Minimal Maxima
맥시마 맛뵈기

Robert Dodier
양사장 애벌 번역

April 18, 2010

1 What is Maxima?
맥시마가 뭐여?

Maxima[1] is a system for working with expressions, such as $x+y$, $\sin(a+b\pi)$, and $u \cdot v - v \cdot u$.

맥시마는 $x + y$, $\sin(a + b\pi)$, 그리고 $u \cdot v - v \cdot u$와 같은 표현식[2]을 다루는 시스템입니다.

Maxima is not much worried about the meaning of an expression. Whether an expression is meaningful is for the user to decide.

맥시마는 표현식이 가지는 의미는 별로 신경 안씁니다. 표현식이 지대로 된 지 아닌 지는 쓰는 사람 나름 아니겠어요?

Sometimes you want to assign values to the unknowns and evaluate the expression. Maxima is happy to do that. But Maxima is also happy to postpone assignment of specific values; you might carry out several manipulations of an expression, and only later (or never) assign values to unknowns.

[1] Home page: http://maxima.sourceforge.net
Documents: http://maxima.sourceforge.net/docs.shtml
Reference manual: http://maxima.sourceforge.net/docs/manual/en/maxima.html

[2] 역자 주: 'expression'은 식, 수식, 표현식, 표현 등으로 번역 가능하나 이 문서에서는 한 단어로 통일하지 않았습니다. 문맥에 따라 식, 수식, 표현, 표현식으로 대충 번역하였습니다.

시나브로 모르는 수식 변수에 숫자를 대입해보고 답이 뭔지 알고 싶지요? (별로 안 그런데...) 맥시마는 그런 거 아주 잘 해요. 하지만 꼭 단번에 대입하지는 않아요. 되도록이면 수식을 먼저 풀고, 다 풀고 나서 막판에 변수에 대입하든지.... 아님 말고...

Let's see some examples.
예를 들어보면,

1. I want to calculate the volume of a sphere.
   동그란 공이 있고, 그 부피를 구하고 싶을 때,

```
(%i1) V: 4/3 * %pi * r^3;
                                      3
                                 4 %pi r
(%o1)                            --------
                                    3
```

2. The radius is 10.
   반지름이 10이고,

```
(%i2) r: 10;
(%o2)                               10
```

3. V is the same as before; Maxima won't change V until I tell it to do so.
   부피 V는 아직 앞에서 구한 거랑 같아요, 맥시마는 하랄 때까지 V를 바꾸지 않고 기다리거든요.

```
(%i3) V;
                                      3
                                 4 %pi r
(%o3)                            --------
                                    3
```

4. Please re-evaluate V, Maxima.
   착한 맥시마님, (반지름을 대입해서) 부피를 다시 계산해주셔요!

```
(%i4) ''V;
                                 4000 %pi
(%o4)                            --------
                                    3
```

5. I'd like to see a numerical value instead of an expression.
   그럼 이제 부피를 수식 대신 숫자로 보고 싶으면,

```
(%i5) ''V, numer;
(%o5)                    4188.79020478639
```

2 Expressions
표현식

Everything in Maxima is an expression, including mathematical expressions, objects, and programming constructs. An expression is either an atom, or an operator together with its arguments.

맥시마는표현식을 빼면 시체입니다. 수식이든, 'object'든, 아님 'programming construct'든 말이지요. 이 표현식은 'atom'이든 지, 연산자(operator)이든 지, 아니면 짬뽕이든 지.. 그렇습니다. 관련된 (연산) 인자(argument)[3]들과 같이요.

An atom is a symbol (a name), a string enclosed in quotation marks, or a number (integer or floating point).

'atom'은 기호(또는 이름)인데요. 따옴표로 감싼 문자열이든 지, 아님 정수나 실수와 같은 숫자입니다.

All nonatomic expressions are represented as $op(a_1, \ldots, a_n)$ where op is the name of an operator and $a_1, \ldots, a_n$ are its arguments. (The expression may be displayed differently, but the internal representation is the same.) The arguments of an expression can be atoms or nonatomic expressions.

'nonatomic'[4] 표현식은 $op(a_1, \ldots, a_n)$로 나타내고요, op는 연산자(operator) 이름입니다. 그리고 $a_1, \ldots, a_n$은 인자(argument)입니다.

Mathematical expressions have a mathematical operator, such as $+ \; - \; * \; / \; < \; = \; >$ or a function evaluation such as $\mathbf{sin}(x), \mathbf{bessel\_j}(n, x)$. In such cases, the operator is the function.

수식은 $+ \; - \; * \; / \; < = >$와 같은 산술 연산자나 $\mathbf{sin}(x)$, $\mathbf{bessel_j}(n, x)$와 같은 함수로 구성됩니다.

Objects in Maxima are expressions.
A list $[a_1, \ldots, a_n]$ is an expression $\mathbf{list}(a_1, \ldots, a_n)$. A matrix is an expression

$$\mathbf{matrix}(\mathbf{list}(a_{1,1}, \ldots, a_{1,n}), \ldots, \mathbf{list}(a_{m,1}, \ldots, a_{m,n}))$$

[3] 역자 주: 때에 따라, 인자, 연산 인자 등으로 대충 번역 되었습니다.

[4] 역자 주: 'atom', 'atomic', 'nonatomic'은 번역이 의미를 망칠까봐 번역하지 않았습니다. 데이터 베이스나 OS 시스템 콜에서 쓰이는 의미와 같다고 보시면 되지요. '원자적 표현식'으로 번역하면 누가 알아듣겠어요? 적당한 번역이 있으면 알려주시기를...

모든 오브젝트(Object)는 표현식입니다.
리스트 $[a_1, \ldots, a_n]$도 식 $\mathbf{list}(a_1, \ldots, a_n)$이고, 행렬도 그렇습니다.

$$\mathbf{matrix}(\mathbf{list}(a_{1,1}, \ldots, a_{1,n}), \ldots, \mathbf{list}(a_{m,1}, \ldots, a_{m,n}))$$

Programming constructs are expressions. A code block $\mathbf{block}(a_1, \ldots, a_n)$ is an expression with operator **block** and arguments $a_1, \ldots, a_n$. A conditional statement **if** a **then** b **elseif** c **then** d is an expression $\mathbf{if}(a, b, c, d)$. A loop **for** a **in** L **do** S is an expression similar to $\mathbf{do}(a, L, S)$.

