IBM ILOG OPL V6.3

IBM ILOG OPL Language User's Manual

Copyright

Trademarks

Acknowledgement

The language manuals are based on, and include substantial material from, The OPL Optimization Programming Language by Pascal Van Hentenryck, © 1999 Massachusetts Institute of Technology.

Table of contents

Language User's Manual

Describes how to use OPL, the IBM® ILOG® Optimization Programming Language. The language is documented in two manuals (the *Language User's Manual* and the *Language Reference Manual*), both partly based on Pascal Van Hentenryck's book, *The OPL Optimization Programming Language*, published by The MIT Press, 1999, Cambridge, Massachusetts. This *Language User's Manual* is composed mostly of tutorials for both OPL and IBM ILOG Script for OPL.

In this section

Introduction to OPL

Introduces modeling languages in general, then gives a short tour of the OPL modeling language, discusses some modeling issues, and finally illustrates optimization modeling with two examples.

The application areas

Describes applications of linear and integer programming, constraint programming, quadratic programming, and CPLEX® logical constraints.

IBM ILOG Script for OPL

After an introduction to scripting, provides tutorials for flow control and multiple searches, flow control and column generation, and for changing default behaviors in flow control.

Advanced features

A tutorial on external functions.

Performance and memory usage

Recommends practices that are known to improve the modeling and the solving time of your models and/or their ability to find good solutions.

Introduction to OPL

Introduces modeling languages in general, then gives a short tour of the OPL modeling language, discusses some modeling issues, and finally illustrates optimization modeling with two examples.

In this section

Language overview
Explains why modeling languages were created, presents and compares math programming and constraint programming, and provides a brief introduction to scripting with references for more information.

A short tour of OPL
Give readers a preliminary understanding of the language and shows how OPL supports linear programming (production planning problem), integer programming (knapsack problem), mixed integer-linear programming (a blending problem), and constraint programming (inventory matching problem). See also *Quadratic programming* and *Getting Started with Scheduling in IBM ILOG OPL*.

Modeling tips
Describes a few recommended practices to help you write more efficient models.

Language overview

Explains why modeling languages were created, presents and compares math programming and constraint programming, and provides a brief introduction to scripting with references for more information.

In this section

Modeling languages
Provides a general introduction to modeling languages.

Mathematical programming
Defines linear programming, integer programming, and nonlinear programming.

Constraint programming
Explains what leads to the development of constraint programming and briefly presents OPL CP Optimizer.

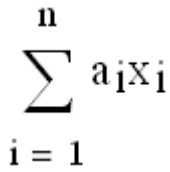
Constraint programming versus mathematical programming
Explains why it makes sense to compare CP and MP, and provides details on the salient features of each approach.

Scripting
Defines scripting languages in general and IBM® ILOG Script in particular.

Modeling languages

Modeling languages were motivated by the desire to simplify the solving of mathematical programming problems. The fundamental insight underlying traditional modeling languages is the recognition that many mathematical programming problems can be expressed in a computer language whose syntax is close to the standard presentation of these problems in textbooks and scientific papers. These languages typically provide a number of data types such as arrays and sets, as well as computer-language equivalents to traditional algebraic notations.

For instance, in AMPL, an expression such as



can be written as

```
sum {i in 1..n} a[i] * x[i]
```

In addition, some of these languages provide a clean separation between the model and the instance data.

Finally, they are sometimes extended by a command language that makes it possible to solve sequences of related models and to make modifications to the models and solve the modified models. Traditional modeling languages have many benefits that make them appealing for stating and solving mathematical programming problems. Perhaps their most significant contribution is to provide a language that directly supports the natural statement of these problems. This language abstracts away the implementation details of the underlying solver and users are then relieved of mundane, low-level, considerations and can focus on the modeling of their applications. Also important is the clear separation between the model and the instance data, which ensures that the same model can be applied to many instances without inducing additional work. Note that, in these languages, the solver is a black box that can only be accessed through a set of well-defined parameters.

Traditional modeling languages are particularly strong in mathematical programming applications, e.g., linear and integer programming. This is not surprising since this is the area from where they emerged. In addition, these problems are naturally expressed using traditional algebraic notations and effective solvers are available to solve the resulting models.

Mathematical programming

Defines linear programming, integer programming, and nonlinear programming.

In this section

Linear programming
Defines linear programming.

Integer programming
Defines integer programming.

Nonlinear programming
Defines nonlinear programming.

Linear programming

Linear programming is an important tool for combinatorial search problems, not only because it solves efficiently a large class of important problems, but also because it is the basic block of some fundamental techniques in this area.

A linear program consists of minimizing a linear objective function subject to a set of linear constraints over real variables constrained to be nonnegative or, in symbols,

$$\begin{array}{ll} \text{minimize} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \\ & \sum_{j=1}^{n} a_{ij} x_j = b_i \quad (1 \leq i \leq m) \\ & x_j \geq 0 \qquad\qquad (1 \leq j \leq n). \end{array}$$

Note first that considering only equations, nonnegative variables, and minimization is not restrictive. An inequality `t ≥ 0` can be recast as an equation `t - s = 0` by adding a new variable, an arbitrary variable can be expressed as the difference of two nonnegative variables, and maximization can be expressed by negating the objective function. In addition, decision problems (i.e., finding if a set of constraints is satisfiable) can be recast by adding a variable per constraint and minimizing their sum. The problem is satisfiable if and only if the optimum is zero. Note also that linear programs can be solved in polynomial time and robust solvers are now available that solve large scale linear programs. The success of linear programming led many researchers to investigate some of its generalizations.

Integer programming

Integer programming is a natural extension of linear programming where variables are required to take integer values.

$$
\begin{array}{ll}
\text{minimize} & \sum_{j=1}^{n} c_j x_j \\
\text{subject to} & \\
 & \sum_{j-1}^{n} a_{ij} x_j = b_i \quad (1 \leq i \leq m) \\
 & x_j \geq 0 \quad (1 \leq j \leq n) \\
 & x_j \text{ integer} \quad (1 \leq j \leq n).
\end{array}
$$

Unfortunately, these integrality constraints make the problem NP-complete. Integer programming has been investigated extensively in the past decades and good solvers are now available for various classes of integer programs. However, many integer programs remain challenging from a computational standpoint.

Nonlinear programming

Nonlinear programming is another generalization of linear programming that amounts to minimizing a nonlinear function subject to nonlinear constraints.

In other words:

```
minimize g(x1,...,xn)
subject to f1(x1,...,xn) ≥ 0
....
           fm (x1,...,xn) ≥ 0
```

where `g, f 1 ,..., fm` are real functions of *n* variables. Nonlinear programs are generally very challenging from a computational standpoint; local methods are often used to solve them, sacrificing optimality for speed of execution. Note also that integer programs can be recast as nonlinear programs.

Constraint programming

Explains what leads to the development of constraint programming and briefly presents OPL CP Optimizer.

In this section

Why constraint programming?
Describes the evolution of constraint programming.

OPL CP Optimizer in a nutshell
Provides the basics of using constraint programming in an OPL model.

Why constraint programming?

Constraint programming is a native satisfiability technology that takes its roots in computer science—logic programming, graph theory, and the artificial intelligence efforts of the 1980s. Recent progress in the development of tunable and robust black-box search for constraint programming engines have turned this technology into a powerful and easy-to-use optimization technology.

Constraint programming has proven very efficient for solving scheduling problems. *Getting Started with Scheduling in IBM ILOG OPL*

is an introductory tutorial on the use of constraint programming-based scheduling in OPL. The “Language Reference Manual” provides more information.

Constraint programming is also an efficient approach to solving and optimizing problems that are too irregular for mathematical optimization. This includes time tabling problems, sequencing problems, and allocation or rostering problems.

The reasons for these irregularities that make the problem difficult to solve for mathematical optimization can be:

- Constraints that are nonlinear in nature
- A non convex solution space that contains many locally optimal solutions
- Multiple disjunctions, which result in poor information returned by a linear relaxation of the problem

Read *Constraint programming versus mathematical programming* for a detailed comparison.

OPL CP Optimizer in a nutshell

A CP model must be declared as such. It uses only discrete decision variables for which you must define a domain. The product of all domain sizes makes up the search space. CP models are further characterized by specific constraints and expressions, and parameterizable propagation, and search.

If you already have experience of constraint programming with OPL 3.x, read also *Data preprocessing* in *Migration from OPL 3.x (CP projects)*.

Here are the basics of using constraint programming in an OPL model. Read the CP Optimizer documentation for details.

1. **Declaration**: A CP model must start with the statement

   ```
   using CP;
   ```

   Otherwise, it will be considered an MP model solved by the CPLEX® engine.

2. **Decision variables:** Use only discrete variables as decision variables.

3. **Decision expressions:** It is possible to constrain floating point expressions, or to use them as an objective term. It is also possible to declare floating-point expressions with the `dexpr` keyword, as in the `floatexpr.mod` code sample.

4. **Domain definition:** From OPL 5.2 onwards, you must define domains for your decision variables. CP does not work well with undefined domains.

5. **Search space:** The search space is the product of all domain sizes, measured by its log. The search space is a measure of how difficult a problem is for the CP Optimizer engine. It is also a limit for the trial version. See the *Licensing Scheme* document.

6. **Constraints and expressions:** Specific arithmetic, logical, temporal, and specialized constraints and expressions are supported by the CP Optimizer engine for CP combinatorial and scheduling models. See *Constraints available in constraint programming* in the *Language Reference Manual*.

7. **Parameters:** You can set various parameters for propagation control, log control, search control, and so on. See Constraint programming options in *Parameters and settings in OPL*.

8. **Propagation:** Constraints in CP model are propagated at execution time by the CP solving engine. Constraint propagation is the process of communicating the domain reduction of a decision variable to all of the constraints that are stated over this variable. This process can result in more domain reductions. These domain reductions, in turn, are communicated to the appropriate constraints. This process continues until no more variable domains can be reduced or when a domain becomes empty. In the latter case, an empty domain means that the model has no solution.

9. **Search:** To find a solution, the CP Optimizer search functionality implicitly generates combinations of values for decision variables by means of *constructive strategies*. these strategies are executed and guided towards optimal solutions in order to converge rapidly. The CP Optimizer search uses a variety of guides and uses the most appropriated one depending on the model structure and on constraint propagation. The powerful default search gives satisfactory results in most cases but in specific cases, you can fine-tune search strategies.

10. **Search phases:** The engine parameters modify the search behavior in a global way. The impact of the parameter is the same on all parts of the model. Sometimes, a useful knowledge of some part of the model can be used to modify how the search should be performed on a limited part of the model. In that case, search modifiers can be used to apply some kind of local search settings on this limited part. They are applied by specifying which modifiers are to be used on which variables at each phase. You can specify as many phases as you want.

    See *Using IBM ILOG Script in constraint programming*.

See also

Scheduling with IBM ILOG OPL
Scheduling
Constraints

Constraint programming versus mathematical programming

Explains why it makes sense to compare CP and MP, and provides details on the salient features of each approach.

In this section

Why a comparison?
Summarizes the differences between CP and MP.

Benefits of constraint programming
Technology that solves time tabling problems and sequencing problems. It can also be an alternative to mathematical programming for allocation problems that have a slow convergence.

Differences with mathematical programming
Describes what is required by constraint programming, in contrast with math programming.

Why a comparison?

CP works with the same concepts as mathematical programming: decision variables, objective function, and constraints. However, there are some differences between CP models and MP models.

In short:

- CP models have only discrete decision variables (integer or Boolean) while MP models support both discrete and continuous decision variables.
- CP models natively support logical constraints as well as a full range of arithmetic expressions including modulo, integer division, or the `element` expression which indexes an array of values by a decision variable. In contrast, MP models support only linear constraints, linearized logical constraints, or quadratic convex constraints.
- CP models have no limitation on the arithmetic constraints that can be set on decision variables, while an MP engine is specific to a class of problems whose solution space satisfies certain mathematical properties.

  See *http://www.ilog.com/products/cplex/product/algorithms.cfm* for a list of problems supported by IBM® ILOG® CPLEX® .
- Each optimization engine uses different techniques and algorithms to find feasible solutions and optimize them.

Constraint programming vs. mathematical programming

Feature	MP	CP
Relaxation	Yes	No
GAP measure	Yes	No
Optimality proof	Yes	Yes
Modeling limitations	Quadratic problems are limited to PSD (Positive Semi Definite) problems and Second Order Cone Programming (SOCP) problems	Discrete problems
Specialized constraints	No	Yes
Logical constraints	Yes	Yes
Theoretical grounds	Algebra	Graph theory and algorithmic
Modeler support	Yes	Yes
Model and run	Yes	Yes

Benefits of constraint programming

Constraint programming is a suitable technology to solve time tabling problems and sequencing problems. It can also be an alternative to mathematical programming for allocation problems that have a slow convergence.

Constraint programming has native support for:

- *Nonlinear costs or constraints*
- *Logical constraints and statements*
- *Constraints on and between interval variables*
- *Compatibility or incompatibility constraints*
- *More useful features*

Nonlinear costs or constraints

For example, a quadratic assignment problem can be modeled in CP as follows:

A quadratic assignment problem (CP)

```
using CP;

int nbPerm = ...;
range r = 1..nbPerm;
int dist[r][r] = ...;
int flow[r][r] =...;

execute {
      cp.param.timeLimit=30;
}


dvar int perm[1..nbPerm] in r;

dexpr int cost[i in r][j in r] = dist[i][j]*flow[perm[i]][perm[j]];

minimize sum(i in r, j in r) cost[i][j];
subject to {

  allDifferent(perm);

};
```

Logical constraints and statements

For example, `forall` statements such as the following one are efficiently taken into account by constraint programming:

A forall statement in CP

```
  forall(s in 1..nbSlabs)
    colorCt: sum (c in 1..nbColors) (or(o in 1..nbOrders : colors[o] == c)
(where[o] == s)) <= 2;
```

Constraints on and between interval variables

CP scheduling can express several types of constraint on and between interval variables:

- to limit the possible positions of an interval variable (forbidden start/end values 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).

Compatibility or incompatibility constraints

For instance, the following is a concise model of the knight problem:

The knight problem

More useful features

When it comes to computing operational plans or schedules that must be executable, you cannot always use the linear form to simplifying costs or constraints. Fortunately, constraint programming can accurately model these problems.

Constraint programming can also be used as a fast generator of feasible solutions. This can be extremely useful in combination with other models and engines, for instance to implement column generation for a complex optimization model.

Differences with mathematical programming

In contrast with math programming, constraint programming requires:

- Explicit modeling for max, min, abs
- More memory usage per decision variable

and supports:

- Only discrete decision variables
- No gap measure

Explicit modeling for max, min, abs

Since constraint programming does not have linear relaxation to optimize a relaxed problem after each decision on integer variables, the MP way of modeling constraints such as `max`, `min`, cannot be used directly for CP. For instance, the following MP linearization would put the maximum value of `x[i]` in `m`.

```
minimize m + …;
subject to {
   forall(i in 1..10) m >= x[i];
   ….
}
```

In CP, it is safer and more efficient to write:

```
minimize …;
subject to {
m == max(i in 1..10) x[i];
…
}
```

More memory usage per decision variable

For an MP engine, a decision variable is stored as one more column in a matrix. For a CP engine, it may require much more memory, because the CP engine stores domain information in the variable. Therefore, a CP engine scales apparently less than an MP engine, in term of the number of variables and of constraints. However, since the set of constraints of a CP engine enables often a more compact formulation of a problem, there is no direct connection between this property and the size of problems that either engine can address.

Only discrete decision variables

IBM ILOG CP Optimizer engine handles only discrete decision variables. You can use continuous expressions to define costs or intermediate expressions, but these continuous expressions must be computed only from discrete decision variables.

No gap measure

Because the CP Optimizer engine addresses problems that are potentially non convex or too irregular for mathematical optimization, it cannot compute valuable relaxed solutions of a problem, and does not have gap information between the best found solution and a theoretical bound that an MP engine can provide for a linear problem.

Scripting

Modeling languages are sometimes extended by a command language that makes it possible to interact with models, to solve several instances of the same model, or to solve sequences of models. IBM® ILOG® Script is a scripting language for OPL supporting these functionalities..

In other words, while OPL is the language to express optimization, IBM ILOG Script for OPL is the language for the non-modeling aspects (flow control, preprocessing, postprocessing).

The main novelty in IBM ILOG Script is to consider models as first-class objects, providing a clear separation of concerns between models and scripting, and making the overall system compositional. As a consequence, models can be developed, tested, and maintained independently of the scripts using them.

For more information, see:

- *IBM ILOG Script for OPL* in this *Language User's Manual*.
- *IBM ILOG Script for OPL* in the *Language Reference Manual*
- The *Reference Manual for IBM ILOG Script extensions for OPL*

A short tour of OPL

Give readers a preliminary understanding of the language and shows how OPL supports linear programming (production planning problem), integer programming (knapsack problem), mixed integer-linear programming (a blending problem), and constraint programming (inventory matching problem). See also *Quadratic programming* and *Getting Started with Scheduling in IBM ILOG OPL*.

In this section

Linear programming: a production planning example
Explains how OPL expresses LP problems, describes the production planning problem, presents the elements of a production model, shows how results can be displayed, and how to change a parameter value.

Integer programming: the knapsack problem
Explains what integer programming is and describes the knapsack problem.

Mixed integer-linear programming: a blending problem
Presents OPL and MILP and describes a blending problem.

Constraint programming: an inventory matching problem
Gives a short tour of constraint programming support in IBM® ILOG® OPL via an inventory problem and its model elements. Does not contain an overview of more basic modeling features such as arrays, data, aggregation, tuples, etc. For such information, see *Linear programming: a production planning example* in this manual. For tutorials that introduce scheduling problems in OPL, see *Getting Started with Scheduling in IBM ILOG OPL*.

Linear programming: a production planning example

Explains how OPL expresses LP problems, describes the production planning problem, presents the elements of a production model, shows how results can be displayed, and how to change a parameter value.

In this section

How OPL expresses an LP problem
Describes a typical optimization problem.

The production planning problem
Describes a linear programming problem.

Elements of the production model
Describes the details of this linear programming model.

Displaying results
Describes how to display results by writing an execute IBM® ILOG Script block.

How OPL expresses an LP problem

An optimization problem is typically specified by an objective function and a set of constraints over some decision variables. A solution to the problem is an assignment of values to the variables that satisfies the constraints and optimizes the value of the objective function. The purpose of an OPL statement is thus to express these two components for the application at hand.

Note: In OPL 4.0 and later, the keyword `dvar` is used to note decision variables in the OPL modeling language while the keyword `var` is used for IBM® ILOG Script variables.

The production planning problem

Consider a Belgian company Volsay, which specializes in producing ammoniac gas (NH3) and ammonium chloride (NH4Cl). Volsay has at its disposal 50 units of nitrogen (N), 180 units of hydrogen (H), and 40 units of chlorine (Cl). The company makes a profit of 40 Euros for each sale of an ammoniac gas unit and 50 Euros for each sale of an ammonium chloride unit. Volsay would like a production plan maximizing its profits given its available stocks.

The OPL statement shown in *A simple production problem (`volsay.mod`)* formalizes this problem.

A simple production problem (`volsay.mod`)

```
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;
}
```

This statement declares two real decision variables, `gas` and `chloride`, representing the production of ammoniac gas and ammonium chloride. These variables are of type `float`. The objective function

```
maximize
  40 * Gas + 50 * Chloride;
```

states that the profit must be maximized. The constraints ensure that the production plan does not exceed the available stocks of nitrogen, hydrogen, and chlorine, respectively. The constraint `gas + chloride <= 50` represents the capacity constraint for nitrogen, since each unit of ammoniac gas and of ammonium chloride uses one unit of nitrogen. The next two constraints, for hydrogen and chlorine respectively, are similar in nature. As mentioned at the beginning of this section, a solution to an optimization problem is typically an assignment of values to the variables that satisfies the constraints and optimizes the objective function.

Note that in *A simple production problem (`volsay.mod`)*, the constraints are identified with so-called “labels”. It is recommended to label constraints in a model. See *Constraints* in the *Language Reference Manual* for details.

A solution to volsay.mod

For the Volsay production-planning problem, OPL returns the optimal solution

```
Final Solution with objective 2300.0000:
```

```
gas = 20.0000;
chloride = 30.0000;
```

Elements of the production model

The `volsay` model shown in *A simple production problem (`volsay.mod`)* is a linear programming model. Linear programming is the class of problems that can be expressed as the optimization of a linear objective function subject to a set of linear constraints (i.e., linear equations and inequalities) over real numbers. Linear programming models can be solved for large numbers of variables and constraints and are, from a computational standpoint, the simplest applications considered in this manual.

This section examines:

- Arrays
- Data declarations
- Aggregate operators and quantifiers
- Isolating the data
- Data initialization
- Tuples
- Displaying results
- Setting CPLEX parameters
- Integer programming: the knapsack problem
- Mixed integer-linear programming: a blending problem

For more information

Applications of linear and integer programming studies the application of OPL to linear programming, integer programming, mixed-integer linear programming, and piecewise-linear programming.

Arrays

The above statement is very specific to the application at hand. In general, it is desirable to write generic models that can be extended, modified easily, and applied in different contexts. The next sections describe a number of OPL concepts to simplify the process of creating such models. A first step towards more genericity is the use of arrays, which makes it easier, for instance, to accommodate new products in the future.

The Volsay production planning model can be rewritten using arrays as:

The volsay production model with arrays

```
{string} Products =  {"gas","chloride"};
dvar float production[Products];
maximize
      40 * production["gas"] + 50 * production["chloride"];
subject to {
      production["gas"] + production["chloride"] <= 50;
```

```
      3 * production["gas"] + 4 * production["chloride"] <= 180;
      production["chloride"] <= 40;
}
```

This new statement illustrates several features of the language. First, the instruction

```
{string} Products =  {"gas","chloride"};
```

declares a set of strings `Products` that represents the set of products of the company. The declaration

```
dvar float production[Products];
```

declares an array of two decision variables, `production["gas"]` and `production["chloride"]`, to represent the optimal production of ammoniac gas and ammonium chloride. These decision variables are used in the rest of the statement, which remains essentially the same as in *A simple production problem (`volsay.mod`)*. As will become clear subsequently, one of the novel features of OPL is the generality of its arrays: OPL arrays can have an arbitrary number of dimensions and their index sets can be arbitrary finite sets, possibly involving complex data structures.

Data declarations

A second fundamental step towards more genericity in the model amounts to representing the problem data explicitly. In addition to the products, the problem data obviously consists of the components (nitrogen, hydrogen, and chloride), the demand of each product for each component, the profit of each product, and the stock available for each component. *Declaring and initializing data (`gas.dat`)* declares and initializes these data:

Declaring and initializing data (`gas.dat`)

```
Products = { "gas" "chloride" };
Components = { "nitrogen" "hydrogen" "chlorine" };

Demand = [ [1 3 0] [1 4 1] ];
Profit = [30 40];
Stock = [50 180 40];
```

The data element `Components` is a set of strings that defines the chemical components necessary for the products, `demand` is a two-dimensional array whose element `demand[p][c]` represents the demand of product `p` for component `c`, and `profit` and `stock` are two arrays representing the profit of each product and the stock available for each component. The rest of the statement can be obtained easily by replacing the numbers by the relevant data items. For instance, the objective function is simply written as

```
maximize
  sum( p in Products )
    Profit[p] * Production[p];
```

Aggregate operators and quantifiers

It should be clear, however, that the statement above contains much redundancy. All constraints, and all arithmetic terms in these constraints and in the objective function, are similar: they differ only in their indices.

OPL has two features to factorize these commonalities, aggregate operators and quantifiers, as shown in *A simple production model (`gas1.mod`)*.

A simple production model (`gas1.mod`).

```
{string} Products = { "gas", "chloride" };
{string} Components = { "nitrogen", "hydrogen", "chlorine" };

float Demand[Products][Components] = [ [1, 3, 0], [1, 4, 1] ];
float Profit[Products] = [30, 40];
float Stock[Components] = [50, 180, 40];

dvar float+ Production[Products];

maximize
  sum( p in Products )
    Profit[p] * Production[p];
subject to {
  forall( c in Components )
    ct:
      sum( p in Products )
        Demand[p][c] * Production[p] <= Stock[c];
}
```

The objective function

```
maximize
  sum( p in Products )
    Profit[p] * Production[p];
```

illustrates the use of the aggregate operator `sum` to take the summation of the individual profits. A variety of aggregate operators are available in OPL, including `sum`, `prod`, `min`, and `max`.

The instruction

```
subject to {
  forall( c in Components )
    ct:
      sum( p in Products )
        Demand[p][c] * Production[p] <= Stock[c];
}
```

shows how the universal quantifier `forall` can be used to state closely related constraints. It generates one constraint for each chemical component, each constraint stating that the total demand for the component cannot exceed its available stock. OPL supports rich parameter specifications in aggregate operators and quantifiers (see *Expressions* in the *Language Reference Manual*).

Isolating the data

Another fundamental step in making models reusable is to separate the model and the instance data. OPL supports this clean separation through the notion of *projects*.

A project is the association of a model file, one or more data files (optional), and one or more settings files (optional), associated in run configurations. A minimal project has one run configuration containing only one model. Model files use the file name extension `.mod` while data files use the file name extension `.dat`. The model declares the data but does not initialize it. The data files contain the initialization instructions for each declared data item. See *Understanding OPL projects* in *Quick Start*.

Here we do not describe the details of IBM® ILOG® OPL, but generally describe applications by giving the model and the instance data separately.

For instance, *The production model (`gas.mod`)* and *Instance data for the production model (`gas.dat`)* together make up a project for the Volsay production-planning problem. The model part is essentially the same as the one presented earlier in *Linear programming: a production planning example*, except that it declares the data but does not initialize it.

The production model (`gas.mod`)

```
{string} Products = ...;
{string} Components = ...;

float Demand[Products][Components] = ...;
float Profit[Products] = ...;
float Stock[Components] = ...;
dvar float+ Production[Products];

maximize
  sum( p in Products )
    Profit[p] * Production[p];
subject to {
  forall( c in Components )
    ct:
      sum( p in Products )
        Demand[p][c] * Production[p] <= Stock[c];
}
```

A declaration of the form

```
float profit[Products] = ...;
```

declares the array `profit` and specifies that its initialization is given in a data file. The data file simply associates an initialization with each non-initialized piece of data.

Instance data for the production model (`gas.dat`)

```
Products = { "gas" "chloride" };
Components = { "nitrogen" "hydrogen" "chlorine" };

Demand = [ [1 3 0] [1 4 1] ];
Profit = [30 40];
Stock = [50 180 40];
```

Data initialization

OPL offers a variety of ways of initializing data. One particularly useful feature is the possibility of associating indices with values to avoid various kinds of errors. *Instance data with indices for the production model (`gasn.dat`)* illustrates this feature on the instance data for the Volsay production model.

Instance data with indices for the production model (`gasn.dat`)

```
Products = { "gas", "chloride" };
Components = { "nitrogen", "hydrogen", "chlorine" };

Profit = #["gas":30, "chloride":40]#;
Stock = #["nitrogen":50, "hydrogen":180, "chlorine":40]#;
Demand = #[
            "gas":      #[ "nitrogen":1 "hydrogen":3 "chlorine":0 ]#,
            "chloride": #[ "nitrogen":1 "hydrogen":4 "chlorine":1 ]#
          ]#;
```

The initialization

```
profit = #["gas":30 "chloride":40]#;
```

describes the initialization of array `profit` by associating the value 30 with index `gas` and the value 40 with index `chloride`. (Of course, the order of the pairs has no importance in these initializations.) When using `index:value` pairs, the delimiters `#[` and `]#` must be used instead of `[` and `]`. Note also that, in data files, the items can be initialized in any order and the commas can be omitted freely.

Tuples

OPL offers a variety of data structures in addition to arrays and sets of strings. Tuples, a fundamental tool for structuring the application data, offer an alternative to the traditional approach of representing data in parallel arrays. To see the use of tuples in OPL, consider the following production-planning model. To meet the demands of its customers, a company manufactures its products in its own factories (*inside* production) or buys them from other companies (*outside* production).

Inside production is subject to some resource constraints: each product consumes a certain amount of each resource. In contrast, outside production is theoretically unlimited. The problem is to determine how much of each product should be produced inside and outside the company while minimizing the overall production cost, meeting the demand, and satisfying the resource constraints. *A production-planning problem (`production.mod`)* below depicts an OPL model for this problem that uses only the concepts introduced so far, and *Data for the production-planning problem (`production.dat`)* presents the data for a specific instance.

A production-planning problem (`production.mod`)

```
{string} Products = ...;
{string} Resources = ...;

float Consumption[Products][Resources] = ...;
float Capacity[Resources] = ...;
float Demand[Products] = ...;
```

```
float InsideCost[Products] = ...;
float OutsideCost[Products]  = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

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];
}
```

An instance of the problem must specify the products, the resources, the capacity of the resources, the demand for each product, the consumption of resources by the different products, and the inside and outside costs of each product. These various data items are specified in the standard way in *Data for the production-planning problem (`production.dat`)* below. The model contains two arrays of variables: Element `inside[p]` (respectively `outside[p]`) represents the inside (respectively outside) production of product `p`. The objective function specifies that the production cost must be minimized.

Data for the production-planning problem (`production.dat`)

```
Products =  { "kluski", "capellini", "fettucine" };
Resources = { "flour", "eggs" };

Consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
Capacity = [ 20, 40 ];
Demand = [ 100, 200, 300 ];
InsideCost = [ 0.6, 0.8, 0.3 ];
OutsideCost  = [ 0.8, 0.9, 0.4 ];
```

The production cost is simply the sum of the individual production costs, which are obtained by multiplying the inside and outside productions of the given product by their respective costs. Finally, the model has two types of constraints. The first set of constraints expresses the capacity constraints, the second set states the demand constraints. The model is once again a linear programming problem.

A solution to production.mod

For the instance data given in *Data for the production-planning problem (`production.dat`)*, OPL outputs the following solution:

```
Final Solution with objective 372.0000:
  inside = [40.0000 0.0000 0.0000];
  outside = [60.0000 200.0000 300.0000];
```

Although the model is simple, it is inconvenient in separating the data associated with each product in different arrays: for instance, array `demand` stores the demand for the products, while array `insideCost` stores their inside costs. This technique, sometimes called *parallel arrays*, may be error-prone and less readable for more complicated models. Tuples provide a simple way to cluster related data and impose more structure on a model. This is illustrated in *The production-planning problem revisited (`product.mod`)* and *Data for the production-planning problem (`production.dat`)* below, which exhibit an alternative model for the production-planning problem.

The production-planning problem revisited (`product.mod`)

```
{string} Products = ...;
{string} Resources = ...;
tuple productData {
   float demand;
   float insideCost;
   float outsideCost;
   float consumption[Resources];
}
productData Product[Products] = ...;
float Capacity[Resources] = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

execute CPX_PARAM {
  cplex.preind = 0;
  cplex.simdisplay = 2;
}


minimize
  sum( p in Products )
    (Product[p].insideCost * Inside[p] +
    Product[p].outsideCost * Outside[p] );
subject to {
  forall( r in Resources )
    ctInside:
      sum( p in Products )
        Product[p].consumption[r] * Inside[p] <= Capacity[r];
  forall( p in Products )
    ctDemand:
      Inside[p] + Outside[p] >= Product[p].demand;
}
```

Data for the revised production-planning problem (`product.dat`)

```
Products =  { "kluski", "capellini", "fettucine" };
Resources = { "flour", "eggs" };
Product = #[
      kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      fettucine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
          ]#;
Capacity = [ 20, 40 ];
```

The instruction

```
tuple productData {
   float demand;
   float insideCost;
   float outsideCost;
   float consumption[Resources];
}
```

declares a tuple type with four fields. The first three fields, of type `float`, are used to represent the demand and costs of a product; the last field is an array representing the resource consumptions of the product. These fields are intended to hold all the data related to a given product.

