Prometheus

v1.2

From Tesseract Software Solutions
Created by Jason Cohen

Prometheus is a programming language specially designed for logic, mathematics, and artificial intelligence. It contains elements from C, Pascal, LISP, and Prolog, but has many novel features. It is high-level and very weakly typed. Some highlights of Prometheus are:

- flexible syntax — functions and operators can be entered in LISP or C/Pascal style, functions with one argument can omit parenthesis, free use of whitespace, implicit multiplication.
- extensive mathematical simplification, calculation, expansion, collection, and transformation, both numeric and symbolic, complete with arbitrary size integers, reals, imaginaries, fractions, constants, variables, vectors, matrices, and functions. All 24 trig functions both circular and hyperbolic, special matrix, vector, and number theory functions. Symbolic differentiation, integration, and differential equations. Set theory, logics, statistics, probability, combinations.
- easy list processing (general, stacks, and queues), string processing, and I/O with console, files, and scripts.
- smart operators — operators will work without error on any of the data types, operators can be used in an intuitive manner: the expression "a<b<c" means a is less then b which is less then c, "5!" means 5 factorial.
- variables can take on a variety of values, from the usual assignment, to holding the names of other variables along with expressions, to dynamic values which change as other variables change. Variables don't need declaration, and there is automatic garbage collection.
- output in LISP, Prometheus, or mathematical format. Also output C++ source, fully functional with given source and libraries.
- portable across many platforms.
- 170 functions, over 350 simplification rules

In general, Prometheus should be used for problems which involve:

list processing	string processing	symbolic math	logic
vectors	matrices	AI	number theory

Table Of Contents

Basics

When running Prometheus, you have a window that lets you type input and receive output. This window is called the command console. The line in which text is entered is called the command line. If you enter something there and press return, the text is input to Prometheus. On Macintosh computers, you may use the mouse to move the insertion point, select text, and even use the clipboard.

The style and syntax of Prometheus are very easy to understand and use. For example, “1+2” means one plus two, and “csch -.5” is the hyperbolic cosecant of negative one half. For a few examples of the computational power of Prometheus, which also demonstrates its unique display capability, type “run "ex"” at the command line. The best way to get started is just to dive into the manual and the program simultaneously, running the examples, and exploring the program. It’s never a bad idea to experiment on the command line and explore possibilities. Remember, if you want to know how Prometheus will handle a particular expression, just type it into the command line and see what happens. It’s easier, faster, and more accurate then any manual. You can’t inadvertently do any damage, so feel free to explore.

Operators

As the name suggests, operators operate on inputs and return outputs. Their usage is intuitive and where applicable reflects the operators found in mathematics. Some operators are used between the inputs (also known as arguments and parameters) like “1+2” and are called in-fix. Some come before the argument like “neg(2)” and are called pre-fix. Some come after, such as “(5)!”, and they are called post-fix. Pre- and post-fix operators enclose their arguments in parenthesis as the examples show. However, if the operator takes only one argument, then the parentheses can be omitted, e.g. “neg 2” and “5!”. This emulates mathematical style. If there are no arguments, then parentheses can be empty “()” or omitted. For more then one parameter, parentheses must exist, and the arguments are separated by spaces. The only exception is if the command is the first on the command line, in which case those parentheses may be omitted. The identifying feature of the operator (i.e. “+” and “!” and “neg”) is called the operator name, or symbol.

You may also enter operators in LISP style. This style has parentheses that are never optional,. Also, the operator name must be the first element with the arguments following. For example, “(neg 2)” is equivalent to “neg(2)” and “neg 2”, and “(+ 1 2 3)” is the same as “1+2+3”, and “(! 5)” is equal to “5!”. “(+ 1)” results in “1” and “(+)” results in “[ ]”. This special symbol crops up in many different places and is called the null. It means “nothing” or “not applicable” or “undefined.” It does *not* mean null set. However, a null set does exist in Prometheus, and it is covered in the Sets section. In the expression “(+)”, Prometheus doesn’t know what value to return, so it returns null. You may mix LISP and regular style syntax; therefore, “(+ a b * c d)” is equivalent to “a + b * c + d”.

Prometheus has 149 operators that are applicable to mathematics, logic, list and string processing, and file and console I/O.

Numbers

A number in Prometheus is the same as a number in mathematics. There are integers, reals, fractions, and complex numbers. This section will describe each in that order.

To enter an integer, simply type the number. For example, "7" and "-314159" are integers. In Prometheus, integers may grow to any size, limited only by available memory. However when calculations get to be more then several thousand digits long, performance is noticeably lower. The basic arithmetic operators for numbers include addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). Try typing expressions involving these operators and numbers. You may use parenthesis in the regular mathematical way. Prometheus maintains the standard mathematical order of precedence so that "2+2*2" is evaluated as 6, not 8. A table containing the order of precedence for every operator can be found in the Order of Precedence section. Negative signs can be used freely and Prometheus will understand what is meant. For example, "--1" is "1", "---1" is "-1", etc., "1-1" is 0, "1--1" is 2, and "-1--1" is 0. Also, you can use the neg operator to produce a negative and the inv operator to produce an arithmetic inverse. For example, "neg 2" is "-2" and "inv 5" is "0.2".

Reals are entered exactly they way they look on paper. "3.2", "-.2", "0.5", and "4." are all reals. Notice that an integer followed by a decimal point is a real. Unlike integers, reals cannot grow to any size. Reals have at least 10 digits of accuracy—usually about 15, the exact number determined by your computer; more on this momentarily—and they can range from about 10^{308} to -10^{308}. Arithmetic with reals works the same way as integers. Executing operations on reals is faster then with integers, but reals are limited in accuracy and size. If you mix reals and integers, the result will be real. Therefore, "3.2+2" is the real "5.2" and "3.+2" is the real "5". Also, "3.5+0.5" is the real "4". If you type "1/3" you see the decimal expansion of one third. The number of digits displayed for irrationals and repeating rationals depends on the precision of your computer. When Prometheus starts up, it displays the precision as the smallest number x such that $1.0+x$ is not equal to 1.0. The quantity x is known as the eps. At startup, Prometheus displays the eps, the square root of the eps, and the maximum number of digits that will be displayed. See the Calc section of the Programming chapter for information about how to access the eps in your calculations. Also, if there are more digits than can be displayed, the last digit is rounded.

If you type "e" and another integer immediately after an integer or a real, then the "e" means "times ten to the". Therefore, "1234e52" means 1234 followed by 54 zeros, "1234e-2" means "12.34", and "1.42e7" means "1.42 times ten to the 7" or "14200000". Reals will be displayed in this scientific notation if the number is too big or small to fully display with the maximum number of digits.

Fractions are built with the back-slash (\) operator. They work just like mathematical fractions. Therefore, "3/2" is "1.5", but "3\2" is the fraction three halves. Try using the operators on the fractions, e.g. "1+3\2" yields "5\2". Fractions can contain any expression in the numerator and denominator. To illustrate, try "(1 \ 2) \ 3" and "1 \ (2 \ 3)". You can access the numerator and the denominator of fractions with the record operator whose symbol is the period. The numerator is accessed like "(2\3).a" and the denominator like "(2\3).b". The outputs of these two examples are "2" and "3" respectively. The terms "a" and "b" are called indices.

Complex numbers are of the form $a + b * i$ where i is the square root of -1. They are written in Prometheus as "(a , b)". Any expression can be substituted for a and b. Therefore, "(0,1)" is equal to i, "(1,0)" is equal to 1, and "(2,-3)" is equal to $2\text{-}3i$. Try typing "i" at the command line. This demonstrates that Prometheus uses the letter i as a short cut to typing "(0,1)". Also, "(2,3)" is equivalent to "2+3*i", and since Prometheus understands implicit multiplication, "2+3i". Again, all operators work on complex numbers exactly as mathematics dictates. You can access the real and imaginary parts of complex numbers with the record operator in the same way that numerator and denominator are accessed in fractions. In the case of complex numbers, the "a" means the real, and the "b" means the imaginary part.

As you work with longer calculations, you will find it useful to get the last expression that was evaluated. This is possible with the "last" operator which takes no arguments. Variables and scripts, both of which are covered in later sections, let you make extensive calculations more swiftly, efficiently, and easily then if you entered expressions one at a time at the command line.

Infinity

Infinity is represented as the letters "INF". Negative infinity is simply "-INF". All of the functions and operators work with infinities without error. For example, "INF+2+3" yields "INF", "INF-INF" remains unchanged because it is not defined, "inv INF" equals "0" and "1/0" is "INF".

Vectors, Matrices, and Lists

In Prometheus, vectors, matrices, and lists are all manifestations of the same entity. To create one, use brackets to enclose, and spaces to separate, the elements. For example, for the vector [2,3] you type "[2 3]". You may also use parenthesis like "(2 3)". In general, however, you will want to use the brackets for clarity. Also, "((2 3))" is the same as "[2 3]", although "[[2 3]]" is not because parenthesis are primarily grouping symbols, and only secondarily substitutes for brackets. Matrices are vectors that contain vectors as elements. For example, type "[[1 2 3] [4 5 6]]" at the command line and observe the results. If a vector contains vectors of unequal lengths or a mixture of vectors and non-vectors, then the entity is called a vector. Type "[[[[a b] [c d]] pi \ 4352] [[f (g , h)] 7]]" for an example of complicated vector and matrix usage. A list is just another name for a vector. The difference is conceptual; a vector is mathematical whereas a list is general. However syntax is identical, and Prometheus does not differentiate between the two.

As always, all operators work on vectors and matrices. Try "1+[1 2 3 4]" and "[1 2]-[3 4]" and "[1 2 3]+[4 5]" and "2[1 2 3]" and "inv [1 2 3]" and "inv [[1 2] [3 4]]" and "[[1 2] [3 4]] / [[1 2] [3 4]]" for some examples.

You can access an element in a vector with the record operator. Instead of letters as indices, use integers to specify the number of the element you want. The elements are numbered starting with 1. For example, "[[1 2] [3 4]].2" is "[3 4]". You can pile up record operators to access specific elements, so "[[1 2] [3 4]].2.1" is "3". This also works with fractions and complex numbers, so "[[1 (2,3)] [4 5]].1.2.a" is "2" and "([1 2],[3 4]).b.2" is "4". Alternatively, all the arguments you were piling up can be put into a vector, so "[[1 2] [3 4]].[2 1]" is equivalent to "[[1 2] [3 4]].2.1". Also, you can specify a range of elements using the range ".." operator. For example, "[2 4 6 8 10].(2..4)" would be "[4 6 8]". Note the parenthesis around the range operator. That is needed for order of precedence, which is listed in the Order of Precedence section. If you specify elements that are out of range, then Prometheus accepts and pretends you were in bounds. For example, "[2 4 6].(-2..2)" is the same as "[2 4 6].(1..2)", and "[2 4 6].(2..500)" is the same as "[2 4 6].(2..3)". You can get the number of elements in a vector by supplying it to the "size" operator.

Rows of a matrix can be accessed using the record operator, but columns can not. For column extraction, use the col operator. The two arguments it takes are the matrix and the column number. You may give a range as the column number. If the matrix has unequal numbers of elements in each row, gaps may appear in the column. In that case, null will be substituted for the gap.

Constants

A constant in Prometheus is equivalent to a constant in mathematics; it is a place holder for an unchanging value. A constant is a one word identifier that can include lower and upper case letters, the underscore, and the single quote (e.g. a b' hi_there SeeThisConstant). Case is significant, so HowAreYou and howAreYou are different. Constants are used for identifiers, labels, and for working symbolically with mathematical expressions. It is possible to give constants values, and that is covered in the next section.

Some words are not constants. For example, "TRUE" is the truth value True, not a constant called "TRUE". FALSE, i, and INF are other examples of non-constants. However, function names are constants.

Typing a constant at the command line shows that constants don't change in evaluation. Type "a + b" and note the result. Then try "b + a". This demonstrates that Prometheus understands the commutative property of addition. Try "1+2+a" and "1+a+2". To see some other examples of Prometheus simplification, try "--a", "0+a", "a + b - a", and "a + a". More complex examples include "a ^ n * a ^ m" and "a+[[b c] [d e]]". Prometheus simplifies in these and other ways with all data types, but often constants are used to demonstrate and explore the simplification. You can use constants to derive formulas. For example, to see the formula for the inverse of a 4x4 matrix, type in "inv [[a b c d] [e f g h] [j k l p] [q w z x]]". Of course, you can use any constant names you wish. You can use constants anywhere in expressions, even in fractions and complex numbers. Try "cos(a , b)" and "(a , b)^c" and "(a * b * c)\(c * d)".

Variables

A variable is a constant which has a value. It is useful for holding values in an organized fashion for later use. A variable has two parts: the name and the value. The name is a way of specifying which variable you are referring to, while the value is the data the variable contains. The values can be any expression at all. If you use a variable name in an expression, the value is substituted before the expression is evaluated. Initially, the are no variables.

Now enter "a = 4" into the command line. This makes the constant *a* turn into a variable containg the value 4. This command introduces the "=" or assignment operator. This is the same construct as the C/C++ variable assignment. The assignment operator takes the expression on the right and assigns it to the variable on the left. Notice that the value for the variable is returned. This allows such statements as "a = b = hi" meaning both *a* and *b* are assigned the constant "hi". If you try typing "a" at the command line after "a = 4", the value 4 is returned. "a+1" yields 5. Anywhere "a" appears, 4 is substituted. You can make *a* have a different value by reassigning it using the assignment operator. For example, to make *a* be the vector [b c] type "a = [b c]".

Give values to several variables. Then type "vars" at the command line. A list of all the variable names and values is returned. It may seem strange that the variable names were returned without their values substituted. This demonstrates another facet of Prometheus: sometimes variable values are *not* substituted. For example, the assignment operator does not substitute values for the left side of the operator. Doing so would obscure the variable you are referring to! Variable names on the right side of the assignment operator are substituted. In general, substitutions will occur unless you explicitly tell Prometheus to not make them.

It is okay to use a variable on both sides of the assignment operator. That is, "a = a+1" and even "a = a" are acceptable. This works because the right side is evaluated first, and then the result is assigned.

