优化软件与应用

提醒：最后要提交一份打印版的作业（A4）

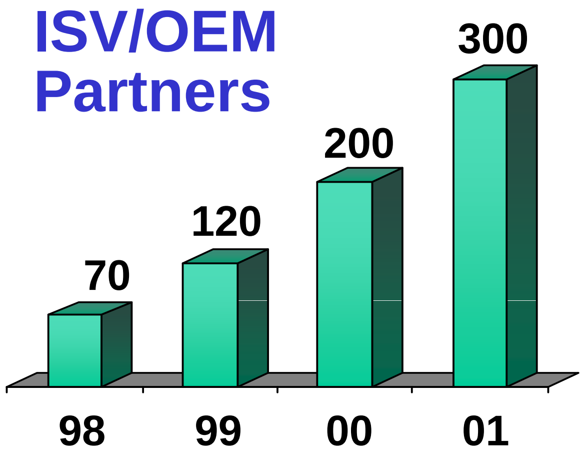
主讲人:　　雒兴刚
东北大学系统工程研究所
Email：luoxinggang@ise.neu.edu.cn
Tel: 83682292

信息科学与工程学院

第六章 ILOG OPL 基础

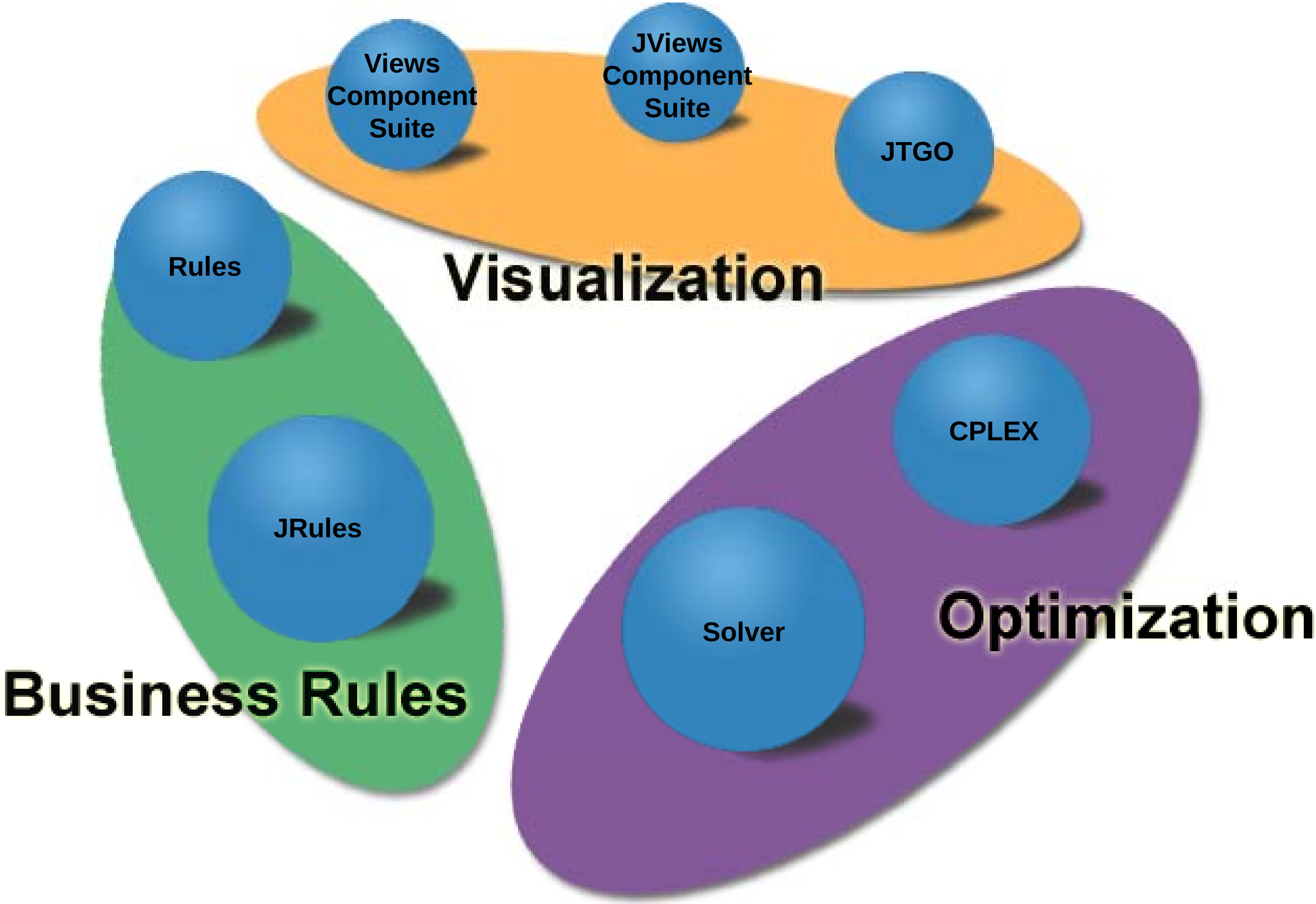
ILOG 简　介

- Founded 1987
- 590 employees
- 2,000+ customers
- Selling in 30 countries
- NASDAQ/Euronext