'construct'도 표현식입니다. 코드 $\mathbf{block}(a_1, \ldots, a_n)$는 **block**이라는 연산자와 $a_1, \ldots, a_n$와 같은 인자들로 구성된 표현식입니다. 조건문(참 어려운 말이네요. '조건문') **if** a **then** b **elseif** c **then** d은 $\mathbf{if}(a, b, c, d)$로 나타낼 수 있는 표현식이고요. **for** a **in** L **do** S와 같은 반복은 $\mathbf{do}(a, L, S)$와 비슷한 표현식입니다.

The Maxima function **op** returns the operator of a nonatomic expression. The function **args** returns the arguments of a nonatomic expression. The function **atom** tells whether an expression is an atom.

맥시마 함수인 **op**는 'nonatomic expression'의 연산자를 리턴하고요. **args** 함수는 인자들을, **atom** 함수는 표현식이 'atom'인 지 아닌 지 알려줍니다.

Let's see some more examples.
예를 더 들어보면,

1. Atoms are symbols, strings, and numbers. I've grouped several examples into a list so we can see them all together.
   'atom'은 기호, 문자열, 숫자입니다. 여러 예를 같이 볼 수 있도록 리스트로 묶어보았습니다.

   ```
   (%i2) [a, foo, foo_bar, "Hello, world!", 42, 17.29];
   (%o2)      [a, foo, foo_bar, Hello, world!, 42, 17.29]
   ```

2. Mathematical expressions.
   수식

   ```
   (%i1) [a + b + c, a * b * c, foo = bar, a*b < c*d];
   (%o1)       [c + b + a, a b c, foo = bar, a b < c d]
   ```

3. Lists and matrices. The elements of a list or matrix can be any kind of expression, even another list or matrix.
   리스트와 행렬.
   아무 표현식이라도 리스트나 행렬의 원소가 될 수 있습니다. 다른 리스트나 행렬도 원소가 될 수 있구요.

```
(%i1) L: [a, b, c, %pi, %e, 1729, 1/(a*d - b*c)];
                                                1
(%o1)          [a, b, c, %pi, %e, 1729, ---------]
                                             a d - b c
(%i2) L2: [a, b, [c, %pi, [%e, 1729], 1/(a*d - b*c)]];
                                                  1
(%o2)        [a, b, [c, %pi, [%e, 1729], ---------]]
                                               a d - b c
(%i3) L [7];
                                 1
(%o3)                         ---------
                              a d - b c
(%i4) L2 [3];
                                          1
(%o4)            [c, %pi, [%e, 1729], ---------]
                                       a d - b c
(%i5) M: matrix ([%pi, 17], [29, %e]);
                              [ %pi  17 ]
(%o5)                         [         ]
                              [ 29   %e ]
(%i6) M2: matrix ([[%pi, 17], a*d - b*c], [matrix ([1, a], [b, 7]), %e]);
                     [ [%pi, 17]  a d - b c ]
                     [                      ]
(%o6)                [ [ 1  a ]             ]
                     [ [      ]      %e     ]
                     [ [ b  7 ]             ]
(%i7) M [2][1];
(%o7)                            29
(%i8) M2 [2][1];
                               [ 1  a ]
(%o8)                          [      ]
                               [ b  7 ]
```

4. Programming constructs are expressions. $x : y$ means assign y to x; the value of the assignment expression is y. **block** groups several expressions, and evaluates them one after another; the value of the block is the value of the last expression.

   'construct'도 표현식입니다. $x : y$는 y를 x에 대입하라는 건데, y가 대입되는 값입니다. **block**은 여러 표현식을 묶어서 하나씩 풀어갑니다. 그래서 block 값은 마지막 표현식의 값이 되지요.

```
(%i1) (a: 42) - (b: 17);
```

```
(%o1)                          25
(%i2) [a, b];
(%o2)                       [42, 17]
(%i3) block ([a], a: 42, a^2 - 1600) + block ([b], b: 5, %pi^b);
                             5
(%o3)                      %pi  + 164
(%i4) (if a > 1 then %pi else %e) + (if b < 0 then 1/2 else 1/7);
                                   1
(%o4)                       %pi + -
                                   7
```

5. **op** returns the operator, **args** returns the arguments, **atom** tells whether an expression is an atom.
   op 함수는 연산자를 리턴합니다. **args** 함수는 인자들을 리턴하고요, **atom** 함수는 표현식이 'atom'인 지 아닌 지를 리턴합니다.

```
(%i1) op (p + q);
(%o1)                           +
(%i2) op (p + q > p*q);
(%o2)                           >
(%i3) op (sin (p + q));
(%o3)                         sin
(%i4) op (foo (p, q));
(%o4)                         foo
(%i5) op (foo (p, q) := p - q);
(%o5)                         :=
(%i6) args (p + q);
(%o6)                       [q, p]
(%i7) args (p + q > p*q);
(%o7)                   [q + p, p q]
(%i8) args (sin (p + q));
(%o8)                       [q + p]
(%i9) args (foo (p, q));
(%o9)                      [p, - q]
(%i10) args (foo (p, q) := p - q);
(%o10)               [foo(p, q), p - q]
(%i11) atom (p);
(%o11)                       true
(%i12) atom (p + q);
(%o12)                       false
(%i13) atom (sin (p + q));
(%o13)                       false
```

6. Operators and arguments of programming constructs. The single quote tells Maxima to construct the expression but postpone evaluation. We'll come back to that later.
'construct'의 연산자와 인자.
작은 따옴표는 표현식을 만들게 하면서도 그 평가는 뒤로 미루지요. 나중에 다시 얘기하기로 하구요.

```
(%i1) op ('(block ([a], a: 42, a^2 - 1600)));
(%o1)                          block
(%i2) op ('(if p > q then p else q));
(%o2)                           if
(%i3) op ('(for x in L do print (x)));
(%o3)                          mdoin
(%i4) args ('(block ([a], a: 42, a^2 - 1600)));
                                    2
(%o4)               [[a], a : 42, a  - 1600]
(%i5) args ('(if p > q then p else q));
(%o5)                   [p > q, p, true, q]
(%i6) args ('(for x in L do print (x)));
(%o6)     [x, L, false, false, false, false, print(x)]
```

3 Evaluation
평가

The value of a symbol is an expression associated with the symbol. Every symbol has a value; if not otherwise assigned a value, a symbol evaluates to itself. (E.g., x evaluates to x if not otherwise assigned a value.)