Assignment is static. This means that once a value is assigned to a variable, it won't change until you explicitly change it with the assignment operator. Sometimes, however, dynamic values are more useful. For example, let's say you wanted to represent the function $y = x^2-9$. If you type "x = 2" and then "y = x^2-9", then y will have the value -5. If we then type "x = 4" and evaluate y, we find that y is still -5. But we want to link x to y so that y changes to reflect x—that is, you don't want to retype "y = x^2-9" every time x changes. Type "y := x^2-9". Now y evaluates to 7. Type "x = 3" and y evaluates to 0. The new operator we used is called the definition operator, because rather then assigning a value, it defines a value. Try the vars command to see what this kind of variable assignment looks like. With define, the definition is saved without substituting variable values. The values are substituted when you use the variable, not when it is defined.

Sometimes you need a cross between assign and define. To illustrate, let's continue with the above example. Let's say we want y defined as "x^2-c" where y changes as x changes as before. However, we want to substitute the value for c right now, not later. We could have this requirement for many reasons. One is that c might change later but we don't want that to affect y. Another is that c is only relevant right now, and the future value of c is unpredictable. To implement y as "x^2-c", we need the value function whose name is the number sign (#). In our example, the command would be "y := x^2-#c". Notice we are still using the define operator. The value function tells Prometheus that we don't mean the name "c", but the value itself. You may put a whole expression in the argument to the value operator and the entire argument will be evaluated. For example, "y := x^2-#(c + d)" takes the value of c, adds it to the value of d, and then proceeds normally. Using the value operator in an expression other then the right side of a define operator has no effect, but is not an error. In general, if at all possible, the value operator should be used in a define statement because it will allow the defined variable to be evaluated faster, and more simplification can occur. Also, it is good programming practice to keep variables "free" from define statements so that you don't set a variable and inadvertently change another.

With the assign, define, and value operators, you can create statements which evaluate in complicated, flexible, and useful ways.

Truth

There are two truth types in Prometheus: TRUE and FALSE. They are denoted with capital letters. Truth is used in logics, relations, and control statements such as if-then-else.

For logics, truth is used as is commonly accepted. The "and", "or", and "not" operators work just like logic dictates. For example, "a and TRUE" is "a" and "a or b or FALSE" is "a or b" and "not TRUE" is "FALSE". Not can also be written "~". The logical constructs are simplified so that the result is in conjunctive normal form. In this form, we have a group of or's separated by and's, with the not's distributed to the elements. Therefore, "(a or ~b) and (b or c) and d" is in conjunctive normal form. "(a and b) or (c and d)" is simplified to "a or c and a or d and b or c and b or d" with order of precedence making parentheses unnecessary. Some examples showing logical simplification are as follows: "(a and b) or (a and ~b)" yields "a", and "p and q or p" yields "p".

Truth is also used in relations. A relation is a statement involving equality, greater then, or less then. There are six relational operators: equal (==), not equal (<>), less (<), less or equal (<=), greater (>) and greater or equal (>=). They return TRUE or FALSE according to the arguments given. If you give it incompatible arguments like “1<a” then nothing happens, unless you are dealing with equality or inequality. You may combine relational operators as well as supply many parameters. For example, “a == b == c” means a is equal to b which is equal to c. Equivalent statements would be “(a == b) and (b == c)” and “(== a b c)”. You can even combine different types, so “a < b >= c = d” means a is less then b which is greater then or equal to c which is equal to d.

Strings

A string is a sequence of characters. It is typed by enclosing the characters in double quotes. Any characters at all except the double quote may appear in a string. It is used to hold arbitrary text and data.

The addition, subtraction, and multiplication operators have special results when used with strings. Addition concatenates two strings, or a string and another expression. For example, “"this " + "is a test."” results in “"this is a test."”, and “"this" + 42.3” results in “"this42.3"”. Because order is significant, addition is non-commutative with strings. If you enter “"hi" + 1 + 2 + 3 + 4” you will get “"hi1234"”, but entering “1 + 2 + 3 + 4 + "hi"” yields “"10hi"“ because of the left to right evaluation. The empty string “""” is a valid expression, and can be used at the beginning of a set of addition operators to force everything to be a string.

Subtraction causes every occurrence of a substring to be deleted from a string. For example, “"this is a test."-" "” yields “"thisisatest."”. Multiplication by a positive integer makes copies of the string. Multiplication by 1 and 0 is acceptable. Therefore, “10 * "hi"” yields “"hihihihihihihihihihi"”, and “0"hi"” yields “""”. Notice the implicit multiplication in the latter example.

Two other operators create special strings that are not possible to type except with escape sequences discussed later in this section. The nl operator returns a string with the return carriage character, and tb returns a string with the tab character. You may access characters in a string with the record command exactly like accessing elements in a vector. You can also use the size operator on strings to get the number of characters in the string.

Prometheus supports C-style escape sequences in strings. An escape sequence starts with the backslash “\” character and is followed by another character. Together they create a character not usually possible to type. As an example, "\n" is the carriage return character, and is identical to nl. Therefore, both "This" + nl + "test" and "This\ntest" produce the same results. The following table details each escape sequence:

Sequence	Result
\\	backslash ("\")
\r	carriage return
\n	new line (equivalent to \r in the command line, but different in files.)
\t	tab
\v	vertical tab (practically obsolete nowadays)
\a	alert (rings the bell)
\Any_Number	the character specified in octal code
\xAny_Number	the character specified in hexadecimal code

Sets

A set is a group of expressions that follows the rules of sets according to standard set theory. Sets may contain other sets. There cannot be identical elements in a set. Sets are denoted with the curly braces “{}”. You enter the elements exactly as you would elements in a vector. For example, “{b c d a}” is a set. “{}” is the empty set. Typing the former example at the command line yields “{ a b c d }” indicating that the elements are automatically sorted.

If you add two sets, the result is the union of the two sets. Multiplying two sets gives the intersection. Therefore “{a b c}+{b c d}” yields “{a b c d}”, and “{a b c}*{b c d}” yields “{b c}”.

Mathematics

Overview

This chapter covers all of the math–related operators. In general, every operator accepts any expression and computes and simplifies to the greatest imaginable extent. The sections in this chapter do not detail the rules of simplification because they are so extensive and complex. See the Simplification Rules section of the Operator chapter for an austere but complete description of the simplification rules. This chapter defines the operators and their effects on various data types, and makes no mention of the automatic simplifications.

It's never a bad idea to experiment on the command line and explore possibilities. Remember, if you want to know how Prometheus will handle a particular expression, just type it into the command line, possibly substituting constants where you want to track the behavior more closely. It's easier, faster, and more accurate then any manual.

Arithmetic

The arithmetic operators addition, subtraction, multiplication, division, exponentiation, negation and inversion have already been covered in the Numbers section of the Elements chapter. This section describes some other basic arithmetic operators.

Some languages use "**" where Prometheus uses "^". To aid in translation and increase ease of use, Prometheus will also recognize the double asterisk as the power operator. However the caret is always used for display. The "abs" operator takes the absolute value of its input. For vectors the norm of the vector is returned, and for square matrices the determinant is computed. The "sgn" operator is the signum function which returns 1 if the input is positive, -1 if it is negative, and 0 if it is 0. The sqr and sqrt functions are defined for convenience: "sqr x" is the same as "x^2", and "sqrt x" is the same as "x^.5". You can use the "%" command to take the remainder of division. Therefore "5 % 4" is 1 because when you divide 5 by 4 the remainder is 1. You can also think of this as modulus; therefore the above statements are equivalent to 5 mod 4.

Three additional operators take integers or reals as their argument: floor, ceil, and round. Given a number, they return the greatest integer less then the given number, the least integer greater then the given number, and the given number rounded off respectively. Floor is also known as the "greatest integer" function. In mathematics, "floor x" is often written $\lfloor x \rfloor$ and "ceil x" is written $\lceil x \rceil$. Given infinities, these operators return infinities. If you supply a second argument, then the result will not be shifted according to the integers, but according to the given tolerance. For example, "round 3.4" yields 3, but "round 3.4 0.5" yield 3.5. A tolerance of 1 is identical to not supplying the tolerance. The tolerance is useful for getting rid of excess digits. For example, if you make a computation which has four significant figures, but after division you get "1.243763452432", then you could use "round(1.243763452432 1e-4)" to get the correct answer.

The "sum" operator represents sigma notation for summations. Therefore "sum(k a b f)" means $\sum_{k=\alpha}^{\beta} f$. The second and third parameters do not have to satisfy $a \le b$. Many rules simplify the sum operator by, for example, taking out factors in the last parameter not related to the variable, and using formulas for basic summations. The "prod" operator is identical, except "prod(k a b f)" means $\prod_{k=\alpha}^{\beta} f$.

Algebra

One of the most basic algebra operations is substitution. To substitute all instances of an expression *a* with *b* in an expression *c*, use “subst(c a b)”. Non-mathematically, this is a search and replace. It can be useful for simplifiying expressions and substituting variable values.

The isol command isolates an expression in an equation. The equation is formed with any of the six relational operators ==, <>, >, <, <=, or >=. The expression may be anything. For example, to isolate the variable x in $\frac{1}{\ln x} > 7$ you would type “isol(1/ln x > 7 x)” and the result would be $x < \varepsilon^{0.142857142857143}$. If the expression you are isolating appears more then once in the equation, the first one found in DFLR order (see the Searching Elements section of the List and String Processing for details) is isolated. All possible solutions are found, so “isol(x^2==9 x)” yields “x==3 or x==-3”, “isol(x^3==8 x)” yields “(2==x) or ((-1,1.732050807568877)==x) or ((-1,-1.732050807568877)==x)”, “isol(abs(x-2)<=14 x)” yields “x<=16 and x>=-12”, and “isol(cos x == y x)” yields “x==2*pi*N+acos y or x==2*pi*N+-acos y”.

You can use ± in your expressions with the “+-” operator. For example, “3 +- 1” results in “2 or 4”.

Transcendental Functions

Transcendental functions always return real or complex results, and mimic the mathematical functions directly. There are 24 trig functions and 3 logarithm functions.

The operators “cos”, “sin”, “tan”, “csc”, “sec”, and “cot” work exactly the same as their mathematical counterparts, i.e., cosine, sine, tangent, cosecant, secant, and cotangent, respectively. Putting an “a” before each yields the arc, or inverse, function. Appending an “h” to each yields the hyperbolic function. Both prepending an “a” and appending an “h” is the arc, or inverse hyperbolic function. They work with complex arguments. The “cis” function is also implemented, so “cis x” is the same as “cos x + i*sin x”.

The “log” operator takes a base in the first argument and a value for the second, so “log(a b)” means $\log_a b$. The “ln” operator means “natural log” and the “lg” operator means $\log_2$. These work on complex numbers and accept non-positive reals including 0 (which returns “-INF”).

Logic

Many operators work with the logical operators “and” and “or” to produce useful expressions. This can be used to combine mathematics with logics for accurate mathematical results. For example, “a+(b or c)” simplifies to “(a + b) or (a + c)”.

Besides the three logical operators already described (and, or, and not) there are two other logical operators, “forall” and “exist”. They represent the logical quantifiers ∀ and ∃ respectively. To represent “∀x: y”, use “forall(x y)”. Exist works in the same manner. Conjunctive normal form dictates that not’s are distributed into quantifiers, so that “~(forall x y)” results in “(exists x ~y)” and “not (exists x y)” results in “forall(x ~y)”.

You can also use the imp operator, which means “implies.” The “xand” operator means “this and not that” and is logically equivalent to “$p \wedge \sim q$”. The “xor” operator means “this is not the same as that” and is logically equivalent to “$(p \vee q) \wedge \sim (p \wedge q)$”.

The “unify” operator is used in logical deduction. The two arguments are compared, and if any non-variable quantities don’t match, the operator returns FALSE. Otherwise, the operator returns a list of variable substitutions that would be necessary to make the expressions equal. This list is *not* evaluated, but you can evaluate it yourself with the “#” operator that always takes the value of all variables. No variables within the unify arguments are evaluated.

Probability

Probabilities are reals between 0 and 1 inclusive. You can work with them like any number, but there are some special operators which let you compute with them.

Using the not operator, which can be written as “not” or “~”, you can get complementary probability. Therefore “~0.25” is “0.75” and “~0.5” is “0.5”. The “&” operator answers the question “what is the chance that both this and that will happen”. Therefore “0.5 & 0.5” is “0.25” and “0.2 & 0.4” is “0.08”. The “|” operator answers the question “what is the chance that either this or that will happen.” Therefore “0.5 | 0.5” is “0.75” and “0.2 | 0.4” is “0.52”. The imp operator answers the question, “what is the probability that if this is true then that is true.”

The factorial operator “!” comes after the argument as it does in math. “5!” is 120, and “0!” is 1. Currently, factorials work only on integers, both positive and negative, and on 0. With the Factorials Package loaded, large factorials like “1000!” will be computed very quickly. See the Package chapter for information concerning packages. Closely related are the nPr and nCr operators that compute permutations and combinations. For example, “5 nPr 3” is “20”, and “5 nCr 3” is “10”. The gamma function is defined as $\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt, x > 0$, or alternatively $\Gamma(x+1) = x!$, and is computed by the “gamma” operator. You will get a maximum of 5 decimal places of accuracy, that number decreasing as the argument to the gamma operator increases. See the Technical Reference section of the Operators chapter for more information about gamma.

The erf operator is the error function defined as “erf x = P(0.5,x)” where P is the incomplete gamma function. The cerf operator returns the complementary error function defined as “cerf x = 1-erf x”. Always use “cerf” in place of “1-erf” because it is much faster.

Statistics

There are several operators which let you run statistical tests and find statistical parameters from a list of data.

If you supply a list as the sole parameter to “sum”, it will add up the elements of the list. So “sum [a 1 b 2 a 3]” yields “6+b+2*a”. Similarly, “prod” will multiply the elements of a list. The “size” command returns the number of elements in a list. To average the elements of a list, supply it to the “avg” operator. Variance and standard deviation are computed with “var” and “std” respectivly.