第六章 ILOG OPL 基础

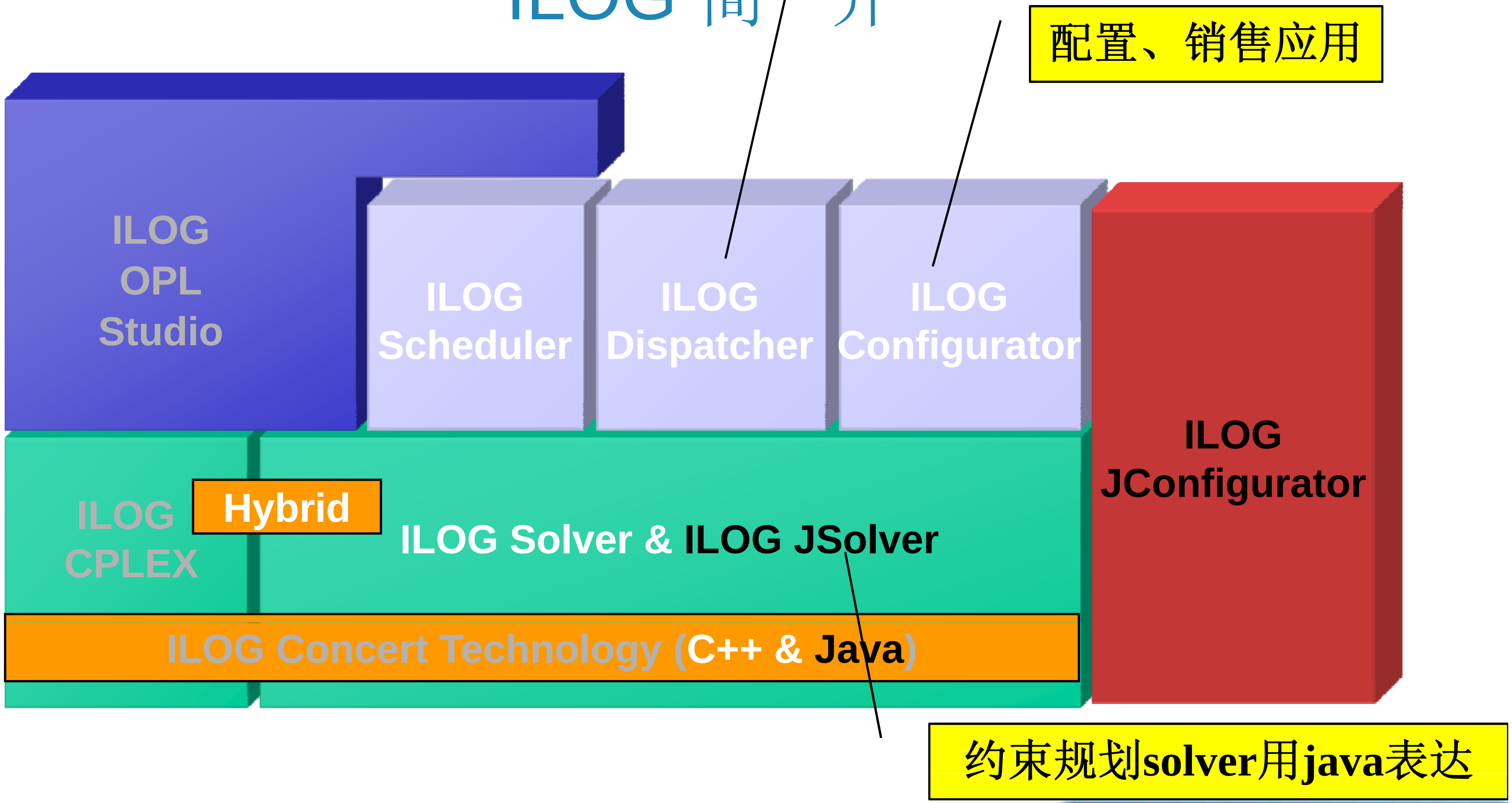
ILOG 简　介



第六章 ILOG OPL 基础

运输行业用

ILOG 简 介

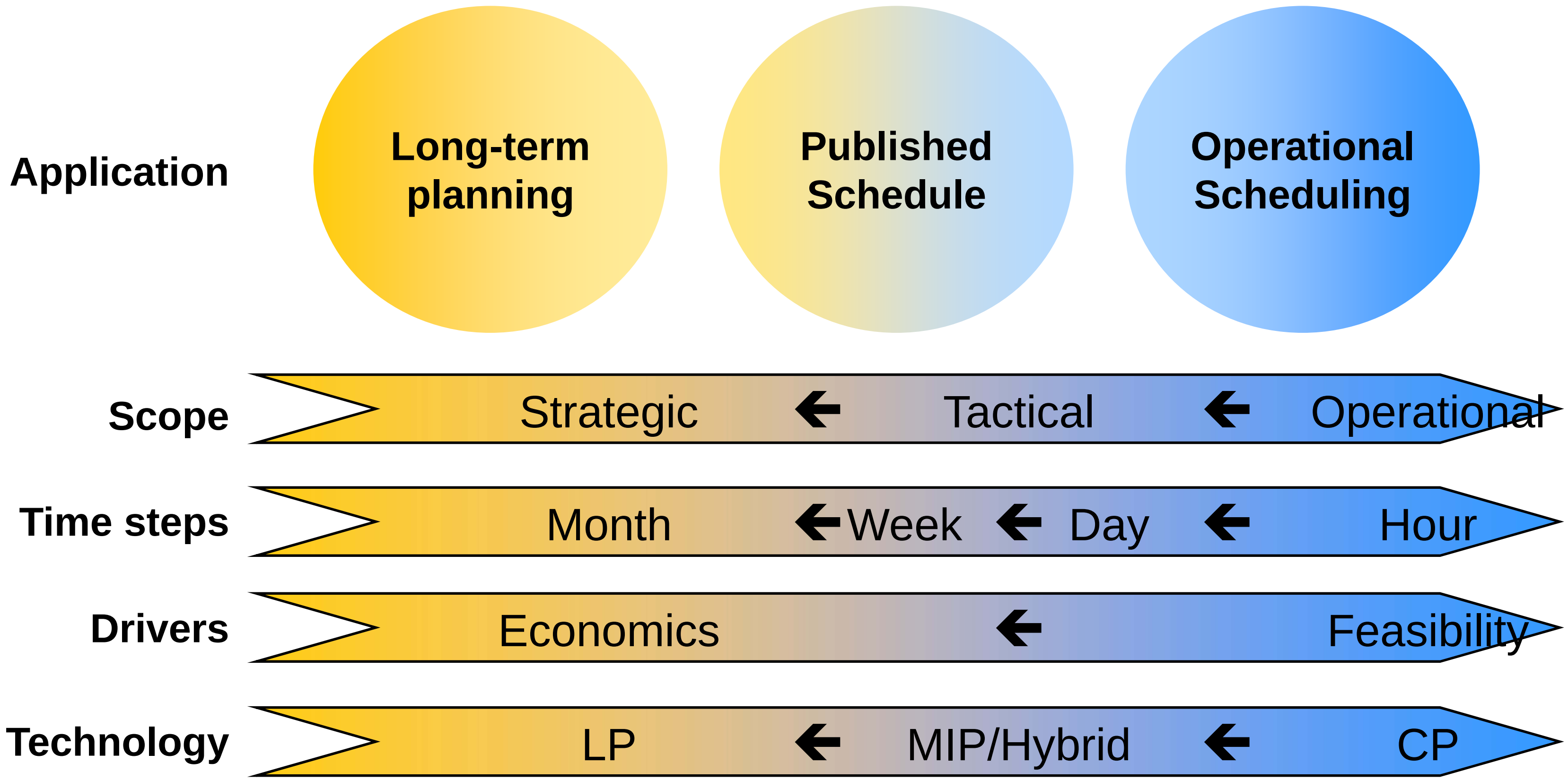


ILOG 简 介

- Core Engines
  - ILOG Solver - Constraint Programming Engine
  - ILOG CPLEX - Math Programming Engine
- Vertical Engine Extensions
  - ILOG Scheduler - Constraint-Based Scheduling
  - ILOG Dispatcher - Vehicle Routing, Technician Dispatching
  - ILOG Configurator - Product and Service Configuration
- Modeling Tools
  - OPL Studio - Rapid Development of Optimization Apps
  - AMPL - Modeling Support for CPLEX

We use OPL Studio here since its high level language makes it an easy starting point

Range of Optimization Applications



ILOG has optimization technology for the entire planning horizon

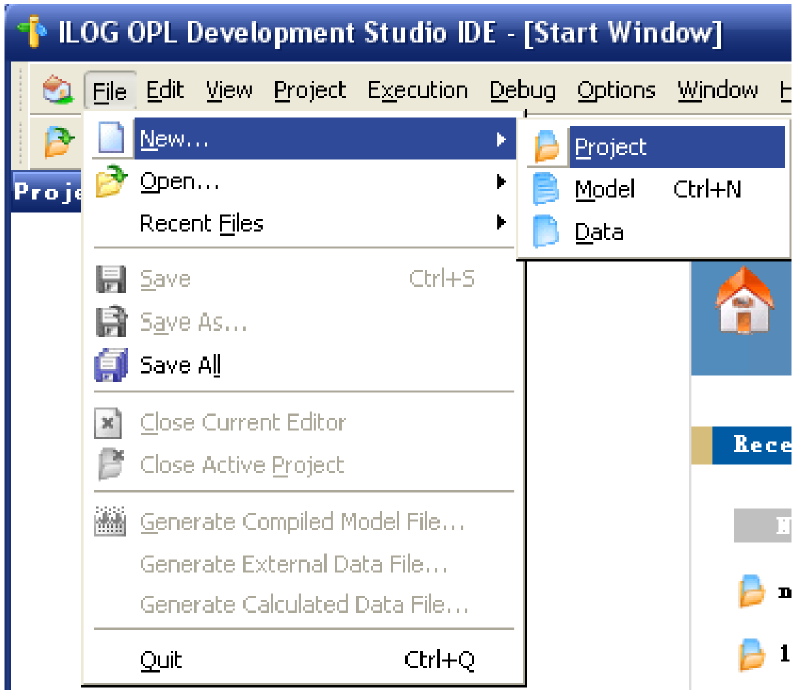
ILOG OPL 简明教程-(1-IDE简介)

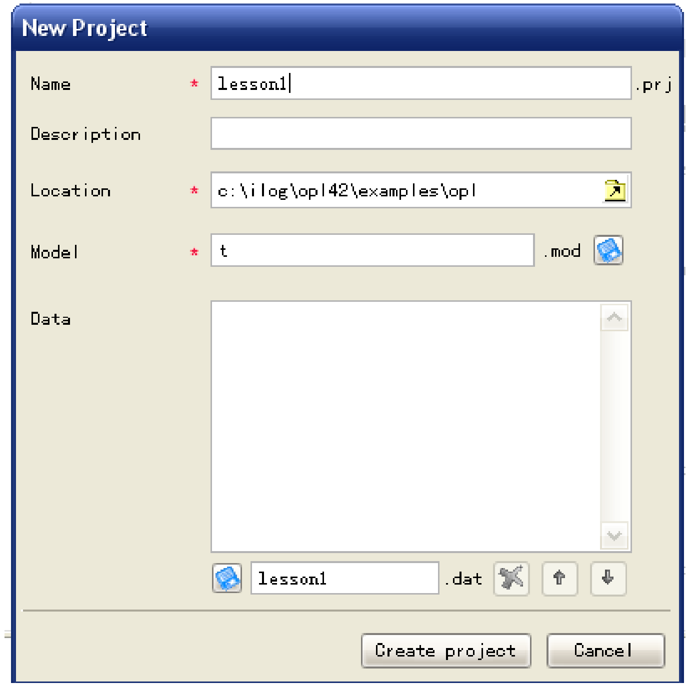
OPL IDE开发环境介绍

1) 打开OPL IDE.