기호의 값은 그 기호의 표현식입니다. 모든 기호는 값을 가지고요. 값이 없으면 그냥 그 기호가 값이 됩니다. x에 지정된 값이 없으면 그냥 x인 셈이지요.

Numbers and strings evaluate to themselves.
숫자랑 문자열은 그 자체가 값이고요.

A nonatomic expression is evaluated approximately as follows.
'nonatomic' 표현은 다음과 같이 평가됩니다.

1. Each argument of the operator of the expression is evaluated.
연산자에 관련된 인자들이 먼저 평가되고요.

2. If an operator is associated with a callable function, the function is called, and the return value of the function is the value of the expression.
연산자가 함수를 부를 수 있으면, 함수를 부르고 그 함수의 리턴 값이 표현식의 값이됩니다.

Evaluation is modified in several ways. Some modifications cause less evaluation:
여러가지 평가 방법이 있는 데요, 어쩔 때는 뜨문뜨문 하기도 합니다:

1. Some functions do not evaluate some or all of their arguments, or otherwise modify the evaluation of their arguments.
어떤 함수는 그 인자들 중에서 일부, 어쩔 때는 전부 다 건너 뛰고, 아니면 인자들 평가 값을 바꾸기도 합니다.

2. A single quote $'$ prevents evaluation.
작은 따옴표 $'$는 평가를 못하게 해요.

   (a) $'a$ evaluates to a. Any other value of a is ignored.
   $'a$ 는 그냥 a로... 다른 값은 무시.

   (b) $'f(a_1, \ldots, a_n)$ evaluates to $f(\mathbf{ev}(a_1), \ldots, \mathbf{ev}(a_n))$. That is, the arguments are evaluated but f is not called.
   $'f(a_1, \ldots, a_n)$는 $f(\mathbf{ev}(a_1), \ldots, \mathbf{ev}(a_n))$처럼.
   뭐냐면 함수 인자들은 평가하고, 함수는 부르지 않는다는 얘기지요. 헤헤.

   (c) $'(\ldots)$ prevents evaluation of any expressions inside (...).
   $'(\ldots)$는 (...) 안에 어떤 표현식이 들었든 지 왕무시.

Some modifications cause more evaluation: 어쩔 때는 디따시 많이 평가해요:

1. Two single quotes $''a$ causes an extra evaluation at the time the expression a is parsed.
   $''a$처럼 작은 따옴표 두 개는[5] a가 처리[6]될 때 한 번 더 평가되게 하고요.

2. $\mathbf{ev}(a)$ causes an extra evaluation of a every time $\mathbf{ev}(a)$ is evaluated.
   $\mathbf{ev}(a)$는 $\mathbf{ev}(a)$가 평가될 때마다 a도 평가되게 합니다.

[5] 역자 주: LaTeX에서는 작은 따옴표 두 개가 위 영문 본문처럼 큰 따옴표 하나로 보입니다. 원저자에게 중간에 블랭크를 넣는 게 어떨까 물었더니, 혼란을 줄 수 있기 때문에 싫다고 해서시험삼아 저만 넣어 봅니다. 위의 영문과 아래 번역된 곳에서 따옴표의 모양이 틀리지요. 번역본 버전 1.1 에서는 Jaime Villate의 충고를 받아들여 작은 따옴표들을 수학 문구 바깥으로 뺐습니다.

[6] 역자 주: parse에 적당한 우리말이 뭔 지 몰라서 그냥 '처리'라고 하였습니다. 구문 분석이란 말은 좀 어렵지 않나요?

3. The idiom **apply**$(f, [a_1, \ldots, a_n])$ causes the evaluation of the arguments $a_1, \ldots, a_n$ even if f ordinarily quotes them.
apply$(f, [a_1, \ldots, a_n])$는 함수 f가 그냥 끌고 다니드라도 $a_1, \ldots, a_n$을 평가하게 합니다.

4. **define** constructs a function definition like :=, but **define** evaluates the function body while := quotes it.
define는 :=처럼 함수 정의를 하는 데, **define**은 함수 본체를 평가하고요, :=는 안 합니다.

Let's consider how some expressions are evaluated.
어떻게 되는 건 지 한 번 볼까요?

1. Symbols evaluate to themselves if not otherwise assigned a value.
기호는 지정된 값이 없으면 그냥 그대로.

```
(%i1) block (a: 1, b: 2, e: 5);
(%o1)                           5
(%i2) [a, b, c, d, e];
(%o2)                  [1, 2, c, d, 5]
```

2. Arguments of operators are ordinarily evaluated (unless evaluation is prevented one way or another).
연산자의 인자들은, 우야둥둥 방해받지 않으면, 일반적인 방법으로 평가됩니다.

```
(%i1) block (x: %pi, y: %e);
(%o1)                            %e
(%i2) sin (x + y);
(%o2)                       - sin(%e)
(%i3) x > y;
(%o3)                       %pi > %e
(%i4) x!;
(%o4)                         %pi!
```

3. If an operator corresponds to a callable function, the function is called (unless prevented). Otherwise evaluation yields another expression with the same operator.
연산자가 부를 수 있는 함수[7]이면 함수를 부릅니다. (금지 되어 있지 않으면요.) 아니면 같은 연산자로 묶인 다른 표현식을 평가합니다.

