Modelling Objects

Peter Stuckey & Guido Tack

MONASH
University

The Toy Shop Planner

- We only solved a single instance of this problem
- Everything was fixed:
  - the productivity, the cost per robot, the number of robots available
  - and, crucially: **the set of robot types**!
- What if we want to **reuse** the same model for different instances?

Toys per hour	6	10	8	40
Cost per robot	$13k	$21k	$17k	$100k
Robots available	1000	400	500	150

The Toy Shop Planner

```
enum ROBOT;
int: budget;

array[ROBOT] of int: available;
array[ROBOT] of int: cost;
array[ROBOT] of int: tph; % toys per hour

% how many robots to buy
arr                    .inf

constraint
    forall(r in ROBOT) (buy[r] <= available[r]);

constraint
    sum(r in ROBOT) (cost[r]*buy[r]) <= budget;

solve maximize
    sum(r in ROBOT) (tph[r]*buy[r]);
```

array declarations

forall expression

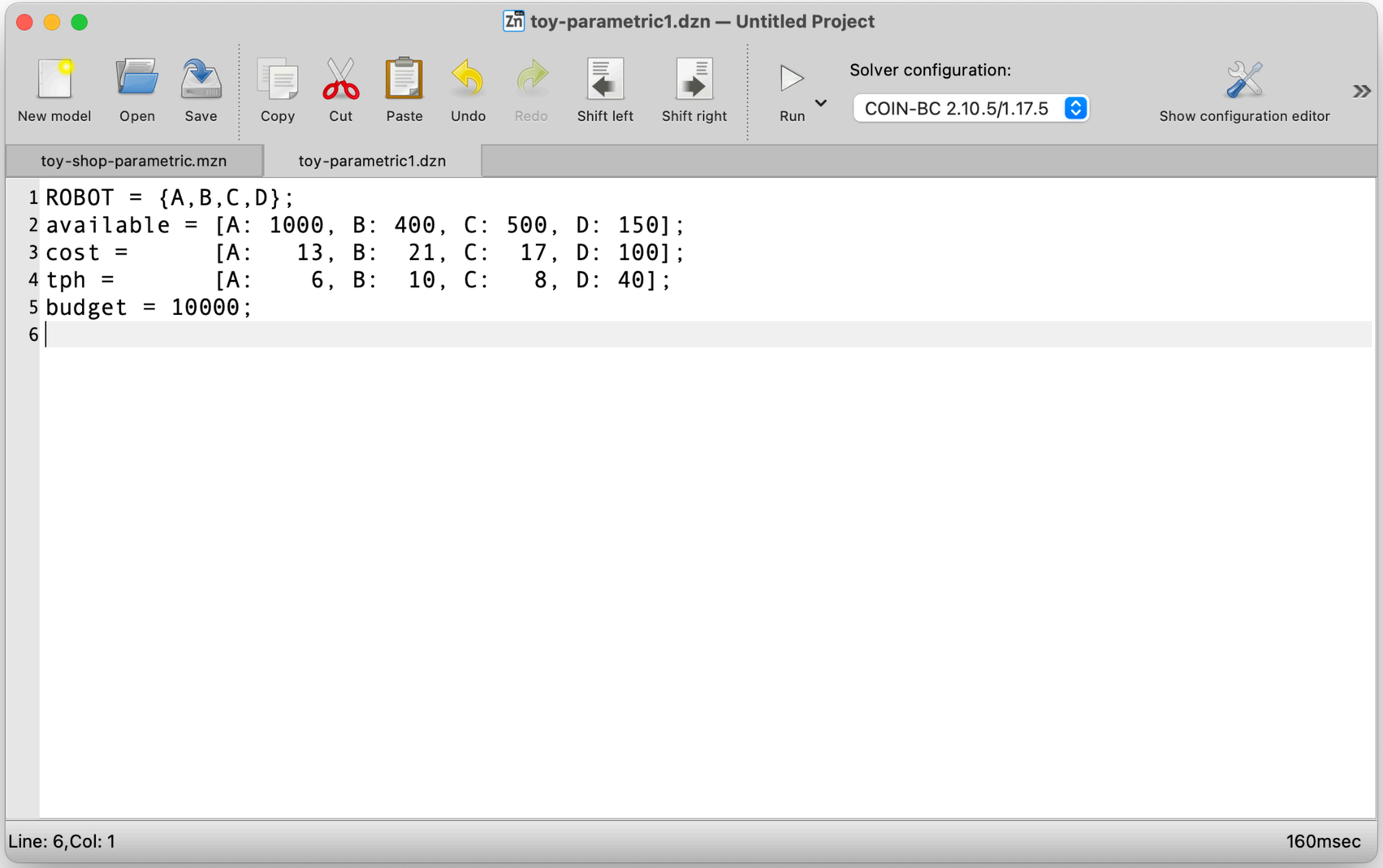
array lookup

sum over an array

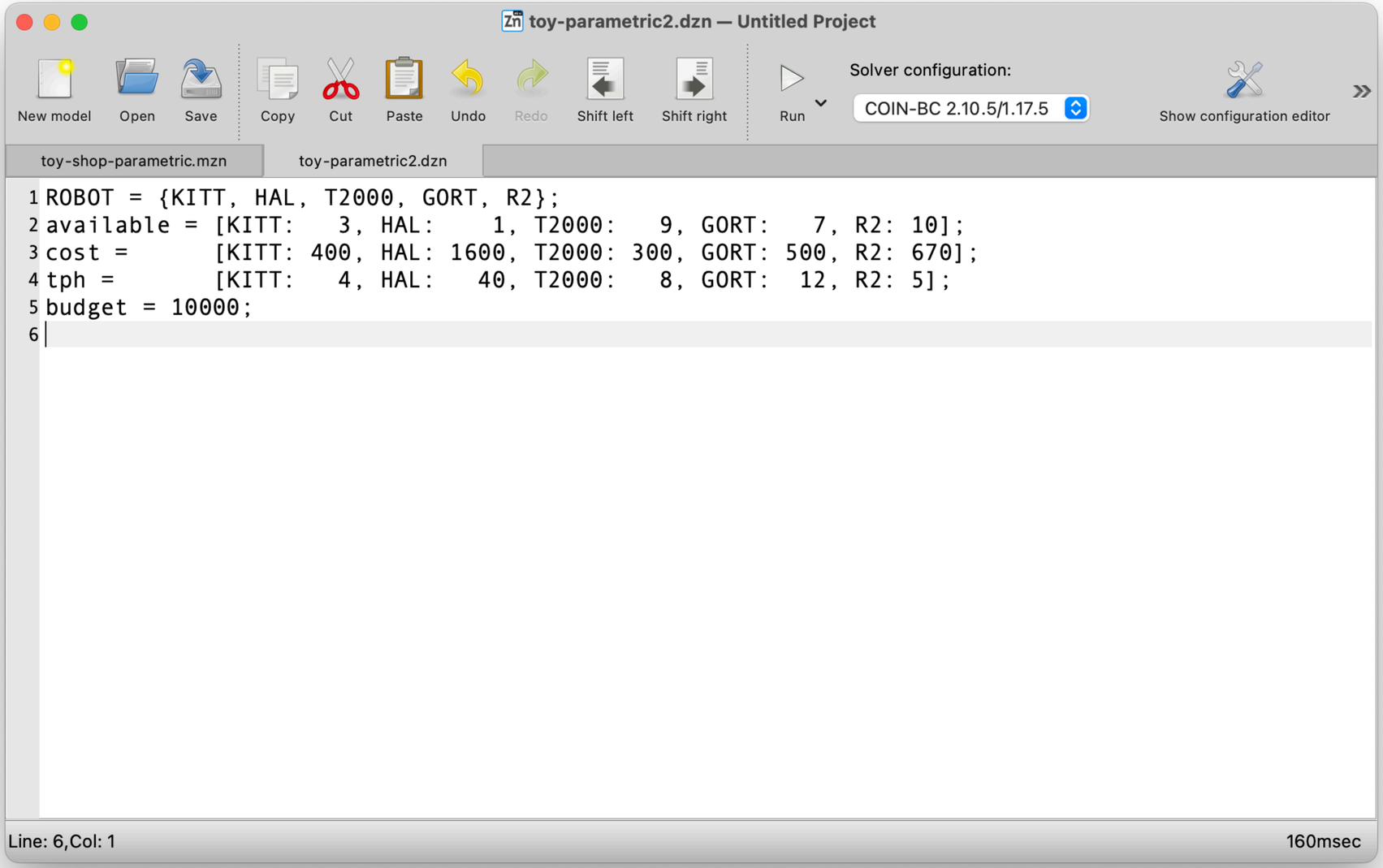
Arrays and Generators

- Arrays can hold parameters and variables
  `array[range] of variable declaration`
- Range expressions:
  `l..u` (where `l`, `u` are integers), or enumerated type
- Array lookup
  `array-name[index-expression]`
- Generator expressions
  - `forall(i in range) (bool-expression)`
    for all `i` in the range, `bool-expression` is true
  - `sum(i in range) (expression)`