ILOG OPL 简明教程-(1-IDE简介)

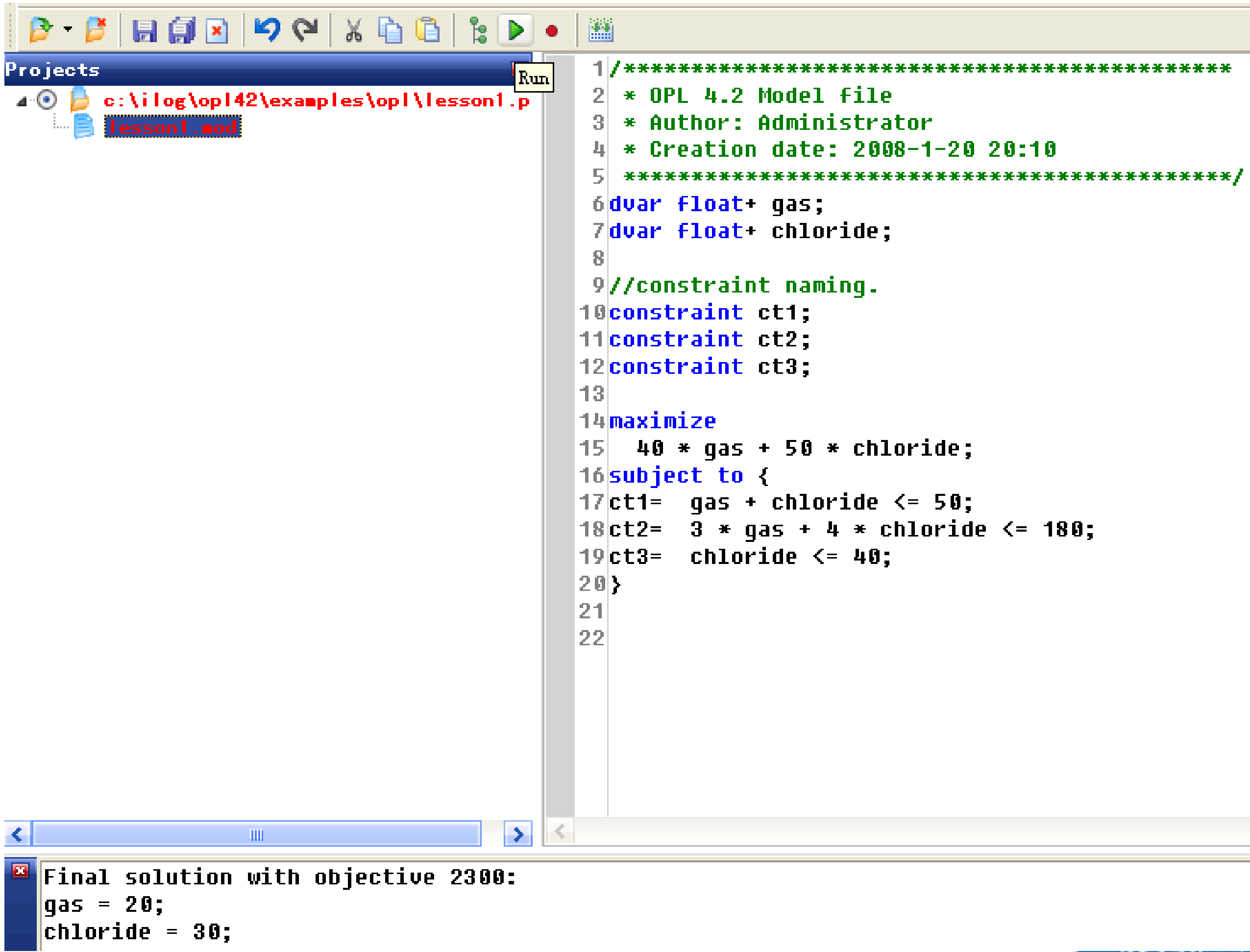
2) 在OPL IDE开发环境中有两种方式可以实现运行上面的代码：
1以建立工程的方式：





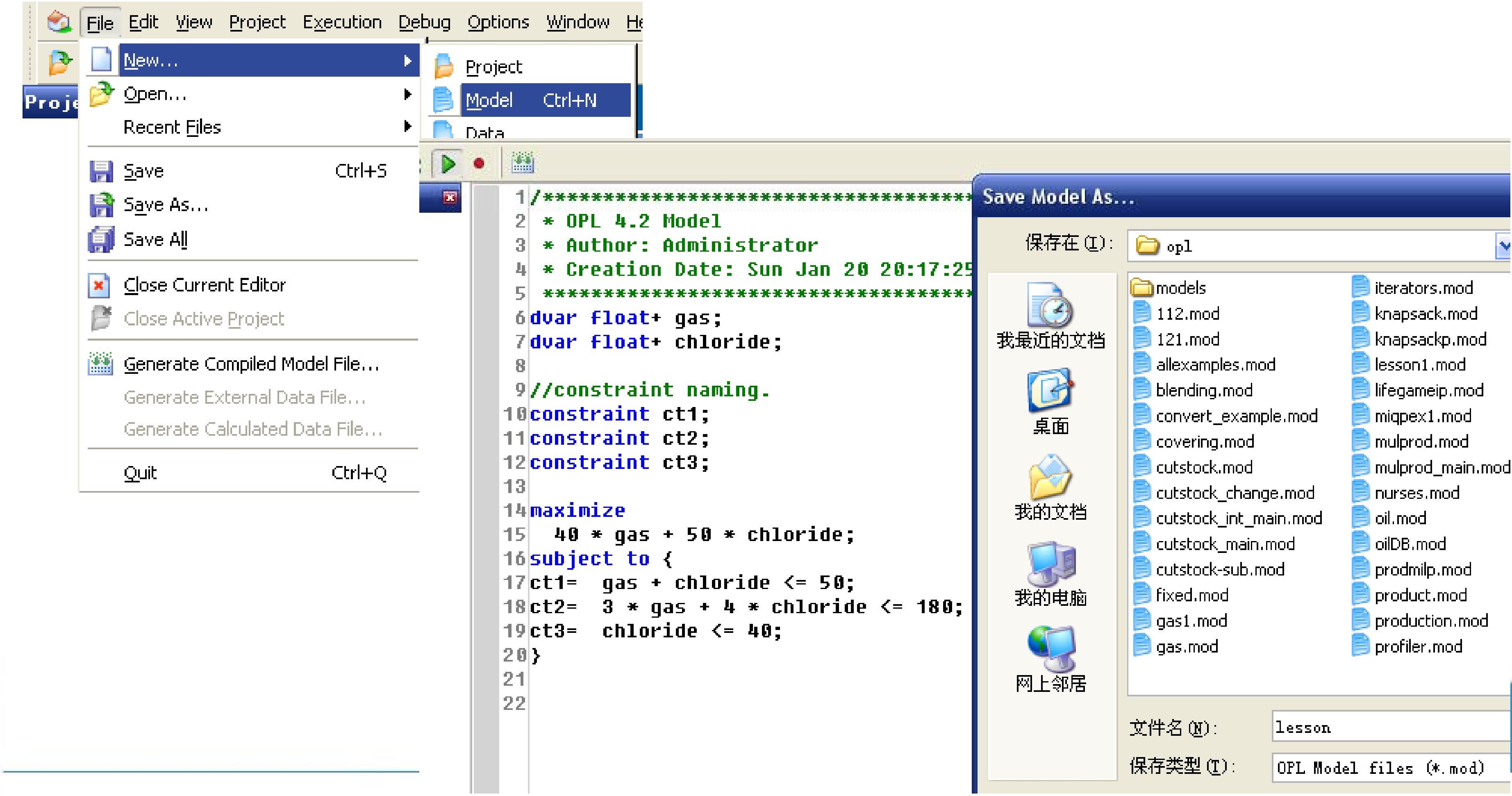
第六章 ILOG OPL 基础

ILOG OPL 简明教程-(1-IDE简介)



ILOG OPL 简明教程-(1-IDE简介)

2 以建立模型的方式:



ILOG OPL 简明教程-(1-IDE简介)