[7]역자 주: '호출가능 함수'라는 말이 좀 딱딱해 보여서...

```
(%i1) foo (p, q) := p - q;
(%o1)                    foo(p, q) := p - q
(%i2) p: %phi;
(%o2)                           %phi
(%i3) foo (p, q);
(%o3)                         %phi - q
(%i4) bar (p, q);
(%o4)                       bar(%phi, q)
```

4. Some functions quote their arguments. Examples: **save**, :=, **kill**.
   save, :=, **kill**처럼 인자를 인용하는 함수도 있습니다.

```
(%i1) block (a: 1, b: %pi, c: x + y);
(%o1)                           y + x
(%i2) [a, b, c];
(%o2)                       [1, %pi, y + x]
(%i3) save ("tmp.save", a, b, c);
(%o3)                          tmp.save
(%i4) f (a) := a^b;
                                       b
(%o4)                         f(a) := a
(%i5) f (7);
                                  %pi
(%o5)                             7
(%i6) kill (a, b, c);
(%o6)                             done
(%i7) [a, b, c];
(%o7)                          [a, b, c]
```

5. A single quote prevents evaluation even if it would ordinarily happen.
   작은 따옴표는 놔두면 진행될 평가를 방지하고요. (아, 놔!)

```
(%i1) foo (x, y) := y - x;
(%o1)                    foo(x, y) := y - x
(%i2) block (a: %e, b: 17);
(%o2)                             17
(%i3) foo (a, b);
(%o3)                          17 - %e
(%i4) foo ('a, 'b);
(%o4)                             b - a
(%i5) 'foo (a, b);
(%o5)                         foo(%e, 17)
(%i6) '(foo (a, b));
(%o6)                          foo(a, b)
```

6. Two single quotes (quote-quote) causes an extra evaluation at the time the expression is parsed.
   작은 따옴표 두 개는 표현식이 (구문) 분석될 때 평가되게 합니다.

```
(%i1) diff (sin (x), x);
(%o1)                          cos(x)
(%i2) foo (x) := diff (sin (x), x);
(%o2)                foo(x) := diff(sin(x), x)
(%i3) foo (x) := ''(diff (sin (x), x));
(%o3)                    foo(x) := cos(x)
```

7. **ev** causes an extra evaluation every time it is evaluated. Contrast this with the effect of quote-quote.
   ev는 매 (**ev**) 평가시 한번 더 (인자를) 평가되게 합니다. 작은 따옴표 두 개랑 비교하시면서 밑 예제를 봐주세요.

```
(%i1) block (xx: yy, yy: zz);
(%o1)                             zz
(%i2) [xx, yy];
(%o2)                           [yy, zz]
(%i3) foo (x) := ''x;
(%o3)                         foo(x) := x
(%i4) foo (xx);
(%o4)                             yy
(%i5) bar (x) := ev (x);
(%o5)                      bar(x) := ev(x)
(%i6) bar (xx);
(%o6)                             zz
```

8. **apply** causes the evaluation of arguments even if they are ordinarily quoted.
   apply는 인자들을 평가하게 합니다.

```
(%i1) block (a: aa, b: bb, c: cc);
(%o1)                             cc
(%i2) block (aa: 11, bb: 22, cc: 33);
(%o2)                             33
(%i3) [a, b, c, aa, bb, cc];
(%o3)                 [aa, bb, cc, 11, 22, 33]
(%i4) apply (kill, [a, b, c]);
(%o4)                           done
(%i5) [a, b, c, aa, bb, cc];
```

```
(%o5)                [aa, bb, cc, aa, bb, cc]
(%i6) kill (a, b, c);
(%o6)                           done
(%i7) [a, b, c, aa, bb, cc];
(%o7)                  [a, b, c, aa, bb, cc]
```

9. **define** evaluates the body of a function definition.
   define은 함수 본체를 평가합니다.

```
(%i1) integrate (sin (a*x), x, 0, %pi);
                           1   cos(%pi a)
(%o1)                      - - ----------
                           a       a
(%i2) foo (x) := integrate (sin (a*x), x, 0, %pi);
(%o2)       foo(x) := integrate(sin(a x), x, 0, %pi)
(%i3) define (foo (x), integrate (sin (a*x), x, 0, %pi));
                                1   cos(%pi a)
(%o3)               foo(x) := - - ----------
                                a       a
```

4 Simplification
단순화, 인수 분해

After evaluating an expression, Maxima attempts to find an equivalent expression which is "simpler." Maxima applies several rules which embody conventional notions of simplicity. For example, $1 + 1$ simplifies to 2, $x + x$ simplifies to $2x$, and **sin**(%**pi**) simplifies to 0.

표현식을 평가한 후, 맥시마는 더 단순한 표현식을 찾으려고 합니다. 단순하게 표현하기 위해서 여러가지 법칙을 적용하지요. 예를 들자면, $1+1$는 2로, $x+x$는 $2x$로, **sin**(%**pi**)는 0로 단순해집니다.

However, many well-known identities are not applied automatically. For example, double-angle formulas for trigonometric functions, or rearrangements of ratios such as $a/b + c/b \rightarrow (a + c)/b$. There are several functions which can apply identities.

하지만 전부 다 자동으로 되는 건 아니구요. 예를 들자면, 삼각함수의 배각 공식이라든가 $a/b + c/b \rightarrow (a + c)/b$처럼 공통 분모 같은 거 있잖아요. 이런 속성을 가진 함수들이 좀 있습니다.

Simplification is always applied unless explicitly prevented. Simplification is applied even if an expression is not evaluated.

단순화는 금지되지 않았으면 항상 이루어지고요, 표현식이 평가되지 않더라도 적용됩니다.

tellsimpafter establishes user-defined simplification rules.
tellsimpafter은 쓰는 사람 맘대로 단순화 법칙을 만들 수 있게 합니다.

Let's see some examples of simplification.
예를 들어 볼까요? 헤헤.