The instruction

```
ProductData product[Products] = ...;
```

declares an array of these tuples, one for each product. The initialization

```
Product = #[
      kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      fettucine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
          ]#;
```

from *Data for the revised production-planning problem (`product.dat`)* specifies these various data items: tuples are initialized by giving values for each of their fields. It is of course possible to use a named initialization for the tuple, as shown in *Named data for the revised production-planning problem (`productn.dat`)*, in which case the initialization is enclosed with `#<` and `>#`. Tuple fields can be obtained by suffixing the tuple with a dot and the field name. For instance, in the objective function

```
minimize
  sum( p in Products )
    (Product[p].insideCost * Inside[p] +
    Product[p].outsideCost * Outside[p] );
```

the expression `product[p].insideCost` represents the field `insideCost` of the tuple `product[p]`.

Similarly, in the constraint

```
forall(r in Resources)
   sum(p in Products) product[p].consumption[r] * inside[p] <= capacity[r];
```

the expression `product[p].consumption` represents the field consumption of tuple `product[p]`. This field is an array that can be subscripted in the traditional way.

Displaying results

The statements presented so far did not specify what elements of the solution should be displayed. OPL, and the Windows OPL IDE in particular, offers a way to display the results of an application. An interesting feature of OPL is the ability to display tuples of expressions.

To display results using an `execute` block:

1. Add the following IBM® ILOG Script `execute` block to the `product.mod` file (see *The production-planning problem revisited (`product.mod`)*)

```
tuple R { float x; float y; };
{R} Result = { <Inside[p],Outside[p]> | p in Products };
execute { writeln("Result=",Result); }
```

You see the following output:

```
Optimal solution found with objective: 372
result= {<40.0000 60.0000> <0.0000 200.0000> <0.0000 300.0000>}
```

> **Note**: If you are working in the Windows IDE, your user-defined solution displays in the Console tab, not in the Solutions tab.

2. Run the product model with the `productn.dat` data file shown in *Named data for the revised production-planning problem (`productn.dat`)*.

You can visualize the inside and outside productions of a product simultaneously.

```
Final Solution with objective 372.0000:
  inside = [40.0000 0.0000 0.0000];
  outside = [60.0000 200.0000 300.0000];
```

Named data for the revised production-planning problem (`productn.dat`)

```
Products =  { "kluski", "capellini", "fettucine" };
Resources = { "flour", "eggs" };

Product = #[
      kluski :
   #< demand:100
            insideCost:0.6
            outsideCost:0.8
            consumption:[0.5 0.2]
        >#,
      capellini :
   #< demand:200
            insideCost:0.8
            outsideCost:0.9
```

```
            consumption:[0.4 0.4]
          >#,
       fettucine :
   #< demand:300
            insideCost:0.3
            outsideCost:0.4
            consumption:[0.3 0.6]
          >#
            ]#;

Capacity = [ 20, 40 ];
```

3. Add the following IBM ILOG Script postprocessing lines to the `product.mod` file

```
execute {
   for(p in Products)
     writeln("inside[",p,"].reducedCost = ", inside[p].reducedCost);
}
```

You can see both the inside production of a product and its reduced cost.

```
Optimal solution found with objective: 372
 inside[kluski].reducedCost = 0
 inside[capellini].reducedCost = 0.06000000000000005
 inside[fettucine].reducedCost = 0.02000000000000002
```

Integer programming: the knapsack problem

Explains what integer programming is and describes the knapsack problem.

In this section

What is integer programming?
Defines linear programming.

A typical integer program: the knapsack problem
Presents the model and data files.

What is integer programming?

Integer programming expresses the optimization of a linear function subject to a set of linear constraints over integer variables.

The statements presented in *Linear programming: a production planning example* are all linear programming models. However, linear programs with very large numbers of variables and constraints can be solved efficiently. Unfortunately, this is no longer true when the variables are required to take integer values. *Integer programming* is the class of problems that can be expressed as the optimization of a linear function subject to a set of linear constraints over integer variables. It is in fact NP-hard. More important, perhaps, is the fact that the integer programs that can be solved to provable optimality in reasonable time are much smaller in size than their linear programming counterparts. There are exceptions, of course, and this documentation describes several important classes of integer programs that can be solved efficiently, but users of OPL should be warned that discrete problems are in general much harder to solve than linear programs.

A typical integer program: the knapsack problem

A typical example of integer programs is the knapsack problem, which can be intuitively understood as follows. We have a knapsack with a fixed capacity (an integer) and a number of items. Each item has an associated weight (an integer) and an associated value (another integer). The problem consists of filling the knapsack without exceeding its capacity, while maximizing the overall value of its contents. A multi-knapsack problem is similar to the knapsack problem, except that there are multiple features for the object (e.g., weight and volume) and multiple capacity constraints. *A multi-knapsack model (`knapsack.mod`)* depicts a model for the multi-knapsack problem, while *Data for the multi-knapsack problem (`knapsack.dat`)* describes an instance of the problem.

A multi-knapsack model (`knapsack.mod`)

```
int NbItems = ...;
int NbResources = ...;
range Items = 1..NbItems;
range Resources = 1..NbResources;
int Capacity[Resources] = ...;
int Value[Items] = ...;
int Use[Resources][Items] = ...;
int MaxValue = max(r in Resources) Capacity[r];


dvar int Take[Items] in 0..MaxValue;

maximize
  sum(i in Items) Value[i] * Take[i];

subject to {
  forall( r in Resources )
    ct:
      sum( i in Items )
        Use[r][i] * Take[i] <= Capacity[r];
}
```

This model has several novel features. It represents items and resources not by string sets but rather by integers. In other words, the items (respectively the resources) are represented by successive integers starting at 1. The instructions

```
int NbItems = ...;
int NbResources = ...;
range Items = 1..NbItems;
range Resources = 1..NbResources;
```

declare the number of items and the number of resources, as well as two ranges, `Items` and `Resources`, to represent the set of items and the set of resources.

The next three instructions

```
int Capacity[Resources] = ...;
int Value[Items] = ...;
int Use[Resources][Items] = ...;
```

are similar to the data declarations presented in *Data declarations* and the subsequent sections. The array `capacity` represents the capacity of the resources, the array `value` the value of each item, and `use[r][i]` the use of resource `r` by item `i`.

The next instruction

```
int MaxValue = max(r in Resources) Capacity[r];
```

is more interesting. It declares an integer `maxValue` whose value is given by an expression. OPL and IBM ILOG Script have many features for computing and preprocessing data, since this is fundamental in simplifying and improving the efficiency of many models.

The instruction

```
dvar int Take[Items] in 0..MaxValue;
```

declares the problem variables: `take[Items]` represents the number of times item `i` is selected in the solution. The variable is of type integer and is restricted to range in `0..maxValue`.

The rest of the statement is rather standard and should raise no difficulty. *Data for the multi-knapsack problem (`knapsack.dat`)* describes an instance of the problem.

Data for the multi-knapsack problem (`knapsack.dat`)

```
NbResources = 7;
NbItems = 12;
Capacity = [ 18209, 7692, 1333, 924, 26638, 61188, 13360 ];
Value = [ 96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81 ];
Use = [
      [ 19,   1,  10,  1,   1,  14, 152, 11,  1,   1, 1, 1 ],
      [  0,   4,  53,  0,   0,  80,   0,  4,  5,   0, 0, 0 ],
      [  4, 660,   3,  0,  30,   0,   3,  0,  4,  90, 0, 0],
      [  7,   0,  18,  6, 770, 330,   7,  0,  0,   6, 0, 0],
      [  0,  20,   0,  4,  52,   3,   0,  0,  0,   5, 4, 0],
      [  0,   0,  40, 70,   4,  63,   0,  0, 60,   0, 4, 0],
      [  0,  32,   0,  0,   0,   5,   0,  3,  0, 660, 0, 9]];
```

A solution to knapsack.mod

For the instance of the problem specified in *Data for the multi-knapsack problem (`knapsack.dat`)*, here are the final solution and the solutions that satisfy all the constraints but are not the best with respect to the objective function:

```
Feasible solution with objective 261890.0000:
  take = [0 0 0 154 0 0 0 912 333 0 6505 1180];

Feasible solution with objective 261922.0000:
  take = [0 0 0 153 0 0 0 912 333 0 6499 1180];

Final solution with objective 261922.0000:
  take = [0 0 0 154 0 0 0 913 333 0 6499 1180];
```

Although integer programs are, in general, substantially harder to solve than linear programs, they have also been the topic of intensive investigation. OPL recognizes when a statement is an integer programming model and uses CPLEX algorithms to solve it.

Mixed integer-linear programming: a blending problem

Presents OPL and MILP and describes a blending problem.

In this section

OPL and MILP
Discusses how OPL solves mixed integer-linear programs.

The blending problem
Describes the problem and presents the model.

Elements of the blending model
Presents the data file, decision variables and constraints.

OPL and MILP

Mixed integer-linear programs include both integer and real variables.

OPL can also solve models that include both integer and real variables, generally known as mixed integer-linear programs (MILP). OPL approaches them in essentially the same way as integer programs. A branch-and-bound algorithm can exploit the linear relaxation except, of course, that branching takes place only on integer variables.

The blending problem

The following blending problem is taken from W. Winston's book (see the Bibliography). Consider the following application involving mixing some metals into an alloy. The metal may come from several sources: in pure form or from raw materials, scraps from previous mixes, or ingots. The alloy must contain a certain amount of the various metals, as expressed by a production constraint specifying lower and upper bounds for the quantity of each metal in the alloy. Each source also has a cost and the problem consists of blending the sources into the alloy while minimizing the cost and satisfying the production constraints. Similar problems arise in other domains, e.g., the oil, paint, and the food processing industries. *A blending problem: part I (`blending.mod`)* and *A blending problem: part II (`blending.mod`)* show the two parts of the model for the blending problem.

A blending problem: part I (`blending.mod`)

```
int   NbMetals = ...;
int   NbRaw = ...;
int   NbScrap = ...;
int   NbIngo = ...;

range Metals = 1..NbMetals;
range Raws = 1..NbRaw;
range Scraps = 1..NbScrap;
range Ingos = 1..NbIngo;

float CostMetal[Metals] = ...;
float CostRaw[Raws] = ...;
float CostScrap[Scraps] = ...;
float CostIngo[Ingos] = ...;
float Low[Metals] = ...;
float Up[Metals] = ...;
float PercRaw[Metals][Raws] = ...;
float PercScrap[Metals][Scraps] = ...;
float PercIngo[Metals][Ingos] = ...;

int Alloy  = ...;
```

A blending problem: part II (`blending.mod`)

```
dvar float+    p[Metals];
dvar float+    r[Raws];
dvar float+    s[Scraps];
dvar int+      i[Ingos];
dvar float    m[j in Metals] in Low[j] * Alloy .. Up[j] * Alloy;


minimize
  sum(j in Metals) CostMetal[j] * p[j] +
  sum(j in Raws)   CostRaw[j]   * r[j] +
  sum(j in Scraps) CostScrap[j] * s[j] +
  sum(j in Ingos)  CostIngo[j]  * i[j];
subject to {
  forall( j in Metals )
    ct1:
      m[j] ==
```

```
      p[j] +
      sum( k in Raws )   PercRaw[j][k] * r[k] +
      sum( k in Scraps ) PercScrap[j][k] * s[k] +
      sum( k in Ingos )  PercIngo[j][k] * i[k];
    ct2:
      sum( j in Metals ) m[j] == Alloy;
}
```

Elements of the blending model

Problem data

The model is described in terms of a number of constants specifying the various types of metals, raw materials, scrap, and ingots. In the instance data shown in *Instance data for the blending problem (`blending.dat`)*, there are three metals, two raw materials, two kinds of scrap, and one kind of ingot. The model also defines ranges for each of the components. It then defines the cost of the various components in `costMetal`, `costRaw`, `costScrap`, `costIngo`. In the instance data, for example, the second raw material has a cost of 5. The data items `low` and `up` specify the production constraints and give lower and upper bounds on the quantity of each sort of metal in the alloy. For example, in the instance data, between 30% and 40% of the alloy must be the second metal. The next data items, `percRaw`, `percScrap`, and `percIngo`, specify the percentage of each metal in the sources. In *Instance data for the blending problem (`blending.dat`)*, the second type of scrap contains 1% of the first metal, none of the second metal, and 70% of the third metal. Finally, the data alloy specifies the amount of alloy to be produced.

Instance data for the blending problem (`blending.dat`)

```
NbMetals = 3;
NbRaw = 2;
NbScrap = 2;
NbIngo = 1;

CostMetal = [22, 10, 13];
CostRaw = [6, 5];
CostScrap = [ 7, 8];
CostIngo = [ 9 ];
Low = [0.05, 0.30, 0.60];
Up = [0.10, 0.40, 0.80];
PercRaw = [ [ 0.20, 0.01 ], [ 0.05, 0 ], [ 0.05, 0.30 ] ];
PercScrap = [ [ 0 , 0.01 ], [ 0.60, 0 ], [ 0.40, 0.70 ] ];
PercIngo = [ [ 0.10 ], [ 0.45 ], [ 0.45 ] ];
Alloy  = 71;
```

Decision variables

The decision variables specify how much of each source is used in the alloy: the array `p` specifies the quantities of pure metals, array `r` specifies the quantities of raw materials, array `s` specifies the quantities of scrap, array `i` specifies the number of ingots. All variables are of type `float` except number of ingots, which are integers. The problem is thus a mixed integer-linear program. The instruction

```
dvar float m[j in Metals] in low[j] * alloy .. up[j] * alloy;
```

is particularly interesting, since it shows how to specify the range of decision variables in a generic fashion. More precisely, the range of variables `m[j]` is given by the expression;

```
low[j] * alloy .. up[j] * alloy
```

Note also that the model uses the variables in array `m` as intermediary variables to represent the quantity of each metal produced.

Constraints

There are two types of constraints in this problem.

- The `forall` constraint

```
subject to {
  forall( j in Metals )
    ct1:
      m[j] ==
      p[j] +
      sum( k in Raws )   PercRaw[j][k] * r[k] +
      sum( k in Scraps ) PercScrap[j][k] * s[k] +
      sum( k in Ingos )  PercIngo[j][k] * i[k];
    ct2:
      sum( j in Metals ) m[j] == Alloy;
}
```

  makes sure that the right amounts of metal are produced. The amount `m[j]` of metal `j` must be equal to the amount of pure metal `p[j]` added to the quantity of metal `j` contained in the raw materials, the scrap, and the ingots. The correct amount of metals are computed using the percentage of metals contained in the sources.

- The `sum` constraint

```
sum(j in Metals) m[j] == alloy;
```

  makes sure that the various metals produced give the correct amount of alloy. The objective function in this model is rather simple. It consists of computing the price of each source from its unit price (e.g., `costMetal`) and the amount produced (e.g., `p[j]`).

A solution to blending.mod

For the instance data given in *Instance data for the blending problem (`blending.dat`)*, OPL returns the solution

```
Final Solution with objective 653.6100:
  p = [0.0467 0.0000 0.0000];
  r = [0.0000 0.0000];
  s = [17.4167 30.3333];
  i = [32];
  m = [3.5500 24.8500 42.6000];
```

Constraint programming: an inventory matching problem

Gives a short tour of constraint programming support in IBM® ILOG® OPL via an inventory problem and its model elements. Does not contain an overview of more basic modeling features such as arrays, data, aggregation, tuples, etc. For such information, see *Linear programming: a production planning example* in this manual. For tutorials that introduce scheduling problems in OPL, see *Getting Started with Scheduling in IBM ILOG OPL*.

In this section

The inventory problem
Describes the problem and gives the path to the files.

Modeling elements of the inventory problem
Examines some modeling aspects specific to constraint programming and discusses the CP Optimizer.

The search process
How to influence the default search of CP Optimizer.

The inventory problem

The problem is to build steel coils from slabs that are available in a work-in-progress inventory of semi-finished products. The assumption is that there is no limitation in the number of slabs that can be requested, but only a finite number of slab sizes is available (sizes 12, 14, 17, 18, 19, 20, 23, 24, 25, 26, 27, 28, 29, 30, 32, 35, 39, 42, 43, 44...). The problem is to select a number of slabs to build the coil orders, and to satisfy the following constraints:

- Each coil order requires a specific process to be built from a slab. This process is encoded by a color.
- A coil order can be built from only one slab.
- Several coil orders can be built from the same slab. But a slab can be used to produce at most two different 'colors' of coils.
- The sum of the sizes of each coil order built from a slab must not exceed the slab size.

Where to find the files

You will work with the steel mill example, supplied as the `steelmill` example at the following location:

```
<OPL_dir>\examples\opl\steelmill
```

where `<OPL_dir>` is your installation directory.

Data for the model `steelmill.mod` is contained in the file `steelmill.dat`.

Modeling elements of the inventory problem

This section:

- examines some modeling aspects that are specific to constraint programming:
  - specialized constraints
  - aggregate `or` /`and`
- focuses on the search for solutions and how the CP Optimizer engine can be tuned to find better and faster solutions:
  - changing constraint programming parameters
  - using search phases to describe a more specific search
- explains how you can run this model from the OPL IDE and how some specific IDE features work in the constraint programming context.

The CP Optimizer engine can currently solve any model than can be solved by the CPLEX engine, provided that all the decision variables are discrete. This means that any modeling object (constraint or expressions) in OPL can be interpreted by this engine. The reverse is not true: some of the new global expressions and constraints can be used only in a CP model to be solved with the CP Optimizer engine.

The solving engine

By default (that is, if nothing different is specified), OPL uses the CPLEX engine to solve an OPL model. To specify that you want your model to be solved by the CP Optimizer engine, you must start the model with this statement:

```
using CP;
```

The OPL model

You do not need to learn any new syntax to develop a CP model. The organization of an OPL model for CP does not change. You define and manipulate data and decision variables in the same way. For example, to define an array of integer decision variables indexed by integers from `1` to `nbOrders` and taking values between `1` and `nbSlabs`, you can write:

```
dvar int where[1..nbOrders] in 1..nbSlabs;
```

Modeling constraints and specialized constraints

As for any OPL model, constraints are stated in a `constraints {}` or `subject to {}` block. When the model is solved by the CP Optimizer engine, you can include some specialized constraints in this set of modeling constraints. For example, in the steel mill example, there is a packing constraint.

```
subject to {
  packCt: pack(load, where, weight);
```

This `pack` constraint is a simple but powerful one-dimensional packing constraint. It constrains the way coils are associated with slabs with respect to the weights of the coils and the capacity of the slabs. More precisely, the decision variable `where[i]` states with which slab coil `i` is to be associated. The decision variable `load[j]` represents the total weight of all the coils associated with slab `j`, using the values from the array weights as data. In this case, the loads are constrained by the maximum value of `maxLoad` which is used as upper bound to create the load variables.

The "all" syntax

In the steel mill example, the arrays of values and decision variables used in the specialized constraint are modeling arrays. They make sense as a whole and have been named in the model. Sometimes, you will want to apply a specialized constraint to a set of variables that is not defined as a named array of variables, but that is made of dynamically collected variables. The `all` keyword is the syntax that enables you to collect variables dynamically in an array. This syntax is important for CP models. It is not illustrated in the steel mill example. See all in the *Language Quick Reference* for a complete description.

Aggregate and/or

The other constraint of the model illustrates how to use the aggregate `or` constraint.

```
  forall(s in 1..nbSlabs)
    colorCt: sum (c in 1..nbColors) (or(o in 1..nbOrders : colors[o] == c)
(where[o] == s)) <= 2;
```

You use this constraint just as the usual `forall` constraint or `sum` expression. The `or` constraint can express a complex combination of constraints in one single statement. As a result of the expressiveness of the OPL language, the steel mill model uses only two OPL constraints to represent a very complete and realistic model.

The search process

Constraint programming techniques have been seen for a long time as harder to use than mathematical programming because it was sometimes necessary to define search procedures using complex syntaxes and concepts. Now the IBM® ILOG CP Optimizer engine includes a powerful default search. This means that without giving any particular indication on how the solution or optimal solution is to be found, some combinations of techniques are used with default behaviors so that good solutions are quickly found for a wide variety of problems.

However, you can influence the search process by:

- Changing CP parameters
- Defining search phases

Changing CP parameters

You can change CP Optimizer parameters by means of script statements. See *Changing CP parameters*.

Setting limits

Limits are just a particular type of parameters you can change with a certain impact on the search process. You can set a limit on the number of failures during the search or on the time spent searching. Here are examples on how to change these parameters :

```
execute {
   cp.param.FailLimit = 500;
   cp.param.TimeLimit = 20;
}
```

Changing the search behavior

You can also change the search behavior by using CP parameters to change the nature of the search. The CP Optimizer engine includes several different methods to search for solutions. Some parameters enable you to decide which search method to use exactly. This is useful in some cases. The default search uses default combinations that are proven to be the best choice on average. However, better combinations can be found on particular cases. Here are examples on how to change such parameters:

```
execute {
   cp.param.searchType = "multiPoint";
   writeln("Search type is " + cp.param.searchType);
   cp.param.DefaultInferenceLevel = "Extended";
}
```

Defining search phases

Sometimes, the default search may not be capable of finding good enough values in an appropriate amount of time. By changing parameters, you can modify slightly how these

default algorithms work. When this is not sufficient, you can also help the engine by providing some indications on the structure of the search space. The default search algorithm can use them efficiently to find good solutions. This is referred to as “search phases”.

For example, in the steel mill code sample, the preprocessing part indicates that the engine must start instantiating the `where` variables.

```
execute {
   var f = cp.factory;
   cp.setSearchPhases(f.searchPhase(where));
}
```

In each phase, you can use search modifiers to apply specific settings that will be local to the search phase. To this effect, you specify which ones are used on which variables at each phase.

The syntax to apply search modifiers in a phase is:

```
var phase1 = searchPhase(variable_array_from_model,
                                variable_selector,
                                value_selector);
```

You can have as many phases as you want. You tell the CP Optimizer engine the phases to use with the syntax:

```
cp.setSearchPhases(phase1, phase2);
```

See *Using IBM ILOG Script in constraint programming* and *Understanding solving statistics and progress (CP models)*.

Modeling tips

Describes a few recommended practices to help you write more efficient models.

In this section

Efficient models
In the sense of “running as efficiently as possible”.

Sparsity
Discusses sparsity, tuples of parameters, and filtering in the context of model efficiency.

About arrays
Discusses sparsity, tuples of parameters, and filtering in the context of model efficiency.

Other modeling tips
Collects miscellaneous modeling tips that are already mentioned elsewhere in the documentation set.

Efficient models

An application can often be described by various models that may exhibit fundamentally different performances in terms of memory and computing time. This is particularly important for large-scale models.

In this context, “running as efficiently as possible” applies to the part of the execution that is related to modeling, NOT to model design itself. In other words, this section does not explain how to write a better model that finds an optimal solution faster. Refer also to *Performance and memory usage* in *The Language User's Manual*.

Sparsity

Discusses sparsity, tuples of parameters, and filtering in the context of model efficiency.

In this section

Sparsity in the transportation problem
Describes the problem, and presents `transp1.mod`.

Exploiting sparsity - a first attempt
Presents `transp2.mod`.

Exploiting sparsity - a better model
Presents `transp3.mod`.

Sparsity in the transportation problem

Consider the transportation problem in which the shipments of products between each pair of cities may not exceed a given limit. *A simple transportation model (`transp1.mod`)* shows a simple model for this problem, which implicitly assumes that all cities are connected and that all products may be shipped between two cities. It is thus not appropriate for large-scale problems where only a fraction of the cities are connected.

A small data set can easily illustrate the issue. Consider the set of cities

```
{"Amsterdam","Antwerpen","Bergen","Bonn","Brussels","Cassis","London","Madrid",
"Milan","Paris"}
```

and the set of products `{"Godiva","Leonidas","Neuhaus"}`. There are three hundred ways of shipping a product from one city to another. However, only a small fraction of these may be explored in the application and *A sparse data set for a transportation problem* displays a possible subset.

Using the statement in *A simple transportation model (`transp1.mod`)* would induce a substantial loss in (memory and time) efficiency. The following sections explore how to exploit this sparsity.

A simple transportation model (`transp1.mod`)

```
{string} Cities =...;
{string} Products = ...;
float Capacity = ...;

float Supply[Products][Cities] = ...;
float Demand[Products][Cities] = ...;
assert
  forall(p in Products)
    sum(o in Cities) Supply[p][o] == sum(d in Cities) Demand[p][d];
float Cost[Products][Cities][Cities] = ...;

dvar float+ Trans[Products][Cities][Cities];


minimize
  sum( p in Products , o , d in Cities )
    Cost[p][o][d] * Trans[p][o][d];

subject to {
  forall( p in Products , o in Cities )
    ctSupply:
      sum( d in Cities )
        Trans[p][o][d] == Supply[p][o];
  forall( p in Products , d in Cities )
    ctDemand:
      sum( o in Cities )
        Trans[p][o][d] == Demand[p][d];
   forall( o , d in Cities )
     ctCapacity:
       sum( p in Products )
```

```
         Trans[p][o][d] <= Capacity;
}

execute DISPLAY {
  writeln("trans = ",Trans);
}
```

A sparse data set for a transportation problem

<"Godiva","Brussels","Paris">	<"Godiva","Brussels","Bonn">	<"Godiva","Amsterdam","Lon
<"Godiva","Amsterdam","Milan">	<"Godiva","Antwerpen","Madrid">	<"Godiva","Antwerpen","Berg
<"Neuhaus","Brussels","Milan">	<"Neuhaus","Brussels","Bergen">	<"Neuhaus","Amsterdam","M
<"Neuhaus","Amsterdam","Cassis">	<"Neuhaus","Antwerpen","Paris">	<"Neuhaus","Antwerpen","Bo
<"Leonidas","Brussels","Bonn">	<"Leonidas","Brussels","Milan">	<"Leonidas","Amsterdam","P
<"Leonidas","Amsterdam","Cassis">	<"Leonidas","Antwerpen","London">	<"Leonidas","Antwerpen","Be

The rest of this section explores how to exploit this sparsity.

Exploiting sparsity - a first attempt

A first attempt at exploiting the sparsity available in a large-scale transportation problem consists of representing the data as a set `routes` of tuples of type

```
tuple Route { string p; string o; string d; }
```

The array `cost` and `trans` can then be indexed with this set. A model based on this idea appears in *A sparse transportation model: first attempt (`transp2.mod`)*.

A sparse transportation model: first attempt (`transp2.mod`)

```
{string} Cities = ...;
{string} Products = ...;
float Capacity = ...;

tuple route {
  string p;
  string o;
  string d;
}
{route} Routes = ...;
tuple supply {
  string p;
  string o;
}
{supply} Supplies = { <p,o> | <p,o,d> in Routes };
float Supply[Supplies] = ...;
tuple customer {
  string p;
  string d;
}
{customer} Customers = { <p,d> | <p,o,d> in Routes };
float Demand[Customers] = ...;
float Cost[Routes] = ...;

{string} Orig[p in Products] = { o | <p,o,d> in Routes };
{string} Dest[p in Products] = { d | <p,o,d> in Routes };

assert forall(p in Products)
  sum(o in Orig[p])
    Supply[<p,o>] == sum( d in Dest[p] ) Demand[<p,d>];

dvar float+ Trans[Routes];
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>];
  ctCapacity:  forall( o , d in Cities )
                 sum( <p,o,d> in Routes )
                   Trans[<p,o,d>] <= Capacity;
}
```

The data for the supplies and demands are also represented in a sparse way by projecting the set `Routes` to obtain their index sets. In addition to that, the model also precomputes, in a generic way, the cities `orig[p]` that can ship product `p` and the cities `dest[p]` that can receive product `p`. Most of the resulting model is elegant and efficient.

Unfortunately, the constraint

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

is not particularly efficient because it does not exploit the structure of the application. Indeed, the `forall` statement iterates not over actual connections but rather over all pairs of cities. In addition, the aggregate operator on the second line

```
sum(<p,o,d> in Routes) trans[<p,o,d>] <= capacity;
```

cannot exploit the “connection” structure to obtain all products of a connection, since `o` and `d` are separate entities.

Exploiting sparsity - a better model

The application can be modeled more effectively by closely reflecting the structure of the application. *A sparse transportation model: second attempt (`transp3.mod`)* gives a statement illustrating this principle. The main novelty is the explicit representation of connections and the fact that a route is now simply the association of a connection and a product. Connections are also computed automatically from routes. The rest of the model is generally similar but reflects the new data organization. The most interesting change is the capacity constraint, which becomes

```
forall(c in connections)
   sum(<c,p> in Routes) trans[<c,p>] <= capacity;
```

This constraint is much more efficient than in the previous model presented in *Exploiting sparsity - a first attempt*. First, it iterates over the routes, not over all pairs of cities. Second, the aggregate operator `sum` uses parameter `c` to index the set `Routes`, retrieving the relevant products effectively.

A sparse transportation model: second attempt (`transp3.mod`)

```
{string} Cities =...;
{string} Products = ...;
float Capacity = ...;
tuple connection { string o; string d; }
tuple route {
  string p;
  connection e;
}
{route} Routes = ...;
{connection} Connections = { c | <p,c> in Routes };
tuple supply {
  string p;
  string o;
}
{supply} Supplies = { <p,c.o> | <p,c> in Routes };
float Supply[Supplies] = ...;
tuple customer {
  string p;
  string d;
}
{customer} Customers = { <p,c.d> | <p,c> in Routes };
float Demand[Customers] = ...;
float Cost[Routes] = ...;
{string} Orig[p in Products] = { c.o | <p,c> in Routes };
{string} Dest[p in Products] = { c.d | <p,c> in Routes };

{connection} CPs[p in Products] = { c | <p,c> in Routes };
assert forall(p in Products)
   sum(o in Orig[p]) Supply[<p,o>] == sum(d in Dest[p]) Demand[<p,d>];

dvar float+ Trans[Routes];

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( <o,d> in CPs[p] )
        Trans[< p,<o,d> >] == Supply[<p,o>];
  forall( p in Products , d in Dest[p] )
    ctDemand[p][d]:
      sum( <o,d> in CPs[p] )
        Trans[< p,<o,d> >] == Demand[<p,d>];
  forall(c in Connections)
    ctCapacity:
      sum( <p,c> in Routes )
        Trans[<p,c>] <= Capacity;
}
```

About arrays

Order of indexers

When you use multidimensional arrays, the order of the dimensions may be significant. For instance, in the following example:

```
/*..*/

range r1 = 1..n1;
range r2 = 1..n2;

dvar int+ x[r1][r2];

/*..*/

a1 == sum(i in r1, j in r2) x[i][j];
a2 == sum(j in r2, i in r1) x[i][j];
```

the calculation of `a1` is more efficient because OPL internal caching mechanism recalculates `x[i]` only when `i` changes.

Array initialization

A better-performing array initialization syntax has been introduced in OPL4.1. For example, the `profiler.mod` example contains two semantically equivalent initializations:

Two ways of initializing arrays

```
int Values1[r][r];

execute INIT_Values1 {
  for( var i in r )
    for( var j in r )
      if ( i == 2*j )
        Values1[i][j] = i+j;
  writeln(Values1);
}

int Values2[i in r][j in r] = (i==2*j) ? i+j : 0;

execute INIT_Values2 {
  writeln(Values2);
}
```

Initialization of `Values2` is much faster than initialization of `Values1`, as shown by the profiling facility described in *Profiling the execution of a model* in *IDE Tutorials*.

Generic arrays

It is recommended to use generic arrays, or generic indexed arrays, whenever possible because they make the model more explicit and readable.

Other modeling tips

Labeling constraints

It is recommended to give labels to constraints but be aware of the performance cost. See section *Cost* of *Labeling constraints* in the *Language Reference Manual*.

Tuples of parameters

The general expression

```
p in S
```

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

```
<p1,...,pn> in S
```

for more readability. For more information on tuples of parameters, see *Formal parameters* in the *Language Reference Manual*.

The application areas

Describes applications of linear and integer programming, constraint programming, quadratic programming, and CPLEX® logical constraints.

In this section

Some examples
Demonstrates how OPL is used in linear programming (product mix problem) and integer programming (warehouse location problem).

Applications of linear and integer programming
Studies the application of OPL to linear programming, integer programming, mixed integer-linear programming, and piecewise linear programming.

Applications of constraint programming
Defines constraint programming and describes a column generation problem (vellino example), a production problem (car sequencing example), a time tabling problem (time tabling example), and an introductory scheduling problem.

Quadratic programming
Defines quadratic programming (QP), including quadratically-constrained programming (QCP), mixed integer quadratic programming (MIQP), and mixed-integer quadratically-constrained programming (MIQCP).

Tutorial: Using CPLEX logical constraints
Demonstrates how to use logical constraints in an application.

