# PDL Manual

# Contents

# Chapter 1

# Introduction

Problem Description Language (PDL) is a modeling language for specifying constraint optimization problems. When using PDL, users only need to list all constraints between the variables without indicating how to solve the problem – the PDL runtime system will automatically design a feasible solving algorithm and generate the corresponding code.

PDL supports many common-used operators and functions. PDL lets users describe models in a way that is close to a mathematical formulation of the problem using familiar notations.

Compared with programming with traditional programming languages, programming with PDL has a lot of advantages:

- Users can solve the problems without any programming abilities, only some basic mathematical knowledge is required;
- PDL models are more readable than programs, which makes it easier to comprehense and debug;
- Users can focus on modeling, and save plenty of time in thinking of how to solve the problem;
- Bugs introduced by human errors can be avoided while using an automatic generation tool;
- Users can apply advanced optimizations based on the generated programs.

# Chapter 2

# PDL Examples

This chapter demonstrates PDL models of two problems. **White space can be freely inserted between tokens to improve readability.** Detailed explanations of usage will be provided in Chapter 3.

## 2.1 Cuboid Problem

Given the volume $V$, find a cuboid with the smallest surface area such that (1) the volume is V and (2) the lengths of all edges are integers.

```
#input
        V of int in [1,10^4];

#required
        L of int in [1,?];
        W of int in [1,?];
        H of int in [1,?];
        V = L * W * H;

#objective
        minimize (2 * (L * W + L * H + W * H));
```

## 2.2 0/1 Knapsack Problem

There are $N$ items, each has value and weight. Select a set of items to put in the knapsack, such that the total weight is not greater than the capacity and the total value is maximum.

```
#input
        N of int in [1,100];
        capacity of int in [1,1000];
```

```
        profits of (int in [1,1000])[1~N];
        weights of (int in [1,1000])[1~N];

#required
        knapsack of (int in [1,N]){};
        summation
            [weights[i] : forall i (i in knapsack)]
            <= capacity;

#objective
        maximize summation
            [profits[i] : forall i (i in knapsack)];
```

# Chapter 3

# PDL Tutorial

## 3.1 Basic Framework

A PDL model consists of three segments: `#input`, `#required`, and `#objective`. Each statement should be ended with a semicolon (";"). We will take 0/1 knapsack problem as an example to describe the basic framework of a PDL model in detail.

### 3.1.1 `#input` Section

This section declares all input parameters and their types as well as value ranges.

In the example, the `#input` section lists four input parameters of this problem: two variables `N` and `capacity` of integer type, and two variables `profits` and `weights` of integer array type. Meanwhile, the value ranges as well as index ranges are also declared.

**Note: These inputs will be read in the same order by the generated program, so please make sure that the declaring order is correct.**

### 3.1.2 `#required` Section

This section contains several statements. Each statement can be:

- a variable declaration, which is similar to the one in `#input` section, or
- a boolean expression indicating a mathematical relation between inputs and variables.

In the example, `#required` section first define a variable `knapsack` of set type—the indexes of items we are going to pick. Then follows a constraint that should be satisfied—the total weight of the selected items should not exceeding `capacity`.

### 3.1.3 `#objective` Section

This section declares zero or one objective function. If no objective function is defined, the resulting program will stop search after finding one solution; otherwise, it will search for the optimal solution that produces either the maximized or minimized value.

In the example, the objective function is to maximize the total profits of the selected items.

## 3.2 Variable Declaration

In PDL, input parameters or variables are declared by:

```
<identifier> of <type-and-range>
```

### 3.2.1 Identifier

An identifier of a user-defined item (input, variable, function ...) should be a sequence of letters (`a-z` or `A-Z`, case-sensitive), numbers (`0-9`), and underscores (_). The first character should be a letter. Some reserved keywords cannot be used as identifiers, including:

```
int, real, bool, char, function, of, in, true, false,
and, or, not, xor, mod, if, else, forall, exists,
summation, product, count, min, max, minimize, maximize
```

**Some reserved keywords in C/C++ cannot be used either**, such as `return`, `main`, etc.

### 3.2.2 Type and Range

PDL supports four primitive types and three composite types.