1. Quote mark prevents evaluation but not simplification. When the global flag **simp** is **false**, simplification is prevented but not evaluation.

   인용 부호는 평가를 금지하지만 단순화는 놔두지요. 전역 플래그 **simp**가 **false**면, 단순화가 금지되고, 평가는 놔둡니다. 반대네요.

```
(%i1) '[1 + 1, x + x, x * x, sin (%pi)];
                                    2
(%o1)                      [2, 2 x, x , 0]
(%i2) simp: false$
(%i3) block ([x: 1], x + x);
(%o3)                           1 + 1
```

2. Some identities are not applied automatically. **expand**, **ratsimp**, **trigexpand**, **demoivre** are some functions which apply identities.

   자동으로 안 되는 것도 있다고 했지요. **expand**, **ratsimp**, **trigexpand**, **demoivre** 같은 것은 되게 합니다.

```
(%i1) (a + b)^2;
                                    2
(%o1)                         (b + a)
(%i2) expand (%);
                           2            2
(%o2)                     b  + 2 a b + a
(%i3) a/b + c/b;
                              c   a
(%o3)                         - + -
                              b   b
(%i4) ratsimp (%);
```

```
                                c + a
(%o4)                           -----
                                  b
(%i5) sin (2*x);
(%o5)                        sin(2 x)
(%i6) trigexpand (%);
(%o6)                     2 cos(x) sin(x)
(%i7) a * exp (b * %i);
                                  %i b
(%o7)                          a %e
(%i8) demoivre (%);
(%o8)                 a (%i sin(b) + cos(b))
```

5 apply, map, and lambda

1. **apply** constructs and evaluates an expression. The arguments of the expression are always evaluated (even if they wouldn't be otherwise).
apply는 표현식을 만들고 평가합니다. 표현식 안의 인자들은 (안되게 되어 있더라도) 항상 평가됩니다.

```
(%i1) apply (sin, [x * %pi]);
(%o1)                         sin(%pi x)
(%i2) L: [a, b, c, x, y, z];
(%o2)                      [a, b, c, x, y, z]
(%i3) apply ("+", L);
(%o3)                    z + y + x + c + b + a
```

2. **map** constructs and evaluates an expression for each item in a list of arguments. The arguments of the expression are always evaluated (even if they wouldn't be otherwise). The result is a list.
map은 인자 리스트 내 각각에 대하여 표현식을 만들고 평가합니다. 표현식 안의 인자들은 항상 평가됩니다. 결과는 리스트입니다.

```
(%i1) map (foo, [x, y, z]);
(%o1)                 [foo(x), foo(y), foo(z)]
(%i2) map ("+", [1, 2, 3], [a, b, c]);
(%o2)                    [a + 1, b + 2, c + 3]
(%i3) map (atom, [a, b, c, a + b, a + b + c]);
(%o3)             [true, true, true, false, false]
```

3. **lambda** constructs a lambda expression (i.e., an unnamed function). The lambda expression can be used in some contexts like an ordinary named function. **lambda** does not evaluate the function body.
lambda는 lambda 식 (즉, 이름없는 함수)를 만듭니다. lambda 식은 이름있는 함수처럼 특정 경우에 사용될 수 있는 데, 함수 본체를 평가하지는 않습니다.

```
(%i1) f: lambda ([x, y], (x + y)*(x - y));
(%o1)            lambda([x, y], (x + y) (x - y))
(%i2) f (a, b);
(%o2)                      (a - b) (b + a)
(%i3) apply (f, [p, q]);
(%o3)                      (p - q) (q + p)
(%i4) map (f, [1, 2, 3], [a, b, c]);
(%o4)  [(1 - a) (a + 1), (2 - b) (b + 2), (3 - c) (c + 3)]
```

6 Built-in object types
내장 오브젝트 형

An object is represented as an expression. Like other expressions, an object comprises an operator and its arguments.

오브젝트는 표현식으로 표현됩니다. 다른 표현식들과 마찬가지로, 오브젝트는 연산자와 그 인자들로 구성됩니다.

The most important built-in object types are lists, matrices, and sets.

리스트, 행렬, 집합과 같은 것들이 있습니다.

6.1 Lists
리스트

1. A list is indicated like this: $[a, b, c]$.
리스트는 다음과 같이 나타냅니다: $[a, b, c]$

2. If L is a list, $L[i]$ is its i'th element. $L[1]$ is the first element.
L이 리스트면, $L[i]$는 i번째 원소이구요. $L[1]$이 첫번째 원소[8]가 됩니다.

[8] 역자 주: 보통 리스트에서는 그 원소를 노드라고도 하지요.

3. **map**(f, L) applies f to each element of L.
map(f, L)는 함수 f를 각 원소 L에 적용합니다.

4. **apply**$(" + ", L)$ is the sum of the elements of L.
apply$(" + ", L)$는 리스트 L 원소의 합이 됩니다.

5. **for** x **in** L **do** *expr* evaluates *expr* for each element of L.
for x **in** L **do** *expr*는 리스트 L 각각 원소에 *expr*을 적용하여 평가합니다.

6. **length**(L) is the number of elements in L.
length(L)는 L의 원소의 갯수를 구합니다.

6.2 Matrices
행렬

1. A matrix is defined like this: **matrix**$(L_1, \ldots, L_n)$ where $L_1, \ldots, L_n$ are lists which represent the rows of the matrix.
행렬은 다음과 같이 정의 됩니다: $L_1, \ldots, L_n$와 같은 리스트가 행렬의 각 행이 될 때, 행렬 **matrix**$(L_1, \ldots, L_n)$이 됩니다.

2. If M is a matrix, $M[i, j]$ or $M[i][j]$ is its (i, j)'th element. $M[1, 1]$ is the element at the upper left corner.
M이 행렬일 때, $M[i, j]$ 또는 $M[i][j]$은 (i, j) 자리에 위치한 원소(행렬에서는 성분 또는 항이라고 하지요)가 되고, $M[1, 1]$은 맨위 왼쪽의 항입니다.