    sum over all `expression` for all `i` in the range

Data files with arrays (toy-parametric1.dzn)



Data files with arrays (toy-parametric2.dzn)



Modelling Objects

- Create an **enumerated type** with the object names (e.g. `ROBOT`)
- Create **parameter arrays** for each attribute of the object (e.g. `available`, `cost`, `tph`)
- Create **variable arrays** for each decision of the objects (e.g. `buy`)
- Use **generators** to add constraints over the objects (e.g. `forall`, `sum`)
- Models often have multiple sets of objects

Summary

- Use enumerated types to represent **sets of objects**
- Arrays represent objects **attributes** and **decisions**
- **Generator expressions** are used to construct expressions over multiple objects
- The toy shop planner problem is a version of the well-known **knapsack problem**

Arrays

Peter Stuckey & Guido Tack

MONASH
University

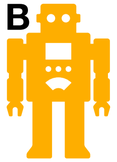
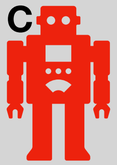
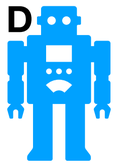
Production Planning

- Now that we've bought the robots, let's plan the actual toy production.
- We can produce 5 types of toys, each with an associated profit:

toy:					
profit:	11	18	15	17	11

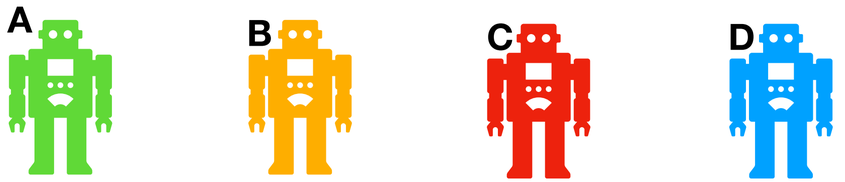
Production Planning

- Producing each toy needs a certain amount of time of each robot:

					
A 	1.5	2.0	1.5	0.5	0.1
B 	1.0	0.0	0.5	1.0	2.5