Some examples

Demonstrates how OPL is used in linear programming (product mix problem) and integer programming (warehouse location problem).

In this section

Linear programming: a product mix problem
Describes the problem and presents the model and data files.

Integer programming: a warehouse location problem
Describes the problem and presents the model and data files.

Linear programming: a product mix problem

As a first example, let's consider a simple mathematical programming (MP) problem to determine an optimal production mix.

To meet the demands of its customers, a company manufactures its products in its own factories (*inside* production) or buys them from other companies (*outside* production). Inside production is subject to some resource constraints: each product consumes a certain amount of each resource. In contrast, outside production is theoretically unlimited. The problem is to determine how much of each product should be produced inside the company and how much outside, while minimizing the overall production cost, meeting the demand, and not exceeding the resource constraints.

The statement of the problem must specify the set of products and the set of resources. For each product, we need to know the inside and outside production costs, and for each resource we need to know the available capacity of that resource. Finally, we need to know the consumption of resources by the different products.

This is a general outline of an optimization problem. The `production` example illustrates a specific pasta manufacturing problem. The project contains a model, `product.mod`, shown in *A pasta manufacturing problem (`product.mod`)* , which states the problem to be solved, and the data to be used by the model, `product.dat`, shown in *Data for the pasta manufacturing problem (`product.dat`)* .

A pasta manufacturing problem (`product.mod`)

```
{string} Products = ...;
{string} Resources = ...;
tuple productData {
   float demand;
   float insideCost;
   float outsideCost;
   float consumption[Resources];
}
productData Product[Products] = ...;
float Capacity[Resources] = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

execute CPX_PARAM {
  cplex.preind = 0;
  cplex.simdisplay = 2;
}


minimize
  sum( p in Products )
    (Product[p].insideCost * Inside[p] +
    Product[p].outsideCost * Outside[p] );
subject to {
  forall( r in Resources )
    ctInside:
      sum( p in Products )
        Product[p].consumption[r] * Inside[p] <= Capacity[r];
```

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

Data for the pasta manufacturing problem (`product.dat`)

```
Products =  { "kluski", "capellini", "fettucine" };
Resources = { "flour", "eggs" };
Product = #[
      kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      fettucine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
          ]#;
Capacity = [ 20, 40 ];
```

In the model, the instruction

```
tuple productData {
   float demand;
   float insideCost;
   float outsideCost;
   float consumption[Resources];
}
```

declares a tuple type with four fields. The first three fields, of type `float`, are used to represent the demand and costs of a product; the last field is an array representing the resource consumptions of the product. These fields are intended to hold all the data related to a given product.

The instruction

```
ProductData product[Products] = ...;
```

declares an array of these tuples, one for each product.

The model also contains two arrays of decision variables to represent the inside and outside production, respectively. There is an objective function to minimize the total production cost, and there are two types of constraints: a set of constraints to avoid exceeding the capacity limitation, and another set of constraints to satisfy the demand requirements.

Initialization of the data given in `product.dat` for one instance of this problem specifies these various data items: to initialize the tuples, values are given for each of their fields.

```
Product = #[
      kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      fettucine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
          ]#;
```

In the data file we use a set of strings called `Products` to represent the varieties of pasta and another set of strings called `Resources` to represent the raw ingredients of flour and eggs.

Integer programming: a warehouse location problem

Warehouse location is a typical discrete optimization problem. A company is considering a number of locations for building warehouses to supply a set of stores. Each possible warehouse has a fixed operating cost and a maximum capacity specifying how many stores it can support. In addition, each store must be supplied by exactly one warehouse and the cost to supply a store depends on the warehouse selected. The model consists of choosing which warehouses to build and which warehouse to assign to each store in order to minimize the total cost, i.e., the sum of the fixed and supply costs. Consider an example with five warehouses and ten stores. The fixed costs for the warehouses are all identical and equal to 30. The instance data for the problem, shown in the table below, reflects the transportation costs and the capacity constraints defined in the data file `warehouse.dat`.

Instance data for the warehouse location problem

Warehouses	Bonn	Bordeaux	London	Paris	Rome
Capacity	1	4	2	1	3
store1	20	24	11	25	30
store2	28	27	82	83	74
store3	74	97	71	96	70
store4	2	55	73	69	61
store5	46	96	59	83	4
store6	42	22	29	67	59
store7	1	5	73	59	56
store8	10	73	13	43	96
store9	93	35	63	85	46
store10	47	65	55	71	95

To represent our warehouse location problem as an integer program, the model, `warehouse.mod`, uses a `0-1` Boolean variable for each combination of warehouse and store to represent whether or not a warehouse supplies a store. In addition, the model associates a variable with each warehouse to indicate whether the warehouse is selected. Once these variables are declared, the constraints state that each store must be supplied by a warehouse, that each store can be supplied by only an open warehouse, and that each warehouse cannot deliver more stores than its allowed capacity.

Warehouse location, as a discrete optimization problem (`warehouse.mod`)

```
int Fixed = ...;
{string} Warehouses = ...;
int NbStores = ...;
range Stores = 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores][Warehouses] = ...;
dvar boolean Open[Warehouses];
dvar boolean Supply[Stores][Warehouses];
```

```
minimize
  sum( w in Warehouses )
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];


subject to{
  forall( s in Stores )
    ctEachStoreHasOneWarehouse:
      sum( w in  Warehouses )
        Supply[s][w] == 1;
  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
      sum( s in Stores )
        Supply[s][w] <= Capacity[w];
}

{int} Storesof[w in Warehouses] = { s | s in Stores : Supply[s][w] == 1 };
execute DISPLAY_RESULTS{
  writeln("Open=",Open);
  writeln("Storesof=",Storesof);
}
```

The most delicate aspect of the modeling is expressing that a warehouse can supply a store only when it is open. These constraints can be expressed by inequalities of the form

```
supply[w][s] <= open[w];
```

which ensure that when warehouse `w` is not open, it cannot supply store `s`. This follows from the fact that `open[w] == 0` implies `supply[w][s] == 0`. In fact, these constraints can be combined with the capacity constraints to obtain

```
  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
      sum( s in Stores )
        Supply[s][w] <= Capacity[w];
```

This formulation implies that a closed warehouse has no capacity.

The statement declares the warehouses and the stores, the fixed cost of the warehouses, and the supply cost of a store for each warehouse. The problem variables

```
dvar boolean Supply[Stores][Warehouses];
```

represent which warehouses supply the stores, i.e., `supply[s][w]` is 1 if warehouse `w` supplies store `s`, and zero otherwise.

The objective function

```
minimize
  sum( w in Warehouses )
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];
```

expresses the goal that the model minimizes the fixed cost of the selected warehouse and the supply costs of stores.

The constraint

```
  forall( s in Stores )
    ctEachStoreHasOneWarehouse:
      sum( w in  Warehouses )
        Supply[s][w] == 1;
```

states that a store must be supplied by exactly one warehouse.

The constraints

```
  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
      sum( s in Stores )
        Supply[s][w] <= Capacity[w];
```

express the capacity constraints for the warehouses and make sure that a warehouse supplies a store only if the warehouse is open.

A solution to warehouse.mod

For the instance data depicted in the table *Instance data for the warehouse location problem*, OPL returns the following optimal solution:

```
Final Solution with objective 383.0000:
  open = [1 1 1 0 1];
  supply = [[0 0 0 0 1]
            [0 1 0 0 0]
            [0 0 0 0 1]
            [1 0 0 0 0]
            [0 0 0 0 1]
            [0 1 0 0 0]
            [0 1 0 0 0]
            [0 0 1 0 0]
            [0 1 0 0 0]
            [0 0 1 0 0]];
```

Applications of linear and integer programming

Studies the application of OPL to linear programming, integer programming, mixed integer-linear programming, and piecewise linear programming.

In this section

Linear programming
Defines linear programming and describes a simple production planning problem, a multiperiod production planning problem, a blending problem, and sensitivity analysis.

Integer programming
Defines integer programming and describes a set covering problem, a warehouse location problem, a fixed-charge problem, and integer relaxation.

Mixed integer-linear programming
Defines mixed integer-linear programming and describes an upgrade to the production-planning problem to include a fixed charge for the products.

Piecewise linear programming
Defines piecewise linear programming, describes an inventory problem with piecewise linear functions, compares pwl to plain linear programming, and indicates complexity issues.

Linear programming

Defines linear programming and describes a simple production planning problem, a multiperiod production planning problem, a blending problem, and sensitivity analysis.

In this section

What is linear programming
Defines linear programming.

A production problem
Uses again the model `production.mod`.

A multi-period production planning problem
Extends the production planning problem to several production periods.

A blending problem
Presents the problem of calculating different blends of gasoline according to specific quality criteria.

Exploiting sparsity
Discusses how to exploit the sparsity of large-scale problems, beyond the classical transportation problem exposed in the `transp1.mod` sample.

Sensitivity analysis
Explains how to obtain sensitivity information on variables and constraints.

What is linear programming

Linear programming (LP) consists in optimizing a linear function subject to linear constraints over real variables.

In LP, the model of a problem is expressed through numeric variables combined in linear constraints and governed by a linear objective function and by bounds on the variables. OPL can efficiently solve large instances of linear programs.

A production problem

Consider again the production planning problem first presented in the section *Tuples*. The model is depicted again in *A Production planning problem (`production.mod`)* below and the instance data in *Instance data for the production-planning problem (`production.dat`)* .

A Production planning problem (`production.mod`)

```
{string} Products = ...;
{string} Resources = ...;

float Consumption[Products][Resources] = ...;
float Capacity[Resources] = ...;
float Demand[Products] = ...;
float InsideCost[Products] = ...;
float OutsideCost[Products]  = ...;

dvar float+ Inside[Products];
dvar float+ Outside[Products];

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];
}
```

Instance data for the production-planning problem (`production.dat`)

```
Products =  { "kluski", "capellini", "fettucine" };
Resources = { "flour", "eggs" };

Consumption = [ [0.5, 0.2], [0.4, 0.4], [0.3, 0.6] ];
Capacity = [ 20, 40 ];
Demand = [ 100, 200, 300 ];
InsideCost = [ 0.6, 0.8, 0.3 ];
OutsideCost  = [ 0.8, 0.9, 0.4 ];
```

The model aims at minimizing the production cost for a number of products while satisfying customer demand. Each product can be produced either inside the company or outside, at a higher cost. The inside production is constrained by the company's resources, while outside production is considered unlimited. The model first declares the products and the resources. The data consists of the description of the products, i.e., the demand, the inside and outside costs, and the resource consumption, and the capacity of the various resources. The variables for this problem are the inside and outside production for each product.

A solution to production.mod

For these statements, OPL returns the optimal solution

```
Final Solution with objective 372.0000:
  inside = [40.0000 0.0000 0.0000];
  outside = [60.0000 200.0000 300.0000];
```

A multi-period production planning problem

Large linear-programming problems are often obtained from simpler ones by generalizing them along one or more dimensions. A typical extension of production-planning problems is to consider several production periods and to include inventories in the model. This section presents a multiperiod production planning model that generalizes the model of the previous section *A production problem*.

The main generalization is to consider the demand for the products over several periods and to allow the company to produce more than the demand in a given period. Of course, there is an inventory cost associated with storing the additional production. *A multi-period production-planning problem (`mulprod.mod`)* depicts the new model and *Instance data for multi-period production-planning problem (`mulprod.dat`)* describes the instance data.

A multi-period production-planning problem (`mulprod.mod`)

```
{string} Products = ...;
{string} Resources = ...;
int NbPeriods = ...;
range Periods = 1..NbPeriods;

float Consumption[Resources][Products] = ...;
float Capacity[Resources] = ...;
float Demand[Products][Periods] = ...;
float InsideCost[Products] = ...;
float OutsideCost[Products]  = ...;
float Inventory[Products]  = ...;
float InvCost[Products]  = ...;

dvar float+ Inside[Products][Periods];
dvar float+ Outside[Products][Periods];
dvar float+ Inv[Products][0..NbPeriods];


minimize
  sum( p in Products, t in Periods )
      (InsideCost[p]*Inside[p][t] +
       OutsideCost[p]*Outside[p][t] +
       InvCost[p]*Inv[p][t]);

subject to {
  forall( r in Resources, t in Periods )
    ctCapacity:
      sum( p in Products )
        Consumption[r][p] * Inside[p][t] <= Capacity[r];
  forall( p in Products , t in Periods )
    ctDemand:
      Inv[p][t-1] + Inside[p][t] + Outside[p][t] == Demand[p][t] + Inv[p][t];

  forall( p in Products )
    ctInventory:
      Inv[p][0] == Inventory[p];
};
tuple plan {
```

```
   float inside;
   float outside;
   float inv;
}
plan Plan[p in Products][t in Periods] = <Inside[p,t],Outside[p,t],Inv[p,t]>;
execute DISPLAY {
  writeln("plan=",Plan);
}
```

Instance data for multi-period production-planning problem (`mulprod.dat`)

```
Products =  { kluski capellini fettucine };
Resources = { flour eggs };
NbPeriods = 3;

Consumption = [
                [ 0.5, 0.4, 0.3 ],
                [ 0.2, 0.4, 0.6 ]
              ];
Capacity = [ 20, 40 ];
Demand = [
           [ 10 100 50 ]
           [ 20 200 100]
           [ 50 100 100]
         ];
Inventory = [ 0 0 0];
InvCost = [ 0.1 0.2 0.1];
InsideCost = [ 0.4, 0.6, 0.1 ];
OutsideCost  = [ 0.8, 0.9, 0.4 ];
```

Most of the model generalizes smoothly. For instance, the capacity constraints stated for all resources and all periods become

```
  forall( r in Resources, t in Periods )
    ctCapacity:
      sum( p in Products )
        Consumption[r][p] * Inside[p][t] <= Capacity[r];
```

The most novel part of the statement is the constraint linking the demand, the inventory, and the production:

```
  forall( p in Products , t in Periods )
    ctDemand:
      Inv[p][t-1] + Inside[p][t] + Outside[p][t] == Demand[p][t] + Inv[p][t];
```

The constraint states that, for each product `p` and each period `t`, the inventory of period `t -1` added to the production of period `t` is equated to the demand of period `t` plus the inventory of period `t`. Of course, the fact that the variables `inv[p][t]` are constrained to be nonnegative is critical to satisfying the demand and to disallow back orders. The objective function is also generalized to add the inventory costs.

Note also the type declaration

```
tuple plan {
   float inside;
```

```
   float outside;
   float inv;
}
```

and the display instructions

```
plan Plan[p in Products][t in Periods] = <Inside[p,t],Outside[p,t],Inv[p,t]>;
execute DISPLAY {
  writeln("plan=",Plan);
}
```

which were added to produce a visually pleasing display.

A solution to mulprod.mod

For example, on the instance data depicted in *Piecewise linear functions leading to linear programs*, OPL produces the optimal solution

```
Optimal solution found with objective: 457
plan=
[[<10.0000 0.0000 0.0000> <0.0000 100.0000 0.0000> <0.0000 50.0000 0.0000>]
[<0.0000 20.0000 0.0000> <0.0000 200.0000 0.0000> <0.0000 100.0000 0.0000>]
[<50.0000 0.0000 0.0000> <66.6667 33.3333 0.0000> <66.6667 33.3333 0.0000>]]
```

A blending problem

Blending problems are another typical application of linear programming. Consider the following problem. An oil company manufactures three types of gasoline: `super`, `regular`, and `diesel`. Each type of gasoline is produced by blending three types of crude oil: `crude1`, `crude2`, and `crude3`. *Prices for the blending problem* depicts the sales price and the purchase price per barrel of the various products. The gasoline must satisfy some quality criteria with respect to their lead content and their octane ratings, thus constraining the possible blendings.

Prices for the blending problem

	Sales Price		Purchase Price
super	$70	crude1	$45
regular	$60	crude2	$35
diesel	$50	crude3	$25

Octane and lead data for the blending problem describes the relevant instance data.

Octane and lead data for the blending problem

	Octane Rating	Lead Content		Octane Rating	Lead Contents
super	greater than or equal to 10	less than or equal to 1	crude1	12	0.5
regular	greater than or equal to 8	less than or equal to 2	crude2	6	2.0
diesel	greater than or equal to 6	less than or equal to 1	crude3	8	3.0

The company must also satisfy its customer demand, which is 3,000 barrels a day of `super`, 2,000 of `regular`, and 1,000 of `diesel`. The company can purchase 5,000 barrels of each type of crude oil per day and can process at most 14,000 barrels a day. In addition, the company has the option of advertising a gasoline, in which case the demand for this type of gasoline increases by ten barrels for every dollar spent. Finally, it costs four dollars to transform a barrel of oil into a barrel of gasoline. The model is depicted in *An oil-blending planning problem (`oil.mod`)* and the instance data is shown in *Data for the oil-blending planning problem (`oil.dat`)* .

An oil-blending planning problem (`oil.mod`)

```
{string} Gasolines = ...;
{string} Oils = ...;
tuple gasType {
  float demand;
  float price;
  float octane;
  float lead;
}

tuple oilType {
  float capacity;
  float price;
```

```
  float octane;
  float lead;
}
gasType Gas[Gasolines] = ...;
oilType Oil[Oils] = ...;
float MaxProduction = ...;
float ProdCost = ...;

dvar float+ a[Gasolines];
dvar float+ Blend[Oils][Gasolines];


maximize
  sum( g in Gasolines , o in Oils )
    (Gas[g].price - Oil[o].price - ProdCost) * Blend[o][g]
    - sum(g in Gasolines) a[g];
subject to {
  forall( g in Gasolines )
    ctDemand:
      sum( o in Oils )
        Blend[o][g] == Gas[g].demand + 10*a[g];
  forall( o in Oils )
    ctCapacity:
      sum( g in Gasolines )
        Blend[o][g] <= Oil[o].capacity;
  ctMaxProd:
    sum( o in Oils , g in Gasolines )
      Blend[o][g] <= MaxProduction;
  forall( g in Gasolines )
    ctOctane:
      sum( o in Oils )
        (Oil[o].octane - Gas[g].octane) * Blend[o][g] >= 0;
  forall( g in Gasolines )
    ctLead:
      sum( o in Oils )
        (Oil[o].lead - Gas[g].lead) * Blend[o][g] <= 0;
}

execute DISPLAY_REDUCED_COSTS{
  for( var g in Gasolines ) {
    writeln("a[",g,"].reducedCost = ",a[g].reducedCost);
  }
}
```

Data for the oil-blending planning problem (`oil.dat`)

```
Gasolines = { "Super", "Regular", "Diesel" };
Oils = { "Crude1", "Crude2", "Crude3" };

Gas = [ <3000, 70, 10, 1>,
        <2000, 60, 8, 2>,
        <1000, 50, 6, 1> ];

Oil = [ <5000, 45, 12, 0.5>,
        <5000, 35, 6,  2>,
```

```
        <5000, 25, 8,  3> ];


MaxProduction = 14000;
ProdCost = 4;
```

The model uses two sets of variables. Variable `a[Gasolines]` represents the amount spent in advertising gasoline `g`. Variable `blend[Oils][Gasolines]` represents the number of barrels of crude oil `o` used to produced gasoline `g`. The demand constraints

```
  forall( g in Gasolines )
    ctDemand:
      sum( o in Oils )
        Blend[o][g] == Gas[g].demand + 10*a[g];
```

use both types of variables, since `sum(o in Oils) blend[o][g]` represents the amount of gasoline `g` produced daily.

These constraints capture the purchase limitations for each type of oil.

```
  forall( o in Oils )
    ctCapacity:
      sum( g in Gasolines )
        Blend[o][g] <= Oil[o].capacity;
```

This constraint enforces the capacity limitation on production.

```
  forall( o in Oils )
    ctCapacity:
      sum( g in Gasolines )
        Blend[o][g] <= Oil[o].capacity;
```

This constraints enforce the quality criteria for the gasoline.

```
  forall( g in Gasolines )
    ctOctane:
      sum( o in Oils )
        (Oil[o].octane - Gas[g].octane) * Blend[o][g] >= 0;
  forall( g in Gasolines )
    ctLead:
      sum( o in Oils )
        (Oil[o].lead - Gas[g].lead) * Blend[o][g] <= 0;
```

The objective function has four parts: the sales price of the gasoline, the purchase cost of the crude oils, the production costs, and the adverting costs.

```
maximize
  sum( g in Gasolines , o in Oils )
    (Gas[g].price - Oil[o].price - ProdCost) * Blend[o][g]
    - sum(g in Gasolines) a[g];
```

A solution to oil.mod

For the instance data given in *Data for the oil-blending planning problem (`oil.dat`)* , the optimal solution to this problem is

```
Final Solution with objective 287750.0000:
  blend = [[2088.8889 2111.1111 800.0000]
           [777.7778 4222.2222 0.0000]
           [133.3333 3166.6667 200.0000]];
  a = [0.0000 750.0000 0.0000];
```

Exploiting sparsity

A sparse multi-product transportation model (`transp3.mod`) depicts the model of the transportation problem, known as a multicommodity flow problem on a bipartite graph. Instance data are available in this file:

```
<OPL_dir>\examples\opl\transp\transp3.dat
```

which is too long to be shown here.

A sparse multi-product transportation model (`transp3.mod`)

```
{string} Cities =...;
{string} Products = ...;
float Capacity = ...;
tuple connection { string o; string d; }
tuple route {
  string p;
  connection e;
}
{route} Routes = ...;
{connection} Connections = { c | <p,c> in Routes };
tuple supply {
  string p;
  string o;
}
{supply} Supplies = { <p,c.o> | <p,c> in Routes };
float Supply[Supplies] = ...;
tuple customer {
  string p;
  string d;
}
{customer} Customers = { <p,c.d> | <p,c> in Routes };
float Demand[Customers] = ...;
float Cost[Routes] = ...;
{string} Orig[p in Products] = { c.o | <p,c> in Routes };
{string} Dest[p in Products] = { c.d | <p,c> in Routes };

{connection} CPs[p in Products] = { c | <p,c> in Routes };
assert forall(p in Products)
   sum(o in Orig[p]) Supply[<p,o>] == sum(d in Dest[p]) Demand[<p,d>];

dvar float+ Trans[Routes];

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( <o,d> in CPs[p] )
        Trans[< p,<o,d> >] == Supply[<p,o>];
```

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

This is a classic transportation problem with the addition of a capacity constraint on the inter-cities connections. The model is, of course, not appropriate for large-scale transportation problems, where only a fraction of the cities are connected and a fraction of the products are sent along the connections. This section discusses how to exploit the sparsity of large-scale problems. OPL offers more support than other modeling languages in this respect, because it can use tuples and arrays indexed by arbitrary finite sets. The methodology for exploiting sparsity in OPL consists of mirroring, in the model, the structure of the application. This structure can be inferred from the objective function and the constraints of the application. For instance, the capacity constraint for the transportation application can be phrased as

"The products sent along any given connection may not exceed the given capacity."

This constraint helps identify two main concepts in the application. The first is the connection between two cities, which can be represented explicitly by a data type

```
tuple connection { string o; string d; }
```

to manipulate connections as first-class objects. The second fundamental concept is the transportation of a product along a connection, called a *route* in this section. Once again, this concept can be represented explicitly by a data type

```
tuple connection { string o; string d; }
```

to manipulate routes directly. The supply and demand constraints exhibit two other fundamental concepts: product suppliers (i.e., the association of a product and a city supplying it) and product consumers (i.e., the association of a product and a city consuming it). The data types

```
tuple Supply { string p; string o; };
tuple Customer { string p; string d; };
```

may be used to represent them.

Once the concepts are identified, an appropriate data representation can be chosen so that the model can generate constraints efficiently. Of course, the user data is not necessarily expressed in this representation, but it is usually easy in OPL to transform the user data into an appropriate representation.

A good representation for this application consists of a set of connections, a set of routes, the cost of the routes, and the demand and supply information. For example:

```
{route} Routes = ...;
{connection} Connections = { c | <p,c> in Routes };
tuple supply {
```

```
  string p;
  string o;
}
{supply} Supplies = { <p,c.o> | <p,c> in Routes };
float Supply[Supplies] = ...;
tuple customer {
  string p;
  string d;
}
{customer} Customers = { <p,c.d> | <p,c> in Routes };
float Demand[Customers] = ...;
float Cost[Routes] = ...;
```

Note that the connections, suppliers, and customers are derived automatically from the routes. It is also useful to derive the following data to simplify the constraint statement:

```
{string} Orig[p in Products] = { c.o | <p,c> in Routes };
{string} Dest[p in Products] = { c.d | <p,c> in Routes };

{connection} CPs[p in Products] = { c | <p,c> in Routes };
```

The objective function and the constraints can now be stated naturally. The objective function

"minimize the transportation costs along all routes"

is expressed elegantly as

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

The supply constraint, which can be phrased as

"for every product and every city shipping the product, the summation of all transportations from that city to a city where the product is in demand is equal to the supply of the product at the supplying city"

is formalized by

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

The demand constraints are stated in a similar way. The capacity constraints are stated elegantly as

```
  forall(c in Connections)
    ctCapacity:
      sum( <p,c> in Routes )
        Trans[<p,c>] <= Capacity;
```

This statement is efficient, since OPL retrieves the product from the routes in an efficient way when the connection is known. The complete model is shown in *A sparse multi-product transportation model (`transp3.mod`)*.

Assume now that part of the user data is given by a relational table that contains tuples of the form `<o,d,p,c>` indicating that a connection between cities `o` and `d` transports product `p` at cost `c`. This data can be transformed into the representation used in *A sparse multi-product transportation model (`transp3.mod`)*. The routes can be obtained as

```
{Route} Routes = { < <o,d>,p> | <p,o,d,c> in TableRoutes };
```

and the costs as

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

Both preprocessing instructions are linear in the size of the table.

Sensitivity analysis

Finding the optimal solution to a linear programming model is important, but very often you need to know what happens when data values are changed. You need sensitivity information such as the reduced cost, or the basis status for variables.

Some types of sensitivity information are made available by IBM® ILOG® OPL.

- Basis status
- Reduced cost or opportunity cost
- Information on constraints

Basis status

You can get the basis status by calling the method `getBasisStatus` on a `IloOplCplexBasis` object. This API is documented in the *IBM ILOG Script Extensions Reference Manual*. See also `mulprod_main.mod` for an example.

Reduced cost or opportunity cost

The reduced cost provides the rate of change in the objective for each nonbasic variable as it moves from the bound at which it resides. The most common type of variable has a lower bound of 0 and an infinite upper bound. In this case, the reduced cost indicates the rate of change in the objective as the variable moves to a nonzero value.

If `v` is a decision variable, call:

```
v.reducedCost
```

See also *Displaying results*.

Information on constraints

For constraints, the dual variable measures the rate of change in the objective as the right hand side of the constraint changes. For example, with a capacity constraint, the dual variable measures the improvement in the objective per unit of additional capacity.

Sensitivity analysis on constraints summarizes what information you can get on a constraint `c`.

Sensitivity analysis on constraints

To get	Call
the value of the associated dual variable	`c.dual`
the slack	`c.slack`
the lower bound	`c.LB`
the upper bound	`c.UB`

This API is documented in the *IBM ILOG Script Extensions Reference Manual*. You can also read this information in the Problem Browser of the IDE by clicking on a constraint label after a run configuration has been executed.

Integer programming

Defines integer programming and describes a set covering problem, a warehouse location problem, a fixed-charge problem, and integer relaxation.

In this section

What is integer programming?
Defines integer programming.

Set covering
Describes the problem and presents the model and data files.

Warehouse location
Describes the problem and presents the model and data files.

Fixed-charge problems
Describes the problem and presents the model and data files.

Integer relaxation
Presents a model that shows how to relax integer constraints then undo the relaxation.

What is integer programming?

Integer programming is the class of problems defined as the optimization of a linear function subject to linear constraints over integer variables.

Integer programs are, in general, much harder to solve than linear programs and the size of integer programs that can be solved efficiently is much smaller than that of linear programs. This section reviews a number of typical integer programs.

Set covering

Consider selecting workers to build a house. The construction of a house can be divided into a number of tasks, each requiring a number of skills (e.g., plumbing or masonry). A worker may or may not perform a task, depending on skills. In addition, each worker can be hired for a cost that also depends on his qualifications. The problem consists of selecting a set of workers to perform all the tasks, while minimizing the cost. This is known as a set-covering problem. The key idea in modeling a set-covering problem as an integer program is to associate a 0/1 variable with each worker to represent whether the worker is hired. To make sure that all the tasks are performed, it is sufficient to choose at least one worker by task. This constraint can be expressed by a simple linear inequality.

A set-covering model (`covering.mod`) describes a set-covering model for this problem and *Instance data for the set-covering model (`covering.dat`)* shows some instance data.

A set-covering model (`covering.mod`)

```
int NbWorkers = ...;
range Workers = 1..NbWorkers;
{string} Tasks = ...;
{int} Qualified[Tasks] = ...;
assert
  forall( t in Tasks , i in Qualified[t] ) i in asSet(Workers);
//alternate formulation:
assert
  forall( t in Tasks )
    card(Qualified[t] inter asSet(Workers))==card(Qualified[t]);
int Cost[Workers] = ...;
dvar boolean Hire[Workers];

minimize
  sum(c in Workers) Cost[c] * Hire[c];
subject to {
  forall(j in Tasks)
    ct:
      sum( c in Qualified[j] )
        Hire[c] >= 1;
}
{int} Crew = { c | c in Workers : Hire[c] == 1 };
execute DISPLAY {
  writeln("Crew=",Crew);
}
```

The first instruction in the model declares a number of workers as an integer, a range for the workers, and a `string` type for the tasks. The instruction

```
{int} Qualified[Tasks] = ...;
```

declares the workers qualified to perform a given task, Therefore, `Qualified[Tasks]` is the set of workers able to perform task `t`.

The problem variables

```
dvar boolean Hire[Workers];
```

indicate whether a worker is hired for the project.

The constraints

```
  forall(j in Tasks)
    ct:
      sum( c in Qualified[j] )
        Hire[c] >= 1;
```

make sure that each task is covered by at least one worker.

Note also the declaration

```
{int} Crew = { c | c in Workers : Hire[c] == 1 };
```

which collects all the hired workers in the set `crew` to produce a more pleasing representation of the results.

Instance data for the set-covering model (`covering.dat`) shows data for an instance of this model.

Instance data for the set-covering model (`covering.dat`)

```
NbWorkers = 32;
Tasks = { masonry, carpentry, plumbing, ceiling,
     electricity, heating, insulation, roofing,
     painting, windows, facade, garden,
     garage, driveway, moving };
Qualified = [
   { 1  9 19  22  25  28  31 }
   { 2 12 15 19 21 23 27 29 30 31 32 }
   { 3 10 19 24 26 30 32 }
   { 4 21 25 28 32 }
   { 5 11 16 22 23 27 31 }
   { 6 20 24 26 30 32 }
   { 7 12 17 25 30 31 }
   {  8 17 20 22 23  }
   {  9 13 14  26 29 30 31 }
   {  10 21 25 31 32 }
   {  14 15 18 23 24 27 30 32 }
   {  18 19 22 24 26 29 31 }
   {  11 20 25 28 30 32 }
   {  16 19 23 31 }
   {  9 18 26 28 31 32 }
];
Cost = [ 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 6 6 6 7 8 9 ];
```

A solution to covering.mod

For the instance data given in *Instance data for the set-covering model (`covering.dat`)*, OPL returns the solution

```
Optimal solution found with objective: 14
crew= {23 25 26}
```

Warehouse location

Warehouse location is another typical integer-programming problem. Suppose a company that is considering a number of locations for building warehouses to supply its existing stores. Each possible warehouse has a fixed maintenance cost and a maximum capacity specifying how many stores it can support. In addition, each store can be supplied by only one warehouse and the supply cost to the store differs according to the warehouse selected. The application consists of choosing which warehouses to build and which of them should supply the various stores in order to minimize the total cost, i.e., the sum of the fixed and supply costs. The instance used in this section considers five warehouses and 10 stores. The fixed costs for the warehouses are all identical and equal to 30. *Instance data for the warehouse-location problem* depicts the transportation costs and the capacity constraints.