3. The operator . represents noncommutative multiplication. $M.L$, $L.M$, and $M.N$ are noncommutative products, where L is a list and M and N are matrices.
행렬 연산자 .는 (교환법칙이 성립되지 않는) 행렬 곱셈을 나타냅니다. L이 리스트[9]이고, M과 N이 행렬일 때, $M.L$, $L.M$랑 $M.N$은 교환법칙이 성립하지 않는 곱셈입니다.

4. **transpose**(M) is the transpose of M.
transpose(M)는 행렬 M의 전치.

5. **eigenvalues**(M) returns the eigenvalues of M.
eigenvalues(M)는 행렬 M의 고유값.

6. **eigenvectors**(M) returns the eigenvectors of M.
eigenvectors(M)는 행렬 M의 고유벡터.

[9] 역자 주: 벡터

7. **length**(M) returns the number of rows of M.
   length(M)는 행렬 M의 행길이.

8. **length**(**transpose**(M)) returns the number of columns of M.
   length(**transpose**(M))는 행렬 M의 열길이.

6.3 Sets
집합

1. Maxima understands explicitly-defined finite sets. Sets are not the same as lists; an explicit conversion is needed to change one into the other.
   맥시마는 명확하게 정의된 유한 집합을 알아봅니다. 집합은 리스트와 비슷하지만 같지는 않아서, 리스트에서 집합으로 또는 그 반대로 변환하려면 명백하게 표현하는 게 필요합니다.

2. A set is specified within curly braces {} like this: $\{a, b, c, \ldots\}$ where the set elements are $a, b, c, \ldots$.
   집합은 꼬부랑괄호 {}로 나타냅니다: $a, b, c, \ldots$가 집합의 원소일 때, $\{a, b, c, \ldots\}$처럼 하면 집합이 되지요.

3. **union**(A, B) is the union of the sets A and B.
   union(A, B)는 집합 A 랑 B의 합집합.

4. **intersection**(A, B) is the intersection of the sets A and B.
   intersection(A, B)는 집합 A 랑 B의 교집합.

5. **cardinality**(A) is the number of elements in the set A.
   cardinality(A)는 집합 A의 원소 갯수.

7 How to...
워떠케...

7.1 Define a function
함수 정의하기

1. The operator := defines a function, quoting the function body.
   연산자 :=는 함수본체를 인용하여 함수를 정의합니다.

   In this example, **diff** is reevaluated every time the function is called. The argument is substituted for x and the resulting expression is evaluated. When the argument is something other than a symbol, that causes

an error: for **foo**(1) Maxima attempts to evaluate **diff**(**sin**(1)^2, 1).
아래 예를 들자면, 함수 **diff**는 호출될 때마다 재평가됩니다. 함수의 변수 x는 함수 호출 인자(argument)로 대치되어서는 재평가됩니다. 인자가 기호가 아니면 에러가 납니다. **foo**(1)에 대해서 맥시마는 **diff**(**sin**(1)^2, 1)로 간주합니다.

```
(%i1) foo (x) := diff (sin(x)^2, x);
                                          2
(%o1)               foo(x) := diff(sin (x), x)
(%i2) foo (u);
(%o2)                      2 cos(u) sin(u)
(%i3) foo (1);
Non-variable 2nd argument to diff:
1
#0: foo(x=1)
 -- an error.
```

2. **define** defines a function, evaluating the function body.
define도 함수를 정의하는 데 쓰입니다. 함수 본체를 (정의할 때) 평가합니다.

   In this example, **diff** is evaluated only once (when the function is defined). **foo**(1) is OK now.
   아래 예를 들면, 함수가 정의될 때 **diff**가 한 번만 돌아가지요. 이번에는 **foo**(1)도 에러 안나고 잘 돌지요.

```
(%i1) define (foo (x), diff (sin(x)^2, x));
(%o1)                foo(x) := 2 cos(x) sin(x)
(%i2) foo (u);
(%o2)                      2 cos(u) sin(u)
(%i3) foo (1);
(%o3)                      2 cos(1) sin(1)
```

7.2 Solve an equation
방정식 풀기

```
(%i1) eq_1: a * x + b * y + z = %pi;
(%o1)                   z + b y + a x = %pi
(%i2) eq_2: z - 5*y + x = 0;
(%o2)                     z - 5 y + x = 0
(%i3) s: solve ([eq_1, eq_2], [x, z]);
               (b + 5) y - %pi      (b + 5 a) y - %pi
```

```
(%o3)   [[x = - ---------------, z = -----------------]]
                     a - 1                 a - 1
(%i4) length (s);
(%o4)                            1
(%i5) [subst (s[1], eq_1), subst (s[1], eq_2)];
       (b + 5 a) y - %pi   a ((b + 5) y - %pi)
(%o5) [----------------- - ------------------- + b y = %pi,
             a - 1                 a - 1
                   (b + 5 a) y - %pi   (b + 5) y - %pi
                   ----------------- - --------------- - 5 y = 0]
                         a - 1               a - 1
(%i6) ratsimp (%);
(%o6)                   [%pi = %pi, 0 = 0]
```

7.3 Integrate and differentiate
미적분

integrate computes definite and indefinite integrals.
정적분, 부정적분 모두 **integrate**로 계산합니다.

```
(%i1) integrate (1/(1 + x), x, 0, 1);
(%o1)                          log(2)
(%i2) integrate (exp(-u) * sin(u), u, 0, inf);
                                1
(%o2)                           -
                                2
(%i3) assume (a > 0);
(%o3)                         [a > 0]
(%i4) integrate (1/(1 + x), x, 0, a);
(%o4)                       log(a + 1)
(%i5) integrate (exp(-a*u) * sin(a*u), u, 0, inf);
                                1
(%o5)                           ---
                                2 a
(%i6) integrate (exp (sin (t)), t, 0, %pi);
                         %pi
                        /
                        [      sin(t)
(%o6)                   I    %e       dt
                        ]
                        /
```

```
                          0
(%i7) 'integrate (exp(-u) * sin(u), u, 0, inf);
                         inf
                        /
                        [      - u
(%o7)                   I    %e    sin(u) du
                        ]
                        /
                         0
```

diff computes derivatives.
diff는 미분을 계산합니다.

```
(%i1) diff (sin (y*x));
(%o1)         x cos(x y) del(y) + y cos(x y) del(x)
(%i2) diff (sin (y*x), x);
(%o2)                        y cos(x y)
(%i3) diff (sin (y*x), y);
(%o3)                        x cos(x y)
(%i4) diff (sin (y*x), x, 2);
                               2
(%o4)                       - y  sin(x y)
(%i5) 'diff (sin (y*x), x, 2);
                             2
                            d
(%o5)                       --- (sin(x y))
                              2
                            dx
```

