

IBM ILOG CPLEX Optimization Studio OPL Language Reference Manual

Version 12 Release 7

Copyright notice

Describes general use restrictions and trademarks related to this document and the software described in this document.

© Copyright IBM Corp. 1987, 2017

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Other company, product, or service names may be trademarks or service marks of others.

© Copyright IBM Corporation 1987, 2017.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures	v
--------------------------	----------

Tables	vii
-------------------------	------------

Language Reference Manual	ix
--	-----------

Chapter 1. Why an Optimization Programming Language?	1
---	----------

Chapter 2. OPL, the modeling language	3
--	----------

Building a model	3
Data types	4
Basic data types	4
Data structures	7
Data sources	15
Data initialization	15
Spreadsheet Input/Output	24
Data consistency.	27
Preprocessing of data	30
Pitfall of lazy initialization of data.	31
Decision types	32
Decision variables	32
Expressions of decision variables	34
Dynamic collection of elements into arrays	34
Expressions	36
Usage of expressions	36
Data and decision variable identifiers.	36
Integer and float expressions	37
Aggregate operators	38
Piecewise-linear functions	39
Discontinuous piecewise-linear functions	40
Set expressions	43
Boolean expressions	45
Constraints	46
Introduction	46
Using constraints	46
Constraint labels.	48
Types of constraints	53

Limitations on constraints	59
Formal parameters	59
Basic formal parameters	59
Tuples of parameters	61
Filtering in tuples of parameters	62
Scheduling	63
Introduction	63
Piecewise linear and stepwise functions	64
Interval variables	66
Unary constraints on interval variables	69
Precedence constraints between interval variables	70
Constraints on groups of interval variables	70
A logical constraint between interval variables: presenceOf	71
Expressions on interval variables	72
Sequencing of interval variables	73
Cumulative functions	77
State functions	81
Notation	84

Chapter 3. IBM ILOG Script for OPL	87
---	-----------

Language structure	87
Syntax	87
Expressions in IBM ILOG Script	88
Statements.	94
Built-in values and functions	99
Numbers	99
IBM ILOG Script strings.	106
IBM ILOG Script Booleans	113
IBM ILOG Script arrays	115
Objects	120
Dates	122
The null value	126
The undefined value	126
IBM ILOG Script functions.	127
Miscellaneous functions	128

Index	131
------------------------	------------

Figures

- | | | | | | |
|----|---|----|----|------------------------------|----|
| 1. | A Piecewise-linear function. | 39 | 3. | Discontinuous costs. | 43 |
| 2. | The discontinuous piecewise-linear function
sign() | 41 | 4. | Labeled assert. | 51 |

Tables

1. Escape sequences inside literal strings	5	23. Escape sequences in strings	106
2. Operations allowed on sets	12	24. Examples of string literals using escape sequences	107
3. Initializing sets in the model file	22	25. String property	108
4. A discontinuous cost function	42	26. String methods	108
5. Functions over sets	44	27. String functions	110
6. Reference syntax of script variables	89	28. String operators	112
7. Property access syntax	90	29. Boolean method	114
8. Assignment operator syntax	90	30. Logical operators	114
9. Syntactic shorthand	91	31. Array constructor	116
10. Function call syntax	91	32. Array properties	117
11. Special keyword syntax	92	33. Array methods	118
12. Special operator syntax	92	34. Object constructor	121
13. Other operator syntax	94	35. User-defined constructors	121
14. Conditional statement syntax	95	36. Built-in method	121
15. Loop syntax	95	37. Date constructor	122
16. Declaration syntax for script variables	97	38. Date methods	124
17. Function definition syntax	98	39. Date functions	125
18. Default value syntax	99	40. Methods of null	126
19. Number method	102	41. Methods of undefined	127
20. Numeric functions	102	42. Function methods	127
21. Numeric constants	103	43. Miscellaneous functions	128
22. Numeric operators	104		

Language Reference Manual

This manual provides reference information about IBM® ILOG® Optimization Programming Language (OPL), the modeling language used in CPLEX® Studio. For details of prerequisites, naming conventions, and other general information, see *How to use the documentation* in the Starting Kit.

Chapter 1. Why an Optimization Programming Language?

OPL is a modeling language for combinatorial optimization that aims at simplifying the solving of optimization problems.

Linear programming, integer programming, and combinatorial optimization problems arise in a variety of application areas, which include planning, scheduling, sequencing, resource allocation, design, and configuration.

In this context, OPL is a modeling language for combinatorial optimization that aims at simplifying the solving of these optimization problems. As such, it provides support in the form of computer equivalents for modeling linear, quadratic, and integer programs, and provides access to state-of-the-art algorithms for linear programming, mathematical integer programming, and quadratic programming.

Within the IBM ILOG OPL product, OPL as a modeling language has been redesigned to better accommodate IBM ILOG Script, its associated script language.

Chapter 2. OPL, the modeling language

Presents the modeling language of IBM ILOG OPL, namely: the overall structure of OPL models; the basic modeling concepts; how data can be initialized internally as it is declared or externally in a .dat file; how to connect to, read from, and write to databases and spreadsheets; expressions and relations; constraints; and formal parameters.

Building a model

Describes the primary elements of the OPL language and how they are used to build an optimization model.

The basic building blocks of OPL are integers, floating-point numbers, identifiers, strings, and the keywords of the language. Identifiers in OPL start with a letter or the underscore character (`_`) and can contain only letters, digits, and the underscore character. Note that letters in OPL are case-sensitive. Integers are sequences of digits, possibly prefixed by a minus sign. Floats can be described in decimal notation (3.4 or -2.5) or in scientific notation (3.5e-3 or -3.4e10).

The OPL reserved words are listed in OPL keywords.

Comments in OPL are written in between `/*` and `*/` as in:

```
/*  
This is a  
multiline comment  
*/
```

The characters `//` also start a comment that terminates at the end of the line on which they occur as in:

```
dvar int cost in 0..maxCost; // decision variable
```

An OPL model consists of:

- a sequence of declarations
- optional preprocessing instructions
- the model/problem definition
- optional postprocessing instructions
- optional flow control (main block)

Details about these elements are given in the chapters that follow.

A simple model (vol say.mod)

A typical model is shown here.

```
dvar float+ Gas;  
dvar float+ Chloride;  
  
maximize  
    40 * Gas + 50 * Chloride;  
subject to {  
    ctMaxTotal:  
        Gas + Chloride <= 50;
```

```

ctMaxTotal2:
  3 * Gas + 4 * Chloride <= 180;
ctMaxChloride:
  Chloride <= 40;
}

```

In this example, there is only the declarative initial part and the model definition. There is no preprocessing, postprocessing, or flow control.

Data types

Describes basic data types and data structures available for modeling data in OPL.

Basic data types

Describes integers, floats, strings, piecewise linear functions, and stepwise functions in OPL.

Integers

Describes integers (`int`) in OPL.

Shows how to declare integers in the OPL language.

OPL contains the integer constant `maxint`, which represents the largest positive integer available. OPL provides the subset of the integers ranging from `-maxint` to `maxint` as a basic data type.

A declaration of the form

```
int i = 25;
```

declares an integer, `i`, whose value is 25.

The initialization of an integer can be specified by an expression. For instance, the declaration

```
int n = 3;
int size = n*n;
```

initializes `size` as the square of `n`. Expressions are covered in detail in “Expressions” on page 36.

Floats

Describes floats (`float`) in OPL.

Shows how to declare floats in the OPL language.

OPL also provides a subset of the real numbers as the basic data type `float`. The implementation of floats in OPL obeys the IEEE 754 standard for floating-point arithmetic and the data type `float` in OPL uses double-precision floating-point numbers. OPL also has a predefined float constant, `infinity`, to represent the IEEE infinity symbol. Declarations of floats are similar to declarations of integers.

The declaration

```
float f = 3.2;
```

declares a float `f` whose value is 3.2.

The value of the float can be specified by an arbitrary expression.

Strings

Describes strings (string) in OPL.

Shows how to declare strings in the OPL language.

OPL supports a string data type. The excerpt

```
{string} Tasks = {"masonry", "carpentry", "plumbing", "ceiling",  
                  "roofing", "painting", "windows", "facade",  
                  "garden", "moving"};
```

defines and initializes a set of strings. Strings can appear in tuples and index arrays.

Strings are manipulated in the preprocessing phase via ILOG Script. See Chapter 3, “IBM ILOG Script for OPL,” on page 87 for details on the scripting language.

The OPL parser supports the following escape sequences inside literal strings:

Table 1. Escape sequences inside literal strings

Escape sequence	Meaning
\b	backspace
\t	tab
\n	newline
\f	form feed
\r	carriage return
\"	double quote
\\	backslash
\ooo	octal character ooo
\xXX	hexadecimal character XX

To continue a literal string over several lines, you need to escape the new line character:

```
"first line \
```

```
second line"
```

Piecewise linear functions

Describes piecewise linear functions (pwlFunction) in OPL.

Shows how to declare piecewise linear functions in the OPL language.

Piecewise linear functions are typically used to model a known function of time, for instance the cost incurred for completing an activity after a known date.

Note that you must ensure that the array of values $T[i]$ is sorted.

The meanings of the S , T , and V vectors are described in “Piecewise linear and stepwise functions” on page 64, in the *Language Reference Manual*.

Syntax

```

pwlfFunction F = piecewise(i in 1..n){ S[i]->T[i]; S[n+1] } (t0, v0);

pwlfFunction F = piecewise{ V[1]->T[1], ..., V[n]->T[n], V[n+1] };

pwlfFunction F[i in ...] = piecewise (...)[ ... ];

```

Example

```

int n=2;
float objectiveforxequals0=300;
float breakpoint[1..n]=[100,200];
float slope[1..n+1]=[1,2,-3];

int points[1..n+1] = [5, 150, 400];
float values[x in 1..n+1] = piecewise(i in 1..n) {slope[i] -> breakpoint[i]; slope[n+1]}(0,0) points

execute {
    writeln( values );
}

dvar int x;

maximize piecewise(i in 1..n) {slope[i] -> breakpoint[i]; slope[n+1]}(0,objectiveforxequals0) x;

subject to {true;}

```

Piecewise linear functions are covered in detail in “Piecewise linear and stepwise functions” on page 64.

Stepwise functions

Describes stepwise functions (stepFunction) in OPL.

Shows how to declare stepwise functions in the OPL language.

Stepwise linear functions are typically used to model the efficiency of a resource over time. A stepwise function is a special case of piecewise linear function where all slopes are equal to 0 and the domain and image of F are integer.

Note that you must ensure that the array of values T[i] is sorted.

Syntax

```

stepFunction F = stepwise(i in 1..n){ V[i]->T[i]; V[n+1] };

stepFunction F = stepwise{ V[1]->T[1], ..., V[n]->T[n], V[n+1] };

stepFunction F[i in ...] = stepwise (...)[ ... ];

```

Example

A declaration of the form

```

stepFunction f=stepwise {0->3; 2};

assert f(-1)==0;
assert f(3)==2;
assert f(3.1)==2;

```

declares a stepwise function, f.

Example

Another example, declaring the stepwise function F2:


```

stepFunction F2 = stepwise{ 0->0; 100->20; 60->30; 100 };
int ii= F2( 10 );

execute {
    writeln( ii );
    writeln( F2( 25 ) );
}

```

Stepwise functions are covered in detail in “Piecewise linear and stepwise functions” on page 64.

Data structures

Describes how the basic data types can be combined using arrays, tuples, and sets to obtain complex data structures.

Ranges

Integer ranges are fundamental in OPL, since they are often used in arrays and variable declarations, as well as in aggregate operators, queries, and quantifiers.

Declaring ranges

To specify an integer range, you give its lower and upper bounds, as in
`range Rows = 1..10;`

which declares the range value 1..10. The lower and upper bounds can also be given by expressions, as in

```

int n = 8;
range Rows = n+1..2*n+1;

```

Once a range has been defined, you can use it as an array indexer.

Whenever a range is empty, i.e. its upper bound is less than its lower bound, it is automatically normalized to 0..-1 (in other words, all empty ranges are equal).

The range declaration

An integer range is typically used:

- as an array index in an array initialization expression

```

range R = 1..100;
int A[R]; // A is an array of 100 integers

```

- as an iteration range

```

range R = 1..100;
forall(i in R) {
    //element of a loop
    ...
}

```

- as the domain of an integer decision variable

```

dvar int i in R;

```

The range float declaration

A range float data type consists of a couple of float values specifying an interval. It is typically used as the domain of a floating-point decision variable.

```

range float X = 1.0..100.0;
dvar float x in X;

```

Arrays

Describes one-dimensional arrays and multidimensional arrays.

Arrays are fundamental in many applications.

One-dimensional arrays

One-dimensional arrays are the simplest arrays in OPL and vary according to the type of their elements and index sets. A declaration of the form

```
int a[1..4] = [10, 20, 30, 40];
```

declares an array of four integers $a[1], \dots, a[4]$ whose values are 10, 20, 30, and 40. It is of course possible to define arrays of other basic types. For instance, the instructions

```
int a[1..4] = [10, 20, 30, 40];
float f[1..4] = [1.2, 2.3, 3.4, 4.5];
string d[1..2] = ["Monday", "Wednesday"];
```

declare arrays of natural numbers, floats, and strings, respectively.

The index sets of arrays in OPL are very general and can be integer ranges and arbitrary finite sets. In the examples so far, index sets were given explicitly, but it is possible to use a previously defined range, as in

```
range R = 1..4;
int a[R] = [10, 20, 30, 40];
```

The declaration:

```
int a[Days] = [10, 20, 30, 40, 50, 60, 70];
```

describes an array indexed by a set of strings; its elements are $a["Monday"], \dots, a["Sunday"]$.

Arrays can also be indexed by finite sets of arbitrary types. This feature is fundamental in OPL to exploit sparsity in large linear programming applications, as discussed in detail in Exploiting sparsity in the *Language User's Manual*.

For example, the declaration:

```
tuple Edges {
    int orig;
    int dest;
}
{Edge} Edges = {<1,2>, <1,4>, <1,5>};
int a[Edges] = [10,20,30];
```

defines an integer array, a , indexed by a finite set of tuples. Its elements are $a[<1,2>]$, $a[<1,4>]$, and $a[<1,5>]$. Tuples are described in detail in “Tuples” on page 9.

Multidimensional arrays

OPL supports the declaration of multidimensional arrays (see “Data initialization” on page 15 about the ellipsis syntax). For example, the declaration:

```
int a[1..2][1..3] = ...;
```

declares a two-dimensional array, a , indexed by two integer ranges. Indexed sets of different types can of course be combined, as in

```
int a[Days][1..3] = ...;
```

which is a two-dimensional array whose elements are of the form `a[Monday][1]`. It is interesting to contrast multidimensional and one-dimensional arrays of tuples.

Consider the declaration:

```
{string} Warehouses = ...;
{string} Customers = ...;

int transp[Warehouses,Customers] = ...;
```

that declares a two-dimensional array `transp`. This array may represent the units shipped from a warehouse to a customer. In large-scale applications, it is likely that a given warehouse delivers only to a subset of the customers. The array `transp` is thus likely to be sparse, i.e. it will contain many zero values.

The sparsity can be exploited by declarations of the form:

```
{string} Warehouses = ...;
{string} Customers = ...;
tuple Route {
    string w;
    string c;
}

{Route} routes = ...;
int transp[routes] = ... ;
```

This declaration specifies a set, `routes`, that contains only the relevant pairs (warehouse, customer). The array `transp` can then be indexed by this set, exploiting the sparsity present in the application. It should be clear that, for large-scale applications, this approach leads to substantial reductions in memory consumption.

You can initialize arrays by listing its values, as in most of the examples presented so far. See “Initializing arrays” on page 17, “As generic arrays” on page 18, and “As generic indexed arrays” on page 18 for details.

Tuples

Data structures in OPL can be constructed using tuples that cluster together closely related data. This topic describes how to declare tuples, use keys on tuples, and initialize tuples.

Declaring tuples

For example, the declaration:

```
tuple Point {
    int x;
    int y;
};

Point point[i in 1..3] = <i, i+1>;
```

declares a tuple `Point` consisting of two fields `x` and `y` of type integer. Once a tuple type `T` has been declared, tuples, arrays of tuples, sets of tuples of type `T`, tuples of tuples can be declared, as in:

```
Point p = <2,3>;
Point point[i in 1..3] = <i, i+1>;
{Point} points = {<1,2>, <2,3>;}
```

```

tuple Rectangle {
    Point ll;
    Point ur;
}

```

These declarations respectively declare a point, an array of three points, a set of two points, and a tuple type where the fields are points. The various fields of a tuple can be accessed in the traditional way by suffixing the tuple name with a dot and the field name, as in

```

Point p = <2,3>;
int x = p.x;

```

which initializes `x` to the field `x` of tuple `p`. Note that the field names are local to the scope of the tuples.

Note:

Multidimensional arrays are not supported in tuples.

Keys in tuple declarations

As in database systems, tuple structures can be associated with *keys*. Tuple keys enable you to access data organized in tuples using a set of unique identifiers. In the following example, the nurse tuple is declared with the key name of type string.

Declaring a tuple using a single key (`nurses.mod`)

```

tuple nurse {
    key string name;
    int seniority;
    int qualification;
    int payRate;
}

```

Then, supposing Isabelle must not work more than 20 hours a week, just write:

```
NurseWorkTime[<"Isabelle">]<=20;
```

leaving out the fields with no keys. This is equivalent to:

```
NurseWorkTime[<"Isabelle",3,1,16>]<=20;
```

Using keys in tuple declarations has practical consequences, in particular:

- The key field can be used as a unique identifier for the tuple, for example the field name in Declaring a tuple using a single key (`nurses.mod`). In this example, it means that there will be no two tuples with the same name in a set of tuples of the type `nurse`. If a user inadvertently attempts to add two different tuples with the same name, OPL raises an error.
- Defining keys enables you to access elements of the tuple set by using only the value of the key field (name in the `nurses` example). Slicing is one of the features that benefit from it: you can slice on the tuple set using only key fields.

You can also declare a tuple using a non singleton set of keys, such as the shift tuple of the `nurses` example below.

Declaring a tuple using a set of keys (`nurses.mod`)

```

tuple shift {
    key string departmentName;
    key string day;
}

```

```

    key int startTime;
    key int endTime;
    int minRequirement;
    int maxRequirement;
}

```

In this example, a shift is uniquely identified by the department name, the date, and start and end times, all defined as key fields.

Initializing tuples

You initialize tuples by giving the list of the values of the various fields, as in:

```
Point p = <2,3>;
```

which initializes `p.x` to 2 and `p.y` to 3. See “Initializing tuples” on page 19 for details.

Limitations on tuples

When using tuples in your models, you should be aware of various limitations.

Data types in tuples

Not all data types are allowed inside tuples. The limitations are given here.

Data types allowed in tuples

- Primitives (int, float, string)
- Tuples (also known as subtuples)
- Arrays with primitive items (not string), that is: integer or float arrays
- Sets with primitive items, that is: integer, float or string sets

Note:

Arrays and sets in tuples are not compared by content. If collections inside a tuple are modified, duplicates are not detected.

Data types not allowed in tuples

- Sets of tuples (instances of `ILoTupleSet`)
- Arrays of strings, tuples, and tuple sets
- Multidimensional arrays

Tuple indices and tuple patterns

You cannot mix tuple indexes and patterns within the declaration and the use of decision expressions. For example, these code lines raise the following error message Data not consistent for "xxx": can not mix pattern and index between declaration of `dexpr` and instantiation.

Do not mix tuple indices and tuple patterns in `dexpr`

```

dexpr float y[i in t] = ...;
subject to {
    forall(<a,b,c> in t) y[<a,b,c>]==...; };

dexpr float y[<a,b,c> in t] = ...;
subject to {
    forall(i in t) y[i]==...;
};

```

Performance and memory consumption

If you choose to label constraints in large models, use tuple indices instead of tuple patterns to avoid increasing the performance and memory cost. See “Constraint labels” on page 48.

Sets

Gives a definition of sets, a list of the operations allowed on sets, and a few words on their initialization.

Definition

Sets are non-indexed collections of elements without duplicates.

OPL supports sets of arbitrary types to model data in applications. If T is a type, then $\{T\}$, or alternatively $\text{setof}(T)$, denotes the type “set of T ”. For example, the declaration:

```
{int} setInt = ...;
setof(Precedence) precedences = ...;
```

declares a set of integers and a set of precedences.

Sets may be ordered, sorted, or reversed. By default, sets are ordered, which means that:

- Their elements are considered in the order in which they have been created.
- Functions and operations applied to ordered sets preserve the order.

See “Sorted and ordered sets” on page 13 for details.

Operations on sets

The following operations are allowed on sets. See OPL functions in *Language Quick Reference* for more information about functions. For the functions on sets, the index starts at 0.

Table 2. Operations allowed on sets

Operations	Syntax
union, inter, diff, symdiff	set = set1 function set2
first, last	elt = function(set)
next, prev	elt = function(set,elt,int)
nextc, prevc	elt = function(set,elt,int)
item	elt = function(set,int)
ord	int = function(set,elt)

Initializing sets

A set can be initialized in various ways. The simplest way is by listing its values explicitly. For example:

```
tuple Precedence {
    int before;
    int after;
}
{Precedence} precedences = {<1,2>, <1,3>, <3,4>};
```

See “Initializing sets” on page 21 for details.

Methods for sets

For advanced users:

The methods for sets can be found in the class `IloDiscreteDataCollection`, in the *ILOG Script Reference Manual*.

Sorted and ordered sets

Shows the effect of the sorted property on simple sets, tuple sets, and sets used in piecewise linear functions.

Definitions

- An ordered set is a set whose elements are arranged in the order in which they were added to the set. Note that this is how sets are created by default. For example:

```
{int} S1 = {3,2,5};
```

and

```
ordered {int} S1 = {3,2,5};
```

are equivalent.

- A sorted set is a set in which elements are arranged in their natural, ascending (or descending) order. For strings, the natural order is the lexicographic order. The natural order also depends on the system locale. For example:

```
sorted {int} sortedS = {3,2,5};
```

and

```
ordered {int} orderedS = {2,3,5};
```

are equivalent, and iterating over `sortedS` or `orderedS` will have the same behavior.

To specify the descending order, you add the keyword `reversed`.

Simple sets

The following code sample enables you to see the difference between the union of ordered sets and the union of sorted sets.

```
{int} s1 = {3,5,1};
{int} s2 = {4,2};
{int} orderedS = s1 union s2;
sorted {int} sortedS = s1 union s2;
execute{
    writeln("ordered union = ", orderedS);
    writeln("sorted union = ", sortedS);
}
```

The statement

```
{int} orderedS = s1 union s2;
```

returns

```
ordered union = {3 5 1 4 2}
```

while the statement

```
sorted {int} sortedS = s1 union s2;
```

returns

```
sorted union = {1 2 3 4 5}
```

Sorted tuple sets

When a tuple set does not use keys, the entire tuple, except set and array fields, is taken into account for sorting. For tuple sets with keys, sorting takes place on all keys in their order of declaration. In other words, it is not possible to sort a tuple set on one (or more) given column(s) only.

The following code extract declares a team of people who are defined by their first name, last name, and nickname, then prints the list of team members first in the creation order, then in alphabetical order.

```
tuple person {
    string firstname;
    string lastname;
    string nickname;
}
tuple personKeys {
    key string firstname;
    key string lastname;
    string nickname;
}
{person} devTeam = {
<"David", "Atkinson", "Dave">,
<"David", "Doe", "Skinner">,
<"Gregory", "Simons", "Greg">,
<"David", "Smith", "Lewis">,
<"Kevin", "Morgan", "Kev">,
<"Gregory", "McNamara ", "Mac">
};
sorted {personKeys} sortedDevTeam = {<i,j,k> | <i,j,k> in devTeam};
execute{
    writeln(devTeam);
    writeln(sortedDevTeam);
}
```

The person tuple uses no keys.

```
tuple person {
    string firstname;
    string lastname;
    string nickname;
}
```

The personKeys tuple uses keys for the first and last names, not for the nickname.

```
tuple personKeys {
    key string firstname;
    key string lastname;
    string nickname;
}
```

The data shows that the team includes three people whose first name is David, two people whose first name is Gregory, and one person whose first name is Kevin.

As a consequence, the statement

```
sorted {personKeys} sortedDevTeam = {<i,j,k> | <i,j,k> in devTeam};
```

lists the David tuples before the Gregory tuples, which themselves appear before the Kevin tuple. Within the David tuples, "David" "Doe" "Skinner" comes before "David" "Smith" "Lewis" because a second sorting also takes place on the second

field with the key `lastname`. In contrast, since there is no person with the same first name and last name, no sort is ever done on the last field `nickname`.

The output of `sortedDevTeam` is displayed in the CPLEX Studio IDE as:

```
{<"David" "Atkinson" "Dave"> <"David" "Doe" "Skinner">
<"David" "Smith" "Lewis"> <"Gregory" "McNamara " "Mac">
<"Gregory" "Simons" "Greg"> <"Kevin" "Morgan" "Kev">}
```

Sorted sets in piecewise linear functions

In piecewise linear functions, breakpoints must be strictly increasing. However, in most cases, the data supplied by a database or a `.dat` file is not sorted in an increasing numeric or lexicographic order. As a consequence, you have to add complex and verbose scripting statements to sort the data.

To avoid these extra code lines, the `sorted` property of sets enables you to sort data by specifying a single keyword, as shown in the following code extract. Writing piecewise linear functions becomes easier, as one code line is sufficient instead of several dozens.

```
tuple Cost{
    key int BreakPoint;
    float Slope;
}
sorted {Cost} sS = { <1, 1.5>, <0, 2.5>, <3, 4.5>, <2, 4.5>};

float lastSlope = 3.5;

dvar float+ x;
minimize piecewise(t in sS)
    {t.Slope -> t.BreakPoint; lastSlope} x;
```

See also “Piecewise-linear functions” on page 39.

Other related topics:

“Data sources”

Describes data initialization, spreadsheet input/output, data consistency, and preprocessing.

Describes data initialization.

Introduction to scripting

Defines IBM ILOG Script as a scripting language and describes the situations in which it is used: preprocessing, postprocessing, and flow control. Also provides programming tips and warns you of pitfalls to avoid.

Describes how to set declarations.

Data sources

Describes data initialization, spreadsheet input/output, data consistency, and preprocessing.

Data initialization

Defines internal versus external initialization, describes how to initialize arrays, tuples, and sets, and discusses memory allocation aspects of data initialization.

Internal and external initialization

Defines these two kinds of data initialization.

In OPL, you can initialize data internally or externally. Your choice affects memory allocation. See “Initialization and memory allocation” on page 23 for details.

Internally

This initialization mode consists in initializing the data in the model file at the same time as it is declared. Inline initializations can contain expressions to initialize data items, such as

```
int a[1..5] = [b+1, b+2, b+3, b+4, b+5];
```

Note:

If you choose to initialize data within a model file, you will get an error message if you try to access it later by means of a scripting statement such as:

```
myData.myArray_inMod[1] = 2;
```

Externally

This initialization mode consists in specifying initialization subsequently as an OPL statement in a separate .dat file (see OPL Syntax in the *Language Quick Reference*). This includes reading from a spreadsheet as explained in “Spreadsheet Input/Output” on page 24.

You declare external data using the ellipsis syntax. However, data initialization instructions cannot contain expressions, since they are intended to specify data. Data initialization instructions make it possible to specify sets of tuples in very compact ways. Consider these types in a .mod file:

```
{string} Product = {"flour", "wheat", "sugar"};
{string} City = {"Providence", "Boston", "Mansfield"};
tuple Ship {
  string orig;
  string dest;
  string p;
}
{Ship} shipData = ...;
```

and assume that the set of shipments is initialized externally in a separate .dat file like this:

```
shipData =
{
  <"Providence", "Boston", "wheat">
  <"Providence", "Boston", "flour">
  <"Providence", "Boston", "sugar">
  <"Providence", "Boston", "wheat">
  <"Providence", "Mansfield", "wheat">
  <"Providence", "Mansfield", "flour">
  <"Boston", "Providence", "sugar">
  <"Boston", "Providence", "flour">
};
```

Note:

In .dat files, the separating comma is optional. For strings without any special characters, even the enclosing quotes are optional.

Initializing arrays

Describes the various ways in which you can initialize arrays.

You can initialize arrays:

- “Externally”
- “Internally” on page 18
- “In preprocessing instructions” on page 18
- “As generic arrays” on page 18
- “As generic indexed arrays” on page 18

Externally

Arrays can be initialized by external data, in which case the declaration has the form:

```
int a[1..2] [1..3] = ...;
```

and the actual initialization is given in a data source, which is a separate .dat file in IBM ILOG OPL.

- **By listing values**

This is how arrays are initialized in most of the examples presented so far. Multidimensional arrays in OPL are, in fact, arrays of arrays and must be initialized accordingly. For example, the declaration:

```
/* .mod file */
int a[1..2][1..3] = ...;
/* .dat file */
a = [
  [10, 20, 30],
  [40, 50, 60]
];
```

initializes a two-dimensional array by giving initializations for the one-dimensional arrays of its first dimension. It is easy to see how to generalize this initialization to any number of dimensions.

- **By specifying pairs**

An array can also be initialized by specifying pairs (index, value), as in the declaration:

```
/* .mod file */
int a[Days] = ...;
/* .dat file */
a = #[
  "Monday": 1,
  "Tuesday": 2,
  "Wednesday": 3,
  "Thursday": 4,
  "Friday": 5,
  "Saturday": 6,
  "Sunday": 7
]#;
```

Note:

1. When the initialization is specified by (index, value) pairs, the delimiters #[] and]# must be used instead of [and].
2. The ordering of the pairs can be arbitrary.

These two forms of initialization can be combined arbitrarily, as in:

```

/* .mod file */
int a[1..2][1..3] = ...;
/* .dat file */
a = #[
2: [40, 50, 60],
1: [10, 20, 30]
]#;

```

Internally

You can initialize arrays internally (that is, in the .mod file) using the same syntax as in .dat files. Here, the array items may be expressions that are evaluated during initialization. The syntax for pairs #[,]# is not available for internal initialization.

In preprocessing instructions

Arrays can also be initialized in the preprocessing instructions, as in:

```

range R = 1..8;
int a[R];
execute {
  for(var i in R) {
    a[i] = i + 1;
  }
}

```

which initializes the array in such a way that $a[1] = 2$, $a[2] = 3$, and so on. See “Preprocessing of data” on page 30.

As generic arrays

OPL also supports generic arrays, that is, arrays whose items are initialized by an expression. These generic arrays may significantly simplify the modeling of an application. The declaration:

```
int a[i in 1..10] = i+1;
```

declares an array of 10 elements such that the value of $a[i]$ is $i+1$. Generic arrays can of course be multidimensional, as in:

```
int m[i in 0..10][j in 0..10] = 10*i + j;
```

which initializes element $m[i][j]$ to $10*i + j$. Generic arrays are useful in performing some simple transformations. For instance, generic arrays can be used to transpose matrices in a simple way, as in:

```

int m[Dim1][Dim2] = ...;
int t[j in Dim2][i in Dim1] = m[i][j];

```

More generally speaking, generic arrays can be used to permute the indices of arrays in simple ways.