Instance data for the warehouse-location problem

	Bonn	Bordeaux	London	Paris	Rome
capacity	1	4	2	1	3
store1	20	24	11	25	30
store2	28	27	82	83	74
store3	74	97	71	96	70
store4	2	55	73	69	61
store5	46	96	59	83	4
store6	42	22	29	67	59
store7	1	5	73	59	56
store8	10	73	13	43	96
store9	93	35	63	85	46
store10	47	65	55	71	95

The key idea in representing a warehouse-location problem as an integer program consists of using a 0-1 variable for each (warehouse, store) pair to represent whether a warehouse supplies a store. In addition, the model also associates a variable with each warehouse to indicate whether the warehouse is selected. Once these variables are declared, the constraints state that each store must be supplied by a warehouse, that each store can be supplied by only an open warehouse, and that each warehouse cannot deliver more stores than its allowed capacity. The most delicate aspect of the modeling is expressing that a warehouse can supply a store only when it is open. These constraints can be expressed by inequalities of the form

```
forall( w in Warehouses, s in Stores )
  ctUseOpenWarehouses:
    Supply[s][w] <= Open[w];
forall( w in Warehouses )
  ctMaxUseOfWarehouse:
    sum( s in Stores )
      Supply[s][w] <= Capacity[w];
```

which ensures that when warehouse `w` is not open, it does not supply store `s`. This follows from the fact that `open[w] == 0` implies `supply[w][s] == 0`.

As an alternative, you can write:

```
  forall(c in Connections)
    ctCapacity:
      sum( <p,c> in Routes )
        Trans[<p,c>] <= Capacity;
```

This formulation implies that a closed warehouse has no capacity.

A warehouse-location model (`warehouse.mod`) describes an integer program for the warehouse-location problem, and *Data for the warehouse-location model (`warehouse.dat`)* depicts some instance data.

A warehouse-location model (`warehouse.mod`)

```
int Fixed = ...;
{string} Warehouses = ...;
int NbStores = ...;
range Stores = 0..NbStores-1;
int Capacity[Warehouses] = ...;
int SupplyCost[Stores][Warehouses] = ...;
dvar boolean Open[Warehouses];
dvar boolean Supply[Stores][Warehouses];


minimize
  sum( w in Warehouses )
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];


subject to{
  forall( s in Stores )
    ctEachStoreHasOneWarehouse:
      sum( w in  Warehouses )
        Supply[s][w] == 1;
  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
      sum( s in Stores )
        Supply[s][w] <= Capacity[w];
}

{int} Storesof[w in Warehouses] = { s | s in Stores : Supply[s][w] == 1 };
execute DISPLAY_RESULTS{
  writeln("Open=",Open);
  writeln("Storesof=",Storesof);
}
```

Data for the warehouse-location model (`warehouse.dat`)

```
Fixed = 30;
Warehouses = {Bonn,Bordeaux,London,Paris,Rome};
NbStores = 10;
Capacity = [1,4,2,1,3];
SupplyCost = [
       [ 20, 24, 11, 25, 30 ],
       [ 28, 27, 82, 83, 74 ],
       [ 74, 97, 71, 96, 70 ],
       [  2, 55, 73, 69, 61 ],
       [ 46, 96, 59, 83,  4 ],
       [ 42, 22, 29, 67, 59 ],
       [  1,  5, 73, 59, 56 ],
       [ 10, 73, 13, 43, 96 ],
       [ 93, 35, 63, 85, 46 ],
       [ 47, 65, 55, 71, 95 ] ];
```

The statement declares the warehouses and the stores, the fixed cost of the warehouses, and the supply cost of a store for each warehouse. The problem variables

```
dvar boolean Open[Warehouses];
dvar boolean Supply[Stores][Warehouses];
```

represent which warehouses supply the stores, i.e., `supply[s][w]` is `1` if warehouse `w` supplies store `s` and zero otherwise.

The objective function

```
minimize
  sum( w in Warehouses )
    Fixed * Open[w] +
  sum( w in Warehouses , s in Stores )
    SupplyCost[s][w] * Supply[s][w];
```

expresses the goal that the model minimizes the fixed cost of the selected warehouses and the supply costs of the stores.

The constraint

```
  forall( s in Stores )
    ctEachStoreHasOneWarehouse:
      sum( w in  Warehouses )
        Supply[s][w] == 1;
```

states that a store must be supplied by exactly one warehouse.

The constraint

```
  forall( w in Warehouses, s in Stores )
    ctUseOpenWarehouses:
      Supply[s][w] <= Open[w];
  forall( w in Warehouses )
    ctMaxUseOfWarehouse:
```

```
      sum( s in Stores )
        Supply[s][w] <= Capacity[w];
```

expresses the capacity constraints for the warehouses and makes sure that a warehouse supplies a store only if the warehouse is open.

A solution to warehouse.mod

For the instance data shown in *Data for the warehouse-location model (`warehouse.dat`)*, OPL returns the optimal solution

```
Optimal solution found with objective: 383
open= [1 1 1 0 1]
storesof= [{3} {1 5 6 8} {7 9} {} {0 2 4}]
```

Fixed-charge problems

Fixed-charge problems are another classic application of integer programs (see *Applications and Algorithms* by W. Winston in the Bibliography). They resemble some of the production problems seen previously but differ in two respects: the production is an integer value (e.g., a factory must produce bikes or toys), and the factories need to rent (or acquire) some tools to produce some of the products. Consider the following problem. A company manufactures shirts, shorts, and pants. Each product requires a number of hours of labor and a certain amount of cloth, and the company has a limited capacity of both. In addition, all of these products can be manufactured only by renting an appropriate machine. The profit on the products (excluding the cost of renting the equipment) are also known. The company would like to maximize its profit.

A fixed-charge model (`fixed.mod`) shows a model for fixed charge problems, while *Data for the fixed-charge model (`fixed.dat`)* gives some instance data.

A fixed-charge model (`fixed.mod`)

```
{string} Machines = ...;
{string} Products = ...;
{string} Resources = ...;

int Capacity[Resources] = ...;
int MaxProduction = max(r in Resources) Capacity[r];
int RentingCost[Machines] = ...;
tuple productType {
   int profit;
   {string} machines;
   int use[Resources];
}
productType Product[Products] = ...;

dvar boolean Rent[Machines];
dvar int Produce[Products] in 0..MaxProduction;

constraint ctMaxProd[Products][Machines];

maximize
  sum( p in Products )
    Product[p].profit * Produce[p] -
  sum( m in Machines )
    RentingCost[m] * Rent[m];

subject to {
  forall( r in Resources )
    ctCapacity:
      sum( p in Products )
        Product[p].use[r] * Produce[p] <= Capacity[r];
    forall( p in Products , m in Product[p].machines )
      ctMaxProd[p][m]:
        Produce[p] <= MaxProduction * Rent[m];
}
```

Data for the fixed-charge model (`fixed.dat`)

```
Machines = { shirtM shortM pantM };
Products = { shirt shorts pants };
Resources = { labor cloth };
Capacity = [ 150 160 ];
RentingCost = [ 200 150 100 ];
Product = [
      <6 {shirtM} [ 3 4 ] >
      <4 {shortM} [ 2 3 ] >
      <7 {pantM} [ 6 4 ] >
          ];
```

The integer program for this model uses two sets of variables: production variables and rental variables. A production variable `produce[p]` describes the production of product `p`; a rental variable `rent[m]` is a `0-1` variable representing whether machine `m` is rented. Most of the constraints are similar to traditional production problems and pose few difficulties. The most delicate aspect of the modeling is expressing that a product can be produced only if its equipment is rented.

It is not possible to use the same idea as in the warehouse-location problem: e.g., a constraint

```
produce[p] <= rent[m]
```

is not correct, since `produce[p]` is not a Boolean variable in this model. One might be tempted to write

```
produce[p] <= produce[p] * rent[m]
```

but this constraint is not linear. The solution adopted in the model is to use an integer representing the maximal production of any product:

```
int MaxProduction = max(r in Resources) Capacity[r];
```

and write

```
produce[p] <= MaxProduction * rent[m]
```

If machine `m` is rented, then the constraint is redundant, since `MaxProduction` is chosen to be larger than `produce[p]`. Otherwise, the right-hand side is zero and product `p` cannot be manufactured. Note the data representation in this model: the type

```
tuple productType {
   int profit;
   {string} machines;
   int use[Resources];
}
```

clusters all data related to a product: its profit, the set of machines it requires, and the way it uses the resources. Note also the constraint

```
forall( p in Products , m in Product[p].machines )
  ctMaxProd[p][m]:
    Produce[p] <= MaxProduction * Rent[m];
```

which features a `forall` statement that quantifies over each product and over each machine used by the product.

Integer relaxation

OPL allows relaxation of integer constraints on decision variables. With OPL, there is a simple way to relax all integer decision variables at once and to convert a MIP problem to an LP problem: just call the method `IloOplModel.convertAllIntVars` as shown in *Relaxing an integer constraint and undoing relaxation (`convert_example.mod`)*.

The inverse operation is also available. To undo integer relaxation, call the method `IloOplModel.unconvertAllIntVars`.

Relaxing an integer constraint and undoing relaxation (`convert_example.mod`)

```
dvar int x in 0..10;


minimize x;
subject to {
  ct :
    x >= 1/2;
}

main {
  var status = 0;
  thisOplModel.generate();
  cplex.solve();
  writeln("Integer Model");
  writeln("OBJECTIVE: ",cplex.getObjValue());
  if (cplex.getObjValue() != 1) {
    status = -1;
  }

  thisOplModel.convertAllIntVars();
  cplex.solve();
  writeln("Relaxed Model");
  writeln("OBJECTIVE: ",cplex.getObjValue());
  if (cplex.getObjValue() != 0.5) {
    status = -1;
  }

  thisOplModel.unconvertAllIntVars();
  cplex.solve();
  writeln("Unrelaxed Model");
  writeln("OBJECTIVE: ",cplex.getObjValue());
  if (cplex.getObjValue() != 1) {
    status = -1;
  }
  status;
}
```

Both methods are available for flow control scripting, and in the C++, Java™ , and .NET™ Interfaces.

For more information

For details on scripting, see *IBM ILOG Script for OPL*.

For details on the API, see each *Interfaces Reference Manual*.

To learn more about what MIP relaxation is and how to use it, see the CPLEX® document or consult a textbook about linear-programming.

Mixed integer-linear programming

Defines mixed integer-linear programming and describes an upgrade to the production-planning problem to include a fixed charge for the products.

In this section

What is mixed integer-linear programming?
Defines mixed integer-linear programming.

Fixed charge in a production planning problem
Presents the model and data files, and a solution to the problem.

What is mixed integer-linear programming?

Mixed integer-linear programs are linear programs in which some variables are required to take integer values, and arise naturally in many applications.

The integer variables may come from the nature of the products (e.g., a machine may, or may not, be rented). Mixed integer-linear programs are solved using the same technology as integer programs (or vice-versa). For instance, a branch-and-bound algorithm can exploit the linear relaxation and its branching procedure is applied only to integer variables.

Fixed charge in a production planning problem

Consider how to upgrade the production-planning problem presented in *Tuples* to include a fixed charge for the products. *A fixed-charge production model (`prodmilp.mod`)* describes the new model and *Data for the fixed-charge production model (`prodmilp.dat`)* describes the instance data.

A fixed-charge production model (`prodmilp.mod`)

```
{string} Products = ...;
{string} Resources = ...;
{string} Machines = ...;
float MaxProduction = ...;

tuple typeProductData {
  float demand;
  float incost;
  float outcost;
  float use[Resources];
  string machine;
}

typeProductData Product[Products] = ...;
float Capacity[Resources] = ...;
float RentCost[Machines] = ...;

dvar boolean Rent[Machines];
dvar float+ Inside[Products];
dvar float+ Outside[Products];


minimize
  sum( p in Products )
    ( Product[p].incost * Inside[p] +
      Product[p].outcost * Outside[p] ) +
  sum( m in Machines )
    RentCost[m] * Rent[m];

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

  forall( p in Products )
    ctDemand:
      Inside[p] + Outside[p] >= Product[p].demand;

  forall( p in Products )
    ctMaxProd:
      Inside[p] <= MaxProduction * Rent[Product[p].machine];
}
```

Data for the fixed-charge production model (`prodmilp.dat`)

```
Products =  { "kluski" "capellini" "fettucine" };
Resources = { "flour" "eggs" };
Machines = { m1 m2  m3 };
RentCost = [ 20 10 5 ];
MaxProduction = 100000;
Product = #[
      kluski : < 100, 0.6, 0.8, [ 0.5, 0.2 ] m1 >
      capellini : < 200, 0.8, 0.9, [ 0.4, 0.4 ] m2 >
      fettucine : < 300, 0.3, 0.4, [ 0.3, 0.6 ] m3>
          ]#;
Capacity = [ 20, 40 ];
```

Note that the model now contains two types of variables: 0-1 variables that represent whether to rent a machine and production variables of type `float`. The product data is enhanced with a field describing the required machine, while the new constraints are modeled as in the fixed-charge problem in *A fixed-charge model (`fixed.mod`)*.

A solution to prodmilp.mod

For the instance data in *Data for the fixed-charge production model (`prodmilp.dat`)* , OPL returns the optimal solution

```
Final Solution with objective 378.3333:
  inside = [0.0000 0.0000 66.6667];
  outside = [100.0000 200.0000 233.3333];
  rent = [0 0 1];
```

Piecewise linear programming

Defines piecewise linear programming, describes an inventory problem with piecewise linear functions, compares pwl to plain linear programming, and indicates complexity issues.

In this section

What is piecewise linear programming?
Defines piecewise linear programming.

An inventory application with piecewise linear functions
Describes the problem, with its solution, and presents the model and data files.

Piecewise-linear vs. linear
Enforces a constraint that results in a mixed integer-linear program.

Complexity issues
Discusses when a piecewise linear program corresponds to a mixed integer-linear program.

What is piecewise linear programming?

Piecewise linear programs are in fact syntactic sugar for linear, integer, or mixed integer-linear programs.

In other words, a piecewise linear program can always be transformed into a mixed integer linear program and, sometimes, into a linear program. This last case is particularly interesting from a computational standpoint. Piecewise linear programs are also useful in simplifying the models for a variety of applications.

An inventory application with piecewise linear functions

This section introduces piecewise linear programs using an inventory application. This piecewise linear application is adapted from *Applications and Algorithms* by W. Winston (see the Bibliography).

The company Sailco must determine how many sailboats to produce over four time periods. The demand for the four periods is known (40, 60, 75, 25) and, in addition, an inventory of ten boats is available initially. In each period, Sailco can produce 40 boats at a cost of $400 per boat. Additional boats can be produced at a cost of $450 per boat. The inventory cost is $20 per boat and per period. *A simple inventory model (`sailco.mod`)* describes a linear program for this application and *Data for the simple inventory model (`sailco.dat`)* describes the instance data.

A simple inventory model (`sailco.mod`)

```
int NbPeriods = ...;
range Periods = 1..NbPeriods;

float Demand[Periods] = ...;
float RegularCost = ...;
float ExtraCost = ...;
float Capacity = ...;
float Inventory = ...;
float InventoryCost = ...;

dvar float+ RegulBoat[Periods];
dvar float+ ExtraBoat[Periods];
dvar float+ Inv[0..NbPeriods];


minimize
   RegularCost *
   ( sum( t in Periods ) RegulBoat[t] ) +
   ExtraCost *
   ( sum( t in Periods ) ExtraBoat[t] ) +
   InventoryCost *
   ( sum(t in Periods ) Inv[t] );

subject to {
  ctInit:
    Inv[0] == Inventory;
  forall( t in Periods )
    ctCapacity:
      RegulBoat[t] <= Capacity;
  forall( t in Periods )
    ctBoat:
      RegulBoat[t] + ExtraBoat[t] + Inv[t-1] == Inv[t] + Demand[t];
}
```

Data for the simple inventory model (`sailco.dat`)

```
NbPeriods = 4;
Demand = [ 40, 60, 75, 25 ];
RegularCost = 400;
```

```
ExtraCost = 450;
Capacity = 40;
Inventory = 10;
InventoryCost = 20;
```

The key idea underlying this model is to use two sets of variables for describing the production: variables `regulBoat[t]` represent the number of boats built at the regular price ($400 in the instance data) for period `t`, while `extraBoat[t]` represents the number of extra boats, i.e., boats built at the higher price. The model also contains inventory variables. Most of the constraints are typical for inventory problems.

In addition, the constraint

```
  forall( t in Periods )
    ctCapacity:
      RegulBoat[t] <= Capacity;
```

states that there is a capacity constraint on the regular boats. This constraint could be expressed directly as a bound but this is not of concern since it will disappear in the next model. Note also that all the variables will be given integral values for this application, although they are of type float. This is due to the problem structure, not to chance.

The constraint matrix of this problem is totally unimodular, which guarantees that the optimum has only integer values for integer data. See for instance *Combinatorial Optimization: Algorithms and Complexity* by *C.H. Papadimitriou and K. Steiglitz* for a discussion of total unimodularity (see the Bibliography).

A solution to sailco.mod

For the instance data given in *Data for the simple inventory model (`sailco.dat`)* , OPL returns the optimal solution

```
Final Solution with objective 78450.0000:
  regulBoat = [40.0000 40.0000 40.0000 25.0000];
  extraBoat = [0.0000 10.0000 35.0000 0.0000];
  inv = [10.0000 10.0000 0.0000 0.0000 0.0000];
```

Note: It is interesting to observe that the model does not preclude producing extra boats even if the production of regular boats does not reach its full capacity. This is not an issue in this model, since the extra boats are more expensive and thus are not produced in an optimal solution. It would become an issue, of course, if the cost of the extra boats is less than the regular price (because of, say, economies of scale). This case is discussed in *Piecewise-linear vs. linear*.

A piecewise linear model for this application is given in *A piecewise linear model for the simple inventory problem (`sailcopw.mod`)*.

A piecewise linear model for the simple inventory problem (`sailcopw.mod`)

```
int NbPeriods = ...;
range Periods = 1..NbPeriods;
```

```
float Demand[Periods] = ...;
float RegularCost = ...;
float ExtraCost = ...;
float Capacity = ...;
float Inventory = ...;
float InventoryCost = ...;

dvar float+ Boat[Periods];
dvar float+ Inv[0..NbPeriods];


minimize
   sum(t in Periods)
       piecewise{ RegularCost -> Capacity ; ExtraCost } Boat[t] +
                  InventoryCost  * (sum(t in Periods) Inv[t]);

subject to  {
  ctInventory:
    Inv[0] == Inventory;
  forall(t in Periods)
    ctDemand:
      Boat[t] + Inv[t-1] == Inv[t] + Demand[t];
}
```

The data description is similar in this model. What differs from the previous model presented in *A simple inventory model (`sailco.mod`)* is the choice of variables, the objective function, and the constraints. There is only one type of production variable in this model and hence there is no distinction between “regular”boats and “extra” boats. In this model, `boat[t]` represents the total production of boats during period `t`. Even more interesting is how the objective function is described: it makes it explicit that the cost of building the boats is in fact a piecewise linear function of the production

```
       piecewise{ RegularCost -> Capacity ; ExtraCost } Boat[t] +
```

OPL recognizes that this statement is in fact a linear program, applies a transformation to obtain the same code as in *A simple inventory model (`sailco.mod`)*, and returns the same optimal solution.

Piecewise-linear vs. linear

Note that *not all* piecewise linear programs are linear programs. Recall the *Note* and assume that the cost of extra boats decreases to $350, for instance (because of economies of scale). The transformation would not be correct, because a linear program would tend to use "extra"boats before all the "regular" boats have been built. The transformation must enforce a constraint stipulating that "extra" boats can only be used when all the "regular" boats have been manufactured. The resulting program is a mixed integer-linear program.

A solution to sailcopwg.mod

For the instance data given in *Data for the generalized piecewise-linear model (`sailcopwg1.dat`)*, OPL returns the optimal solution

```
Final Solution with objective 72200.0000:
  boat = [90.0000 -0.0000 100.0000 0.0000];
  inv = [10.0000 60.0000 0.0000 25.0000 0.0000];
```

This solution is interesting since it uses "extra" boats as much as possible while trying to minimize the use of boats in inventory. As a result, there is no production in the second and fourth periods. *A simple inventory model (`sailco.mod`)* can be generalized further to include more pieces. *A generalized piecewise-linear model for the simple inventory problem (`sailcopwg.mod`)* depicts such a model.

A generalized piecewise-linear model for the simple inventory problem (`sailcopwg.mod`)

```
int NbPeriods = ...;
range Periods = 1..NbPeriods;
int NbPieces = ...;

float Cost[1..NbPieces] = ...;
float Breakpoint[1..NbPieces-1] = ...;
float Demand[Periods] = ...;
float Inventory = ...;
float InventoryCost = ...;

dvar float+ Boat[Periods];
dvar float+ Inv[0..NbPeriods];


minimize
  sum( t in Periods )
    piecewise(i in 1..NbPieces-1) {
      Cost[i] -> Breakpoint[i];
      Cost[NbPieces]
    } Boat[t] +
  InventoryCost  * ( sum( t in Periods ) Inv[t] );

subject to {
  ctInit:
    Inv[0] == Inventory;
  forall( t in Periods )
```

```
    ctBoat:
      Boat[t] + Inv[t-1] == Inv[t] + Demand[t];

}
```

The interesting feature is the objective function

```
minimize
  sum( t in Periods )
    piecewise(i in 1..NbPieces-1) {
      Cost[i] -> Breakpoint[i];
      Cost[NbPieces]
    } Boat[t] +
  InventoryCost  * ( sum( t in Periods ) Inv[t] );
```

which is now generic in the number of pieces. *Data for the generalized piecewise-linear model (`sailcopwg1.dat`)* describes the same instance data for this model.

Data for the generalized piecewise-linear model (`sailcopwg1.dat`)

```
NbPeriods = 4;
Demand = [ 40, 60, 75, 25 ];
NbPieces = 2;
Cost = [ 400, 450 ];
Breakpoint = [ 40 ];
Inventory = 10;
InventoryCost = 20;
```

Consider now adding a constraint stipulating that the maximum number of boats produced in each period cannot exceed fifty on the instance data depicted in *Another instance data item for the generalized piecewise-linear model (`sailcopwg2.dat`)*.

Another instance data item for the generalized piecewise-linear model (`sailcopwg2.dat`)

```
NbPeriods = 4;
Demand = [ 40, 60, 75, 25 ];
NbPieces = 3;
Cost = [ 300, 400, 450 ];
Breakpoint = [ 30, 40 ];
Inventory = 10;
InventoryCost = 20;
```

This new constraint has a dramatic effect on the model, which is now infeasible. Piecewise linear functions can be used here to understand where the infeasibility comes from. The key insight is to replace the capacity constraint by yet another piece in the piecewise linear function and to associate a huge cost with this new piece. *Instance data to deal with infeasibility (`sailcopwg3.dat`)* depicts the instance data needed to do this:

Instance data to deal with infeasibility (`sailcopwg3.dat`)

```
NbPeriods = 4;
Demand = [ 40, 60, 75, 25 ];
NbPieces = 4;
Cost = [ 300, 400, 450, 100000 ];
Breakpoint = [ 30, 40, 50 ];
Inventory = 10;
InventoryCost = 20;
```

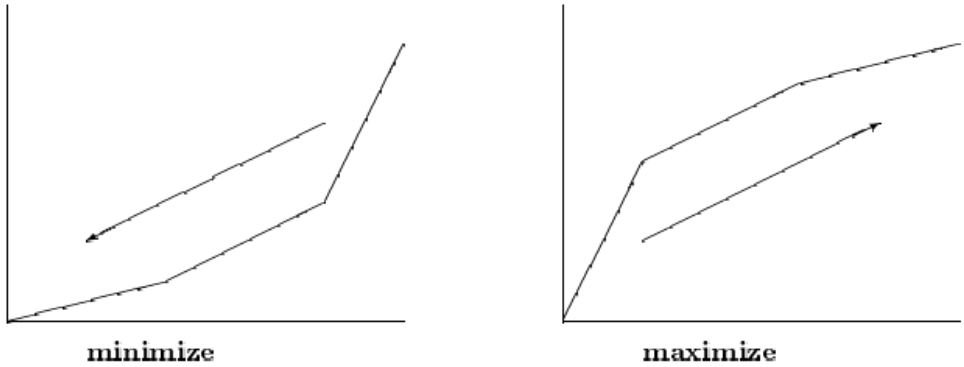
OPL produces the following optimal solution:

```
Final Solution with objective 1560600.0000:
  boat = [50.0000 50.0000 65.0000 25.0000];
  inv = [10.0000 20.0000 10.0000 0.0000 0.0000];
```

which indicates clearly where the bottlenecks (i.e., the third period) are located. The result may help Sailco to plan ahead and take appropriate measures.

Complexity issues

It is important to understand, of course, when a piecewise linear program corresponds to a mixed integer-linear program. The diagrams below describe the shapes of functions that can be used in objective functions to produce linear programs: convex piecewise linear functions in minimization problems and concave piecewise linear functions in maximization problems. Of course, summations of such functions, possibly on different variables, are also appropriate. Similar considerations apply to constraints. A convex piecewise linear function may appear on the left-hand side of an ≤ inequality and on the right-hand side of an ≥ inequality. A concave piecewise linear function may appear on the right-hand side of an ≤ inequality and on the left-hand side of an ≥ inequality. In other cases, the piecewise linear program is transformed into a mixed integer-linear program.



Piecewise linear functions leading to linear programs

Applications of constraint programming

Defines constraint programming and describes a column generation problem (vellino example), a production problem (car sequencing example), a time tabling problem (time tabling example), and an introductory scheduling problem.

In this section

What is constraint programming?
Provides a brief definition of constraint programming.

The vellino example (column generation)
Demonstrates how OPL not only supports constraint programming, but also cooperative uses of mathematical and constraint programming.

The car sequencing example
Demonstrates how to use the pack constraint and search phases to improve the efficiency of a sequencing model.

The time tabling example
Shows how to build a school timetable, given a set of room specifications, teacher skills, and educational requirements.

Modeling and solving a simple problem: house building
Presents a simple problem of scheduling the tasks to build a house in such a manner that minimizes an objective.

What is constraint programming?

Constraint programming consists of optimizing a function subject to logical, arithmetic, or functional constraints over discrete variables or interval variables, or finding a feasible solution to a problem defined by logical, arithmetic, or functional constraints over discrete variables or interval variables.

See *Constraint programming* for more information.

For an introduction to scheduling in OPL using constraint programming, see *Getting Started with Scheduling in IBM ILOG OPL*.

The vellino example (column generation)

Demonstrates how OPL not only supports constraint programming, but also cooperative uses of mathematical and constraint programming.

In this section

Description of the problem
Tells you what to do and where to find the files.

The models
Presents the four different model files used in the vellino example.

The results
Shows the trace displayed in the IDE after executing a run configuration.

Description of the problem

What you are going to do

You will solve a configuration problem using a decomposition schema known as column generation.

You will use:

- the CP Optimizer engine to solve the problem of generating new possible configurations, and
- the CPLEX® engine to solve the problem of selecting which is the best combination of configurations.

This configuration problem involves placing objects of different materials (glass, plastic, steel, wood, and copper) into bins of various types (red, blue, green), subject to capacity (each bin type has a maximum) and compatibility constraints. All objects must be placed into a bin and the total number of bins must be minimized.

Where to find the files

You will work with the `vellino` example at the following location:

`<OPL_dir>\examples\opl\vellino`

where `<OPL_dir>` is your installation directory.

Compatibility constraints

The compatibility constraints are the following:

1. Red bins cannot contain plastic or steel.
2. Blue bins cannot contain wood or plastic.
3. Green bins cannot contain steel or glass.
4. Red bins contain at most one wooden component.
5. Green bins contain at most two wooden components.
6. Wood requires plastic.
7. Glass excludes copper.
8. Copper excludes plastic.

Decomposition and column generation

To solve this problem, the technique used is known as column generation. Basically, the problem is a linear set covering problem where each decision variable corresponds to a possible configuration. An auxiliary problem is to generate all the possible configurations. Constraint programming is used to solve this configuration generation problem, as most of

the compatibility constraints are logical constraints for which the CP Optimizer engine offers good support.

The models

In the distributed project, you can see that there are four different model files described in the following sections:

- The model for common definitions: `vellinocommon.mod` contains the declaration of data common to all other models.
- The generation model: `vellinogenBin.mod` is the model to generate the possible configurations for bins. This is a CP model.
- The selection model: `vellinochooseBin.mod` selects a subset of configurations. This is a CPLEX® model.
- The flow control script: `vellino.mod` is only a flow control script that executes the two other models in the right order and transfers information from one to the other.
- Selection of the bins to use: passing the generated bins to the selection model

The model for common definitions

The model `vellinocommon.mod` contains definitions that are common to all the models. For example, the tuple definition:

```
tuple Bin {
   key int id;
   string color;
   int n[Components];
};
```

is used to represent configurations that are passed between the different models.

The set of available configurations is given by:

```
{Bin} Bins = ...;
```

This model is included in other models that use these definitions as follows:

```
include "vellinocommon.mod";
```

The generation model

The generation model `vellinogenBin.mod` starts with the statement:

```
using CP;
```

which means that it is solved by the CP Optimizer engine.

The decision variables are:

```
dvar int color in RColors;
dvar int n[Components] in 0..maxCapacity;
```

The decision variable `color` indicates the color of the generated configuration. Colors are represented by integer values and each `n[c]` indicates how many components of type `c` are included in the bin configuration. Then, all the compatibility constraints are easily written:

Some are just a direct expression of the problem description. For example, this constraint:

```
   n["wood"] >= 1 => n["plastic"] >= 1;
```

means that if there is at least one wood component, there needs to be at least one plastic component.

Others use intermediate structures. For example, this constraint:

```
   sum(c in Components) n[c] <= capacity_int_idx[color];
```

states that the total number of components must not exceed the capacity, depending on the color. For this, a preprocessed calculated structure has been created to go from the capacity indexed by strings to the capacity given by color indexes.

```
int capacity_int_idx[RColors] = [ord(Colors,c) : capacity[c] | c in Colors];
```

The selection model

The selection model `vellinochooseBin.mod` is also a very simple CPLEX® model.

A variable is created for each available configuration given as input, by means of the tuple structure `Bin` and given in the tuple set `Bins`:

```
dvar int produce[Bins] in 0..maxDemand;
```

The objective is to minimize the number of bins produced.

```
minimize
   sum(b in Bins) produce[b];
```

The only constraint is to cover the demand in terms of number of components:

```
subject to {
   forall(c in Components)
     demandCt: sum(b in Bins) b.n[c] * produce[b] == demand[c];
 };
```

The flow control script

The flow control script defined in `vellino.mod` links the other models with each other.

It works as follows:

1. It solves the generation model as many times as necessary to find all the possible solutions. This is easily done because the CP Optimizer engine can iterate on the feasible solutions.

2. For each solution, a new `Bin` is created and added to the current list.
3. The selection model is created and it uses the set of Bins just generated as input data.

Generation of all the configurations

Generating all the possible configurations consists in:

1. creating an instance of the `vellinogenBin.mod` model using the common data
2. using the following methods of the`IloCP` class to iterate on the feasible solutions:
   - `startNewSearch`
   - `next`
   - `endSearch`

At each solution, a new tuple is added to Bins from the current solution. You need to call the method`postProcess` on the generation model to be able to use postprocessing elements.

```
var genBin = new IloOplRunConfiguration("vellinogenBin.mod");
genBin.oplModel.addDataSource(data);
genBin.oplModel.generate();
genBin.cp.startNewSearch();
while (genBin.cp.next()) {
  genBin.oplModel.postProcess();
   data.Bins.add(genBin.oplModel.newId,
                 genBin.oplModel.colorStringValue,
                 genBin.oplModel.n.solutionValue);
}
genBin.cp.endSearch();
genBin.end();
```

Selection of the bins to use

As the generated bin configurations have been added to the `Bins` data element, you can pass this data element object to the selection model.

```
var chooseBin = new IloOplRunConfiguration("vellinochooseBin.mod");
chooseBin.cplex = cplex;
chooseBin.oplModel.addDataSource(data);
chooseBin.oplModel.generate();
chooseBin.cplex.solve();
chooseBin.oplModel.postProcess();
chooseBin.end();
```

The results

To solve the full model, you need to execute the **Vellino** run configuration. You can use the other two run configurations to test the two submodels individually. Run configurations are generally used to enable users to test their models by running different models related to the same problem (or the same model with different data sets), all within a single project. (See also *Understanding OPL projects* in *From OR to OPL and ODM*.)