7.4 Make a plot
도표 작성

plot2d draws two-dimensional graphs.
2차원 도표는 **plot2d**로 그립니다.

```
(%i1) plot2d (exp(-u) * sin(u), [u, 0, 2*%pi]);
(%o1)
(%i2) plot2d ([exp(-u), exp(-u) * sin(u)], [u, 0, 2*%pi]);
(%o2)
(%i3) xx: makelist (i/2.5, i, 1, 10);
```

```
(%o3)  [0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 3.6, 4.0]
(%i4) yy: map (lambda ([x], exp(-x) * sin(x)), xx);
(%o4) [0.261034921143457, 0.322328869227062, .2807247779692679,
.2018104299334517, .1230600248057767, .0612766372619573,
.0203706503896865, - .0023794587414574, - .0120913057698414,
- 0.013861321214153]
(%i5) plot2d ([discrete, xx, yy]);
(%o5)
(%i6) plot2d ([discrete, xx, yy], [gnuplot_curve_styles, ["with points"]]);
(%o6)
```

See also **plot3d**.
plot3d도 봐주세요. (3차원 도표 작성 함수입니다.)

7.5 Save and load a file
파일 저장및 불러오기

save writes expressions to a file.
save는 표현식을 파일에 저장하는 데 씁니다.

```
(%i1) a: foo - bar;
(%o1)                          foo - bar
(%i2) b: foo^2 * bar;
                                      2
(%o2)                          bar foo
(%i3) save ("my.session", a, b);
(%o3)                         my.session
(%i4) save ("my.session", all);
(%o4)                         my.session
```

load reads expressions from a file.
load는 파일에 저장된 표현식을 불러오는 데 씁니다.

```
(%i1) load ("my.session");
(%o4)                         my.session
(%i5) a;
(%o5)                          foo - bar
(%i6) b;
                                      2
(%o6)                          bar foo
```

See also **stringout** and **batch**.
stringout랑 **batch**도 참조 하시고요.

8 Maxima programming
맥시마 프로그래밍

There is one namespace, which contains all Maxima symbols. There is no way to create another namespace.
맥시마 심볼을 저장하는 네임스페이스는 하나만 존재합니다. 다른 네임스페이스를 만드는 건 안됩니다.

All variables are global unless they appear in a declaration of local variables. Functions, lambda expressions, and blocks can have local variables.
변수는 지역[10] 변수로 선언되지 않는 한 모두 글로발[11] 입니다. 함수, 'lambda' 식, 'block'은 지역 변수를 가질 수 있습니다.

The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable in a block, function, or lambda expression. This policy is known as *dynamic scope*.
변수 값은 최근에 지정(대입)된 값을 가지고요. 보통 대입이나 함수, 'lambda' 식, 'block'에서는 지역변수값에 대입하는 걸로 말이지요. *dynamic scope* 정책입니다.

If a variable is a local variable in a function, lambda expression, or block, its value is local but its other properties (as established by **declare**) are global. The function **local** makes a variable local with respect to all properties.
함수, 'lambda'식, 'block'의 지역 변수는 그 값은 지역적(local)이지만 그 속성은 **declare**로 된 것처럼 전역적(global)입니다.

By default a function definition is global, even if it appears inside a function, lambda expression, or block. **local**$(f), f(x) := \ldots$ creates a local function definition.
함수정의는 함수, 'lambda'식, 'block'안에 들어 있어도, 절로 전역적(global)입니다. **local**$(f), f(x) := \ldots$처럼 로칼로 선언해야 로칼이 되는 거지요.

trace(*foo*) causes Maxima to print an message when the function *foo* is entered and exited.
trace(*foo*)는 함수 *foo*에 들락거릴 때마다 메시지를 프린트하게 합니다.

Let's see some examples of Maxima programming.

[10] 역자 주: 지역, 지역적, 로칼 등이 같은 의미로 쓰였습니다.
[11] 역자 주: 글로발, 전역적, 전역 등이 같은 의미로 쓰였습니다. 고치려고 했는 데, 한글은 검색이 잘 안되어서...

예를 좀 들어봅시다.

1. All variables are global unless they appear in a declaration of local variables. Functions, lambda expressions, and blocks can have local variables.
   변수는 지역 변수로 선언되지 않는 한 모두 글로발이고요, 함수, 'lambda' 식, 'block'은 지역 변수를 가질 수 있습니다.

```
(%i1) (x: 42, y: 1729, z: foo*bar);
(%o1)                          bar foo
(%i2) f (x, y) := x*y*z;
(%o2)                     f(x, y) := x y z
(%i3) f (aa, bb);
(%o3)                        aa bar bb foo
(%i4) lambda ([x, z], (x - z)/y);
                                        x - z
(%o4)                  lambda([x, z], -----)
                                         y
(%i5) apply (%, [uu, vv]);
                               uu - vv
(%o5)                          -------
                                1729
(%i6) block ([y, z], y: 65536, [x, y, z]);
(%o6)                       [42, 65536, z]
```

2. The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable. 변수값은 최근에 지정(대입)된 값을 가지고요. 보통 대입이나 지역변수값에 대입하는 걸로 말이지요.

```
(%i1) foo (y) := x - y;
(%o1)                       foo(y) := x - y
(%i2) x: 1729;
(%o2)                            1729
(%i3) foo (%pi);
(%o3)                          1729 - %pi
(%i4) bar (x) := foo (%e);
(%o4)                      bar(x) := foo(%e)
(%i5) bar (42);
(%o5)                            42 - %e
(%i6) bar ('u);
(%o6)                            u - %e
```

9 Lisp and Maxima
리스프(리스트 아님)랑 맥시마

The construct **:lisp** *expr* tells the Lisp interpreter to evaluate *expr*. This construct is recognized at the input prompt and in files processed by **batch**, but not by **load**.
'construct' **:lisp** *expr*는 리스프번역기가 *expr*를 평가하게 합니다. 입력 프롭프트나 **batch**로 진행할 때 인식되고, **load**할 때는 모릅니다.