C 	1.0	2.0	1.0	0.9	0.1
D 	1.0	0.0	1.0	1.5	2.5

Production Planning

- And each robot has a limited production capacity:



5000 7500 4000 3000

Production Planning

- This problem is very similar to earlier problems
  - We have "products" to make
  - Each product consumes some resources
  - There are limits on resources
  - We want to maximise profit
- Original toy shop problem: resource = budget, product = robots, profit = toys per hour
- A generic model can capture all kinds of production planning problems

Production Planning: Parameters

```
% products
enum PRODUCT;
% Profit per unit for each product
array[PRODUCT] of float: profit;

% resources
enum RESOURCE;
% Amount of each resource available
array[RESOURCE] of float: capacity;

% Units of each resource required to produce
%       1 unit of product
array[RESOURCE,PRODUCT] of float: consumption;
```

two-dimensional array

Production Planning: Constraints

```
% Variables: how much to make of each product
array[PRODUCT] of var 0..infinity: produce;


% Production can only use the available resources:
constraint forall(r in RESOURCE)(
  sum (p in PRODUCT)
      (consumption[r,p] * produce[p]) <= capacity[r]
);


% Maximize profit
var int: objective;
constraint objective= sum(p in PRODUCT)
   (profit[p]*produce[p]);
solve maximize objective;
```

two-dimensional array lookup

Arrays

- Arrays can have multiple dimensions:

  `array[`*`index_set_1`*`, `*`index_set_2`*`, ...] of `*`type`*

- Each index set needs to be
  - an integer range (e.g. `1..10`)
  - an enumerated type (e.g. `PRODUCT`)
  - or a fixed set expression whose value is a range (e.g. `min(S)..max(T)`)
- Arrays can contain any type **except** other arrays

Array literals

- 1-dimensional arrays are initialised using a list, e.g.

```
profit = [400, 500];
capacity = [4000, 6, 2000, 60, 50];
```

- We have special syntax for 2-dimensional arrays:

```
consumption = [| 1.5, 1.0, 1.0, 1.0
               | 2.0, 0.0, 2.0, 0.0
               | 1.5, 0.5, 1.0, 1.0
               | 0.5, 1.0, 0.9, 1.5
               | 0.1, 2.5, 0.1, 2.5 |];
```

start with `[|`

separate rows with `|`

end with `|]`

Indexed Array Literals

- We can also specify the indices directly
- For 1d literals:

```
capacity = [R1: 4000, R2: 6, R3: 2000,
            R4: 60, R5: 50];
```

- For 2d literals:

```
consumption =
  [|              A:   B:   C:   D:
   | Duck:        1.5, 1.0, 1.0, 1.0
   | Rattle:      2.0, 0.0, 2.0, 0.0
   | PiggyBank:   1.5, 0.5, 1.0, 1.0
   | Puzzle:      0.5, 1.0, 0.9, 1.5
   | Balloon:     0.1, 2.5, 0.1, 2.5 |];
```

Array index set coercion

- We can *coerce* any array into an array with a different index set, e.g. 1D to 2D:

```
consumption = array2d(1..5, 1..4,
    [1.5, 1.0, 1.0, 1.0,
     2.0, 0.0, 2.0, 0.0,
     1.5, 0.5, 1.0, 1.0,
     0.5, 1.0, 0.9, 1.5,
     0.1, 2.5, 0.1, 2.5];
```

index sets of resulting array

- Similar functions exist for 1 to 6 dimensions:

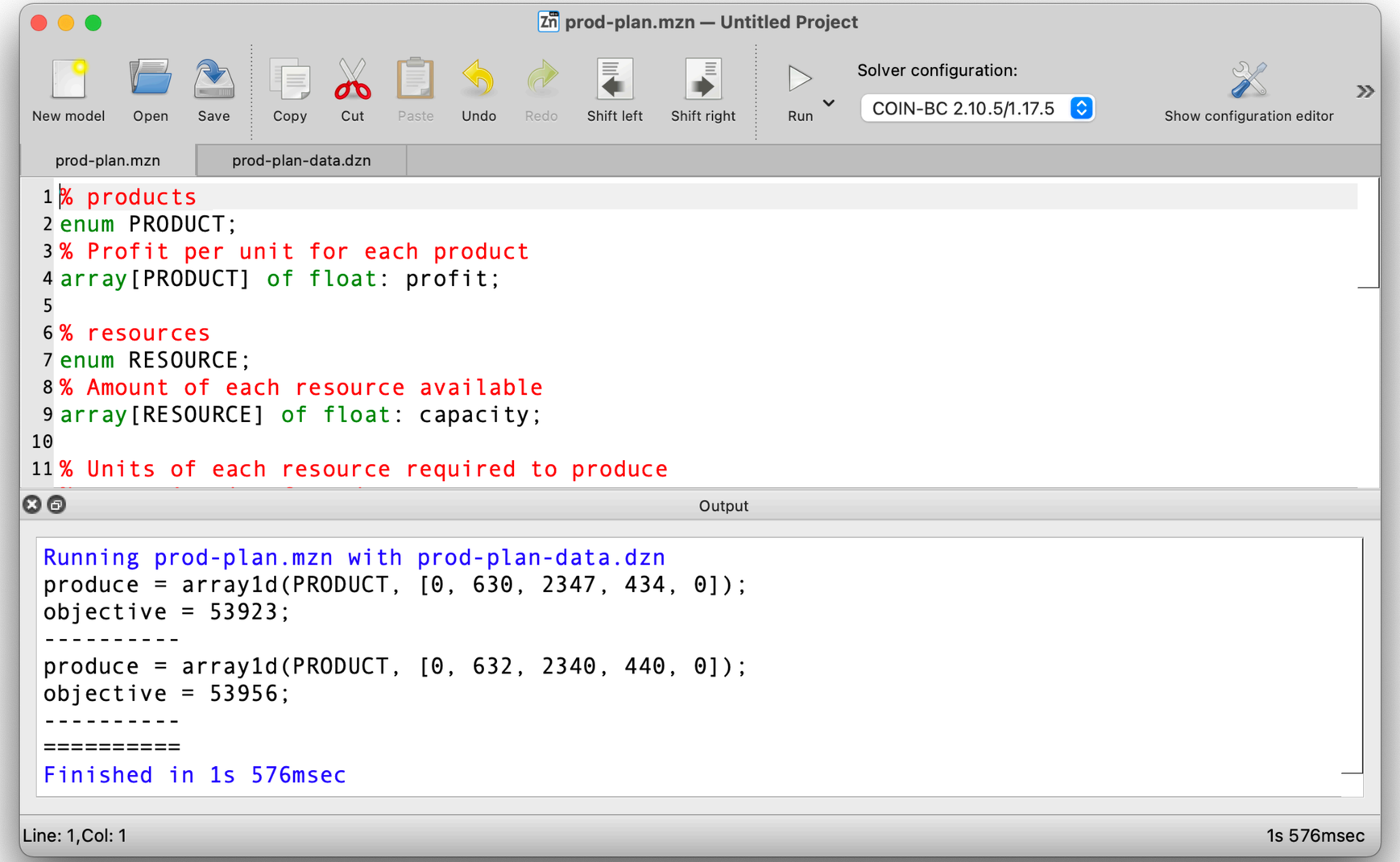
```
array1d, array2d, array3d, array4d,
array5d, array6d
```

- Error if the coerced array has the wrong size

Production Planning: Data

```
PRODUCT = { Duck, Rattle, PiggyBank,
            Puzzle, Ballon };

RESOURCE = { A, B, C, D };

capacity = [5000, 7500, 4000, 3000];

consumption =
  [| 1.5, 2.0, 1.5, 0.5, 0.1
   | 1.0, 0.0, 0.5, 1.0, 2.5
   | 1.0, 2.0, 1.0, 0.9, 0.1
   | 1.0, 0.0, 1.0, 1.5, 2.5 |];

profit = [11, 18, 15, 17, 11];
```

Production Planning: Solve



Production Planning: Other examples

- Toy shop planning problem 1:

availability of each type is a resource

```
PRODUCT = { A, B, C, D };
RESOURCE = { MONEY, AVAIL_A, AVAIL_B,
             AVAIL_C, AVAIL_D };
capacity = [ MONEY: 10000, AVAIL_A: 1000, AVAIL_B: 400,
             AVAIL_C: 500, AVAIL_D: 150];
consumption =
  [|          A:    B:    C:
   | MONEY:   13.0, 21.0, 17.0
   | AVAIL_A:  1.0,  0.0,  0.0
   | AVAIL_B:  0.0,  1.0,  0.0,   0.0
   | AVAIL_C:  0.0,  0.0,  1.0,   0.0
   | AVAIL_D:  0.0,  0.0,  0.0,   1.0 |];

profit = [A: 6.0, B: 10.0, C: 8.0, D: 40.0];
```

each robot requires exactly one of the availability resources

Production Planning: Other examples

- Toy shop planning problem 2:

```
PRODUCT = { KITT, HAL, T2000, GORT, R2 };
RESOURCE = { MONEY, AVAIL_KITT, AVAIL_HAL,
             AVAIL_T2000, AVAIL_GORT, AVAIL_R2 };
capacity = [10000, 3, 1, 9, 7, 10];
consumption =
  [| 400.0, 1600.0, 300.0, 500.0, 670.0
   | 1.0, 0.0, 0.0, 0.0, 0.0
   | 0.0, 1.0, 0.0, 0.0, 0.0
   | 0.0, 0.0, 1.0, 0.0, 0.0
   | 0.0, 0.0, 0.0, 1.0, 0.0
   | 0.0, 0.0, 0.0, 0.0, 1.0 |];
profit = [4.0, 40.0, 8.0, 12.0, 5.0];
```

Which solver should you use?