As generic indexed arrays

To have more flexibility when initializing arrays in a generic way, OPL enables you to control the **index** value in addition to the **item** value, as described earlier in *As generic arrays*. To illustrate the syntax, the same examples can be expressed as follows:

```

int a[1..10] = [ i-1 : i | i in 2..11 ];
int m[0..10][0..10] = [ i : [ j : 10*i+j ] | i,j in 0..10 ];

```

This syntax is close to the syntax used for initializing arrays in .dat files by means of indices, delimited by #[and] #, as explained above in *Specifying pairs*. Using this syntax is an efficient means of initializing arrays used to index data.

The oilDB.mod example contains an execute block that performs initialization. Instead of:

```
GasType gas[Gasolines];
execute {
  for(var g in gasData) {
    gas[g.name] = g
  }
}
```

the same can be expressed with the syntax for generic indexed arrays as:

```
GasType gas[Gasolines] = [ g.name : g | g in gasData ];
```

Likewise, this syntax, from transp4.mod

```
float Cost[Routes];
execute INITIALIZE {
  for( var t in TableRoutes ) {
    Cost[Routes.get(t.p,Connections.get(t.o,t.d))] = t.cost;
  }
}
```

is equivalent to

```
float Cost[Routes] = [ <t.p,<t.o,t.d>>:t.cost | t in TableRoutes ];
```

Note:

1. It is recommended to use generic arrays or generic indexed arrays whenever possible, since they make the model more explicit and readable.
2. If an index is met more than once, no warning is issued and the latest value set for this index is the one kept.

For example:

```
int n=5;

{int} s= {1,3,4,2,5};
sorted {int} s2=asSet(1..n);;
reversed {int} s3=asSet(1..n);;

int x[1..n]=[max1(n-i,i): i | i in s];
int x2[1..n]=[max1(n-i,i): i | i in s2];
int x3[1..n]=[max1(n-i,i): i | i in s3];

execute
{
  writeln(x);
  writeln(x2);
  writeln(x3);
}

gives out
[0 0 2 4 5]
[0 0 3 4 5]
[0 0 2 1 5]
```

Initializing tuples

Describes the two ways of initializing tuples.

You initialize tuples either by giving the list of the values of the various fields (see “Tuples” on page 9) or by listing the fields and values. For example:

In the .mod file, you write:

```
tuple point
{
  int x;
  int y;
}
point p1=...;
point p2=...;
```

In the .dat file, you write:

```
p1=#<y:1,x:2>#;
p2=<2,1>;
```

As with arrays, the delimiters < and > are replaced by #< and ># and the ordering of the pairs is not important. OPL checks whether all fields are initialized exactly once.

The type of the fields can be arbitrary and the fields can contain arrays and sets.

Example 1: tuple Rectangle

For example, the following code lines declare a tuple with three fields: the first is an integer and the other two are arrays of two points.

```
tuple Rectangle {
    int id;
    int x[1..2];
    int y[1..2];
}
```

```
Rectangle r = ...;
```

```
execute
{
  writeln(r);
}
```

A specific “rectangle” can be declared in the data file as:

```
r=<1, [0,10], [0,10]>;
```

Example 2: tuple Precedence

The declaration

```
tuple Precedence {
    string name;
    {string} after;
}
```

defines a tuple in which the first field is a set item and the second field is a set of values. A possible precedence can be declared as follows:

```
Precedence p = <a1, {a2, a3, a4, a5}>;
```

assuming that a1,...,a5 are strings.

You can also initialize tuples internally within the .mod file. If you choose to do so, you cannot use the named tuple component syntax #<, >#, which is supported in .dat files but **not** in .mod files. Components may be expressions and will be evaluated during initialization.

Initializing sets

You can initialize sets in three different ways.

- “Externally”
- “Internally”
- “As generic sets” on page 22

Externally

As stated in “Initializing sets” on page 12, in the *Sets* topic, the simplest way to initialize a set is by listing its values explicitly in the `.dat` file.

For example, the declaration:

```
/* .mod file */
tuple Precedence {
    int before;
    int after;
}
{Precedence} precedences = ...;
/* .dat file */
precedences = {<1,2>, <1,3>, <3,4>};
```

initializes a set of tuples.

Internally

You can also initialize sets internally (in the `.mod` file), more precisely by using set expressions using previously defined sets and operations such as union, intersection, difference, and symmetric difference. The symmetric difference of two sets A and B is

$$(A \cup B) \setminus (A \cap B)$$

described in “Expressions” on page 36.

For example, the declarations:

```
{int} s1 = {1,2,3};
{int} s2 = {1,4,5};
{int} i = s1 inter s2;
{int} j = {1,4,8,10} inter s2;
{int} u = s1 union {5,7,9};
{int} d = s1 diff s2;
{int} sd = s1 symdiff {1,4,5};
```

initialize i to $\{1\}$, u to $\{1,2,3,5,7,9\}$, d to $\{2,3\}$, and sd to $\{2,3,4,5\}$.

It is also possible to initialize a set from a range expression. For example, the declaration:

```
{int} s = asSet(1..10)
```

initializes s to $\{1,2,\dots,10\}$

It is important to point out at this point that sets initialized by ranges are represented explicitly (unlike ranges). As a consequence, a declaration of the form

```
{int} s = asSet(1..100000);
```

creates a set where all the values 1, 2, ..., 100000 are explicitly represented, while the range

```
range s = 1..100000;
```

represents only the bounds explicitly.

Note that when writing the assignment `s2=s1`, you add one element to `s1`, that element is also added to `s2`. If you do not want this, write

```
s1={i | i in s2}
```

For example, compare the statements in the following table.

Table 3. Initializing sets in the model file

If you write	the result is
<pre>{int} s1={1,2}; {int} s2=s1; execute { s2.add(3); writeln(s1); }</pre>	<pre>{1 2 3}</pre>
<pre>{int} s1={1,2}; {int} s2={ i i in s1}; //{int} s2=s1; execute { s2.add(3); writeln(s1); }</pre>	<pre>{1 2}</pre>

As generic sets

OPL supports *generic sets* which have an expressive power similar to relational database queries. For example, the declaration:

```
{int} s = {i | i in 1..10: i mod 3 == 1};
```

initializes `s` with the set `{1,4,7,10}`. A generic set is a conjunction of expressions of the form

```
p in S : condition
```

where `p` is a parameter (or a tuple of parameters), `S` is a range or a finite set, and `condition` is a Boolean expression. These expressions are also used in `forall` statements and aggregate operators and are discussed in detail in “Formal parameters” on page 59.

The declaration:


```

{string} Resources ...;
{string} Tasks ...;
Tasks res[Resources] = ...;
tuple Disjunction {
    {string} first;
    {string} second;
}
{Disjunction} disj = {<i,j> |
    r in Resources, ordered i,j in res[r]
};

```

is a more interesting example, showing a conjunction of expressions, and is explained in detail in “Formal parameters” on page 59. Generic sets are often useful when you transform a data structure (e.g. the data stored in a file) into a data structure more appropriate for stating the model effectively. Consider, for example, the declarations:

```

{string} Nodes ...;
int edges[Nodes][Nodes] = ...;

```

which describe the edges of a graph in terms of a Boolean adjacency matrix. It may be important for the model to use a sparse representation of the edges (because, for instance, edges are used to index an array). The declaration:

```

tuple Edge {
    Nodes o;
    Nodes d;
}
{Edge} setEdges = {<o,d> | o,d in Nodes : edges[o][d]==1};

```

computes this sparse representation using a simple generic set. It is of course possible to define generic arrays of sets. For example, the declaration:

```

{int} a[i in 3..4] = {e | e in 1..10: e mod i == 0};

```

initializes `a[3]` to `{3,6,9}` and `a[4]` to `{4,8}`.

Initialization and memory allocation

Describes how memory is allocated to data initialization.

In OPL, the initialization mode you choose affects memory allocation. Namely, external initialization from a `.dat` file, while enabling a more modular design, may have a significant impact on memory usage.

Internal initialization

Internal data (directly from the model file) is initialized when first used. This is also called “lazy initialization”. Unused internal data elements are not allocated any memory. In other words, internal data is “pulled” from OPL as needed.

Example of lazy initialization

```

int a=2;
int b=2;

int a2=2*a;
int b2=2*b;

execute
{
    a2;

    a++;
    b++;
}

```

```

writeln(a2);
writeln(b2);
}

assert a2==4;
assert b2==6;

```

External initialization

In contrast, data from a data file is initialized while the .dat file is parsed and is allocated memory whether it is used by the model or not. In other words, external data is “pushed” to OPL.

See also “Pitfall of lazy initialization of data” on page 31.

Spreadsheet Input/Output

Describes how to connect an MS Excel spreadsheet, read from it, and write the results to the connected spreadsheet.

The oilsheet example

Explains spreadsheet input and output in the context of an oil spreadsheet.

This section uses the oilSheet example to demonstrate operations with an MS Excel spreadsheet. You can find this example in

```
<Install_dir>/opl/examples/opl/oil
```

where <Install_dir> is your installation directory.

Connection to a spreadsheet

Explains how to connect OPL to a spreadsheet.

The spreadsheet operations in OPL all refer to a spreadsheet connection. The instruction

```

/* .dat file */
SheetConnection sheet("transport.xls");

```

establishes a connection sheet to a spreadsheet named transport.xls. The connection sheet should be viewed as a handle on the spreadsheet. It is possible in OPL to connect to several spreadsheets within the same model.

The instruction SheetConnection takes only one parameter and you do not need to specify the full path to the spreadsheet name. Relative paths are resolved using the current directory of .dat files.

Note that, in this section, we often use the word “spreadsheet” for “spreadsheet connection”.

Reading from a spreadsheet

Explains how to read from a spreadsheet from within OPL.

In OPL, spreadsheet ranges can be read into one- or two-dimensional arrays or sets. For instance, the instructions:

```

/* .mod file */
{string} Gasolines = ...;

tuple GasType {

```

```

float demand;
float price;
float octane;
float lead;
}

GasType gas[Gasolines] = ...;

/* .dat file */
SheetConnection sheet("oilSheet.xls");

Gasolines from SheetRead(sheet,"gas data"!A2:A4");
Gas from SheetRead(sheet,"gas data"!B2:E4");

```

What data can be read from an Excel spreadsheet?

OPL opens a spreadsheet in read-only mode to read data from it.

The types of data elements supported are:

- sets with integers, floats, strings, or tuples;
- scalar integers, floats, or strings;
- arrays with integers, floats, one- or two-dimensional strings, or one-dimensional tuples;
- one- or two-dimensional arrays of simple types: for such arrays, the data must be formatted, that is, it must have the same width/length as the array to be filled. OPL automatically determines whether the data must be read line by line or column by column. When facing a square zone (a two-dimensional array with $[x][x]$ as dimensions), the engine reads the data line by line.

Note:

- Only tuples with integer, float, and string components are supported.
- If the sheet name contains a space, the name should be surrounded by single quotes in the SheetRead instruction. For example:

```
Oil from SheetRead(sheet,"'oil data'!B2:E4");
```
- OPL does not support the R1C1 reference style to specify the range when reading data from an Excel spreadsheet.

Accessing named ranges in Excel

IBM ILOG OPL supports the convention of *names*, which are a word or string of characters used to represent a cell, range of cells, formula, or constant value, and that can be used in other formulas.

Thus you can use easy-to-understand names, such as *Nutrients*, to refer to hard-to-understand ranges, such as B4:J15 or *IncreasedProtein* to refer to a constraint. You can then substitute these names in formulas for the range of cells or constraint.

Excel *names*, or named ranges, can be accessed using the SheetRead command, using the following syntax:

```

SheetConnection sheetData("C:\\ILOG_Files\\myExcelFile.xls");
prods from SheetRead(sheetData,"Product");

```

Note the double separator `\\` in the SheetConnection command.

The SheetRead command is normal, and in this example the Excel name Product replaces the normal syntax of, say, C13:072.

To create named ranges in Excel 2007:

1. Highlight the range of cells you want to name, then click the **Name** box at the left end of the Formula Bar.
2. Type the name you want to assign to this range and press **Enter**.
3. Save the spreadsheet file.

Note: For a list of supported versions of Microsoft Excel, please see the detailed system requirements <http://www-01.ibm.com/support/docview.wss?uid=swg27019100>.

Additional information on named ranges

- Excel automatically updates (expands) a named range when a row is added somewhere within the range. However, one must be careful adding rows at the end of a range as the range does not get automatically updated in that case. It would have to be updated manually.
- OPL allows blank rows in a named range. If you are reading a set of strings, it will consider the blank cells as having the value 0. If you are reading a set of strings, then it inserts an empty string "" into the set. For example:
`s2 = {"Monday" "" "Wednesday" "Thursday" "Friday"}`

This behavior is the same when you don't use named range but instead use explicit ranges like C1:C5, where C2 is empty.

- With the Excel VBA one can name the first (top left) cell of a named range and access the whole range. OPL does not support this.
- When using sheetWrite to write to named ranges, the size of the range does not have to match the size of the data you are writing to Excel. If the set is smaller, then only the top most cells will be filled.

If you try to write more data than the range can accommodate, you receive the error message: "Exception from IBM ILOG Concert: excel: range is not wide enough to write the set".

In this sense, named ranges behave in exactly the same way as "regular" ranges.

Format of the Excel data

Here we must differentiate between simple types and tuples:

- Sets of simple types: The engine reads data from left to right and top to bottom. Data can therefore be read either horizontally, vertically, or from a rectangular zone.
- Sets of tuples: The data has to be formatted because the tuple schema has an arity. As in databases and manual tables, the data format is "fixed width, variable length". Therefore, tuple sets are read only line by line in Excel: this is the same representation as in pure data files.

Writing to a spreadsheet

Explains how to write to a spreadsheet from within OPL.

This section uses extracts from the oilSheet.dat data file.

Publishing results to a spreadsheet can be performed using instructions such as:

```
a to SheetWrite(sheet,"RESULT!A2:A4");  
blend to SheetWrite(sheet,"RESULT!B2:D4");
```

OPL then opens spreadsheets in read-write mode. This action may fail if another process is already using the .xls file.

The types of data elements supported for writing are just the same as for reading. Cells in Microsoft Excel spreadsheets are filled from left to right and from top to bottom.

Excel names (or named ranges) can be accessed using the SheetWrite command, using the following syntax:

```
SheetConnection sheetData("C:\\ILOG_Files\\myExcelFile.xls");  
prods to SheetWrite(sheetData,"Product");
```

Note the double separator \\ in the SheetConnection command. The SheetWrite command is normal, and, in this example, the Excel name Product replaces the normal syntax of, say, C13:072.

Note:

- OPL does not support the R1C1 reference style to specify the range when writing data to an Excel spreadsheet.

For more information on named ranges, see “Accessing named ranges in Excel” on page 25.

Data consistency

Defines the purpose of data consistency and describes data membership and assertions as ways to ensure consistency.

Purpose

Provides an overview of data consistency issues in OPL.

For an optimization problem to give relevant solutions, it is fundamental that you provide good quality data to your projects. In particular, it may be interesting to check that the data is consistent. If the project data is not consistent, the solving engine may find a wrong solution, or no solution, and you may think that the model is erroneous and therefore waste time trying to improve it.

OPL offers several ways to check the consistency of the data used by your projects.

In particular:

- “Data membership consistency”: use the keyword `with` to ensure that cells of a given tuple in a tuple set correctly belong to a given set of possible values.

Note:

You can also use the keyword `key` for data consistency when declaring tuples. See “Keys in tuple declarations” on page 10.

- “Assertions” on page 29: use the keyword `assert` to ensure that some assertions on the data are respected.

Data membership consistency

Explains the use of the `with` keyword to ensure data consistency.

The keyword `with` enables you to indicate that a given element of a tuple must be contained in a given set. If you use it, OPL checks the consistency of the tuple set at run time when initializing the set. The syntax is:

```
{tupletype} tupleset with cell1 in set1, cell2 in set = ...;
```

Let's take an example. You have a set of arcs between nodes. Nodes are defined by a tuple set of tuples consisting of an origin node and a destination node. The `with` syntax enables you to ensure that the origin and destination nodes do belong to a given set of nodes. Compare “Data found inconsistent (keyword with)” and “Data found consistent (keyword with)”:

Data found inconsistent (keyword with)

```
{int} nodes = {1, 5, 7};

tuple arc {
    int origin;
    int destination;
}

{arc} arcs2 with origin in nodes, destination in nodes =
    {<1,4>, <5,7>};

execute {
    writeln(arcs2);
};
```

Data found consistent (keyword with)

```
{int} nodes = {1, 5, 7};

tuple arc {
    int origin;
    int destination;
}

{arc} arcs1 with origin in nodes, destination in nodes =
    {<1,5>, <5,7>};

execute {
    writeln(arcs1);
};
```

If you write “Data found inconsistent (keyword with),” an error will be raised when the set `arcs2` is initialized because the `with` syntax will detect that the statement

```
(int) nodes = (1, 5, 7);
```

is not consistent with the statement

```
with origin in nodes, destination in nodes =
    {<1,4>, <5,7>}
```

If you write “Data found consistent (keyword with),” the initialization of the set `arcs1` will work properly because the `with` syntax will find that the statement

```
(int) nodes = (1, 5, 7);
```

is consistent with the statement

```
with origin in nodes, destination in nodes =
    {<1,5>, <5,7>}
```

Initializing tuple sets referring to other sets

To initialize tuple sets that refer to other sets **with keys** for data consistency, you must use initialization expressions that provide only those key values, as shown in

“Initializing tuple sets referring to other sets” on page 28. This is true if you initialize those tuple sets as internal data or as external data in .dat files, databases, or spreadsheets.

Initializing tuple sets referring to other sets

```
tuple node
{
    key int node_id;
    string city;
    string country;
}

{node} nodes = {<1,"Paris","France">,<5,"Madrid","Spain">, <7,"New York","USA">};

tuple arc {
    node origin;
    node destination;
}

{arc} arcs1 with origin in nodes, destination in nodes=...;

execute {
    writeln(arcs1);
};
```

Assertions

Explains the use of assertions with regard to data consistency.

OPL provides assertions to verify the consistency of the model data. This functionality enables you to avoid wrong results due to incorrect input data. In their simplest form, assertions are simply Boolean expressions that must be true; otherwise, they raise an execution error. For instance, it is common in some transportation problems to require that the demand matches the supply. The declaration

```
int demand[Customers] = ...;
int supply[Suppliers] = ...;

assert sum(s in Suppliers) supply[s] == sum(c in Customers) demand[c];
```

makes sure that the total supply by the suppliers meets the total demand from the customers. This assertion can be generalized to the case of multiple products, as in

```
int demand[Customers] [Products] = ...;
int supply[Suppliers] [Products] = ...;
assert
    forall(p in Products)
        sum(s in Suppliers) supply[s][p] == sum(c in Customers) demand[c][p];
```

This assertion verifies that the total supply meets the total demand for each product. The use of assertions is highly recommended, since they make it possible to detect errors in the data input early, avoiding tedious inspection of the model data and results.

Assertions can be labeled. See “Labeled assert statements” on page 50.

Note:

In OPL, elements are initialized when they are first used. This is called 'lazy initialization' (see “Data initialization” on page 15).

However, the execution of an assert statement triggers an evaluation of the elements involved in the assertion. So, if you use an OPL element for the first time in a skipped assert expression, the lazy initialization will be triggered later. Skipping or removing assert statements may impact the order in which OPL elements are initialized and thus change the behavior of a program.

Preprocessing of data

Provides an overview of preprocessing operations in OPL.

You can preprocess data before the optimization model is created by using IBM ILOG Script/JavaScript syntax encapsulated in execute blocks.

OPL provides script integration with the modeling language. All declared model elements are available for scripting via their name.

The functionality available for an element depends on its type. All elements can be read, but modifications are possible only for primitive types (int, float, string) and primitive items of arrays and tuples. See the to the *Reference Manual of IBM ILOG Script Extensions for OPL* about these limitations.

You can change the domain boundaries for decision variables, as well as their priority, in the preprocessing phase.

You can also use preprocessing to change CPLEX or CP Optimizer parameter settings (see Changing option values in the *Language User's Manual*).

Elements of a range or constraint type are immutable.

Example:

```
int n = ...;
range R = 1..n;
int A[R] = ...

execute {
  for(r in R) {
    if ( A[r]<0 ) {
      A[r] = 0;
    }
  }
}
```

Initialization and processing order

Preprocessing items are processed according to their category, not in absolute declaration order.

Within categories, the order is:

1. External data, in the order in which the data sources are added to the OPL model
2. All execute blocks and assert statements, in declaration order

Internal data is initialized as encountered during external data initialization, in execute blocks and in assert statements. The following examples use internal data.

If you write:


```

{int} s1={1,2};
{int} s2={ i | i in s1};
execute
{
writeln(s2);
s1.add(3);
writeln(s1,s2);
}

```

the result is:

```

{1 2}
{1 2 3} {1 2}

```

whereas if you write:

```

{int} s1={1,2};
{int} s2={ i | i in s1};
execute
{
//writeln(s2);
s1.add(3);
writeln(s1,s2);
}

```

the result is:

```

{1 2 3} {1 2 3}

```

The order in which you use the internal data is important. The order in which you use external data is not important because it is all created, or initiated, at the beginning of the run.

See also “Pitfall of lazy initialization of data.”

Use the profiler feature to inspect the instantiation sequence of your model. See *Profiling the execution of a model in IDE Tutorials*.

See Chapter 3, “IBM ILOG Script for OPL,” on page 87 for details on the scripting language and its extensions for OPL.

Lazy initialization

It is important to be aware from OPL 5.2 onwards that during processing, declared data elements are initialized on demand when referenced for the first time. See *Data preprocessing* in OPL Migration Guide for migration aspects.

Pitfall of lazy initialization of data

In OPL, external data is always initialized before internal data.

If you are not aware of the lazy initialization of data in OPL, you may be surprised by unexpected results.

OPL is a declarative language. The order of elements as they are written in the source file is less important than with imperative languages. We initialize OPL data elements when they are first used, which may result in an initialization order different from the declaration order in a .mod file. External data (with =... syntax) is always initialized before internal data. Scripting blocks are executed after the external data.

Example

In the following example, the array `arrayInitExternally` is initialized with three elements in the `.dat` file, but, because `set1` has four elements, the result shows that `arrayInitExternally` has one more element (initialized by default with zero).

```
string arrayInitByScript[0..3];

execute initializeArray {
    for(var i=0; i<=3; i++) {
        if(i >= 1) arrayInitByScript[i] = "A";
        else arrayInitByScript[i] = "B";
    }
}
// set1 is later used to index an array initialized externally.
// This external initialization of arrayInitExternally alters the content of set1
// because external initialization is done prior to script execution.
// So instead of having set1={1 2 3}, set1 has an additional element "0"
// Before the script is executed, arrayInitByScript contains four empty elements [" " " " " "]
{int} set1 = union(s in 0..3 : arrayInitByScript[s] != "B") {s};
{int} set2 = union(s in 0..3 : arrayInitByScript[s] != "B") {s};

int arrayInitExternally[set1] = ...; // [0, 0, 0]; //

execute {
    writeln(arrayInitByScript);
    writeln(set1);
    writeln(set2);
    writeln(arrayInitExternally);
}
```

This produces the output:

```
["B" "A" "A" "A"]
{0 1 2 3}
{1 2 3}
[0 0 0 0]
```

Decision types

Variables in an OPL application are decision variables (dvar). OPL also supports decision expressions, that is, expressions that enable you to reuse decision variables (dexpr). A specific syntax is available in OPL to dynamically collect elements into arrays.

Decision variables

Shows how to declare and use decision variables in the OPL language.

A decision variable is an unknown in an optimization problem. It has a domain, which is a compact representation of the set of all possible values for the variable. Decision variable types are *references* to objects whose exact nature depends on the underlying optimizer of a model. A decision variable can be instantiated only in the context of a given model instance.

The purpose of an OPL model is to find values for the decision variables such that all constraints are satisfied or, in optimization problems, to find values for the variables that satisfy all constraints and optimize a specific objective function. Variables in OPL are thus essentially decision variables and differ fundamentally from variables in programming languages such as Java, and ILOG Script.

Note:

OPL decision variables are noted with the `dvar` keyword while the keyword `var` denotes ILOG Script variables.

A decision variable declaration in OPL specifies the type and set of possible values for the variable. Once again, decision variables can be of different types (integer, float) and it is possible to define multidimensional arrays of decision variables. The declaration

```
dvar int transp[Orig][Dest] in 0..100;
```

declares a two-dimensional array of integer variables. The decision variables are constrained to take their values in the range `0..100` ; i.e., any solution to the model containing this declaration must assign values between 0 and 100 to these variables. Note that all integer variables need a finite range in OPL. Arrays of decision variables can be constructed using the same index sets as arrays of data. In particular, it is also possible, and desirable for larger problems, to index arrays of decision variables by finite sets. For example, the excerpt:

```
tuple Route {
    City orig;
    City dest
}
{Route} routes = ...;
dvar int transp[routes] in 0..100;
```

declares an array of decision variables `transp` that is indexed by the finite set of tuples `routes`. Genericity can be used to initialize the domain of the variables. For example, the excerpt:

```
tuple Route {
    City orig;
    City dest;
}
{Route} routes = ...;
int capacity[routes] = ...;
dvar int transp[r in routes] in 0..capacity[r];
```

declares an array of decision variables indexed by the finite set `routes` such that variable `transp[r]` ranges over `0..capacity[r]`. The array `capacity` is also indexed by the finite set `routes`. Note that decision variables can be declared to range over a user-defined range. For example, the excerpt:

```
range Capacity = 0..limitCapacity;
dvar int transp[Orig][Dest] in Capacity;
```

declares an array of integer variables ranging over `Capacity`.

Decision variables can of course be declared individually, as in:

```
dvar int averageDelay in 0..maxDelay;
```

For convenience, OPL proposes the types `float+`, `int+` and `boolean` to define the domain of a decision variable. The declarations

```
dvar int+ x; // non negative integer decision variable
dvar float+ y; // non-negative decision variable
dvar boolean z; // boolean decision variable
```

are therefore equivalent to

```
dvar int x in 0..maxint;
dvar float y in 0..infinity;
dvar int z in 0..1;
```

Decision variables in an array can be assigned item-specific ranges, as in

```
dvar float transp[o in Orig][d in Dest] in 0..cap[o][d];
```

which declares a two-dimensional array of float variables, where variable `transp[o][d]` ranges over the set `0..cap[o][d]`.

Expressions of decision variables

Shows how to declare and use decision variable expressions in the OPL language.

The keyword `dexpr` allows you to create reusable decision expressions. Indeed, if an expression has a particular meaning in terms of the original problem, writing it as a decision expression (`dexpr`) makes the model more readable.

For example, the `scalableWarehouse.mod` example expresses the total fixed costs as a decision expression:

```
dexpr int TotalFixedCost = sum( w in Warehouses ) Fixed * Open[w];  
dexpr float TotalSupplyCost = sum( w in Warehouses, s in Stores ) SupplyCost[s][w] * Supply[s][w];
```

This way, the two total cost expressions defined are shown in the OPL IDE along with their values.

You can also use arrays of decision expressions. For example:

```
dexpr int slack[i in r] = x[i] - y[i];
```

This array is handled efficiently as only the “definition” is kept. Not all the expressions for each value of the indices are created. As a consequence, you cannot change the definition of the `dexpr` for a particular element of the array.

Using decision expressions is particularly useful and recommended if you plan to write objectives to be used with ODM Enterprise. Please refer to the ODM Enterprise documentation.

Dynamic collection of elements into arrays

Discusses the “all” syntax, explicit arrays, appending arrays, and dynamic initialization of decision variable arrays.

Introduction

Provides an overview of how elements are collected into arrays in OPL.

Some expressions (such as `count`) and constraints (such as `allDifferent`) need arrays of variables or constants to be created. In some models, these expressions or constraints can be used in an aggregate statement (for example, in a `forall` statement) and the exact content of the arrays depends on the iteration.

The all syntax

Shows how to use the `all` syntax in the OPL language.

Then, the `all` syntax allows you to dynamically collect some decision variables or constants into an array. The syntax is similar to `sum` and `forall`, it contains a series of possible generators (an index and a set or a range in which this index is to be contained), some possible filters (to filter out some of the enumerated combinations), and a body (here of the form `x[i][j]...`). The variables or values in the resulting array follow the logical order of enumerating the index combinations as defined by the generators.

By default, this dynamic array is indexed from 0 to `numberOfElements-1`. As some constraints make a particular usage of the index, it may be interesting to define another indexing schema. For this, it is possible to dynamically define the range of the resulting array of variables by using the syntax `[minindex..maxindex]`. Finally, it is possible to use `"*"` as `maxIndex` to indicate that only the `minIndex` is defined; the `maxIndex` will be set accordingly depending on the number of elements.

Here is a complete of usage of the syntax:

```
using CP;

int n = 5;

range R = 1..n;
dvar int x[R] in R;

subject to {
    allDifferent(all(i in R:i%2==1) x[i]);
}
```

This is a way of defining an array of variables or values, and in all the constraints and expressions that take arrays. You can use either this syntax or pass directly a named array. When you pass a named array and indexes make sense in the constraint, its indexer will be used to index the elements if it has one dimension only. If it has two dimensions, the indexer cannot be used.

Explicit arrays

Describes explicit arrays in OPL.

Another useful syntax to dynamically create arrays to be used in expressions or constraints is to explicitly define the array using the `[]` notation and including any variables or values into it.

Mixes are not allowed. For example, you can write:

```
forall(i in R)
    allDifferent([x[i], y[i], z[i]]);
```

Appending arrays

You can concatenate several arrays using the `append` function.

For example, if you want to express that all variables from array `x` and array `y` are different, you can use an `allDifferent` constraint applied to the appended arrays, as shown in this example.

Appending arrays

```
using CP;

range R=1..10;
dvar int x[R] in 0..20;
dvar int y[R] in 0..20;

minimize sum(i in R)(x[i]+y[i]);
subject to
{
    allDifferent(append(all(i in 1..2) x[i],all(i in 4..6) y[i]));
}
```

Initialization of decision variable arrays

Shows how to initialize your decision variable arrays in OPL.

The dynamic collection of decision variables allows you to dynamically initialize an array of decision variables. The variables are then shared between the two arrays of variables.

Here is an example of what is possible:

```
int n = 5;
range R = 1..n;

dvar int x[i in R] in R;

dvar int y1 = x[1];
dvar int y[R] = all[R](i in R) x[i];
dvar int y2[i in R] = x[i];
dvar int y3[R] = [ i:x[i] | i in R ];
dvar int y4[R] = [ x[1], x[2], x[3], x[4], x[5] ];

dvar int y5[0..n-1] =
append(all(i in R: i<2) x[i], all(i in R: i>=2) x[i]);
```

Expressions

Describes data and decision variable identifiers, integer and float expressions, aggregate operators, piecewise-linear functions (continuous and discontinuous), set expressions, and Boolean expressions.

Usage of expressions

Describes how to use expressions in OPL.

Expressions are used in fundamentally different ways in OPL:

- to specify items in generic arrays and sets (described in this chapter)
- to filter iterations (see “Formal parameters” on page 59)
- to state constraints over decision variables (see “Constraints” on page 46)

In the first two cases, the expressions do not contain decision variables, since decision variables have no value at this stage of the computation. These expressions are said to be *ground* and they are subject to almost no restrictions.

In the second case, of course, the expressions may contain decision variables. Boolean expressions containing decision variables are called *constraints* and are subject to a number of restrictions; for example, float constraints must be linear, piecewise-linear, or quadratic.

Data and decision variable identifiers

Describes the use of identifiers within OPL expressions.