Primitive types are: integer type `int`, real type `real`, boolean type `bool`, and character type `char`. To limit range of a primitive type `T` between `lb` and `ub`, use `T in [lb,ub]`. Specially, use `?` if a bound of the range is unspecified.

Composite types are:

- Array type: an array of `T` type variables is represented as `T[]`, for example, `int[]` is an array of integers, `bool[][]` is a two-dimensional array of booleans. To limit the range of index between `li` and `ui`, use `T[li~ui]`. Similarly, use `?` if a bound is unspecified.
- Set type: a set of `T` type variables is represented as `T{}`.
- Tuple type: a tuple can include variables of different types, which is represented as `(T1,T2,...)`.

### 3.2.3 Examples of Variable Declaration

```
// an integer variable
i of int
// an integer variable with value range
n of int in [1,10^5]
// an integer array
a of int[]
// an integer array with index range
b of int[1~2*n]
// an integer array with both range and index ranges
c of (int in [1,10^6])[1~100]
// a multi-dimensional array with index ranges
d of (int in [1,100])[0~?][][-5~5]
// an integer set
s of int{}
// an integer set with value range
t of (int in [1,n]){}
// a tuple consists of elements of different types
p of (int, bool[1~10], int in [1,?])
// a set of tuples
q of (int, bool){}
```

## 3.3 Primitive Operator

PDL supports many common-used mathematical operators on primitive types, listed from greatest to smallest priority:

1. parenthesis `()`
2. power `^`
3. multiply `*`, divide `/`, modulus `mod`
4. add `+`, subtract `-`
5. type check `of`, range check `in`
6. `not`, `and`
7. `or`, `xor`
8. comparisons `=`, `!=`, `>`, `<`, `>=`, `<=`

## 3.4 Composite Operator

PDL also supports some built-in operator for composite types:

- Element selections: `[]`, the priority of which is second only to `()`. For instance, `a[5]`, `a[i+1]`, where `a` is an array or tuple. Note that the index of elements in a tuple starts at 1.
- Set operators: intersect `*`, union `+`, minus `-`. Each set operator has the same priority as the primitive operator sharing the same notation.
- Set comparisons, similar to primitive comparisons.
- Belong to: `in`, the priority of which equals to the comparisons'. An expression `e in C` returns a boolean value, indicating whether there exists an element `e` in the variable `C` of a composite type.
- Aggregate operators: `summation`, `product`, `count`, `max`, `min`. These operators can be applied to an array or set. For instance, `summation a` calculates the sum of all elements in array `a`, `max s` obtains the maximum element in set `s`.

## 3.5 Quantifier

PDL supports two quantifiers $\exists$ and $\forall$ in first-order logic, represented by `exists` and `forall`.

### 3.5.1 Existential Quantifier

An existential quantifier can be used as

```
exists <v1, v2, ...> (<bool-exp>)
```

The expression returns a boolean value, indicating whether there exists an assignment to all variables in the list that satisfies the following boolean expression.

Example: checking whether there exists an inversion in the array `a`.

```
exists i,j ((i<j) and (a[i]>a[j]))
```

### 3.5.2 Universal Quantifier

In PDL, a universal quantifier can be used for two different purposes.

**Generating an array** :

```
[<exp> : forall <v1, v2, ...> (<bool-exp>)]
```

This statement enumerates each value combination of variables that satisfies the boolean expression, generates an element by the front expression, and puts it into an array.

Example: select each element from array `a` whose index belongs to set `s`.

```
[ a[i] : forall i (i in s) ]
```

**Generating statements** :

```
<bool-exp> : forall <v1, v2, ...> (<bool-exp>)
```

This statement enumerates each value combination of variables that satisfies the latter boolean expression, generates a sequence of constraints by the first boolean expression.