推荐使用以建立工程的方式进行开发，规范且方便以后开发。

Projects

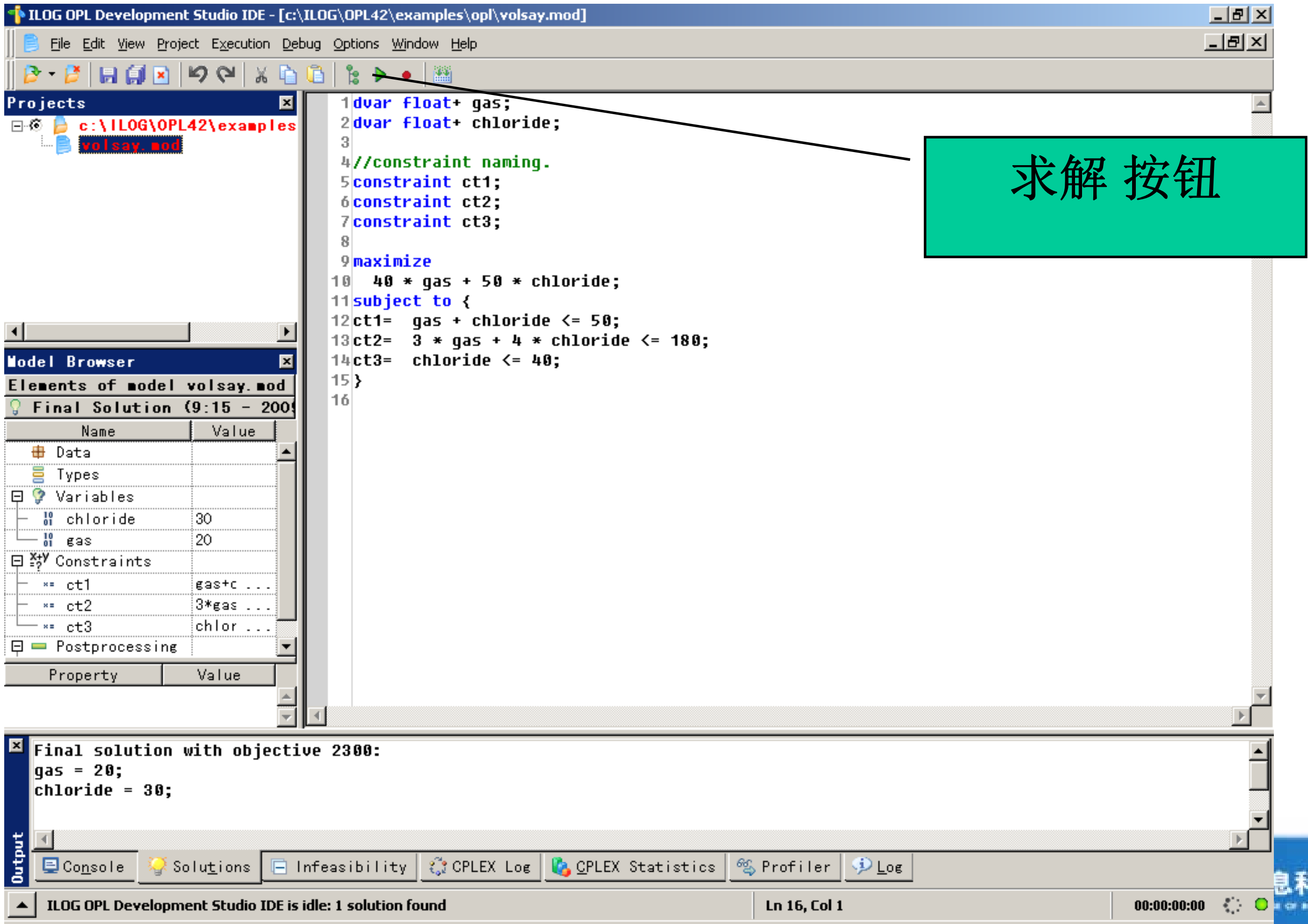
C:\ILOG\OPL42\examples\opl\lesson.mo

```
/*********************************************
 * OPL 4.2 Model
 * Author: Administrator
 * Creation Date: Sun Jan 20 20:17:25 2008
 *********************************************/
dvar float+ gas;
dvar float+ chloride;

//constraint naming.
constraint ct1;
constraint ct2;
constraint ct3;

maximize
  40 * gas + 50 * chloride;
subject to {
ct1=  gas + chloride <= 50;
ct2=  3 * gas + 4 * chloride <= 180;
ct3=  chloride <= 40;
}
```

```
Final solution with objective 2300:
gas = 20;
chloride = 30;
```

第六章 ILOG OPL 基础

ILOG OPL 简明教程-(1-IDE简介)



ILOG OPL 简明教程-(2-最简单的例子)

例：一个简单的线性规划问题

某公司生产氨气 (N H3) and 氯化铵 (N H4 Cl). 公司的日处理能力为50 单位的氮 (N), 180 单位的氢 (H), 40 单位氯 (Cl).氨气的利润是 40 euros每单位、 氯化铵的利润是50 euros 每单位. 如何确定氨气 和 氯化铵的产量，使利润最大

目标函数：max z=40*Gas+50*Chloride

满足约束条件：Gas+Chloride<=50

3*Gas+4*Chloride<=180

Chloride<=40

ILOG OPL 简明教程-(2-最简单的例子)

OPL IDE开发环境中对应编码：

```
dvar float+ gas;
dvar float+ chloride;
//constraint naming.
constraint ct1;
constraint ct2;
constraint ct3;
maximize
  40 * gas + 50 * chloride;
subject to {
ct1=  gas + chloride <= 50;
ct2=  3 * gas + 4 * chloride <= 180;
ct3=  chloride <= 40;
}
```

在OPL IDE开发环境中Console窗口的输出结果：

Final solution with objective 2300:
gas = 20;
chloride = 30;

注意：注释语句和C语言同，支持//和/* */

注意：**OPL**语言区分大小写！

说明: ILOG OPL 简明教程-(2-最简单的例子)

dvar,+,constraint,maximize,subject to都是什么含义?

dvar:(decision variable) 是OPL的关键字,放在前面讲过的“定义变量”之前,表示此定义的变量是决策变量。基本格式是:**dvar** 数据类型 变量名; 例如:**dvar float gas;**

+:一般放在前面讲过的定义的“决策变量”中的“基本数据类型”之后,表示所定义的决策变量是正数。基本格式是:数据类型+ 变量名; 例如:**dvar float+ gas;**//“+”只能在决策变量中使用.
? 有没有 **dvar float- gas;** 的用法?

constraint:是OPL的关键字,定义方式同“定义变量”,放在定义的约束变量名之前,表示此定义的变量是约束变量。基本格式:**constraint** 约束变量名;例如:**constraint ct1;**说明:例子中的程序在改写成不加入“约束变量”的情况后,仍然可以正常运行,在以后的例子中会发现有“约束变量”的程序要更健壮一些,所以推荐使用“约束变量”。

ILOG OPL 简明教程-(2-最简单的例子)

maximize： 是OPL的关键字，放在表达式之前，表示求此表达式的最大值。基本格式：**maximize 表达式;**例如：

maximize 40 * gas + 50 * chloride;

如果是最小化问题，则使用**minimize**

subject to： 是OPL的关键字，放在一组约束之前，是用于约束的另一种形式。基本格式：**subject to {一组约束};**例如：

```
subject to {
ct1=  gas + chloride <= 50;
ct2=  3 * gas + 4 * chloride <= 180;
ct3=  chloride <= 40;
}
```

ILOG OPL 简明教程-(2-最简单的例子)

将数学模型转化成**OPL**语言方法

数学模型中的目标函数：max z=40*Gas+50*Choride

⇕

OPL语言：maximize 40*Gas+50*Chloride;

ILOG OPL 简明教程-(2-最简单的例子)

将数学模型转化成**OPL**语言方法

数学模型中的约束条件：Gas+Chloride<=50
3*Gas+4*Chloride<=180
Chloride<=40

OPL语言：
```
subject to {
 ct1= gas*chloride<=50;
 ct2=3*gas+4*chloride<=180;
 ct3=chloride<=40;
 }
```

ILOG OPL 简明教程-(3-使用数组)

使用数组使得模型可读性好，而且容易扩展。通过使用数组，前面的例子可以表示为：

对比LINGO的集；
对比C的数组下标

```
{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;
}
```

具体解释参见下页

ILOG OPL 简明教程-(3-使用数组)

说明：

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

声明一组字串集合（**a set of strings**），表示公司的两个产品

dvar float production[Products];

声明一个决策变量数组，包含2个变量, **production[“gas”]** 和 **production["chloride"]**

ILOG OPL 简明教程-(3-使用数组)

注意，很多程序员会把前面的例子简化如下：

```
dvar float production[2];
maximize
        40 * production[1] + 50 * production[2];
subject to {
        production[1] + production[2] <= 50;
        3 * production[1] + 4 * production[2] <= 180;
        production[2] <= 40;
}
```