Since data and decision variable identifiers are the basic components of expressions, we will review briefly here how they are used to build expressions. If *r* is a tuple with a field capacity of type *T*, then *r.capacity* is an expression of type *T*. If *a* is an *n*-dimensional array of type *T*, *a[e 1] ... [e n]* is an expression of type *T*, provided that *e i* is well-typed. For instance, the excerpt

```
int limit[routes] = ...;
dvar int transp[r in routes] in 0..limit[r];
```

contains an expression *limit[r]* of type integer. Indices of arrays can be complex expressions. For instance, the excerpt

```

int nbFlights = ...;
range Flight = 1..nbFlights;
{string} Employee = ...;
dvar int crew[Flight][Employee] in 0..1;
constraints {

    forall(e in Employee)
        forall(i in 1..nbFlights - 2)
            crew[i][e] + crew[i+1][e] + crew[i+2][e] >= 1;
}

```

contains an integer expression `crew[i+1][e]` whose first index is itself an integer expression.

Integer and float expressions

Describes the use of constants, data, decision variables, and operators within OPL expressions.

Integer expressions

Integer expressions are constructed from integer constants, integer data, integer decision variables, and the traditional integer operators such as `+`, `-`, `*`, `div`, `mod` (or `%`). The operator `div` represents the integer division (for example, `8 div 3 == 2`) and the operator `mod` or `%` represents the integer remainder. OPL also supports the function `abs`, which returns the absolute value of its argument, and the built-in constant `maxint`, which represents the largest integer representable in OPL.

Note that expressions involving large integers may produce an overflow. In the following example for `int`, an overflow is detected.

```

int a=maxint+2;
float b=infinity+2;

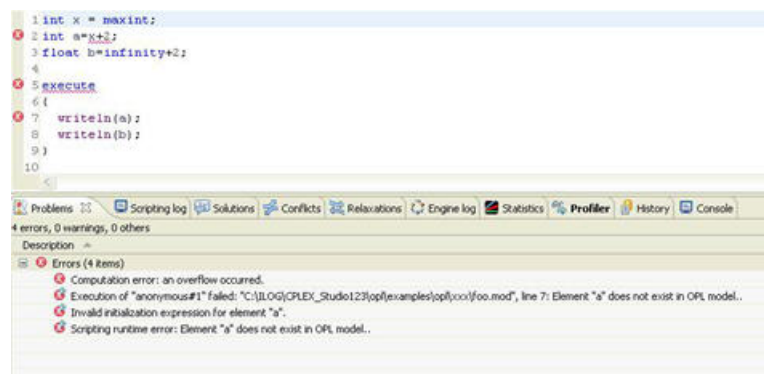
```

```

execute
{
    writeln(a);
    writeln(b);
}

```

The Problems tab in the IDE displays the error messages.



Most `int` expressions (such as `%` or `div`) are not available for constraints defined in CPLEX models, but are available for CP models. See also “Constraints available in constraint programming” on page 56.

Float expressions

Float expressions are constructed from floats, float data and variables, as well as operators such as $+$, $-$, $/$, $*$. In addition, OPL contains a float constant infinity to represent ∞ and a variety of float functions, depicted in *OPL functions* in the *Language Quick Reference*.

Conditional expressions

Conditional expressions are expressed like this:

`(condition)?thenExpr : elseExpr`

where *condition* is a ground condition with no decision variable. If *condition* is true, the condition evaluates to *thenExpr* ; otherwise, it evaluates to *elseExpr*.

Examples

```
int value = ...;
int signValue = (value>0) ? 1 : (value<0) ? -1 : 0;
int absValue = (value>=0) ? value : -value;
```

See the numeric functions in Summary table of OPL functions in the *Language Quick Reference*.

Counting expressions

Among integer expressions, there are also some combinatorial expressions. For example, you can use the `count` function to count the number of times a particular value appears in an array of decision variables. You can use such an expression in modeling constraints only if the modeling constraints are part of a model that is solved by the CP Optimizer engine (that is, starting with the `using CP;` statement).

The constraint

```
count(x, 2) == 3;
```

states that in the array of variables *x*, exactly three variables take the value 2.

For more information, see `count` in the *Language Quick Reference*.

Aggregate operators

Describes the operators available for computing integer and float summations.

Integer and float expressions can also be constructed using aggregate operators for computing summations (`sum`), products (`prod`), minima (`min`), and maxima (`max`) of a collection of related expressions. For instance, the excerpt

```
int capacity[Routes] = ...;
int minCap = min(r in Routes) capacity[r];
```

uses the aggregate operator `min` to compute the minimum value in array `capacity`. The form of the formal parameters in these aggregate operators is very general and is discussed at length in “Formal parameters” on page 59.

For information on operators in general, see *Operators* in the *Language Quick Reference*.

Piecewise-linear functions

Describes the use of piecewise-linear functions in OPL.

Piecewise-linear functions are important in many applications. They are often specified by giving a set of slopes, a set of breakpoints at which the slopes change, and the value of the functions at a given point. Consider, for instance, a transportation problem in which the transportation cost between two locations o and d depends on the size of the shipment $\text{ship}[o][d]$. The piecewise-linear expression

```
piecewise{10 -> 100;20 -> 200;40}(0,0) ship[o][d];
```

describes the piecewise-linear function of $\text{ship}[o,d]$ depicted in Figure 1. The function has slopes 10, 20, and 40, breakpoints 100 and 200, and evaluates to 0 at point 0.

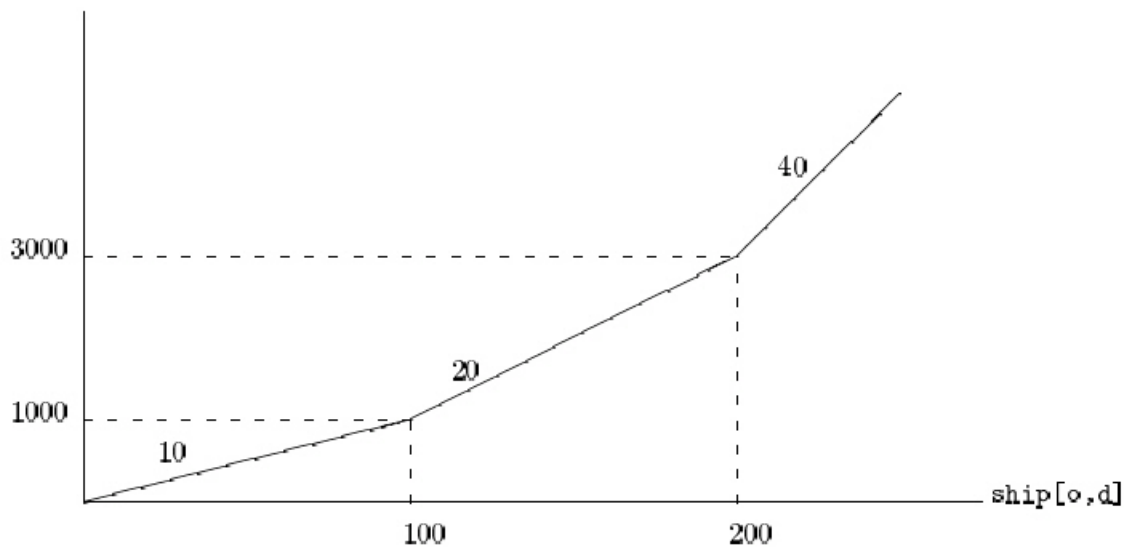


Figure 1. A Piecewise-linear function.

In other words, the piecewise-linear expression is equivalent to the expression:

```
10 * ship[o][d]
```

when

```
ship[o,d] <= 100
```

equivalent to

```
(10 * 100) + 20 * (ship[o][d] - 100)
```

when

```
100 <= ship[o][d] <= 200
```

and equivalent to

```
(10 * 100) + (20 * 200) + 40 * (ship[o][d] - 200)
```

otherwise.

By default, OPL assumes that a piecewise-linear function evaluates to zero at the origin, so that the last piecewise-linear function could actually be written as

```
piecewise{10 -> 100;20 -> 200;40} ship[o][d];
```

The last piecewise-linear function has a fixed number of pieces, but OPL also allows generic pieces. The number of pieces may then depend on the input data, as in

```
piecewise(i in 1..n) {
    slope[i] -> breakpoint[i];
    slope[n+1];
} ship[o][d];
```

If $\text{breakpoint}[1] \geq 0$ this piecewise-linear function is equivalent to

```
slope[1] * ship[o][d]
```

when

```
ship[o][d] <= breakpoint[1]
```

It is equivalent to

```
slope[1] * breakpoint[1] +

$$\sum_{i=2}^{k-1} \text{slope}[i] * (\text{breakpoint}[i] - \text{breakpoint}[i-1]) +$$

slope[k] * (ship[o][d] - breakpoint[k-1])
```

when

```
breakpoint[k-1] < ship[o][d] <= breakpoint[k] (1 < k <= n)
```

and equivalent to

```
slope[1] * breakpoint[1] +

$$\sum_{i=2}^n \text{slope}[i] * (\text{breakpoint}[i] - \text{breakpoint}[i-1]) +$$

slope[n+1] * (ship[o][d] - breakpoint[n])
```

otherwise.

Note that there may be several generic pieces in piecewise-linear functions. It is important to stress that breakpoints and slopes in piecewise-linear functions must always be grounded by a point on the piecewise-linear function. Such a point (called an anchor point) uniquely defines the function. Also, the breakpoints must be strictly increasing.

To sort your model data for this purpose, use sorted sets, as explained in “Sorted and ordered sets” on page 13.

Section Piecewise linear programming in the *Language User’s Manual* discusses piecewise-linear functions applied to an inventory problem.

Discontinuous piecewise-linear functions

Describes the use of discontinuous piecewise-linear functions in OPL.

OPL also allows you to write discontinuous piecewise-linear functions. This is the case when, in the syntax of a piecewise-linear function with slopes and breakpoints, two successive breakpoints are identical and the value associated with the second one is considered to be a “step” instead of a “slope”. The CPLEX and

the CP Optimizer engines behave differently with respect to what limit they consider as the discontinuity value. Because CPLEX allows either of these limits, note that the anchor point used to ground the breakpoints and slopes must not reside at the discontinuity. Otherwise, the piecewise-linear function would not be uniquely defined.

Behavior with the CPLEX engine

Example 1: the sign function

The following piecewise function:

```
piecewise{0 -> 0; 2 -> 0; 0}(1,1) x;
```

has a slope of 0 up to breakpoint 0, then a step of 2 at this breakpoint, then a slope of 0. It takes the value 1 at point 1. This piecewise represents the function `sign()` which returns the sign (1 or -1) of its argument, as represented in Figure 1 below.

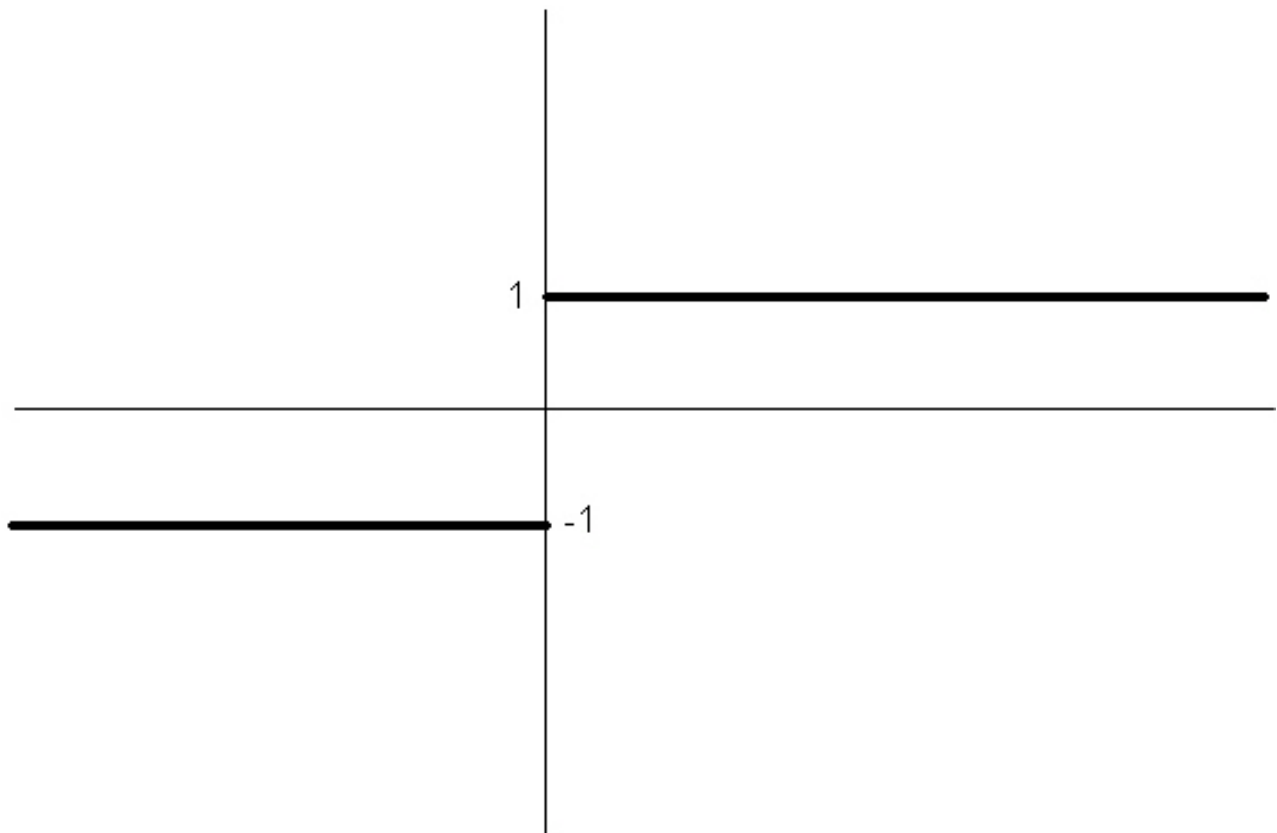


Figure 2. The discontinuous piecewise-linear function `sign()`

Then this model

```
dvar float x;  
dvar float signx;
```

```
dvar float y;  
dvar float signy;
```

```
maximize x;  
subject to {  
    x == 2;
```

```

    signx == piecewise{0->0; 2->0; 0}(1,1) x;
    y == -2;
    signy == piecewise{0->0; 2->0; 0}(1,1) y;
}

```

gives the following output:

```

Final solution with objective 2.0000:
x = 2.0000;
signx = 1.0000;
y = -2.0000;
signy = -1.0000;

```

Figure 2 on page 41 shows that the value of the sign function at the breakpoint is either -1 (on the left-hand slope) or 1 (on the right-hand slope).

For example, this model takes this into account and sets the constraint $x=y$; on both values.

```

dvar float x;
dvar float signx;

dvar float y;
dvar float signy;

maximize signx-signy;
subject to {
    x == y;
    signx == piecewise{0->0; 2->0; 0}(1,1) x;
    signy == piecewise{0->0; 2->0; 0}(1,1) y;
}

```

This model solves with the following output:

```

Final solution with objective 2:
signx = 1;
signy = -1;
x = 0;
y = 0;

```

Example 2: discontinuous cost

The following piecewise function

```

piecewise{0->0; 10->0; 0->10; 5->10; 0->20; 5->20; 0} (5,10) unit;

```

represents a discontinuous cost.

This function is illustrated in Figure 2, *Discontinuous costs*, for the values summarized in Table 1.

Table 4. A discontinuous cost function

Values of Unit	Cost
<0	0
0 to 10	10
10 to 20	15
>20	20

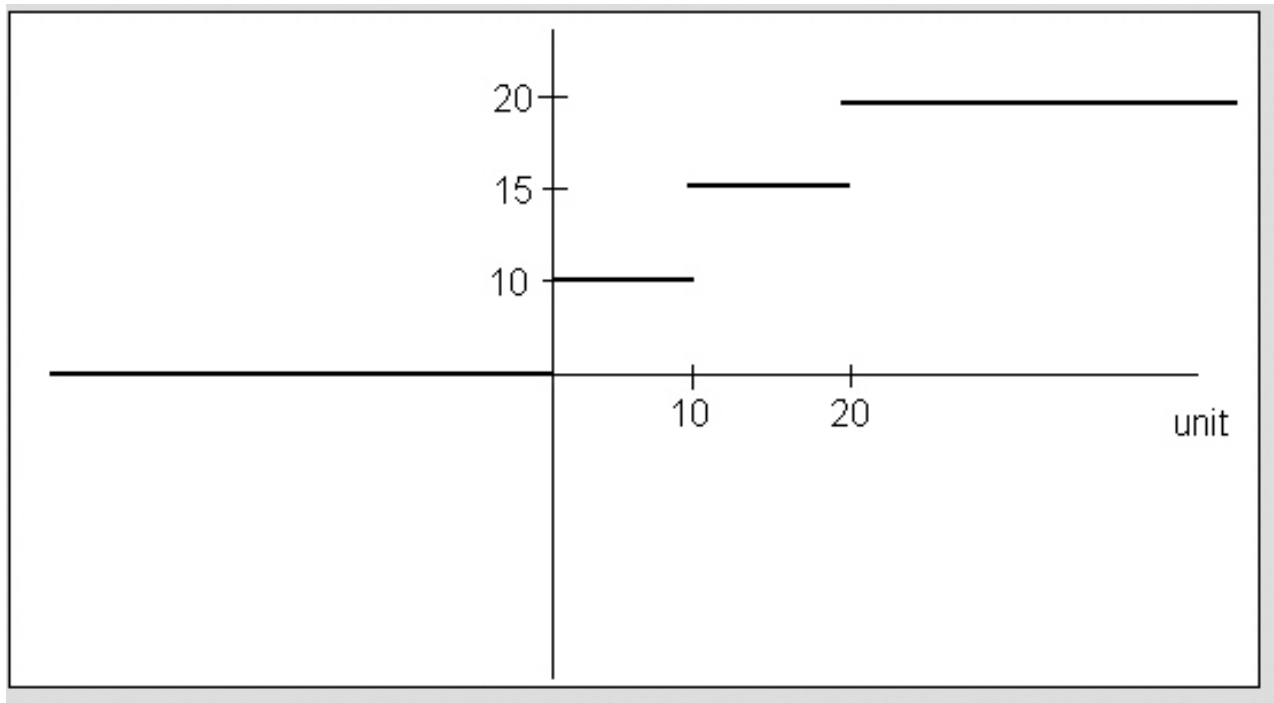


Figure 3. Discontinuous costs

Different behavior with the CP Optimizer engine

Consider the following model:

```
//using CP;
dvar int x in -10..10;
dvar int signx;
dvar int y in -10..10;
dvar int signy;
maximize signx-signy;
subject to {
    x == y;
    signx == piecewise{0->0; 2->0; 0}(1,1) x;
    signy == piecewise{0->0; 2->0; 0}(1,1) y;
}
execute
{
    writeln(signx-signy);
}
```

Depending on which solving engine you write for, you get a different result because CPLEX and CP Optimizer do not handle limit values in the same way.

- If you comment out the `using CP;` statement, the model is solved by the CPLEX engine and the result is 2 because CPLEX handles symmetry in such a way that it interprets either limit as the discontinuity value.
- However, if you uncomment the `using CP;` statement, the model is solved by the CP Optimizer engine and the result is 0 because CP Optimizer considers the left limit as the discontinuity value.

Set expressions

Describes the use of set expressions in OPL.

Set data can be initialized by set expressions, as mentioned in “Data types” on page 4. This section describes how these expressions are constructed and what functions are defined over sets.

- “Construction of set expressions”
- “Functions for sets”

Construction of set expressions

Set expressions are constructed from previously defined sets and the set operations union, inter, diff, and symdiff. For instance:

```
{int} s1 = {1,2,3};
{int} s2 = {1,4,5};
{int} i = s1 inter s2;
{int} u = s1 union s2;
{int} d = s1 diff s2;
{int} sd = s1 symdiff s2;
```

initializes *i* to {1}, *u* to {1,2,3,4,5}, *d* to {2,3}, and *sd* to {2,3,4,5}. In addition, set expressions can be constructed from ranges. For instance, the excerpt

```
{int} s = asSet(1..10);
```

initializes *s* to the finite set {1,2,...,10}

Important:

The range 1..10 takes constant space, while the set *s* takes space proportional to the number of elements in the range.

Functions for sets

Table 5 shows the functions available over sets. These methods apply to all sets, including sets of tuples and other complex sets. In this section, we assume that:

- *S* is the set 3, 6, 7, 9
- *item* is an element of *S*
- *n* is an integer number

Table 5. Functions over sets

Function	Description
card	card(<i>S</i>) returns the size of <i>S</i> , that is, the number of items.
ord	ord(<i>S</i> , <i>item</i>) returns the position of <i>item</i> in <i>S</i> . Positions start at 0 and ord(<i>S</i> , <i>item</i>) produces an execution error if <i>item</i> is not in <i>S</i> . Example: ord(<i>S</i> , 6) evaluates to 1 and ord(<i>S</i> , 9) to 3. The order of items in an explicit set is by order of appearance in the initialization and is implementation-dependent when the sets are the results of a set operation.
first	first(<i>S</i>) returns the first item in <i>S</i> , 3 in this example.

Table 5. Functions over sets (continued)

Function	Description
item	<p>item(S, n) returns the n-th item in set S. Counting starts from 0. This is equivalent to next(first(S), n)</p> <p>Example: item($S, 1$) = 6</p>
last	<p>last(S) returns the last item in S, 9 in this example.</p>
next	<p>next($S, item$) returns the item in S that comes after $item$ and produces an execution error if $item$ is the last item.</p> <p>Example: next($S, 3$) = 6</p> <p>next($S, item, n$) returns the n-th next item. next($S, item$) is equivalent to next($S, item, 1$).</p>
nextc	<p>A circular version of next. nextc($S, item$) returns the first item in S if $item$ is the last item.</p> <p>Example: nextc($S, 9$) = 3</p> <p>nextc($S, item, n$) returns the n-th circular next item. nextc($S, item$) is equivalent to nextc($S, item, 1$).</p>
prev	<p>prev($S, item$) returns the item in S that comes before $item$ and produces an execution error if $item$ is the first item.</p> <p>Example: prev($S, 6$) = 3</p> <p>prev($S, item, n$) returns the n-th previous item. prev($S, item$) is equivalent to prev($S, item, 1$).</p>
prevc	<p>A circular version of prev. prevc($S, item$) returns the last item in S if $item$ is the first item.</p> <p>Example: prevc($S, 3$) = 9</p> <p>prevc($S, item, n$) returns the n-th circular previous item. prevc($S, item$) is equivalent to prevc($S, item, 1$).</p>

Boolean expressions

Describes the use of Boolean expressions in OPL.

Boolean expressions can have various operand types in OPL. They are constructed in different ways:

- from integer expressions using the traditional relational operators ==, != (not equal), >=, >, <, and <=.
- from float expressions using the same relational operators.
- from string expressions and support the same operators as well.

For convenience, OPL offers a range expression to express special combinations for constraints.

They are of the form

`a op1 x op2 b`

where

- `op1` and `op2` are either of the relational operators `<=` or `<`
- `a` and `b` are boundary expressions which need to be ground
- `x` is an expression

Those expressions are equivalent to

`a op1 x; x op2 b`

which is itself equivalent to

`a op1 x && x op2 b`

Constraints

Specifies the constraints supported by OPL and discusses various subclasses of constraints to illustrate the support available for modeling combinatorial optimization applications.

Introduction

Constraints are a subset of Boolean expressions.

The availability of certain constraints depends on their context. The contexts can be:

- Data initialization when declared data is assigned
- Optimization model
 - a constraints block
 - a subject to block
- An expression that filters an iteration for aggregation or generation.

Using constraints

Explains how to apply a constraint to a decision variable and, possibly, conditionalize it. Also explains why constraints are identified by a label, and why constraints are used for filtering purposes.

Modeling constraints

Shows how to define modeling constraints in OPL.

Constraints passed to the algorithm, and which as such define the optimization problem, usually apply to decision variables; that is, they are Boolean expressions of some decision variables. To be taken into account by the solving algorithm, constraints must be stated using the optimization instruction:

`constraints`

or

`subject to`

as shown in the following example.

Stating constraints by means of an optimization instruction

```
minimize
    sum(p in Products) (insideCost[p]*inside[p] + outsideCost[p]*outside[p]);
subject to {
    forall(r in Resources)
        sum(p in Products) consumption[p,r] * inside[p] <= capacity[r];
    forall(p in Products)
        inside[p] + outside[p] >= demand[p];
}
```

Note:

1. Optimization instructions require an objective function of type integer or float.
2. That objective function must be defined before the constraints. Otherwise, a warning message is displayed.

Conditional constraints

Shows how to define conditional constraints in OPL.

If-then-else statements make it possible to state constraints conditionally, as in:

```
if ( d>1 ) {
    abs(freq[f] - freq[g]) >= d;
} else {
    freq[f] == freq[g];
}
```

Conditions in if-else statements must be *ground*; that is, they must not contain decision variables. Also, the if block should not contain forall statements such as:

```
if (..) {
    forall(...)
    ...
}
```

Implications of constraints can be used instead when conditions contain decision variables. Conditionals can also be used in OPL to make different choices according to the truth value of a condition.

Filtering with constraints

Describes the process of using constraints to filter decision variables or aggregates.

In addition to applying constraints to decision variables, you can also create constraints on formal parameters to filter aggregates, like this:

Filtering with constraints

```
// The cities where we are doing business
{string} cities={"Paris","Berlin","Washington","Rio"};

{string} EuropeanMainCapitals = {"London","Paris","Berlin","Madrid","Roma"};

// Should we expand business in this city ?
dvar boolean x[cities];

// We want to expand business in Europe
maximize sum(c in cities: c in EuropeanMainCapitals) x[c];

subject to
{
    // We can expand business in 2 cities
    sum(c in cities) x[c]<=2;
```

```

}

{string} expanded_cities = {c | c in cities : x[c]==1};

execute
{
  for(c in expanded_cities) writeln("We should expand business in ",c);
}

```

The result is:

```

We should expand business in Paris
We should expand business in Berlin

```

In this context, filtering means placing a condition on an iteration to limit the iteration loop. See “Formal parameters” on page 59 for more examples.

Constraint labels

Explains why label constraints, the benefits, costs, and limitations, how to label constraints, how to use indexed labels, and how to deal with compatibility between constraint names and labels.

Why label constraints?

Explains why attaching labels to your constraints is the recommended practice.

You can identify constraints by attaching labels to them. It is the recommended practice but it has a performance cost.

Benefits

- Constraint labels enable you to benefit from the expand feature in the IDE Problem Browser to find which constraints are tight in a given application or to find dual variable values in linear programs. See Understanding the Problem Browser in *Getting Started with the IDE*.
- You can access the slack and dual values for labeled constraints when a solution is available. See the class in the *Reference Manual of IBM ILOG Script Extensions for OPL*.
- Only labeled constraints are considered by the relaxation and conflict search process in infeasible models (see Relaxing infeasible models in *IDE Tutorials*).

Cost

However, labeling constraints has a performance and memory cost which can be significant, especially when a tuple pattern is used as the index. Therefore, you are encouraged to not use labels for large models or, if you do, at least use tuple indices instead of tuple patterns.

More precisely, constraint labels are used in three cases: IDE expand actions, in slack and dual values with solutions, and with relaxation and conflict detection. If you do not need these three use cases, you should get rid of the label to speed up the execution and lower memory consumption.

Labeling constraints

Shows how to label constraints in OPL.

Procedure

Type the character string you want, followed by the colon (:) sign, before the constraint you want to label, as shown in **Labelling constraints (production.mod)**. If you used to declare constraint names in your existing OPL models, see “Compatibility between constraint names and labels” on page 52.

Note:

A constraint label or name cannot start with a number.
Labeling constraints (production.mod)

```
minimize
    sum( p in Products )
        ( InsideCost[p] * Inside[p] + OutsideCost[p] * Outside[p] );

subject to {
    forall( r in Resources )
        ctCapacity:
            sum( p in Products )
                Consumption[p][r] * Inside[p] <= Capacity[r];

    forall(p in Products)
        ctDemand:
            Inside[p] + Outside[p] >= Demand[p];
}
```

This is equivalent to “Stating constraints by means of an optimization instruction” on page 47. The only difference is that the constraint on the production capacity has been labeled

ctCapacity:

and the constraint on the demand of products has been labeled

ctDemand:

These labels can be used to display the data.

Using indexed labels

Shows how to use indexed labels for your OPL constraints.

About this task

In some cases, it is more convenient to use indexed labels. Indexed labels enable you to control how a constraint is assigned to an array item.

The following example, Labeling constraints with indexed labels shows that the transp2.mod model identifies constraints using indexed labels following this syntax:

```
constraint ctDemand[Products];
...
ctDemand[p]:...
```

Indexed labels on constraints (transp2.mod)

```
forall( p in Products , d in Dest[p] )
    ctDemand[p][d]:
        sum( o in Orig[p] )
            Trans[< p,o,d >] == Demand[<p,d>];
ctCapacity: forall( o , d in Cities )
    sum( <p,o,d> in Routes )
        Trans[<p,o,d>] <= Capacity;
```

A case where you need indexed labels to reduce memory overhead is when you use forall iterations with variable sizes, as shown in the following code.

```
forall iterations with variable sizes
forall( p in Products , o in Orig[p] )
```

In this forall example, the second formal parameter o iterates on sets of potentially different sizes, depending on the value of the formal parameter p.

To use indexed labels:

Procedure

1. Declare the constraint array that will receive the labeled constraints.
constraint ctSupply[Products][Cities];
2. Add the indexing expressions to the label.
ctSupply[p][o]:

Example

The full code extract is shown here.

Labeling constraints with indexed labels

```
constraint ctSupply[Products][Cities];
constraint ctDemand[Products][Cities];

minimize
    sum(l in Routes) Cost[l] * Trans[l];

subject to {
    forall( p in Products , o in Orig[p] )

        ctSupply[p][o]:
            sum( d in Dest[p] )
                Trans[< p,o,d >] == Supply[<p,o>];
    forall( p in Products , d in Dest[p] )
        ctDemand[p][d]:
            sum( o in Orig[p] )
                Trans[< p,o,d >] == Demand[<p,d>];
```

Labeled assert statements

Shows how to label assertions in OPL.

Assertions can be labeled. When you label a constraint that is part of an assert statement, and if the assertion fails, the context of the failing assertions appears in the Issues output window. For example:

```
{int} vals = {1, 2, 3};
assert forall(i in vals) ct:i<2;
```

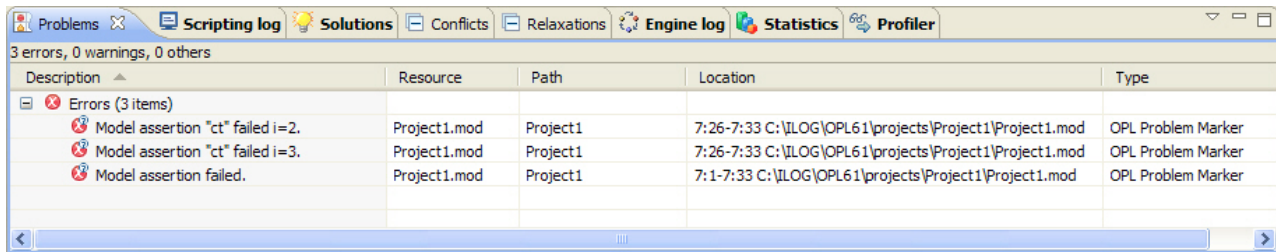


Figure 4. Labeled assert

Limitations on constraint labeling

Explains what constraints cannot be labeled or at what cost they can be.