Number Theory

Prometheus’s arbitrary length integers and complex simplification methods are especially suited to the exploration of number theory. Several functions from number theory are provided, all of which facilitate working with both integers and expressions.

With the “factor” operator, you can factor expressions into prime factors. The factors are returned in a list in ascending order. The number “1” is never included, but if the number supplied is prime, it will be returned. Factoring large numbers may take a long time, and sometimes a factor returned as prime may in fact not be. Refer to the Technical Reference section of the Operators chapter for more information. The “gcd” operator returns the greatest common divisor of the two arguments, and the “lcm” operator returns their least common multiple. These operators facilitate working with integers or expressions. Therefore, “gcd(a ^ b a ^ c)” results in “a ^ min(b c)” and “lcm(a ^ b a ^ c)” yields “a ^ max(b c)”. You may supply more then two arguments to both gcd and lcm.

Vectors and Matrices

All operators work on vectors and matrices in the usual mathematical way. Some, like abs and inv, are redefined to match their meaning with matrices. Also, there are some special functions which work only on vectors and matrices to produce certain quantities. You may want to try out some commands on matrices or vectors, but you don't want to type in a large set of data. You can use the genmat command which takes two arguments and returns a generic matrix of those dimensions. For example, "genmat(2 3)" results in "[[a b c] [d e f]]".

The inverse operator works for vectors by returning a scalar value that, when multiplied by the original vector, yields another vector in the same direction as the original, but has a norm of 1. The inverse operator applied to a square matrix yields the matrix inverse, However, if the determinant of the matrix is 0 and therefore it does not have an inverse, an inverse is returned anyway with either INF or -INF in each element. When multiplication is performed on vectors, the inner, or dot, product is computed.

The "adj" or adjunct operator returns the adjunct of a given square matrix. The "minor" operator takes a minor of a matrix. For example, "minor([[a b] [c d]] 1 2)" means the minor of "[[a b] [c d]]" about the first row, and second column, and in this case is "c". The "trans" operator returns the transpose of the given matrix. It works correctly on non-square matrices, and even vectors where the rows are of unequal length. In this case it pads empty cells with null as described about "col" in the Vectors, Matrices, and Lists section of the Elements chapter.

You can solve augmented matrices with the "saug" operator. It takes an augmented matrix created form the coefficients of a system linear equations and returns the solutions to the variables as defined below:

Assume you have a system of *n* linear equations in *n* variables. Let x_1,x_2, ... , x_n be the variables, and the equations be:

$$
\begin{aligned}
&a_1x_1+a_2x_2+...+a_nx_n=a_{n+1}\\
&b_1x_1+b_2x_2+...+b_nx_n=b_{n+1}\\
&...\\
&z_1x_1+z_2x_2+...+z_nx_n=z_{n+1}
\end{aligned}
$$

Then the *n* by *n*+1 augmented matrix which corresponds to this system is:

$$
\begin{bmatrix}
a_1 & a_2 & \cdots & a_n & a_{n+1}\\
\beta_1 & \beta_2 & \cdots & \beta_\nu & \beta_{\nu+1}\\
\cdots & \cdots & \cdots & \cdots & \cdots\\
\zeta_1 & \zeta_2 & \cdots & \zeta_\nu & \zeta_{\nu+1}
\end{bmatrix}
$$

The vector $\langle x_1 \quad x_2 \quad \cdots \quad x_n \rangle$ is returned. If the system is over determined, under determined, inconsistent, or dependent, then all variables will be set to INF or -INF. You may, of course, include non-numbers such as constants, variables, and vectors in the augmented matrix, but the solution will take longer to find. See the Technical Reference section of the Operators chapter for more information about saug.

Functions

The “fzero” operator finds a zero of a function. You specify five parameters: the function in question, the variable to vary, the lower and upper bound for the zero, and a tolerance. The first two must not be evaluated. Therefore, they must either contain the literal operator or be passed by a holding variable. The other three must be integral or real. The tolerance specifies the allowable error. Therefore, a tolerance of .01 and a result of 4 means the real root could be anywhere from 3.99 to 4.01. You can use “calc eps” to indicate the smallest possible tolerance. (The calc command is covered in the Calc section of the Programming chapter.) If there is more then one root in that range, there is no way to know which will be found. To find the minimum of a function within a given range, use the “fmin” operator with the same arguments as fzero. If there is more then one minimum in the given range, there is no way to know which will be found. To find the value of the inverse function, use the “finv” operator with the same arguments as fzero, but with one exception: after the variable, an additional argument must be made which is the value of the function. For example, if the function is “x^2” and the variable is x, and you want to know what value of x will make “x^2” equal to 6, and you know the answer lies in the range of zero to 10, then use “finv(x^2 x 6 0 10 0.01)” for an answer accurate to one-tenth. If more then one inverse exists in the given range, there is no way to know which will be found. The variable you supply to vary will contain the value returned by each of these three functions. See the Technical Reference section of the Operators chapter for more information about these three operators.

Calculus

Prometheus allows you to take a semi-numeric limit. It is called semi-numeric because, although it does not always return the exact limit, the vast majority of the time it does. Also, symbols can be used in the limit. To take a limit, supply the lim operator the variable, the value to approach, and the expression to take the limit with. For example, “lim x 3 (x^2 - 9) / (x - 3))” returns 6 and, “lim x c 1/x” returns “(1/(-1e-05+c)+1/(1e-05+c))*0.5”.

Prometheus is also capable of symbolic differentiation and integration. You use the “der” operator with the function and variable. For example, “der x ^ c x” means $\frac{\partial}{\partial x}x^{c}$ and results in “c * x ^(-1+c)”. A derivative is defined for every function. Multiple variables are also acceptable. However, although “der x x” is “1”, “der y x” is 0. To keep partial derivatives (i.e., leave “der y x” unchanged), use the “pder” operator. If you try to take the derivative of an expression with respect to a non-variable expression, the result is 0.

To find integrals, use the “intg” operator. For the indefinate integral $\int f dx$ use “intg(f x)”. The arbitrary constant is *not* added to the results of indefinate integration. For the definate integral $\int_a^b f dx$ use “intg(a b f x)”. Another operator, “range”, is used in conjuction with definate integrals to evaluate the result: “range(a b f x)” means $f\Big]_{\alpha}^{\beta}$. Of course, you may use the range operator independant from intg.

Other Functions

Some miscellaneous mathematical functions do not fit nicely into any of the above sections. They are generally useful for computing special numbers and expressing mathematical statements more accurately and flexibly.

The “fib” operator returns the Fibonacci number of the number given. For example, “fib 0” is “0”, fib “1” is “1”, and “fib 10” is “55”. If the Fibonacci Package is loaded, large Fibonaccis such as “fib 4000” can be computed very quickly. See the Package chapter for information concerning packages.

The “max” and “min” operators are maximum and minimum respectively. They can take any number of parameters.

The “rand” operator returns a random number between 0 and 1 inclusive. See the Technical Reference section of the Operators chapter for more information about the random number generator.

The “tofrac” operator converts reals to fractions. To use tofrac, provide the real in the first parameter, and a tolerance (another real) in the second. The function will return a fraction that is within the tolerance of the given real. Therefore “tofrac(3.1415926 1e-2)” yields “22\7” which is accurate to 1e-2, and “tofrac(3.1415926 1e-6)” yields “355\113” which is accurate to 1e-6. See the Technical Reference section of the Operators chapter for more information about tofrac.

The “re” and “im” operators take the real and imaginary parts of expressions, respectively. Although you can use indexing to get at the real and imaginary parts of a complex number, these operators can extract the same information from any expression. For example, “im(a*b)” simplifies to “((im a * re b)-(im b * re a))” and “re (a/b)” simplifies to “((im a * im b*1/((im b^2)+(re b^2)))+(1/((im b^2)+(re b^2))*re a * re b))”.

The “polyfit” command fits a set of data to a polynomial. For *n* data points given as a list to polyfit, the coefficients of the polynomial of order *n*-1 which best fits them are returned as a list. The first data point is assumed to be the result when the polynomial variable is 0. To set it at some other value, supply it as the second parameter to polyfit.

Programming

Overview

This chapter covers all of the programming–related operators. These include working with files, interacting with the user, executing loops, and executing if-then statements. Unlike mathematical operators which are useful in simplification, programming operators are useful in execution. Mathematical operators compute things while programming operators create the things to compute and do something with the results. Without programming operators, Prometheus would simply be a powerful calculator; with them Prometheus is a powerful programming language.

It's never a bad idea to experiment on the command line and explore possibilities. Remember, if you want to know how Prometheus will handle a particular expression, just type it into the command line, possibly substituting constants where you want to track the behavior more closely. It's easier, faster, and more accurate then any manual.

With programming operators, it is likely that the operator returns nothing important. For example, the pause operator waits for the user to press return. There's no information to return, so pause returns null. Unless specifically stated, all of the operators in this section return null.

Environment

The environment is a set of values which affect the way Prometheus works. For example, one determines whether or not Prometheus prints the result of an expression on the screen, and another determines what style of output Prometheus should use. To see the current state of the environment, use the "env" operator.

The first environmental control is called the echo. When echo is on, every time Prometheus evaluates an expression, the result is printed to the screen. The default setting is on. To alter this, use the "echo" operator that takes one argument and sets the echo accordingly. Use the constants "on" and "off" or the integers "1" and "0" as arguments.

The second control is the display style. Prometheus can display in three styles: LISP, regular, and mathematical. The default is mathematical. Use the constants "lisp", "reg", and "math" as the argument to the disp command to change the display type. In general, LISP style prints like a LISP program, regular prints as you would type it in the command line, and mathematical prints typographically accurate results within the limitations of text-based art. If the expression would be too big to fit in the command console, then the regular style is used. The best way to get a feel for the different styles is to experiment with different expressions in different styles.

The next control is not setable. It is just an indicator of something called the shell level. This will be discussed in the Scripts section. The fourth control determines wether degrees or radians are used as angle measure. You set it with the "deg" and "rad" operators which take no arguments. The default is radians.

The fifth control lets you force Prometheus to execute the "calc" operator on every expression. For a description of the "calc" operator, see the Calc section of this chapter. The default is off. You set it with the "acalc" (for auto-calc) command which takes the same argments as "echo".

Another environment–related operator is "mem". It returns the number of bytes left in the heap and stack in a list. Generally, you will only need this to determine how much memory is left in the heap, that is, how much memory you have to manipulate expressions. Generally at least 100K of memory should be kept free. If you let Prometheus run without entering expressions, memory will automatically free up.

Scripts

If all you need is a powerful calculator, the command line is sufficient. However, you need scripts to take advantage of the programming operators, and to save a series of commands for later revision, reference, and execution. A script is a text file containing one command per line. While running a script, Prometheus simply steps through and executes each line as if you had typed each one in the command line. In addition, you can do several other things in a script that are not possible on the command line.

You can create a script in any text-processing program. Just be sure that the file itself is purely text. To run a script, use the run operator that takes one argument: a string containing the name of the script file. Because you supply a string, you may name the file anything at all, as long as double quotes don't appear. An example of a script is the "ex" file that contains examples of Prometheus's display and computational abilities. Run the script and examine its contents. You will notice that it is okay to have blank lines. However, for reasons discussed in the Blocking section, it is *not* okay to put tabs at the beginning of lines in a script file because the tabs have a special meaning. You can call a script within another script, i.e., a script file can contain a run command.

If there is a file called "init.p" in the same directory as the Prometheus program, then it is executed when Prometheus first starts up. If it is missing, nothing happens. You can put initialization commands here, such as defining your own useful functions or physical constants.

The environmental indicator "shell level" pertains to scripts. It indicates the number of scripts currently active. Therefore, on the command line, no scripts are running, so the shell level is 0. If you ran a script which printed the env operator, the shell level would be 1. If you ran another script within the first and it printed the env operator, the shell level would be 2. However, if you had a script where the only line was "env", nothing would be printed because every time you run a script, the environment is reset to echo off, disp math. When that script finishes, the previous environment is restored. Therefore, a script containing:

```
echo on
env
```

would print the environment. For printing expressions regardless of the state of echo, refer to the Console I/O section.

Console I/O

This section covers the operators "read" and "print" that read expressions from the keyboard and print them on the screen, respectively. Also, the pause operator is discussed.

The read command produces a ">" symbol and waits for user input. Whatever the user types is converted to an expression and simplified just as if it were typed at the command line; the read operator returns this result. If the user types nothing, null is returned. It is okay to have an expression like "read + read". This will read two expressions from the keyboard and return their sum. It will *not* be simplified to "2 * read" and then executed. The reason for this is covered in the Properties section of the Operators chapter.

"print" displays an expression on the screen according to the current environmental display setting. You can give any number of arguments to print, and all will be printed in order. Be sure to use the nl function to get a new line. When you display strings with print, the quotes are not printed unless the display mode is LISP.

The pause operator takes no arguments. It prints "Paused..." on the screen and waits for the user to press return before returning. Any other characters typed appear on the screen but are ignored.

File I/O

You can save and restore expressions as files. This is not the same as scripts, which are covered in the Scripts section. This section covers files used as storage units for data that must be saved from session to session. The save operator takes the expression to be saved as the first argument, and a string of the file name as the second. The open operator takes a string of the file name as a parameter, and returns the saved expression. You can display any text file with the "disp" command which takes a file name as a string parameter.

You can also read data from database files, which are files containing data in certain formats. Each file contains one or more data sets which in turn contains one or more fields. A field is another name for datum. Prometheus can read database files in three formats: tab-delimited, exact count, and terminated. Tab-delimited files contain fields separated by tabs and data sets separated by carriage returns. Exact count files have fields separated by carriage returns, and there is no separation between data sets. However, there are an equal number of fields in each data set. Terminated files have fields separated by carriage returns, and data sets are separated by some sort of terminating symbol which is entered just like a field.

