $\mathcal{P}$.

Evaluate and Print. A command of the form

$$\mathcal{E}\texttt{;}$$

(that is, an expression followed by a semicolon) simply prints the canonical form of $\mathcal{E}$ (see §4).

Quitting. The commands `QUIT` and `EXIT`, or the end of the input to the calculator, all cause the calculator to terminate.

3 Format of Polynomial Expressions in Input

Expressions are defined recursively as follows:

- An *expression* is a sequence of one or more terms, separated by `+`'s and `-`'s. The first term (only) may be negated.

- A *term* is a sequence of one or more factors, each of which may be followed by an *exponent* of the form `^`n, where n is a non-negative integer constant. A term denotes the result of raising the factors to the powers indicated by the exponents and multiplying them.
- A *factor* is either a variable ($\mathcal{V}$), an integer constant, a substitution, or (recursively) an expression in parentheses.
- A *substitution* is an expression of the form

  $$\mathcal{P}(\mathcal{E}_1, \ldots, \mathcal{E}_m),$$

  and denotes the result of substituting the m expressions $\mathcal{E}_i$ into $\mathcal{P}$, which must be a defined polynomial with m indeterminates.

To prevent ambiguity, there are a couple of additional rules:

- Whenever two integers are juxtaposed, they must be separated by whitespace. For example, if you want to write the product 3 times 2, it would be `3␣2`, not `32`. Likewise, $x^2 2$ is written `x^2␣2`.
- A minus sign following a factor denotes subtraction, not negation. So, for example, `x-3` denotes $x - 3$, not $x(-3)$ or $-3x$.

For example, the following are all expressions

```
31x^2 + y + 1          (x+y^2+1)^2-A(x,y)^2      (-x+1)(x-1)3
x^2 3y                 42                          z
```

4 Canonical Form

When a class of mathematical expressions provides many different ways to write equivalent things, it can get hard to tell when two such expressions are equal. It's easy, though, if the expressions have a *canonical form* to which all can be converted. The term *canonical* here is used in the sense of "standard" (the primary meaning is ecclesiastical: "ordered by canon law"). For us, a canonical polynomial form is either

- an integer constant, or
- A sum of terms, with each term having the form shown in Eqn. (2), such that
  - Each coefficient $\kappa_{i_1 i_2 \cdots i_m} \neq 0$,
  - No two terms have the same sequence of exponents $(i_1 \cdots i_m)$,
  - The variables $x_1, \cdots, x_m$ are in alphabetical order with no repetitions,
  - The terms are listed in decreasing lexicographic order by exponents, which means that terms are ordered in decreasing order of i_1; terms with equal values of i_1 are listed in decreasing order of i_2; terms with equal values of i_1 and i_2 are listed in decreasing order of i_3, etc.
  - Every variable in $x_1, \ldots, x_m$ appears in some term with a positive exponent.

For example, consider the two polynomials $(x^2 - 4y^2)(3y - 6x)$ and $3(-2y + x)(x + 2y)(y - 2x)$. Both of them canonicalize to $-6x^3y^0 + 3x^2y^1 + 24x^1y^2 + -12x^0y^3$, and so they are the same polynomial. Canonicalizing, in other words, consists of expanding out fully and combining coefficients of like terms.

Printed Form. When printing polynomials that are in canonical form, apply a few other conventional abbreviations:

- Display addition of a negative term as subtraction of a positive term.
- Except for the constant term, don't show coefficients of 1 or −1. Write, for example, `x^2-x` rather than `x^2+ -1x` or `x^2-1x`. Write `-x`, not `-1x`.
- Don't write variables with an exponent of 0.
- Don't write an exponent if it is 1.

5 Your Task

The directory `~cs61b/hw/proj1` will contain a few skeleton files that may suggest some structure for this project. Copy them into a fresh directory as a starting point. Use the command

```
cp -r ~cs61b/hw/proj1 mydir
```

to create a new directory called `mydir` containing copies of all our files (with the right protections).