- We could use Gecode (the default solver)
  - it supports float variables and linear constraints
  - but solving takes a very long time (and finds lots of intermediate solutions before the optimal one)
- Better use COIN-BC: it solves almost instantly!
- Production planning is a **M**ixed **I**nteger Linear **P**rogramming Problem, therefore **MILP** solvers like COIN-BC are ideal
  - constraints and objective are linear
  - variables are mix of integer and float

Summary

- Real models apply to data of different sizes
- MiniZinc uses
  - enumerated types to name objects
  - arrays to capture information about objects
  - array literals can be indexed
- Generic models (using arrays) allow us to reuse models for many instances
- Production planning is a MIP problem, so MIP solvers work best

Comprehensions

Peter Stuckey & Guido Tack

MONASH
University

Array Comprehensions

- Comprehensions are used to construct arrays
- Known from many other programming languages
- Python: `[ 2 * x for x in range(20) if x*x > 3 ]`
- Haskell: `[ 2 * x | x <- [0..19], x^2 > 3 ]`
- MiniZinc: `[ x * 2 | x in 0..19 where x^2 > 3 ]`
- General form in MiniZinc:
  `[ expr | generator`$_1$ `where test`$_1$`, ...`
  `generator`$_k$ `where test`$_k$ `]`
- Each `generator` is of the form
  `identifier,...,identifier in expression`

Array Comprehensions: Examples

```
   [i + j | i, j in 1..4 where i < j]
= [1+2, 1+3, 1+4, 2+3, 2+4, 3+4]
= [3, 4, 5, 5, 6, 7]
```

```
   [ x[i] ≠ x[j] | i,j in 1..5 where i<j ]
= [ x[1]≠x[2], x[1]≠x[3], x[1]≠x[4],x[1]≠x[5],
    x[2]≠x[3], x[2]≠x[4], x[2]≠x[5],
    x[3]≠x[4], x[3]≠x[5],
    x[4]≠x[5] ]
```

```
   [ x[j,i] | i in 1..n, j in 1..m ]
= [ x[1,1], x[2,1], x[3,1], ..., x[m,1],
    ...
    x[1,n], x[2,n], x[3,n], ..., x[m,n] ]
```

Generators revisited

- Remember generator expressions?

```
sum(r in ROBOT) (cost[r]*buy[r]) <= budget;
```

- This is just special syntax for

```
sum([ cost[r]*buy[r] | r in ROBOT ]) <= budget
```

- It can be used with any function that takes a list as its argument, e.g.
  - for lists of numbers: `sum, product, min, max`
  - for lists of constraints: `forall, exists`

Example: pairwise different variables

- Constrain all pairs of variables in an array to take pairwise different values
- We will see later that this is one of the most useful constraints, at the core of many problems
- Easy to express with comprehensions/generators:

```
constraint
  forall (i,j in 1..n where i < j)
         (x[i] != x[j]);
```

Example: matrix transpose

- We can use comprehensions and `array2d` to transpose an array (flip it over its diagonal):

```
% original array:
array[RESOURCE,PRODUCT] of float: consumption;

% transposed:
array[PRODUCT,RESOURCE] of float: con_t =
  array2d(PRODUCT,RESOURCE,
    [ consumption[r,p] | p in PRODUCT,
                         r in RESOURCE ]);
```

- Important:
  - arrays are represented in "row-wise" order
  - construct by iterating through rows first, then columns

Indexed Array Comprehensions

- Similar to indexed arrays:

```
[ (idx_1,...,idx_k) : expr
| generator_1 where test_1, ...
  generator_k where test_k ]
```

- Generates a k-dimensional array
- Each element is placed at the specified index
- **Note:** index sets have to be **contiguous**

