wp

```
double wp(continuous):
```

The wp function returns the current value that the **witness point** gives to the expression passed as a parameter. The parameter to this function must be an affine expression of type continuous.

string

The keyword string is used to create storage for character strings. Example:

```
string s,t,u[12];
```

subject_to

This keyword is used with the **find_min**, and **find_max** keywords. Example:

```
find_min: Z;
subject_to
      search();
```

then

This keyword is used as part of an if/then/else statement.

See also **Implication and Conditional Statements**.

ub

double ub(continuous)

This function returns the current upper bound of the continuos variable passed to it as a parameter. The current upper bound is the upper bound that the simplex solver has for this variable. Example:

```
check_bound(continuous R, int dir, double limit)
{
      if dir == 1;
      then ub(R) >= limit;
      else lb(R) <= limit;
}
```

See also **lb**.

void

The void keyword is used to declare external functions that are called from a 2LP model and interpreted as procedures that always succeed. Example:

extern void insert_states(int);

```
if k==0
then {
      a = 15;
      return;
}
```

seed

```
void seed(int)
```

A call to this procedure seeds the built-in pseudo-random number generator. Pseudo-random sequences are created by calls to **random.**

sigma

This keyword is for writing a constraint in a notation similar to the ordinary mathematical usage for the equation

For example, the 2LP expression is

```
sigma(int i=0;i<100;i=i+1) a[i]*X[i] == b;
```

or

This keyword is used after an either statement or to code an or loop. For example:

```
or(int i=0;i<N;i++)
      X[j] == i;
```

See also **Persistent Disjunction**.

printf

This keyword has the same syntax as in C. It supports the ANSI C standard format options as given in The C Programming Language, B. Kernighan and D. Ritchie, Prentice Hall.

random

```
double random()
```

This built-in function returns a floating point number in the interval (0,1). Successive calls produce a pseudo-random sequence. The underlying pseudo-random number generator is seeded by a call to **seed**

range

```
double range ( continuous );
```

This built-in function returns an estimate of the amount the bound on a variable can be altered without changing its reduced cost. This estimate will be exact or conservative; over this range the reduced cost is equal to the shadow price.

rc

```
double rc( continuous );
```

This built-in function returns the reduced cost of a continuous variable.

return

This statement exits the current **procedure** and returns to the calling procedure. Example:

nint

```
double nint( double )
```

This function, when called with a double returns with the integer value nearest to itself. When it is called with a continuous variable, it returns the nearest integer to its **witness point**.

not

`not` *<statement>*

The keyword not followed by statement succeeds if *<statement>* fails and conversely, the negation fails if *<statement>* succeeds. The **feasible region** is restored to its state preceding the execution of the construct regardless of its success or failure. Examples:

```
if not X <= Y;
then printf("The constraint would fail\n");
else printf("The constraint would succeed\n");

if not p();
then printf("A call to p() can be successful\n");
else printf("A call to p() can not be successful\n");
```

objcdn

```
double objcdn( continuous )
```

This function returns a true or conservative estimate of how much the coefficient of the continuous variable can be decreased without requiring a change in the **witness point**. that yields the optimal solution.

See also **objcup.**

objcup

```
double objcup( continuous )
```

This function returns a true or conservative estimate of how much the coefficient of the continuous variable can be increased without requiring a change in the **witness point** that yields the optimal solution.

See also **objcdown**.

```
integral( continuous )
```

The integral function succeeds if the parameter is integer valued to within . Example:

```
no_injuries()
{
      and(int i=0;i<N;i++)
            integral(X[i]);
}
```

lb

```
double lb( continuous )
```

This function returns the current lower bound of the continuous variable passed to it as a parameter. The current lower bound is the lower bound that the simplex solver has for this variable. Example:

```
check_bound(continuous R, int dir, double limit)
{
      if dir == 1;
      then ub(R) >= limit;
      else lb(R) <= limit;
}
```

See also **ub**.

max

`max:` <*affine expression*>

This operator moves the **witness point** to a vertex of the feasible region for which the **affine expression** has the maximum possible value.

See also **min.**

min

`min:` <*affine expression*>

This operator moves the **witness point** to a vertex of the feasible region for which the **affine expression** has the minimum possible value.