Please read *General Guidelines for Programming Projects* (see the "homework" page on the class web site). To submit your result, use the command '`submit proj1`'. You will turn in nothing on paper.

Be sure to include tests of your program (yes, that is part of the grade). The makefile we provide has a convenient target for running such tests. Our skeleton directory contains a couple of trivial tests, but *these do not constitute an adequate set of tests!* Make up your tests ahead of time and update your makefile to run them.

The input to your program will come from fallible humans. Therefore, part of the problem is dealing gracefully with errors. When the user makes a syntax error, your program should not simply halt and catch fire, but should give some sort of message and then try to get back to a usable state. Your choice of recovery is less important than being sure that your program *does* recover. I don't care, for example, whether you throw away the rest of the line, or instead throw away everything up to the next semicolon, just so long as you do something better than to abruptly exit with a cryptic stack trace.

Be sure to include documentation. This is the first project, so the documentation should be very straightforward: a user's manual explaining how to use your program, and a brief internals document describing overall program structure.

Our testing of your projects (but not our grading!) will be automated. The testing program will be finicky, so be sure that:

- Your makefile is set up to compile everything on the command `gmake` and to run all your tests on the command `gmake test`. The makefile provided in our skeleton files is already set up to do this. Be sure to keep it up to date.
- Your main function must be in a class called `pcalc`. The skeleton is already set up this way.
- The first line of output of your program identifies the program. It may contain anything.
- Before reading the first command and on reading each subsequent end-of-line, your program must print the prompt '`>␣`' (greater than followed by a blank). No cute or obscene phrases, please, just '`>␣`'.
- Output things in exactly the format specified in §4 above, with no extra adornment.

- Put your error messages on separate lines, starting with the word '`ERROR`'. The grading program will ignore the text of the message.
- Your program should exit without further output when it encounters a `QUIT` command or an end-of-file in the input.
- Your final version must not print any debugging information.

6 Advice

You will find this project challenging enough without helping to make it harder. Much of what might look complicated to you is actually pretty easy, once you take the time to look to see what has already been written for you—specifically what is in the standard Java library of classes. So, before doing any hard work on anything that looks tricky, browse your texts and the documentation.

For reading input from the user, I expect that you'll find `java.io.StreamTokenizer` or `java.util.StringTokenizer` to be useful. Alternatively, you can use the C and C++-style input provided by `ucb.io.FormatReader` (you will find the `unread` method there rather useful). The standard Java string type, `java.lang.String`, also contains useful member functions for doing comparisons (including comparisons that ignore case). Read the on-line documentation for all of these.

It's important to have *something* working as soon as possible. You'll prevent really serious trouble by doing so. I suggest the following order to getting things working:

1. Write the user documentation.
2. Write some initial test cases.
3. Get the printing of prompts, handling of comments, the QUIT and EXIT commands, and the end of input to work.
4. Handle expressions consisting of a single integer (that is, get your program to read and print such expressions).
5. Handle expressions consisting of a single term without parentheses.
6. Handle expressions consisting of sums of terms, without parentheses.
7. Handle parentheses.
8. Handle definitions.
9. Handle substitution.

Reading the input is always a bit fussy. Observe that our description of the expression input format suggests a set of mutually recursive functions that read an expression, read a factor, read a term, read a substitution, etc. Write each of them in isolation as if the others were already implemented. (The result is called a *recursive descent parser*.)

7 Sample Session

User input is underlined.

```
% java pcalc
Pcalc, version 1.0
> # Here are some examples
> x;
= x
> 42;
= 42
> (y+x)(y
> -x);    # Free format!!
= -x^2+y^2
> DEF A(a,b) = (b+a)(aa + b^2-ab);
> A(2,3);
= 35
> A(a,b);
= a^3+b^3
> A(b,a);
= a^3+b^3
> A(x,x);
= 2x^3
> 4A(x,y)A(a,q)+b;
= 4a^3x^3+4a^3y^3+b+4q^3x^3+4q^3y^3
> QUIT
```