Not all constraints can be labeled. Limitations exist with respect to forall statements and to a variable size indexer.

Labels and forall statements

You can label only constraints that are not nested within a forall statement (leaf constraints). However, you can label a forall constraint, if it is at the root level of constraints. For example, in the code sample Constraint label within forall statement, the constraint

```
ct1: forall(i in r1) forall(j in r2) X[i][j] <= i+j;
```

can also be written

```
forall(i in r1) forall(j in r2) ct1: X[i][j] <= i+j;
```

In both cases, the model executes correctly. However, if you execute:

```
forall(i in r1) ct1: forall(j in r2) X[i][j] <= i+j;
```

the IDE reports “Element "ct1" has never been used” and the constraint does not appear in the Problem Browser.

Constraint label within forall statement

```
range r1 = 1..2;
range r2 = 1..3;

dvar int X[r1][r2] in 0..5;

constraints {
  ct1: forall(i in r1) forall(j in r2) X[i][j] <= i+j;
}
```

Labels and a variable size indexer

Constraint labels with a variable size indexer are not allowed. For example, this model generates an error message.

```
tuple RangeTuple
{
  int i;
  int j;
  string k;
};
{RangeTuple} RT = {<1, 2, "bla">};
```

```

minimize 1;

subject to
{
    forall(<p1, p2, p3> in RT)
        forall(i in p1..p2)
            rangeLabel:
                1 == 1;
}

```

Write the following code instead.

```

tuple RangeTuple
{
    int i;
    int j;
    string k;
};
{RangeTuple} RT = {<1, 2, "bla">};

{int} s={1,2};

constraint rangeLabel[RT][s];

minimize 1;

subject to
{
    forall(<p1, p2, p3> in RT)
        forall(i in p1..p2)
            rangeLabel[<p1, p2, p3>,i]:
                1 == 1;
}

```

Note the difference in rangeLabel.

Compatibility between constraint names and labels

Explains compatibility issues for constraint names between OPL 4.x and subsequent versions.

OPL 4.x constraint names are deprecated in OPL 5.0 and later. They are still supported to maintain the compatibility of your OPL 4.x models and automatically implemented as labels internally, but they will be removed in future versions of IBM ILOG OPL. It is therefore strongly recommended that:

- you label constraints that are currently neither named nor labeled, as shown in “Labeling constraints” on page 48,
- you change possible existing constraint names in your models to labels. Compare Deprecated constraint names with Labeling constraints (production.mod). You do not need to previously declare the label as was the case with constraint names and you use the colon (:) sign instead of the equal (=) sign.

Deprecated constraint names

```

constraint capacityCons[r];
constraint demandCons[p];

minimize
    sum(p in Products) (insideCost[p]*inside[p] + outsideCost[p]*outside[p]);
subject to {
    forall(r in Resources)

```

```

        capacityCons[r]= sum(p in Products) consumption[p,r] * inside[p] <= capacity[r];
    forall(p in Products)
        demandCons[p]= inside[p] + outside[p] >= demand[p];
}

```

Types of constraints

Classifies constraints according to their operand type.

Float constraints

Float constraints are constraints involving float decision variables.

When **modeled** in OPL, they are restricted to be linear or piecewise linear. OPL has efficient algorithms for solving linear, piecewise linear, quadratic, or logical constraints, but in general these algorithms do not apply to nonlinear problems. Note that the linearity requirement precludes the use of relations with variables in constraints and the use of non-ground expressions as indices of float arrays. In addition, operators !=, <, and > are not allowed for float constraints. However, integers and integer variables may occur in a float constraint, provided that the constraint remains linear or quadratic. OPL supports all the expressions supported by CPLEX, provided that the constraint is of one of the types described at the beginning of this paragraph.

Discrete constraints

Discrete constraints are arbitrary Boolean expressions with integer operands, possibly containing variables.

In OPL, these constraints must be well-typed, but no restrictions are imposed on them. It is, however, useful to review subclasses of these constraints to illustrate the functionalities of OPL. MP models, solved by the CPLEX engine, can contain only basic constraints (see the section *Basic constraints* below). Logical constraints can be set by filtering of forall and sum constraints.

Basic constraints

Basic discrete constraints are constructed from discrete data, discrete variables, and the arithmetic operators and functions defined in “Expressions” on page 36. For instance, the excerpt

```

range r=1..5;
dvar int x[1..5] in 0..10;
dvar int obj;

maximize obj;

subject to
{
    obj==sum(ordered i,j in r) abs(x[i]-x[j]);
    forall(ordered i,j in r) abs(x[i]-x[j])>=1;
}

```

generates distance constraints between integer variables.

Note that the following code creates an error because IBM ILOG CPLEX does not accept non linear constraints.

```

dvar int+ X in 0..1000;
minimize X;
subject to {
    X mod 7 == 0;
};

```

String constraints

Describes the use of string constraints in OPL.

String constraints cannot be used on decision variables or added to the model to be solved. They can only be used on indexers to filter aggregates (see “Filtering with constraints” on page 47). For example:

```
{string} s = {"a", "b"};
dvar int x[s] in 0..10;
minimize sum(i in s) x[i];
subject to {
    forall(i in s : i != "a")
        x[i] >= 5;
}
```

Implicit constraints

Explains that implicit constraints may imply infeasibility.

Implicit constraints are implied by operators. For example:

```
using CP;
int a[1..4];

dvar int x in 1..5;

maximize a[x];

subject to
{
    x==5;
}
```

makes the model infeasible even though

```
using CP;
int a[1..4];

dvar int x in 1..5;

subject to
{
    x==5;
}
```

is feasible.

Logical constraints for CPLEX

Describes the use of logical constraints in OPL.

Logical constraints are one particular kind of discrete or numerical constraints. OPL and CPLEX can translate logical constraints automatically into their transformed equivalent that the discrete (MIP) or continuous (LP) optimizers of IBM ILOG CPLEX can process efficiently. This section describes all the available logical constraints, as well as the logical expressions that can be used in logical constraints. Logical constraints are available in constraint programming models without linearization.

For an example of how OPL uses logical constraints, see Tutorial: Using CPLEX logical constraints in the *Language User's Manual*.

In this section, you will learn:

- “What are logical constraints?” on page 55

- “What can be extracted from a model with logical constraints?”
- “Which nonlinear expressions can be extracted?”
- “Logical constraints for counting” on page 56
- “How are logical constraints extracted?” on page 56

What are logical constraints?

For IBM ILOG CPLEX, a logical constraint combines linear constraints by means of logical operators, such as logical-and, logical-or, implication, negation (not), conditional statements (if ... then ...) to express complex relations between linear constraints. IBM ILOG CPLEX can also handle certain logical expressions appearing within a linear constraint. One such logical expression is the minimum of a set of variables. Another such logical expression is the absolute value of a variable. There’s more about logical expressions in “Which nonlinear expressions can be extracted?”

What can be extracted from a model with logical constraints?

The following table lists the logical constraints that CPLEX can extract.

Symbol	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT
=>	Imply
!=	Different from
==	Equivalence

All those constructs accept as their arguments other linear constraints or logical constraints, so you can combine linear constraints with logical constraints in complicated expressions in your application.

Note:

With the C++ API you should use the symbol <= to express implication.

Which nonlinear expressions can be extracted?

Some expressions are easily recognized as nonlinear, for example, a function such as

$$x^2 + y^2 = 1$$

However, other nonlinearities are less obvious, such as absolute value as a function. In a very real sense, MIP is a class of nonlinearly constrained problems because the integrality restriction destroys the property of convexity which any linear constraints otherwise might possess. Because of that characteristic, certain (although not all) nonlinearities are capable of being converted to a MIP formulation, and thus can be solved by IBM ILOG CPLEX. The following nonlinear expressions are accepted in an OPL model:

- min and min| : the minimum of several numeric expressions
- max and max| : the maximum of several numeric expressions
- abs : the absolute value of a numeric expression

- `piecewise` : the piecewise linear combination of a numeric expression
- A linear constraint can appear as a term in a logical constraint.

In fact, ranges containing logical expressions can, in turn, appear in logical constraints. It is important to note here that only linear constraints can appear as arguments of logical constraints extracted by IBM ILOG CPLEX. That is, quadratic constraints are not handled in logical constraints. Similarly, quadratic terms can not appear as arguments of logical expressions such as `min`, `max`, `abs`, and `piecewise`.

Logical constraints for counting

In many cases it is even unnecessary to allocate binary variables explicitly in order to gain the benefit of linear constraints within logical expressions. For example, optimizing how many items appear in a solution is often an issue in practical problems. Questions of counting (how many?) can be represented formally as cardinality constraints. Suppose that your application includes three variables, each representing a quantity of one of three products, and assume further that a good solution to the problem means that the quantity of at least two of the three products must be greater than 20. Then you can represent that idea in your application, like this:

```
(x[0] >= 20) + (x[1] >= 20) + (x[2] >= 20) >= 2;
```

How are logical constraints extracted?

Logical constraints are transformed automatically into equivalent linear formulations when they are extracted by an IBM ILOG CPLEX algorithm. This transformation involves automatic creation by IBM ILOG CPLEX of new variables and constraints. For more details on this transformation, refer to the IBM ILOG CPLEX documentation.

Constraints available in constraint programming

Lists the types of constraints and expressions available in OPL for constraint programming models. See the *OPL Language Quick Reference* and the *CP Optimizer User's Manual* for further details.

- “Arithmetic constraints and expressions”
- “Logical constraints for CP” on page 57
- “Element constraints” on page 57
- “Compatibility constraints” on page 58
- “Specialized constraints” on page 58

Arithmetic constraints and expressions

The following arithmetic constraints and expressions are available for OPL CP models. The references point to the *OPL Language Quick Reference*.

- Arithmetic operations
 - addition
 - subtraction
 - multiplication
 - scalar products
 - integer division
 - floating-point division
 - modular arithmetic

- Arithmetic expressions for use in constraints.
 - standard deviation: see standardDeviation
 - minimum: see min
 - maximum: see max
 - counting: see count
 - absolute value: see abs
 - element or index: see element
- Arithmetic constraints
 - equal to
 - not equal to
 - strictly less than
 - strictly greater than
 - less than or equal to
 - greater than or equal to

Logical constraints for CP

Logical operators, such as logical-and, logical-or, implication, negation (not), difference, and equivalence are available in constraint programming.

The following logical constraints are available for OPL CP models.

Symbol	Meaning
&&	Logical AND
	Logical OR
!	Logical NOT
=>	Imply
!=	Different from
==	Equivalence

Note:

With the C++ API you should use the symbol <= to express implication.

Element constraints

The following subscripting operators return constrained integer or floating-point expressions, also known as element expressions.

- $A[X]$, where A is an array of integer values (with one or several dimensions) and X is a dexpr int, returns a dexpr int such that: when X is fixed to the value i , the value of the expression is $A[i]$.
- $A[X]$, where A is an array of floating-point values (with one or several dimensions) and X is a dexpr int, returns a dexpr float such that: when X is fixed to the value i , the value of the expression is $A[i]$.

More generally, the domain of the expression is the set of values $A[i]$, where i is in the domain of X .

In both of the previous cases, the expression can include several indices ($A[X][k]$, $A[X][Y]$, $A[X][k]\{Y\}$, ...).

- For constant indices, all the usual index types are available (int, float, string, tuple).
- For a decision expression index, only the integer type is available (dvar int or dexpr int).

Compatibility constraints

The CP Optimizer engine supports allowed and forbidden assignments for OPL CP models. See `allowedAssignments` and `forbiddenAssignments` in the *Language Quick Reference*.

Scheduling constraints

This section lists the scheduling constraints available for OPL CP scheduling models and provides links to the reference documentation for these constraints. For a more detailed description of these constraints, please refer to the “Scheduling” on page 63 section of this manual.

The following constraints are available for OPL CP scheduling models:

- Precedence constraint: `endAtEnd`
- Precedence constraint: `endAtStart`
- Precedence constraint: `endBeforeEnd`
- Precedence constraint: `endBeforeStart`
- Precedence constraint: `startAtEnd`
- Precedence constraint: `startAtStart`
- Precedence constraint: `startBeforeEnd`
- Precedence constraint: `startBeforeStart`
- Interval grouping constraint: `alternative`
- Interval grouping constraint: `span`
- Interval grouping constraint: `synchronize`
- Interval presence constraint: `presenceOf`
- Sequence constraint: `first (scheduling)`
- Sequence constraint: `last (scheduling)`
- Sequence constraint: `before`
- Sequence constraint: `prev`
- Sequence constraint: `noOverlap`
- Cumulative or state function constraint: `<= (scheduling), alwaysIn`
- State function constraint: `alwaysConstant`
- State function constraint: `alwaysEqual`
- State function constraint: `alwaysNoState`

Specialized constraints

The CP Optimizer engine also accepts some powerful combinatorial constraints known as specialized constraints. For these constraints, some powerful propagation algorithms are used to reduced the decision variable domains.

- `allDifferent` : constrains variables within a dvar array to all take different values
- `allMinDistance` : constrains variables within a dvar array to all take values that are one-to-one different by at least a given gap

- `inverse` : takes two arrays of integer variables that must be indexed by an integer and be one-dimensional
- `lex` : states that the first array of variables is less than or equal to the second array of variables in the alphabetical order
- `pack` : represents some simple but powerful one-dimensional packing constraint

See the individual entries under OPL functions in *Language Quick Reference* for a complete description of each constraint.

Limitations on constraints

There is a limitation on certain types of constraints.

You cannot change the bound of a top level inequality constraint unless that bound is a literal expression, such as $x \geq 13$ or $x \leq 13$.

As a workaround, you can declare your constraint either by using a range syntax, or by putting the inequality inside a `forall` statement.

Instead of

```
ct0: x ≤ a[0];
```

you can write

```
ct1: -infinity ≤ x ≤ a[0];
```

or

```
forall(i in 0..0)
  ct2: x ≤ a[0];
```

and change the upper (or lower) bound of the constraint with a script instruction such as:

```
ct1.UB = 1313;
```

or

```
ct2[0].UB = 1313;
```

Formal parameters

Describes basic formal parameters, tuples of parameters, and filtering in tuples of parameters.

Basic formal parameters

Formal parameters play a fundamental role in OPL; they are used in aggregate operators, generic sets, and `forall` statements.

The simplest formal parameter has the form

p in S

where p is the formal parameter and S is the set from which p takes its values.

The set S can be:

- an integer range, as in

```

int n=6;
int s = sum(i in 1..n) i*i;

```

- a string set, as in

```

{string} Products = {"car", "truck"};
float cost[Products] = [12000, 10000];
float maxCost = max(p in Products) cost[p];

```
- or a tuple set, as in

```

{string} Cities = { "Paris", "London", "Berlin" };
tuple Connection
{
    string orig;
    string dest;
}
{Connection} connections = { <"Paris", "Berlin">, <"Paris", "London"> };
float cost[connections] = [ 1000, 2000 ];
float maxCost = max(r in connections) cost[r];

```

If you need to filter the range of the formal parameters using conditions, the formal parameter then takes the form

p in S : filtering condition

and assigns to *p* all elements of *S* according to the filter applied.

For instance, in the excerpt

```

int n=8;
dvar int a[1..n][1..n];
subject to
{
    forall(i in 1..8)
        forall(j in 1..8: i < j)
            a[i][j] >= 0;
}

```

the constraint $a[i][j] \geq 0$ is modeled for all *i* and *j* such that $1 \leq i < j \leq 8$.

Note:

OPL does not support aggregates in filter expressions. For example:

- For an expression such as:

```

{int} notFirst = {i | i in 1..10 : card({j | <i,j> in pairs}) == 0};

```

the type check displays the error: "Aggregate set is currently not supported for filter expressions."
- For an expression such as:

```

{int} notFirst = {i | i in 1..10 : sum(<i,j> in pairs) i == 0};

```

the type check displays the error: "Aggregate sum is currently not supported for filter expressions."

Several parameters can often be combined together to produce more compact statements. For instance, the declaration

```
int s = sum(i,j in 1..n: i < j) i*j;
```

is equivalent to

```
int s = sum(i in 1..n) sum(j in 1..n: i < j) i*j;
```

which is less readable.

The declaration

```
int s = sum(i in 1..n, j in 1..m) i*j;
```

is equivalent to

```
int s = sum(i in 1..n) sum(j in 1..m) i*j;
```

These parameters can, of course, be subject to filtering conditions. The excerpt

```
forall(i,j in 1..n : i < j)
    a[i][j] >= 0;
```

is equivalent to

```
forall(i in 1..n, j in 1..n : i<j)
    a[i][j] >= 0;
```

Here is an even more compact form:

```
forall(ordered i,j in 1..n)
    a[i][j] >= 0;
```

Indeed, in many applications one is interested, given a set S , in stating filters or conditions over all pairs (i, j) of elements of S satisfying $i < j$ in the ordering associated with S . In this excerpt

```
{T} S = ...;
forall(ordered s, t in S)...;
forall(s in S, t in S: ord(S,s) < ord(S,t)) ...
```

the first `forall` line is equivalent to the second one and illustrates the functionality `ordered`, often useful in practical applications. T can be one of the types `int`, `float`, `string`, or a tuple type.

Important:

This ordering does not refer to the ordering associated with type T but to the order of the items within the set.

Tuples of parameters

OPL allows tuples of formal parameters to be created in aggregate operators, `forall` statements, and generic sets.

The following code sample states the precedence constraints between tasks. The constraint declaration requires explicit accesses to the fields of the tuple to state the constraints. In addition, the field `before` is accessed twice. An alternative way to state the same constraint is to use a tuple of formal parameters, as shown in the last line of the code sample, precluding the need to access the tuple fields explicitly. The tuple `<p in Prec>` in the `forall` quantifier contains two components that are successively given the values of the fields of each tuple in `Prec`.

Tuple of formal parameters

```
int minTime=7*60;
int maxTime=9*60;

{string} Tasks = { "Make dinner","Have dinner","Clean post dinner" };
tuple Precedence {
    string pre;
    string post;
}

{Precedence} Prec = {
```

```

    <"Make dinner","Have dinner">,
    <"Have dinner","Clean post dinner">
};

int Duration[Tasks]= [20,60,10];

dvar int Start[Tasks] in minTime..maxTime;

subject to {
    forall(p in Prec) Start[p.post] >= Start[p.pre] + Duration[p.pre];
}

```

More generally, an expression

p in S

where S is a set of tuples containing n fields, can be replaced by a formal parameter expression

$\langle p_1, \dots, p_n \rangle$ in S

that contains n formal parameters. Each time a tuple r is selected from S , its fields are assigned to the corresponding formal parameters. This functionality is often useful in producing more readable models.

Filtering in tuples of parameters

OPL enables simple equality constraints to be factorized inside tuples, which is important in obtaining more readable and efficient models. In this context, slicing refers to nested iterations with filtering conditions.

Consider, for instance, a transportation problem where products must be shipped from one set of cities to another set of cities. The model may include a constraint specifying that the total shipments for all products transported along a connection may not exceed a specified limit. This can be expressed by a constraint

Explicit slicing

```

forall(c in connections)
    sum(<p,co> in routes: c == co) trans[<p,c>] <= limit;

```

This constraint states that the total products shipped along each connection c is not greater than $limit$. OPL must scan the entire set $routes$ to select the tuples involving each connection. In this example, the expression $c==co$ is used to make slicing *explicit*.

The constraint would be stated equivalently as follows:

Implicit slicing

```

forall(c in connections)
    sum(<p,c> in routes) trans[<p,c>] <= limit;

```

In this constraint, the tuple $\langle p, c \rangle$ contains one new parameter p and uses the previously defined parameter c . Since the value of c is known, OPL uses it to index the set $routes$, avoiding a complete scan of the set $routes$. In this example, slicing is said to be *implicit* because the formal parameter c is used to declare iteration in both the `forall` and `sum` loops. You can also use a constant as a tuple item, for example $\langle p, 2 \rangle$, for implicit slicing.

In OPL 4.0 and later versions, models tend to be more readable when explicit slicing is used. Besides, there is no performance advantage in using implicit slicing over explicit slicing.

More about implicit slicing

You should be aware that the following statement:

```
int array[i in set1] = ((sum(i in set2) 1 >= 1) ? 1:0);
```

is exactly equivalent to

```
int array[i in set1] = ((sum(j in set2) 1 >= 1) ? 1:0);
```

that is, the two “i” on either side of the “equal” sign =, in the first statement, are not linked. This is called *scope hiding* because the second “i” hides the first one in a nested scope.

In contrast, this statement

```
int array[<i,j> in set1] = ((sum(<i,j> in set2) 1 >= 1) ? 1:0);
```

codes implicit slicing, which is equivalent to:

```
int array[i in set1] = ((sum(j in set2 : j==i) 1 >= 1) ? 1:0);
```

In other words, there is no implicit slicing outside tuple patterns.

See also Modeling tips in the *Language User's Manual*.

Scheduling

Describes how to model scheduling problems in OPL.

Introduction

OPL and CP Optimizer introduce a set of modeling features for applications that deal with scheduling over time.

In OPL and CP Optimizer, time points are represented as integers, but the possible very wide range of time points means that time is effectively continuous. A consequence of scheduling over effectively continuous time is that the evolution of some known quantities over time (for instance the instantaneous efficiency/speed of a resource or the earliness/tardiness cost for finishing an activity at a given date t) needs to be compactly represented in the model. To that end, CP Optimizer provides the notion of *piecewise linear* and *stepwise* functions.

Most scheduling applications consist in scheduling in time some activities, tasks or operations that have a start and an end time. In CP Optimizer, this type of decision variable is captured by the notion of the *interval variable*. Several types of constraints are expressed on and between interval variables:

- to limit the possible positions of an interval variable (forbidden start/end or “extent” values)
- to specify precedence relations between two interval variables
- to relate the position of an interval variable with one of a set of interval variables (spanning, synchronization, alternative).

An important characteristic of scheduling problems is that time intervals may be optional, and whether to execute a time-interval or not is a possible decision

variable. In CP Optimizer, this is captured by the notion of a boolean *presence* status associated with each interval variable. Logical relations can be expressed between the presence of interval variables, for example to state that whenever interval a is present then interval b must also be present.

Another aspect of scheduling is the allocation of scarce resources to time intervals. The evolution of a resource over time can be modelled by two types of variables:

- The evolution of a disjunctive resource over time can be described by the sequence of intervals that represent the activities executing on the resource. For that, CP Optimizer introduces the notion of an *interval sequence variable*. Constraints and expressions are available to control the sequencing of a set of interval variables.
- The evolution of a cumulative resource often needs a description of how the accumulated usage of the resource evolves over time. For that purpose, CP Optimizer provides the concept of the *cumulative function expression* that can be used to constrain the evolution of the resource usage over time.
- The evolution of a resource of infinite capacity, the state of which can vary over time, is captured in CP Optimizer by the notion of the *state function*. The dynamic evolution of a state function can be controlled with the notion of transition distance, and constraints are available for specifying conditions on the state function that must be satisfied during fixed or variable intervals.

Some classical cost functions in scheduling are earliness/tardiness costs, makespan, and activity execution or non-execution costs. CP Optimizer generalizes these classical cost functions and provides a set of basic expressions that can be combined together; this allows you to express a large spectrum of scheduling cost functions that can be efficiently exploited by the CP Optimizer search.

For the description of the symbolic notation used throughout this section, see “Notation” on page 84.

Piecewise linear and stepwise functions

Describes piecewise linear and stepwise functions as related to scheduling.

In CP Optimizer, piecewise linear functions are typically used to model a known function of time, for instance the cost incurred for completing an activity after a known date t . Stepwise functions are typically used to model the efficiency of a resource over time.

A **piecewise linear function** $F(t)$ is defined by a tuple $F = \text{piecewise}(S, T, t_0, v_0)$ where:

- S is a vector of $n + 1$ “change” values $s_i \in \mathbb{R}$ for F
- T is a vector of n values $t_i \in \mathbb{R}$ for F such that
 $\forall i \in [1, n - 1], t_i \leq t_{i+1} \wedge \forall i \in [1, n - 2], t_i < t_{i+2}$
- (t_0, v_0) is a reference point of the function

The function is defined everywhere on $(-\infty, +\infty)$ as follows:

- $F(t_0) = v_0$
- the slope $F'(t)$ of F on the segment $(-\infty, t_1)$ is equal to s_1
- the slope $F'(t)$ of F on the segment $[t_n, +\infty)$ is equal to s_{n+1}
- $\forall i \in [1, n - 1]$:
 - if $t_i \neq t_{i+1}$, then the slope $F'(t)$ of F on the segment $[t_i, t_{i+1})$ is equal to s_{i+1} . That is, $\forall t \in [t_i, t_{i+1})$, $F(t) = F(t_i) + s_{i+1}(t - t_i)$. Note that the point t_{i+1} is not included in the segment.
 - if $t_i = t_{i+1}$, then F is discontinuous at point t_i . s_{i+1} represents the height of the step (positive or negative) at t_i . The value of $F(t_i)$ is equal to the value of the leftmost extremity of the segment immediately to the right of the discontinuity. Formally, this is given by:
 - * if $i \neq n - 1$ then $F(t_i) = F(t_{i+2}) + s_{i+2}(t_i - t_{i+2})$
 - * else if $i \neq 1$ then $F(t_i) = F(t_{i-1}) + s_i(t_i - t_{i-1}) + s_{i+1}$
 - * { Otherwise, function has form slope / step / slope, and $i = 1$ }
 - * else if $t_0 \geq t_i$ then $F(t_i) = v_0 + s_3(t_i - t_0)$
 - * else $F(t_i) = v_0 + s_1(t_i - t_0) + s_2$

For a complete description of the OPL syntax of a piecewise linear function, see `piecewise` and `pwlFunction` in the *OPL Language Quick Reference*.

A **stepwise function** is a special case of the piecewise linear function, where all slopes are equal to 0 and the domain and image of F are integer. A stepwise function $F(t)$ is defined by a tuple $F = \text{stepwise}(V, T)$ where:

- V is a vector of $n + 1$ function values $v_i \in \mathbb{Z}$ for F
- T is a vector of n values $t_i \in \mathbb{Z}$ for F such that $\forall i, t_i < t_{i+1}$

The function is defined everywhere on $(-\infty, +\infty)$ and the function value at a point t is given by:

- $\forall t < t_1, F(t) = v_1$
- $\forall i \in [1, n - 1], \forall t \in [t_i, t_{i+1}), F(t) = v_{i+1}$
- $\forall t \geq t_n, F(t) = v_{n+1}$

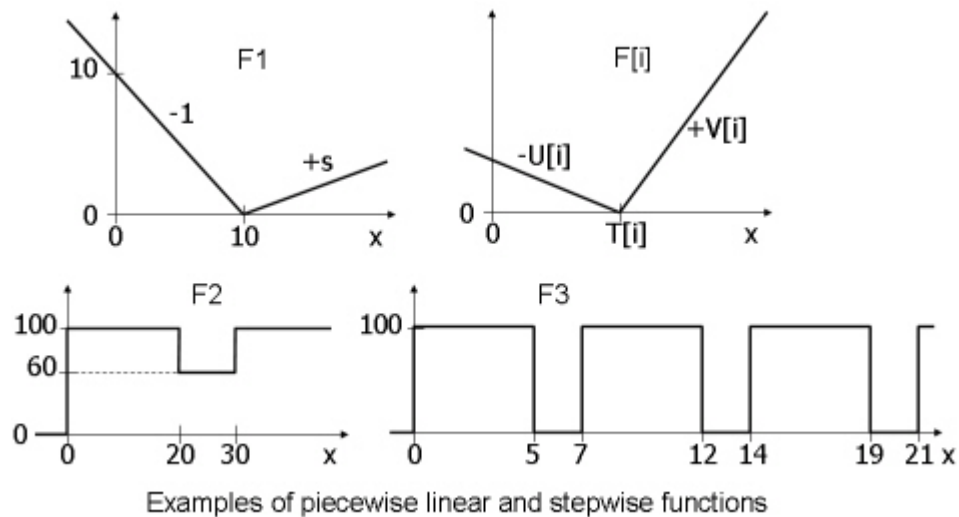
For a complete description of the OPL syntax of a stepwise linear function, see `stepwise` and `stepFunction` in the *OPL Language Quick Reference*.

Examples

- A V-shape function with value 0 at $x = 10$, slope -1 before $x = 10$ and slope s afterwards:
`pwlFunction F1 = piecewise{ -1->10; s } (10, 0);`
- An array of V-shaped functions indexed by i in $[1..n]$ with value 0 at $T[i]$, slope $-U[i]$ before $T[i]$ and slope $V[i]$ afterwards (T , U and V are data integer arrays):
`pwlFunction F[i in 1..n] = piecewise{ -U[i]->T[i]; V[i] } (T[i], 0);`

- A stepwise function with value 0 before 0, 100 on $[0, 20)$, value 60 on $[20, 30)$, and value 100 later on:
`stepFunction F2 = stepwise{ 0->0; 100->20; 60->30; 100 };`
- A stepwise function with value 0 everywhere except on intervals $[7i, 7i+5)$ for i in $[0, 51]$ where the value is 100:
`stepFunction F3 = stepwise(i in 0..51, p in 0..1) { 100*p -> (7*i)+(5*p)
; 0 };`

The previous piecewise and stepwise function are depicted in the following diagram.



Interval variables

Describes a basic building block of scheduling, the *interval*.

Informally speaking, an interval variable represents an interval of time during which something happens (a task, an activity is carried out) and whose position in time is an unknown of the scheduling problem. An interval is characterized by a start value, an end value and a size. An important feature of interval variables is the fact that they can be optional; that is, one can decide not to consider them in the solution schedule. This concept is crucial in applications that present at least some of the following features:

- optional activities (operations, tasks) that can be left unperformed (with an impact on the cost); examples include externalized, maintenance or control tasks
- activities that can execute on a set of alternative resources (machines, manpower) with possibly different characteristics (speed, calendar) and compatibility constraints
- operations that can be processed in different temporal modes (for instance in series or in parallel)
- alternative modes for executing a given activity, each mode specifying a particular combination of resources
- alternative processes for executing a given production order, a process being specified as a sequence of operations requiring resources
- hierarchical description of a project as a work-breakdown structure with tasks decomposed into sub-tasks, part of the project being optional (with an impact on the cost if unperformed), and so forth.

Formally, an interval variable a is a variable whose domain $dom(a)$ is a subset of $\{\perp\} \cup \{[s, e) \mid s, e \in \mathbb{Z}, s \leq e\}$. An interval variable is said to be *fixed* if its domain is reduced to a singleton; that is, if \underline{a} denotes a fixed interval variable:

- interval is absent: $\underline{a} = \perp$; or
- interval is present: $\underline{a} = [s, e)$

Absent interval variables have special meaning. Informally speaking, an absent interval variable is not considered by any constraint or expression on interval variables it is involved in. For example, if an absent interval variable is used in a noOverlap constraint, the constraint will behave as if the interval was never specified to the constraint. If an absent interval variable a is used in a precedence constraint between interval variables a and b this constraint does not impact interval variable b . Each constraint specifies how it handles absent interval variables.

The semantics of constraints defined over interval variables is described by the properties that fixed intervals must have in order the constraint to be true. If a fixed interval \underline{a} is present and such that $\underline{a} = [s, e)$, we will denote $s(\underline{a})$ its integer start value s , $e(\underline{a})$ its integer end value e and $l(\underline{a})$ its positive integer length defined as $e(\underline{a}) - s(\underline{a})$. The presence status $x(\underline{a})$ will be equal to 1. For a fixed interval that is absent, $x(\underline{a}) = 0$ and the start, end and length are undefined.