To read a tab-delimited file, use the opendb (for **open d**ata**b**ase) operator which takes the same arguments as open. It returns a matrix with each row being a data set, and each element in a row being a field. Each data set can have different numbers of fields. To read exact count files, use the opendb operator, but supply in the second parameter the number of fields in each data set. The data are returned in the same manner as tab-delimited files. To read terminated files, supply the terminating string as a second argument to opendb, and the data are returned in the same manner as tab-delimited files.

Normally Prometheus returns the data in strings. Sometimes, especially when reading numbers, you will want to read the data as Prometheus expressions. To do this, you use format specifiers that specify the type of data you will read in. They are used after the other arguments in opendb. The constants "r" and "p" are used to mean regular and Prometheus mode, respectively. You use it as follows: "opendb("Database" p r p r)" means read the first and third fields in a data set as expressions, and the second and fourth fields regularly. If there are more format specifiers then fields in a data set, no error occurs. If there are more fields in a data set then format specifiers, the default is "r". You can use format specifiers in any of the opendb forms. Another format specifier is "x", and using it causes Prometheus to ignore that field. You can use it to strip unwanted fields to keep memory usage down and speed up. You can also use "e" which causes that field to be read like "r", but then exploded as described in the Splitting and Combining Elements section of the List and String Processing chapter.

Sometimes the database file will not be in one of the standard formats of tab-delimited, exact count, and terminated. However, the data still have an underlying structure. With the openfm (for **open f**or**m**at) operator, you can specify a complex rule to read in a database file. The arguments are identical to open, except you have format specifiers. Some of these specifiers are followed by additional data to tell Prometheus how to handle a field. For example, the "skippast" specifier directs Prometheus to skip past some data. If it is followed by an integer, then that many characters are skipped. If it is followed by a string, everything up to and including that string is skipped. This extra information is called an option. The following table shows the effects of the format specifiers and their options.

Format S.	Option	Effect
get	integer	adds a string to the data set with length specified
	string	adds a string to the data set with characters up to, but not including the given string. The given string is then

		skipped.
skip	integer	skips a number characters according to the option
	string	skips characters until a character in the given string is encountered
skippast	string	skips characters up to and including the string
STRIPLF	*none*	instructs Prometheus to ignore line feeds (use for documents from DOS or Windows machines)

Sometimes, you'll want to skip over a header. Since openfm cycles through the format specifiers for each data set just like opendb, you need some way of specifying that some format specifiers should only be used once. To do that, use the "DATA" specifier to separate the header format specifiers from the data set specifiers. For example, "openfm("data" skippast "Start Here:\n" DATA get "\n" get "\n"" skips past all data up to and including "Start Here" and a carriage return. Prometheus then gets two fields, puts them in a data set, and repeats until the database is exhausted.

You can also write data in database format with the savedb operator. It takes the same parameters as save, but outputs in one of the three database formats. With no additional parameters, it writes expressions in tab-delimited format. With an integer third parameter, it writes in exact count format with the integer being the count; with a string third parameter, it writes in terminated format with the string as the terminator. Format specifiers do not work with savedb.

Del

This command is used to delete variables and files. You may want to delete a variable to conserve memory, or clean up the variable list. Deleting variables when you are done with them is not at all necessary, but it will speed things up. To delete a variable, simply supply the name of the variable as an argument to del. It is okay to delete variables you have never referred to. To delete any kind of file from a disk, supply the file name as a string to del. You can also use del to delete elements in a list or string. See the Deleting Elements section in the List and String Processing chapter.

User Stack

Some operations can be more easily carried out using a stack. This is a FIFO (First In First Out) list where you "push" expressions onto the stack and "pop" them off. The push command takes any number of arguments and pushes each in turn onto the user stack. The pop command pops one expression off the stack and returns it. The stack operator returns a copy of the stack for inspection.

Swap

The swap operator takes either zero or two arguments. With no arguments it switches the top two expressions on the user stack. With two variable names as arguments, it swaps the values of the two variables, including weather the variables are assigned, defined, or holding.

Calc

The calc operator calculates an expression numerically instead of symbolically. For example, the constant "pi" is calculated as 3.14159..., and the fraction "3\2" is calculated "1.5". The calc operator takes an expression as an argument and calculates it. Some constants get values when they are included in a calculated expression. The following table summarizes the values:

Constant	Calculated Value
e	2.71828182845904523536028747135266249775724… (Napiernian base)
eps	machine accuracy (see the Numbers section of the Elements chapter)
euler	0.57721566490153286060651209008240243104222… (Euler's constant)
phi	1.61803398874989484820458683436563811772030… (golden ratio)
phi'	-0.61803398874989484820458683436563811772030… (conjugate golden ratio)
pi	3.14159265358979323846264338327950288419720… (π)

Expd

The expd command expands algebraic expressions. It is mostly useful for expanding polynomials to integral powers. Therefore, "expd((a+b)^4)" yields "4 * a * b ^ 3 + 4 * b * a ^ 3 + 6 * a ^ 2 * b ^ 2 + a ^ 4 + b ^ 4", and "expd((a + b + c)^2)" yields "2 * a * (b + c) + a ^ 2 + 2 * b * c + b ^ 2 + c ^ 2".

Simp

You can provide additional rules to the simplification process. For example, you may be using the constants c and d in such a way that c + d will always equal 1. Then you could write a simplification rule that states "simplify c + d to 1". Or perhaps c times anything is c. Then you could write a rule stating "simplify c * anything to be c." This is done with the simp operator. It takes two arguments: the first is the expression to look for and the second is the result. In our examples, we could type "simp(c + d 1)" and "simp(c * ?a c)". Notice the use of the variable in the second case *without* the literal operator. The use of the literal operator would make this mean "simplify c times the variable name ?a to be c". Note that ?a in this case does not refer to ?a in the usual sense, since when this rule executes, you don't want the variable ?a specifically, but just any value. The simps function returns a list of the user simplifications. You can see that the ?a was changed to a variable called ?_0001. This is a generic name that will not conflict with any other variable name.

Currently, user simplification is not fully supported. It works exactly as stated here, but it does not work in some cases where intuitively it seems that it should. Right now it is good for very simple rules like "(simp hi there)"; other capabilities are planned. An example of user simplification falling short of expectations is with the first example being entered, "c + d + e" will be simplified as "1+e", but "b + c + d" will not. This will be smoothed out in later versions.

For-Loop

Sometimes you want to do something repetitively. For example, you may want to do something to each element of a list, or read input from the user an unspecified number of times. This requires the for-loop. This is a control structure, so called because control of the program is being restructured from just executing each line in order, to executing some lines many times. You can have one or more expressions be evaluated many times. This section only covers looping though exactly one expression. The Blocking section explains how to add more expressions to the loop.

The most basic form of the for-loop makes a statement repeat for a specified number of times. You use the for operator with two arguments: the number of times to repeat which must be an integer, and the expression to execute. For example, typing “for(10 print "hi")” results in “"hihihihihihihihihihi"”. If the number of times to repeat is less then one, the statement is not executed.

Usually, you want to loop with a counter. This means you have a variable which is changed each time the loop goes though one cycle. The counter could be 0 the first time through, then 1, then 2, etc. up to some point. A more complicated form of the for-loop makes this possible. You give it three arguments: the first is the variable name which will be the counter, the second is a range of values, and the third is the expression to be repeated. The range of values is entered with the range operator “..” which takes two arguments and sets up a range. Therefore, “0..10” means zero to ten, “2.3...4” means 2.3 to .4, and “a .. g” means a to g. Integers, reals, constants, and strings may be used in the range. Note that the range can go up as in “0..10” or down as in “10..0”. The counter will start at the first value, and each time the loop finishes executing, the value will be either increased or decreased to the other end of the range. The loop stops executing when changing the counter again would result in a value out of range. For numbers, the change is ±1, and for constants and strings the letters go up or down. For example, typing “for(a aa .. bj (print a nl))” will result in “aa” then “ab” then “ac”, etc., then “ay” then “az” then “ba” then “bb” etc. to “bj”. After the loop completes, the counter remains in the vars list, and its value is the last value used in the loop. Continuing our example, after the loop is complete, *a* holds the value “bj”. You may change the counter from within the loop.

Many times you will want to loop through the elements in a list. The following example shows how to use a variation on the for operator to accomplish this. First, here is the listing of code which prints each element of “[b c d e]” on the screen:

```
for(a 1..4 (print [b c d e].a nl))
```

And here it is using the variation:

```
for(a [b c d e] (print a nl))
```

As you can see, if you supply a list as the second argument to for, it will loop over that list and put each element in *a*. If you change the value of *a*, it will be reset after the cycle is complete. This form is especially useful when the list is inside a variable.

Bag

Bag is a variant on for. Instead of throwing away the results after each loop, bag returns them as a list. This is useful for generating vectors and lists with loops. For example, to return the first 10 squares in a bag, use “bag(k 1..10 k^2)”. To get the general form of the sixth-degree polynomial, use “[a b c d f g h] * bag(k 6..0 x^k)”.

If-Then-Else

The if-then-else statements are also known as conditional statements because they cause expressions to be evaluated “on condition”. Like the for-loop, if-then-else is a control structure, causing certain statements to be executed and others not. You set up statements which say “if so and so is TRUE, do this” or statements which go further and say “if so and so is TRUE, do this, otherwise do that.” There can be one or more expressions in the if-then-else statement, but like the For-Loop section, this section will defer the multi-expression if-then-else to the Blocking section, and concentrate on single-lined if-then-else constructs.

The if operator takes two arguments. The first must evaluate to a truth expression. If the truth expression is FALSE then the second argument is not executed, otherwise it is. If the argument is not a truth statement, the second argument *is* executed. This allows for some functions which return FALSE if they fail, and a useful expression if they don't. If directly after that the else operator is present then its argument is either executed or not in opposition to the if operator. So if the if operator executed its statement, the else won't, and vice versa. The else operator is optional, and is undefined if it doesn't follow an if statement.

Blocking

Most of the time, only one statement in a control structure is not enough. In that case, you want to group a bunch of expressions together, and call them one. This is exactly what a list does, and you could write the expressions in a list, but with many lines that could become unreadable and unmanageable, especially when some of the lines contain control structures themselves. Therefore, you want Prometheus to automatically collect a group of lines and make a list out of them. This is called blocking and it can only be done in a script. To do it, simply indent the lines you want blocked with tabs, and it will be done. For example,

```
for 10
	print "hi"
	print nl
```

prints "hi" and a return carriage ten times. You may indent multiple times:

```
if (?a = 2)
	for 7
		print "hi "
		print "there" nl
	print "done here" nl
else
	print "didn't go into loop!" nl
```

and Prometheus will group accordingly. Note how you don't supply any expression in the last arguments of the control structures. That construction is only applicable on the command line. You may include it in scripts, but you cannot use both blocking and the last argument constructs in the same control structure. You may, however, have a block of only one expression as shown in the above example.

The reason blocking can only be done in scripts is you cannot enter tabs in the command line, which is why the tb command is supplied. You may use any number of tabs to indent, but it is common practice to use only one at a time for readability. Also, it is okay to have blank lines in the middle of a block — Prometheus will not think you have abandoned the block.

Comments

Sometimes you want to put some text in scripts which is not executed. For example, you may want to write a reminder or an explanation of a line or some documentation. This text is called a comment. You can write comments by putting a double slash "//" at the beginning of a line in a script, and everything on that line will be ignored. Only tabs can appear before the double slashes. Blocking will not be affected by comments even if tabs on a commented line would normally affect the blocking.

Implicit Assignment

Many times you will change the value of a variable using the variable itself. For example, “a = a+1” or “a = append(a 1)”. Implicit assignment gives you a shortcut to writing these kinds of statements. With an extra question mark in front of a variable, the entire value of that expression is assigned to that variable. Therefore the previous two expressions could be written “?a+1” and “append(?a 1)”. If the variable appears more then once in the expression, then you only have to use the double question mark once. Using it more times is not an error, but the code will run slightly slower. You may use implicit assignment with more then one variable in the same expression, and all variables will take on the value of the expression. For example, “?a + ?b” adds the values in *a* and *b* and puts the result in both *a* and *b*.

List and String Processing

Overview

Prometheus has a variety of list and string processing commands. Other chapters have covered some of the operators, and this chapter will not repeat those discussions. If there seams to be a blaring gap in what you can do, or operators are used in ways you don't understand, try looking it up in another chapter. This chapter discusses strings and lists together since most commands work on both. However, bear in mind that the two are different entities.

It's never a bad idea to experiment on the command line and explore possibilities. Remember, if you want to know how Prometheus will handle a particular expression, just type it into the command line, possibly substituting constants where you want to track the behavior more closely, and see what happens. It's easier and faster and more accurate then any manual.

Adding Elements

There are a few ways to add elements to lists and strings. Append and prepend are two operators which add new elements to the beginning and end of a list. The first argument is the original list, and any number of arguments can follow. Each argument is appended or prepended to the original in succession. Therefore "append([a b] c d)" is "[a b c d]" and "append([a b] [c d])" is "[a b [c d]]" and "prepend([a b] c d)" is "[d c a b]". If the first argument is not a list, one is created, so "append(a b c)" is "[a b c]".

Sometimes a list will be sorted, and you will want to insert new elements in their sorted location. In that case use the insert command which takes the same arguments as append and prepend, and inserts the elements into the sorted list. If you use insert on an unsorted list, the elements will be inserted in an undefined location. Like append and prepend, if you give insert a non-list as the first argument, a new list will be created. To sort an unsorted list, use the sort command giving the list to be sorted as an argument.

If you want to insert an element into a particular location in a list, use the put operator with the same arguments as append and the index to the location in the third argument. This does not replace the element which is already there, and it will pad elements with null if you specify an index which doesn't exist. For example, "put([a b c] d 2)" is "[a d b c]", and "put([a b c] d 5)" is "[a b c [ ] d]". To place an element in a list at a location replacing the element which is currently there, use the putr command with the same arguments as put. The index can be any valid index, so "put([[a b] [c d]] e [1 2])" creates "[[a e b] [c d]]".

Besides the genmat operator described in the Vectors, Matrices, and Lists section of the Elements chapter, there is the setmat operator that creates a matrix of given dimensions in the first and second argument, and a value for each cell in the third argument. For example, "setmat(2 3 hi)" results in "[[hi hi hi] [hi hi hi]]".