但是会导致编译出错。定义数组的语句中，数组元素个数不能像高级语言那样直接给出一个常量，而应该是一个范围(**Range**)。正确的写法是：

ILOG OPL 简明教程-(3-使用数组)

```
range kinds =1..2;
dvar float production[kinds];
maximize
    40 * production[1] + 50 * production[2];
subject to {
    production[1] + production[2] <= 50;
    3 * production[1] + 4 * production[2] <= 180;
    production[2] <= 40;
}
```

ILOG OPL 简明教程-(3-使用数组)

看看前面的模型代码：

```
{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;
}
```

可读性还是不好！ 数据直接嵌入到了程序中，不利于扩展

ILOG OPL 简明教程-(3-使用数组)

可将数据定义部分进一步修改为：

2种产品

3种成分

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

产品和成分之间的关系的数组，即每种产品需要的成分的数量。

```
float demand[Products][Components] = [ [1, 3, 0], [1, 4, 1] ];
float profit[Products] = [40, 50];
float stock[Components] = [50, 180, 40];

dvar float+ production[Products];
```

受益系数的数组

库存的数组

对比LINGO集的用法---
更接近于C习惯

ILOG OPL 简明教程-(3-使用数组)

那么，原来的目标函数

```
maximize
    40 * production["gas"] + 50 * production["chloride"];
```

就可以修改为：

收益系数

决策变量

```
maximize
    sum (p in Products) profit[p] * production[p];
```

函数，表明针对对每一个成员p，计算表达式profit[p] * production[p]的和。

对比LINGO用法：
MAX=@SUM(A(I):P(I)*X(I));

ILOG OPL 简明教程-(3-使用数组)

同理，原来的约束

```
subject to {
    production["gas"] + production["chloride"] <= 50;
    3 * production["gas"] + 4 * production["chloride"] <= 180;
    production["chloride"] <= 40;
}
```

Sum函数的用法见上页

函数，表明针对每种成分c（故共有3个约束），后面的sum表达式小于stock[c]必须满足

可以修改为：

```
constraint ct;
subject to {
 ct = forall (c in Components)
   sum (p in Products) demand[p][c] * production[p] <= stock[c];
}
```

对比LINGO用法：@FOR(WH(I):@SUM(VD(J):X(I,J))<=AI(I));

完整代码：ILOG OPL 简明教程-(3-使用数组)

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

float demand[Products][Components] = [ [1, 3, 0], [1, 4, 1] ];
float profit[Products] = [40, 50];
float stock[Components] = [50, 180, 40];

dvar float+ production[Products];

//constraint naming.
constraint ct;

maximize
  sum (p in Products) profit[p] * production[p];
subject to {
 ct = forall (c in Components)
    sum (p in Products) demand[p][c] * production[p] <= stock[c];
}
```

ILOG OPL 简明教程-(4-分离数据)

接上节。目前的文件中，数据和模型代码集中在一起，都保存在mod文件中。好的程序结构应该将数据和模型代码分离，保存在不同的文件中。

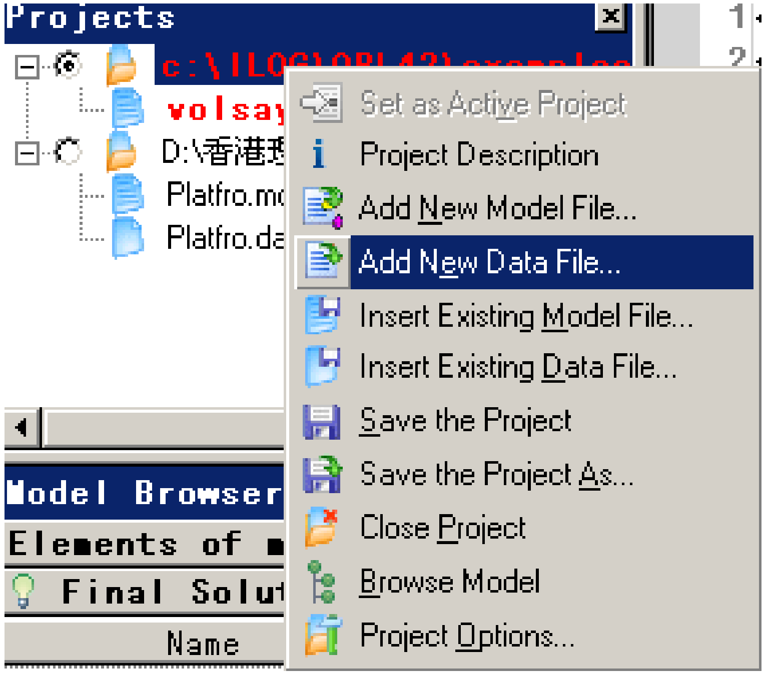
下面我们通过例子来说明在OPL中如何实现上述目的。

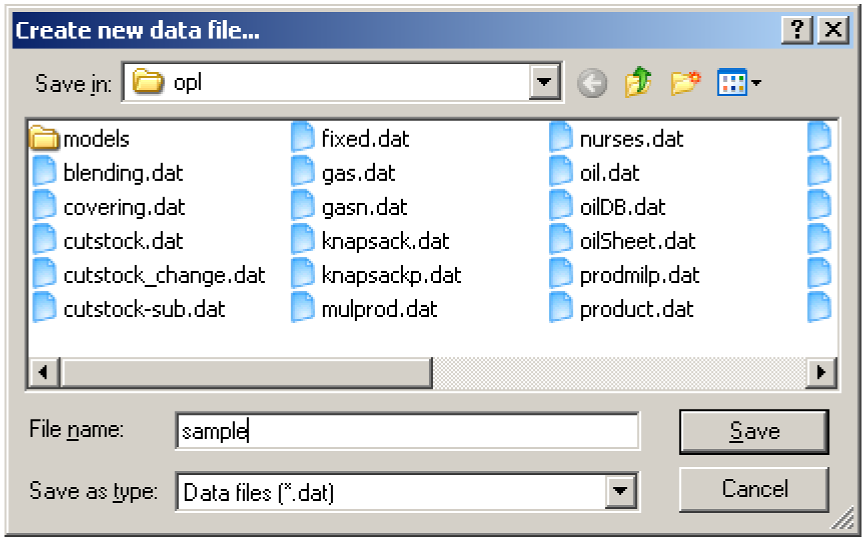
将前面代码中带有初始化数据的部分，全部换成3个点：

```
{string} Products = ...;
{string} Components = ...;
float demand[Products][Components] = ...;
float profit[Products] = ...;
float stock[Components] = ...;
```

ILOG OPL 简明教程-(4-分离数据)

如下所示，添加一个**sample.dat**文件到当前工程。





ILOG OPL 简明教程-(4-分离数据)

在数据文件中键入下面的内容：

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

profit = [40, 50];
stock = [50, 180, 40];
demand = [[1 3 0 ], [ 1 4 1] ];
```

运行程序，结果和前面的相同。

ILOG OPL 简明教程-(4-分离数据)

规范的完整写法：

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

profit = #["gas":40, "chloride":50]#;
stock = #["nitrogen":50, "hydrogen":180, "chlorine":40]#;
demand = #[
        "gas":     #[ "nitrogen":1 "hydrogen":3 "chlorine":0 ]#,
        "chloride": #[ "nitrogen":1 "hydrogen":4 "chlorine":1 ]#
      ]#;
```

成员的次序无关

中间可用逗号或者空格

注意：数组类型初始化用#加中括号，

ILOG OPL 简明教程-(5-结构体)

下面举例说明OPL结构体(Tuples)的用法：

例，一个工厂有3种产品(面条,面包,蛋糕)，各产品的市场需求量为(100,200,300)。 工厂可以自己生产产品，也外包生产。如果自己生产，每个产品消耗一定的资源（面粉和鸡蛋），资源总量为(20,40)。如何确定每种产品自己生产和外包的产量，使得总费用最小。

		面条	面包	蛋糕
资源消耗	面粉	0.5	0.4	0.3
	鸡蛋	0.2	0.4	0.6
费用情况	自己生产	0.6	0.8	0.3
	外包	0.8	0.9	0.4

ILOG OPL 简明教程-(5-结构体)

容易得到这个线性规划模型的**OPL**程序：

```
{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];

//constraint naming.
constraint ct1;
constraint ct2;
```

```
{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];