Until a solution is found it may not be known whether an interval will be present or not. In this case we say that the interval is optional. To be precise, an interval is said to be absent when $dom(a) = \{\perp\}$, present when $\perp \notin dom(a)$ and optional in all other cases.

Intensity and size

Sometimes the intensity of “work” is not the same during the whole interval. For example let’s consider a worker who does not work during weekends (his work intensity during weekends is 0%) and on Friday he works only for half a day (his intensity during Friday is 50%). For this worker, 7 man-days of work will span for longer than just 7 days. In this example 7 man-days represents what we call the *size* of the interval; that is, what the length of the interval would be if the intensity function was always at 100%.

To model such situations, you can specify a range for the *size* of an interval variable and an integer *stepwise intensity function* F . For a fixed present interval \underline{a} the following relation will be enforced at any solution between the start, end, size sz of the interval and the integer granularity G (by default, the intensity function is expressed as a percentage so the granularity G is 100):

$$sz(\underline{a}) \leq \int_{s(\underline{a})}^{e(\underline{a})} \frac{F(t)}{G} dt - < sz(\underline{a}) + 1$$

That is, the length of the interval will be at least long enough to cover the work requirements given by the interval size, taking into account the intensity function. However, any over-estimation is always strictly less than one work unit.

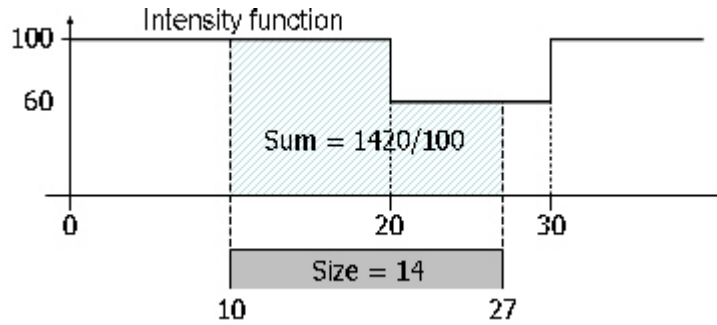
If no intensity is specified, it is supposed to be the constant full intensity function $\forall t, F(t) = 100\%$ so in that case $sz(a) = l(a)$. Note that the size is not defined for absent intervals.

Important:

The intensity step function F should be a stepwise function with integer values and is not allowed to exceed the granularity (100 by default).

The following figure depicts an interval variable of size 14 with its intensity function. A valid solution is represented where the interval starts at 10 and ends at 27. Indeed in this case:

$$\int_{s(a)}^{e(a)} \frac{F(t)}{G} dt = \frac{1420}{100} = 14.2$$



OPL formulation

Typically, the problem structure will indicate if an interval can be optional or not, and the keyword `optional` is used (or not) in the definition of the interval variable. In the case where the optionality depends on input data, you can specify a boolean parameter to the optionality field: `optional(true)` being equivalent to `optional` and `optional(false)` being equivalent to the omission of `optional`.

A window $[StartMin, EndMax]$ can be specified to restrict the position of the interval variable. By default, an interval variable will start after 0 and end before $maxint/2$. The fixed size or the size range for the interval is specified with the `size` keyword. Note that these bounds are taken into account only when the interval variable is present in the final schedule, that is, they allow specifying conditional bounds on the interval variable *would the interval be present in the final schedule*. For absent intervals, they are just ignored.

```
dvar interval a [optional[(IsOptional)]]
                [in StartMin..EndMax]
                [size SZ | in SZMin .. SZMax]
                [intensity F]
```

Where:

```
int IsOptional, StartMin, EndMax, SZ, SZMin, SZMax;
stepFunction F;
-maxint/2 + 1 <= StartMin <= maxint/2 - 1
```

```

-maxint/2 + 1 <= EndMax <= maxint/2 - 1
0 <= SZ <= maxint/2 - 1
0 <= SZMin <= maxint/2 - 1
0 <= SZMax <= maxint/2 - 1

```

Examples

For examples of using *interval*, see the CP keywords *interval*, *optional*, *size*, and *intensity* in the *OPL Language Quick Reference*.

Display of interval variable domain

The domain of an interval variable is displayed as shown in this example:

```
A1[0..1: 10..990 -- (5..10)5..990 --> 25..1000]
```

After the name of the interval variable (here *A1*), the first range (here *0..1*) represents the domain of the boolean presence status of the interval variable. Thus *0..1* represents an optional interval variable whose status has still not been fixed, *0* an absent interval variable and *1* a present interval variable.

The remaining fields describe the position of the interval variable, it is omitted if the interval variable is absent as this information is not relevant in this case. Thus, an absent interval variable is displayed as:

```
A1[0]
```

When the interval variable is possibly present:

- the first range in the remaining fields represents the domain of the interval start
- the second range (between parenthesis) represents the domain of the interval size
- the third range represents the domain of the interval length
- the fourth and last range represents the domain of the interval end

Note that the second range may be omitted in case the size and length of the interval variable are necessarily equal.

When the values are fixed, the ranges *min..max* are replaced by a single value. For instance, the following display represents a fixed interval variable of size 5 that is present, starts at 10 and ends at 35:

```
A1[1: 10 -- (5)25 --> 35]
```

Unary constraints on interval variables

CP Optimizer provides constraints for modeling restrictions that an interval cannot start, cannot end or cannot overlap a set of fixed dates.

Let \underline{a} denote a fixed interval and F an integer stepwise function.

- **Forbidden start.** The constraint *forbidStart*(\underline{a} , F), states that whenever the interval is present, it cannot start at a value t where $F(t) = 0$.
- **Forbidden end.** The constraint *forbidEnd*(\underline{a} , F), states that whenever the interval is present, it cannot end at a value t where $F(t - 1) = 0$.
- **Forbidden extent.** The constraint *forbidExtent*(\underline{a} , F), states that whenever the interval is present, it cannot overlap a point t where $F(t) = 0$.

More formally:

$$\begin{aligned}
\text{forbidStart}(\underline{a}, F) &\Leftrightarrow (\underline{a} = \perp) \vee (F(s(\underline{a})) \neq 0) \\
\text{forbidEnd}(\underline{a}, F) &\Leftrightarrow (\underline{a} = \perp) \vee (F(e(\underline{a}) - 1) \neq 0) \\
\text{forbidExtent}(\underline{a}, F) &\Leftrightarrow (\underline{a} = \perp) \vee (\forall t \in [s(\underline{a}), e(\underline{a})], F(t) \neq 0)
\end{aligned}$$

For syntax and examples of these constraints, see `forbidEnd`, `forbidExtent`, and `forbidStart` in the *OPL Language Quick Reference*. Note that none of these constraints can be used in meta-constraints.

Precedence constraints between interval variables

Precedence constraints are common scheduling constraints used to restrict the relative position of interval variables in a solution.

For example a precedence constraint can model the fact that an activity a must end before activity b starts (optionally with some minimum delay z). If one or both of the interval variables of the precedence constraint is absent, then the precedence is systematically considered to be true; therefore it does not impact the schedule.

More formally, the semantics of the relation $TC(\underline{a}, \underline{b}, z)$ on a pair of fixed intervals $\underline{a}, \underline{b}$ and for a value z depending on the constraint type TC is given in the following table.

Relation	Semantics
<i>startBeforeStart</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z \leq s(\underline{b})$
<i>startBeforeEnd</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z \leq e(\underline{b})$
<i>endBeforeStart</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z \leq s(\underline{b})$
<i>endBeforeEnd</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z \leq e(\underline{b})$
<i>startAtStart</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z = s(\underline{b})$
<i>startAtEnd</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow s(\underline{a}) + z = e(\underline{b})$
<i>endAtStart</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z = s(\underline{b})$
<i>endAtEnd</i>	$x(\underline{a}) \wedge x(\underline{b}) \Rightarrow e(\underline{a}) + z = e(\underline{b})$

Precedence relations semantics

For syntax and examples, see the following functions described in the *OPL Language Quick Reference*. Note that none of these constraints may be used in a meta-constraint.

- `endAtEnd`
- `endAtStart`
- `endBeforeEnd`
- `endBeforeStart`
- `startAtEnd`
- `startAtStart`
- `startBeforeEnd`
- `startBeforeStart`

Constraints on groups of interval variables

Describes constraints that act to encapsulate a group of intervals together.

The main purpose of a group constraint is to encapsulate a group of interval variables into one effective higher level interval.

Three “interval grouping” constraints are available: span, alternative, and synchronize.

Span constraint: The constraint $\text{span}(a, \{b_1, \dots, b_n\})$ states that the interval a spans over all present intervals from the set $\{b_1, \dots, b_n\}$. That is, interval a starts together with the first present interval from $\{b_1, \dots, b_n\}$ and ends together with the last one.

Alternative constraint: The constraint $\text{alternative}(a, \{b_1, \dots, b_n\})$ models an exclusive alternative between $\{b_1, \dots, b_n\}$. If interval a is present then exactly one of intervals $\{b_1, \dots, b_n\}$ is present and a starts and ends together with this chosen one. The alternative constraint can also be specified by a non-negative integer cardinality c , $\text{alternative}(a, \{b_1, \dots, b_n\}, c)$. In this case, it is not 1 but c interval variables that will be selected among the set $\{b_1, \dots, b_n\}$ and those c selected intervals will have to start and end together with interval variable a .

Synchronize constraint: The constraint $\text{synchronize}(a, \{b_1, \dots, b_n\})$ makes intervals $b_1 \dots b_n$ start and end together with interval a (if a is present).

More formally, let $\underline{a}, b_1, \dots, b_n$ be a set of fixed intervals and c a fixed integer.

The **span constraint** $\text{span}(\underline{a}, \{b_1, \dots, b_n\})$ holds if and only if:

$$\begin{aligned} \neg x(\underline{a}) &\Leftrightarrow \forall i \in [1, n], \neg x(b_i) \\ x(\underline{a}) &\Leftrightarrow \begin{cases} \exists i \in [1, n], x(b_i) \\ s(\underline{a}) = \min_{i \in [1, n], x(b_i)} s(b_i) \\ e(\underline{a}) = \max_{i \in [1, n], x(b_i)} e(b_i) \end{cases} \end{aligned}$$

The **alternative interval constraint** $\text{alternative}(\underline{a}, \{b_1, \dots, b_n\}, c)$ holds if and only if:

$$\begin{aligned} \neg x(\underline{a}) &\Leftrightarrow \forall i \in [1, n], \neg x(b_i) \\ x(\underline{a}) &\Leftrightarrow \exists K \subset [1, n] \begin{cases} K \neq \emptyset \wedge |K| = c \\ \forall k \in K, x(b_k) \wedge (s(\underline{a}) = s(b_k)) \wedge (e(\underline{a}) = e(b_k)) \\ \forall j \in [1, n] \setminus K, \neg x(b_j) \end{cases} \end{aligned}$$

When parameter c is omitted, we assume $c = 1$. Note that when $c \leq 0$, as there cannot be any subset $K \subset [1, n]$ such that $K \neq \emptyset$ and $|K| = c$, it means that necessarily, interval a is absent. And the contrapositive: if interval a is present, then necessarily, for the constraint to be satisfied it must be that $c \geq 1$.

The **synchronization constraint** $\text{synchronize}(\underline{a}, \{b_1, \dots, b_n\})$ holds if and only if:

$$x(\underline{a}) \Rightarrow \forall i \in [1, n] | x(b_i), (s(\underline{a}) = s(b_i)) \wedge (e(\underline{a}) = e(b_i))$$

Note that the alternative, span, and synchronize constraints cannot be used in meta-constraints.

For syntax and examples, see the functions as described in the *OPL Language Quick Reference*.

- alternative
- span
- synchronize

A logical constraint between interval variables: presenceOf

The presence constraint states that a certain interval must be present in the solution.

The semantics of the presence constraint on a fixed interval \underline{a} is simply: $\text{presenceOf}(\underline{a}) \Leftrightarrow x(\underline{a})$. The truth value of this constraint can be used in arithmetical expressions, and thereby restricted by logical constraints.

This constraint can be used in meta-constraints to indicate, for example, that there may be two optional intervals a and b ; if interval a is present then b must be present as well. This is modelled by the constraint $presenceOf(a) \rightarrow presenceOf(b)$.

OPL formulation

The constraint to express that the interval variable must be present:

```
presenceOf(a);
```

where:

```
dvar interval a;
```

For an example of the *presenceOf* constraint see *presenceOf* in the *OPL Language Quick Reference*.

Expressions on interval variables

Integer and numerical expressions are available to access or evaluate different attributes of an interval variable.

These expressions can be used, for example, to define a term for the cost function or to connect interval variables to integer and floating point expressions.

The integer expressions are *startOf*, *endOf*, *lengthOf*, and *sizeOf* and they provide access to the different attributes of an interval variable. Special care must be taken for optional intervals, as an integer value *dval* must be specified which represents the value of the expression when the interval is absent. If this value is omitted, it is supposed to be 0. For the syntax of integer expressions, see *endOf*, *lengthOf*, *sizeOf*, and *startOf* in the *OPL Language Quick Reference*.

The numerical expressions are *startEval*, *endEval*, *lengthEval*, and *sizeEval*, and they allow evaluation of a piecewise linear function on a given bound of an interval. As with integer expressions, in the case of optional intervals an integer value *dval* must be specified which represents the value of the expression when the interval is absent. If this value is omitted, it is supposed to be 0. For the syntax and examples of the use of a numerical expression, see *endEval*, *lengthEval*, *sizeEval*, and *startEval* in the *OPL Language Quick Reference*.

Let \underline{a} denote a fixed interval variable. The semantics of these expressions is shown in the table.

Expression	Semantics
$startOf(\underline{a}, dval)$	$expr = \begin{cases} s(\underline{a}) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$endOf(\underline{a}, dval)$	$expr = \begin{cases} e(\underline{a}) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$lengthOf(\underline{a}, dval)$	$expr = \begin{cases} l(\underline{a}) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$sizeOf(\underline{a}, dval)$	$expr = \begin{cases} sz(\underline{a}) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$startEval(\underline{a}, F, dval)$	$expr = \begin{cases} F(s(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$endEval(\underline{a}, F, dval)$	$expr = \begin{cases} F(e(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$lengthEval(\underline{a}, F, dval)$	$expr = \begin{cases} F(l(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$
$sizeEval(\underline{a}, F, dval)$	$expr = \begin{cases} F(sz(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases}$

Expression semantics

Sequencing of interval variables

Describes a basic building block of scheduling, the *interval sequence*.

$\subset \pi \forall$

An *interval sequence variable* is defined on a set of interval variables A . Informally speaking, the value of an interval sequence variable represents a total ordering of the interval variables of A . Note that any absent interval variables are not considered in the ordering.

More formally, an interval sequence variable p on a set interval variables A represents a decision variable whose possible values are all the permutations of the intervals of A . Let \underline{A} be a set of fixed intervals and n denote the cardinality of \underline{A} .

A permutation π of \underline{A} is a function $\pi : \underline{A} \rightarrow \{\perp\} \cup [1, n]$ such that if we denote $length(\pi) = |\{\underline{a} \in \underline{A}, x(\underline{a})\}|$ then the number of present intervals:

1. $\forall \underline{a} \in \underline{A}, x(\underline{a}) \Leftrightarrow (\pi(\underline{a}) \neq \perp)$
2. $\forall \underline{a} \in \underline{A}, x(\underline{a}) \Rightarrow (\pi(\underline{a}) \leq length(\pi))$
3. $\forall \underline{a}, \underline{b} \in \underline{A}, (x(\underline{a}) \wedge x(\underline{b}) \wedge \underline{a} \neq \underline{b}) \Rightarrow (\pi(\underline{a}) \neq \pi(\underline{b}))$

Note that the sequence alone does not enforce any constraint on the relative position of intervals end-points. For instance, an interval variable a could be sequenced before an interval variable b in a sequence p without any impact on the relative position between the start/end points of a and b (a could still be fixed to start after the end of b). This is because different semantics can be used to define

how a sequence constrains the positions of intervals. We will see later how the *noOverlap* constraint implements one of these possible semantics.

The sequence variable also allows associating a fixed non-negative integer *type* with each interval variable in the sequence. In particular, these integers are used by the *noOverlap* constraint. $T(p, a)$ denotes the fixed non-negative integer type of interval variable a in the sequence variable p .

Constraints on sequence variables

The following constraints are available on sequence variables:

- $\text{first}(\text{scheduling})(p, a)$ states that if interval a is present, then it will be the first interval of the sequence p .
- $\text{last}(\text{scheduling})(p, a)$ states that if interval a is present, then it will be the last interval of the sequence p .
- $\text{before}(p, a, b)$ states that if both intervals a and b are present then a will appear before b in the sequence p .
- $\text{prev}(\text{scheduling})(p, a, b)$ states that if both intervals a and b are present then a will be just before b in the sequence p , that is, it will appear before b and no other interval will be sequenced between a and b in the sequence.

The formal semantics of these basic constraints is shown in the following table.

Relation	Semantics
$\text{first}(\pi, \underline{a})$	$x(\underline{a}) \Rightarrow (\pi(\underline{a}) = 1)$
$\text{last}(\pi, \underline{a})$	$x(\underline{a}) \Rightarrow (\pi(\underline{a}) = \text{length}(s))$
$\text{before}(\pi, \underline{a}, \underline{b})$	$(x(\underline{a}) \wedge x(\underline{b})) \Rightarrow (\pi(\underline{a}) \leq \pi(\underline{b}))$
$\text{prev}(\pi, \underline{a}, \underline{b})$	$(x(\underline{a}) \wedge x(\underline{b})) \Rightarrow (\pi(\underline{b}) = \pi(\underline{a}) + 1)$

Sequence relation semantics

The no overlap constraint

The *no overlap* constraint on an interval sequence variable p states that the sequence defines a chain of non-overlapping intervals, any interval in the chain being constrained to end before the start of the next interval in the chain. This constraint is typically useful for modelling disjunctive resources.

More formally, the condition for a permutation value π for sequence p to satisfy the *noOverlap* constraints is defined as:

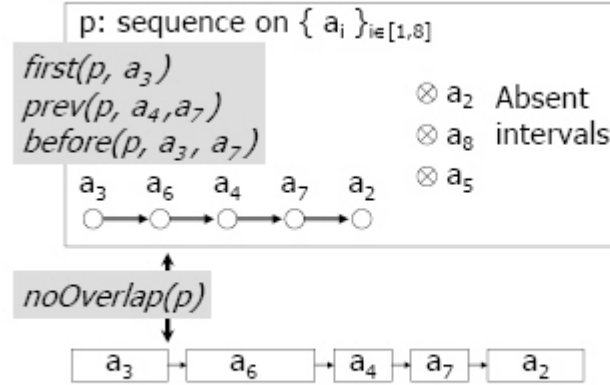
$$\text{noOverlap}(\pi) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A}, \neg x(\underline{a}) \vee \neg x(\underline{b}) \vee ((\pi(\underline{a}) < \pi(\underline{b})) \Leftrightarrow (e(\underline{a}) \leq s(\underline{b})))$$

If a transition distance matrix M is specified, it defines the minimal non-negative distance that must separate two consecutive intervals in the sequence.

More formally, if $T(p, a)$ denotes the non-negative integer type of interval a in the sequence variable p :

$$noOverlap(\pi, M) \Leftrightarrow \forall \underline{a}, \underline{b} \in \underline{A}, \neg x(\underline{a}) \vee \neg x(\underline{b}) \vee ((\pi(\underline{a}) < \pi(\underline{b})) \Leftrightarrow (e(\underline{a}) + M[T(\pi, \underline{a}), T(\pi, \underline{b})] \leq s(\underline{b})))$$

A sequence variable together with a no-overlap constraint using it are illustrated in this figure:



Example of sequence variable and no-overlap constraint

The sameSequence and sameCommonSubsequence constraints

The sameSequence and sameCommonSubsequence constraints are binary constraints on a pair of sequence variables p_1 and p_2 . These constraints state that the relative position of some related subsets of interval variables is the same in both sequences. Typically, if a is before b in sequence p_1 then the interval related to a is before the interval related to b in sequence p_2 . Here are two examples of use-cases for these constraints.

- First in/first out and no-bypass constraints. In some physical systems, like trains on a single line railway, or items on a conveyor belt, bypassing is not possible and items must enter and exit a given section of the system in the same order. If entering and exiting the sections or the junctions of the system is modeled by two related interval variables, those constraints can be modeled by sameSequence constraints (or sameCommonSubsequence constraints if the items do not follow the same path) on the sequences of entering and exiting intervals. A classic example of such a constraint is the permutation flow-shop scheduling problem.
- Scenario-based approaches for scheduling with uncertainties. In presence of uncertainties (for instance, uncertain activity durations) one may be interested in building sequences of activities on unary resources that optimize some robustness or statistical criterion (for instance the expected makespan). A scenario is a sub-model that defines a particular realization of the uncertainties in the environment. As one must find robust sequences that optimize some criterion over all scenarios, the different sequences of a given unary resource across all scenarios are linked with sameSequence constraints.

A constraint $sameCommonSubsequence(p_1, p_2, B_1, B_2)$ is defined on two sequence variables p_1 (defined on a set of interval variables A_1) and p_2 (defined on a set of interval variables A_2), with two arrays of interval variables B_1 and B_2 of identical size and such that $B_1 \subset A_1, B_2 \subset A_2$. The constraint states that the sub-sequences

defined by p_1 and p_2 by only considering the pairs of present intervals $(B_1[i], B_2[i])$ are identical modulo the mapping between intervals $B_1[i]$ and $B_2[i]$.

For instance, let's suppose sequence variable p_1 defined on interval variables $\{a,b,c,d,e,f\}$ and sequence variable p_2 defined on interval variables $\{u,v,w,x,y,z\}$ with a constraint $\text{sameCommonSubsequence}(p_1, p_2, [a,c,d,e,f], [u,w,v,x,y])$. Assuming that, in a solution, interval variables d and y are absent while all the other interval variables are present. The pair of permutations $\pi_1=(c,f,a,e,b)$ and $\pi_2=(w,v,u,x)$ satisfies the $\text{sameCommonSubsequence}$ constraints. Indeed if one considers only pairs of present intervals, the mapping B_1, B_2 maps $[a,c,e]$ with $[u,w,x]$. Sub-sequence of π_1 over $\{a,c,e\}$ is (c,a,e) , sub-sequence of π_2 over $\{u,w,x\}$ is (w,u,x) and one can see that the two sub-sequences are the same modulo the mapping.

More formally, let n denote the size of both interval variable arrays B_1 and B_2 . The condition for permutations π_1 and π_2 of sequencing variables p_1 and p_2 to satisfy the constraint $\text{sameCommonSubsequence}(\pi_1, \pi_2, \underline{B_1}, \underline{B_2})$ is defined as:

$$\forall i, j \in [1, n], i \neq j, x(\underline{B_1}[i]), x(\underline{B_1}[j]), x(\underline{B_2}[i]), x(\underline{B_2}[j]), (\pi_1(\underline{B_1}[i]) < \pi_1(\underline{B_1}[j]) \Leftrightarrow \pi_2(\underline{B_2}[i]) < \pi_2(\underline{B_2}[j]))$$

A constraint $\text{sameSequence}(p_1, p_2, B_1, B_2)$ is defined on two sequence variables of identical size p_1 (defined on a set of interval variables A_1) and p_2 (defined on a set of interval variables A_2) with two arrays of interval variables B_1 and B_2 that are an ordering of sets A_1 and A_2 . The constraint states that sequences p_1 and p_2 are identical modulo a mapping between intervals $B_1[i]$ and $B_2[i]$. A sameSequence constraint is stronger than a $\text{sameCommonSubsequence}$ constraint; it additionally imposes that arrays B_1 and B_2 contain all the definition intervals of their relative sequence variables and that the presence status of mapped interval variables $B_1[i]$ and $B_2[i]$ is the same.

More formally, let n denote the size of both sequence variables. The condition for permutations π_1 and π_2 of the sequence variables p_1 and p_2 to satisfy constraint $\text{sameSequence}(\pi_1, \pi_2, \underline{B_1}, \underline{B_2})$ is defined as:

$$\forall i \in [1, n], \pi_1(\underline{B_1}[i]) = \pi_2(\underline{B_2}[i])$$

When parameters B_1 and B_2 are omitted in the definition of the above constraints, we assume B_1 and B_2 are the definition intervals of p_1 and p_2 , following the definition order of interval variables in the sequence.

Syntax and examples

For syntax and examples of use of the sequence interval variable, see sequence in the *OPL Language Quick Reference*.

For the syntax and examples of use of the no overlap constraint, which needs to be defined as a set of integer triples, see noOverlap in the *OPL Language Quick Reference*.

For the syntax and examples of the other constraints available on an interval sequence variable, see first, last (scheduling), prev (scheduling), before, sameSequence, and sameCommonSubsequence in the *OPL Language Quick Reference*. (Note that there are similarly-named constraints available for set operations in OPL.)

Note that none of the constraints mentioned in this section can be used in a meta-constraint.

Sequence variables can be viewed in a Gantt chart in the CPLEX Studio IDE. A simple example is given in Viewing the results of scheduling problems.

Cumulative functions

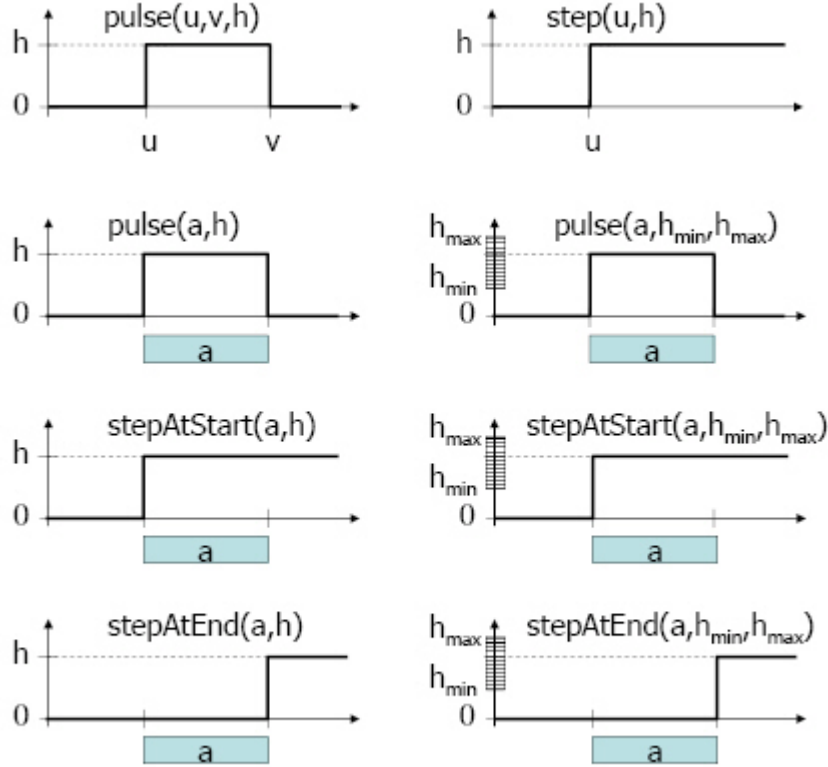
Describes the cumulative function.

In scheduling problems involving cumulative resources (also known as renewable resources), the cumulated usage of the resource by the activities is usually represented by a function of time. An activity usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time (pulse function). For resources that can be produced and consumed by activities (for instance the content of an inventory or a tank), the resource level can also be described as a function of time; production activities will increase the resource level whereas consuming activities will decrease it. In these type of problems, the cumulated contribution of activities on the resource can be represented by a function of time and constraints can be modeled on this function, for instance a maximal or a safety level.

CP Optimizer introduces the notion of the *cumulative function expression*, which is a function that represents the sum of individual contributions of intervals. A panel of elementary cumul function expressions is available to describe the individual contribution of an interval variable (or a fixed interval of time) which cover the main use-cases mentioned previously: *pulse* for usage of a cumulative resource, *step* for resource production/consumption. When the elementary cumulative function expressions that define a cumul function expression are fixed (and thus, so are their related intervals), the expression is fixed. CP Optimizer provides several constraints over cumul function expressions. These constraints allow restricting the possible values of the function over the complete horizon or over some fixed or variable interval. For applications where the actual quantity of resource that is used, produced or consumed by intervals is an unknown of the problem, expressions are available for constraining these quantities.

Cumul function expressions

Let \mathcal{F}^+ denote the set of all functions from \mathbb{Z} to \mathbb{Z}^+ . A cumul function expression f is an expression whose value is a function of \mathcal{F}^+ and thus, whose domain $\text{dom}(f)$ is a subset of \mathcal{F}^+ . Let $u, v \in \mathbb{Z}$ and $h, h_{\min}, h_{\max} \in \mathbb{Z}^+$ and a be an interval variable, we consider the following elementary cumul function expressions illustrated in the following figure: *pulse*(u, v, h), *step*(u, h), *pulse*(a, h), *pulse*(a, h_{\min}, h_{\max}), *stepAtStart*(a, h), *stepAtStart*(a, h_{\min}, h_{\max}), *stepAtEnd*(a, h), and *stepAtEnd*(a, h_{\min}, h_{\max}).



Elementary cumul function expressions

More formally, let $u, v \in \mathbb{Z}$ and $h \in \mathbb{Z}^+$ and we define the following particular functions of \mathcal{F}^+

- $\underline{0}$ is the null function that is, the function F such that $\forall t \in \mathbb{Z}, F(t) = 0$
- $\underline{pulse}(u, v, h)$ is the function F such that $F(t) = h$ if $t \in [u, v]$, $F(t) = 0$ otherwise
- $\underline{step}(u, h) = \underline{pulse}(u, +\infty, h)$

The semantics of the elementary function expressions is listed in the following table, together with the formal definition of their domain. The function set $\underline{0}_a$ is equal to the singleton $\{\underline{0}\}$ if $\perp \in \text{dom}(a)$; that is, if interval variable a is possibly absent, and equal to the empty set otherwise.

Function expression	Domain
$pulse(u, v, h)$	$\{ pulse(u, v, h) \}$ (singleton)
$step(u, h)$	$\{ step(u, h) \}$ (singleton)
$pulse(a, h)$	$0_a \cup \{ pulse(s(\underline{a}), e(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp \}$
$pulse(a, h_{min}, h_{max})$	$0_a \cup \{ pulse(s(\underline{a}), e(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp \}$
$stepAtStart(a, h)$	$0_a \cup \{ step(s(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp \}$
$stepAtStart(a, h_{min}, h_{max})$	$0_a \cup \{ step(s(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp \}$
$stepAtEnd(a, h)$	$0_a \cup \{ step(e(\underline{a}), h) \mid \underline{a} \in dom(a) \setminus \perp \}$
$stepAtEnd(a, h_{min}, h_{max})$	$0_a \cup \{ step(e(\underline{a}), h) \mid h \in [h_{min}, h_{max}] \wedge \underline{a} \in dom(a) \setminus \perp \}$

Elementary cumul function expressions

A cumul function expression f is an expression built as the algebraical sum of the elementary function expressions in the table (or negations). More formally, it is a construct of the form $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$ and f_i is an elementary function expression.

When all elementary function expressions f_i related with a cumul function expression f are fixed (that is, their domain is reduced to a singleton \underline{f}_i), the value of the cumul function expression f is fixed and equal to the function $\underline{f} = \sum_i \epsilon_i \cdot \underline{f}_i$.