After you have solved the **Vellino** run configuration, you see a trace. If you work in the IDE, the trace is in the **Console** output window. This trace first shows first what is being generated:

```
genbin
Found bin with color : red and containing elements  [1 0 0 0 0]
Found bin with color : red and containing elements  [2 0 0 0 0]
Found bin with color : red and containing elements  [3 0 0 0 0]
Found bin with color : red and containing elements  [0 0 0 0 1]
Found bin with color : red and containing elements  [0 0 0 0 2]
Found bin with color : red and containing elements  [0 0 0 0 3]
Found bin with color : blue and containing elements  [0 0 0 0 1]
Found bin with color : blue and containing elements  [1 0 0 0 0]
Found bin with color : blue and containing elements  [0 0 1 0 0]
Found bin with color : green and containing elements  [0 1 0 0 0]
Found bin with color : green and containing elements  [0 2 0 0 0]
Found bin with color : green and containing elements  [0 3 0 0 0]
Found bin with color : green and containing elements  [0 4 0 0 0]
Found bin with color : green and containing elements  [0 0 0 0 1]
Found bin with color : green and containing elements  [0 0 0 0 2]
Found bin with color : green and containing elements  [0 0 0 0 3]
Found bin with color : green and containing elements  [0 0 0 0 4]
Found bin with color : green and containing elements  [0 1 0 1 0]
Found bin with color : green and containing elements  [0 2 0 1 0]
Found bin with color : green and containing elements  [0 3 0 1 0]
Found bin with color : green and containing elements  [0 1 0 2 0]
Found bin with color : green and containing elements  [0 2 0 2 0]
```

Then, the trace shows the final selection.

```
choosebin
Chosen :
 <1 "red" [1 0 0 0 0]> : 0
 <2 "red" [2 0 0 0 0]> : 1
 <3 "red" [3 0 0 0 0]> : 0
 <4 "red" [0 0 0 0 1]> : 0
 <5 "red" [0 0 0 0 2]> : 0
 <6 "red" [0 0 0 0 3]> : 0
 <7 "blue" [0 0 0 0 1]> : 0
 <8 "blue" [1 0 0 0 0]> : 0
 <9 "blue" [0 0 1 0 0]> : 3
 <10 "green" [0 1 0 0 0]> : 0
 <11 "green" [0 2 0 0 0]> : 0
```

```
<12 "green" [0 3 0 0 0]> : 0
<13 "green" [0 4 0 0 0]> : 0
<14 "green" [0 0 0 0 1]> : 0
<15 "green" [0 0 0 0 2]> : 0
<16 "green" [0 0 0 0 3]> : 0
<17 "green" [0 0 0 0 4]> : 1
<18 "green" [0 1 0 1 0]> : 0
<19 "green" [0 2 0 1 0]> : 0
<20 "green" [0 3 0 1 0]> : 0
<21 "green" [0 1 0 2 0]> : 2
<22 "green" [0 2 0 2 0]> : 1
```

The car sequencing example

Demonstrates how to use the pack constraint and search phases to improve the efficiency of a sequencing model.

In this section

What to do and where to find the files
Introduces you to the example.

The car sequencing problem
Describes the problem.

Enhancing the model
Shows how to declare a search phase or use the pack constraint.

What to do and where to find the files

You will enhance the initial model using search phases to ease the sequencing process and the pack constraint for better propagation.

The car sequencing project (the `carseq` example) includes the following files:

- `carseq.mod`, the initial model
- `carseq.dat`, the data file.

They can be found in `<OPL_dir>\examples\opl\carseq`

where `<OPL_dir>` is your installation directory.

The car sequencing problem

Consider the following problem. A car assembly line is set up to build cars on a production line divided into cells. Five cells that install options on cars require more than one takt time to perform the operation. Their limitation is defined by a number of cars they can process in a period of time. There are seven different cars, each requires a different set of options. The production plan specifies the quantity of each car to build. The objective of the model is to compute a sequence of cars that the cells can process while minimizing the number of empty cars to insert to respect the load of the cells.

The initial model includes the following elements:

- The demand
- The production capacity
- The decision variables
- The constraints
- The data

The demand

In the following code extract:

```
int nbCars = sum (c in Confs) demand[c];
int nbSlots = ftoi(floor(nbCars * 1.1 + 5)); // 10% slack + 5 slots
int nbBlanks = nbSlots - nbCars;
range Slots = 1..nbSlots;
int option[Options,Confs] = ...;
```

The `demand` element represents the number of cars to build for each type.

The `nbSlots` element is the total number of cars to sequence. This number is multiplied by ten percent to make sure that it is possible to insert enough null cars to make the problem feasible.

The `option` array of Boolean values defines the options required for each car type. See *Instance data for the car sequencing problem (`carseq.dat`)*

The production capacity

In the following code extract, the array defines the number of cars that can be processed for an option in a period of time:

```
tuple CapacitatedWindow {
  int l;
  int u;
};
CapacitatedWindow capacity[Options] = ...;
```

The decision variables

The first decision variable defines the sequence of cars.

The second decision variable defines the length of the sequence; that is, the last non-null car.

```
dvar int slot[Slots] in AllConfs;
dvar int lastSlot in nbCars..nbSlots;
```

The objective

The objective function of the car sequencing model is to compute a sequence of cars that the cells can process while minimizing the number of empty cars to insert to respect the load of the cells.

```
minimize lastSlot - nbCars;
```

The constraints

The mode defines four constraints written as `forall` statements:

- to satisfy the demand:
- to define the options that are used for each car in the sequence.
- to define the length of the sequence

```
subject to {
  // Cardinality of configurations
  count(slot, 0) == nbBlanks;
  forall (c in Confs)
    count(slot, c) == demand[c];

  // Capacity of gliding windows
  forall(o in Options, s in Slots : s + capacity[o].u - 1 <= nbSlots)
    sum(j in s .. s + capacity[o].u - 1) allOptions[o][slot[j]] <= capacity[o]
.l;

  // Last slot
  forall (s in nbCars + 1 .. nbSlots)
    (s > lastSlot) => slot[s] == 0;
};
```

The data

The data for the car sequencing problem is initialized externally in the data file shown in *Instance data for the car sequencing problem (`carseq.dat`)*.

Instance data for the car sequencing problem (`carseq.dat`)

```
nbCars = 7;
nbOptions = 5;
demand = [5, 5, 10, 10, 10, 10, 5];

option = [
           [ 1, 0, 0, 0, 1, 1, 0],
           [ 0, 0, 1, 1, 0, 1, 0],
           [ 1, 0, 0, 0, 1, 0, 0],
           [ 1, 1, 0, 1, 0, 0, 0],
           [ 0, 0, 1, 0, 0, 0, 0]
        ];
capacity = [
      <1,2>,
      <2,3>,
      <1,3>,
      <2,5>,
      <1,5>
   ];
```

Enhancing the model

You can enhance the `carseq.mod` model in two ways:

- Declare a search phase
- Use the `pack` constraint

Search phase

You can declare a search phase that

- Branches on the slot variables in sequence
- Allocates more complex cars first

The `valueEval` expression defines the cars that are hard to sequence (for the search phases). The larger the value, the harder it is to sequence a car. For each car type, the measure is a combination of how difficult the option requirements are to satisfy, and of the number of cars to build.

```
int values[i in 0..nbCars] = i;
int valueEval[i in 0..nbCars] = sum(o in Options) option[o,maxl(i,1)]*
             (capacity[o].u div capacity[o].l)*(i!=0)+
             (demand[maxl(i, 1)]*nbCars*(i!=0)) div nbSlots + (i >0);
```

The `execute` block defines the search phase.

The `selectSmallest` function decides the type of car in the order of the sequence.

The `selectLargest` function selects first the cars that are considered hard to sequence.

```
execute {
   var f = cp.factory;

   var phase1 = f.searchPhase(slot,
               f.selectSmallest(f.varIndex(slot)),
               f.selectLargest(f.explicitValueEval(values, valueEval, 0)));


   cp.setSearchPhases(phase1);

}
```

The pack constraint

For more efficiency, you can also enhance the car sequencing model by modeling the `demandCt` constraint as the specialized constraint `pack`, which expresses the same constraint but propagates better.

```
  demandCt: pack(demandV, slot, one);
```

The time tabling example

Shows how to build a school timetable, given a set of room specifications, teacher skills, and educational requirements.

In this section

What to do and where to find the files
Introduces you to the example.

The data model
Explains how the data is organized.

Decision variables
Presents the decision variables used in the example.

Writing the core constraints
How to write constraints that model the interactions between the various components of the problem.

Adding side constraints
How to add a set of constraints to control the search.

Minimizing the makespans
With a constraint, or a surrogate constraint.

Customizing the search
By writing an `execute` block.

Postprocessing the solution
Using postprocessing statements and scripting capabilities.

What to do and where to find the files

In this tutorial, you work on the following:

- The data model
- Decision variables
- Core constraints
- Side constraints
- Minimizing the makespans
- Customizing the search
- Postprocessing the solution

You will work with the`timetabling` example, supplied at the following location:

`<OPL_dir>\examples\opl\timetabling`

where `<OPL_dir>` is your installation directory.

The data model

The data model specifies basically:

- The set of educational requirements named `RequirementSet`, modeled by a set of tuples:

```
tuple Requirement {
   string Class;            // a set of pupils
   string discipline;       // what will be taught
   int    Duration;         // course duration
   int    repetition;       // how many time the course is repeated
};
```

- A set of teacher skills `TeacherDisciplineSet`, modeled as a set of `<teacher, discipline>` pairs.
- A set of dedicated rooms `DedicatedRoomSet`, modeled as a set of `<room, discipline>` pairs.
- A set of available rooms `Room`, modeled as a set of strings.

For each educational requirement, the model specifies a course repetition parameter. The actual entities to be scheduled are instances of courses that fulfill the requirements. They are described with the following tuple:

```
tuple Instance {
  string Class;
  string discipline;
  int    Duration;
  int    repetition;
  int    id;
  int    requirementId;
};
```

where the `id` and `requirement` parameters indicate the sequential number of the course specified by the requirement of index `requirementId`. All these instances are generated by means of the following OPL construct:

```
{Instance} InstanceSet = {
  <c,d,t,r,i,z> | <c,d,t,r> in RequirementSet
                , z in ord(RequirementSet,<c,d,t,r>) .. ord
(RequirementSet,<c,d,t,r>)
                , i in 1..r
};
```

Decision variables

The instance set is used to index the decision variable arrays used for the search, as follows: the start date of each course, the room in which the course is held, the teacher in charge of the course.

```
dvar int Start[InstanceSet] in Time;              // the course starting point
dvar int room[InstanceSet] in RoomId;              // the room in which the
course is held
dvar int teacher[InstanceSet] in TeacherId;        // the teacher in charge
of the course
```

Writing the core constraints

To produce a valid time table using constraint programming, you must write constraints that model the interactions between the various components of the model: course time intervals, teachers, classes and rooms.

To write the core constraints:

1. Define the time interval corresponding to each course.

```
  forall(r in InstanceSet)
    End[r] == r.Duration + Start[r];
```

2. Ensure that each resource is not used more that once at any moment: they are unary resources.

   One way to do so consists in defining an array of variables that model the demand at any moment, like this:

```
dvar demand[t in Time] in 0..1;
...
demand[t in Time] = sum(i in Instance) (t >= start[i] && t < end[i]);
```

   However, it makes sense to create less variables: what we need is the points in time when the courses start and, therefore, need the resources; or, in other words, the demand on resources when each activity (course) starts. This is why, in the distributed time-tabling example, the choice was made to not model each possible point of time with a variable. Each course start time is considered as a point in time at which the resource usage uniqueness must be preserved.

3. Ensure that a teacher works with one class only at a time.

   When he is teaching, there is no other teaching demand for him at the same time.

```
  forall(r in InstanceSet, x in Teacher) {
    if(r.discipline in PossibleTeacherDiscipline[x])
      (sum(o in InstanceSet
                              : r.discipline in PossibleTeacherDiscipline
[x])
        (Start[o] >= Start[r])
        *(Start[o] < End[r])
        *(teacher[o] == ord(Teacher,x))) < 2;
  }
```

4. Ensure that a room is occupied by one class only, with similar constraints.

```
  forall(r in InstanceSet, x in Room) {
    if(PossibleRoom[r.discipline,x] == 1)
      (sum(o in InstanceSet : 1 == PossibleRoom[o.discipline,x])
        (Start[o] >= Start[r])
        *(Start[o] < End[r])
        *(room[o] == ord(Room,x))) < 2;
  }
```

5. Possibly, constrain the classes on the same pattern, so that they do not follow two courses simultaneously.

```
forall(r in InstanceSet, x in Class) {
  if(r.Class == x)
    (sum(o in InstanceSet : o.Class == x)
     (1 == (Start[o] >= Start[r])*(Start[o] < End[r]))) < 2;
}
```

Adding side constraints

In order to produce a more realistic schedule, you can add a set of constraints that constrain the search to follow room preferences and teacher skills, and to produce more practical time tables.

To add side constraints:

1. Make sure that the chosen room handles the discipline taught.

```
  forall(r in InstanceSet)
    room[r] in PossibleRoomIds[r.discipline];
```

   where `PossibleRoomIds` is a table of integer sets defined as:

   and `PossibleRoom` is a bi-dimensional table of Boolean values specifying which room can support which discipline.

```
int PossibleRoom[d in Discipline, x in Room] =
  <x,d> in DedicatedRoomSet
  || 0 == card({<z,k> | z in Room, k in Discipline
                : (<x,k> in DedicatedRoomSet)
                  || (<z,d> in DedicatedRoomSet)});
```

2. Ensure that a given teacher has the required skills to teach a course.

```
  forall(r in InstanceSet)
    teacher[r] in PossibleTeacherIds[r.discipline];
```

   where `PossibleTeacherIds` is defined as:

   and maps discipline names to the set of the indices for teachers who are capable of teaching this discipline, and `PossibleTeacherDiscipline` is defined as:

```
{string} PossibleTeacherDiscipline[x in Teacher] = {d | <x,d> in
TeacherDisciplineSet };
```

   and maps each teacher to the set of disciplines he can teach.

3. Ensure that, for a given class and a given discipline, the teacher remains the same.

   where the additional `classTeacher` array is modeled as:

```
dvar int classTeacher[Class,Discipline] in TeacherId; // teacher working
 once per time point
```

4. Ensure that, if a course spans more than one unit of time, it does not cross half-day boundaries.

```
  forall(i in InstanceSet : i.Duration > 1)
    (Start[i] div HalfDayDuration) == ((End[i]-1) div HalfDayDuration);
```

   Because the model contains only classes that fit half days, it is not necessary to write a similar “same day” constraint.

To also avoid having one discipline taught immediately after another, the data file contains a set of <discipline,discipline > tuples named NeedBreak.

```
NeedBreak =  {
   <"Maths","Physics">,
   <"Biology","Physics">,
   <"Economy","Biology">,
   <"Geography","Economy">,
   <"History", "Geography">
};
```

5. Using this set, state exclusion constraints.
6. Along the same line, make sure the same discipline is not taught more than once a day.
7. State that a discipline (such as Sport) is preferably taught in the morning.

```
  forall(d in MorningDiscipline, i in InstanceSet
         : i.discipline == d)
    (Start[i] % DayDuration) < HalfDayDuration;
```

8. You can also add a symmetry-breaking constraint which ensures that course numbers, for a given requirement, appear in chronological order.

Minimizing the makespans

In the distributed school time-tabling example, the makespan of the solution can be minimized so that everybody can leave the school early at the end of the week.

To minimize the makespan:

1. Add the following constraint:

   ```
   makespan == max(r in InstanceSet) End[r];
   ```

   where the variable `makespan` is declared with:

   ```
   dvar int makespan in Time;                              // ending date of
   last course
   ```

   and the `Time` range corresponds to the time table period (that is, a week):

   ```
   int HalfDayDuration = DayDuration div 2;
   int MaxTime = DayDuration*NumberOfDaysPerPeriod;
   range Time = 0..MaxTime-1;
   ```

2. Optionally, to help proving optimality, add a surrogate constraint.

   ```
   makespan >= max(c in Class) sum(r in InstanceSet : r.Class == c) r.
   Duration;
   ```

Customizing the search

Before writing constraints, you can customize the CP Optimizer search strategy by writing an `execute` block. In this time tabling problem, this is done by selecting variables by increasing the domain size, and by selecting random values. The method consisting in selecting random values helps the search process to distribute the courses homogeneously over the scheduling period.

```
var f = cp.factory;
var selectVar = f.selectSmallest(f.domainSize());
var selectValue = f.selectRandomValue();
var assignRoom = f.searchPhase(room, selectVar, selectValue);
var assignTeacher = f.searchPhase(teacher, selectVar, selectValue);
var assignStart = f.searchPhase(Start, selectVar, selectValue);
cp.setSearchPhases(assignTeacher, assignStart, assignRoom);
```

Note that the phases are assigned in a specific order:

1. Teachers, because there are only a few to choose from.
2. Start times, because once teachers are determined, there is only a limited number of possible start times for courses.
3. Rooms.

To specify search method and time limits, you can either use a scripting block, like this:

```
var p = cp.param;
p.logPeriod = 10000;
p.searchType = "DepthFirst";
p.timeLimit = 600;
```

or edit the **Constraint Programming/Search Control** page in the project settings file (`timetabling.ops`) from the OPL IDE. See *Setting programming options* in *IDE Reference*.

Postprocessing the solution

To present the solution, the model uses OPL postprocessing statements and scripting capabilities.

To specify postprocessing:

1. Define the `Course` tuple which is used to aggregate information from the `room`, `teacher` and `start` decision variable arrays.
2. Write the postprocessing script which iterates over the solution-derived course set to pretty-print, for each class, what will be the courses, the dedicated teacher, and the assigned room.

```
execute POST_PROCESS {
  timetable;
  for(var c in Class) {
    writeln("Class ", c);
    var day = 0;
    for(var t = 0; t < makespan; t++) {
      if(t % DayDuration == 0) {
        day++;
        writeln("Day ", day);
      }
      if(t % DayDuration == HalfDayDuration)
        writeln("Lunch break");
      var activity = 0;
      for(var x in timetable[t][c]) {
        activity++;
        writeln((t % DayDuration)+1, "\t",
                x.room, "\t",
                x.discipline, "\t",
                x.id, "/",
                x.repetition, "\t",
                x.teacher);
      }
      if(activity == 0)
        writeln((t % DayDuration)+1, "\tFree time");
    }
  }
}
```

Modeling and solving a simple problem: house building

In this section, you will learn how to:

- use the `dvar` interval;
- use the constraint `endBeforeStart`;
- use the expressions `startOf` and `endOf`.

You will learn how to model and solve a simple problem, a problem of scheduling the tasks involved in building a house in such a manner that minimizes an objective. Here the objective is the cost associated with performing specific tasks before a preferred earliest start date or after a preferred latest end date. Some tasks must necessarily take place before other tasks, and each task has a given duration. To find a solution to this problem using IBM® ILOG® OPL, you will use the three-stage method: describe, model, and solve.

Describe

The problem consists of assigning start dates to tasks in such a way that the resulting schedule satisfies precedence constraints and minimizes a criterion. The criterion for this problem is to minimize the earliness costs associated with starting certain tasks earlier than a given date and tardiness costs associated with completing certain tasks later than a given date.

For each task in the house building project, the following table shows the duration (measured in days) of the task along with the tasks that must finish before the task can start.

House construction tasks

Task	Duration	Preceding tasks
masonry	35	
carpentry	15	masonry
plumbing	40	masonry
ceiling	15	masonry
roofing	5	carpentry
painting	10	ceiling
windows	5	roofing
facade	10	roofing, plumbing
garden	5	roofing, plumbing
moving	5	windows, facade, garden, painting

The other information for the problem includes the earliness and tardiness costs associated with some tasks.

House construction task earliness costs

Task	Preferred earliest start date	Cost per day for starting early
masonry	25	200.0
carpentry	75	300.0
ceiling	75	100.0

House construction task tardiness costs

Task	Preferred latest end date	Cost per day for ending late
moving	100	400.0

Solving the problem consists of identifying starting dates for the tasks such that the cost, determined by the earliness and lateness costs, is minimized.

Note: In OPL, the unit of time represented by an interval variable is not defined. As a result, the size of the masonry task in this problem could be 35 hours or 35 weeks or 35 months.

Step 1: Describe the problem

The first step in modeling and solving the problem is to write a natural language description of the problem, identifying the decision variables and the constraints on these variables.

Write a natural language description of this problem. Answer these questions:

- What is the known information in this problem?
- What are the decision variables or unknowns in this problem?
- What are the constraints on these variables?
- What is the objective?

Discussion

What is the known information in this problem?

- There are ten house building tasks, each with a given duration. For each task, there is a list of tasks that must be completed before the task can start. Some tasks also have costs associated with an early start date or late end date.

What are the decision variables or unknowns in this problem?

- The unknowns are the date that each task will start. The cost is determined by the assigned start dates.

What are the constraints on these variables?

- In this case, each constraint specifies that a particular task may not begin until one or more given tasks have been completed.

What is the objective?

- The objective is to minimize the cost incurred through earliness and tardiness costs.

Model

After you have written a description of your problem, you can use IBM ILOG OPL to model and solve it.

Step 2: Open the example file

- Still working with the **scheduling_tutorial** project, open the **sched_time.mod** file in the OPL IDE's editing area.

This file is an OPL model that is only partially completed. You will fill in the blanks in each step in this lesson. At the end, you will have completed the OPL model. IBM ILOG OPL gives you the means to represent the unknowns in this problem, the interval in which each task will occur, as interval variables.

Note: **Interval variable**

Tasks are represented by the decision variable type `interval` in IBM ILOG OPL.

An interval has a start, an end, a size and a length. An interval variable allows these values to be variable in the model.

The length of a present interval variable is equal to the difference between its end time and its start time. The size is the actual amount of time the task takes to process. By default, the size is equal to the length, which is the difference between the end and the start of the interval. In general, the size is a lower bound on the length.

An interval variable may be optional. Whether an interval is present in the solution or not is represented by a decision variable. If an interval is not present in the solution, this means that any constraint on this interval acts like the interval is “not there.” Exact semantics will depend on the specific constraint.

Logical relations can be expressed between the presence statuses of interval variables, allowing, for instance, to state that whenever the interval variable `a` is present then the interval variable `b` must also be present.

In your model, you first declare the interval variables, one for each task. Each variable represents the unknown information, the scheduled interval for each activity. After the model is executed, the values assigned to these interval variables will represent the solution to the problem.

For example, to create an interval with size 35 in OPL:

```
dvar interval masonry size 35;
```

Step 3: Declare the interval variables

Add the following code after the comment `//Declare the interval variables`:

```
dvar interval masonry    size 35;
dvar interval carpentry  size 15;
dvar interval plumbing   size 40;
dvar interval ceiling    size 15;
dvar interval roofing    size 5;
dvar interval painting   size 10;
dvar interval windows    size 5;
dvar interval facade     size 10;
dvar interval garden     size 5;
dvar interval moving     size 5;
```

In this example, certain tasks can start only after other tasks have been completed. IBM ILOG OPL allows you to express constraints involving temporal relationships between pairs of interval variables using *precedence constraints*.

Note: **Precedence constraints**

Precedence constraints are used to specify when one interval variable must start or end with respect to the start or end time of another interval. The following types of precedence constraints are available; if `act1` and `act2` denote interval variables, both interval variables are present, and `delay` is a number or integer expression (0 by default), then:

- `endBeforeEnd(a, b, delay)` constrains at least the given delay to elapse between the end of `a` and the end of `b`. It imposes the inequality `endTime(a) + delay <= endTime(b).`
- `endBeforeStart(a, b, delay)` constrains at least the given delay to elapse between the end of `a` and the start of `b`. It imposes the inequality `endTime(a) + delay <= startTime(b).`
- `endAtEnd(a, b, delay)` constrains the given delay to separate the end of `a` and the end of `ab`. It imposes the equality `endTime(a) + delay == endTime(b).`
- `endAtStart(a, b, delay)` constrains the given delay to separate the end of `a` and the start of `b`. It imposes the equality `endTime(a) + delay == startTime(b).`
- `startBeforeEnd(a, b, delay)` constrains at least the given delay to elapse between the start of `a` and the end of `b`. It imposes the inequality `startTime(a) + delay <= endTime(b).`
- `startBeforeStart(a, b, delay)` constrains at least the given delay to elapse between the start of `act1` and the start of `act2`. It imposes the inequality `startTime(a) + delay <= startTime(b).`
- `startAtEnd(a, b, delay)` constrains the given delay to separate the start of `a` and the end of `b`. It imposes the equality `startTime(a) + delay == endTime(b).`

- `startAtStart(a, b, delay)` constrains the given delay to separate the start of `a` and the start of `b`. It imposes the equality `startTime(a) + delay == startTime(b)`.

If either interval `a` or `b` is not present in the solution, the constraint is automatically satisfied, and it is as if the constraint was never imposed.

Step 4: Add the precedence constraints

Add the following code after the comment `//Add the precedence constraints`:

```
endBeforeStart(masonry,   carpentry);
endBeforeStart(masonry,   plumbing);
endBeforeStart(masonry,   ceiling);
endBeforeStart(carpentry, roofing);
endBeforeStart(ceiling,   painting);
endBeforeStart(roofing,   windows);
endBeforeStart(roofing,   facade);
endBeforeStart(plumbing,  facade);
endBeforeStart(roofing,   garden);
endBeforeStart(plumbing,  garden);
endBeforeStart(windows,   moving);
endBeforeStart(facade,    moving);
endBeforeStart(garden,    moving);
endBeforeStart(painting,  moving);
```

To model the cost for starting a task earlier than the preferred starting date, you use the expression `startOf` that represents the start time of an interval variable as an integer expression.

For each task that has an earliest preferred start date, you determine how many days before the preferred date it is scheduled to start using the expression `startOf`; this expression can be negative if the task starts after the preferred date. By taking the maximum of this value and 0 using `maxl`, you determine how many days early the task is scheduled to start. Weighting this value with the cost per day of starting early, you determine the cost associated with the task.

The cost for ending a task later than the preferred date is modeled in a similar manner using the expression `endOf`. The earliness and lateness costs can be summed to determine the total cost.

Step 5: Add the objective

Add the following code after the comment `//Add the objective`:

```
minimize 400 * maxl(endOf(moving) - 100, 0)     +
         200 * maxl(25 - startOf(masonry), 0)   +
         300 * maxl(75 - startOf(carpentry), 0) +
         100 * maxl(75 - startOf(ceiling), 0);
```

Solve

Solving a problem consists of finding a value for each decision variable so that all constraints are satisfied. You may not always know beforehand whether there is a solution that satisfies all the constraints of the problem. In some cases, there may be no solution. In other cases, there may be many solutions to a problem.

Step 6: Execute and display the solution

After a solution has been found, you can use the `start` and `end` properties of the interval variables to access the assigned intervals. The code for displaying the solution has been provided for you:

```
execute {
  writeln("Masonry  : " + masonry.start   + ".." + masonry.end);
  writeln("Carpentry: " + carpentry.start + ".." + carpentry.end);
  writeln("Plumbing : " + plumbing.start  + ".." + plumbing.end);
  writeln("Ceiling  : " + ceiling.start   + ".." + ceiling.end);
  writeln("Roofing  : " + roofing.start   + ".." + roofing.end);
  writeln("Painting : " + painting.start  + ".." + painting.end);
  writeln("Windows  : " + windows.start   + ".." + windows.end);
  writeln("Facade   : " + facade.start    + ".." + facade.end);
  writeln("Garden   : " + garden.start    + ".." + garden.end);
  writeln("Moving   : " + moving.start    + ".." + moving.end);
}
```

Step 7: Run the model

Run the model. You should get the following results:

```
OBJECTIVE: 5000
Masonry  : 20..55
Carpentry: 75..90
Plumbing : 55..95
Ceiling  : 75..90
Roofing  : 90..95
Painting : 90..100
Windows  : 95..100
Facade   : 95..105
Garden   : 95..100
Moving   : 105..110
```

As you can see, the overall cost is 5000 and moving will be completed by day 110.

You can also view the complete program online in the `<OPL_dir>/examples/opl/sched_time/sched_time.mod` file.

Quadratic programming

OPL supports quadratic programming (QP), including quadratically-constrained programming (QCP), mixed integer quadratic programming (MIQP), and mixed-integer quadratically-constrained programming (MIQCP).

Conventionally, a quadratic program is formulated this way:

Minimize 1/2 $x^T Q x + c^T x$

subject to $Ax \sim b$

with these bounds $l\ b \leq x \leq ub$

where the relation ~ may be any combination of equal to, less than or equal to, greater than or equal to, or range constraints. As in other problem formulations, `l` indicates lower and `u` upper bounds. Q is a matrix of objective function coefficients. That is, the elements `Q jj` are the coefficients of the quadratic terms `xj^ 2`, and the elements `Q ij` and `Q ji` are summed to make the coefficient of the term `x i x j`.

Here is an example of quadratic objective problem:

A quadratic objective problem (`qpex1.mod`)

```
dvar float x[0..2] in 0..40;


maximize
  x[0] + 2 * x[1] + 3 * x[2]
  - 0.5 *
    ( 33*x[0]^2 + 22*x[1]^2 + 11*x[2]^2
    - 12*x[0]*x[1] - 23*x[1]*x[2] );

subject to {
  ct1:  - x[0] +     x[1] + x[2] <= 20;
  ct2:    x[0] - 3 * x[1] + x[2] <= 30;
}
```

Here is an example of quadratic constraint problem.

A quadratic constraint problem (`qcpex1.mod`)

```
dvar float x[0..2] in 0..40;


maximize
  x[0] + 2 * x[1] + 3 * x[2]
  - 0.5 * ( 33 * x[0]^2 + 22 * x[1]^2 + 11 * x[2]^2
          - 12 * x[0] * x[1] - 23 *x [1] * x[2] );

subject to {
  ct1:  - x[0] +     x[1] + x[2] <= 20;
  ct2:    x[0] - 3 * x[1] + x[2] <= 30;
  ct3:    x[0]^2 + x[1]^2 + x[2]^2 <= 1.0;
}
```

Quadratic programming is described in detail in the *IBM ILOG CPLEX User's Manual*.

Refer to the sections:

- *Solving Problems with a Quadratic Objective (QP)*
- *Solving Problems with Quadratic Constraints (QCP)*
- *Solving Mixed Integer Programming Problems (MIP)*

for information on MIQP and MIQCP.

Tutorial: Using CPLEX logical constraints

Demonstrates how to use logical constraints in an application.

In this section

What are logical constraints?
Defines CPLEX logical constraints.

Description of the problem
Includes what to do and where to find the files.

Representing the data
Describes the elements that are necessary to represent the problem accurately.

Using logical constraints
Describes how logical constraints are automatically transformed in OPL as based on the CPLEX solving engine.

What are logical constraints?

CPLEX® logical constraints are a particular kind of discrete or numerical constraint that transforms parts of your problem automatically for you.

See *Logical constraints for CPLEX* in the *Constraints* section of the *Language Reference Manual* for more conceptual information. This tutorial explains how to use logical constraints based on a blending problem.

Description of the problem

The problem is to plan the blending of five kinds of oil, organized in two categories (two kinds of vegetable oils and three kinds of non vegetable oils) into batches of blended products over six months.

Some of the oil is already available in storage. There is an initial stock of oil of 500 tons of each raw type when planning begins. An equal stock should exist in storage at the end of the plan. Up to 1000 tons of each type of raw oil can be stored each month for later use. The price for storage of raw oils is 5 monetary units per ton. Refined oil cannot be stored. The blended product cannot be stored either.

The rest of the oil (that is, any not available in storage) must be bought in quantities to meet the blending requirements. The price of each kind of oil varies over the six-month period. The two categories of oil cannot be refined on the same production line.

There is a limit on how much oil of each category (vegetable or non vegetable) can be refined in a given month:

- Not more than 200 tons of vegetable oil can be refined per month.
- Not more than 250 tons of non vegetable oil can be refined per month.

There are constraints on the blending of oils:

- The product cannot blend more than three oils.
- When a given type of oil is blended into the product, at least 20 tons of that type must be used.
- If either vegetable oil 1 (v1) or vegetable oil 2 (v2) is blended in the product, then non vegetable oil 3 (o3) must also be blended in that product.