//constraint naming.
constraint ct1;
constraint ct2;

minimize
  sum(p in Products) (insideCost[p]*inside[p] + outsideCost[p]*outside[p]);

subject to {
  ct1 = forall(r in Resources)
    sum(p in Products) consumption[p][r] * inside[p] <= capacity[r];

  ct2 = forall(p in Products)
    inside[p] + outside[p] >= demand[p];
}
```

ILOG OPL 简明教程-(5-结构体)

```
minimize
  sum(p in Products) (insideCost[p]*inside[p] + outsideCost[p]*outside[p]);

subject to {
  ct1 = forall(r in Resources)
    sum(p in Products) consumption[p][r] * inside[p] <= capacity[r];

  ct2 = forall(p in Products)
    inside[p] + outside[p] >= demand[p];
}
```

ILOG OPL 简明教程-(5-结构体)

数据文件：

```
Products = {"Noodle", "Bread", "Cake" };
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 ];
```

ILOG OPL 简明教程-(5-结构体)

但是，从数据分离的角度来说，上述模型仍然有问题。

demand, insideCost, outsideCost, consumption都是关于**Products**的相关信息，但是被定义成独立的数组，这样模型可读性差，不宜于维护且容易修改出错。

利用**OPL**的**Tuples** 是一个解决问题的办法。原来的程序可以修改为：

```
{string} Products = ...;
{string} Resources = ...;
tuple ProductData {
  float demand;
  float insideCost;
  float outsideCost;
  float consumption[Resources];
}
ProductData product[Products] = ...;
float capacity[Resources] = ...;
```

相对于声明一个结构体类型
注意：末尾无分号！！

相对于定义一个结构体数组

ILOG OPL 简明教程-(5-结构体)

相对于使用结构体变量中的成员（也用一个点取其成员）

```
dvar float+ inside[Products];
dvar float+ outside[Products];

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

ILOG OPL 简明教程-(5-结构体)

数据文件修改为:

相对于初始化结构体数组

```
Products =  { "Noodle", "Bread", "Cake" };
Resources = { "flour", "eggs" };
product = #[
      Noodle : < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      Bread : < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      Cake : < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
        ]#;
capacity = [ 20, 40 ];
```

注意：每个结构体变量初始化使用<和>

Product的初始化也可以简化写为

```
product = [
      < 100, 0.6, 0.8, [ 0.5, 0.2 ] >,
      < 200, 0.8, 0.9, [ 0.4, 0.4 ] >,
      < 300, 0.3, 0.4, [ 0.3, 0.6 ] >
        ];
```

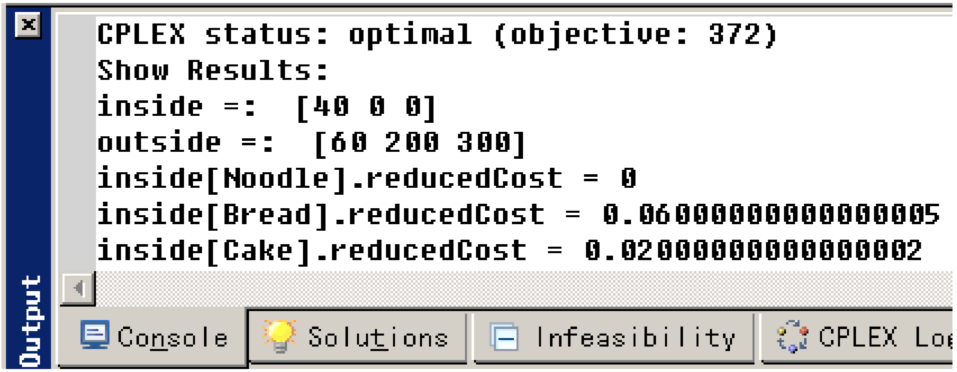
ILOG OPL 简明教程-(6-显示结果)

ILOG提供脚本（Script）可以帮助显示程序运行的结果。

在前面的.mod文件的末尾加入以下代码：

函数：执行脚本

```
execute{
writeln("Show Results: ");
writeln("inside =: ",inside);
writeln("outside =: ",outside);

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



输出一行信息

输出结果如图所示（注意在Console面板中）

比较C语言的sprintf函数

ILOG OPL 简明教程-(7-设置参数)

ILOG提供脚本（Script）可以设置一些CPLEX参数

在前面的**.mod**文件的末尾加入以下代码：

```
execute PARAMS {
  cplex.tilim = 100;
}
```

最大求解时间**time limit=100**秒

CPLEX和**OPL**有很多可以设置的参数，具体可以参见帮助文档中的“**CPLEX Parameters and OPL Parameters**”

例：整数规划—多背包问题

一个背包有7种资源（例如体积、重量），每个资源的总量为(18209, 7692, 1333, 924, 26638, 61188, 13360)。有12个物品，每个物品对应的价格为(96, 76, 56, 11, 86, 10, 66, 86, 83, 12, 9, 81)。一个物品放入背包时，所占用的资源不同（见表）。问题是如何放置，使总的价格最多。

12个物品

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

7种资源

例：整数规划一多背包问题

上述问题给出的是一个案例,但我们按照通用的多背包问题建模.

首先,定义背包的物品总数、资源总数， 并定义相应的range以备数组用:

```
int nbItems = ...;
int nbResources = ...;
range Items = 1..nbItems;
range Resources = 1..nbResources;
```

具体的数量可以由单独的data文件给出.

然后定义资源总量、价格、物品占用资源的数组:

```
int capacity[Resources] = ...;
int value[Items] = ...;
int use[Resources][Items] = ...;
```

资源总量的数组

例：整数规划—多背包问题

放入背包的物品所占的资源都是整数，占资源最少为1。那么，背包放入的物品个数，最小是1个，最大的可能为：

maxCount 为 Max (18209, 7692, 1333, 924, 26638, 61188, 13360)

背包里每种物品放多少？

如果定义**take[Items]** 为决策变量，那么其范围可以定为 **[1, maxCount]**

取最大函数

用**OPL**代码来表示：

```
int maxValue = max(r in Resources) capacity[r];
dvar int take[Items] in 0..maxValue;
```

限定了范围：有利于求解速度

目标是背包里的价格之和，可以表示为：

```
maximize sum(i in Items) value[i] * take[i];
```

例：整数规划—多背包问题

对于每种资源，背包里的物品占用的资源总和都不许超过资源总量，可以表示为：

```
constraint ct;
subject to {
  ct = forall(r in Resources)
  sum(i in Items) use[r][i] * take[i] <= capacity[r];
}
```

例：整数规划—多背包问题

综上，**mod**文件为：

```
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;

constraint ct;

maximize
  sum(i in Items) value[i] * take[i];

subject to {
  ct = forall(r in Resources)
        sum(i in Items) use[r][i] * take[i] <= capacity[r];
}
```

例：整数规划—多背包问题

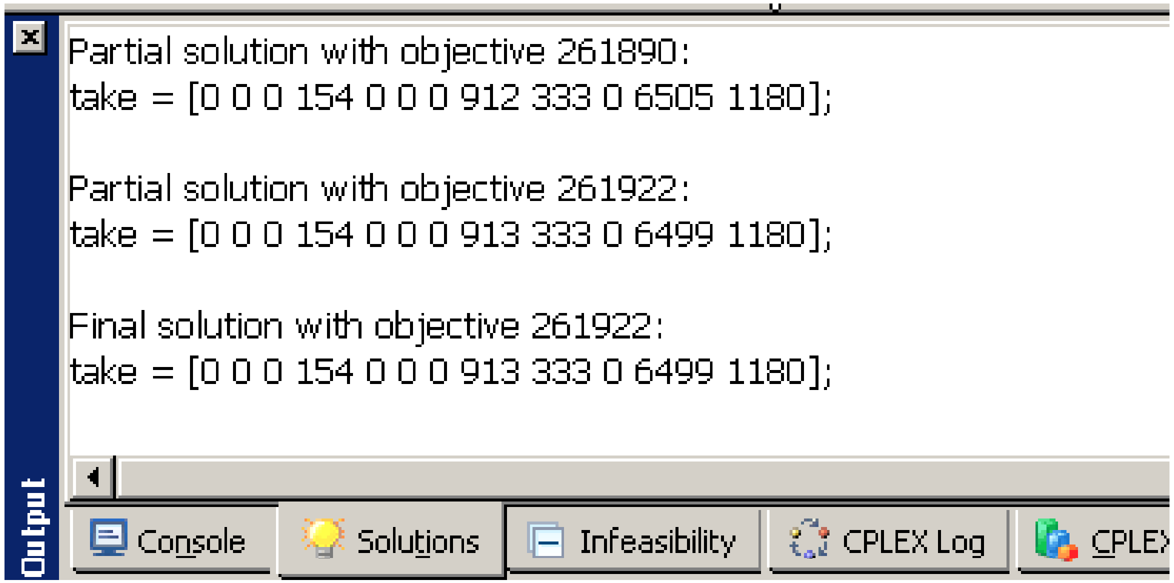
对应的数据文件为：