Deleting Elements

You can delete elements with the del command. Supply the list or string in the first argument, and the index of the element in the second. The index can be any index used with the record operator including integers, lists of indices, and ranges. You cannot pile up del operators, so "del(del([[a b] [c d]] 2) 1)" will not delete "c", but will delete "[c d]" and then delete "a", leaving "[b]". Writing "del([[a b] [c d]] [2 1])" would delete just "c" leaving "[[a b] [d]]".

Characters in a string can be deleted in the same way. In addition, there is a special operator for stripping out the punctuation marks in a string. The operator is called strip and you supply a string that it strips and returns. If you give strip a list, then every element in the list is stripped.

If you have a list, you may apply the "unique" operator to it. Unique deletes any identical items in the list and returns the result. Normally you have only one argument — the list to work on. If the list is sorted, pass the constant "sorted" as the second parameter, and unique will run much faster. Unless the order of the elements is important, you should always pass sorted lists to unique as it will run (n/2)+1 times faster, if n is the number of elements in the list. You can use "unique(sort(YourListGoesHere) sorted)" to force your list to be sorted.

Splitting and Combining Elements

Many times in string processing you want to work with the words in a phrase or the letters in a word. To access the words in a phrase, use the explode operator to split it up into words. For example, "explode "this is a test"" yields "["this" "is" "a" "test"]". Punctuation is skipped, so "explode "this, is} a!-=# test?"" yields "["this" "is" "a" "test"]". If you explode a list, then each element of the list is exploded. You can explode part of a matrix by supplying an integer argument to explode. For example, "explode([["a a" "a b"] ["b a" "b b"]] 2)" would explode the second element of each row in the matrix, so the result would be "[["a a" ["b" "a"]] ["b a" ["b" "b"]]]". The operator isword returns TRUE if its argument is a word, and FALSE if not, a word being defined as a string whose exploded list has only one element, and that element is identical to the original string.

For lists, the flatten operator can be used to unnest lists. For example, "flatten [a [b c] [[d e] f] [g] h]" would be "[a b c d e f g h]". You can use the group operator to group elements together by specifying an index or range as the second parameter. Therefore "group([a b c d] 3)" is "[a b [c] d]" and "group([a b c d] 2..3)" is "[a [b c] d]". If you specify ranges that encompass elements that don't exist (i.e., accessing element -2 or 20 in a list of 15) then group acts like you were in bounds. For example, "group([a b c d] -3..15)" would have the same results as "group([a b c d] 1..4)".

The "subsets" operator generates a list of all subsets of a given list, including the empty list. For example, "subsets [1 2 3]" returns "[ ] [1] [1 2] [1 2 3] [1 3] [2] [2 3] [3]".

Searching Elements

Prometheus provides a fast and easy way to search through expressions. The find operator looks for an expression called the target in another expression called the search space. If it does not find the target in the search space, it returns FALSE. Otherwise, it returns a bag which contains the index to the found item. Therefore, "find(3 [[1 2] [3 4]])" is "[2 1]". This could be passed as the second parameter to the record operator. It is sometimes useful to remove the last element of the result of a find and then access that element, since that is the list which contains the target. If there is more then one occurrence of the target in the search space, then find returns the first one, determined by DFLR (Depth-First, Left-Right) order. If you need all of the occurrences, or at least more then one, use the findall operator which takes the same parameters as find but returns a list of all the locations of the target in the search space in DFLR order. Search will look in fractions and complex numbers, since it can record their indices in the results. Also, the search will look inside strings to match the target, and return the range of the matched text if found. However, it will only look for one occurrence in a string — if the target appears twice then only the first is returned. To search and replace, use the subst operator defined in the Algebra section of the Mathematics chapter.

Another seach operator is findelm. Given a list of expressions, this returns a list of all those expressions which contain the given target. For example, "findelm([[a b] [b c] [c d] [d a]] a)" would return "[[a b] [d a]]".

Another operator, perm, is useful for finding the permutation vector of a list and a permutation of that list; i.e., if A is a list, and B is a permutation of that list, then “perm(A B)” returns a list C such that the *i*th element of C is the element number in A which equals the *i*th element in B. Therefore, “perm([a b c d] [d b c a])” returns “[4 2 3 1]”. If an element in A is duplicated, the first occurance in DFLR order is used in the result. B may have duplicated elements without error. Also, A and B may be different lengths, and there can exist elements of A not present in B. If there is an element in B not in A, null is used as that index.

Miscellaneous Operators

The “rev” operator reverses the order of the elements in a list or characters in a string. Therefore “rev [a b c d]” results in “[d c b a]”. The “shuffle” operator moves a column of a matrix to the first column. Therefore “shuffle [[a b c d] [e f g h]] 3” moves the third column to the first yielding “[[c a b d] [g e f h]]”. This can be used to sort a multi-key matrix with a different order.

The “count” operator returns a matrix in which each row has two elements: the first is an expression which appears in the argument, and the second is the number of times it appears. The argument must be sorted or the results will be incorrect. For example, “count [a a a b c c d d e f]” results in “[[a 3] [b 1] [c 2] [d 2] [e 1] [f 1]”. To sort by frequency, shuffle the result bringing the second column first, and then sort.

For strings, there are three operators which change capitalization. The upper operator changes all characters to upper case, lower changes to lower case, and capt capitalizes the first character in the string, and makes the rest lower case.

Packages

Overview

Packages are files which extend the capabilities of Prometheus. They can be large databases of useful information or data files which Prometheus can exploit to run faster. For example, one package allows you to access any word in the English language, and another makes factorial computation from 2 to 1000 times faster. Packages are loaded on startup and unloaded at termination. Loading simply means that Prometheus is aware of their existence and will utilize them. To get Prometheus to load a package, you must put its file in a directory called "(Packs)". If this directory does not exist then no packages are loaded, and packages are never loaded from any other directory. Some packages like Factorial and Fibonacci create their own files if they do not already exist, but this will only happen if a "(Packs)" directory already exists. The sections in this chapter explain the purpose of each package.

Fibonacci and Factorial Packages

Although these are separate packages, they work in much the same way. Both enable you to execute some operations faster, namely getting the nth Fibonacci number or the nth factorial. When they are loaded, diagnostics are printed detailing the nature of the data. The important thing to note here is the last piece of information, the QuickSpan. Every time you use a number from zero up through the QuickSpan, you will get almost instantaneous results. Moving out of that range will begin to get sluggish, but it is still many times faster then not having the package loaded. Both of these packages have files associated with them, but you need not worry about them because they are automatically created for you if they don't already exist, and if the files are outdated, they are automatically updated.

Other Packages

There are other packages in which the data are there but the engine to drive the data from within Prometheus has not yet been established. In future versions of Prometheus, expect packages dealing with prime numbers, all the English words, a thesaurus, all the male and female names, all of the homonyms, and many more.

Porting

Overview

Porting is the technical term for translating between one computer language and another. Porting to and from Prometheus is easier then most languages because of the flexible input and output syntax. This chapter discusses issues involved when porting to and from some of the major languages.

Porting from LISP

Prometheus is intended to replace LISP, and it is one of the easiest porting languages because Prometheus understands LISP function structure, and can output it as well. The only changes come with the commands which are unique in LISP. Fortunately, it is possible to express every LISP command very easily in Prometheus. The following table shows each LISP command, and its equivalent in LISP:

LISP Command	Prometheus Equivalent
(car x)	x.1
(cdr x)	(del x 1)
(cons x y)	(append y x)
(list x y ... z)	[x y ... z] *or* (x y ... z)
nil	[ ]
(seq x y)	x = y
(reverse x)	(rev x)
(zerop x)	x == 0
(length x)	(size x)
(prine x)	(print x)
(print x)	(print x nl)
(last x)	x.(size x)

Also, Prometheus does everything LISP does and more with the following functions:

append	and	or	not
<	>	read	

Porting to C++

You can port Prometheus scripts to C++, which means you can convert scripts to C++ source code. The “outc” operator takes the name of a script file in the first argument, and creates a file in the same location and same name with “.cp” attached to the end. This file contains all the code needed to run that script with C++ (except for a few files, detailed later). The entire file is ANSI-compliant, so any compiler should accept it. A function is defined which returns “void” and has no arguments. Its name is identical to the script file name you supplied to outc. Calling this function will behave exactly as though “run "*FileName*"” had been typed on the command line. Even the environment is set up as it is in Prometheus. All console output uses cout, and all console input uses cin. When translating script to C++, comments are translated as well and imbedded in the C++ code.

Of course, this source code is not sufficient — the entire Prometheus intelligence and environment must be simulated from within your code. To do this you need two files. One is a library which contains all of the necessary code. The other is group of header (".h") files which lets the output from "outc" access this code. These files are included in each copy of Prometheus, unless this feature is not available to that kind of computer. For example, you cannot use this feature on non-Power Macintoshes because the library file is too large for the 680x0 segment loader to digest.

Properties

All operators have properties which determine how they are typed in, how they work, how many arguments can be accepted, and general simplification. The following table lists each property and what it indicates.

Property	Description
NUNARY	takes no arguments
URNARY	takes exactly one argument, degenerates to null with 0.
BINARY	takes exactly two arguments, degenerates to null with 1 or 0.
MULNARY	takes any number of arguments, degenerates into the argument if there is only one, and degenerates into null if there is none
FUNCT	with another argument type, does not degenerate
ASSOC	associative (e.g. a+(b + c)+d = a + b + c + d)
COMMUN	commutative (e.g. a + b = b + a)
EVEN	even (e.g. (func -x) = (func x)
ODD	odd (e.g. (func -x) = -(func x)
SELFOPP	inverse of self (e.g. neg neg x = x, inv inv x = x)
POST	post-fix
NONEST	nesting is gratuitous (e.g. abs abs abs x = abs x)
RTOL	evaluate from right to left (e.g. ?a := ?b := 2 evaluates from right to left so that the 2 is propagated to all variables)
EXECBLE	executable. Indicates that, for example, read + read shouldn't result in 2 * read.

Also, many functions distribute over lists and the logical connectives "and" and "or." These properties are not listed in the Command Summary.

Command Summary

Cmd	Properties	Description
^	BINARY	exponentiation
~	URNARY, SELFOPP	logical not, complementary probability
!	URNARY, POST	factorial
$	URNARY	literal
%	BINARY	modulus
&	MULNARY, ASSOC, COMMUN	probabilistic and (∧)
*	MULNARY, ASSOC, COMMUN	multiplication
+	MULNARY, ASSOC, COMMUN	addition
,	BINARY	complex separator
-	MULNARY	subtraction, arithmetic negation (-x)
.	BINARY	field specifier
..	BINARY	range
/	MULNARY	division

:=	BINARY, RTOL	define
;	NUNARY	row specifier (matrix)
<	MULNARY, ASSOC	less than
<=	MULNARY, ASSOC	less than or equal to (≤)
<>	MULNARY, ASSOC, COMMUN	not equal to (≠)
=	BINARY, RTOL	assign
=	MULNARY, ASSOC, COMMUN	equal to
>	MULNARY, ASSOC	greater than
>=	MULNARY, ASSOC	greater that or equal to (≥)
@	URNARY	pointer specifier
abs	URNARY, EVEN, NONEST	absolute value
acalc	URNARY	set the status of auto-calc
acos	URNARY	arc (inverse) cosine
acosh	URNARY	arc (inverse) hyperbolic cosine
acot	URNARY	arc (inverse) cotangent
acoth	URNARY, ODD	arc (inverse) hyperbolic cotangent
acsc	URNARY	arc (inverse) cosecant
acsch	URNARY, ODD	arc (inverse) hyperbolic cosecant
adj	URNARY	adjunct (matrix)
and	MULNARY, ASSOC, COMMUN	logical and (∧)
append	MULNARY, ASSOC, FUNCT	appends elements to a list
asec	URNARY	arc (inverse) secant
asech	URNARY	arc (inverse) hyperbolic secant
asin	URNARY, ODD	arc (inverse) sine
asinh	URNARY, ODD	arc (inverse) hyperbolic sine
atan	URNARY, ODD	arc (inverse) tangent
atanh	URNARY, ODD	arc (inverse) hyperbolic tangent
avg	URNARY	average
bag	MULNARY	fill elements of a list
calc	URNARY	calculate
capt	URNARY	capitalize
ceil	BINARY, FUNCT	computes the ceiling, optionally to a degree
cerf	URNARY	complementary error function (cerf x = 1-erf x)
cis	URNARY	cis (cis x = cos x + i*sin x)
col	BINARY	returns a column of a matrix
cos	URNARY, EVEN	cosine
cosh	URNARY, EVEN	hyperbolic cosine
cot	URNARY, ODD	cotangent
coth	URNARY, ODD	hyperbolic cotangent
count	URNARY, EXECBLE	counts consecutive occurrences
csc	URNARY, ODD	cosecant
csch	URNARY, ODD	hyperbolic cosecant

deg	NUNARY	change to degrees mode
del	URNARY, EXECBLE	delete (files, variables, elements)
der	BINARY, EXECBLE	derivative
disp	URNARY	display file; change display style
do	BINARY	performs math on elements of a list
echo	URNARY	echo
else	NUNARY	part of if-then-else control structure
env	NUNARY	environment list
erf	URNARY, ODD	error function
exec	URNARY	execute
exist	BINARY	existential quantifier (∃)
expdApr 18, 2021	EXECBLE	algebraic expand
explode	URNARY	splits elements in a list
factor	URNARY	factorization
fib	URNARY	Fibonacci number
find	BINARY	locates a target in a search space
findall	BINARY	locates all targets in a search space
finv	MULNARY	finds the inverse of a function
flatten	URNARY	flattens elements in a list
floor	BINARY, FUNCT	computes the floor, optionally to a degree
fmin	MULNARY	finds the minimum of a function
for	MULNARY, FUNCT	part of for control structure
forall	BINARY	universal quantifier (∀)
fuse	URNARY	combines elements in a list
fzero	MULNARY	finds the root of a function
gamma	URNARY	gamma function
gcd	BINARY	GCD (Greatest Common Divisor)
genmat	BINARY	generic matrix of constants
group	BINARY	group elements in a list
i	NUNARY	short for (0,1)
if	URNARY	part of if-then-else control structure
im	URNARY, ODD	imaginary part of an expression
imp	BINARY	logical implication (➔)
insert	MULNARY, ASSOC, FUNCT	inserts elements into a list
intg	MULNARY	integration
inv	URNARY, SELFOPP	arithmetic inverse (1/x)
isol	BINARY	isolate expression
isword	URNARY	tests if a string is one word
last	NUNARY	last evaluated result
lcm	BINARY	LCM (Least Common Multiple)
lg	URNARY	log base 2
lim	MULNARY	take a limit
ln	URNARY	Napiernian log
log	BINARY	general log
lower	URNARY	change to lower case