The final product (refined and blended) sells for a known price: 150 monetary units per ton.

The aim of the six-month plan is to minimize production and storage costs while maximizing profit.

What you are going to do

The example used is a standard industrial problem of food manufacturing as formulated by H.P. Williams (food manufacturing 2 in *Model Building in Mathematical Programming*). The aim of the problem is to blend a number of oils cost effectively in monthly batches. In this form of the problem, the number of ingredients in a blend must be limited, and extra conditions are added to govern which oils can be blended.

Where to find the files

The food manufacturing example is supplied as the `foodmanufact` example at the following location:

`<OPL_dir>\examples\opl\foodmanufact`

where `<OPL_dir>` is your installation directory.

Representing the data

Describes the elements that are necessary to represent the problem accurately.

In this section

What is known?
Describes how known facts are represented in the data file.

What is unknown?
Describes how the decision variables can be represented.

What are the constraints?
Describes how to represent the various constraints in the problem.

What is the objective?
Describes how to represent the profit on a monthly basis.

What is known?

In this particular example, the planning period is six months, and there are five kinds of oil to be blended. Those details are expressed as constants, like this:

```
{string} Products = ...;

int NbMonths   = ...;
range Months = 1..NbMonths;
```

represented in the `foodmanufact.dat` data file as:

```
Products = { "v1", "v2", "o1", "o2", "o3" };

NbMonths   = 6;
```

The varying price of the five kinds of oil over the six-month planning period is expressed like this:

```
float Cost[Months][Products] = ...;
```

represented in the `foodmanufact.dat` data file as the following numeric matrix:

```
Cost = [[110.0, 120.0, 130.0, 110.0, 115.0],
[130.0, 130.0, 110.0,  90.0, 115.0],
[110.0, 140.0, 130.0, 100.0,  95.0],
[120.0, 110.0, 120.0, 120.0, 125.0],
[100.0, 120.0, 150.0, 110.0, 105.0],
[90.0, 100.0, 140.0,  80.0, 135.0]];
```

What is unknown?

The variables of the problem can be represented in arrays:

- How much blended, refined oil to produce per month?
- How much raw oil to use per month?
- How much raw oil to buy per month?
- How much raw oil to store per month?

like this:

```
dvar float+ Produce[Months];
dvar float+ Use[Months][Products];
dvar float+ Buy[Months][Products];
dvar float Store[Months][Products] in 0..1000;
```

Notice that how much to use and buy is initially unknown, and thus has an infinite upper bound, whereas the amount of oil that can be stored is limited, as you know from *Description of the problem*. Consequently, one of the constraints is expressed here as the upper bound of 1000 on the amount of oil by type that can be stored per month.

What are the constraints?

As you know from *Description of the problem*, there are a variety of constraints in this problem.

For each type of oil, there must be 500 tons in storage at the end of the plan. That idea can be expressed like this:

```
forall( p in Products )
  ctStore:
    Store[NbMonths][p] == 500;
```

The constraints on production in each month can all be expressed as statements in a `forall` statement:

- Not more than 200 tons of vegetable oil can be refined.

```
  ctUse1:
    Use[m]["v1"] + Use[m]["v2"] <= 200;
```

- Not more than 250 tons of non-vegetable oil can be refined.

```
  ctUse2:
    Use[m]["o1"] + Use[m]["o2"] + Use[m]["o3"] <= 250;
```

- A blend cannot use more than three oils; or equivalently, of the five oils, two cannot be used in a given blend.

```
    ctUse7:
      (Use[m]["v1"] == 0) + (Use[m]["v2"] == 0) + (Use[m]["o1"] == 0) +
                  (Use[m]["o2"] == 0) + (Use[m]["o3"] == 0) >= 2;
```

- Blends composed of vegetable oil 1 (v1) or vegetable oil 2 (v2) must also include non vegetable oil 3 (o3).

```
    ctUse9:
      (Use[m]["v1"] >= 20) || (Use[m]["v2"] >= 20) => Use[m]["o3"] >= 20;
```

- The constraint that if an oil is used at all in a blend, at least 20 tons of it must be used is expressed like this:

```
      ctUse8:
        (Use[m][p] == 0) || (Use[m][p] >= 20);
```

- The fact that a limited amount of raw oil can be stored for later use is expressed like this:

```
  forall( p in Products ) {
    ctUse6:
      if (m == 1) {
        500 + Buy[m][p] == Use[m][p] + Store[m][p];
      }
      else {
```

```
          Store[m-1][p] + Buy[m][p] == Use[m][p] + Store[m][p];
        }
```

What is the objective?

On a monthly basis, the profit can be represented as the sale price per ton (150) multiplied by the amount produced minus the cost of production and storage, like this:

```
maximize
  sum( m in Months )
    (150 * Produce[m]
    - sum( p in Products )
      Cost[m][p] * Buy[m][p]
    - 5 * sum( p in Products )
      Store[m][p]);
```

Using logical constraints

You have already seen how to represent the logical constraints of this problem in *What are the constraints?*. However, they deserve a second glance because they illustrate an important point about logical constraints and automatic transformation in OPL as based on the CPLEX® solving engine.

```
    forall( m in Months ) {
      // Using some constraints as boolean expressions to state that at least

      // 2 of the given 5 constraints must be true.
      ctUse7:
        (Use[m]["v1"] == 0) + (Use[m]["v2"] == 0) + (Use[m]["o1"] == 0) +
                    (Use[m]["o2"] == 0) + (Use[m]["o3"] == 0) >= 2;

      // Using the "or" operator, set each Use variable to be
      // zero or greather than 20.
      forall( p in Products )
        ctUse8:
          (Use[m][p] == 0) || (Use[m][p] >= 20);

      // Using "or" and "implication" operator, set that if one of 2 given
products
      // is used more than 20 then a third one must be used more than 20 too.

      ctUse9:
        (Use[m]["v1"] >= 20) || (Use[m]["v2"] >= 20) => Use[m]["o3"] >= 20;
```

Consider, for example, the constraint that the blended product cannot use more than three oils in a batch. Given that constraint, many programmers might naturally write the following statement (or something similar):

```
(use[i][v1] != 0)
+ (use[i][v2] != 0)
+ (use[i][o1] != 0)
+ (use[i][o2] != 0)
+ (use[i][o3] != 0)
<= 3;
```

That statement expresses the same constraint without changing the set of solutions to the problem. However, the formulations are different and can lead to different running times and different amounts of memory used for the search tree. In other words, given a logical English expression, there may be more than one logical constraint for expressing it, and the different logical constraints may perform differently in terms of computing time and memory.

Logical constraints for CPLEX in the section *Constraints* of the *Language Reference Manual* introduces overloaded logical operators that you can use to combine linear, or piecewise linear constraints in OPL. In this example, notice the logical operators `==`, `>=`, `||` that appear in these logical constraints.

IBM ILOG Script for OPL

After an introduction to scripting, provides tutorials for flow control and multiple searches, flow control and column generation, and for changing default behaviors in flow control.

In this section

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.

Tutorial: Flow control and multiple searches
Shows how to use IBM® ILOG® Script flow control statements to solve a production planning model iteratively, modifying the data after each iteration.

Tutorial: Flow control and column generation
Shows how to use flow control and multiple searches to create more complex flow control scripts that involve several model definitions.

Tutorial: Changing default behaviors in flow control
Describes how to achieve finer control on the execution of a CPLEX® model by using flow control scripts to change the default behavior.

Searching for relaxation and conflicts
Explains how to write scripting statements to search for relaxation and conflicts in a model.

Using IBM ILOG Script in constraint programming
Explains how to use IBM® ILOG® Script statements to set parameters that control propagation and search and to define search phases.

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.

In this section

What is IBM ILOG Script?
Describes the scripting language for combining OPL models and interacting with them.

Preprocessing and postprocessing
Use preprocessing instructions to prepare your data for modeling (`transp4.mod` example) and postprocessing instructions to manipulate solutions (`warehouse.mod` example).

A few tips
Comments on various characteristics of IBM® ILOG® Script for OPL of which you should be aware when writing script statements.

Common pitfalls
Lists syntax errors you should avoid when writing IBM® ILOG® Script statements in OPL models.

What is IBM ILOG Script?

IBM® ILOG® Script is an implementation of JavaScript™ .

The OPL language described so far covers the requirements for modeling in optimization, that is, expressing constraints on decision variables. However, an optimization application might also need functionality for manipulating data. This “non-modeling” expressiveness of the OPL language is called *scripting*, and is available as IBM ILOG Script, a scripting language for combining OPL models and interacting with them. IBM ILOG Script an implementation of the ECMA-262 standard (also known as ECMAScript or JavaScript).

You can use OPL functions, except `and` and `or`, within IBM ILOG Script blocks by specifying the OPL namespace:

```
( Opl.xxx() )
```

Important: IBM ILOG Script for OPL manipulates script variables which are denoted by means of the keyword `var` and are different from OPL modeling decision variables, denoted by means of the keyword `dvar`.

Scripting is used in three different situations, as described in the following sections:

- Preprocessing and postprocessing to prepare data and work on solutions
- Flow control to orchestrate model, model data, and solving
- A few tips: additional information on how the language interpreter works and on data declaration
- Common pitfalls: common errors you should avoid when writing script statements in your OPL models

See also the *Reference Manual of IBM ILOG Script Extensions for OPL* for an overview and a detailed description of the IBM ILOG Script API.

Preprocessing and postprocessing

Use preprocessing instructions to prepare your data for modeling (`transp4.mod` example) and postprocessing instructions to manipulate solutions (`warehouse.mod` example).

In this section

General syntax
Discusses `execute` blocks.

Initializing arrays
The recommended method is to use a generic indexed array.

Changing option values
Use `execute` blocks to change CPLEX parameters, CP parameters, and OPL settings.

Flow control
Flow control scripting enables you to control how models are instantiated and solved.

General syntax

A block of statements for preprocessing or postprocessing is marked by the keyword `execute`:

```
execute {
  writeln("Hello World.");
}
```

Execute blocks can be named:

```
execute HELLO {
  writeln("Hello World.");
}
```

Warning: No two `execute` blocks can have the same name within the same model.

Any `execute` block placed before the objective or constraints declaration is part of preprocessing; other blocks are part of postprocessing.

The scripting context within an `execute` block corresponds to the model declarations. You can think of the statements within an `execute` block being embedded in an IBM® ILOG® Script block named `with`.

```
with (thisOPLModel) {
  writeln("Hello World.");
}
```

where `thisOPLModel` is the instance of `IloOplModel` representing the current OPL model.

Initializing arrays

In most cases, the recommended method for indexing a set of data within an array is to use a generic indexed array, as shown in *Initializing data within a generic indexed array (`transp4.mod`)*.

Initializing data within a generic indexed array (`transp4.mod`)

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

Alternatively, you can use IBM ILOG Script and write an `execute` block, as shown in *Initializing data within an execute block*.

Initializing data within an execute block

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

The `get` method throws an exception on non-existing tuples to allow you to use the result directly and continue processing instead of checking for non-null values.

Note: You don't need to initialize your array elements to zero as OPL does that for you by default.

Changing option values

You can also use `execute` blocks to change CPLEX® parameters, CP parameters, and OPL settings within an OPL model.

The code examples are available at the following location:

```
<OPL_dir>\examples\opl
```

where `<OPL_dir>` is your installation directory.

Changing CPLEX parameters

Any CPLEX® parameter can be set from a script statement in an `execute` block. In case of conflict, if a different value has been set from the IDE for the same parameter, the value set in the script statement takes precedence.

Changing CPLEX parameters via scripting (`product.mod`) shows how to switch off CPLEX presolve and enable simplex logging in the `product.mod` model.

Changing CPLEX parameters via scripting (`product.mod`)

```
execute CPX_PARAM {
  cplex.preind = 0;
  cplex.simdisplay = 2;
}
```

In *Preprocessing script statement setting a parameter (`transp4.mod`)*, the script named `PARAMS` sets a time limit on each call to the optimizer:

Preprocessing script statement setting a parameter (`transp4.mod`)

You can find the `product.mod` and `transp4.mod` models at the following location respectively:

```
<OPL_dir>\examples\opl\production
```

```
<OPL_dir>\examples\opl\transp
```

where `<OPL_dir>` is your installation directory.

CPLEX solution status

This table lists the status values for solutions to LP, QP, or MIP problems. The status value is returned by `IloCplex.status` or `IloCplex.getCplexStatus`.

Code number	Solution status
1	Optimal solution is available.
2	Problem has an unbounded ray.
3	Problem has been proven infeasible.
4	Problem has been proven either infeasible or unbounded.
5	Optimal solution is available, but with infeasibilities after unscaling.
6	Solution is available, but not proved optimal, due to numeric difficulties during optimization.
10	Stopped due to limit on number of iterations.
11	Stopped due to a time limit.
12	Stopped due to an objective limit.
13	Stopped due to a request from the user.
14	This status occurs only with the parameter `feasoptmode` set to `0` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is minimal.
15	This status occurs only with the parameter `feasoptmode` set to `1` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is optimal.
16	This status occurs only with the parameter `feasoptmode` set to `2` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is minimal.
17	This status occurs only with the parameter `feasoptmode` set to `3` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is optimal.
18	This status occurs only with the parameter `feasoptmode` set to `4` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is minimal.
19	This status occurs only with the parameter `feasoptmode` set to `5` on a continuous problem. A relaxation was successfully found and a feasible solution for the problem. (if relaxed according to that relaxation) was installed. The relaxation is optimal.
20	Model has an unbounded optimal face.
21	Stopped due to a limit on the primal objective.
22	Stopped due to a limit on the dual objective.
23	The problem under consideration was found to be feasible after phase 1 of FeasOpt. A feasible solution is available.
30	The problem appears to be feasible; no conflict is available.
31	The conflict refiner found a minimal conflict.
32	The conflict refiner concluded contradictory feasibility for the same set of constraints due to numeric problems. A conflict is available, but it is not minimal.

Code number	Solution status
33	The conflict refiner terminated because of a time limit. A conflict is available, but it is not minimal.
34	The conflict refiner terminated because of an iteration limit. A conflict is available, but it is not minimal.
35	The conflict refiner terminated because of a node limit. A conflict is available, but it is not minimal.
36	The conflict refiner terminated because of an objective limit. A conflict is available, but it is not minimal.
37	The conflict refiner terminated because of a memory limit. A conflict is available, but it is not minimal.
38	The conflict refiner terminated because a user terminated the application. A conflict is available, but it is not minimal.
101	Optimal integer solution has been found.
102	Optimal solution with the tolerance defined by `epgap` or `epagap` has been found.
103	Solution is integer infeasible
104	The limit on mixed integer solutions has been reached.
105	Node limit has been exceeded but integer solution exists.
106	Node limit has been reached; no integer solution.
107	Time limit exceeded, but integer solution exists.
108	Time limit exceeded; no integer solution.
109	Terminated because of an error, but integer solution exists.
110	Terminated because of an error; no integer solution.
111	Limit on tree memory has been reached, but an integer solution exists.
112	Limit on tree memory has been reached; no integer solution.
113	Stopped, but an integer solution exists.
114	Stopped; no integer solution.
115	Problem is optimal with unscaled infeasibilities.
116	Out of memory, no tree available, integer solution exists.
117	Out of memory, no tree available, no integer solution.
118	Problem has an unbounded ray.
119	Problem has been proved either infeasible or unbounded.
120	This status occurs only with the parameter `feasoptmode` set to `0` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is minimal.

Code number	Solution status
121	This status occurs only with the parameter `feasoptmode` set to `1` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is optimal.
122	This status occurs only with the parameter `feasoptmode` set to `2` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is minimal.
123	This status occurs only with the parameter `feasoptmode` set to `3` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is optimal.
124	This status occurs only with the parameter `feasoptmode` set to `4` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is minimal.
125	This status occurs only with the parameter `feasoptmode` set to `5` on a mixed integer problem. A relaxation was successfully found and a feasible solution for the problem (if relaxed according to that relaxation) was installed. The relaxation is optimal.
126	This status occurs only after a call to `feasOpt`, when the algorithm terminates prematurely, for example after reaching a limit. This status means that a relaxed solution is available and can be queried.
127	The problem under consideration was found to be feasible after phase 1 of `FeasOpt`. A feasible solution is available. This status is also used in the status field of solution and mipstart files for solutions from the solution pool.
128	This status occurs only after a call to the method `populate` on a MIP problem. The limit on mixed integer solutions generated by populate, as specified by the parameter `populatelim`, has been reached.
129	This status occurs only after a call to the method `populate` on a MIP problem. Populate has completed the enumeration of all solutions it could enumerate.
130	This status occurs only after a call to the method `populate` on a MIP problem. Populate has completed the enumeration of all solutions it could enumerate whose objective value fit the tolerance specified by the parameters `solnpoolagap` and `solnpoolgap`.

For more information

See the description of class `IloCplex` in the *Reference Manual of IBM ILOG Script Extensions for OPL*. You can also find the complete reference documentation of CPLEX® parameters in the CPLEX documentation (Parameters of IBM ILOG CPLEX/Parameter Table).

Changing CP parameters

You can set any constraint programming parameter from a script statement in an `execute` block. In case of conflict, if a different value has been set from the IDE for the same parameter, the value set in the script statement takes precedence.

Changing CP parameters via scripting (`timetabling.mod`) extends the `logPeriod` parameter to `10000`, sets the search type to `DepthFirst` and the time limit to `600`.

The `timetabling` example is available at the following location:

```
<OPL_dir>\examples\opl\timetabling
```

where `<OPL_dir>` is your installation directory.

Changing CP parameters via scripting (`timetabling.mod`)

```
   var p = cp.param;
   p.logPeriod = 10000;
   p.searchType = "DepthFirst";
   p.timeLimit = 600;
```

For more information

See the description of class `IloCP` in the *Reference Manual of IBM ILOG Script Extensions for OPL*. You can also find the complete reference documentation of CP parameters in the CP Optimizer documentation.

In the steel mill example, the solution is found very quickly. However, if you want to illustrate the engine log, you can decrease the periodicity (that is, the number of branches between which a line of log is printed). To do so, write:

```
execute {
  cp.param.LogPeriod = 50;
}
```

The general syntax to change engine parameters is:

```
execute {
   cp.param.param_name = param_value
}
```

Changing OPL settings

You can set certain OPL settings from a script statement in an `execute` block. Not all OPL parameters can be set by scripting: you can change only the parameters that are listed as properties of `IloOplSettings` in the *Reference Manual of IBM ILOG Script Extensions for OPL*. In case of conflict, if a different value has been set from the IDE for the same parameter, the value set in the script statement takes precedence. For an example, see *Executing preprocessing scripts* in *Using IBM ILOG Script for OPL*.

For more information

See also the description of class `IloOplSettings` in the *Reference Manual of IBM ILOG Script Extensions for OPL*.

Flow control

Flow control scripting enables you to control how models are instantiated and solved. You can use it in addition to pre- and postprocessing. More specifically, it enables you to use several models with different data, to run multiple “solve” on a model, and to modify the model data between one solve and the next. It is particularly useful when you want to solve a model with modified data several times, or if you want to use different models to solve your problem (model decomposition).

Examples

Mathematical programming (CPLEX engine)

A `main` block in an MP model:

```
model.generate();
if (cplex.solve()) {
var obj=cplex.getObjValue();
opl.postProcess();
}
```

Constraint programming (CP Optimizer engine)

Two different `main` blocks in a CP model:

```
model.generate();
if (cp.solve()) {
var obj=cp.getObjValue()
model.postProcess();
}
```

Or, to find all solutions:

```
model.generate();
cp.startNewSearch();
while (cp.next()) {
model.postProcess();
}
```

For more information

The design of the OPL extensions to IBM ILOG Script available for flow control is close to that of OPL Interfaces in C++. See *Working with OPL interfaces* in the *Interfaces User's Manual* for details.

The IBM ILOG Script API available for solving is very limited compared to `IloCplex` and `IloCP` in C++,. NET, or Java™ . See the *Reference Manual of IBM ILOG Script Extensions for OPL* for a complete list of available properties and methods.

In *Tutorial: Flow control and multiple searches*, you will work from the `mulprod_main` example to learn how to solve several times the same model with modified data.

In *Tutorial: Flow control and column generation*, you will work from the `cutstock` example to learn how to solve two different models one after the other by using the output from the first one as data to the second one.

See OPL language options in *IDE Reference* for performance aspects.

A few tips

The OPL interpreter

The OPL interpreter performs the following tasks:

- declarations: types, names for data and decision variables
- instantiation
- data sources: data given in files or other sources
- preprocessing: scripting blocks
- modeling: objective and constraints
- postprocessing

Script variables

Script variables declared in one `execute` block are not visible in other `execute` blocks.

Processing order

After the declarations, all the data sources are processed. Preprocessing is done before modeling. Postprocessing is available after a solution is found. However, some postprocessing instructions are not triggered unless the `postProcess` method is explicitly called on the model object.

Note: When the "Force element usage" option is turned off (the default value), all the declared elements are instantiated "on demand", that is, when they are first used and the interpreter issues warnings for unused elements. When you turn this option on, all elements are used and no warning message is issued.

Data initialization

If you declare the data of your project internally in the model file (as opposed to externally in a data file), you cannot access it later by means of a script statement such as:

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

Otherwise, OPL throws an error message because data elements only hold external data elements declared using the `=... (ellipsis)` syntax and read from a `.dat` file or other data source.

Control on the solve operation

The solve operation is performed by the flow control in a `main` block, via the `oplrun` command, or in the IDE.

A specific API is provided to enable advanced users to control these tasks. Please refer to the `oplrunsample.cpp` example. This file is at the following location:

```
<OPL_dir>\examples\opl_interfaces\cpp\src
```

where `<OPL_dir>` is your installation directory.

Ending objects

In preprocessing, postprocessing, or flow control context, you can end the OPL elements you don’t need to reduce overall memory consumption.

Debugging

To improve the response time of your script blocks, OPL provides the Profiler as a debugging tool. See *Profiling the execution of a model* for more information on how to identify the blocks that are good candidates for improvement.

Common pitfalls

No range syntax

The range syntax you can use in OPL modeling statements does not exist for script statements. However, that syntax is valid for JavaScript™ (ECMAScript) parsing in some cases.

For example, this statement:

```
for(var i in 1..n)
```

iterates an empty loop. The expression "`1..n`" is interpreted as the named property `n` for the number object `1`. As that property does not exist, it evaluates to `undefined`. Iterating the `undefined` value is an empty loop.

No tuple syntax

The tuple syntax in OPL modeling statements does not exist for script statements. Use the `find()` or the `get()` methods to get control of tuple objects. For example, instead of writing:

```
A[<a,b>]
```

which results in a parsing error, write

```
A[S.get(a,b)]
```

where `S` is the indexer for `A`.

IBM ILOG Script variables

All variables you declare using the `var` keyword in a scripting block are undefined in the block until they are declared. For example:

```
int a=2;

execute
{
    writeln(a);
    var a=2;
    writeln(a);
}
```

gives out

```
undefined
2
```

Tutorial: Flow control and multiple searches

Shows how to use IBM® ILOG® Script flow control statements to solve a production planning model iteratively, modifying the data after each iteration.

In this section

The production planning problem
Describes the problem and tells you where to find the files.

Procedure summary
Explains how to solve a model iteratively.

Detailed steps
Provides more detail on each step of the procedure summary.

Doing more with mulprod_main
Shows further work with the mulprod_main example, such as passing information to another model, writing an output file, modifying the CPLEX® matrix incrementally.

Basic flow control script
Presents two templates to help you write flow control scripts.

The production planning problem

What you are going to do

In this tutorial, you are going to work with the production problem presented in *A multi-period production planning problem*. This models extends the basic production-planning problem presented in *A production problem* by considering the demand for the products over several periods and allowing the company to produce more than the demand in a given period. Of course, there is an inventory cost associated with storing the additional production.

Where to find the files

You will work on a multiperiod production example, supplied as the `mulprod` example at the following location:

```
<OPL_dir>\examples\opl\mulprod
```

where `<OPL_dir>` is your installation directory.

Procedure summary

More precisely, let us assume that you want to solve the original model, then increase the capacity for the flour ingredient, then solve again. You want to do this as many times as possible. Doing so, you will obtain the optimal value for this problem for each possible value of flour capacity. When the flour capacity is too high and no solution is found, you will stop the experience.

To solve a model iteratively:

1. Define a "main" block to indicate that you want do flow control scripting to manipulate different models and/or searches.
2. Load the necessary structures: the model definition, the initial model data, and the `IloOplModel` instance that creates the link between them.
3. Generate the optimization model from the initial data.
4. Solve the current optimization model with the current data:
   - if there is no solution, then quit
   - if there is a solution, print the objective value
5. Get the data elements from an `IloOplModel` instance.
6. Modify some of the data elements: modify the data element for the flour capacity.
7. Create a new OPL model with the modified data: use the model definition and modified data elements and create a new `IloOplModel` instance to link them.
8. Complete model: generate the new current optimization model.

You can see the complete model in `mulprod_main.mod`. It iterates on the last items as long as a solution is found.

Detailed steps

Provides more detail on each step of the procedure summary.

In this section

Defining a “main” block
To operate flow control, you need to add a `main` block to your model.

Loading the necessary structures
Lists the structures needed to manipulate models and data.

Generating the optimization model
Presents the `generate()` function.

Solving the current optimization model
Presents the `solve()` function.

Getting the data elements from an IloOplModel instance
Defines data elements and explains how to access them.

Modifying data from “main” scripting
Data elements can be modified and then used as a data source for another model.

Creating a new OPL model with the modified data
Explains how to generate a new model with the modified data.

Defining a “main” block

Flow control means solving several models in sequence and, possibly, modifying data or passing results from one model to the data of another model. To operate flow control, you need to add a `main` block to your `.mod` file using this syntax:

```
main {
 ...
}
```

When a `.mod` file contains a `main` block, the IDE (or the `oplrun` command) starts the execution of the model by running the `main` block first.

Note that the two optimization models can use the same model definition (that is, the same `.mod` file) as is the case in this example.

Loading the necessary structures

The structures you will use to manipulate models and data are listed in the table below.

Scripting: structures to manipulate models and data

Name	Role
`IloOplModelDefinition`	Links to the `.mod` file representation of the model
`IloOplDataSource`	Links to a `.dat` file representation of the data
`IloCplex`	An instance of the CPLEX algorithm
`IloOplModel`	A structure linking one model definition to (possibly) one or several data sources

See the *Reference Manual of IBM ILOG Script Extensions for OPL* for more information on these classes.

When a `main` block is executed, a variable called `thisOplModel` representing the `IloOplModel` instance is available by default. This variable links to the model definition that contains the `main` block currently executed and to the associated `.dat` files (if they exist) to run the model. The model definition uses `IloOplModelSource` instance that is initialized with the model name. There is also a variable called `cplex` which corresponds to an already created instance of the CPLEX® algorithm.

If you want to run another model and/or use other data, you may create your own `IloOplModel` instance, like this:

```
var src = new IloOplModelSource("cutstock_sub.mod");
var def = new IloOplModelDefinition(src);
var opl2 = new IloOplModel(def,cplex);
```

To create a new data source using a different `.dat` file, you can write:

```
var data = new IloOplDataSource("mulprod.dat")
```

Then, to link the `IloOplModel` instance to a new data source, write:

```
opl2.addDataSource(data);
```

In the `mulprod_main` example, you don't need to create all these structures since you want to use the already defined `thisOplModel` instance which corresponds to the model included in the `mulprod_main.mod` file.

Generating the optimization model

When your `IloOplModel` instance is created, you can generate the optimization model and feed it to your CPLEX® algorithm by calling the `generate()` function. In the `mulprod_main` example, it is called on the `thisOplModel` instance:

```
thisOplModel.generate();
```

After this call, the CPLEX instance is ready to solve.

Solving the current optimization model

To solve the current optimization model, just call the `solve()` function on the `IloCplex` instance. This function returns true or false depending on whether a solution has been found. If a solution is found, you can ask for the objective value as follows:

```
if ( cplex.solve() ) {
  curr = cplex.getObjValue();
  writeln();
  writeln("OBJECTIVE: ",curr);
  ofile.writeln("Objective with capFlour = ", capFlour, " is ", curr);

}
else {
  writeln("No solution!");
  break;
}
```

Getting the data elements from an IloOplModel instance

What are data elements?

You cannot directly change the data in a "data source". A data source represents what is in a `.dat` file and the only way to change it would be to modify the `.dat` file. However, you can ask for another editable view of the data. This other view is referred to as the "data elements" of the OPL model. These data elements can be modified and then used as a data source for another model.

Using data elements

In the `mulprod_main` example, you want to get the data from the current model at each successful iteration, modify it, and use it to solve another optimization model.

1. To get the elements of the `IloOplModel` instance, write:

```
var data = produce.dataElements;
```

2. To reuse the same model definition, write:

```
var def = produce.modelDefinition;
```

Note: All the scalar elements that are in the `.dat` file (`string`, `int`, `float`) are copied whereas complex data such as arrays, sets, and tuples are shared. In other words, scalar data elements are passed by value while nonscalar data is passed by reference.

Data elements and data publishers

When calling the method `dataElements` on an `IloOplModel` instance, you obtain a container of all the data elements of this model. This container does **not** include the data source publishers: if the `IloOplModel` instance has been created by means of a data source that contained publishers, these publishers will not be included in the data elements structure. For example, if you have created an `IloOplModel` instance with a `.dat` file containing a publisher like:

```
result to DBUpdate (db,"INSERT INTO writeOPL(id) VALUES (?)");
```

and you want to use the same publisher on another `IloOplModel` instance, you must use a specific `.dat` file containing only the publishers and add it to the new `IloOplModel`, as shown in *Reusing data source publishers*.

Reusing data source publishers

```
main {
   var Source = new IloOplModelSource("writeDB.mod");
   var Def = new IloOplModelDefinition(Source);
   var Cplex = new IloCplex();
   var Data = new IloOplDataSource("writeDBfromScript.dat");


   var Opl = new IloOplModel(Def,Cplex);
   Opl.addDataSource(Data);
   Cplex.solve();
   var DataElts = Opl.dataElements;

   var Opl2 = new IloOplModel(Def,Cplex);
   Opl2.addDataSource(DataElts);
   var Data2 = new IloOplDataSource("publisherData.dat");
   Opl2.addDataSource(Data2);
   DataElts.lb = 5;
   Opl2.generate();
   if (Cplex.solve()) {
      Opl2.postProcess();
   }
   else
     writeln("no solution");
}


writeFromScript.dat :
lb=2;
DBConnection db("access","testDB.mdb");
result to DBUpdate (db,"INSERT INTO writeOPL(id) VALUES (?)");

publisherData.dat would be:
DBConnection db("access","testDB.mdb");
result to DBUpdate (db,"INSERT INTO writeOPL(id) VALUES (?)");
```

Modifying data from “main” scripting

You can access and modify data in the data elements obtained from the current OPL model, as follows:

```
data.capacity["flour"] = capFlour;
```

Then, the value of the variable `capacity["flour"]` is modified in the structure of the data elements.

Note, however, the following:

1. Scalar data, whether in the `.mod` file or the `.dat` file, cannot be modified via scripting.
2. You can use data elements to add new elements, but only for scalar types.
3. Only *external* data can be modified by script. If, in the `.mod` file, you have, for example:

   ```
   int arr[1..3] = [1,2,3];
   ```

   you cannot modify the array `arr`. You need to declare it as:

   ```
   int arr[1..3] = ...;
   ```

   and initialize it externally.

   So you would need to create a `.dat` file that contains the data to update, except for scalar data. The scalars that need to be updated would not be initialized in the `.mod`, or in a `.dat`, but in a new instance of `OplDataElements` that you can then add as a data source:

   ```
   float maxOfx = ...;
   .
   .
   main {
   .
   .
     var data = new IloOplDataSource("basicmodel.dat");
     var opl = new IloOplModel(def,cplex);
     var data2 = new IloOplDataElements();
     data2.maxOfx=11;
     opl.addDataSource(data);
     opl.addDataSource(data2);
     opl.generate();
   .
   .
   }
   ```

Creating a new OPL model with the modified data

You can now:

1. Reuse the model definition and use the modified data elements to create a new OPL model.

```
produce = new IloOplModel(def,cplex);
produce.addDataSource(data);
```

2. Generate the optimization model as before:

```
produce.generate();
```

The `cplex` instance is filled with the new optimization model which corresponds to the same model definition but uses the modified data elements.