See also **max**.

```
subject_to
      new_central_loop();
```

find_min

The find_min keyword is used with the **subject_to** keyword to use a branch and bound search to find the minimum value of a variable or expression (the objective function) subject to the conditions in the subject_to block. This construct forces the code to constrain the objective function to be as good as the previous solutions found. The find_min construct fails if the statement following the subject_to fails. At exit the find_min construct shrinks the **feasible region** to a single point, namely the best solution that was found. Example:

```
find_min: Cost; subject_to
      and(int i=0;i<ITEMS;i++)
            Serving[i] == k;
```

floor

```
double floor( double )
```

This function, when called with a double returns with the nearest integer less than or equal to itself. When it is called with a continuous variable, it returns the nearest integer less than or equal to its **witness point**.

if

The keyword for the first part of an if/then/else statement.

See also **Implication and Conditional Statements**.

int

Used to create storage for integer variables. Example:

```
int i,j,k[40];
```

See also **Types**.

integral

This keyword is used to tell the 2LP system that the variables or **procedures** that follow are actually declared in an external DLL. This keyword must be followed by one of the following types: **int**, **double**, **boolean**, or **void**. Example:

```
extern int wait_list[QUEUESIZE][JOBS];
extern boolean goal(int);
```

fabs

```
double fabs(double)
```

This function, when called with a double returns the absolute value of the variable. When it is called with a continuous variable, it returns the absolute value of the **witness point** of that variable. Example:

```
d = fabs(wp(X));
```

find_all

The find_all keyword must be followed by a statement:

find_all <statement>

After *<statement>* succeeds, backtracking takes place and the next solution to *<statement>* is sought. This process continues until all solutions have been found. When no more solutions are possible, the find_all exits successfully without having changed the original **feasible region**.. However, if *<statement>* itself never succeeds, then the find_all *<statement>* also fails. Example:

```
find_all {
      solve_salt_and_mustard();
      count = count + 1;
}
```

find_max

The find_max keyword is used with the **subject_to** keyword to use a branch and bound search to find the maximum value of a variable or expression (the objective function) subject to the conditions in the subject_to block. This construct forces the code to constrain the objective function to be as good as the previous solutions found. The find_max construct fails if the statement following the subject_to fails. At exit the find_max construct shrinks the **feasible region** to a single point, namely the best solution that was found. Example:

```
find_max: Return;
```

continuous

The continuous keyword is used to create storage for continuous variables. Example:

```
continuous X,Y,Z[10];
```

See also **Types**.

double

The double keyword is used to create storage for double variables. Example:

```
double x,y,a[40];
```

See also **Types**.

else

This keyword indicates the beginning of the else part of an if/then/else statement.

See also **Implication and Conditional Statements.**

either

This keyword is used to indicate persistent disjunction. The either construct is followed by one or more or constructs.

See also **Persistent Disjunction**.

exit

exit()

This procedure terminates the 2LP code.

extern

boolean

The boolean keyword is used to create external functions that either succeed or fail. The external function should return an integer. For example:

```
extern boolean start_empty_job(int,int);
```

break

The break statement is used to break out of either an **and**, an **or** loop or a **c_or** loop. If it is used in an and loop, the and loop succeeds. If it is used in an or loop or c_or loop, the loop fails and returns without having created any choice points.

c_either

This keyword indicates that the disjunction is conditional as in C/C++.

See also **Conditional Disjunction**.

c_or

This keyword follows a **c_either** statement when using a conditional disjunction or begins a **c_or** loop construct.

See also **Conditional Disjunction.**

ceil

```
double ceil(double)
```

This function, when called with a double returns with next highest integer value. When it is called with a continuous variable, it returns the next highest integer value of the **witness point** of that variable.

continue

This keyword is used in the body of **and**, **or** and **c_or** loops. In an and loop the body is exited successfully and control returns to the head of the loop. In an or loop the body is exited unsuccessfully and control is returned to the head of the loop; no choice points that were created in the body of the loop are kept.

Keywords and Terms.

2lp_main
absgap
and
boolean
break
c_either
c_or
ceil
continue
continuous
double
else
either
exit
extern
fabs
integral
lb
max
min
nint
not
objcdn
objcup
or
printf
random
range
rc
return
seed
sigma

2lp_main()

This keyword must be the **procedure** name for the main procedure.

absgap

```
absgap(double)
```

The absgap procedure establishes a gap between the last solution found in a **find_max** or **find_min** block. Example:

```
find_max: Z;
subject_to {
      absgap(1.0);
      search();
}
```

and

This keyword used to code an `and` loop. For example

```
and(int i=0;i<N;i++)
```

See also **Conjunction**.