Constraints on cumul function expressions

The following constraints can be expressed on a cumul function expression f . Let

$u, v \in \mathbb{Z}$ and $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and a be an interval variable:

- $alwaysIn(f, u, v, h_{min}, h_{max})$ means that the values of function f must remain in the range $[h_{min}, h_{max}]$ everywhere on the interval $[u, v)$.
- $alwaysIn(f, a, h_{min}, h_{max})$ means that if interval a is present, the values of function f must remain in the range $[h_{min}, h_{max}]$ between the start and the end of interval variable a .
- $f \leq h_{max}$ means that function f cannot take values greater than h_{max} . It is semantically equivalent to $alwaysIn(f, -\infty, +\infty, 0, h_{max})$.
- $f \geq h_{min}$ means that function f cannot take values lower than h_{min} . It is semantically equivalent to $alwaysIn(f, -\infty, +\infty, h_{min}, +\infty)$

More formally:

$$\begin{aligned}
 alwaysIn(\underline{f}, u, v, h_{min}, h_{max}) &\Leftrightarrow \forall t \in [u, v), \underline{f}(t) \in [h_{min}, h_{max}] \\
 alwaysIn(\underline{f}, \underline{a}, h_{min}, h_{max}) &\Leftrightarrow \forall t \in [s(\underline{a}), e(\underline{a})], \underline{f}(t) \in [h_{min}, h_{max}]
 \end{aligned}$$

Expressions on cumulative functions

The following elementary cumul function expressions are based on an interval variable a : $pulse(a, h)$, $pulse(a, h_{min}, h_{max})$, $stepAtStart(a, h)$, $stepAtStart(a, h_{min}, h_{max})$, $stepAtEnd(a, h)$, and $stepAtEnd(a, h_{min}, h_{max})$.

Some of these expressions define a range $[h_{min}, h_{max}]$ of possible values for the actual height of the function when the interval variable a is present. The actual height is an unknown of the problem. CP Optimizer provides some integer expressions to control this height. These expressions are based on the notion of the contribution of a given interval variable a to the (possibly composite) cumul function expression f . This contribution is defined as the sum of all the elementary

cumul function expressions based on a in f . This contribution is a discrete function that can change value only at the start and at the end of interval a and is equal to 0 before the start of a .

For instance, let a and b be two interval variables and a cumul function expression f defined by: $f = \text{pulse}(a, 3) + \text{pulse}(a, 2) - \text{stepAtEnd}(a, 1) + \text{stepAtStart}(b, 2) - \text{stepAtEnd}(b, 3)$. The contribution of a to f is the function $\text{pulse}(a, 3) + \text{pulse}(a, 2) - \text{stepAtEnd}(a, 1)$ and the contribution of b to f is the function $\text{stepAtStart}(b, 2) - \text{stepAtEnd}(b, 3)$.

If interval a is present, the expression $\text{heightAtStart}(a, f)$ returns the value of the contribution of a to f evaluated at the start of a that is, it measures the contribution of interval a to cumul function expression f at its start point. Similarly, the expression $\text{heightAtEnd}(a, f)$ returns the value of the contribution of a to f evaluated at the end of a that is, it measures the contribution of interval a to cumul function expression f at its end point. An additional integer value $dval$ can be specified at the construction of the expression, which will be the value returned by the expression when the interval is absent. Otherwise, if no value is specified, the expression will be equal to 0 when the interval is absent.

In the previous example, assuming both interval a and b to be present we would get: $\text{heightAtStart}(a, f) = 5$, $\text{heightAtEnd}(a, f) = 4$, $\text{heightAtStart}(b, f) = 2$, $\text{heightAtEnd}(b, f) = -1$. Of course, in general when using ranges for the height of elementary cumul function expressions, the heightAtStart/End expressions will not be fixed until all the heights have been fixed by the search.

More formally, if an elementary cumul function expression f_i is based on an interval variable, we denote $\text{ivar}(f_i)$ this interval variable. Let f be a cumul function expression defined as $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$ where f_i are elementary cumul function expressions. The contribution of an interval variable a to f is defined as $f|_a = \sum_{i | \text{ivar}(f_i)=a} \epsilon_i \cdot f_i$ and the expressions heightAtStart/End are defined as follows:

$$\begin{aligned} \bullet \text{ heightAtStart}(\underline{a}, \underline{f}, dval) &= \begin{cases} \underline{f}|_{\underline{a}}(s(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases} \\ \bullet \text{ heightAtEnd}(\underline{a}, \underline{f}, dval) &= \begin{cases} \underline{f}|_{\underline{a}}(e(\underline{a})) & \text{if } x(\underline{a}) \\ dval & \text{otherwise} \end{cases} \end{aligned}$$

Syntax and examples

For the syntax and examples of use of a cumulative function see `cumulFunction`, `pulse`, `step`, `stepAtEnd`, and `stepAtStart` in the *OPL Language Quick Reference*.

The results of a solved scheduling model can be viewed in the CPLEX Studio IDE. An example of results containing a cumulative function is provided in Viewing the results of cumulative functions in the IDE.

These are the constraints available on cumulative function expressions:

- $f \leq hmax$;
- $hmin \leq f$;
- `alwaysIn(f, u, v, hmin, hmax)`
- `alwaysIn(f, a, hmin, hmax)`

Note that these constraints cannot be used in meta-constraints.

The following expressions are available on cumulative functions:

- `dexpr int h = heightAtStart(a, f[,dval]);`
- `dexpr int h =heightAtEnd(a, f[,dval]);`

More information on these constraints and expressions is available in the *OPL Language Quick Reference*.

State functions

Describes the state function.

Some scheduling problems involve reasoning with resources whose state may change over time. The state of the resource can change because of the scheduled activities or because of exogenous events; yet some activities in the schedule may need a particular condition on the resource state to be true in order to execute. For instance, the temperature of an oven may change due to an activity that sets the oven temperature to a value v , and a cooking activity may follow that requires the oven temperature to start at and maintain a temperature level v' throughout its execution. Furthermore, the transition between two states is not always instantaneous and a transition time may be needed for the resource to switch from a state v to a state v' .

CP Optimizer introduces the notion of *state function* which is used to describe the evolution of a given feature of the environment. The possible evolution of this feature is constrained by interval variables of the problem. The main difference between state functions and cumulative functions is that interval variables have an incremental effect on cumul functions (increasing or decreasing the function value) whereas they have an absolute effect on state functions (requiring the function value to be equal to a particular state or in a set of possible states).

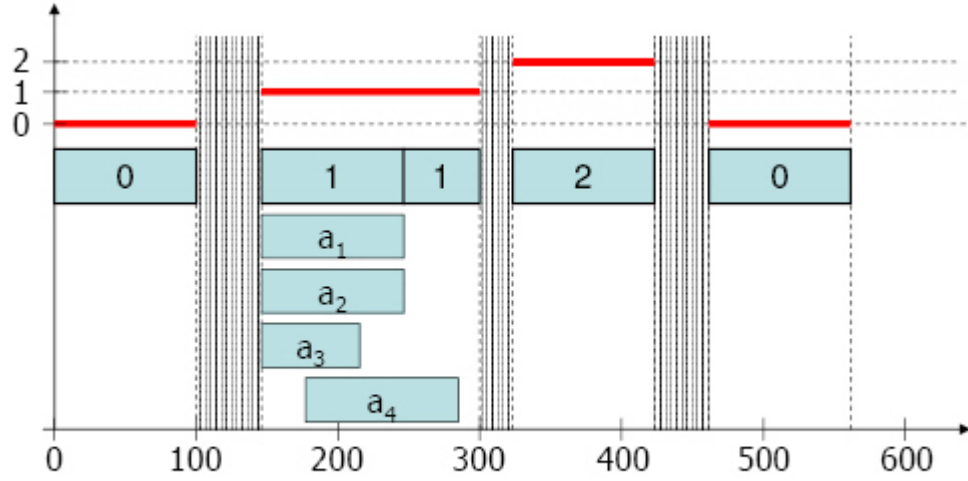
Informally speaking, a state function is a set of non-overlapping intervals over which the function maintains a particular non-negative integer state. In between those intervals, the state of the function is not defined, typically because of an ongoing transition between two states. For instance for an oven with three possible temperature levels identified by indexes 0, 1 and 2 we could have:

- [start=0, end=100): state=0,
- [start=150, end=250): state=1,
- [start=250, end=300): state=1,
- [start=320, end=420): state=2,
- [start=460, end=560): state=0, ...

Constraints are available to restrict the evolution of a state function. These constraints allow you to specify:

- That the state of the function must be defined and should remain equal to a given non-negative state everywhere over a given fixed or variable interval (*alwaysEqual*).
- That the state of the function must be defined and should remain constant (no matter its value) everywhere over a given fixed or variable interval (*alwaysConstant*).
- That intervals requiring the state of the function to be defined cannot overlap a given fixed or variable interval (*alwaysNoState*).
- That everywhere over a given fixed or variable interval, the state of the function, if defined, must remain within a given range of non-negative states $[v_{\min}, v_{\max}]$ (*alwaysIn*).

Additionally, the two first constraints can be complemented to specify that the given fixed or variable interval should have its start and/or end point synchronized with the start and/or end point of the interval of the state function that maintains the required state. This is the notion of *start* and *end alignment* which is particularly useful for modelling parallel batches. For instance in the oven example above, all interval variables that would require an oven temperature of level 1 and specify a start and end alignment, if executed over the interval [150, 250) would have to start exactly at 150 and end at 250. This is depicted in the following figure where a_1 and a_2 are two start and end aligned interval variables, a_3 is start aligned only and a_4 is not aligned at all.



State function and interval variable alignments

State functions and transition distance

A state function f is a decision variable whose value is a set of non-overlapping intervals, each interval $[s_i, e_i)$ being associated a non-negative integer value v_i that represents the state of the function over the interval.

Let \underline{f} be a fixed state function, we will denote $\underline{f} = ([s_i, e_i) : v_i)_{i \in [1, n]}$:

- $\forall i \in [1, n], s_i \in \mathbb{Z}, e_i \in \mathbb{Z}, v_i \in \mathbb{Z}^+, s_i < e_i$
- $\forall i \in [1, n-1], e_i < s_{i+1}$

We denote $D(\underline{f}) = \cup_{i \in [1, n]} [s_i, e_i)$ the definition domain of \underline{f} , that is, the set of points where the state function is associated a state.

For a fixed state function \underline{f} and a point $t \in D(\underline{f})$, we will denote $[s(\underline{f}, t), e(\underline{f}, t))$ the unique interval of the function that contains t and $\underline{f}(t)$ the value of this interval:

$$\begin{aligned} [s(\underline{f}, t), e(\underline{f}, t)) &= [s_i, e_i), t \in [s_i, e_i) \\ \underline{f}(t) &= v_i, t \in [s_i, e_i) \end{aligned}$$

For instance, in the example of the oven introduced previously, we would have $\underline{f}(200) = 1$, $s(\underline{f}, 200) = 150$ and $e(\underline{f}, 200) = 250$.

A state function can be associated with a *transition distance*. The transition distance defines the minimal distance that must separate two consecutive states in the state function. More formally, if $M[v, v']$ represents a transition distance matrix between state v and state v' , the transition distance means that:

$$\forall i \in [1, n - 1], e_i + M[v_i, v_{i+1}] \leq s_{i+1}$$

The transition distance matrix M must satisfy the triangular inequality. For instance, in the example of the oven, the state function depicted in the previous figure is consistent with the following transition distance:

$$M = \begin{pmatrix} 0 & 50 & 60 \\ 20 & 0 & 20 \\ 40 & 30 & 10 \end{pmatrix}$$

Constraints on state functions

If f is a state function, a an interval variable, v, v_{\min}, v_{\max} non-negative integers and aln_s, aln_e two boolean values:

- The constraint $alwaysEqual(f, a, v, aln_s, aln_e)$ specifies that whenever a is present, state function f must be defined everywhere between the start and the end of interval a and be constant and equal to non-negative value v over this interval. If aln_s is true, it means that interval a is start-aligned with f : Interval a must start at the beginning of the interval where f is maintained in state s . If aln_e is true, it means that interval a is end-aligned with f : Interval a must end at the end of the interval where f is maintained in state s . More formally:

$$\begin{aligned} & - \forall t \in [s(\underline{a}), e(\underline{a})], t \in D(\underline{f}) \wedge \underline{f}(t) = v \\ & - aln_s \Rightarrow s(\underline{a}) = s(\underline{f}, s(\underline{a})) \\ & - aln_e \Rightarrow e(\underline{a}) = e(\underline{f}, e(\underline{a})) \end{aligned}$$

- The constraint $alwaysConstant(f, a, aln_s, aln_e)$ specifies that whenever a is present, state function f must be defined everywhere between the start and the end of interval a and be constant over this interval. More formally:

$$\exists v \in Z^+, alwaysEqual(f, a, v, aln_s, aln_e)$$

- The constraint $alwaysNoState(f, a)$ specifies that whenever a is present, state function f cannot provide any valid state between the start and the end of interval a . As a consequence, any interval constrained with $alwaysEqual$ or $alwaysConstant$ on this function and thus requiring the function to be defined cannot overlap interval a . Formally: $[s(\underline{a}), e(\underline{a})) \cap D(\underline{f}) = \emptyset$

- The constraint $alwaysIn(f, a, v_{\min}, v_{\max})$ specifies that whenever a is present, everywhere between the start and the end of interval a the state of function f , if defined, must belong to the range $[v_{\min}, v_{\max}]$ where $0 \leq v_{\min} \leq v_{\max}$. Formally:

$$\forall t \in [s(\underline{a}), e(\underline{a})) \cap D(\underline{f}), \underline{f}(t) \in [v_{\min}, v_{\max}]$$

Syntax and examples

For the syntax and examples of use of a state function see `stateFunction`.

The results of a solved scheduling model can be viewed in the CPLEX Studio IDE. An example of results containing a state function is provided in Viewing the results of state functions in a Gantt chart.

Another example featuring a transition and the *alwaysEqual* constraint is shown in the following list.

The following list includes the constraints available on a state function. A full description and example for each syntax is available in the *OPL Language Quick Reference*.

- `alwaysEqual(f,s,e,v[,aligns,aligne]);`
- `alwaysEqual(f,a,v[,aligns,aligne]);`
- `alwaysConstant(f,s,e[,aligns,aligne]);`
- `alwaysConstant(f,a[,aligns,aligne]);`
- `alwaysNoState(f,s,e);`
- `alwaysNoState(f,a);`
- `alwaysIn(f,u,v,hmin,hmax);`
- `alwaysIn(f,a,hmin,hmax);`

Note that these constraints cannot be used in meta-constraints.

Example with stateFunction, transition, and alwaysEqual.

A machine can be equipped with a tool among a set of n possible tools. Each operation o executed on the machine needs a specific tool *RequiredTool*[o]. The machine can process several operation simultaneously provided these operations are compatible with the tool currently installed on the machine. Changing the tool installed on the machine needs some constant set-up time which is supposed to be independent from the tools.

```
int nbTools = ...;
int nbOps = ...;
int setupTime = ...;
range Tools = 1..nbTools;
range Operations = 1..nbOps;
int Duration [Operations] = ...;
int RequiredTool [Operations] = ...;

dvar interval op[o in Operations] size Duration[o];

tuple triplet { int t11; int t12; int value; };
{ triplet } Transition = { <t11,t12,setupTime> } t11, t12 in Tools };
stateFunction machineTool with Transition;

constraints {
  forall(o in Operations) {
    alwaysEqual(machineTool, op[o], RequiredTool[o]);
  }
}
```

Notation

The main types of notation used throughout the scheduling section are defined here.

Vectors are denoted by capital letters, for example Y . The size of a vector Y is denoted $|Y|$. If $n = |Y|$, the vector is denoted $Y = (y_1, \dots, y_n)$.

Notation	Type	Description
a, b	interval variable	
$x(a)$	boolean expression	execution status of a
$s(a), e(a), l(a)$	integer expression	start, end, length of a
$sz(a)$	integer expression	size of a
A, B	vector of interval variables	
p	interval sequence variable	
f, g, h	cumul function expression state function	
$\underline{a}, \underline{b}$	fixed interval	
$x(\underline{a})$	boolean	presence status of \underline{a}
$s(\underline{a}), e(\underline{a}), l(\underline{a})$	integer	start, end, length of \underline{a}
$sz(\underline{a})$	integer	size of \underline{a}
π	interval variable permutation	
z	integer (expression)	delay of a precedence
t	integer	date
T	vector of integers	array of dates
S	vector of floats	array of slopes
v, w	integer or float	miscellaneous integer or float value
V, W	vector of integers or floats	vector of miscellaneous values
$align$	boolean	alignment flag
F, G, H	piecewise linear function $\mathbb{R} \rightarrow \mathbb{R}$ vector of cumul function expressions	
$T(p, a)$	integer	type of interval a in sequence p
M	integer matrix	transition distance matrix
\mathcal{F}^+	set of functions	set of all functions $f : \mathbb{Z} \rightarrow \mathbb{Z}^+$

Chapter 3. IBM ILOG Script for OPL

Describes the structure and built-in values and functions of the scripting language.

Language structure

Presents the structure of the IBM ILOG Script language for OPL: the language constructs, the elements from which expressions can be constructed, and the possible types of statement.

Syntax

What composes a scripting statement, compound statements, comments, identifiers.

General

Provides a general overview of OPL syntax.

A script comprises a sequence of statements. An expression can also be used whenever a statement is expected, in which case its value is ignored and only its side effect is taken into account.

You can put multiple statements or expressions on a single line if you separate them with a semi-colon (;), for example, the following two scripts are equivalent:

Script1

```
writeln("Hello, world")
x = x+1
if (x > 10) writeln("Too big")
```

Script2

```
writeln("Hello, world"); x = x+1; if (x > 10) writeln("Too big")
```

Compound statements

Explains the use of compound statements in OPL syntax.

A compound statement is a sequence of statements and expressions enclosed in curly brackets ({}). It can be used to perform multiple tasks whenever a single statement is expected, for example, in the following conditional statement, the three statements and expressions in curly brackets are executed when the condition $a > b$ is true:

```
if (a > b) {
  var c = a
  a = b
  b = c
}
```

The last statement or expression before a closing curly bracket does not need to be followed by a semicolon, even if it is on the same line. For example, the following program is syntactically correct and is equivalent to the previous one:

```
if (a > b) { var c = a; a = b; b = c }
```

Comments

Explains the syntax of comments in OPL.

Script supports two different styles of comments:

- **Single line comments:** A single line comment starts with `//` and stops at the end of the line.

Example:

```
x = x+1 // Increment x,  
y = y-1 // then decrement y.
```

- **Multiple line comments:** To span on more than one line, comments must start with a `/*` and ends with a `*/`; Nested multiple line comments are **not** allowed.

Example:

```
/* The following statement  
increments x. */  
x = x+1  
/* The following statement  
decrements y. */  
y = y /* A comment can be inserted here */ -1
```

Identifiers

Shows how to use identifiers in OPL syntax.

Identifiers are used to name script variables and functions. An identifier starts with either a letter or an underscore, and is followed by a sequence of letters, digits, and underscores.

These are examples of identifiers:

```
car  
x12  
main_window  
_foo
```

The language is case-sensitive, so that the uppercase letters A-Z are distinct from the lowercase letters a-z. For example, the identifiers `car` and `Car` are distinct.

Expressions in IBM ILOG Script

Expressions are a combination of literals, script variables, special keywords, and operators.

Literals

Explains the use of literals in IBM ILOG Script.

Literals can represent the following elements:

- Numbers, for example: `12` `14.5` `1.7e-100`
- Strings, for example, `"Ford"` `"Hello world\n"`
- Booleans, either `true` or `false`
- The null value: `null`.

For further details about number and string literal syntax, see “Numbers” on page 99 and “IBM ILOG Script strings” on page 106.

Operators

Explains the use of and precedence of operators in IBM ILOG Script.

The precedence of operators determines the order in which they are applied when an expression is evaluated. You can override operator precedence using parentheses.

For a list of IBM ILOG Script operators, see Operators.

Syntax of different types of expression

Describes the syntax of several types of expression in OPL.

The following types of expression are described:

- A reference to a script variable
- Access to a property
- Assignment operators
- Function calls
- Special keywords
- Special operators
- Other operators

Note:

For C/C++ programmers: The syntax of Script expressions is very close to the C and C++ syntax. Expressions include assignments, function calls, property access, and so on.

Script variable reference

Provides a reference for the syntax of script variables in OPL.

Table 6. Reference syntax of script variables

Syntax	Effect
<i>variable</i>	Returns the value of <i>variable</i> . See “Identifiers” on page 88 for the syntax of script variables. If <i>variable</i> does not exist, an error is signalled. This is not the same as referencing an existing script variable whose value is the undefined value, which is legal and returns the undefined value. When used in the body of a <i>with</i> statement, a variable reference is first looked up as a property of the current default value.

Property access

Provides a reference for the syntax used to access properties in OPL.

There are two ways of accessing a property value.

Table 7. Property access syntax

Syntax	Effect
value.name	<p>Returns the value of the name property of value, or the undefined value if this property is not defined. See “Identifiers” on page 88 for the syntax of name.</p> <p>Examples: <code>str.length</code> <code>getCar().name</code></p> <p>Because name must be a valid identifier, this form cannot be used to access properties which do not have a valid identifier syntax. For example, the numeric properties of an array cannot be accessed this way: <code>myArray.10</code> // Illegal syntax</p> <p>For these properties, use the second syntax.</p>
value[name]	<p>Same as the previous syntax, except that this time name is an evaluated expression which gives the property name.</p> <p>Examples: <code>str["length"]</code> // Same as <code>str.length</code> <code>getCar()[getPropertyName()]</code> <code>myArray[10]</code> <code>myArray[i+1]</code></p>

Assignment operators

Provides a reference for the syntax of assignment operators in OPL.

The equals (=) operator can be used to assign a new value to a script variable or a property.

Table 8. Assignment operator syntax

Syntax	Effect
variable = expression	<p>In scripting, all objects are assigned by reference, except strings, numbers and Booleans, which are assigned by value. See the ECMA standard for details.</p> <p>Example: <code>x = y+1</code></p> <p>The whole expression returns the value of expression.</p>
value.name = expression value[name] = expression	<p>Assigns the value of expression to the given property.</p> <p>If value does not have such a property, then if it is either an array or an object, the property is created; otherwise, an error is signalled.</p> <p>Example: <code>car.name = "Ford"</code> <code>myArray[i] = myArray[i]+1</code></p> <p>The whole expression returns the value of expression.</p>

In addition, shorthand operators are also defined.

Function calls

Provides a reference for the shorthand syntax used with function calls in OPL.

Table 9. Syntactic shorthand

Syntax	Shorthand for
<code>++X</code>	<code>X = X+1</code>
<code>X++</code>	Same as <code>++X</code> , but returns the initial value of <code>X</code> instead of its new value.
<code>--X</code>	<code>X = X-1</code>
<code>X--</code>	Same as <code>--X</code> , but returns the initial value of <code>X</code> instead of its new value.
<code>X += Y</code>	<code>X = X + Y</code>
<code>X -= Y</code>	<code>X = X - Y</code>
<code>X *= Y</code>	<code>X = X * Y</code>
<code>X /= Y</code>	<code>X = X / Y</code>
<code>X %= Y</code>	<code>X = X % Y</code>
<code>X <<= Y</code>	<code>X = X << Y</code>
<code>X >>= Y</code>	<code>X = X >> Y</code>
<code>X >>>= Y</code>	<code>X = X >>> Y</code>
<code>X &= Y</code>	<code>X = X & Y</code>
<code>X ^= Y</code>	<code>X = X ^ Y</code>
<code>X = Y</code>	<code>X = X Y</code>

Table 10. Function call syntax

Syntax	Effect
<code>function (arg1, ..., argn)</code>	<p>Calls function with the given arguments, and returns the result of the call.</p> <p>Examples: <code>parseInt(field)</code> <code>writeln("Hello ", name)</code> <code>doAction()</code> <code>str.substring(start, start+length)</code></p> <p>The function is typically either a script variable reference or a property access, but it can be any expression; the expression must yield a function value, or an error is signalled.</p> <p>Examples: <code>callbacks[i](arg)</code> // Calls the function in <code>callbacks[i]</code> <code>"foo"()</code> // Error: a string is not a function</p>

Special keywords

Lists special reserved keywords in OPL.

Table 11. Special keyword syntax

Syntax	Effect
this	When referenced in a method, returns the current calling object; when referenced in a constructor, returns the object currently being initialized. Otherwise, returns the global object. See “this as a keyword” on page 120 for examples.
arguments	Returns an array containing the arguments of the current function. When used outside a function, an error is signalled. For example, the following function returns the sum of all its arguments: <pre>function sum() { var res = 0 for (var i=0; i<arguments.length; i++) res = res+arguments[i] return res }</pre> The call <code>sum(1, 3, 5)</code> returns 9.

Special operators

Lists the special operators in OPL and their syntax.

Table 12. Special operator syntax

Syntax	Effect
<code>new constructor(arg1, ..., argn)</code>	<p>Calls the constructor with the given arguments, and returns the created value.</p> <p>Examples:<code>new Array()</code> <code>new MyCar("Ford", 1975)</code></p> <p>The constructor is typically a script variable reference, but it can be any expression.</p> <p>Example:<code>new ctors[i](arg)</code> // Invokes constructor <code>ctors[i]</code></p>
<code>typeof value</code>	Returns a string representing the type of value, as follows: Array "object" Boolean "boolean" Date "date" Function "function" Null "object" Number "number" Object "object" String "string" Undefined "undefined"

Table 12. Special operator syntax (continued)

Syntax	Effect
delete variable	<p>Deletes the global script variable variable. This does not mean that the value in variable is deleted, but that it is removed from the global environment.</p> <p>Example:myVar = "Hello, world" // Create the global variable myVar delete myVar writeln(myVar) // Signals an error because myVar is undefined</p> <p>If variable is a local variable, an error is signalled; if variable is not a known variable, nothing happens.</p> <p>The whole expression returns true.</p> <p>For C/C++ programmers: The delete operator has a radically different meaning in IBM ILOG Script; in C++, it is used to delete objects, not script variables and properties.</p>
delete value.name delete value[name]	<p>Remove the property name from the object value.</p> <p>If value does not contain the name property, this expression does nothing. If the property does exist but cannot be deleted, an error is signalled. If value is not an object, an error is signalled.</p> <p>The whole expression returns the true value.</p>
expression1 , expression2	<p>Evaluates expression1 and expression2 sequentially, and returns the value of expression2. The value of expression1 is ignored.</p> <p>The most common use for this operator is inside for loops, where it can be used to evaluate several expressions where a single expression is expected:</p> <pre>for (var i=0, j=0; i<10; i++, j+=2) { writeln(j, " is twice as big as ", i); }</pre>

Other operators

Provides a reference of other operators used in OPL.

Other operators are described in detail in the section dedicated to the data type they operate on.

Table 13. Other operator syntax

Syntax	Effect
<code>- X</code> <code>X + Y</code> <code>X - Y</code> <code>X * Y</code> <code>X / Y</code> <code>X % Y</code>	<p>Arithmetic operators.</p> <p>These operators perform the usual arithmetic operations. In addition, the <code>+</code> operator can be used to concatenate strings. See “Numeric operators” on page 104 and “String operators” on page 112.</p>
<code>X == Y</code> <code>X != Y</code>	<p>Equality operators.</p> <p>These operators can be used to compare numbers and strings; see “Numeric operators” on page 104 and “String operators” on page 112.</p> <p>For other types of values, such as dates, arrays, and objects, the <code>==</code> operator is true if, and only if, X and Y are the exact same value. For example:</p> <pre>new Array(10) == new Array(10) false var a = new Array(10); a == a true</pre>
<code>X > Y</code> <code>X >= Y</code> <code>X < Y</code> <code>X <= Y</code>	<p>Relational operators.</p> <p>These operators can be used to compare numbers and strings. See “Numeric operators” on page 104 and “String operators” on page 112.</p>
<code>~ X</code> <code>X & Y</code> <code>X Y</code> <code>X ^ Y</code> <code>X << Y</code> <code>X >> Y</code> <code>X >>> Y</code>	<p>Bitwise operators.</p> <p>See “Numeric operators” on page 104.</p>
<code>! X</code> <code>X Y</code> <code>X && Y</code> <code>condition ? X : Y</code>	<p>Logical operators.</p> <p>See “Logical operators” on page 114.</p>

Statements

A statement can be a conditional statement, a loop statement, a local script variable declaration, a function definition, or a default value.

Conditional statement

Provides a reference for the syntax of conditional statements in OPL.

Loops

Table 14. Conditional statement syntax

Syntax	Effect
<code>if (expression) statement1</code> <code>[else statement2]</code>	<p>Evaluate <code>expression</code> ; if it is true, execute <code>statement1</code> ; otherwise, if <code>statement2</code> is provided, execute <code>statement2</code>.</p> <p>If <code>expression</code> gives a non-Boolean value, this value is converted to a Boolean value.</p> <p>Examples:</p> <pre>if (a == b) writeln("They are equal") else writeln("They are not equal") if (s.indexOf("a") < 0) { write("The string ", s) writeln(" doesn't contains the letter a") }</pre>

Table 15. Loop syntax

Syntax	Effect
<code>while (expression)</code> <code>statement</code>	<p>Execute <code>statement</code> repeatedly as long as <code>expression</code> is true. The test takes place before each execution of <code>statement</code>.</p> <p>If <code>expression</code> gives a non-Boolean value, this value is converted to a Boolean value.</p> <p>Examples:</p> <pre>while (a*a < b) a = a+1 while (s.length) { r = s.charAt(0)+r s = s.substring(1) }</pre>
<code>for ([initialize] ;</code> <code>[condition] ;</code> <code>[update])</code> <code>statement</code> <i>where <code>condition</code> and <code>update</code> are expressions, and <code>initialize</code> is either an expression or has the form:</i> <code>var variable = expression</code>	<p>Evaluate <code>initialize</code> once, if present. Its value is ignored. If it has the form:<code>var variable = expression</code></p> <p>then <code>variable</code> is declared as a local script variable and initialized as in the <code>var</code> statement.</p> <p>Then, execute <code>statement</code> repeatedly as long as <code>condition</code> is true. If <code>condition</code> is omitted, it is taken to be true, which results in an infinite loop. If <code>condition</code> gives a non-Boolean value, this value is converted to a Boolean value.</p> <p>If present, <code>update</code> is evaluated at each pass through the loop, after <code>statement</code> and before <code>condition</code>. Its value is ignored.</p> <p>Example:<pre>for (var i=0; i < a.length; i++) { sum = sum+a[i] prod = prod*a[i] }</pre></p>

Table 15. Loop syntax (continued)

Syntax	Effect
for ([var] variable in expression) statement	<p>Iterate over the properties of the value of expression : for each property, variable is set to a string representing this property, and statement is executed once.</p> <p>If the var keyword is present, variable is declared as a local script variable, as with the var statement.</p> <p>For example, the following function takes an arbitrary value and displays all its properties and their values: <code>function printProperties(v) { for (var p in v) writeln(p, " -> ", v[p]) }</code></p> <p>Properties listed by the <code>for..in</code> statement include method properties, which are merely regular properties whose value is a function value. For example, the call <code>printProperties("foo")</code> would display: <code>length -> 3 toString -> [primitive method toString] substring -> [primitive method substring] charAt -> [primitive method charAt]</code> etc</p> <p>The only properties which are not listed by <code>for..in</code> loops are the numeric properties of arrays.</p>
break	<p>Exit the current <code>while</code>, <code>for</code> or <code>for..in</code> loop, and continue the execution at the statement immediately following the loop. This statement cannot be used outside a loop.</p> <p>Example: <code>while (i < a.length) { if (a[i] == "foo") { foundFoo = true break } i = i+1 } //</code> Execution continues here</p>
continue	<p>Stop the current iteration of the current <code>while</code>, <code>for</code> or <code>for..in</code> loop, and continue the execution of the loop with the next iteration. This statement cannot be used outside a loop.</p> <p>Example: <code>for (var i=0; i < a.length; i++) { if (a[i] < 0) continue writeln("A positive number: ", a[i]) }</code></p>

Declaration of script variables

Provides a reference for the syntax of script variable declarations in OPL.