max	MULNARY, ASSOC, COMMUN	maximum
mem	NUNARY	bytes in heap and stack
min	MULNARY, ASSOC, COMMUN	minimum
minor	MULNARY	minor (matrix)
nCr	BINARY	combinations
neg	URNARY, SELFOPP	arithmetic negation (-x)
nl	NUNARY	new line string
not	URNARY, SELFOPP	logical not (~)
nPr	BINARY	permutations
open	URNARY, EXECBLE	read a saved expression from a file
opendb	MULNARY, FUNCT, EXECBLE	reads from a database file
openfm	MULNARY, FUNCT, EXECBLE	reads from a database file in flexible format
or	MULNARY, ASSOC, COMMUN	logical or ($\vee$)
outc	URNARY, EXECBLE	converts a script to C++ code
pause	NUNARY, EXECBLE	wait for user signal
pder	BINARY, EXECBLE	take partial derivative
perm	BINARY	find permutation
polyfit	BINARY, FUNCT	fit polynomial function
pop	NUNARY, EXECBLE	pop expression off the user stack
prepend	MULNARY, ASSOC, FUNCT	prepends elements to a list
print	MULNARY, ASSOC, FUNCT, EXECBLE	screen output
prod	MULNARY	product
push	MULNARY, ASSOC, FUNCT, EXECBLE	push expressions onto the user stack
put	MULNARY	inserts elements in a list
putr	MULNARY	inserts elements in a list replacing existing elements
rad	NUNARY	change to radians mode
rand	NUNARY	random number between 0 and 1 inclusive
range	MULNARY	computes ranged result
re	URNARY, ODD, NONEST	real part of an expression
read	NUNARY, EXECBLE	screen input
rev	URNARY, EXECBLE	reverses list
rot	MULNARY, FUNCT, EXECBLE	rotate user stack, rotate variable values
round	BINARY, FUNCT	rounds off, optionally to a degree
run	URNARY, EXECBLE	execute script
saug	URNARY	solve augmented matrix
save	BINARY, EXECBLE	save an expression to a file

savedb	MULNARY, EXECBLE	writes to a database file
sec	URNARY, EVEN	secant
sech	URNARY, EVEN	hyperbolic secant
setmat	MULNARY	creates a matrix with a value in each element
sgn	URNARY, ODD, NONEST	signum function
shuffle	BINARY, EXECBLE	moves columns of a matrix
simp	BINARY, EXECBLE	simplification specifier
simps	NUNARY	user simplification list
sin	URNARY, ODD	sine
sinh	URNARY, ODD	hyperbolic sine
sort	URNARY	sorts a list
sqr	URNARY, EVEN	arithmetic square
sqrt	URNARY	square root
stack	NUNARY	user stack
std	URNARY	standard deviation
subst	MULNARY	substitution
subsets	URNARY	generate subsets
sum	MULNARY, EXECBLE	summation
swap	BINARY, FUNCT, EXECBLE	swap the top two expressions on the user stack, swap the contents of two variables
tan	URNARY, ODD	tangent
tanh	URNARY, ODD	hyperbolic tangent
tb	NUNARY	tab string
tofrac	BINARY	convert real to fraction
trans	URNARY, SELFOPP	transpose (matrix)
unify	BINARY	modes ponens unification (logic)
unique	URNARY, EXECBLE	deletes copies in a list
upper	URNARY	change to upper case
var	URNARY	variance
vars	NUNARY	variable list
xand	BINARY	logical xand (p∧~q)
xor	MULNARY, ASSOC, COMMUN	logical xor ((p∨q)∧~(p∧q))
\	BINARY	fraction specifier
|	MULNARY, ASSOC, COMMUN	probabilistic or (∨)
sumsq	URNARY	sum the squares of the data
+-	MULNARY	plus or minus

Order of Precedence

Interior of Grouping Symbols (Parenthesis, Brackets, and Braces)
Post-fix Functions
Pre-fix Functions
In-Fix Functions
.
^
* / \ %
+ - +-
== <> > >= < <=
| imp xor xand
&
or
and
,
..
:= =
;

Simplification Rules

1. Notes
    1.1. /* this is a comment */
    1.2. a = any Expression except INF and -INF
    1.3. k = any Integer
    1.4. n = any Integer
    1.5. m = any Integer
    1.6. p = either Integer or Real
    1.7. q = either Integer or Real
    1.8. ?v = any variable
    1.9. anything else = any expression

1.10. rules are listed in approximate order of which types are involved

2. Intra Expression

2.1. Fraction

2.1.1. a\b /* where b is negative */ --> (-a)\(-b)

2.1.2. a\1 --> a

2.1.3. a\0 --> INF

2.1.4. (a\b)\c --> a\(b*c)

2.1.5. a\(b\c) --> (a*b)\b

2.1.6. a\a --> 1

2.1.7. (a*b*...*c)\(a*d*...*e) --> (b*...*c)\(d*...*e)

2.1.8. a\(a*d*...*e) --> \(d*...*e)

2.1.9. (a*b*...*c)\a --> (b*...*c)

2.2. Complex

2.2.1. (a,0) --> a

2.2.2. ((a,b),c) --> (a,b+c)

2.2.3. (a,(b,c)) --> (a-c,b)

3. Logical

3.1. not

3.1.1. not TRUE --> FALSE

3.1.2. not FALSE --> TRUE

3.1.3. not (x and ... and y) --> (not x or ... or not y)

3.1.4. not (x or ... or y) --> (not x and ... and not y)

3.1.5. not (forall x y) --> (exists x not y)

3.1.6. not (exists x y) --> (forall x not y)

3.2. and

3.2.1. (and TRUE ...) --> (and ...)

3.2.2. (and FALSE ...) --> FALSE

3.3. or

3.3.1. (or TRUE ...) --> TRUE

3.3.2. (or FALSE ...) --> (or ...)

3.3.3. (or b ... (and x ... y) ... d) --> (or b ... x ... d) and ... and (or b ... y ... d)

4. Probabilities

4.1. &

4.1.1. TRUE & x --> x

4.1.2. FALSE & x --> FALSE

4.1.3. p & q --> p*q

4.2. |

4.2.1. TRUE | x --> x

4.2.2. FALSE | x --> FALSE

4.2.3. p | q --> p+q-p*q

4.3. not

4.3.1. not p --> -p

5. Arithmetic

5.1. neg

5.1.1. -(-INF) --> INF

5.1.2. -(INF) --> -INF

5.1.3. -(x,y) --> (-x,-y)

5.1.4. -(x\y) --> -x\y

5.1.5. -[x ... y] --> [-x ... -y]

5.1.6. -(x+y+....+z) --> (-x+-y+...+-z)

5.1.7. -(p*x*...*y) --> (-p*x*...*z)

5.1.8. -(x and ... and y) --> -x and ... and -y

5.1.9. -(x or ... or y) --> -x or ... or -y

5.2. inv

5.2.1. 1/0 --> INF

5.2.2. 1/INF --> 0

5.2.3. 1/-INF --> 0

5.2.4. 1/(x,y) --> 1/(x^2+y^2)*(x,-y)

5.2.5. 1/(x\y) --> y\x

5.2.6. 1/[x ... y] -->

5.2.6.1. 1/abs [x ... y] /* if Vector */

5.2.6.2. 1/abs [x ... y]*adj trans [x ... y] /* matrix inverse otherwise */
5.2.7. 1/-x --> -1/x
5.2.8. 1/(x^y) --> x^-y
5.2.9. 1/(x*...*y) --> 1/x * ... * 1/y
5.2.10. 1/(x and ... and y) --> 1/x and ... and 1/y
5.2.11. 1/(x or ... or y) --> 1/x or ... or 1/y
5.3. +
5.3.1. 0+x --> x
5.3.2. -INF+a --> -INF
5.3.3. a+INF --> INF
5.3.4. INF+INF --> INF
5.3.5. -INF+-INF --> -INF
5.3.6. z+(x,y) --> (x+z,y)
5.3.7. (x,y)+(p,q) --> (x+p,y+q)
5.3.8. z+(x\y) --> (x+z*y)\y
5.3.9. x\y+p\q --> (x*q+y*p)\(y*q)
5.3.10. x+"text" --> "xtext" /* x is turned into a string and prepended */
5.3.11. "text"+x --> "textx" /* x is turned into a string and appended */
5.3.12. x+[y ... z] --> [x+y ... x+z] /* if x is not a Function */
5.3.13. [x y ... z] + [q r ... w] --> [x+q y+r ... z+w] /* if the vectors have the same number of elements */
5.3.14. { } + { } --> /* the conjunction of the two sets */
5.3.15. x+x --> 2*x
5.3.16. x+-x --> 0
5.3.17. a*x+x --> x*(a+1)
5.3.18. a*x+-x --> x*(a-1)
5.3.19. a*x+b*x --> x*(a+b)
5.3.20. a*x-b*x --> x*(a-b)
5.4. -
5.4.1. (- x y ... z) --> (+ x -y ... -z)
5.5. *
5.5.1. 0*a --> 0
5.5.2. 1*x --> x
5.5.3. -1*x --> -(x)
5.5.4. -INF*a --> -INF /* unless a<0, in which case INF */
5.5.5. a*INF --> INF /* unless a<0, in which case -INF */
5.5.6. INF*INF --> INF
5.5.7. -INF*-INF --> INF
5.5.8. -INF*INF --> -INF
5.5.9. z*(x,y) --> (x*z,y*z)
5.5.10. (x,y)*(p,q) --> (x*p-y*q,y*p+x*q)
5.5.11. z*(x\y) --> (x*z)\y
5.5.12. x\y*p\q --> (x*p)\(y*q)
5.5.13. n*"text" --> "texttexttext...text" /* text repeated n times, n>=0 */
5.5.14. x*[y ... z] --> [x*y ... x*z] /* if x is not a Function */
5.5.15. [x y ... z] * [q r ... w] -->
5.5.15.1. x*q + y*r + ... + z*w /* if either is a Vector */
5.5.15.2. /* the result of matrix multiplication otherwise */
5.5.16. { } * { } --> /* the disjunction of the two sets */
5.5.17. x*x --> x^2
5.5.18. x/x --> 1
5.5.19. x^y/x --> x^(y-1)
5.5.20. x*...*-y*...*z --> -(x*...*y*...*z)
5.5.21. x*x^y --> x^(y+1)
5.5.22. x^y*x^z --> x^(y+z)
5.6. /
5.6.1. (/ x y ... z) --> (* x /y ... /z)
5.7. ^
5.7.1. INF^a -->
5.7.1.1. INF /* if a > 0 */

5.7.1.2. 0 /* otherwise */
5.7.2. a^INF --> INF /* if a > 0 */
5.7.3. a^-INF --> 0 /* if a > 0 */
5.7.4. 0^a --> 1
5.7.5. 1^a --> 1
5.7.6. a^0 --> 1
5.7.7. x^1 --> x
5.7.8. x^-1 --> /x
5.7.9. (x,y)^z --> sqrt((x^+y^)^z/(z^+(y/x)^))*(z,y/x)
5.7.10. z^(x,y) --> z^x*(cos (y*ln z),sin (y*ln z))
5.7.11. (x\y)^z --> x^z\y^z
5.7.12. (1/x)^y --> x^-y
5.7.13. (x^y)^z --> x^(y*z)
5.7.14. (x*...*y)^z --> x^z*...*y^z
5.7.15. e^ln x --> x
5.7.16. 2^lg x --> x
5.7.17. x^log(x y) --> y
5.7.18. (x and ... and y)^z --> x^z and ... and y^z
5.7.19. (x or ... or y)^z --> x^z or ... or y^z
5.7.20. x^(y and ... and z) --> x^y and ... and x^z
5.7.21. x^(y or ... or z) --> x^y or ... or x^z

6. Algebra

6.1. %
6.1.1. x % 0 --> INF
6.1.2. INF % a --> INF
6.1.3. -INF % a --> -INF
6.1.4. 0 % x --> 0

6.2. abs
6.2.1. abs -INF --> INF
6.2.2. abs (x,y) --> sqrt(x^2+y^2) /* normal */
6.2.3. abs (x\y) --> abs x\y
6.2.4. abs [x ... y] -->
6.2.4.1. sqrt(x^+...+y^) /* if Vector */
6.2.4.2. /* determinant of matrix otherwise */
6.2.5. abs -x --> abs x
6.2.6. abs /x --> /abs x
6.2.7. abs x! --> x!
6.2.8. abs (x*...*y) --> abs x*...*abs y

6.3. sqr
6.3.1. sqr x --> x^2

6.4. sqrt
6.4.1. sqrt x --> x^0.5

6.5. sgn
6.5.1. sgn e --> 1
6.5.2. sgn pi --> 1
6.5.3. sgn phi --> 1
6.5.4. sgn phi' --> -1
6.5.5. sgn euler --> 1
6.5.6. sgn neg x --> -1
6.5.7. sgn abs x --> 1
6.5.8. sgn INF --> 1
6.5.9. sgn -INF --> -1

6.6. floor
6.6.1. floor [x ... y] --> [floor x ... floor y]
6.6.2. floor [x ... y] z --> [floor(x z) ... floor(y z)]

6.7. ceil
6.7.1. ceil [x ... y] --> [ceil x ... ceil y]
6.7.2. ceil [x ... y] z --> [ceil(x z) ... ceil(y z)]