```
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]];
```

例：整数规划—多背包问题

运行结果：

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



例：混合整数规划—混成问题

一个工厂要生产71吨合金。该合金使用3种金属合成。生产该合金有四种途径：

1）直接购买这3种金属合成，对应3种金属的价格是(22, 10, 13)万/吨；

2）购买原材料（如矿石）炼制，现有2种原材料，价格分别为(6, 5)万/吨；第1种原材料含3种金属的百分比为(0.2, 0.05, 0.05)，第二种是(0.01, 0, 0.3);

3）购买废料炼制，现有2种废料，价格分别为(7, 8)万/吨；第1种原材料含3种金属的百分比为(0, 0.6, 0.4)，第二种是(0.01, 0, 0.7);

4）购买锭铁。其价格为9万/吨；含3种金属的百分比为(0.1, 0.45, 0.45)

生产合金时，合金中3种金属的最低含量的百分比为(0.05, 0.30, 0.60)，最高含量为(0.10, 0.40, 0.80)。

决策问题是：选用哪种途径生产合金，每种途径购买的物料的量是多少。

注意锭铁的量是整数类型

例：混合整数规划—混成问题

首先定义金属、原材料、废料、锭铁的种类个数，以及对应的**range**：

```
int nbMetals = ...;
int nbRaw = ...;
int nbScrap = ...;
int nbIngo = ...;
```

3
2
2
1

然后定义上述4种材料对应的价格数组：

```
float costMetal[Metals] = ...;
float costRaw[Raws] = ...;
float costScrap[Scraps] = ...;
float costIngo[Ingos] = ...;
```

22, 10, 13
6, 5
7, 8
9

定义合金中3种金属的最低含量的百分比的数组：

```
float low[Metals] = ...;
float up[Metals] = ...;
```

0.05, 0.30, 0.60
0.10, 0.40, 0.80

例：混合整数规划—混成问题

然后定义 原材料、废料、锭铁中 含3种金属的百分比，注意这里是二维数组。例如，原材料有2种，每种的3种金属的百分比都不同。

```
float percRaw[Metals][Raws] = ...;
float percScrap[Metals][Scraps] = ...;
float percIngo[Metals][Ingos] = ...;
```

```
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 ] ];
```

最后定义合金的总重量：

```
int alloy = ...;
```

```
71
```

定义决策变量。设购买3种金属的重量为w，购买原材料、废料的重量为r、s，购买锭铁的个数为i：

```
dvar float+    w[Metals];
dvar float+    r[Raws];
dvar float+    s[Scraps];
dvar int+      i[Ingos];
```

注意锭铁的量是整数类型

例：混合整数规划—混成问题

不论采取何种购买策略，定义最终3种金属的合计重量为：

```
dvar float m[j in Metals]；//实际上这是一个中间变量
```

考虑到 合金 包含的3种金属 有上限和下限，上述定义可以修改为：

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

3种金属的总重量必然等于合金重量，这显然有约束

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

对于每种金属，其重量等于购买的重量，故有约束

```
forall(j in Metals)
m[j] ==
        w[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];
```

直接购买的金属量

例：混合整数规划—混成问题

目标是总价格总小：

```
minimize
  sum(j in Metals) costMetal[j] * w[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];
```

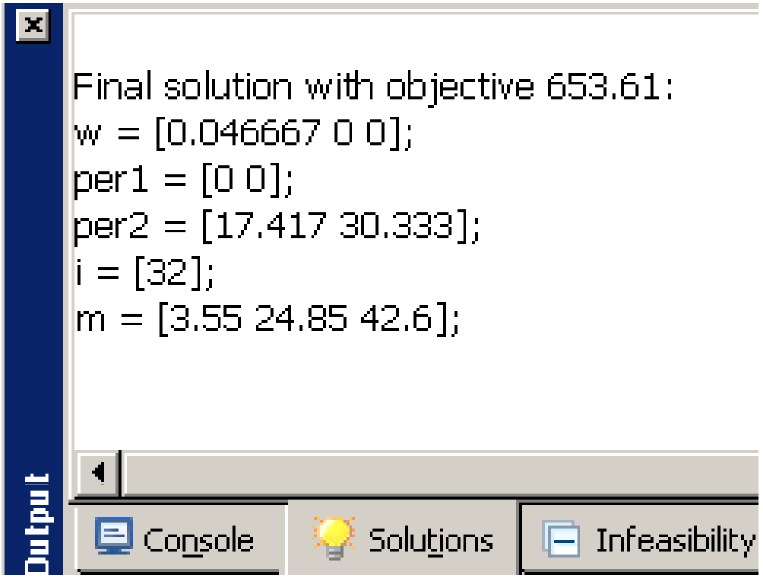
数据文件：

```
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 ] ];
```

例：混合整数规划—混成问题

Mod文件

```
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  = ...;

dvar float+    w[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;

constraint ct1;
constraint ct2;

minimize
   sum(j in Metals) costMetal[j] * w[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 {
ct1=forall(j in Metals)
          m[j] ==
           w[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;
}
```



最优结果：

3种金属中，只购买第一种0.046667吨；
不购买原材料；
购买2种废料的重量分别是(17.417,30.333)吨；
购买锭铁32个。

OPL 建模技巧

1、稀疏数据

在OPL建模时，要特别注意稀疏数据的处理，以便于大规模问题的求解。下面举例说明。

有若干城市，两两相通。有若干产品。每个城市出产一些产品，同时需要一些产品。假定总的产品出产和需要相等。货运城市运输量给定。不同产品从不同城市到另一个城市的运输费用可能不同。问题是如何确定运输方案，使得总运输费用最小。

OPL 建模技巧

首先定义城市、产品、总运输量：

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

然后定义城市出产产品和需要产品的数组：

```
float supply[Products][Cities] = ...;
float demand[Products][Cities] = ...;
```

因为有个“假定总的产品出产和需要相等”的假定，可以用assert检查数据：

```
assert forall(p in Products)
        sum(o in Cities)  supply[p][o] == sum(d in Cities) demand[p][d];
```

OPL 建模技巧

定义不同产品从不同城市到另一个城市的运输费用数组：

```
float cost[Products][Cities][Cities] = ...;
```

定义决策变量，即不同产品从不同城市到另一个城市的运输量：

```
dvar float+ trans[Products][Cities][Cities];
```

目标是总的运输费用最小，即：

```
minimize sum(p in Products, o,d in Cities) cost[p][o][d] * trans[p][o][d];
```

对于某个城市的某个产品的出产而言，可能被分送到若干其它城市，但总量一定和出产量相等，即：

```
forall(p in Products, o in Cities)
        sum(d in Cities) trans[p][o][d] == supply[p][o];
```

OPL 建模技巧

对于某个城市的某个产品的需求而言，可能从若干其它城市进货，但总量一定和需求量相等，即：

```
forall(p in Products, d in Cities)
        sum(o in Cities) trans[p][o][d] == demand[p][d];
```

另外，运输的总量不能超过给定的运输量，即：

```
forall(o, d in Cities)
        sum(p in Products) trans[p][o][d] <= capacity;
```

总的OPL代码的mod文件和dat文件见下页。

OPL 建模技巧

```
{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];

//constraint naming.
constraint ct1;
constraint ct2;
constraint ct3;

minimize
  sum(p in Products, o,d in Cities) cost[p][o][d] *
trans[p][o][d];

subject to {
ct1=   forall(p in Products, o in Cities)
        sum(d in Cities) trans[p][o][d] == supply[p][o];
ct2=   forall(p in Products, d in Cities)
        sum(o in Cities) trans[p][o][d] == demand[p][d];
ct3=   forall(o, d in Cities)
        sum(p in Products) trans[p][o][d] <= capacity;
}
```