The Maxima symbol **foo** corresponds to the Lisp symbol $foo, and the Lisp symbol foo corresponds to the Maxima symbol ?**foo**.
맥시마 심볼 **foo**는 리스프 심볼 $foo에 해당하고, 리스프 심볼 foo는 맥시마 심볼 ?**foo**이 됩니다.

:lisp (**defun** $foo (a) (...)) defines a Lisp function foo which evaluates its arguments. From Maxima, the function is called as **foo**(a).
:lisp (**defun** $foo (a) (...))는 리스프 함수 foo를 정의하고 그 함수 인자를 평가합니다. 맥시마에서는 **foo**(a)로 호출하고요.

:lisp (**defmspec** $foo (e) (...)) defines a Lisp function **foo** which quotes its arguments. From Maxima, the function is called as **foo**(a). The arguments of $foo are (**cdr** e), and (**caar** e) is always $foo itself.
:lisp (**defmspec** $foo (e) (...))는 리스프 함수 함수 인자를 인용하는 **foo**를 정의합니다. 맥시마에서는 **foo**(a)로 호출하고요. 함수 $foo의 인자는 (**cdr** e) 랑함수 $foo 자체를 나타내는 (**caar** e)가 됩니다.

From Lisp, the construct (**mfuncall** '$foo $a_1 \ldots a_n$) calls the function **foo** defined in Maxima.
리스프에서는 (**mfuncall** '$foo $a_1 \ldots a_n$)와 같은 'construct'로 맥시마 함수 **foo**를 부를 수 있습니다.

Let's reach into Lisp from Maxima and vice versa.
예를 좀 보지요.

1. The construct **:lisp** *expr* tells the Lisp interpreter to evaluate *expr*.
   'construct' **:lisp** *expr*는 리스프번역기가 *expr*를 평가하게 합니다.

```
(%i1) (aa + bb)^2;
                                        2
(%o1)                          (bb + aa)
(%i2) :lisp $%
((MEXPT SIMP) ((MPLUS SIMP) $AA $BB) 2)
```

2. **:lisp** (**defun** $foo (a) (...)) defines a Lisp function **foo** which evaluates its arguments.

:lisp (**defun** $foo (a) (...))는 리스프 함수 foo를 정의하고 그 함수 인자를 평가합니다.

```
(%i1) :lisp (defun $foo (a b) `((mplus) ((mtimes) ,a ,b) $%pi))
$FOO
(%i1) (p: x + y, q: x - y);
(%o1)                          x - y
(%i2) foo (p, q);
(%o2)                 (x - y) (y + x) + %pi
```

3. **:lisp** (**defmspec** $foo (e) (...)) defines a Lisp function **foo** which quotes its arguments.
:lisp (**defmspec** $foo (e) (...))는 리스프 함수 함수 인자를 인용하는 **foo**를 정의합니다.

```
(%i1) to_lisp ();
Type (to-maxima) to restart, ($quit) to quit Maxima.
MAXIMA> (defmspec $bar (e) (let ((a (cdr e)))
          `((mplus) ((mtimes) ,@a) $%pi)))
#<FUNCTION (LAMBDA (E)) {B59873D}>
MAXIMA> (to-maxima)
Returning to Maxima
(%o1)                          true
(%i2) (p: x + y, q: x - y);
(%o2)                          x - y
(%i3) bar (p, q);
(%o3)                       p q + %pi
(%i4) bar (''p, ''q);
(%o4)                 (x - y) (y + x) + %pi
```

4. From Lisp, the construct (**mfuncall** '$foo $a_1 \ldots a_n$) calls the function **foo** defined in Maxima.
리스프에서는 (**mfuncall** '$foo $a_1 \ldots a_n$)와 같은 'construct'로 맥시마 함수 **foo**를 부를 수 있습니다.

```
(%i1) blurf (x) := x^2;
                                        2
(%o1)                     blurf(x) := x
(%i2) :lisp (displa (mfuncall '$blurf '((mplus) $grotz $mumble)))
                2
(mumble + grotz)
NIL
```

The below is not the integral part of the original document.

p.s. by Sajang Yang: I found Maxima just a couple of days ago, and I was immediately attracted to it. I have not even run any Maxima software such as wxMaxima or XMaxima or just plain old command-line Maxima yet, but I would like to translate some documents first so that I can introduce this interesting software, and I think that it's good idea to have free software Computer Algebra System(CAS). If you want to help improving this translation, please let me know through email: Sajang.Yang (at) gmail (dot) com. This TeX document is compiled with TexLive 2009, kotex package.

역자 후기: 하루이틀 전에 우연히 맥시마를 알게 되고, 뻑 갔습니다. 아직 맥시마 소프트웨어를 제대로 시작해보지도 않았지만 번역을 해서 다른 분들께도 착한 소프트웨어를 소개하고 싶은 기특한 마음으로 어설프게 번역하였습니다. 맥시마는 자유 소프트웨어 컴퓨터 앨지브라 시스템입니다. 번역에 불만이 있으신 분들은 Sajang.Yang (at) gmail (dot) com으로 이멜 날려 주세요. 이 TeX문서는 kotex 패키지와 함께 TeXLive 2009으로 작성하였습니다.