6.8. round
6.8.1. round [x ... y] --> [round x ... round y]
6.8.2. round [x ... y] z --> [round(x z) ... round(y z)]

7. Logarithm

- 7.1. log
  - 7.1.1. log (x 0) --> -INF
  - 7.1.2. log (x x) --> 1
  - 7.1.3. log (y -x) --> (log (y x),pi) /* also works if x < 0 */
  - 7.1.4. log (z x^y) --> y*log (z x)
- 7.2. lg
  - 7.2.1. lg 0 --> -INF
  - 7.2.2. lg -x --> (lg x,pi) /* also works if x < 0 */
  - 7.2.3. lg (x^y) --> y*lg x
- 7.3. ln
  - 7.3.1. ln 0 --> -INF
  - 7.3.2. ln e --> 1
  - 7.3.3. ln -x --> (ln x,pi) /* also works if x < 0 */
  - 7.3.4. ln (x,y) --> (ln (x^+y^)/,atan(y/x))
  - 7.3.5. ln (x^y) --> y*ln x

8. Trigonometric
- 8.1. cos
  - 8.1.1. cos (a,b) --> (cos a*cosh b,-sin a*sinh b)
- 8.2. sin
  - 8.2.1. sin (a,b) --> (sin a*cosh b,cos a*sinh b)
- 8.3. tan
  - 8.3.1. tan (a,b) --> sin (a,b)/cos (a,b)
- 8.4. csc
  - 8.4.1. csc x --> 1/sin x
- 8.5. sec
  - 8.5.1. sec x --> 1/cos x
- 8.6. cot
  - 8.6.1. cot x --> 1/tan x
- 8.7. acos
  - 8.7.1. acos (a,b) --> ((0,-1)*ln ((a,b)+(((((2*a*b)^2)+((1+(a^2)+-(b^2))^2))^0.25)*(cos (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))),sin (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2))))))))
- 8.8. asin
  - 8.8.1. asin (a,b) --> ((0,-1)*ln ((-b,a)+(((((2*a*b)^2)+((1+(a^2)+-(b^2))^2))^0.25)*(cos (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))),sin (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2))))))))
- 8.9. atan
  - 8.9.1. atan (a,b) --> (((-0.5*ln (((1+-b)^2)+(-a^2)))+(0.5*ln ((a^2)+((1+b)^2)))),(atan ((1+b)*1/a)+atan ((1+-b)*1/a)))
- 8.10. acsc
  - 8.10.1. acsc x --> asin 1/x
- 8.11. asec
  - 8.11.1. asec x --> acos 1/x
- 8.12. acot
  - 8.12.1. acot x --> atan 1/x
- 8.13. cosh
  - 8.13.1. cosh (a,b) --> ((cos b*cosh a),(sin b*sinh a))
- 8.14. sinh
  - 8.14.1. sinh (a,b) --> ((cos b*sinh a),(cosh a*sin b))
- 8.15. tanh
  - 8.15.1. tanh (a,b) --> (((((cos b^2)*cosh a*sinh a)+((sin b^2)*cosh a*sinh a)),(((cosh a^2)*cos b*sin b)+-((sinh a^2)*cos b*sin b)))*1/(((cos b*cosh a)^2)+((sin b*sinh a)^2)))
- 8.16. csch
  - 8.16.1. csch x --> 1/sinh x
- 8.17. sech
  - 8.17.1. sech x --> 1/cosh x
- 8.18. coth
  - 8.18.1. coth x --> 1/tanh x

8.19. acosh
8.19.1. acosh (a,b) --> ln ((a,b)+(((((2*a*b)^2)+((1+(a^2)+-(b^2))^2))^0.25)*(cos (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))),sin (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))))))
8.20. asinh
8.20.1. asinh (a,b) --> ln ((a,b)+(((((2*a*b)^2)+((1+(a^2)+-(b^2))^2))^0.25)*(cos (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))),sin (0.5*atan (2*a*b*1/(1+(a^2)+-(b^2)))))))
8.21. atanh
8.21.1. atanh (a,b) --> ((0.5*((-0.5*ln (((1+-a)^2)+(-b^2))) +(0.5*ln ((b^2)+((1+a)^2))))),(0.5*(atan (b*1/(1+a))+atan (b*1/(1+-a)))))
8.22. acsch
8.22.1. acsch x --> asinh 1/x
8.23. asech
8.23.1. asech x --> acosh 1/x
8.24. acoth
8.24.1. acoth x --> atanh 1/x
9. Relation
9.1. /* any relation with TRUE or FALSE is simplified as such */
10. Calculus
10.1. der
10.1.1. der($?x $?x) --> 1
10.1.2. der(b+...+c $?x) --> der(b $?x)+...+der(c $?x)
10.1.3. der(b*c...*d*e $?x) --> der(b $?x)*c*...*d*e+b*der(c $?x)*...*d*e+...+b*c*...*d*der(e $?x)
10.1.4. der(-b $?x) --> -der(b $?x)
10.1.5. der(1/b $?x) --> -der(b $?x)/b^2
10.1.6. der(b^c $?x) --> b^c*der(c $?x)*ln b + b^(c-1)*c*der(b $?x)
10.1.7. der(ln b $?x) --> der(b $?x)/b
10.1.8. der(sin b $?x) --> der(b $?x)*cos b
10.1.9. der(cos b $?x) --> neg der(b $?x)*sin b
10.1.10. der([b ... c] $?x) --> [der(b $?x) ... der(c $?x)]
10.1.11. der(r $?x) /* where r is none of the above cases, and not a function */ --> 0
10.2. pder
10.2.1. /* same rules, except */
10.2.2. der($?v $?x) /* where v ≠ x */ --> der($?v $?x)
10.3. intg
11. Number Theory
11.1. gcd
11.1.1. gcd(a b c ... d) --> gcd(...(gcd(a b)) c) ... d)
11.1.2. gcd(b c) /* b and c are neither numbers nor functions */ --> 1
11.1.3. gcd(b*c d) --> gcd(b d)*gcd(c d)
11.1.4. gcd(b^n d) --> gcd(b d)
11.1.5. gcd(b^n d^m) --> gcd(b d)^min(n m)
11.2. lcm
11.2.1. lcm(x y) --> x*y/gcd(x y)
11.2.2. lcm(x y ... z) --> lcm(lcm(x y) ... z)
12. Misc.
12.1. max
12.1.1. (max p q ...) --> (max q ...)
12.1.2. (max x x) --> (max x)
12.1.3. (max -INF ...) --> (max ...)
12.1.4. (max INF ...) --> INF
12.1.5. a+b-max(a b) --> min(a b)
12.2. min
12.2.1. (min p q ...) --> (min p ...)
12.2.2. (min x x) --> (min x)
12.2.3. (min -INF ...) --> -INF
12.2.4. (min INF ...) --> (min ...)
12.2.5. a+b-min(a b) --> max(a b)

12.3. re
12.3.1. re p --> p
12.3.2. re INF --> INF
12.3.3. re -INF --> -INF
12.3.4. re im p --> im p
12.3.5. re (b,c) --> b
12.3.6. re [b ... c] --> [re b ... re c]
12.3.7. re (b+...+c) --> re b+...+re c
12.3.8. re (b*c) --> ((re a*re b)+-(im a*im b))
12.3.9. re (1/b) --> (1/((im a^2)+(re a^2))*re a)
12.3.10. re cos b --> (cos re a*cosh im a)
12.3.11. re sin b --> (cosh im b*sin re b)
12.3.12. re tan b --> (1/(cos (2*re a)+cosh (2*im a))*sin (2*re a))
12.3.13. re (b^2) --> ((re b^2)+-(im b^2))
12.4. im
12.4.1. im p --> 0
12.4.2. im INF --> 0
12.4.3. im -INF --> 0
12.4.4. im im p --> 0
12.4.5. im re p --> 0
12.4.6. im (b,c) --> c
12.4.7. im [b ... c] --> [im b ... im c]
12.4.8. im (b+...+c) --> im b+...+im c
12.4.9. im (b*c) --> ((im a*re b)+-(im b*re a))
12.4.10. im (1/b) --> -(im a*1/((im a^2)+(re a^2)))
12.4.11. im cos b --> (sin re a*sinh im a)
12.4.12. im sin b --> (cos re a*sinh im a)
12.4.13. im tan b --> (1/(cos (2*re a)+cosh (2*im a))*sinh (2*im a))
12.4.14. im (b^2) --> (2*im b*re b)
12.5. nPr
12.5.1. nPr(x x) --> 1
12.5.2. nPr(x y) /* where x and y are not integers */ --> x!/y!
12.6. nCr
12.6.1. nCr(x x) --> 1
12.6.2. nCr(x 0) --> 1
12.6.3. nCr(x 1) --> x
12.6.4. nCr(x y) /* where x and y are not integers */ --> nPr(x,y)/(x-y)!
12.7. sum
12.7.1. sum(?v a b x) /* b < a */ --> sum(?v b a x)
12.7.2. sum(?v a b x) /* x does not contain ?v */ --> (b-a+1)*x
12.7.3. sum(?v a b -x) --> -sum(?v a b x)
12.7.4. sum(?v a b (x*y)) /* x does not contain ?v but y does */ --> x*sum(?v a b y)
12.7.5. sum(?v a b ?v) --> (b*(b+1)+a*(1-a))/2

Technical Reference

This section details various operators and describes how they work.

The fzero operator uses an excellent algorithm which minimizes the number of times the function must be evaluated. It uses the bisection procedure combined with linear or quadratic inverse interpolation. At each step, both bisection and interpolation approximations are calculated. Whichever falls within the given interval and lies closest to the middle is deemed the best and is used as the new approximation. If *a* is the lower bound, *b* is the upper bound, and *t* is the tolerance, then the maximum number of times the function could be evaluated is $\ln\frac{b-\alpha}{1.6\tau}$. The fmin operator uses the same algorithm, but subtracts the desired inverse from the function to make it a zero.

The fmin operator uses the gold section procedure combined with parabolic interpolation. At each step the best of the two is taken, making the maximum number of times the function could be evaluated $\ln \frac{b-\alpha}{1.324\tau}$.

The error function and the complementary error function use a table to determine small values, and call each other to reduce computation on large values. The real to rational engine works by creating partial fractions to greater and greater depths until the tolerance has been achieved.

The random number algorithm is a combination of a Fibonacci sequence with lags of 97 and 33, a "subtraction plus one, modulo one" operation, and an arithmetic sequence using subtraction. It has a period of 2^{144} (22300745198530623141535718272648361505980416 or about $2.23\text{x}10^{43}$) and passes all the tests for random number generation. The initial conditions are picked to maximize the period. There are algorithms which produce longer periods, but this scores better on randomness tests, and is very fast, and it is unlikely that the period is too short for some problem. Even if you had a computer which could generate a trillion of these random numbers every second, the amount of time it would take to start seeing repeated numbers is longer then the age of the universe about a trillion times over. In the words of David LaSalle, (see the Writer/Contributor Information and Bibliography chapter for more information) and with his own capitalization, "THIS IS THE BEST KNOWN RANDOM NUMBER GENERATOR AVAILABLE. .... It passes ALL of the tests for random number generators." Of course, this was asserted a number of years ago (the exact date is unclear, but it was definitely after 1987 and probably around 1990).

Taking the determinant currently uses a very inefficient but accurate method. It uses expansion by minors, and therefore the determinant of an *n* by *n* matrix takes *n*! multiplications and ?? additions. LU factorization or some other decomposition method would be preferable as long as Prometheus could still maintain constants and other non-numbers in the matrix. Taking the inverse is expensive mainly due to the inefficient determinants: the inverse of an *n* by *n* matrix takes $n^2(1+(\nu-1)!)$ multiplications and $n^2 ADDITIONS(n-1)$ additions. The saug operator triangularizes the matrix, and then works bottom up to build the values for the variables. It is very efficient: for *n* equations it uses only $n^2-\nu$ additions and n^2 multiplications.

The gamma operator uses the identity $\frac{1}{\Gamma(\xi)} = \xi\varepsilon^{\gamma\xi}\prod_{\mu=1}^{\infty}\frac{1+\frac{\xi}{\mu}}{\varepsilon^{\frac{\xi}{\mu}}}$ where γ is Euler's constant to compute the gamma function. However, several changes must be made since the computer cannot loop to $m = \infty$, and there are better ways to compute the product which require less computer time. The form actually used is $\frac{1}{\Gamma(\xi)} = \xi\varepsilon^{\xi\beta}\prod_{\mu=1}^{\nu}(1+\frac{\xi}{\mu})$ where *n* is a reasonable large number and $b=\gamma-\sum_{\kappa=1}^{\nu}\frac{1}{\kappa}$ (with $n=1000000$, $b\approx-13.815510057963456$). As $n\to\infty$, the gamma operator becomes more accurate, but it takes longer to evaluate. Because this formula converges so slowly, the value is not very accurate.

For factoring integers, a very fast method is used for numbers with fewer then ten digits. Otherwise factorization occurs, but more slowly. Any prime factors greater then or equal to $2^{16}+1$ (65537) will not be found, and their product will appear as the largest prime factor in the factorization.

Case Study

This chapter describes a real world problem which Prometheus is designed to solve. It takes advantage of the string and list processing commands more then anything else. The problem is you are trying to pick a good software package to fill some need. The information you have is a large database of companies and their products with descriptions. This chapter explains the creation and the actual code for various programs which search and analyze data, and put it into useful formats.