```
Cities = { GARY CLEV PITT FRA  DET  LAN  WIN
STL  FRE  LAF };
Products   = { bands coils plate };
capacity    = 625;

supply = #[
    bands: #[
       GARY: 400
              CLEV: 700
              PITT: 800
              FRA: 0
              DET: 0
              LAN: 0
              WIN: 0
              STL: 0
              FRE: 0
              LAF: 0
           ]#
     coils: #[
       GARY: 800
              CLEV: 1600
              PITT: 1800
              FRA: 0
              DET: 0
              LAN: 0
              WIN: 0
              STL: 0
              FRE: 0
              LAF: 0
           ]#
     plate: #[
       GARY: 200
              CLEV: 300
              PITT: 300
```

OPL 建模技巧

下面我们看看这个例子的问题。这个例子有10个城市，城市之间的连通可能是10×10；产品总数为3，那么，把一个产品从一个城市运送到另一个城市，总的可能有3×10×10种。

但是，实际问题中，不是所有城市直接都连通，也不是每个产品都出产和需要每种产品。例如，下面用《产品，出发城市，到达城市》的方式表示了所有可能的产品运输方式。

```
<"Godiva","Brussels","Paris">        <"Godiva","Brussels","Bonn">        <"Godiva","Amsterdam","London">
<"Godiva","Amsterdam","Milan">       <"Godiva","Antwerpen","Madrid">     <"Godiva","Antwerpen","Bergen">
<"Neuhaus","Brussels","Milan">       <"Neuhaus","Brussels","Bergen">     <"Neuhaus","Amsterdam","Madrid">
<"Neuhaus","Amsterdam","Cassis">     <"Neuhaus","Antwerpen","Paris">     <"Neuhaus","Antwerpen","Bonn">
<"Leonidas","Brussels","Bonn">       <"Leonidas","Brussels","Milan">     <"Leonidas","Amsterdam","Paris">
<"Leonidas","Amsterdam","Cassis">    <"Leonidas","Antwerpen","London">   <"Leonidas","Antwerpen","Bergen">
```

OPL 建模技巧

很显然，这是一个稀疏数据的问题。在问题规模很大时，上述的模型的数据初始化量大、求解效率不高。下面我们首先用比较直观的方法修改模型。

套用上面《产品，出发城市，到达城市》的方式，在OPL中可用tuple结构实现：

```
tuple Route { string p; string o; string d; }
{Route} Routes = ...;
```

product, origin, destination

产品出产和需求的数组也可以用tuple来实现

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

Tuple的条件过滤：出发城市对应供应方

```
tuple Customer { string p; string d; }
{Customer} Customers = { <p,d> | <p,o,d> in Routes };
float demand[Customers] = ...;
```

OPL 建模技巧

费用数组改为：

```
float cost[Routes] = ...;
```

注意：数组下标集合，orig是一个数组

可能始发和终点的城市集合为：

```
{string} orig[p in Products] = { o | <p,o,d> in Routes };
{string} dest[p in Products] = { d | <p,o,d> in Routes };
```

Assert语句改为：

```
assert forall(p in Products)
        sum(o in orig[p]) supply[<p,o>] == sum(d in dest[p]) demand[<p,d>];
```

决策变量改为：

```
dvar float+ trans[Routes];
```

注意：Route为一维下标，简化了问题
参见前面决策变量（3维）

OPL 建模技巧

目标和约束相应可改为：

```
constraint ct1;
constraint ct2;
constraint ct3;
minimize sum(l in Routes) cost[l] * trans[l];
subject to {
ct1= forall(p in Products, o in orig[p])
        sum(d in dest[p]) trans[< p,o,d >] == supply[<p,o>];
ct2= forall(p in Products, d in dest[p])
        sum(o in orig[p]) trans[< p,o,d >] == demand[<p,d>];
ct3= forall(o, d in Cities) sum(<p,o,d> in Routes) trans[<p,o,d>] <= capacity;
}
```

Mod文件和**dat**文件见下页。

注意Route是下标集合，所以外面用{}；
因为元素是tuple，所以内部用<>

OPL 建模技巧

```
{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 naming.
constraint ct1;
constraint ct2;
constraint ct3;

minimize
  sum(l in Routes) cost[l] * trans[l];

subject to {
ct1=  forall(p in Products, o in orig[p])
      sum(d in dest[p]) trans[< p,o,d >] == supply[<p,o>];
```

```
Cities = { GARY CLEV PITT FRA  DET  LAN  WIN  STL  FRE  LAF };
Products   = { bands coils plate };
capacity   = 625;

Routes = {
   <bands GARY FRA>,
   <bands GARY DET>,
   <bands GARY LAN>,
   <bands GARY WIN>,
   <bands GARY STL>,
   <bands GARY FRE>,
   <bands GARY LAF>,
   <bands CLEV FRA>,
   <bands CLEV DET>,
   <bands CLEV LAN>,
   <bands CLEV WIN>,
   <bands CLEV STL>,
   <bands CLEV FRE>,
   <bands CLEV LAF>,
   <bands PITT FRA>,
   <bands PITT DET>,
   <bands PITT LAN>,
   <bands PITT WIN>,
   <bands PITT STL>,
   <bands PITT FRE>,
   <bands PITT LAF>,

   <coils GARY FRA>,
   <coils GARY DET>,
   <coils GARY LAN>,
   <coils GARY WIN>,
   <coils GARY STL>,
   <coils GARY FRE>,
```

OPL 建模技巧

上述的模型看起来每什么问题，但Route 采用{ string p; string o; string d; } 的模式，多个产品可能对应一个 <string o; string d >， 模型的表达和初始化仍然存在冗余，可以进一步改进成下面的结构:

Tuple嵌套定义

```
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] = ...;
```

注意c.o的用法

OPL 建模技巧

```
{string} orig[p in Products] = { c.o | <p,c> in Routes };
{string} dest[p in Products] = { c.d | <p,c> in Routes };
{Connection} CP[p in Products] = { c | <p,c> in Routes };
```

针对每个产品的connection集

约束：

```
ct1= forall(p in Products, o in orig[p])
        sum(<o,d> in CP[p]) trans[< p,<o,d> >] == supply[<p,o>];
ct2= forall(p in Products, d in dest[p])
        sum(<o,d> in CP[p]) trans[< p,<o,d> >] == demand[<p,d>];
ct3= forall(c in Connections)
        sum(<p,c> in Routes) trans[<p,c>] <= capacity;
```

Mod文件和dat文件见下页。

OPL 建模技巧

```
{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} CP[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 naming.
constraint ct1;
constraint ct2;
constraint ct3;

minimize
  sum(l in Routes) cost[l] * trans[l];
subject to {
ct1=  forall(p in Products, o in orig[p])
```

```
Cities = { GARY CLEV PITT FRA  DET  LAN  WIN  STL  FRE  LAF };
Products   = { bands coils plate };
capacity   = 625;

Routes = {
  <bands <GARY FRA> >,
  <bands <GARY DET> >,
  <bands <GARY LAN> >,
  <bands <GARY WIN> >,
  <bands <GARY STL> >,
  <bands <GARY FRE> >,
  <bands <GARY LAF> >,
  <bands <CLEV FRA> >,
  <bands <CLEV DET> >,
  <bands <CLEV LAN> >,
  <bands <CLEV WIN> >,
  <bands <CLEV STL> >,
  <bands <CLEV FRE> >,
  <bands <CLEV LAF> >,
  <bands <PITT FRA> >,
  <bands <PITT DET> >,
  <bands <PITT LAN> >,
  <bands <PITT WIN> >,
  <bands <PITT STL> >,
  <bands <PITT FRE> >,
  <bands <PITT LAF> >,

  <coils <GARY FRA> >,
  <coils <GARY DET> >,
  <coils <GARY LAN> >,
  <coils <GARY WIN> >,
  <coils <GARY STL> >,
  <coils <GARY FRE> >,
```

OPL 建模技巧

2、数组定义

使用多维数组时，数组**index**的次序会影响到程序的效率。例如

```
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];
```

因为**OPL**缓存机制的原因， **a2**的效率比**a1**高

3、数组初始化

OPL 建模技巧

```
range r=1..2;
int values1[r][r];
execute exec_values1 {
  for (i in r)
  {
    for (j in r)
       if (i == 2*j)
            values1[i][j] = i+j;
  }
   writeln(values1);
}
```

定义数组的同时，进行初始化
即generic arrays

```
int values2[i in r][j in r] = (i==2*j) ? i+j : 0;
execute exec_values2 {
writeln(values2);
}
```

values2的效率比values1高

OPL 建模技巧

4、Generic arrays

应该尽量使用**generic arrays**（数组成员由一个表达式初始化）

例如：

:

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

a[i]的值为 **i+1.**

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

m[i][j] 的值是 **10*i + j.**

int m[Dim1][Dim2] = ...;

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

利用一个现有数组进行初始化

OPL 建模技巧

4、其它杂项

变量命名遵循匈牙利命名规则；
尽量给约束起一个有意义的名字；
尽量多写注释语句；
注意语句的缩进；