Table 16. Declaration syntax for script variables

Syntax	Effect
<code>var decl1, ..., decln</code> where each <code>decli</code> has the form <code>variable [= expression]</code>	Declares each script variable as a local variable. If an expression is provided, it is evaluated and its value is assigned to the variable as its initial value. Otherwise, the variable is set to the undefined value. Examples: <code>var x</code> <code>var name = "Joe"</code> <code>var average = (a+b)/2</code> , <code>sum</code> , <code>message="Hello"</code>

Inside a function definition

Script variables declared with `var` are local to the function, and they hide any global variables with the same names; they have the same status as function arguments.

For example, in the following program, the script variables `sum` and `res` are local to the `average` function, as well as the arguments `a` and `b` ; when `average` is called, the global variables with the same names, if any, are temporarily hidden until exit from the function:

```
function average(a, b) {  
    var sum = a+b  
    var res = sum/2  
    return res  
}
```

Script variables declared with `var` at any place in a function body have a scope which is the entire function body. This is different from local variable scope in C or C++. For example, in the following function, the variable `res` declared in the first branch of the `if` statement is used in the other branch and in the `return` statement:

```
function max(x, y) {  
    if (x > y) {  
        var res = x  
    } else {  
        res = y  
    }  
    return res  
}
```

Outside a function definition

At the same level as function definitions, script variables declared with `var` are local to the current program unit. A program unit is a group of statements which is considered a whole; the exact definition of a program unit depends on the application in which the script is embedded. Typically, a script file loaded by the application is treated as a program unit. In this case, variables declared with `var` at the file top level are local to this file, and they hide any global variables with the same names.

For example, suppose that a file contains the following program:

```

var count = 0

function NextNumber() {
    count = count+1
    return count
}

```

When this file is loaded, the function `NextNumber` becomes visible to the whole application, while `count` remains local to the loaded program unit and is visible only inside it.

It is an error to declare the same local variable twice in the same scope. For example, the following program is incorrect because `res` is declared twice:

```

function max(x, y) {
    if (x > y) {
        var res = x
    } else {
        var res = y // Error
    }
    return res
}

```

Function definitions

Provides a reference for the syntax of function definitions in OPL.

Table 17. Function definition syntax

Syntax	Effect
<pre>[static] function name(v1, ..., vn) {statements}</pre>	<p>Defines a function name with the given parameters and body. A function definition can only take place at the top level; function definitions cannot be nested.</p> <p>When the function is called, the script variables v1, ..., vn are set to the corresponding argument values, then the statements are executed. If a return statement is reached, the function returns the specified value; otherwise, after the statements are executed, the function returns the undefined value.</p> <p>The number of actual arguments does not need to match the number of parameters: if there are fewer arguments than parameters, the remaining parameters are set to the undefined value; if there are more arguments than parameters, the excess arguments are ignored.</p> <p>Independently of the parameter mechanism, the function arguments can be retrieved using the arguments keyword described in Table 11 on page 92.</p>
<pre>return [expression]</pre>	<p>Returns the value of expression from the current function. If expression is omitted, returns the undefined value. The return statement can only be used in the body of a function.</p>

Defining a function name is operationally the same as assigning a specific function value to the variable name; thus a function definition is equivalent to:

```
var name = some function value
```

The function value can be retrieved from the script variable and manipulated like any other type of value. For example, the following program defines a function `add` and assigns its value to the variable `sum`, which makes `add` and `sum` synonyms for the same function:

```
function add(a, b) {
    return a+b
}
```

```
sum = add
```

Without the keyword `static`, the defined function is global and can be accessed from the whole application. With the keyword `static`, the function is local to the current program unit, exactly as if `name` was declared with the keyword `var` :

```
var name = some function value
```

Default values

Lists the default values used in OPL.

Table 18. Default value syntax

Syntax	Effect
with (expression) statement	<p>Evaluate expression, then execute statement with the value of expression temporarily installed as the default value.</p> <p>When a reference to an identifier name in statement is evaluated, this identifier is first looked up as a property of the default value; if the default value does not have such a property, name is treated as a regular variable.</p> <p>For example, the following program displays "The length is 3", because the identifier <code>length</code> is taken as the <code>length</code> property of the string <code>"abc"</code>.</p> <pre>with ("abc") { writeln("The length is ", length) }</pre> <p>You can nest <code>with</code> statements; in this case, references to identifiers are looked up in the successive default values, from the innermost to the outermost <code>with</code> statement.</p>

Built-in values and functions

Presents the built-in values and functions of the IBM ILOG Script language for OPL: numbers, strings, Booleans, arrays, objects, dates, the `null` and undefined values, functions.

Numbers

Number representations and functions.

Introduction

Provides an overview of how numbers are expressed in OPL.

Numbers can be expressed in decimal (base 10), hexadecimal (base 16), or octal (base 8.) There are also special numbers.

Note:

For C/C++ programmers: Numbers have the same syntax as C and C++ integers and doubles. They are internally represented as 64-bit double-precision floating-point numbers.

Decimal numbers

Explains the use of decimal numbers in OPL.

A decimal number consists of a sequence of digits, followed by an optional fraction, followed by an optional exponent. The fraction consists of a decimal point (.) followed by a sequence of digits; the exponent consists of an e or E followed by an optional + or - sign and a sequence of digits. A decimal number must have at least one digit.

Here are some examples of decimal number literals:

```
15
3.14
4e100
.25
5.25e-10
```

Hexadecimal numbers

Shows how hexadecimal numbers are used in OPL.

A hexadecimal number consists of a 0x or 0X prefix, followed by a sequence of hexadecimal digits, which include digits 0-9 and the letters a-f or A-F. For example:

```
0x3ff
0x0
```

Octal numbers

Explains the use of octal numbers in OPL.

An octal number consists of a 0 followed by a sequence of octal digits, which include the digits 0-7. For example:

```
0123
0777
```

Special numbers

Explains NaN, Infinity, -Infinity, used in OPL.

There are three special numbers: NaN (Not-A-Number), Infinity (positive infinity), and -Infinity (negative infinity).

NaN

The special number NaN is used to indicate errors in number manipulations. For example, the square root function `Math.sqrt` applied to a negative number returns NaN. There is no representation of NaN as a number literal, but the global script variable NaN contains its value.

The NaN value is contagious, and a numeric operation involving NaN always returns NaN. A comparison operation involving NaN always returns false — even the NaN == NaN comparison.

Examples of NaN

```
Math.sqrt(-1) NaN
Math.sqrt(NaN) NaN
NaN + 3 NaN
NaN == NaN false
NaN <= 3 false
NaN >= 3 false
```

Note that, according to the ECMA specification <http://www.ecma-international.org/publications/standards/Ecma-262.htm>, the operation ToInt32 (NaN) returns the value 0. For example:

```
int i;
float f;

execute {
  i = NaN;
  f = NaN;

  writeln(i);
  writeln(f);
}
```

returns:

```
0
NaN
```

Infinity, -Infinity

The special numbers Infinity and -Infinity are used to indicate infinite values and overflows in arithmetic operations. The global script variable Infinity contains the positive infinity. The negative infinity can be computed using the negation operator (-Infinity).

Examples of Infinity

```
1/0 Infinity
-1/0 -Infinity
1/Infinity 0
Infinity == Infinity true
```

Automatic conversion to a number

Describes the automatic conversion of numbers in OPL functions.

When a function or a method which expects a number as one of its arguments is passed a nonnumeric value, it tries to convert this value to a number using the following rules:

- A string is parsed as a number literal. If the string does not represent a valid number literal, the conversion yields NaN.
- The Boolean true yields the number 1.
- The Boolean false yields the number 0.
- The null value yields the number 0.
- A date yields the corresponding number of milliseconds since 00:00:00 UTC, January 1, 1970.

For example, if the `Math.sqrt` function is passed a string, this string is converted to the number it represents:

```
Math.sqrt("25") 5
```

Similarly, operators which take numeric operands attempt to convert any nonnumeric operands to a number:

```
"3" * "4" 12
```

For operators that can take both strings (concatenation) and numbers (addition), such as `+`, the conversion to a string takes precedence over the conversion to a number (See “Automatic conversion to a string” on page 107.). In other words, if at least one of the operands is a string, the other operand is converted to a string; if none of the operands is a string, the operands are both converted to numbers. For example:

```
"3" + true "3true"
3 + true 4
```

For comparison operators, such as `==` and `>=`, the conversion to a number takes precedence over the conversion to a string. In other words, if at least one of the operands is a number, the other operand is converted to a number. If both operands are strings, the comparison is made on strings. For example:

```
"10" > "2" false
"10" > 2 true
```

Number methods

Provides a reference for the number method in OPL.

There is only one number method.

Table 19. Number method

Syntax	Effect
<code>number.toString()</code>	Returns a string representing the number as a literal. For example: <code>(14.3e2).toString()</code> "1430"

Numeric functions

Provides a reference for numeric functions in the OPL language.

Note:

For C/C++ programmers: Most of the numeric functions are wrap-ups for standard math library functions.

Table 20. Numeric functions

Syntax	Effect
<code>Math.abs(x)</code>	Returns the absolute value of <code>x</code> .
<code>Math.max(x,y)</code>	<code>Math.max(x, y)</code> returns the larger of <code>x</code> and <code>y</code> .
<code>Math.min(x,y)</code>	<code>Math.min(x, y)</code> returns the smaller of <code>x</code> and <code>y</code> .
<code>Math.random()</code>	Returns a pseudo-random number between 0, inclusive, and 1, exclusive.

Table 20. Numeric functions (continued)

Syntax	Effect
Math.ceil(x)	Math.ceil(x) returns the smallest integer value greater than or equal to x.
Math.floor(x)	Math.floor(x) returns the greatest integer value less than or equal to x.
Math.round(x)	Math.round(x) returns the nearest integer value to x.
Math.sqrt(x)	Returns the square root of x.
Math.sin(x)	Math.sin(x) returns the trigonometric function sine of a radian argument.
Math.cos(x)	Math.cos(x) returns the trigonometric function cosine of a radian argument.
Math.tan(x)	Math.tan(x) returns the trigonometric function tangent of a radian argument.
Math.asin(x)	Math.asin(x) returns the arcsine of x in the range $-\pi/2$ to $\pi/2$.
Math.acos(x)	Math.acos(x) returns the arc cosine of x in the range 0 to π .
Math.atan(x)	Math.atan(x) returns the arc tangent of x in the range $-\pi/2$ to $\pi/2$.
Math.atan2(y,x)	Math.atan2(y, x) converts rectangular coordinates (x, y) to polar coordinates (r, a) by computing a as an arc tangent of y/x in the range $-\pi$ to π .
Math.exp(x)	Math.exp(x) computes the exponential function.
Math.log(x)	Math.log(x) computes the natural logarithm of x.
Math.pow(x,y)	Math.pow(x, y) computes x raised to the power y.

Numeric constants

Provides a reference for numeric constants in OPL.

The following numeric constants are defined.

Table 21. Numeric constants

Syntax	Value
NaN	Contains the NaN value.
Infinity	Contains the Infinity value.
Number.NaN	Same as NaN.
Number.MAX_VALUE	The maximum representable number, approximately 1.79E+308.
Number.MIN_VALUE	The smallest representable positive number, approximately 2.22E-308.
Math.E	Napier's constant, e, and the base of natural logarithms, approximately 2.718.

Table 21. Numeric constants (continued)

Syntax	Value
Math.LN10	The natural logarithm of 10, approximately 2.302.
Math.LN2	The natural logarithm of 2, approximately 0.693.
Math.LOG2E	The base 2 logarithm of e, approximately 1.442.
Math.LOG10E	The base 10 logarithm of e, approximately 0.434.
Math.PI	The ratio of the circumference of a circle to its diameter, approximately 3.142.
Math.SQRT1_2	The square root of one-half, approximately 0.707.
Math.SQRT2	The square root of 2, approximately 1.414.

Numeric operators

Provides a reference for numeric operators in OPL.

Note:

For C/C++ programmers: The numeric operators are the same as in C and C++.

Table 22. Numeric operators

Syntax	Effect
$x+y$ $x-y$ $x*y$ x/y	<p>The usual arithmetic operations.</p> <p>Examples:</p> $3 + 4.2 \rightarrow 7.2$ $100 - 120 \rightarrow -20$ $4 * 7.1 \rightarrow 28.4$ $6 / 5 \rightarrow 1.2$
$-x$	<p>Negation.</p> <p>Examples:</p> -142 -142
$x\%y$	<p>Returns the floating-point remainder of dividing x by y.</p> <p>Examples:</p> $12 \% 5 \rightarrow 2$ $12.5 \% 5 \rightarrow 2.5$

Table 22. Numeric operators (continued)

Syntax	Effect
$x == y$ $x != y$	<p>The operator <code>==</code> returns true if x and y are equal, and false otherwise. The operator <code>!=</code> is the converse of <code>==</code>.</p> <p>Examples:</p> <p><code>12 == 12</code> → true</p> <p><code>12 == 12.1</code> → false</p> <p><code>12 != 12.1</code> → true</p>
$x < y$ $x <= y$ $x > y$ $x >= y$	<p>The operator <code><</code> returns true if x is less than y, and false otherwise. The operator <code><=</code> returns true if x is less than or equal to y, and false otherwise; and so on.</p> <p>Examples:</p> <p><code>-1 < 0</code> → true</p> <p><code>1 < 1</code> → false</p> <p><code>1 <= 1</code> → true</p>
$x \& y$ $x y$ $x \wedge y$	<p>The bitwise operations <code>&</code> (AND), <code> </code> (inclusive OR), and <code>^</code> (exclusive OR), where x and y must be integers in the range $-2^{31}+1$ to $2^{31}-1$ (-2147483647 to 2147483647.)</p> <p>Examples:</p> <p><code>14 & 9</code> → 8 (because 1110 & 1001 → 1000)</p> <p><code>14 9</code> → 15 (because 1110 1001 → 1111)</p> <p><code>14 ^ 9</code> → 7 (because 1110 ^ 1001 → 111)</p>
$\sim x$	<p>The bitwise NOT operation, where x must be an integer in the range $-2^{31}+1$ to $2^{31}-1$ (-2147483647 to 2147483647.)</p> <p>Examples:</p> <p><code>~ 14</code> → 1 (because ~ 1110 → 0001)</p>

Table 22. Numeric operators (continued)

Syntax	Effect
<code>x<<y</code> <code>x>>y</code> <code>x>>>y</code>	<p>Binary shift operations, where x and y must be integers in the range $-2^{31}+1$ to $2^{31}-1$ (-2147483647 to 2147483647.) The operator << shifts to the left, >> shifts to the right (maintaining the sign bit). The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted.</p> <p>>>> shifts to the right, shifting in zeros from the left.</p> <p>Examples:</p> <p><code>9 << 2 -> 36</code> (because <code>1001 << 2 -> 100100</code>)</p> <p><code>9 >> 2 -> 2</code> (because <code>1001 >> 2 -> 10</code>)</p> <p><code>-9 >> 2 -> -2</code> (because <code>1..11001 >> 2 -> 1..11110</code>)</p> <p><code>-9 >>> 2 -> 1073741821</code> (because <code>1..11001 >>> 2 -> 01..11110</code>)</p>

IBM ILOG Script strings

String representation and functions.

Introduction

Provides an overview of the use of strings in IBM ILOG Script.

A string literal is zero or more characters enclosed in double (") or single (') quotes.

Note:

For C/C++ programmers: Except for the use of single quotes, string literals have the same syntax as in C and C++.

Here are examples of string literals:

```
"My name is Hal"
'My name is Hal'
"Hi there", he said'
"3.14"
"Hello, world\n"
```

In these examples, the first and second strings are identical.

The backslash character \ can be used to introduce an escape sequence, which stands for a character which cannot be directly expressed in a string literal.

Table 23. Escape sequences in strings

Escape sequence	Stands for
<code>\n</code>	Newline
<code>\t</code>	Tab

Table 23. Escape sequences in strings (continued)

Escape sequence	Stands for
\\	Backslash character (\)
\"	Double quote (")
\'	Single quote (')
\b	Backspace
\f	Form feed
\r	Carriage return
\xhh	The character whose ASCII code is hh, where hh is a sequence of two hexadecimal digits.
\ooo	The character whose ASCII code is ooo, where ooo is a sequence of one, two, or three octal digits.

Here are examples of string literals using escape sequences:

Table 24. Examples of string literals using escape sequences

String literal	Stands for
"Read 'The Black Bean'"	Read 'The Black Bean'
"'Hello", he said"	"Hello", he said
"c:\\temp"	c:\temp
"First line\nSecond line\nThird line"	First line Second line Third line
"\xA9 1995-1997"	© 1995-1997

When a string is converted to a number, an attempt is made to parse it as a number literal. If the string does not represent a valid number literal, the conversion yields NaN.

Automatic conversion to a string

Explains how nonstrings are automatically converted to strings.

When a function or a method that expects a string as one of its arguments is passed a nonstring value, this value is automatically converted to a string. For example, if the string method `indexOf` is passed a number as its first argument, this number is treated like its string representation:

```
"The 10 commandments".indexOf(10) -> 4
```

Similarly, operators that take string operands automatically convert nonstring operands to strings:

```
"The " + 10 + " commandments" -> "The 10 commandments"
```

The conversion to a string uses the `toString` method of the given value. All built-in values have a `toString` method.

String properties

Provides a reference for the properties of strings.

There is a single, read-only string property.

Table 25. String property

Syntax	Value
<code>string.length</code>	Number of characters in string. This is a read-only property. Examples: <code>"abc".length -> 3</code> <code>"".length -> 0</code>

String methods

Provides a reference for string methods.

Characters in a string are indexed from left to right. The index of the first character in a string is 0, and the index of the last character is `string.length-1`.

Table 26. String methods

Syntax	Effect
<code>string.substring(start [, end])</code>	Returns the substring of string starting at the index start and ending at the index end-1. If end is omitted, the tail of string is returned. Examples: <code>"0123456".substring(0, 3) -> "012"</code> <code>"0123456".substring(2, 4) -> "23"</code> <code>"0123456".substring(2) -> "23456"</code>
<code>string.charAt(index)</code>	Returns a one-character string containing the character at the specified index of string. If index is out of range, an empty string is returned. Examples: <code>"abcdef".charAt(0) -> "a"</code> <code>"abcdef".charAt(3) -> "d"</code> <code>"abcdef".charAt(100) -> ""</code>
<code>string.charCodeAt(index)</code>	Returns the ASCII code of the character at the specified index of string. If index is out of range, returns NaN. Examples: <code>"abcdef".charCodeAt(0) -> 97</code> <code>"abcdef".charCodeAt(3) -> 100</code> <code>"abcdef".charCodeAt(100) -> NaN</code>

Table 26. String methods (continued)

Syntax	Effect
<code>string.indexOf(substring [, index])</code>	<p>Returns the index in string of the first occurrence of substring. The string is searched starting at index. If index is omitted, string is searched from the beginning. This method returns -1 if substring is not found.</p> <p>Examples:</p> <pre>"abcdabcd".indexOf("bc") -> 1 "abcdabcd".indexOf("bc", 1) -> 1 "abcdabcd".indexOf("bc", 2) -> 5 "abcdabcd".indexOf("bc", 10) -> -1 "abcdabcd".indexOf("foo") -> -1 "abcdabcd".indexOf("BC") -> -1</pre>
<code>string.lastIndexOf(substring [, index])</code>	<p>Returns the index in string of the last occurrence of substring, when string is searched backwards, starting at index. If index is omitted, string is searched from the end. This method returns -1 if substring is not found.</p> <p>Examples:</p> <pre>"abcdabcd".lastIndexOf("bc") -> 5 "abcdabcd".lastIndexOf("bc", 5) -> 5 "abcdabcd".lastIndexOf("bc", 4) -> 1 "abcdabcd".lastIndexOf("bc", 0) -> -1 "abcdabcd".lastIndexOf("foo") -> -1 "abcdabcd".lastIndexOf("BC") -> -1</pre>
<code>string.toLowerCase()</code>	<p>Returns string converted to lowercase.</p> <p>Example: "Hello, World".toLowerCase() "hello, world"</p>
<code>string.toUpperCase()</code>	<p>string.toUpperCase() Returns string converted to uppercase.</p> <p>Example:"Hello, World".toUpperCase() "HELLO, WORLD"</p>

Table 26. String methods (continued)

Syntax	Effect
<code>string.split(separator)</code>	<p>Returns an array of strings containing the substrings of <code>string</code> that are separated by <code>separator</code>. See also the array method <code>join</code>.</p> <p>Examples:</p> <p><code>"first name,last name,age".split(",")</code> -> an array <code>a</code> such that <code>.length</code> is 3, <code>a[0]</code> is "first name", <code>a[1]</code> is "last name", and <code>a[2]</code> is "age".</p> <p>If <code>string</code> does not contain <code>separator</code>, an array with one element containing the whole string is returned.</p> <p>Example:</p> <p><code>"hello".split(",")</code> -> an array <code>a</code> such that <code>a.length</code> is 1 and <code>a[0]</code> is "hello",</p>
<code>string.toString()</code>	Returns the string itself.

String functions

Provides a reference of string functions.

Table 27. String functions

Syntax	Effect
<code>String.fromCharCode(code)</code>	<p>Returns a single-character string containing the character with the given ASCII code.</p> <p>Examples:</p> <p><code>String.fromCharCode(65)</code> -> "A"</p> <p><code>writeln(String.fromCharCode(0x30));</code> -> "0"</p>

Table 27. String functions (continued)

Syntax	Effect
<code>parseInt(string [, base])</code>	<p>Parses string as an integer written in the given base, and returns its value. If the string does not represent a valid integer, NaN is returned.</p> <p>Leading white space characters are ignored. If parseInt encounters a character that is not a digit in the specified base, it ignores it and all succeeding characters and returns the integer value parsed up to that point.</p> <p>If base is omitted, it is taken to be 10, unless string starts with 0x or 0X, in which case it is parsed in base 16, or with 0, in which case it is parsed in base 8.</p> <p>Examples:</p> <pre>parseInt("123") -> 123 parseInt("-123") -> -123 parseInt("123.45") -> 123 parseInt("1001010010110", 2) -> 4758 parseInt("a9", 16) -> 169 parseInt("0xa9") -> 169 parseInt("010") -> 8 parseInt("123 poodles") -> 123 parseInt("a lot of poodles") -> NaN</pre>
<code>parseFloat(string)</code>	<p>Parses string as a floating-point number and return its value. If the string does not represent a valid number, NaN is returned.</p> <p>Leading white space characters are ignored. The string is parsed up to the first unrecognized character. If no number is recognized, the function returns NaN.</p> <p>Examples:</p> <pre>parseFloat("-3.14e-15") -> -3.14e-15 parseFloat("-3.14e-15 poodles") -> -3.14e-15 parseFloat("a fraction of a poodle") -> NaN</pre>

String operators

Provides a reference of string operators.

Table 28. String operators

Syntax	Effect
<code>string1 + string2</code>	<p>Returns a string containing the concatenation of <code>string1</code> and <code>string2</code>.</p> <p>Examples: "Hello," + " world" -> "Hello, world"</p> <p>When the operator + is used to add a string to a nonstring value, the nonstring value is first converted to a string.</p> <p>Examples:</p> <p>"Your age is " + 23 -> "Your age is 23"</p> <p>23 + " is your age" -> "23 is your age"</p>
<code>string1 == string2</code> <code>string1 != string2</code>	<p>The operator <code>==</code> returns the Boolean true if <code>string1</code> and <code>string2</code> are identical, and false otherwise. Two strings are identical if they have the same length and contain the same sequence of characters. The operator <code>!=</code> is the converse of <code>==</code>.</p> <p>Examples:</p> <p>"a string" == "a string" -> true</p> <p>"a string" == "another string" -> false</p> <p>"a string" == "A STRING" -> false</p> <p>"a string" != "a string" -> false</p> <p>"a string" != "another string" -> true</p> <p>When the operators <code>==</code> and <code>!=</code> are used to compare a string with a number, the string is first converted to a number and the two numbers are compared numerically.</p> <p>Examples:</p> <p>"12" == "+12" -> false</p> <p>12 == "+12" -> true</p>

Table 28. String operators (continued)

Syntax	Effect
<code>string1 < string2</code> <code>string1 <= string2</code> <code>string1 > string2</code> <code>string1 >= string2</code>	<p>The operator <code><</code> returns true if <code>string1</code> strictly precedes <code>string2</code> lexicographically, and false otherwise. The operator <code><=</code> returns true if <code>string1</code> strictly precedes <code>string2</code> lexicographically or is equal to it, and false otherwise; and so on.</p> <p>Examples:</p> <pre>"abc" < "xyz" -> true "a" < "abc" -> true "xyz" < "abc" -> false "abc" < "abc" -> false "abc" > "xyz" -> false "a" > "abc" -> false "xyz" > "abc" -> true</pre> <p>Etc.</p> <p>When one of these operators is used to compare a string with a number, the string is first converted to a number and the two numbers are compared numerically. In all other cases, the other argument is first converted to a string.</p> <p>Examples:</p> <pre>"10" > "2" -> false 10 > "2" -> true 123 < "2" -> false</pre> <p>Hint: Autocasting may cause unexpected behavior.</p>

IBM ILOG Script Booleans

Boolean representation and functions.

Introduction

Provides an overview of the use of Boolean literals in IBM ILOG Script.

There are two Boolean literals: `true`, which represents the Boolean value true, and `false`, which represents the Boolean value false.

When converted to a number, `true` yields 1 and `false` yields 0.

Automatic conversion to Boolean

Describes the automatic conversion of non-Booleans to Booleans.

When a function, method or statement which expects a Boolean value as one of its arguments is passed a non-Boolean value, this value is automatically converted to a Boolean value as follows:

- The number 0 yields false.
- The empty string "" yields false.
- The null value yields false.
- The undefined value yields false.
- Any other non-Boolean values yield true.

For example:

```
if ("" ) writeln("True"); else writeln("False");  
if (123) writeln("True"); else writeln("False");
```

This displays “False”, then “True”.

Boolean methods

Provides a reference of Boolean methods.

There is only one Boolean method.

Table 29. Boolean method

Syntax	Effect
<i>boolean.toString()</i>	Returns a string representing the Boolean value, either "true" or "false". Example: true.toString -> "true" false.toString -> "false"

Logical operators

Provides a reference of logical operators.

The following Boolean operators are available.

Note:

For C/C++ programmers: These operators are the same as in C and C++.

Table 30. Logical operators

Syntax	Effect
<i>! boolean</i>	Logical negation. Examples: ! true -> false ! false -> true

Table 30. Logical operators (continued)

Syntax	Effect
<i>exp1</i> && <i>exp2</i>	<p>Logical AND. Returns true if both Boolean expressions <i>exp1</i> and <i>exp2</i> are true. Otherwise, returns false.</p> <p>If <i>exp1</i> is false, this expression immediately returns false without evaluating <i>exp2</i>, so any side effects of <i>exp2</i> are not taken into account.</p> <p>Examples:</p> <p>true && true -> true</p> <p>true && false -> false</p> <p>false && whatever -> false (whatever is not evaluated)</p>
<i>exp1</i> <i>exp2</i>	<p>Logical OR. Returns true if either Boolean expression <i>exp1</i> or <i>exp2</i> (or both) is true. Otherwise, returns false.</p> <p>If <i>exp1</i> is true, this expression immediately returns true without evaluating <i>exp2</i>, so any side effects of <i>exp2</i> are not taken into account.</p> <p>Examples:</p> <p>false true -> true</p> <p>false false -> false</p> <p>true whatever -> true (whatever is not evaluated)</p>
<i>condition</i> ? <i>exp1</i> : <i>exp2</i>	<p>If <i>condition</i> is true, this expression returns <i>exp1</i> ; otherwise, it returns <i>exp2</i>.</p> <p>When <i>condition</i> is true, the expression <i>exp2</i> is not evaluated, so any side effects it may contain are not taken into account. Similarly, when <i>condition</i> is false, <i>exp1</i> is not evaluated.</p> <p>Examples:</p> <p>true ? 3.14 : whatever -> 3.14</p> <p>false ? whatever : "Hello" -> "Hello"</p>

IBM ILOG Script arrays

Array representation and functions.

Introduction

Provides an overview of the use of arrays in IBM ILOG Script.

Arrays provide a way of manipulating ordered sets of values referenced through an index starting from zero (0). Unlike arrays in other languages, IBM ILOG Script

arrays do not have a fixed size and are automatically expanded as new elements are added. For example, in the following program, an array is created empty, and new elements are then added.

```
a = new Array() // Create an empty array
a[0] = "first" // Set the element 0
a[1] = "second" // Set the element 1
a[2] = "third" // Set the element 2
```

Arrays are internally represented as sparse objects, which means that an array where only the element 0 and the element 10000 have been set occupies just enough memory to store these two elements, not the 9999 which are between 0 and 10000.

Array constructor

Provides a reference for array constructors in IBM ILOG Script.

The array constructor has two distinct forms.

Table 31. Array constructor

Syntax	Effect
<code>new Array(<i>length</i>)</code>	<p>Returns a new array <i>a</i> of length <i>length</i> with its elements from 0 to <i>length</i>-1 set to null.</p> <p>If <i>length</i> is not a number, and its conversion to a number yields NaN, the second syntax is used.</p> <p>Examples:</p> <p><code>new Array(12)</code> → an array <i>a</i> with length 12 and <i>a</i>[0] to <i>a</i>[11] containing null</p> <p><code>new Array("5")</code> → an array <i>a</i> with length 5 and <i>a</i>[0] to <i>a</i>[4] containing null</p> <p><code>new Array("foo")</code> see second syntax</p>
<code>new Array(<i>element1</i>, ..., <i>elementn</i>)</code>	<p>Returns a new array <i>a</i> of length <i>n</i> with <i>a</i>[0] containing <i>element1</i>, <i>a</i>[1] containing <i>element2</i>, and so on. If no argument is given, that is <i>n</i>=0, an empty array is created. If <i>n</i>=1 and <i>element1</i> is a number or can be converted to a number, the first syntax is used.</p> <p>Examples:</p> <p><code>new Array(327, "hello world")</code> → an array <i>a</i> of length 2 with <i>a</i>[0] == 327 and <i>a</i>[1] == "hello world"</p> <p><code>new Array()</code> → an array with length 0</p> <p><code>new Array("327")</code> see first syntax</p>

Array properties

Provides a reference of the properties of arrays.