As a first example, let's make a program which looks for a particular feature in the database and prints a list of all the products which have that feature. We'll make a script to accomplish this. Let's say the database is in exact count format with four fields: the product, the company, the company's address, and the summary of the product. The first thing we need to do is read the database, ignoring the second and third fields. The first line in the script is:

```
data = opendb("database" 4 r x x)
```

Now our database is in the variable data. The next step is to locate all occurrences of the feature, which we can do with the findall command. We can look for "flexible" with:

```
loc = findall("flexible" data)
```

If we examined the contents of loc, we would find a matrix where each row was a found location consisting of three elements: the first is the row in data this was found, then the element in that row which will be "2" unless a product name includes the word "flexible", and then a range giving the exact location of the word "flexible" in the summary. What we wish to do now is make a list of all the product names which go along with the hits in loc. We will loop through the elements of loc and build the list in prod:

```
// first clear prod in case it is already defined
del prod
for(hit loc)
      prod = append(prod data.(hit.1).1)
```

The last line is constructed as follows: We are accessing the data matrix, and we want the row containing the product name. The row number can be found in the first element in hit. Then, the first element of that row is the product name, which we then append to the prod list. Now prod contains a list of product names which we could print to the screen with the print command:

```
print prod
```

Now the script is complete, but let's put in some bells and whistles. First of all, the database may not be sorted by product name, so instead of appending the elements to prod, we could insert them. Also, we could print messages on the screen alerting the user about what the program is doing. Furthermore, let's write the product listing to a file on the disk because we want to use that information elsewhere, or perhaps it's just too much information to fit on the screen at once. Here is a listing with all of these changes enacted:

```
print "Reading database...." nl
data = opendb("database" 4 r x x)
print "Finding flexible...." nl
loc = findall("flexible" data)
print "Getting product names...." nl
del $prod
for(hit loc)
        prod = insert(prod data.(hit.1).1)
print "Saving to Results.flexible...." nl
savedb(prod "Results.flexible" 1)
```

We could also make the code a little more understandable. The line in the loop is very convoluted and it takes some effort to understand what is going on. We could put in a comment to describe the logic, but there is a better way. Since all we are interested in is the first column of loc, let's extract that column in the first place. Then we won't have to access a specific element of hit. To do this, we change the loc assignment statement to "loc = col(findall("flexible" data) 1)" and the looping statement to "prod = insert(prod data.hit.1)".

Now as an additional feature, let's make this program more flexible. Instead of always searching for the word "flexible", let's have it get a word from the user, and search for it. The program would be as follows:

```
print "Input the word to search for:" nl
word = read
print "Reading database...." nl
data = opendb("database" 4 r x x)
print "Finding " word "..." nl
loc = col(findall(word data) 1)
print "Getting product names...." nl
del $prod
for(hit loc)
        prod = insert(prod data.hit.1)
print "Saving to Results."  word "...." nl
savedb(prod sorted "Results."+word 1)
```

This is sufficient for one search term. But what if we want like a report based on a whole list of search terms We will extend the program to read in a list of search terms from a file and make a report about the products for each. We'll call the various search terms "keys," and the program must now loop over these keys, making a list of products for each. The list of keys and products, which we wish to export into a tab-delimited file, will be put in the variable ring. The following listing accomplishes this, and the messages printed for the user and the comments will be helpful in understanding what is done at each step:

```
print "Reading database...." nl
data = opendb("database" 4 r x x)
print "Reading key file...." nl
// we flatten this file because keys separated by return carriages will be in
// each row of the keys matrix when we want keys to be a simple list.
keys = sort(flatten(opendb("keys" 1)))
del $ring
for(key keys)
        print tb "Finding " key "...." nl
        loc = col(findall(key data) 1)
        print tb "Getting product names for " key "..." nl
        // now we REALLY need to delete prod, because we're in a loop.
        del $prod
        for(hit loc)
                prod = insert(prod data.hit.1)
        // let's put the key at the beginning for reference in the output,
        // after uniquing
        prod = prepend(unique(prod sorted) key)
        // now we insert prod into ring
        ring = insert(ring prod)
print "Saving results...." nl
savedb(ring "Results")
```

The next program will produce a matrix for use in a spreadsheet or table program. Down the left side will be product names, and across the top will be terms in the key list. Each cell will contain a "1" or a "0" indicating the feature is present or not present for a given product. We will be modifying the previous program to accomplish this. We remove the last line because we are changing the output. The first thing we need is a list of all the products in the matrix. An easy way to do this is take the first column of data with "prods = col(data 1)". However, that might include products which contain no features in the key list. That might not be a problem, but we could also stick to products with at least one match in the key list. To do this we need to get the union of all but the first column of the ring matrix. Also, to make the matrix, it will be more convenient for ring to contain two elements in each element: the feature and a list of products. To solve both of these problems in one fell swoop, we delete the prod prepend statement, and the ring insert statement changes to "ring = insert(ring [key prod])". Then prods is created by "prods = unique(sort(flatten(col(ring 2))) sorted)".

So now prods contains a sorted list of all relevant products, and ring contains a matrix of keys and a list of products containing that key. We will use mat as the final matrix. We can start by initializing mat to a matrix of 0's, and then put 1's according to prods and ring. To initialize, use the setmat command with the number of rows equal to the size of prods and the number of columns equal to the size of ring with "mat = setmat(size prods size ring 0)". Now we must loop through each feature in ring and for each product listed, mark a 1 in the appropriate element. This is done with the following code:

```
for(hit ring)
        keyLoc = find(hit.1 keys).1
        for(prod hit.2)
                prodLoc = find(prod prods).1
                mat = putr(mat 1 [prodLoc keyLoc])
```

Of course, there are many variations on this piece of code, all of which do the same job. For instance, instead of looping through the elements of ring, you could loop with hit being an index in 1..(size ring) and then keyLoc will be the same as hit. In this case the second for-loop would have to be modified to loop over “ring.hit.2”, and not just “hit.2”. Also, the assignment of prodLoc and keyLoc could be eliminated if the find commands were placed in the putr command. This would actually result in faster and more efficient code because variables take time to process. They were used initially to provide clarity.

Nonetheless, mat now contains the desired matrix without titles. We can refer to prods to get the titles of each row and the first column of ring to get the titles of each column of mat. If we wanted to put these titles into mat to make it more readable, we append the following code:

```
for(k 1..size(prods))
        mat = put(mat prods.k [k 1])
// this isn’t just column 1 of ring because we need to leave a
// blank for the first column in mat which now contains titles.
mat = put(prepend(mat col(ring 1)) "Products" [1 1])
```

Then a simple “savedb(mat "Matrix_Results")” will write the matrix to spreadsheet-readable format. Of course, the 1’s and 0’s could be any expression at all, but they were chosen so that further analysis could be done with mathematics. For example, you could sum a column of mat (without titles) to get the number of companies with that feature, or you could sum a row of mat (without titles) to get the number of features a company has.

Now we’ll add some more flexibility to the report. Let’s say that the feature “BOM” and “Bill of Materials” is the same feature, and therefore we want one column in mat which contains 1’s if either “BOM” or “Bill of Materials” occurs in a particular product. One strategy is to look for “BOM” and “Bill of Materials” separately and then combine later, but a more elegant solution can be found with a key matrix instead of a key list in keys. The key matrix will be a tab-delimited file with the first element in a data set is the term we want as the title of a column in mat, and the other elements are synonymous terms. A complete listing of the program with this implemented follows:

```
print "Reading database...." nl
data = opendb("database" 4 r x x)
print "Reading key file...." nl
keys = opendb("keys_with_syn")
del $ring
for(keyList keys)
        print tb "Gathering information for " keyList.1 "..." nl
        for(key keyList)
                print tb tb "Finding " key "...." nl
                loc = col(findall(key data) 1)
                print tb tb "Getting product names for " key "..." nl
                del $prod
                for(hit loc)
                        prod = insert(prod data.hit.1)
        ring = insert(ring [keyList.1 unique(prod sorted)])
// this is to make future searches faster since all but column #1
// have been assimilated
keys = sort col(keys 1)
print "Building Matrix...." nl
print tb "Initializing matrix...." nl
prods = unique(sort(flatten(col(ring 2))) sorted)
mat = setmat(size prods size ring 0)
print tb "Processing keys...." nl
for(hit ring)
        print tb tb "Processing " hit.1 "...." nl
        keyLoc = find(hit.1 keys).1
        for(prod hit.2)
                prodLoc = find(prod prods).1
                mat = putr(mat 1 [prodLoc keyLoc])
print tb "Adding Titles...." nl
for(k 1..size(prods))
        mat = put(mat prods.k [k 1])
mat = put(prepend(mat col(ring 1)) "Products" [1 1])
print "Saving Results...." nl
savedb(mat "Results")
```

Let us now expand this further to include categories of properties. We could say, for example, that "accounting" could be defined as "accounts receivable" and "invoice." We have a tab-delimited file where the first element in a row is the category, and each successive element is a member of that category. We want to have a matrix output as before, but with categorical columns with breakdowns by function. So, for example, the first two columns of the matrix would be "accounts receivable" and "invoice," and the title of these two columns together would be "accounting." Let us further assume that we wish to use the key list with synonyms that was implemented in the last example.

We will need another variable to hold the categories and an addition loop to go through the category elements. We'll want to sort the properties within the categories, and sort the categories as a whole. Plus, if a key in the category does not have a listing in the key list, then no synonyms should be used. Also, so far we have not been using implicit assignment, but from now on it will be used. The following listing implements these changes:

```
print "Reading database...." nl
data = opendb("database" 4 r x x)
print "Reading key file...." nl
keys = opendb("keys_with_syn")
print "Reading cat file...." nl
cats = opendb("cat")
del $catList
del $Products
nCols = 0
for(cat cats)
        print "Category " cat.1 nl
        del $props
        for(prop del(cat 1))
                print tb "Property " prop nl
                f = find(prop keys)
                del $prod
                if (f)
                        for(key keys.(f.1))
                                print tb tb "Key " key nl
                                loc = col(findall(key data) 1)
                                for(hit loc)
                                        insert(?prod data.hit.1)
                else
                        print tb tb "Key " prop nl
                        loc = col(findall(prop data) 1)
                        for(hit loc)
                                insert(?prod data.hit.1)
                unique(?prod sorted)
                flatten(append(?Products prod))
                insert(?props [prop prod])
        insert(?catList [cat.1 props])
        ?nCols + size props
print "Building matrix" nl
unique(sort(?Products) sorted)
mat = setmat(2+size Products nCols 0)
c = 1
```

```
for(cat catList)
        print tb "Building category " cat.1 nl
        oldc = c
        for(prop cat.2)
                print tb tb "Building Property " prop.1 nl
                putr(?mat "" [1 c])
                putr(?mat prop.1 [2 c])
                for(comp prop.2)
                        f = find(comp Products)
                        putr(?mat 1 [f.1+2 c])
                ?c+1
        putr(?mat cat.1 [1 oldc])
print tb "Attaching Product Titles" nl
put(?mat "" [1 1])
put(?mat "" [2 1])
for(k 1..size(Products))
        put(?mat Products.k [k+2 1])
print "Saving Results...." nl
savedb(mat "Results")
```

The put and putr commands involving “""” place blank space where setmat had initialized with “0”. The first part of the script folds all the information into lists, sorting as it goes, and when the matrix is filled, each list is traversed and titles are placed where necessary. The output is ready to be formatted with a table or spreadsheet program. With this final, complex program, the merits of the print statement become two-fold: one is alerting the user of progress, convenient in a lengthy procedure, and the other is providing landmarks which make reading the code a little easier.

There are still other features we could include in the program. For example, we might like to have the company which produced the product printed alongside it in mat. To do that, we simply write code mirroring the method of prepending product names to each line in the matrix, except we must first read the company data from the database:

```
cData = opendb("database" 4 r r x x)
put(?mat "" [1 1])
put(?mat "" [2 1])
for(k 1..size(Products))
        comp = cData.(find(Products.k cData).1).1
        put(?mat comp [k+2 1])
```

If this is inserted before the loop which puts product titles into the matrix, then the program will run as specified. If it is inserted after that loop, and you sort mat before output, then you can have the same matrix sorted by company.

This case study focused on the list and string processing facets of Prometheus. The program listings exemplify the incredible flexibility and ease of use Prometheus provides. Even the very complex problems took only a handful of lines of code, and forty percent of those lines were just print statements. Prometheus gives you the power of mathematics and computer science with a few keystrokes enabling you to solve many kinds of problems with ease.

Writer/Contributor Information and Bibliography

The writer, Jason Cohen, is a senior at LBJ High School in Austin, Texas and currently works in the Analysis and Applied Research Division at Trãcor Aerospace. He wrote all of the over 130,000 lines of C++ code which make up Prometheus. He also devised all the algorithms except those cited below, translating from FORTRAN where indicated.

The univariate function minimizer engine is based an algorithm from "Brent" (the definitive source is being located) which in turn is based on the algorithm from G. Forsythe, M. Malcolm, C. Moler, Computer methods for mathematical computations. M., Mir, 1980, p.202 of the Russian edition. The univariate function zero locator is also based an algorithm from "Brent" which is in turn based on an algorithm from the same source, p. 180.

The real to rational converter is based on an algorithm from Jerome Spanier and Keith B. Oldham, An Atlas of Functions. Springer-Verlag, 1987, pp. 665-7.

The random number generator has a long history. It started in Toward a Universal C Random Number Generator by George Marsaglia and Arif Zaman. C Florida State University Report: FSU-SCRI-87-50 (1987). It was later modified by F. James and published in "A Review of Pseudo-C random Number Generators". David LaSalle of Florida State University created a FORTRAN program that was translated to C by Jim Butler, and got slightly rewritten before the final insertion into Prometheus.

The GCD engine for integers less then 2^{31} is based on an algorithm due to J. Stein in 1961 (see Journal of Computational Physics, 1 (1967), 397-405). See also D. E. Knuth, Seminumerical Algorithms. The Art of Computer Programming Vol. 2. Addison-Wesley Publishing Company, Reading Mass, 1981. p. 321.

The error function and complementary error function tabulator is based on an algorithm written by Robert C. Tausworthe at the Jet Propulsion Laboratory in 1984.

The C-style escape sequence translator is based on an algorithm by Jerry Coffin.

You can visit the Prometheus web cite at:

http://www.aard.tracor.com/Jason/Prometheus/ where you will find support, information, contacts, and the latest versions of the Prometheus executable and manual. Jason Cohen can be contacted by e–mail at "jason@tracor.com". Questions, comments, suggestions, bug alerts, feedback, new ideas and algorithms, etc. are encouraged!