Example: require the elements in array `a` are in increasing order.

```
a[i] < a[i+1] : forall i ( i of int in [1,N-1] )
```

**Note: the boolean expression after `forall` can be omitted if there is no extra constraint on the variables in the list.**

## 3.6 Conditional Statement

```
if <bool-exp> <exp1> else <exp2>
```

**Note: `else` and the expression after can be omitted.**

## 3.7 Function Define and Call

In PDL, users can self-define a function to simplify the description, or achieve recursive calculations.

Functions can be defined by:

```
<identifier> of function (<v1-def, v2-def, ...>)
    -> <type-and-range> = <exp>
```

where `identifier` is the function name, `(v1-def,v2-def,...)->type-and-range` defines the type of function arguments and the return value. The last expression defines how to calculate the return value.

Functions can be called by:

```
<identifier>(<v1, v2,...>)
```

Example: check whether an integer `x` is an odd number.
Function define:

```
isOdd of function (x of int) -> bool = (x mod 2 = 1)
```

Function call:

```
isOdd(5)
```

# Chapter 4

# Core Syntax of PDL

Program $PG$ :=$IP\ RQ\ OBJ$

Input Segment $IP$ :=**#input** $(VD;)*$

Required Segment $RQ$ :=**#required** $(Stmt;)*$

Objective Segment $OBJ$ :=**#objective** $((\texttt{minimize}|\texttt{maximize})Exp;)?$

Variable Declaration $VD$ :=`Id of` $Type$

Type Declaration $Type$ :=$PrimType \mid CompType$

Primitive Type $PrimType$ :=(`int` | `real` | `char` | `bool`) (`in` $[B,\ B]$)?

Composite Type $CompType$ := $Type[B \sim B]$ // *list*

$\mid Type\{\}$ // *set*

$\mid'('\ TypeList\ ')'$ // *tuple or struct*

Type List $TypeList$ :=$Type(,Type)*$

Bound $B$ :=$RValue \mid$ '?' // '?' *is "unspecified"*

Right Value $RValue$ :=`Num` | $Exp$

Statement $Stmt$ := $VD$ // *variable declaration*

$\mid ExpStmt$ // *expression statement*

$\mid ForEnum$

Expression Statement $ExpStmt$ :=$Exp$

Forall Enumeration $ForEnum$ :=$Exp$ : **forall** `Id`$(,$`Id`$)*'$ $('QExp')'$

$$
\begin{aligned}
\text{Expression } Exp ::= & \ '(' \ Exp \ ')' \\
& | AExp \ // \ atom\ expression \\
& | \texttt{not}\ Exp // \ unary\ expression; \\
& | BExp \ // \ binary\ expression \\
& | GExp \ // \ aggregation\ expression \\
& | \textbf{exists}\ \texttt{Id}(,\texttt{Id}) * ' \ ('QExp')' \\
& | \textbf{if}\ Exp \ '('Exp')' \ (\textbf{else}'('Exp')')? \\
\text{Atomic Expression } AExp ::= & \texttt{Id} \ | Exp \ [RValue] \\
\text{Binary Expression } BExp ::= & Exp \ BinOp \ Exp \\
\text{Aggregation Expression } GExp ::= & AggOp \ [ForEnum] \\
\text{Parameter List } ParamList ::= & Exp(,Exp)* \\
\text{Qualifier Expression } QExp ::= & \ EExp \\
& | \texttt{not}\ QExp \\
& | QExp \ (\ \texttt{and} \mid \texttt{or} \mid \texttt{xor}\ )\ EExp \\
\text{Enumeration Exp } \ EExp ::= & \ \texttt{Id in}\ [B, \ B] // enumerate\ numbers \\
& | \texttt{Id in Id} \ // \ list\ elements \\
& | '(' \ Elem(,Elem)*')' \ \texttt{in Id} \ // \ tuples \\
\text{Element } Elem ::= & \texttt{Id}|'?' \\
\text{Binary Operator } BinOp ::= & + \ | \ - \ | \ * \ | \ / \ | \ \ldots \\
\text{Aggregation Operator } AggOp ::= & summation \mid product \mid min \mid max
\end{aligned}
$$