Table 32. Array properties

Syntax	Effect
<code>array[index]</code>	<p>If <code>index</code> can be converted to a number between 0 and $2^{31}-1$ (see “Automatic conversion to a number” on page 101), <code>array[index]</code> is the value of the index-th element of the array.</p> <p>Otherwise, it is considered as a standard property access.</p> <p>If this element has never been set, <code>null</code> is returned.</p> <p>Example:</p> <p>Suppose that the array <code>a</code> has been created with <code>a = new Array("foo", 12, true)</code></p> <p>Then:</p> <p><code>a[0]</code> -> "foo" <code>a[1]</code> -> 12 <code>a[2]</code> -> true <code>a[3]</code> -> null <code>a[1000]</code> -> null</p> <p>When an element of an array is set beyond the current length of the array, the array is automatically expanded: <code>a[1000] = "bar"</code> // the array is automatically expanded.</p> <p>Unlike other properties, the numeric properties of an array are not listed by the <code>for...in</code> statement.</p>

Table 32. Array properties (continued)

Syntax	Effect
<code>array.length</code>	<p>The length of array, which is the highest index of an element set in array, plus one. It is always included in 0 to 2e31 -1.</p> <p>When a new element is set in the array, and its index is greater than or equal to the current array length, the length property is automatically increased.</p> <p>Example: Suppose that the array a has been created with <code>a = new Array("a", "b", "c")</code></p> <p>Then: <code>a.length -> 3</code> <code>a[100] = "bar"; a.length -> 101</code></p> <p>You can also change the length of an array by setting its length property.</p> <p><code>a = new Array(); a[4] = "foo"; a[9] = "bar"</code> <code>a.length -> 10</code></p> <p><code>a.length = 5</code> <code>a.length -> 5</code> <code>a[4] -> "foo"</code> <code>a[9] -> null</code></p>

Array methods

Provides a reference of array methods.

Table 33. Array methods

Syntax	Effect
<code>array.join([separator])</code>	<p>Returns a string that contains the elements of the array converted to strings, concatenated together and separated with <i>separator</i>. If <i>separator</i> is omitted, it is taken to be <code>" , "</code>. Elements that are not initialized are converted to the empty string. See also the string method <code>split</code>.</p> <p>Example: Suppose that the array a has been created with</p> <p><code>a = new Array("foo", 12, true)</code></p> <p>Then:</p> <p><code>a.join("/") -> "foo//12//true"</code></p> <p><code>a.join() -> "foo,12,true"</code></p>

Table 33. Array methods (continued)

Syntax	Effect
<code>array.sort([function])</code>	<p>Sorts the array. The elements are sorted in place; no new array is created.</p> <p>If <i>function</i> is not provided, <i>array</i> is sorted lexicographically: Elements are compared by converting them to strings and using the < operator. With this order, the number 20 would come before the number 5, since "20" < "5" is true.</p> <p>If <i>function</i> is supplied, the array is sorted according to the return value of this function. This function must take two arguments x and y and return:</p> <ul style="list-style-type: none"> -1 if x is smaller than y 0 if x is equal to y 1 if x is greater than y <p>Example: Suppose that the function <code>compareLength</code> is defined as</p> <pre>function compareLength(x, y) {if (x.length < y.length) return -1; else if (x.length == y.length) return 0; else return 1; }</pre> <p>and that the array <i>a</i> has been created with:</p> <pre>a = new Array("giraffe", "rat", "brontosaurus")</pre> <p>Then <code>a.sort()</code> will reorder its elements as follows:"brontosaurus" "rat" "giraffe"</p> <p>while <code>a.sort(compareLength)</code> will reorder them as follows:"rat" "giraffe" "brontosaurus"</p>
<code>array.reverse()</code>	<p>Transposes the elements of the array: the first element becomes the last, the second becomes the second to last, and so on. The elements are reversed in place; no new array is created.</p> <p>Example: Suppose that the array <i>a</i> has been created with</p> <pre>a = new Array("foo", 12, "hello", true, false)</pre> <p>Then <code>a.reverse()</code> changes <i>a</i> so that:</p> <pre>a[0] false a[1] true a[2] "hello" a[3] 12 a[4] "foo"</pre>

Table 33. Array methods (continued)

Syntax	Effect
<code>array.toString()</code>	Returns the string "[object Object]".

Objects

Object representation and functions.

Introduction

Provides an overview of the use of objects in IBM ILOG Script.

Objects are values which do not contain any predefined properties or methods (except the `toString` method), but where new ones can be added. A new, empty object can be created using the Object constructor. For example, the following program creates a new object, stores it in the variable `myCar`, and adds the properties "name" and "year" to it:

```
myCar = new Object() // o contains no properties
myCar.name = "Ford"
myCar.year = 1985
```

Now:

```
myCar.name -> "Ford"
myCar.year -> 1985
```

Defining methods

Explains how methods are defined in IBM ILOG Script.

Since a method is really a property which contains a function value, defining a method simply consists in defining a regular function, then assigning it to a property.

For example, the following program adds a method `start` to the `myCar` object defined in the previous section:

```
function start_engine() {
    writeln("vroom vroom")
}
myCar.start = start_engine
```

Now, the expression `myCar.start()` will call the function defined as `start_engine`. Note that the only reason for using a different name for the function and for the method is to avoid confusion; we could have written:

```
function start() {
    writeln("vroom vroom")
}

myCar.start = start
```

this as a keyword

Describes the use of the `this` keyword in IBM ILOG Script.

Inside methods, the `this` keyword can be used to reference the calling object. For example, the following program defines a method `getName`, which returns the value of the `name` property of the calling object, and adds this method to `myCar`:

```
function get_name() {
    return this.name
}
myCar.getName = get_name
```

Inside constructors, `this` references the object created by the constructor.

When used in a nonmethod context, `this` returns a reference to the global object. The global object contains script variables declared at top level, and built in functions and constructors.

Object constructor

Provides a reference of object constructors in IBM ILOG Script.

Table 34. Object constructor

Syntax	Effect
<code>new Object()</code>	Returns a new object with no properties.

User-defined constructors

Provides a reference of user-defined constructors.

In addition to the Object constructor, any user-defined function can be used as an object constructor.

Table 35. User-defined constructors

Syntax	Effect
<code>new function(arg1, ..., argn)</code>	Creates a new object, then calls <code>function(arg1, ..., argn)</code> to initialize it.

Inside the constructor, the keyword `this` can be used to make reference to the object being initialized.

For example, the following program defines a constructor for cars:

```
function Car(name, year) {
    this.name = name;
    this.year = year;
    this.start = start_engine;
}
```

Now, calling

```
new Car("Ford", "1985")
```

creates a new object with the properties `name` and `year`, and a `start` method.

Built-in methods

Provides a reference of the built-in methods of IBM ILOG Script.

There is only one object built-in method.

Table 36. Built-in method

Syntax	Effect
<code>object.toString()</code>	Returns the string "[object Object]". This method can be overridden by assigning the <code>toString</code> property of an object.

Dates

Date representation and functions

Introduction

Provides an overview of dates and date functions in IBM ILOG Script.

Date values provide a way of manipulating dates and times. Dates can be best understood as internally represented by a number of milliseconds since 00:00:00 UTC, January 1, 1970. This number can be negative, to express a date before 1970.

Note:

1. **For C/C++ programmers:** Unlike dates manipulated by the standard C library, date values are not limited to the range of 1970 to 2038, but span approximately 285,616 years before and after 1970.
2. IBM ILOG Script measures time in wall-clock time (and not system time).

When converted to a number, a date yields the number of milliseconds since 00:00:00 UTC, January 1, 1970.

Date constructor

Explains the different forms of the date constructor.

The date constructor has four distinct forms.

Table 37. Date constructor

Syntax	Effect
<code>new Date()</code>	Returns the date representing the current time.
<code>new Date(milliseconds)</code>	<p>Returns the date representing 00:00:00 UTC, January 1, 1970, plus milliseconds milliseconds. The argument can be negative, to express a date before 1970. If the argument cannot be converted to a number, the third constructor syntax is used.</p> <p>Examples:</p> <p><code>new Date(0)</code> -> a date representing 00:00:00 UTC, January 1, 1970.</p> <p><code>new Date(1000*60*60*24*20)</code> -> a date representing twenty days after 00:00:00 UTC, January 1, 1970.</p> <p><code>new Date(-1000*60*60*24*20)</code> -> a date representing twenty days before 00:00:00 UTC, January 1, 1970.</p>

Table 37. Date constructor (continued)

Syntax	Effect
<code>new Date(string)</code>	<p>Returns the date described by <i>string</i>, which must have the form:</p> <p>month/day/year hour:minute:second msecond</p> <p>The date expressed in <i>string</i> is taken in local time.</p> <p>Example:</p> <p><code>new Date("12/25/1932 14:35:12 820")</code></p> <p>A date representing December 25th, 1932, at 2:35 PM plus 12 seconds and 820 milliseconds, local time.</p>
<code>new Date(year,</code> <code>month,</code> <code>[, day</code> <code>[, hours</code> <code>[, minutes</code> <code>[, seconds</code> <code>[, mseconds]]]]])</code>	<p>Returns a new date representing the given year, month, day, and so on, taken in local time. The arguments are:</p> <p><i>year</i>: any integer.</p> <p><i>month</i>: range 0-11 (where 0=January, 1=February, and so on)</p> <p><i>day</i>: range 1-31, default 1</p> <p><i>hours</i>: range 0-23, default 0</p> <p><i>minutes</i>: range 0-59, default 0</p> <p><i>seconds</i>: range 0-59, default 0</p> <p><i>mseconds</i>: range 0-999, defaults to 0</p> <p>Examples:</p> <p><code>new Date(1932, 11, 25, 14, 35, 12, 820)</code></p> <p>A date representing December 25th, 1932, at 2:35 PM plus 12 seconds and 820 milliseconds, local time.</p> <p><code>new Date(1932, 11, 25)</code></p> <p>A date representing December 25th, 1932, at 00:00, local time.</p>

Date methods

Provides a reference of date methods in IBM ILOG Script.

Table 38. Date methods

Syntax	Effect
<code>date.getTime()</code> <code>date.setTime(milliseconds)</code>	Returns (or sets) the number of milliseconds since 00:00:00 UTC, January 1, 1970. Example: Suppose that the date <code>d</code> has been created with: <code>d = new Date(3427)</code> Then: <code>d.getTime()</code> -> 3427
<code>date.toLocaleString()</code> <code>date.toUTCString()</code>	Returns a string representing the date in local time or in UTC respectively. Example: Suppose that the date <code>d</code> has been created with: <code>d = new Date("3/12/1997 12:45:00 0")</code> Then: <code>d.toLocaleString()</code> -> "03/12/1997 12:45:00 000" <code>d.toUTCString()</code> -> "03/12/1997 10:45:00 000" assuming a local time zone offset of +2 hours with respect to Greenwich Mean Time.
<code>date.getYear()</code> <code>date.setYear(year)</code>	Returns (or sets) the year of date.
<code>date.getMonth()</code> <code>date.setMonth(month)</code>	Returns (or sets) the month of date.
<code>date.getDate()</code> <code>date.setDate(day)</code>	Returns (or sets) the day of date.
<code>date.getHours()</code> <code>date.setHours(day)</code>	Returns (or sets) the hours of date.
<code>date.getMinutes()</code> <code>date.setMinutes(minutes)</code>	Returns (or sets) the minutes of date.
<code>date.getSeconds()</code> <code>date.setSeconds(seconds)</code>	Returns (or sets) the seconds of date.
<code>date.getMilliseconds()</code> <code>date.setMilliseconds(milliseconds)</code>	Returns (or sets) the milliseconds of date.
<code>date.toString()</code>	Returns the same value as <code>date.toLocaleString()</code>

Related reference:

“Tips on date formats”

Provides some tips on using ILOG Script to improve the date format.

Date functions

Provides a reference of date functions.

Table 39. Date functions

Syntax	Effect
<code>Date.UTC(year, month, [, day [, hours [, minutes [, seconds [, mseconds]]])</code>	Returns a number representing the given date taken in UTC. The arguments are: <i>year</i> : any integer <i>month</i> : range 0-11, where 0 = January, 1 = February, and so on <i>day</i> : range 1-31, default 1 <i>hours</i> : range 0-59, default 0 <i>minutes</i> : range 0-59, default 0 <i>seconds</i> : range 0-59, default 0 <i>mseconds</i> : range 0-999, default 0
<code>Date.parse(string)</code>	Same as <code>new Date(string)</code> , but the result is returned as a number rather than as a date object.

Date operators

Explains the use of date operators in IBM ILOG Script.

There are no specific operators for dealing with dates, but, since numeric operators automatically convert their arguments to numbers, these operators can be used to compute the time elapsed between two dates, to compare dates, or to add a given amount of time to a date. For example:

```
date1 - date2 -> the number of milliseconds elapsed between date1 and date2.
date1 < date2 -> true if date1 is before date2, false otherwise.
new Date(date+10000) ->
  a date representing 10000 milliseconds after date.
```

The following program displays the number of milliseconds taken to execute the statement `<do something>` :

```
before = new Date();
<do something>;
after = new Date();
writeln("Time for doing something: ", after-before, " milliseconds.");
```

Tips on date formats

Provides some tips on using ILOG Script to improve the date format.

Setting the date

The set methods on dates perform safety checks. For example, when you try to set the day, the corresponding month must have the correct number of days. If the date is 2011/ February/15 and you try to change the day to 31, an error will occur. Therefore, to avoid errors when you create a new date, you must respect the order: Year / Month / Day.

Modifying the date format

If your date is not in the format you want, you need to use ILOG Script to modify it. See the following example.

```
execute {
  function converter(dateIn) {
    dateTime = dateIn.split(" ");
    date = dateTime[0]; time = dateTime[1];

    dateSplit = date.split("-");
    dateReverse = dateSplit.reverse();
    dateNew = dateReverse.join("/") + " " + time;
    return Date.parse(dateNew);
  }

  var d1 = converter("2011-07-06 07:00:00.0");
  writeln(d1);
}
```

This code will return a new date in the format day/month/year. So in the example, the date returned is:

06/07/2011 07:00:00.0

The null value

Explains the use of the null value.

The null value is a special value used in some places to specify an absence of information.

For example, an array element which has not yet been set has a default value of null. The null value is not to be confused with the undefined value, which also specifies an absence of information in some contexts. See section “The undefined value” below.

The null value can be referenced in programs with the keyword `null` :

`null` -> the null value

When converted to a number, `null` yields zero (0).

Methods of null

There is only one method of `null`.

Table 40. Methods of null

Syntax	Effect
<code>null.toString()</code>	Returns the string "null".

The undefined value

Explains the use of the undefined value.

The undefined value is a special value used in some places to specify an absence of information. For example, accessing a property of a value which is not defined, or a local variable which has been declared but not initialized, yields the undefined value.

There is no way of referencing the undefined value in programs. Checking if a value is the undefined value can be done using the `typeof` operator:

```
typeof(value) == "undefined" -> true if value is undefined,  
                                false otherwise.
```

Methods of undefined

There is only one method of undefined.

Table 41. Methods of undefined

Syntax	Effect
<i>undefined</i> .toString()	Returns the string "undefined".

IBM ILOG Script functions

Describes the use of functions in IBM ILOG Script.

In IBM ILOG Script, functions are regular values (also known as “first class” values) which can be manipulated like any other type of value: They can be passed to functions, returned by functions, stored into script variables or into object properties, and so on.

For example, the function `parseInt` is a function value which is stored in the `parseInt` variable:

```
parseInt ->  
a function value
```

This function value can be, for example, assigned to another variable:

```
myFunction = parseInt
```

and then called through this variable:

```
myFunction("-25") -> -25
```

Function methods

There is only one method of functions.

Table 42. Function methods

Syntax	Effect
<i>function</i> .toString()	Returns a string which contains some information about the function. Examples: "foo".substring.toString() "[primitive method substring]" eval.toString() "[primitive function eval]"

Miscellaneous functions

Provides a reference of miscellaneous functions in IBM ILOG Script.

Table 43. Miscellaneous functions

Syntax	Effect
<code>stop()</code>	Stops the execution of the program at the current statement and, if the debugger is enabled, enters debug mode.
<code>write (arg1, ..., argn)</code> <code>writeln (arg1, ..., argn)</code>	Converts the arguments to strings and prints them to the current debug output. The implementation depends on the application in which IBM ILOG Script is embedded. The function <code>writeln</code> prints a newline at the end of the output, while <code>write</code> does not.
<code>loadFile(string)</code>	Loads the script file whose path is <code>string</code> . The path can be either absolute or relative. If this path does not designate an existing file, the file is looked up using a method that depends on the application in which the script is embedded. Typically, a file with the name <code>string</code> is searched for in a list of directories specified in the application setup.
<code>includeScript(string)</code> For use in OPL.	<p>Loads the script file whose path is <code>string</code>. The path can be either absolute or relative. If this path does not designate an existing file, the file is looked up using a method that depends on the application in which the script is embedded. Typically, a file with the name <code>string</code> is searched for in a list of directories specified in the application setup.</p> <p>The effect of this function is the same as for <code>loadFile</code>. In OPL, you have to use <code>includeScript</code>, as the function <code>loadFile</code> does not recognize OPL objects, such as <code>thisOplModel</code>.</p>
<code>eval(string)</code>	<p>Executes <code>string</code> as a program, and returns the value of the last evaluated expression. The program in <code>string</code> can use all the features of the language, except that it cannot define functions; in other words, the function statement is not allowed in <code>string</code>.</p> <p>Examples:</p> <pre>eval("2*3") -> 6</pre> <pre>eval("var i=0; for (var j=0; j<100; j++) i=i+j; i") -> 4950</pre> <pre>eval("n=25; eval('Math.sqrt(n)')") -> 5</pre> <pre>eval("function foo(x) { return x+1 }") -> error</pre>

Table 43. Miscellaneous functions (continued)

Syntax	Effect
fail()	<p>Stops the execution of the scripting block at the current statement, reports an error, and goes on.</p> <p>Example:</p> <pre>execute b1 { writeln("A"); fail(); writeln("B"); } execute b2 {writeln("C"); }</pre> <p>gives</p> <p>A C</p> <p>as the output</p> <p>and reports an error line 4</p> <p>Scripting runtime error: fail() called.</p>

Index

A

- abs, OPL function 37
- accessing
 - value of an IBM ILOG Script property 89
- accessing named ranges
 - in Excel 25
- aggregate operators 38, 61
- all, OPL keyword 34
- and, logical constraint 55
- arguments
 - IBM ILOG Script keyword 92
- arrays
 - appending 35
 - constructor (IBM ILOG Script) 116
 - explicit 35
 - in IBM ILOG Script 115
 - initialization 17
 - methods (IBM ILOG Script) 118
 - multidimensional 8
 - of decision variables, initialization 36
 - one-dimensional 8
 - properties (IBM ILOG Script) 117
- assert, OPL keyword 29
 - processing order of statements 30
- assertions 29
- assignment operators 90

B

- Boolean expressions 45
 - constraints 46
- Boolean literals 113
 - conversion to 114
- Boolean method toString 114
- brackets, delimiters in IBM ILOG Script
 - for OPL 87
- break, IBM ILOG Script keyword 95
- breakpoint 40
- breakpoints
 - in piecewise-linear functions 39
- building blocks 3

C

- card, OPL function 44
- cardinality constraints 56
- case-sensitivity of the scripting language 88
- collections
 - not sorted in tuple sets without keys 14
- comments
 - delimiters 88
- compatibility constraints in CP 58
- compound statements
 - in IBM ILOG Script for OPL 87
- conditional constraints 47
- conditional expressions
 - for float and integers 37

- conditional statements
 - in IBM ILOG Script for OPL 87, 95
- connection
 - to a spreadsheet 24
- consistency of model data 27
- constants 103
 - dynamic collection 34
- constraint labeling
 - limitations 51
- constraints
 - basic 53
 - conditional 47
 - declaration 46
 - discrete 53
 - element expressions (CP) 57
 - float 53
 - for compatibility (CP) 58
 - for filtering 47
 - labeling 48
 - limitations 59
 - logical (CPLEX) 54
 - logical, for CP 57
 - nonlinear, rejected by CPLEX 53
 - scheduling 58
 - specialized (CP) 58
 - string 54
 - types 53
 - using 46
- constraints, limitations 59
- constructors
 - for IBM ILOG Script arrays 116
 - for IBM ILOG Script dates 122
 - for IBM ILOG Script objects 121
 - user-defined in IBM ILOG Script 121
- continue, IBM ILOG Script keyword 95
- conventions
 - in models 3
- conversion (IBM ILOG Script)
 - of non-Boolean value to a Boolean 114
 - of nonnumeric value to a number 101
 - of nonstring value to a string 107
- costs
 - discontinuous (pwl) 41
- count, OPL function 37

D

- data
 - assertions for consistency 29
 - consistency 27
 - initializing 15
 - input/output
 - to/from a spreadsheet 24
 - preprocessing 30
 - reading from an Excel spreadsheet 24
- data initialization, lazy 16, 23, 31
- data structures
 - arrays 8

- data structures (*continued*)
 - ranges 7
 - sets 12
- data types
 - arrays 8
 - floats 4
 - integers 4
 - piecewise linear functions 5
 - ranges 7
 - sets 12
 - stepwise functions 6
 - strings 5
- data types allowed in tuples 11
- data, pushing and pulling 23
- date format, tips using script 125
- dates in IBM ILOG Script 122
 - constructor 122
 - functions 125
 - methods 124
 - operators 125
- decimal numbers 100
- decision expressions 34
 - and tuple patterns/tuple indices 11
- decision variables 32
 - and integer ranges 7
 - arrays of, initialization 36
 - definition 32
 - dynamic collection 34
 - reusable (dexpr) 34
- declaration
 - of constraints 46
 - of script variables
 - inside a function definition 97
 - outside a function definition 97
 - of tuples, using keys 10
- default values
 - of statements in IBM ILOG Script 99
- delimiters
 - and internal initialization
 - tuples 20
 - curly brackets 87
 - for comments 88
 - for tuples 19
 - quotes 106, 113
 - semi-colon 87
- dexpr, OPL keyword 34
- diff, OPL keyword 44
- discontinuous piecewise-linear functions 40
- discrete
 - constraints 53
 - data 53
 - variables 53
- dvar, OPL keyword
 - vs. var 32

E

- efficient models 48, 61, 62
- element expressions in CP 57

- else
 - See if-then-else 47
- else, IBM ILOG Script keyword 95
- equivalence, logical constraint 55
- errors
 - tuples with same names 10
- eval function 128
- Excel
 - accessing named ranges 25
- Excel spreadsheet
 - what data can be read 24
- Excel, R1C1 style not supported 24, 26
- execute, IBM ILOG Script block
 - for preprocessing 30
- expressions 88
 - Boolean 45
 - counting 37
 - float 37
 - in IBM ILOG Script for OPL 87
 - in logical constraints 54
 - integer 37
 - set 44
 - syntax 89
- external data 16
 - and memory allocation 24
 - sets 21

F

- fail function 128
- filtering
 - formal parameters 59
 - in tuples of parameters 62
 - with constraints 47
- first, OPL function 44
- float constraints 53
- float, OPL keyword 4
- floats 4
 - expression 37
 - functions in OPL 37
- for, IBM ILOG Script keyword
 - loop syntax 95
- forall, statements
 - and constraint labels 51
- formal parameters
 - basic 59
 - filter expressions 59
 - tuples 61
- function calls 91
- function definitions 98
- function, IBM ILOG Script keyword 98
- functions 127
 - card 44
 - first 44
 - float 37
 - for dates 125
 - for strings 110
 - item 44
 - last 44
 - next 44
 - nextc 44
 - ord 44
 - over set expressions 44
 - prev 44
 - prevc 44
 - sign 41

- functions in ILOG Script
 - miscellaneous 128
- functions, numeric 102

G

- generic sets 21, 61
- ground
 - breakpoints and slopes in
 - piecewise-linear functions 39
 - conditions in if-then-else
 - statements 47
 - expressions and relations 36

H

- hexadecimal numbers 100

I

- IBM ILOG Script
 - compound statements 87
 - default values of statements 99
 - syntax 87
- identifiers 88
 - conventions 3
 - for data and variables 36
- if-then-else
 - conditional constraints 47
- if, IBM ILOG Script keyword 95
- implication
 - logical constraint (CP) 57
 - logical constraint (CPLEX) 55
- implicit slicing 62
- includeScript function 128
- indexed labels 49
- infinity 100
- infinity, OPL keyword 4, 37
- initializing
 - arrays 17
 - arrays of decision variables 36
 - data 15, 16
 - set of tuples 16
 - sets 21
 - tuples 19
- input/output
 - data to/from a spreadsheet 24
- integer constant maxint 4
- integer expressions 37
- integer ranges 7
- integers 4
- inter, OPL keyword 44
- internal data 16
 - and memory allocation 23
 - sets 21
- item function 44

K

- keys in tuple declarations 10
- keywords
 - arguments 92
 - assert 29
 - break 95
 - continue 95

keywords (*continued*)

- diff 44
- dvar 32
- else 95
- float 4
- for 95
- function 98
- if 95
- infinity 4, 37
- inter 44
- max 38
- maxint 37
- min 38
- null 126
- ordered 59
- piecewise 39
- prod 38
- return 98
- setof 12
- SheetConnection 24
- SheetRead 24
- special 92
- static 98
- string 5
- sum 38
- syndiff 44
- this 92, 120, 121
- tuple 9
- union 44
- var 97
- with 27

L

- labeled assertions 50
- labeled constraints 48
- last, OPL function 44
- lazy initialization of data 16, 23, 30
- limitations on constraint labeling 51
- limitations on constraints 59
- limitations on tuples 11
- literals 88
- loadFile function 128
- logical constraints
 - definition and extraction 54
 - for CP 57
- logical expressions 54
- logical operators 114
- loops
 - in IBM ILOG Script 95

M

- max, OPL keyword 38
- maxint, OPL keyword 4, 37
- memory allocation and management
 - and data initialization 23
- memory consumption
 - and multidimensional arrays 8
- methods
 - built-in, for objects 121
 - defining for objects 120
 - for arrays 118
 - for Booleans 114
 - for dates 124
 - for functions 127

- methods (*continued*)
 - for numbers 102
 - for strings 108
 - for the null value 126
 - for the undefined value 127
- methods for sets 13
- min, OPL keyword 38
- models
 - building 3
 - conventions 3
 - efficiency 48
 - readability 61, 62
- multidimensional arrays 8

N

- NaN (Not-a-Number) 100
- next, OPL function 44
- nextc, OPL function 44
- nonlinear constraints 53
- nonlinear expressions in logical constraints 55
- Not-a-Number (NaN) 100
- not, logical constraint 55
- null value 126
- null, IBM ILOG Script keyword 126
- numbers 100
 - automatic conversion to 101
 - methods 102
- numeric constants 103
- numeric functions 102
- numeric operators 104

O

- objective function
 - and decision variables 32
- objects 120
 - constructor 121
- octal numbers 100
- one-dimensional arrays 8
- operations on sets 12
- operators
 - aggregate 38
 - assigning a value 90
 - for IBM ILOG Script dates 125
 - for strings 112
 - logical 114
 - numeric 104
 - precedence 89
 - shorthand 90
 - special 92
 - syntax 93
- or, logical constraint 55
- ord, OPL function 44
- order
 - for processing script blocks 30
- ordered sets 12
 - special ordered sets, not supported 12
- ordered, OPL keyword 59
- overflow
 - and integer expressions 37

P

- piecewise linear functions
 - pwlFunction 5
- piecewise-linear functions 39
 - discontinuous 40
- piecewise, OPL keyword 39
- preprocessing
 - data 30
- prev, OPL function 44
- prevc, OPL function 44
- processing order
 - preprocessing items 30
- prod, OPL keyword 38
- program unit, and local variables (scripting) 97
- properties
 - accessing value of 89
 - for IBM ILOG Script arrays 117
 - for strings 108
- pushing and pulling data 23
- pwlFunction, OPL keyword 5

Q

- quotes 106, 113

R

- R1C1 style not supported in Excel 24, 26
- range float, data type 7
- ranges 7
 - and set expressions 44
- reading
 - from a spreadsheet 24
- return, IBM ILOG Script keyword 98

S

- scheduling constraints in CP 58
- scope hiding 63
- scope of script variables 97
- script functions
 - miscellaneous 128
- script variables
 - declaration 97
 - inside a function definition 97
 - outside a function definition 97
 - reference to 89
 - scope 97
- semi-colon 87
- set expressions 44
 - and ranges 44
 - construction 44
 - functions 44
- setof, OPL keyword 12
- sets 12
 - allowed operations 12
 - and data consistency 27
 - and sparsity 21
 - generic 21
 - initializing 21
 - methods for sets 13
 - of tuples, initialization 16
 - ordered versus sorted 13
- SheetConnection, OPL keyword 24

- SheetRead, OPL keyword 24
- shorthand operators 90
- sign function 41
- slicing
 - explicit/implicit 62
 - using key fields 10
- slopes in piecewise-linear functions 39
- sorted sets 13
- sorted tuple sets 14
- sparsity
 - and multidimensional arrays 8
 - and one-dimensional arrays 8
 - and sets 21
- special numbers 100
- special ordered sets, not supported 12
- specialized constraints 58
- spreadsheets
 - accessing named ranges 25
 - connecting to 24
 - data input/output 24
 - reading from 24
 - writing to 26
- statements 95
 - conditional 95
 - in IBM ILOG Script for OPL 87
 - last 87
- static, IBM ILOG Script keyword 98
- stepFunction, OPL keyword 6
- stepwise functions
 - stepFunction 6
- stop function 128
- string constraints 54
- string, OPL keyword 5
- strings 5, 106
 - automatic conversion to 107
 - functions 110
 - length 99
 - methods 108
 - operators 112
 - properties 108
- structs in C, tuples in OPL 9
- sum, OPL keyword 38
- syndiff, OPL keyword 44
- syntax, in IBM ILOG Script 87
 - accessing property values 89
 - assignment operators 90
 - conditional statements 95
 - default values 99
 - expressions 89
 - function definition 98
 - identifiers 88
 - loops 95
 - other operators 93
 - reference to a script variable 89
 - shorthand 91
 - special keywords 92
 - special operators 92
 - variable declaration 97

T

- then
 - See if-then-else 47
- this, IBM ILOG Script keyword 92, 120, 121
- time representation in IBM ILOG Script 122

- tuple indices in dexpr 11
- tuple indices in labeled constraints 12
- tuple patterns
 - in decision expressions 11
- tuple sets
 - external initialization 16
 - referring to other sets with keys 28
 - sorted 13
- tuple sets, sorted 14
- tuple, OPL keyword 9
- tuples 9
 - and data consistency 27
 - data types allowed, not allowed 11
 - duplicates in collections not
 - detected 11
 - initialization 19
 - keys in declaration 10
 - of parameters 61
 - filtering 62
- tuples, limitations 11

U

- undefined value 126
- union, OPL keyword 44

V

- values
 - and functions (IBM ILOG Script) 115
- values, in IBM ILOG Script
 - and functions
 - Booleans 113
 - dates 122
 - decimal numbers 100
 - hexadecimal numbers 100
 - numbers 100
 - objects 120
 - special numbers 100
 - strings 106
 - assignment operators 90
 - default 99
 - null 126
 - of properties, accessing 89
 - undefined 126
- values, in ILOG Script
 - and functions
 - octal numbers 100
- var, IBM ILOG Script keyword 97
- variables
 - See* script variables

W

- while, IBM ILOG Script keyword
 - loop syntax 95
- with, IBM ILOG Script keyword 99
- with, OPL keyword 27



Printed in USA