Testing report for ExpressionTree assignment

Mary Ellen Foster

21 August 2000

This report describes the testing of my solution to the ExpressionTree assignment. I tested each part of the program separately: first, the driver program; then, reading and printing arithmetic expressions; next, evaluating expressions; and finally, simplifying expressions. The test input is divided into three files: `test1.in`, `test2.in`, and `test3.in`. Specific comments about the result of each test case are included on the printout of the test inputs and outputs.

1 Testing the driver

The first set of test cases exercise the capabilities of the driver program itself. First, the driver is tested on blank lines and invalid commands. Next, each valid command is tested before and after reading a valid expression. The final case tests what happens when the input ends without a "q" command being entered. In all cases, the program performed as expected or printed an appropriate error message.

File	**Case(s)**	**Relevant features**
`test1.in`	1	Blank line in input
	2	Invalid command
	3	Run each command before entering a valid expression
	4,21,33,53	Valid use of each command
	4	• "r" and "p" commands
	21	• "e" command
	33	• "s" command
	53	• "q" command
`test2.in`	1	End of input without "q" command

2 Reading and printing expressions

The following test cases exercise the code for reading and printing arithmetic expressions. First, simple valid single-term expressions are entered. Then, more complex valid multi-term expressions are built up from simpler expressions.

Next, various syntactically valid expressions with unusual components (such as numbers with decimal points) are tested. After that, expressions that are syntactically invalid in a variety of ways are entered—some of these result in an error, while others work but in unexpected ways.

The final case ensures that the program does not crash even if the input ends in the middle of reading in an arithmetic expression.

These cases also demonstrate that a successful "r" command makes the newly-read expression the current expression (case 5), while an unsuccessful "r" does not change the current expression (case 15).

File	Case(s)	Relevant features
`test1.in`	4–7	Single-term expressions
	4	• positive integer
	5	• negative integer; "r" changes the current expression
	6	• single-character variable
	7	• multi-character variable
	8–11	Increasingly complex multi-term expressions
	8	• two integer operands
	9	• two variable operands
	10	• one variable, one integer
	11	• combination of 8–10
	12–16	Syntactically valid expressions with unusual components
	12	• Number with decimal point
	13	• Variable name made up of weird characters
	14	• Invalid operator
	15–20	Syntactically invalid expressions
	15	• No expression entered; failed "r" does not change the current expression
	16	• No parentheses
	17	• Unbalanced parentheses
	18	• Not fully parenthesized
	19	• Extra balanced parentheses
	20	• Parentheses with nothing in them
`test3.in`	1	End of input occurs in the middle of reading an expression

3 Evaluating expressions

The next set of test cases exercise the `eval()` function. First, single positive and negative integers are entered. Then a number of valid simple integer expressions are entered, including both division that has an exact answer and division that has a fractional result. Notice that case 23 also demonstrates that evaluating an expression does not change the value of the current expression.

Next, a more complex expression that includes all of the valid operators is tested. Finally, a number of unusual cases are tested. In all cases, the program either performs as expected or produces an appropriate error message.

File	Case(s)	Relevant features
`test1.in`	21–22	Single integers
	21	• positive
	22	• negative
	23–28	Integer expressions
	23	• addition
	24	• subtraction
	25	• multiplication
	26	• division (exact)
	27	• division (fractional)

File	Case(s)	Relevant features
	28	• combination of all four operations in one
	29–32	Unusual cases
	29	• division by zero
	30	• 0/0 (where both 0s are computed)
	31	• Expression containing a variable
	32	• Invalid operator

Simplifying expressions

The final set of test cases exercise the `simplify()` function. Once again, the testing starts with simple expressions and progresses to more complex ones.

The first four cases demonstrate simple expressions to which none of the simplification rules apply, so simplification does not alter the expressions.

The next set of cases shows each simplification rule being applied individually, while the following set contains expressions with subexpressions that evaluate to 0 or 1, which then simplifies the whole expression. Case 37 also demonstrates that simplifying an expression does not change the current expression.

The final set of cases consists of expressions containing operators other than $+$ and $*$—in all of these cases, only the subexpressions containing $+$ and $*$ are simplified. Even an invalid operator does not pose a problem (case 52).

File	Case(s)	Relevant features
`test1.in`	33–36	Basic cases (expression does not change)
	33	• single integer
	34	• single variable
	35	• integer plus variable
	36	• integer times variable
	37–44	Simple expressions where simplification rules apply
	37	• sum of integers
	38	• product of integers
	39	• $(0 + var)$
	40	• $(var + 0)$
	41	• $(1 * var)$
	42	• $(var * 1)$
	43	• $(0 * var)$
	44	• $(var * 0)$
	45–47	Expressions with subexpressions that cause rules to apply
	45	• variable plus subexpression that evaluates to 0
	46	• variable times subexpression that evaluates to 0
	47	• variable times subexpression that evaluates to 1
	48–52	Expressions with operators other than $+$ and $*$
	48	• Numeric expression with $-$
	49	• Expression with $-$ with a subexpression that gets simplified

File	**Case(s)**	**Relevant features**
	50	• Add another subexpression that also gets simplified
	51	• A very complex expression multiplied by 0
	52	• Invalid operator