Note: The `cplex` instance that was also used for the original model does not contain the original optimization model anymore.

Doing more with mulprod_main

Shows further work with the mulprod_main example, such as passing information to another model, writing an output file, modifying the CPLEX® matrix incrementally.

In this section

Passing information to another model
Using a basis you can pass information from one optimization model to another and accelerate the search.

Writing an output file
How to use script statements to write an output file.

Modifying the CPLEX matrix incrementally
How to change the bounds of a CPLEX constraint or variable. How to change the coefficient of a variable.

Passing information to another model

The distributed example also uses an instance of `IloOplCplexBasis` to pass the basis from one optimization model to another. Using a basis, you can pass information from one optimization model to another and accelerate the search.

To pass information to another model:

1. Create a basis structure.

   ```
   var basis = new IloOplCplexBasis();
   ```

2. Load the structure with the basis contained in a `cplex` instance.

   ```
   basis.getBasis(cplex)
   ```

3. Fill another instance with the basis.

   ```
   basis.setBasis(cplex)
   ```

Note: The basis structure is currently limited to pass basis information between two optimization models that have the same structure (same number of variables and rows), as this is the case for the `mulprod_main` example.

Writing an output file

The distributed example `mulprod_main` also illustrates how to use script statements to write an output file. To do so, you use the `IloOplOutputFile` class.

Opening a file

To open a file:

1. Write the following code:

```
var ofile = new IloOplOutputFile("mulprod_main.txt");
```

2. Then you can write statements such as:

```
     ofile.writeln("Objective with capFlour = ", capFlour, " is ", curr)
;
```

Closing the file

To close the file:

- Write the following code:

```
ofile.close();
```

Modifying the CPLEX matrix incrementally

In this tutorial, you have learned how to solve a sequence of modified OPL models by changing data in OPL. This is a useful technique, however you need to generate the CPLEX® model again after each modification. Sometimes, when the number of iterations is high and generating the new iteration takes a long time compared to solving it, you may prefer to have a direct interaction with the generated optimization model and be able to work incrementally on the result of the previous iteration. You can do so by taking advantage of the API of IBM® ILOG® Script extensions for OPL.

The `mulprod` production planning example illustrates how to change the bounds of a CPLEX constraint.

You can also:

- change the bounds of a variable
- change the coefficient of a variable in a CPLEX constraint or in the CPLEX objective

Changing bounds

Of a CPLEX constraint

To change the bound of a constraint, you can directly change the `LB` and `UB` properties on the `IloConstraint` class.

It is important to understand what happens when these methods are used: the optimization model is directly modified but the OPL model is not. Therefore, the solution given by CPLEX® corresponds to the modified optimization model, but not to the original OPL model any more. On the other hand, the advantage is that the CPLEX matrix is directly modified (not rebuilt from scratch) and any new search can take advantage of the previous ones.

You can see in the `mulprod_change_main.mod` file how the example can be modified to change the optimization model directly. In particular, the important line is the one that changes the bound of a constraint:

Changing the bound of a CPLEX constraint

```
    for(var t in thisOplModel.Periods)
      thisOplModel.ctCapacity["flour"][t].UB = capFlour;
  }
```

Of a variable

To change the bound of a variable, you can directly change the lower-bound (`LB`) and upper-bound (`UB`) properties on the `IloNumVar` class. This does not change the bound of the variable in the OPL model but **only** in the CPLEX matrix. This change is taken into account incrementally by the CPLEX engine.

Changing the coefficient of a variable

You can use the method `IloConstraint.setCoef` to change the coefficient of a variable in the invoking constraint and the method `IloObjective.setCoef` to change the coefficient of a variable in the invoking objective. The coefficient is changed **only** in the CPLEX®

matrix and in the Concert extracted model. The OPL model is not affected. On the other hand, the change is taken into account incrementally by the CPLEX engine.

Note: The method `IloCplex.setCoef` is available for all CPLEX linear constraints. It changes the engine representation directly without going through Concert.

Basic flow control script

To help you write flow control scripts, here are two templates you can start from.

Flow control script template calling a project calls a project file while *Flow control script template calling a model and data* calls model and data files.

Flow control script template calling a project

```
main {

var proj = new IloOplProject("../../../../../opl/mulprod");

var rc=proj.makeRunConfiguration();

rc.oplModel.generate();

if (rc.cplex.solve()) {

writeln("OBJ = " + rc.cplex.getObjValue());

}

else {

writeln("No solution");

}

rc.end();

proj.end();

}
```

Flow control script template calling a model and data

```
main {

var source = new IloOplModelSource("../../../../../opl/mulprod/mulprod.mod");

var cplex = new IloCplex();

var def = new IloOplModelDefinition(source);

var opl = new IloOplModel(def,cplex);

var data = new IloOplDataSource("../../../../../opl/mulprod/mulprod.dat");

opl.addDataSource(data);

opl.generate();

if (cplex.solve()) {
```

```
writeln("OBJ = " + cplex.getObjValue());
}
else {
writeln("No solution");
}
opl.end();
data.end();
def.end();
cplex.end();
source.end();
}
```

Tutorial: Flow control and column generation

Shows how to use flow control and multiple searches to create more complex flow control scripts that involve several model definitions.

In this section

What is model decomposition?
Defines decomposition for complex models.

The cutting stock problem
Describes the example and tells you where to find the files.

Procedure summary
Explains how to work with a master model and a submodel

Detailed steps
Goes into more detail on each step of the procedure summary.

Doing more with cutstock_main
Shows further work with the `cutstock_main` example, such as integer solution or executing postprocessing.

What is model decomposition?

Some models are too complex to solve, either because they are just too big or because they are too long to search. In this case, model decomposition consists in breaking down the model into several smaller models and defining a sequence to solve those smaller models so as to lead to a solution that is also a solution to the big original model.

Column generation techniques use a decomposition into two models called the master model and the submodel. Column generation techniques are the most famous among the model decomposition techniques. The process to solve a cutting stock problem includes one initial step to prepare the submodel, then a series of iterative steps.

The cutting stock problem

What you are going to do

In this tutorial, you are going to solve the cutting stock problem described in Cutting stock problems in the *Samples* manual. Here is a summary:

The problem consists of cutting big wooden boards into small shelves to meet customer demands while minimizing the number of boards used. A given instance specifies the length of the boards, the length of the shelves, and the demand for each shelf type. These variables are expressed as integers, it is therefore an integer programming problem.

In the context of column generation, the two models lend themselves to interpretation:

- The submodel consists of finding possible new patterns (i.e. ways of cutting the items).
- The master model consists of deciding how many of each of the already existing patterns have to be cut.

The whole process involves solving the master and the subproblem iteratively. At each iteration, a new cutting pattern is added, then the master problem is solved again from this new pattern. As there is more freedom in the way the boards are cut, a better solution may be found. The submodel uses a reduced-cost objective so that only the patterns that could improve the total cost are generated.

Note: Although this reduced-cost objective is explained by the simplex theory, it is not necessary to fully understand this theory to follow how the cutting stock example works.

Where to find the files

You will work with the files listed in *Files for the cutting stock example* to solve a column generation problem. You can find them at the following location:

```
<OPL_dir>\examples\opl\cutstock
```

where `<OPL_dir>` is your installation directory.

Files for the cutting stock example

`cutstock_main.mod`	The model definition for the master model; it also contains the flow control script.
`cutstock_main_elements.mod`	Similar to `cutstock_main.mod` but uses `IloOplDataElements` for the data of the submodel.
`cutstock.dat`	The initial data for the master model
`cutstock-sub.mod`	The definition of the submodel

Procedure summary

To work through model decomposition:

1. Prepare the submodel.
2. Solve the master model.
3. Update the submodel*:* prepare the data of the submodel to take into account the result of the master model, and prepare the new subproblem and regenerate its optimization model.
4. Solve the submodel.
   - If no solution with a satisfactory reduced cost is found, the process is finished.
   - If a new solution exists, the process continues.
5. Update the master model: add the current solution of the submodel to the data of the master model.
6. Prepare the new master model and regenerating its optimization model and go back to step 2.

Detailed steps

Goes into more detail on each step of the procedure summary.

In this section

Preparing the submodel
Either by using run configuration and project instances, or by using model and data file instances.

Solving the master model
Provides the syntax.

Updating the submodel
Provides the syntax.

Solving the submodel
Provides the syntax.

Updating the master model
Provides the syntax.

Preparing the new master model and regenerating its optimization model
Provides the syntax.

Ending objects
Discusses the use of the `end` method to terminate objects that are no longer necessary.

Preparing the submodel

There are two ways of initializing all that is necessary for the submodel:

- Using run configuration and project instances
- Using model and data file instances

Using run configuration and project instances

The quickest way of instantiating the model consists in using the class `IloOplProject`. An alternative method (not used in the example is presented afterwards, using `IloOplRunConfiguration`.

IloOplProject

The class `IloOplProject` allows you to create an `IloOplModel` instance. This allows you to handle settings files (`.ops`) easily.

These two classes are fully documented in the *IBM ILOG Script Reference Manual*.

The `IloOplDataElements` is created from scratch and initialized with data from the master model using the following code:

```
var subData = new IloOplDataElements();
subData.RollWidth = masterOpl.RollWidth;
subData.Size = masterOpl.Size;
```

The array `Duals` is now declared in the post processing of the master model and we pass it to the sub model as follows:

```
subData.Duals = masterOpl.Duals;
```

IloOplRunConfiguration

Alternatively, the class `IloOplRunConfiguration` could be used to create an `IloOplModel` instance in a straightforward manner by just passing the file names as arguments, as follows:

```
  var subSource = new IloOplModelSource("cutstock-sub.mod");
  var subDef = new IloOplModelDefinition(subSource);
  var subData = new IloOplDataElements();
  var subCplex = new IloCplex();
```

After the run configuration is created, you can access the `IloOplModel` instance using the `oplModel` property, as follows:

```
    subOpl.generate();
```

Using model and data file instances

You can also create the `IloOplModel` instance for the submodel “from scratch”, using a model source, model definition, and data source (see the list in *Files for the cutting stock example*).

```
var subSource = new IloOplModelSource("cutstock-sub.mod");
var subDef = new IloOplModelDefinition(subSource);
var subCplex = new IloCplex();
```

Solving the master model

The master model is contained in the following variables:

```
   var masterDef = thisOplModel.modelDefinition;
   var masterCplex = cplex;
   var masterData = thisOplModel.dataElements;

   // Creating the master-model
   var masterOpl = new IloOplModel(masterDef, masterCplex);
   masterOpl.addDataSource(masterData);
```

We reuse the `thisOplModel` variable because the master model corresponds to the definition contained in the same file as the flow control script. At each iteration, a new `IloOplModel` instance is created from the newly modified data elements.

Before you can solve, you need to generate the optimization model by calling:

```
thisOplModel.generate()
```

To solve the master model, call:

```
      if ( masterCplex.solve() ) {
        masterOpl.postProcess();
        curr = masterCplex.getObjValue();
        writeln();
        writeln("MASTER OBJECTIVE: ",curr);
      } else {
         writeln("No solution to master problem!");
         masterOpl.end();
         break;
      }
```

Updating the submodel

You need to update the reduced cost objective by setting the new dual values in the submodel data.

Here are the steps:

1. Make the changes to the data elements taken from the initial submodel:

   ```
   var subData = new IloOplDataElements();
   subData.RollWidth = masterOpl.RollWidth;
   subData.Size = masterOpl.Size;
   subData.Duals = masterOpl.Duals;
   ```

   It is easy to change the data to the new dual values from the variables of the master model. To do so, you can use the “dual” property of those variables.

   ```
   for(var i in masterOpl.Items) {
      subData.Duals[i] = masterOpl.ctFill[i].dual;
   }
   ```

2. Create a new submodel with the same definition file and the newly modified data elements:

   ```
   subOpl = new IloOplModel(subDef,subCplex);
   subOpl.addDataSource(subData);
   ```

3. Generate the optimization model:

   ```
   subOpl.generate();
   ```

Solving the submodel

1. Write the `writeln` statement:
2. Check the objective:

```
if (subCplex.getObjValue() > -RC_EPS) {
  break;
}
```

If the objective is not favorable, you stop the process. Remember that the objective represents the reduced cost of the new candidate pattern.

Updating the master model

If a solution has been found in the submodel, then a new pattern can be added to the master model. That pattern is represented by the values of the Use variable arrays in the submodel. In the master model, the patterns are represented by the `Patterns` tuple set. Therefore, you need to move the solution values of the Use variables from the submodel to a new tuple in the `Patterns` tuple set form the master model.

1. Modify the data elements obtained from the current master model.

```
      var masterData = thisOplModel.dataElements;
```

2. Simply add a new tuple in the `Patterns` tuple set using the array of values `Use` from the submodel.

```
      masterData.Patterns.add(masterData.Patterns.size,1,subOpl.Use.
solutionValue);
```

Preparing the new master model and regenerating its optimization model

Using these modified data elements, create a new master problem and generate its optimization model.

```
masterOpl = new IloOplModel(masterDef,masterCplex);
masterOpl.addDataSource(masterData);
masterOpl.generate();
```

You can see the complete version of this model in the file `cutstock_main.mod`.

Ending objects

The `cutstock_main` example also shows how to end the different script elements. Although memory leaks are not so much of a concern in this small example, it is good practice to use the `end` method of the class `IloOplModel` to systematically terminate objects that are no longer necessary. See `IloOplModel. end` in the *IBM ILOG Script Reference Manual* for details.

The `end` and `endAll` methods are disabled by default in the OPL IDE and can be enabled by setting `mainEndEnabled` to true. At the beginning of your script, add:

```
thisOplModel.settings.mainEndEnabled = true;
```

We recommend that you use caution in applying this setting. When it is enabled, you must ensure that memory is properly managed by your script. Faulty memory management, such as attempting to use an object after it has been deleted, may result in crashes.

Note: The `endAll` method is deprecated in OPL 6.2 and will not be supported in the next release.

Doing more with cutstock_main

Integer solution

The model presented in this tutorial only solves the relaxed problem, which is obviously not realistic. Even if this does not ensure an optimal solution, the usual technique consists in solving the integer version of the problem when all the new patterns are generated. This is done in another version of the model: `cutstock_int_main.mod`.

In that version, the final solution is output as follows:

```
      masterOpl = new IloOplModel(masterDef,masterCplex);
      masterOpl.addDataSource(masterData);
      masterOpl.generate();
```

Executing postprocessing

When a model is used from flow control, the postprocessing part is only executed on demand.

In this cutting stock example, the following postprocessing elements are defined:

- a structure to keep a pattern along with a float value,
- a computed set to be filled with the patterns that are used in the solution and their values,
- and a script to print out this set.

Here are the corresponding code lines:

```
tuple r {
   pattern p;
   float cut;
};

{r} Result = {<p,Cut[p]> | p in Patterns : Cut[p] > 1e-3};
```

This postprocessing part is not executed by default. This is useful because there are frequent situations where you won't want the postprocessing instructions to be executed. This is the case here, in the cutting stock example, because of the intermediate iterations.

When you do want to execute postprocessing, call:

```
   masterOpl.postProcess();
```

Tutorial: Changing default behaviors in flow control

Describes how to achieve finer control on the execution of a CPLEX® model by using flow control scripts to change the default behavior.

In this section

What you are going to do
Includes where to find the files.

Setting an initial solution for the CPLEX engine
Uses the `warmstart` example and its model `warmstart.mod` to show how you can use the IBM® ILOG® Script extension class `IloOplCplexVectors` to set up an initial solution for CPLEX on a specific part of the model.

Setting preferences on the search for conflicts and relaxations
Uses the `conflictIterator` example and its model `conflictIterator.mod` to explain how to use the class `IloOplConflictIterator` to refine a conflict with user-defined preferences (not available for CP models).

What you are going to do

This tutorial covers:

- Setting an initial solution for the CPLEX® engine explains how to use the class `IloOplCplexVectors` to pass an initial solution to the CPLEX solving engine before executing the model.
- Setting preferences on the search for conflicts and relaxations explains how to use the class `IloOplConflictIterator` to refine a conflict with user-defined preferences.

Important: The search for conflicts and relaxations is not supported in CP models.

Where to find the files

In this tutorial, you will work with the `conflictIterator` and `warmstart` examples, which you can find at the following location:

`<OPL_dir>\examples\opl\conflictIterator`

`<OPL_dir>\examples\opl\warmstart`

where `<OPL_dir>` is your installation directory.

Setting an initial solution for the CPLEX engine

Uses the `warmstart` example and its model `warmstart.mod` to show how you can use the IBM® ILOG® Script extension class `IloOplCplexVectors` to set up an initial solution for CPLEX on a specific part of the model.

In this section

The warmstart model
Presents the variables and constraints of `warmstart.mod`.

Default behavior
Provides the code and the solution.

Setting the initial solution
Shows how to use a value array.

Conclusion
Concludes on setting an initial solution for the CPLEX engine.

The warmstart model

The `warmstart.mod` model file defines the following variables and constraints.

Variables

```
range r = 1..10;
dvar int+ x[r];
dvar int+ y[r];
```

Constraints

```
minimize
  sum( i in r ) x[i] + sum( j in r ) y[j];
subject to{
  ctSum:
    sum( i in r ) x[i] >= 10;
  forall( j in r )
    ctEqual:
      y[j] == j;
}
```

This model has a lot of different possible solutions with the same objective. The purpose of the example is to show that the solution returned by CPLEX® can be influenced by the initial solution you pass to the solving engine.

Default behavior

The following code from the flow control part of the model (the `main` block) shows what the default behavior would be:

```
  // Default behaviour
  writeln("Default Behaviour");
  var opl1 = new IloOplModel(def, cplex);
  opl1.generate();
  cplex.solve();
  writeln(opl1.printSolution());
```

The solution is:

```
Default Behaviour
x = [10 0 0 0 0 0 0 0 0 0];
y = [1 2 3 4 5 6 7 8 9 10];
```

CPLEX calculates the first variable from the array such as to satisfy the `sum` constraint.

Setting the initial solution

The class that enables you to pass an initial solution to the CPLEX® MIP algorithm is `IloOplCplexVectors`. You can control the part of the model to be set with an initial solution by attaching a pair of elements made up of a constraint array and a value array. The value array is defined in the model.

```
// The following array of values (defined as data) will be used as
// a starting solution to warm-start the CPLEX search.
float values[i in r] = (i==5)? 10 : 0;
```

The second part of the `main` block illustrates how to use it.

```
  // Setting initial solution
  writeln("Setting initial solution");
  var opl2 = new IloOplModel(def, cplex);
  opl2.generate();
  var vectors = new IloOplCplexVectors();
  // We attach the values (defined as data) as starting solution
  // for the variables x.
  vectors.attach(opl2.x,opl2.values);
  vectors.setVectors(cplex);
  cplex.solve();
  writeln(opl2.printSolution());
```

The solution is then:

```
Setting an initial solution
x = [0 0 0 0 10 0 0 0 0 0];
y = [1 2 3 4 5 6 7 8 9 10];
```

The CPLEX log reports:

```
MIP start values provide initial solution with objective 65.0000
```

Conclusion

By attaching pairs of constraint arrays and value arrays, you can determine both:

- to which part of the model an initial solution will be set: use the `setVectors` method;
- from which part of the model the solution will be saved: use the `getVectors` method.

The mechanism is the same; only the method signature is different. Moreover, you can apply the same mechanism to the CPLEX® basis using the `setBasisStatus` and `getBasisStatus` methods of the class `IloOplCplexBasis`..

Setting preferences on the search for conflicts and relaxations

Uses the `conflictIterator` example and its model `conflictIterator.mod` to explain how to use the class `IloOplConflictIterator` to refine a conflict with user-defined preferences (not available for CP models).

In this section

The conflictIterator model
Presents the variables and constraints of this infeasible model.

Default behavior
Shows the code from the flow control and the result.

Setting user-defined preferences
How to assign an ordering to members of a conflict.

Conclusion
Concludes on setting preferences on the search for conflicts and relaxations.

The conflictIterator model

This tutorial uses a simple infeasible model which defines the following variables and constraints.

Variables

```
range r = 1..10;
dvar int+ x[r] in 1..10;
```

Constraints

```
minimize sum(i in r) x[i];
subject to {
   ct: sum(i in r) x[i] >= 10;
   forall(i in r)
     cts: x[i] >= i+5;
}
```

This model is clearly infeasible. All constraints from `cts[6]` through `cts[10]` are infeasible.

Default behavior

The following code lines from the first part of the flow control (the `main` block) show the default behavior if you use the `IloOplConflictIterator` class without setting any user-defined preferences.

```
   // Default behavior
   writeln("Default Behavior");
   var opl1 = new IloOplModel(def, cplex);
   opl1.generate();
   writeln(opl1.printConflict());
   opl1.end();
```

The output is then:

```
Default Behaviour
cts[6] at 9:0-10:17 E:\opl\conflictIterator.mod
  is in conflict.
```

This result was to be expected since CPLEX® refined the solution to the first constraint in the declared order.

Setting user-defined preferences

You can assign preferences to members of a conflict. In most cases, there is no advantage to assigning unique preferences, but if you know something about your model that suggests assigning an ordering to certain members, you can do so.

Guidelines to your choice:

- A preference of -1 means that the member is to be absolutely excluded from the conflict.
- A preference of 0 (zero) means that the member is always to be included, and
- Preferences of positive value represent an ordering by which the conflict refiner will give preference to the members. A group with a higher preference is more likely to be included in the conflict. Preferences can thus help guide the refinement process toward a more desirable minimal conflict.

To set user-defined preferences:

1. Define an array of preferences in the model, as shown in this code:

```
// Preferences are stated as data of the opl model.
// prefs[i] will be used to represent the preference of seeing cts[i] in
 the conflict.
float prefs[i in r] = i;
```

   The value is smaller for higher values of `i` and the CPLEX conflict refinement algorithm gives precedence to lower values.

2. Pass these preferences to the iterator by attaching them to the array of affected constraints as shown in this code:

```
   // With user-defined preferences
   writeln("With user-defined preferences");
   var opl2 = new IloOplModel(def, cplex);
   opl2.generate();
   // We attach prefs (defined as data in the opl model) as preferences

   // for constraints cts for the conflict refinement.
   opl2.conflictIterator.attach(opl2.cts, opl2.prefs);
   writeln(opl2.printConflict());
   opl2.end();
```

The output is then:

```
With user-defined preferences
cts[10] at 9:0-10:17 E:\opl\conflictIterator.mod
  is in conflict.
```

Conclusion

By attaching pairs of constraint arrays and preference arrays, you can control the way in which the CPLEX® solving engine refines the conflict returned by the conflict iterator. You can apply the same mechanism to the CPLEX relaxation algorithm by using the `IloOplRelaxationIterator` class.

Searching for relaxation and conflicts

To search for relaxations and conflicts in an infeasible model, you can use the IDE as explained in *Relaxing infeasible models* in *IDE Tutorials* but you can also use the IBM® ILOG® Script methods `printRelaxation` and `printConflict` on the `IloOplModel` instance. For example:

```
dvar int x in 0..10;

subject to
{
  ct1: x<=4;
  ct2: x>=6;
}

main
{
  thisOplModel.generate();
  writeln(thisOplModel.printRelaxation());
  writeln(thisOplModel.printConflict())

}
```

gives out

```
ct1 at 11:7-12
    relax [-infinity,4] to [-infinity,6] value is 6

ct1 at 11:7-12
  is in conflict.
ct2 at 12:7-12
  is in conflict.
```

Using IBM ILOG Script in constraint programming

Explains how to use IBM® ILOG® Script statements to set parameters that control propagation and search and to define search phases.

In this section

Setting CP parameters
How to set a parameter value by adding script statements to the model.

Defining search phases
To define search phases in OPL, you can use only IBM® ILOG® Script statements. This section explains how.

Accessing solutions in postprocessing
Describes how to access the solution in postprocessing.

Setting CP parameters

The preferred way to set CP parameters is from the IDE settings editor. However, it is sometimes convenient to set a parameter value by adding script statements to the model.

The IBM® ILOG® Script syntax to change a CP parameter is:

```
cp.param.paramName = "paramvalue"
```

For example:

```
cp.param.DefaultInferenceLevel = "Low"
```

or (from the model `timetabling.mod`):

```
var p = cp.param;
p.logPeriod = 10000;
p.searchType = "DepthFirst";
p.timeLimit = 600;
```

See Constraint programming options in *Parameters and settings in OPL* for a detailed description of each parameter.

IBM ILOG Script CP parameters

Parameter	Possible Values	Default Value
AllDiffInferenceLevel	Default, Low, Basic, Medium, Extended	Default
AllMinDistanceInferenceLevel	Default, Low, Basic, Medium, Extended	Default
BranchLimit		2100000000
ChoicePointLimit		2100000000
ConstraintAggregation	On/Off	On
CountInferenceLevel	Default, Low, Basic, Medium, Extended	Default
CumulFunctionInferenceLevel	Low, Basic, Medium, Extended	Basic
DefaultInferenceLevel	Low, Basic, Medium, Extended	Basic
ElementInferenceLevel	Default, Low, Basic, Medium, Extended	Default
FailLimit		2100000000
IntervalSequenceInferenceLevel	Low, Basic, Medium, Extended	Basic
LogPeriod		1000
LogVerbosity	Quiet, Terse, Normal, Verbose	Normal
MultiPointNumberOfSearchPoints		30
NoOverlapInferenceLevel	Low, Basic, Medium, Extended	Basic
OptimalityTolerance		1e-15
PrecedenceInferenceLevel	Low, Basic, Medium, Extended	Basic
PropagationLog	Quiet, Terse, Normal, Verbose	Quiet
RandomSeed		0
RelativeOptimalityTolerance		0
RestartFailLimit		100
RestartGrowthFactor		1.05
SearchType	DepthFirst, Restart, MultiPoint	Restart
SolutionLimit		2100000000
StateFunctionInferenceLevel	Low, Basic, Medium, Extended	Basic
TimeLimit		Infinity (number in secon
Workers	1 to 4	1

Defining search phases

To define search phases in OPL, you can use only IBM® ILOG® Script statements. This section explains how.

In this section

What is a search phase?
Introduces the notion of a search phase.

Writing script statements to define search phases
The model `steelmill.mod` shows this feature.

Multiple search phases
Explain how to pass several search phases to the engine.

Specifying variable and value choosers
A search phase can contain a variable chooser and a value chooser.

Scheduling search phases
Describes the interval variable and sequence variable search phases available for scheduling.

What is a search phase?

In constraint programming, a search phase is a way to guide search types.

One method of tuning the search is to use a search type other than the default search type. You can do this either from the IDE settings editor or by changing a CP parameter from IBM® ILOG® Script, as described in *Setting CP parameters*.

Another way is to guide the search types with search phases. A search phase allows you to specify the order of the search moves and the order in which the values must be tested.

A search phase defines instantiation strategies to help the embedded CP Optimizer search algorithm.

A search phase is either mono-criterion or multi-criteria.

A mono-criterion search phase is composed of:

- either
  - an array of integers to instantiate (or fix), **and**
  - a variable chooser that defines how the next variable to instantiate is chosen, **and**
  - a value chooser that defines how values are chosen when variables are instantiated

```
var phase1 = f.searchPhase(x,
               f.selectLargest(f.varIndex(x)),
               f.selectLargest(f.explicitValueEval(values, varEval, 0)));
```

- or

  an array of integers to instantiate (or fix)

```
var phase1 = f.searchPhase (x);
```

- or
  - a variable chooser that defines how the next variable to instantiate is chosen, **and**
  - a value chooser that defines how values are chosen when variables are instantiated

```
var phase1 = f.searchPhase (f.selectLargest(f.varIndex(x)),
              f.selectLargest(f.explicitValueEval(values, varEval, 0)));
```

A multi-criteria search phase can have:

- either two different search phases, such as:

```
var multiPhaseVar = new Array(f.selectSmallest(f.domainSize()), f.
selectRandomVar());
var multiPhaseValue = new Array(f.selectSmallest(f.value()), f.
selectRandomValue());
var phase1 = f.searchPhase(multiPhaseVar, multiPhaseValue);
```

- or several decision variables, such as:

```
var phase1 = searchPhase(new Array(x[1],x[2]));
```

Writing script statements to define search phases

To define a search phase, you write a script statement after the declaration of decision variables and before the constraint block, as shown below.

Location of the search phase script statement (`steelmill.mod`)

```
dvar int where[1..nbOrders] in 1..nbSlabs;
dvar int load[1..nbSlabs] in 0..maxLoad;

execute{
   writeln("loss = ", loss);
   writeln("maxLoad = ", maxLoad);
   writeln("maxCap = ", maxCap);

}
execute {
  cp.param.LogPeriod = 50;
}
execute {
   var f = cp.factory;
   cp.setSearchPhases(f.searchPhase(where));
}
dexpr int totalLoss = sum(s in 1..nbSlabs) loss[load[s]];

minimize totalLoss;
subject to {
  packCt: pack(load, where, weight);
  forall(s in 1..nbSlabs)
    colorCt: sum (c in 1..nbColors) (or(o in 1..nbOrders : colors[o] == c)
(where[o] == s)) <= 2;
}
```

Multiple search phases

You can pass several search phases to an instance of the CP Optimizer engine. The order of the search phases in the array is significant. The search engine instantiates the variables phase by phase, starting with the first one. It is not necessary that the variables in the search phases cover all the variables of the problem. It can be assumed that a search phase containing all the problem variables is implicitly added to the search phases. For instance, in a model that has three arrays of variables `x`, `y` and `z`, the following search phases:

```
int nbCars=
range Slots =
int values[i in 0..nbCars] = i;
int valueEval[i in 0..nbCars] =

dvar int slot[Slots] in 0..nbCars;

execute {
   var f = cp.factory;
  cp.setSearchPhases(f.searchPhase(x), f.searchPhase(y));
}
```

mean that variables `x` will be instantiated before variables `y`; then, once `x` and `y` are instantiated, variables `z` will be instantiated. In some particularly well-designed models, passing such an order can have a dramatic impact on the solving time.

Specifying variable and value choosers

A search phase can also contain a variable chooser and a value chooser. The example below shows how such a search phase could be defined for the `carseq` example. The full code of the `carseq.mod` model (which does not contain this search) is available at:

```
<OPL_dir>\examples\opl\carseq\carseq.mod
```

where `OPL_dir` is your installation directory.

Variable and Value Choosers in Search Phases (alternative search for `carseq.mod`)

```
execute {
   var f = cp.factory;

   var phase1 = f.searchPhase(slot,
               f.selectSmallest(f.varIndex(slot)),
               f.selectLargest(f.explicitValueEval(values, valueEval, 0)));


   cp.setSearchPhases(phase1);

}
```

List of variable choosers

```
selectSmallest( eval )

selectLargest( eval )

selectRandomVar()
```

List of variable evaluators

```
cp.factory.domainSize()

cp.factory.domainMin()

cp.factory.domainMax()

cp.factory.regretOnMin()

cp.factory.regretOnMax()

cp.factory.successRate()

cp.factory.impact()

cp.factory.localImpact()

cp.factory.impactOfLastBranch()

cp.factory.explicitVarEval(dvar int[],int[])

cp.factory.varIndex(dvar int[])
```

List of value choosers

```
selectSmallest( eval )
```

```
selectLargest( eval )
```

```
selectRandomValue()
```

List of value evaluators

```
cp.factory.value()
```

```
cp.factory.valueImpact()
```

```
cp.factory.valueSuccessRate()
```

```
cp.factory.explicitValueEval(int[],int[])
```

```
cp.factory.valueIndex(int[])
```

Scheduling search phases

Two types of search phases are available for scheduling: Search phases on interval variables and search phases on sequence variables.

Interval variable search phase

A search phase on interval variables works either on a unique interval variable or on an array of interval variables. During this phase, CP Optimizer fixes the value of the specified interval variable(s): Each interval variable will be assigned a presence status and for each present interval, a start and an end value. This search phase fixes the start and end values of interval variables in an unidirectional manner, starting to fix first the intervals that will be assigned a small start or end value.