Example: matrix transpose

- Indexed comprehensions make transposition easier to write:

```
% original array:
array[RESOURCE,PRODUCT] of float:
consumption;

% transposed:
array[PRODUCT,RESOURCE] of float: con_t =
   [ (p,r) : consumption[r,p]
   | r in RESOURCE, p in PRODUCT ]);

% alternatively:
array[PRODUCT,RESOURCE] of float: con_t =
   [ (p,r) : consumption[r,p]
   | p in PRODUCT, r in RESOURCE ]);
```

This determines the index set

Example: distance matrix

- We can compute Euclidian distances given x/y coordinates of an array of points

```
enum POINT;
array[POINT] of float: x; % x-coordinates
array[POINT] of float: y; % y-coordinates

array[POINT,POINT] of float: distance =
   [ (p1, p2) :
     sqrt( (x[p1]-x[p2])^2 + (y[p1]-y[p2])^2 )
   | p1, p2 in POINT]);
```

Summary

- MiniZinc uses comprehensions extensively
  - to express constraints
  - to construct and transform arrays
- Only way to build arrays where the size is a parameter
- Generator calls are just syntactic sugar for comprehensions
- Indexed comprehensions create multi-dimensional arrays with specified index sets

Global Constraints

Peter Stuckey & Guido Tack

MONASH
University

Staff Allocation

- This is a (very simple) version of a staff rostering problem.
  - We have a set of staff members and a number of shifts.
  - Each staff has a certain preference for each available shift (higher = better).
  - Each staff member can only work one shift in the roster.
  - Goal: maximise the overall preferences.

Staff Allocation: Data and Decisions

```
enum Staff;
enum Shift;

% Preferences (higher = better)
array[Shift,Staff] of int: preference;

% Decisions: who works which shift
array[Shift] of var Staff: allocation;
```

Staff Allocation: Constraint

- Assume we have 4 shifts, S1 to S4.
- No two shifts are assigned to the same staff:

```
constraint allocation[S1] ≠ allocation[S2];
constraint allocation[S1] ≠ allocation[S3];
constraint allocation[S1] ≠ allocation[S4];
constraint allocation[S2] ≠ allocation[S3];
constraint allocation[S2] ≠ allocation[S4];
constraint allocation[S3] ≠ allocation[S4];
```

- This is tedious to write, and depends on the data.
- How can we do this generically (if we don't know the number of shifts)?

Staff Allocation: Constraint

- We can of course use `forall` with a generator:

```
constraint
  forall (s1, s2 in Shift where s1 < s2)
         (allocation[s1] ≠ allocation[s2]);
```

- This is a very common pattern!
- The core of many assignment problems.
- Use a **global constraint** instead:

include functionality from the MiniZinc library

```
include "all_different.mzn";
constraint all_different(allocation);
```

enforces pairwise not-equals of an array

Global Constraints

- Historically: any constraint over **more than two variables**
  - So, technically, linear constraints are global
- In our context:
  - constraints that capture **common modelling patterns**
  - pre-defined in the **MiniZinc library**
- Global constraints make
  - models more **concise**
  - solving **faster** (MiniZinc and backend solvers can use information about the structure of the problem)

Staff Allocation: Objective

```
enum Staff;
enum Shift;

% Preferences (higher = better)
array[Shift,Staff] of int: preference;

% Decisions: who works which shift
array[Shift] of var Staff: allocation;

include "all_different.mzn";
constraint all_different(allocation);

solve maximize
    sum(s in Shift)
       (preference[s,allocation[s]]);
```

variable array access!

Variable Array Access

- We can use decision variables to index into arrays.
- This is a very powerful modelling capability!
- You've just seen a simple example:

  map the allocated staff (a decision variable) to their preferences
- We will make use of this a lot in future models.

Include *(not that important...)*

- Syntax:
  `include "filename";`
  - Includes the specified file in your model
  - Will search the MiniZinc library path, the path of the model file and the working directory
- Includes are used to
  - break up large models into more manageable chunks
  - load definitions of global constraints from the library
  - control how the model is translated for a particular solver (more on that in a later module!)

Summary

- Global constraints are one of the most important concepts in this course.
- They capture common model structure and make models more concise and efficient to solve.
- Using decision variables as array indexes is a powerful modelling tool.