The syntax:

```
searchPhase(a);

searchPhase(A);
```

Where:

```
dvar interval a;

dvar interval A[];
```

For instance, this code sample will specify a search that first fixes all interval variables in array A1 before the ones in array A2:

```
dvar interval A1[...] ...;
dvar interval A2[...] ...;

execute{
  var f = cp.factory;
  cp.setSearchPhases(f.searchPhase(A1),
  f.searchPhase(A2));
}
```

Sequence variable search phase

A search phase on sequence variables works on a unique sequence variable or on an array of sequence variables. During this phase CP Optimizer fixes the value of the specified sequence variable(s): Each sequence variable will be assigned a totally ordered sequence of present interval variables. Note that this search phase also fixes the presence statuses of the intervals involved in the sequence variables. This phase does not fix the start and end values of interval variables.

It is recommended to use this search phase only if the possible range for start and end values of all interval variables is limited (for example by some known horizon that limits their maximal values).

```
searchPhase(p);

searchPhase(P);
```

Where:

```
dvar sequence p;
dvar sequence P[];
```

Accessing solutions in postprocessing

You can access information about intervals, sequences, and instances of cumulFunction or stateFunction in a solution.

The value of an **interval variable** *a* in a solution can be accessed using the following instructions:

1) *a.present* returns a boolean describing whether or not interval *a* is present in the solution.

2) for a present interval, *a.start*, *a.end* and *a.size* respectively return the start, end and size value of interval *a* in the solution.

An interval variable *a* in a solution can be displayed using *writeln(a)*. This instruction displays a vector: < *a.present a.start a.end a.size* >.

The value of a **sequence variable** *p* in a solution can be accessed using the following instructions:

1) *p.first()* and *p.last()* respectively return the interval variable that corresponds to the first (resp. last) interval of the sequence. In case the sequence is empty the returned value is null.

2) *p.next(a)* and *p.prev(a)* respectively return the interval variable sequenced just after *a* (resp. just before *a*) in the sequence. In case *a* is the last (resp. first) interval in the sequence, the returned value is null.

A sequence variable *p* in a solution can be displayed using *writeln(p)*. This instruction displays the set of present interval variables in the sequence following the total order specified by the sequence.

In postprocessing, you can also access values of a **cumulative function** or a **state function** in a solution with the following OPL functions:

cumulFunctionValue

stateFunctionValue

segmentValue

numberOfSegments

segmentStart

segmentEnd

Advanced features

A tutorial on external functions.

In this section

Tutorial: External functions
Exposes the purpose and the context of external functions in OPL, and explains how to use an external knapsack algorithm, how to use data other sources, and how to debug custom Java code using Eclipse.

Tutorial: External functions

Exposes the purpose and the context of external functions in OPL, and explains how to use an external knapsack algorithm, how to use data other sources, and how to debug custom Java code using Eclipse.

In this section

Context of external functions
Specifies the purpose of external functions and the environment prerequisites.

Using an external knapsack algorithm
Presents the problem, the code samples, the location of the files, a summary of the procedure, and the detailed steps.

Using data from other sources
Shows how to use calls to external functions to define two customer-specific ways of feeding data to an OPL model: by using a subclass of `IloCustomOplDataSource` and by using a script function from a .dat file.

Debugging custom Java code using Eclipse
Explains how to use the popular Eclipse IDE to debug your code when calling external Java code from OPL script statements.

Context of external functions

Specifies the purpose of external functions and the environment prerequisites.

In this section

Limitations of the language
Explains how limitations of OPL lead to developing external functions.

Environment prerequisites
Lists what software you need to use external functions.

Limitations of the language

The purpose of external functions is to provide alternatives to some limitations of OPL as a language.

Note:
1. For external Java function calls to work on AIX platforms, you must set the LIBPATH variable to point to the libjava.a and libjvm.a libraries.
2. Calling external functions is not possible if the application is statically linked with the OPL libraries (like oplrunsample). This is because the Java OPL library uses the shared library version of OPL (oplxx.dll/.so) which cannot be mixed with the static version. You can use an external call to Java in the OPL IDE, in oplrun, and in custom OPL or ODM applications launched from Java.

The Optimization Programming Language (OPL) is a powerful language to write optimization models. It offers powerful aggregate constructs and slicing filters to describe complex problems in a compact form. Moreover, because it supports JavaScript, OPL enables you to manipulate complex data (preprocessing), results (postprocessing), and models (flow control). However, IBM® ILOG® Script is not as powerful, complete, and efficient as a programming language. As its purpose is to be simple and accessible for nonprogammers, more complex software engineering parts are not supported and should be kept outside of OPL.

However, OPL offers a way to interact with external code written in other programming languages. This external functionality can be plugged into OPL in an easy to use and reuse manner.

This feature allows you to:

- write a complex dedicated OR algorithm (such as a shortest path or flow algorithm) to be used as one step of a decomposed application
- connect your OPL model or data to other external tools such as tools for statistical analysis, to modify your input data or report your results
- connect your OPL data to other sources or destinations that are currently not supported as default sources by the language
- connect CPLEX® callbacks to your search (not described in this tutorial)

All this is possible by calling external functions from OPL. In OPL 5.1 and later, you can call functions written in the Java programming language. Note that the Java language itself also offers ways to interact with other programming languages.

Environment prerequisites

To call Java code, you need to have a Java Runtime Environment installed on your machine. Once it is installed, the JRE is automatically detected at runtime (see the function `IloOplImportJava` for details). OPL supports versions 5.0 and above of IBM JDK or JRE.

On Unix, the `LD_LIBRARY_PATH` must also contain the path to the shared libraries of the Java Virtual Machine. For example, `/jre/lib/i386` and `/jre/lib/i386/client` for Linux.

See also Working Environment.

Using an external knapsack algorithm

Presents the problem, the code samples, the location of the files, a summary of the procedure, and the detailed steps.

In this section

The problem
Gives a summary of the cutstock problem used in this tutorial.

The code samples
Presents the code samples used in this tutorial.

Where to find the files
Reminds you of where code sample files are located.

Procedure summary
Lists the main steps of the knapsack algorithm tutorial.

Writing the Java code
Lists the public methods of the Java knapsack algorithm you can use.

Using the Java code from OPL
Explains how to import Java classes and how to call Java from OPL.

Using IBM ILOG Script classes to make clean and reusable code
Explains how to use IBM® ILOG® Script classes to wrap the calls to external functions.

The problem

This tutorial reuses the same problem and data as the `cutstock_main` example described in *The cutting stock problem*. The main problem consists of cutting big wooden boards into small shelves to meet customer demands while minimizing the number of boards used. The subproblem consists of finding the best new pattern to cut the roll. This is a simple knapsack problem. To solve it, you are going to use a dedicated algorithm implemented in Java (instead of CPLEX®).

The code samples

This tutorial shows how to use external functions and work with a run configuration of the `cutstock` example and with the `externaldataread` example.

You will work with two code samples:

- In *Using an external knapsack algorithm*, the `cutstock_ext_main` example shows how to call a knapsack dynamic programming algorithm written in Java from an OPL main column generation script. This is an extension of the distributed example `cutstock_main`, which uses CPLEX® to solve the subproblem consisting of a simple knapsack constraint. For this type of constraint, some powerful specific algorithms exist that have a polynomial complexity. The same kind of mechanism could be used to solve shortest-path problems, flow in graph problems, or any other subproblem in which dedicated efficient algorithms can be implemented in Java.
- In *Using data from other sources*, the `externaldataread` example illustrates how to use calls to external functions to make other sources of data available from OPL.

  The example shows:

  - how to use instances of the class `IloCustomDataSource` from the IDE or from `oplrun`,
  - how some part of the data can be read from a file that uses any syntax (if you can read that file from Java).

Where to find the files

These files are at the following location:

`<OPL_dir>\examples\opl_interfaces\java\externaldataread`

where `<OPL_dir>` is your installation directory.

You will also use this file `<OPL_dir>\examples\opl_interfaces\java\javaknapsack\src\javaknapsack\Knapsack.java`.

Procedure summary

The tutorial assumes that you can write and compile Java code and that you know how to work with projects in the IDE. If this is not the case, read *Getting Started with the IDE* first.

To use the external knapsack algorithm:

1. Write the Java code.
2. Use the Java code from OPL.
3. Use IBM ILOG Script classes to make clean and reusable code.

Writing the Java code

The Java algorithm can be found in

```
<OPL_dir>\examples\opl_interfaces\java\javaknapsack\src\javaknapsack\
Knapsack.java
```

Note:
1. It is not the purpose of this tutorial to describe the knapsack algorithm.
2. Be aware that this implementation of the algorithm might not be optimal as the purpose in this tutorial is to keep it small and simple to read and understand.

The interesting point is that some methods are public and therefore candidates for being called from OPL.

These methods, written as simple OPL Java code, are:

- `public Knapsack()`: This constructor creates an empty knapsack instance that can be reused with different data.
- `public void updateInputs (IloOplElement oplWeights, IloOplElement oplValues)`: This method allows you to update data from OPL elements and pass it to the algorithm, using arrays of weights and values.
- `public double solve (IloOplElement oplSolution, int size)`: This method runs the algorithm and puts back the solution into the given solution array.

You need to compile the Java source code using `Run.bat`.

For more information on how to use OPL APIs from Java, please refer to the *Interfaces User's Manual* and the *Java API OPL Reference Manual*.

Using the Java code from OPL

Two IBM® ILOG® Script functions enable you to call Java external functions from OPL models in IBM ILOG Script statements.

The successive steps are:

1. Importing Java classes
2. Calling Java

Then, learn more about *Translation of parameters and results* and *Creation of the Java Virtual Machine (JVM)*.

Importing Java classes

The function

```
IloOplImportJava(<directory or path to jar file>);
```

imports the classes into the given directory or JAR file in IBM ILOG Script, so that they can be called. The path can be either absolute or relative to the directory of the model file.

Calling Java

The function

```
<Script object or Java object>=IloOplCallJava(<class name> or <Java
object>,<method name>,[<method signature> or ""],[<parameters>,...]);
```

enables you to call static methods, constructors, and instance methods.

The method signature is only needed when there is an ambiguity (method overloading), that is when several methods have the same name but different signatures. It is a string with the JNI signature, something like:

```
"(Lilog/opl/IloOplModel;ILjava/lang/String;)V"
```

for a method taking an `IloOplModel` instance and a String as parameters.

Therefore, you can call:

```
// static method:
var result=IloOplCallJava("mypackage.MyClass","myStaticMethod","",15);

// create instance:
var myObject=IloOplCallJava("mypackage.MyClass","<init>","","init param");

// call method on instance (two syntaxes are possible):
var mySubObject=IloOplCallJava(myObject,"getSubObject","");
or
var mySubObject=myObject.getSubObject();
```

The classes are looked for in the JAR files or on the paths specified by the `IloOplImportJava` instance (see *Importing Java classes* above).

Translation of parameters and results

Both the parameters and the results of the call are translated from IBM® ILOG® Script to Java (and conversely) as needed.

The rules are the following:

- Simple data types (numbers, strings, Booleans) are translated back and forth.
  - A Java method taking a string can be called with an IBM ILOG Script string.
  - A Java method returning a string appears as returning an IBM ILOG Script string.
- Arrays are also translated back and forth between Java and IBM ILOG Script arrays.
- Some known types that have representations in both IBM ILOG Script and Java are also translated back and forth, so that:
  - A Java method taking an `IloOplModel` object can be called with an IBM ILOG Script model such as `thisOplModel`.
  - A Java method returning a custom Java data source appears as returning an IBM ILOG Script data source, which enables you to add it to the model using regular script statements.
- Unknown Java types (created by Java code) are represented as special `JavaRef` objects in IBM ILOG Script so that you can call any methods on them and pass them as parameters in subsequent calls.
  - A Java method returning a Java object of class `MyClass` appears as returning a special `JavaRef` object from IBM ILOG Script.
  - You can call methods on that `JavaRef` object (syntax: `myObject.myMethod()`), or pass it as a parameter to other Java calls (which will see a normal Java object of class `MyClass`).

Creation of the Java Virtual Machine (JVM)

When the calls are executed:

- Either there is a JVM running: this is the case for ODM applications, the IDE in OPL/ODM mode, custom Java applications. Then, the call is performed within the current JVM.
- Or there is no JVM running: this is the case for the IDE in pure OPL mode and for oplrun. Then, a new JVM is created.

The JVM is initialized at the first call. To create the JVM, the runtime process must find both the Java home and the OPL home to access the OPL `.jar` file. If ODM is installed, the process must also find the ODM home to access the ODM `.jar` files, so that the ODM scripting works. These environment variables are detected automatically. See the *Java API Reference Manual* for details.

If a new JVM is created, it receives the value of the environment variable `ODMS_JAVA_ARGS` as parameter, if this variable is defined. This variable is also already taken into account if the JVM is started by the IDE (IDE in OPL-ODM mode) or in an ODM application (ODM

Player). This enables you to customize the way in which the JVM is created, for example by adding more virtual memory, customizing the default classpath, and so on.

Deploying OPL models with external Java function calls on Linux

When deploying OPL models that make external Java function on Linux platforms, the following information might be useful:

1. The environment variable `JAVA_HOME` need to be defined. OPL will load a JVM from `JAVA_HOME` for the external Java function calls. When the models are solved by a Java Application, the JVM for the Java application and the JVM for the external Java functions calls must be the same version.
2. A JVM might have multiple modes, such as client, server, etc. When the models are solved by a Java Application, the JVM for the Java application and the JVM for the external Java functions calls must also be in the same mode. The default mode of JVM OPL chooses to load for external Java function calls will be printed to standard output when a model with external Java function calls is solved by `oplrun` (OPL 5.x) or `oplrunjava` (OPL 6.x).
3. If you want to use a different mode of JVM other than the default mode, you will need to use the environment variable `ODMS_JVM_LIBRARY_OVERRIDE` to override the default selection. The value of `ODMS_JVM_LIBRARY_OVERRIDE` should be the relative path of `libjvm.so` from the JRE root path. For example, defining `ODMS_JVM_LIBRARY_OVERRIDE` as `/lib/i386/server/libjvm.so` will cause OPL to load a 32-bit server JVM.

Using IBM ILOG Script classes to make clean and reusable code

As shown in this knapsack example, you can wrap the calls to external functions into user-defined IBM® ILOG® Script classes and methods. Then, it is easy to reuse this algorithm in different OPL models. Although this is standard JavaScript code, the example includes a few useful comments. The IBM ILOG Script wrapping functions are all located in a reusable `javaknapsack.mod` file.

To wrap the calls to external functions:

1. Use a function to declare the new algorithm class.

```
function Knapsack() {
  IloOplImportJava("../../java/javaknapsack/classes");

  this.object = IloOplCallJava("javaknapsack.Knapsack","<init>","");

  this.updateInputs = __Knapsack_updateInputs;
  this.solve = __Knapsack_solve;
};
```

   The content of the function is what the constructor will execute. Part of it is to register methods. Here is an example of implementing a method:

```
function __Knapsack_updateInputs(weights,values) {
   this.object.updateInputs(weights,values);
  // The call above is a shortcut as there is no risk of ambiguity.
  // In the general case, if several methods have the same name, you can
 use:
  //IloOplCallJava(this.object,"updateInputs", "(Lilog.opl.
IloOplElement;Lilog.opl.IloOplElement;)V", weights, values);
};
```

   Then, these new IBM ILOG Script class and methods are used in the modified cutstock algorithm, the model of which is the `cutstock_ext_main.mod` file.

2. Include the `javaknapsack.mod` file so that the IBM ILOG Script definition can be used.

```
include "javaknapsack.mod";
```

3. Create the knapsack algorithm.

```
   // Create a subproblem instance:
   var knapsack = new Knapsack();
```

4. Write instructions so that at each iteration, the data is updated, the algorithm called, and the solution retrieved.

```
      knapsack.updateInputs(masterOpl.Size, masterOpl.Duals);
     var solutionValue = knapsack.solve(masterOpl.NewPattern, masterOpl.
RollWidth);
```

5. Run the example.

The rest of the model is the same as the cutstock model that uses a CPLEX® algorithm to solve the submodel. See *The cutting stock problem*.

Using data from other sources

Shows how to use calls to external functions to define two customer-specific ways of feeding data to an OPL model: by using a subclass of `IloCustomOplDataSource` and by using a script function from a .dat file.

In this section

Subclassing IloCustomOplDataSource
Indicates what provided code sample to start from to subclass this class.

Using IBM ILOG Script in data files
Discusses the use of script statements in data files as a way to load data.

Subclassing IloCustomOplDataSource

The associated code sample is the `externaldataread` example.

The Java subclass of `IloCustomOplDataSource` is implemented at the following location:

```
<OPL_dir>\examples\opl_interfaces\java\externaldataread\src\externaldataread\
ExternalDataRead.java
```

This code sample contains standard Java OPL code, like the `Warehouse.java` example documented in *Working with OPL interfaces* of the *Interfaces User's Manual*. The difference in this example is that you will use the custom data source directly from the OPL model by attaching it from the script statement to an instance of `IloOplModel`. To do so, you will reuse the two functions described in *Calling Java* to call Java functions, as shown in the following code extract:

```
  IloOplImportJava("./classes");

  // Create a new model using this model definition and cplex.
  var opl = new IloOplModel(thisOplModel.modelDefinition,cplex);
  opl.addDataSource(new IloOplDataSource("externaldataread.dat"));

  // Create the custom data source.
  var customDataSource = IloOplCallJava("externaldataread.ExternalDataRead",

     "<init>", "(Lilog.opl.IloOplModel;)V", opl);

  // Pass it to the model (notice that you can do this from a script because
the custom data source
  // was converted to a script data source upon return of the Java call).
  opl.addDataSource(customDataSource);
```

Now the custom data source is attached to the OPL model. When the model is generated, using the `generate` method, the data will be filled from that custom data source. In this example, you can see this effect in the way element `a` is given a value. This element is defined as:

```
int a = ...;
```

and filled from the custom data source by means of the `customRead` method:

```
        public void customRead()
        {
                IloOplDataHandler handler = getDataHandler();
                handler.startElement("a");
                handler.addIntItem(1);
                handler.endElement();
        }
```

Using IBM ILOG Script in data files

Another custom way to load data is to use script statements in data files.

It is possible to declare some script functions in a `.dat` file using a `prepare{}` block. If you do so, at each initialization of an element, you can invoke one of these functions using the `invoke` keyword.

In the function, two properties are be defined:

- `name`: the name of the element being initialized.
- `element`: the element being initialized.

You can then use these constructs along with some Java external functions to do some custom reading of data. The example used in this tutorial reads in a file called `externaldatasource.txt` that uses a format where all the elements of a sets are separated with commas (','). For this case, a simple parser has been written (`SimpleTextReader.java`). It has mainly two public methods:

```
public SimpleTextReader (String fileName, String token)

public void fillOplElement(IloOplElement element) throws IOException
```

The parser is used as follows:

```
prepare {
   function read(element, name) {
      var customDataSource =
IloOplCallJava("externaldatasource.SimpleTextReader",
         "<init>", "(Ljava.lang.String;Ljava.lang.String;)V",
                     "C:/ILOG/OPL/examples/opl/externaldatasource.txt", ",");

      customDataSource.fillOplElement(element);
      return true;
   }
}

strings = {"val1"} invoke read;
```

Results

Running the example, you can see that:

- the OPL element `a` takes the value `1`
- the string set `strings` contains not only `val1` as defined in the `.dat` file, but also two more values added from the text file.

Debugging custom Java code using Eclipse

Explains how to use the popular Eclipse IDE to debug your code when calling external Java code from OPL script statements.

In this section

Procedure summary
Indicates the main steps of the debugging procedure.

Creating an Eclipse project
The steps to create the Eclipse project.

Creating a run configuration
The steps to create a run configuration.

Starting the OPL IDE
The steps to start the OPL IDE.

Setting breakpoints and debugging
The steps to set breakpoints and debug.

Procedure summary

To debug custom Java code using Eclipse:

1. Create an Eclipse project: You create an Eclipse project for your custom code.
2. Create a run configuration: You create an **Eclipse Remote Debug** run configuration for this project.
3. Start the OPL IDE with Java in debug mode.
4. Set breakpoints and debug.

   You set breakpoints, run the **Eclipse Debug** run configuration, then run your model as many times as necessary.

You can easily follow similar instructions in other IDEs. Read the documentation of your favorite IDE about remote debugging.

Creating an Eclipse project

To create the Eclipse project:

1. Choose **New Project>Java Project** to open the Eclipse wizard for Java project creation.
2. Give it the directory name where your Java source code is stored.

   If the project uses OPL Java APIs, it must reference the file `oplall.jar` from `<OPL_HOME>/lib` to be able to compile.

Creating a run configuration

To create the run configuration:

1. Choose **Run>Debug** to open the wizard for run configuration creation.
2. Add a new Run Configuration under **Remote Java Application**.

   Give it the same name as your project and keep the default settings (`socket attach`, `localhost`, `port 8000`).

Starting the OPL IDE

You start the IDE with Java in debug mode. You can also use the command line executable like this:

```
oplrun.exe -p
C:\\ILOG\\OPL<version_number\\examples\\java\\javaknapsack\\cutstock\\
cutstock_ext_main.mod
C:\\ILOG\\OPL<version_number\\examples\\java\\javaknapsack\\cutstock\\cutstock.
dat
```

To start the OPL IDE:

1. Define the environment variable.

   ```
   ODMS_JAVA_ARGS to '-Xdebug -
   Xrunjdwp:transport=dt_socket,server=y,address=8000'
   ```

   This specifies that the Java Virtual Machine launched by OPL will be in debug mode and will listen for debugger connections on port 8000 of the current machine.

2. Start the IDE (or `oplrun`) as usual.

   The IDE does not appear on the screen until a debugger connection is received.

   Upon invocation of the Java code, the IDE is suspended:

   - either immediately on startup if you have ODM installed: in this case the OPL IDE is started right from the start with Java support.
   - or when you actually run your project with Java code if you don't have ODM installed: in this case, the JVM is created only when `IloOplImportJava` or `IloOplCallJava` are first invoked (see the *Reference Manual* for details)

   If you have used the oplrun command, you see a message such as:

   ```
   <<< setup
   Listening for transport dt_socket at address: 8000
   ```

   For details, see *http://publib.boulder.ibm.com/infocenter/javasdk/tools/topic/com.ibm.java.doc.igaa/_1vg000156f385c9-11b26a8be3f-7fff_1001.html*.

Setting breakpoints and debugging

At this stage, you run a model in OPL IDE, and if any breakpoints are set in the associated Java project in Eclipse, the Java debugger stops at those lines in the code and you can then debug in Eclipse.

To set breakpoints and debug:

1. Set breakpoints as appropriate in the Eclipse project.
2. Connect to the IDE by running the **Eclipse Debug** run configuration you created in *Creating a run configuration*. The IDE connects to the JVM created and appears.
3. Run your model in OPL IDE.

   The Java Debugger remains connected to the OPL IDE until it is closed, so that you can run your project several times.

Performance and memory usage

Recommends practices that are known to improve the modeling and the solving time of your models and/or their ability to find good solutions.

In this section

Performance tips
Contains a checklist of modeling best practices.

Memory usage
Explains how OPL uses and allocates memory, and suggests actions to improve memory usage, mostly for data structures, object termination, engine parameters, and `oplrun`.

Performance tips

Here is a check list for quick reference:

- Use the profiler to detect `execute` blocks that run for a long time during the preprocessing phase. See *Profiling the execution of a model* in *IDE Tutorials*.
- If you observe that the execution of a model is slow because the `main` scripting block loads many engine instances or submodels, you can improve this by turning off the OPL Language option **Update charts and statistics in main**. See OPL language options in *IDE Reference*.
- In pre- or postprocessing script statements, do not initialize array elements to zero, OPL does that for you. See the note in *Initializing arrays* in the *Language User's Manual*.
- To initialize arrays, prefer generic index arrays rather than an `execute INITIALIZE` block. See *As generic indexed arrays* in the *Language Reference Manual*.
- Avoid dummy formal parameters for tuple components. See *Formal parameters* in the *Language Reference Manual*.
- Cache results for `find()` lookup.
- Calculate iteration sets for conditional blocks.
- Declare local script variables using the keyword `var`. See *Declaration of script variables* in the *Language Reference Manual*.
- In CP models with customized search strategies, consider the order of search phases. See *Multiple search phases* in the *Language User's Manual*.
- Constraint labels may have a significant performance and memory cost. See *Constraint labels* in the *Language Reference Manual*.
- To improve the performance of a model from the IDE, use the Tune Model button. See *Using the performance tuning tool* in *IDE Tutorials*.
- Using sorted versus ordered sets affects the memory consumption and the speed of execution but the effect is different depending on what operations are carried out on the sets. It is therefore not possible to give general recommendations on when to use sorted sets rather than ordered sets.
- Using slicing rather than `if` statements usually saves time and memory. For example, in the following code lines

```
int n=1000;

dvar int x[1..n][1..n];

subject to
{

ct1:forall(i in 1..n,j in 1..n:i==4 && j==5) x[i][j]==5;
ct2:forall(i in 1..n,j in 1..n) if (i==4) if (j==5) x[i][j]==5;
```

```
}
```

the `ct1` constraint is 60 times faster and lighter in memory than `ct2`.

To write efficient models, see also *Modeling tips* in the *Language User's Manual*.

To control memory consumption. See *Memory usage*.

Memory usage

Explains how OPL uses and allocates memory, and suggests actions to improve memory usage, mostly for data structures, object termination, engine parameters, and `oplrun`.

In this section

If your system runs out of memory
Indicates the various parts of IBM® ILOG® OPL that use memory.

Building data structures differently
Explains how to tune the way data structures inside a model are built.

Terminating data objects
Explains how to use the method `end` to free memory in IBM® ILOG® Script and interfaces.

Changing engine parameters
Explains how to change engine parameters to modify the way the engine solves a model.

Using oplrun
Explains how to reduce overhead by using `oplrun` in command line mode.

Changing to a 64-bit platform
If your model requires more than 2GB of memory.

Using 4GT tuning
To raise the addressable space from 2GB to 3GB.

Scaling down the size of the model
For the model to be solved more easily.

If your system runs out of memory

In IBM® ILOG® OPL, memory is used by several different modules:

- by the OPL IDE or the `oplrun` executable
- by the objects that each model declares
- by the optimization engine during the actual optimization process of an individual model
- if you are using OPL Interfaces, by the application that invokes the OPL objects

If you are running out of memory, it is important to determine which parts are using up the most memory. For instance, if memory is exhausted before the underlying instance of the engine has started, then you should evaluate the memory requirements of the earlier stages. System memory profilers (such as the Windows Task Manager or the Unix command `top`) give a very rough gauge to memory use but more accurate figures are provided by the OPL IDE Profiler (see *Profiling the execution of a model* in *IDE Tutorials*).

There are several ways to avoid hitting the "out of memory" message. Basically, you need to lower the amount of memory used by the model (by reformulation, parameters, or smaller size), or raise the amount of memory available on the system (by tuning or changing architectures).

Note: You can change the amount of memory allocated to the OPL IDE in the file `<OPL_dir>\oplide\oplide.ini`. See the topic *OPL IDE memory allocation* for a description.

Building data structures differently

A way of using less memory is to tune the way the data structures inside a model are built.

For instance, creating many intermediate arrays and sets that are not directly used in the model will increase memory. In addition, it is important to take advantage of proper modeling techniques; building a large, sparse, multi-dimensional array generally uses more memory than an equivalent set of tuples. Other treatments of sparsity and generic arrays are covered in greater detail in *Modeling tips* of the *Language User's Manual*. In addition, IDE users can also use the Profiler tab of the Output window to identify constructs that use large amounts of memory; an example is given in *Profiling the execution of a model* of *IDE Tutorials*.

See also the white paper "*Efficient Modeling in ILOG OPL-CPLEX Development System*" available from the IBM ILOG web site for IBM ILOG OPL at:*http://www.ilog.com/products/oplstudio/whitepapers/index.cfm*.

Terminating data objects

Use the method `end` to free memory in IBM® ILOG® Script and interfaces.

A complex model may manage separate submodels, data sources, run configurations, solvers, additional variables to move data between models, and so on. Accordingly, it is a good practice to systemically terminate these additional data objects when they are no longer needed. IBM ILOG Script for OPL provides the `end` method for the class `IloObject`, which is inherited by all other IBM ILOG Script classes. Similarly, the OPL Interfaces APIs have the `end` method for the class `IloExtractable`, which is also inherited by nearly all OPL Interfaces classes.

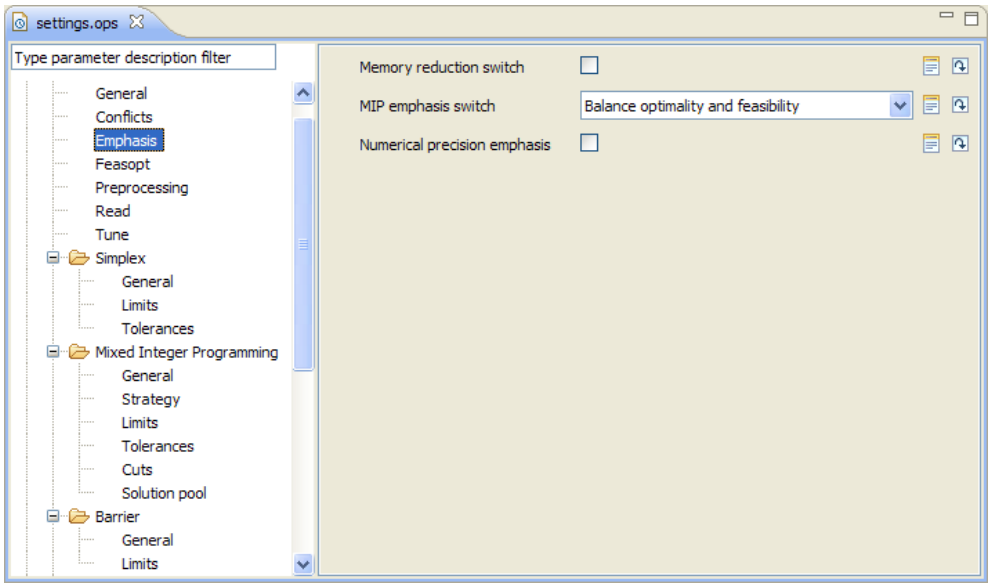
By using the method `end` to free objects, you greatly improve management of the total memory in use.

For more information and examples, please see the documentation for the classes `IloObject` or `IloExtractable`, and section *Ending objects* in the *Language User's Manual*.

Changing engine parameters

You can also change the way the engine solves models by changing engine parameters.

In some rare situations, you may want to instruct the engine to use less memory. To do so, set the `MemoryEmphasis` parameter to true. In the IDE, this parameter is in the **Mathematical Programming/Emphasis** page of the project settings panel.



The Memory Emphasis parameter in the IDE

Note: Changing this parameter value can help in tight memory situations. However, be aware that it may significantly increase the runtime requirements.

Using oplrun

If you are using the OPL IDE, consider working with oplrun.

The IDE is very rich and full-featured, but does impose a small, non-trivial overhead in terms of memory usage. If you need only a small amount in memory savings, and do not require the specific IDE features, consider switching to the command-line oplrun executable. It has somewhat lower memory requirements; the savings may help in some limited cases.

See oplrun Command Line Interface.

Changing to a 64-bit platform

If you are using 32-bit OPL, consider moving to a 64-bit architecture.

A 32-bit application typically has a maximum addressable space of 2GB or 4GB. In practice, this provides a maximum heap size of around 80-90% of the addressable space. If your OPL model runs out of heap space, further optimization will terminate, regardless of the total amount of memory available on the system. If your model requires more than 2GB of memory, consider moving to a 64-bit architecture, which has a substantially higher limit.

Using 4GT tuning

If you are using 32-bit OPL on Windows, consider using 4GT tuning.

In some Windows servers, it is possible to tweak the underlying kernel and applications to raise the addressable space from 2GB to 3GB. This requires several tweaks, but can also be useful in limited situations. This approach is further discussed in an FAQ which you can read from IBM ILOG web site at. *http://www.ilog.com/products/oplstudio*.

Scaling down the size of the model

Lastly, if none of these techniques are viable, then it may be that the model is too large to be solved easily on the target machine. For these remaining situations, you may need to make the model smaller, in terms of the data and constraints, in order to get it to run under the available memory.

Index

D