

# Hexaly Optimizer

## (Hexaly Optimizer12.5)

### 使用手引書

MSI 株式会社

2024/ 3 /1

## 目次

1. Hexaly Optimizer とは.....	2
2. Hexaly Optimizer の実行方法(稼働確認).....	2
2.1 LSP ファイルの実行 .....	3
2.2 実行結果 .....	4
3. Hexaly Optimizer による 定式化 .....	6
3.1 LSP モデリングの考え方 .....	6
3.2 LSP での bool 変数の定義.....	7
3.3 LSP モデル .....	7
4. LSP 言語 .....	8
4.1 Function.....	8
4.2 演算子 .....	9
4.3 言語機能 .....	9
4.4 主なエラーメッセージ .....	12
【付録1】 演算子一覧表 .....	17
【付録2】 Hexaly Optimizer12.5 変更事項 .....	20
【付録3】 サンプルプログラムと実行結果 LSP 言語比較.....	23
【付録4】 LSP 言語 BNF Syntax .....	25

## 1. Hexaly Optimizer とは

Hexaly Optimizer は、実行するプログラム名としては localsolver.exe となっている。プログラム名を変更しないのは、既存の資産の延長として、最適化システムを踏襲するためである。最新の Hexaly Optimizer12.5 は、大規模最適化問題を実用時間内で効率よく求めることを目的とした新しいアプローチの All-In-One ソルバーであり、従来の数理計画法システムを集大成したものとなっている。とくに、スケジューリング問題、大規模組み合わせ問題に対して、大幅な性能向上を実現している。本書では、プログラム名 Hexaly Optimizer として、使用説明をすることにする。

特に、最新の Hexaly Optimizer12.5 は、大規模スケジューリング問題に対する性能強化を実現している。

Hexaly Optimizer はフランスの 6 人の若手 OR 実践者が 10 年の歳月をかけて開発したものであり、All-In-One Solver として、LP (線形計画法)、MIP (混合整数計画法)、CP(制約論理プログラミング)、NLP (非線形計画法)、LP (線形計画法) 問題を解くことができる。特に、MIP、CP では、現実的には解けなかった大規模組み合わせ最適化問題に対して、最新の各種解法を融合させて進化した解法で解くことを実現している。

Hexaly Optimizer の高機能、新機能については、<https://www.HexalyOptimizer.com/> に説明してある。

## 2. Hexaly Optimizer の実行方法 (稼働確認)

**Hexaly Optimizer 12.5 は PC, Unix, MAC の 64bits モードで稼働する**

Hexaly Optimizer は C++で開発されており、実行プログラムを単体で動かすだけでなく、C++、Java、C#、Python のプログラムから呼ぶことができる。

Hexaly Optimizer はインタプリタ型で直接実行することができるモデリング言語

(LSP) を持っている。インタープリター型で Hexaly Optimizer を実行する場合には、最新の関数型プログラミングをベースとした LSP ファイル (xxx.lsp) を作成し、

パラメータとして LSP ファイルを指定するだけで実行可能である。

また、Hexaly Optimizer が提供するクラスライブラリを使うことで、効率よく、C++、

Java、C# (.net)、python で、数理計画法システムのアプリケーションを開発することが可能である。

本ドキュメントでは、PC (64bits 版) で DOS のコマンドプロンプトから Hexaly Optimizer を直接実行する例を説明する。

Hexaly Optimizer を実行するためには、Hexaly Optimizer をインストールし、ライセンスを獲得する必要がある。その手順については、以下を参照されたい。

<https://www.msi-jp.com/Hexaly Optimizer/download/>

## 2.1 LSP ファイルの実行

インストールされた内容を示す。

Hexaly Optimizer\_12\_5 フォルダ

- | - bin : 実行プログラム localsolver.exe、Python 用ファイル
- | - docs : 説明書、C++、C#、Java 等のクラスライブラリの説明
- | - examples : 例題集 (lsp 言語、C++、C#、Java、Python)
- | - include

license.dat : 自分の PC 用のライセンスキーに置き換える。

TERMS\_AND\_CONDITIONS

### 1) 実行手順

- DOS コマンドプロンプトを立ち上げる。
- LSP ファイルが入っているフォルダにディレクトリを移す。

例) `cd c:\Hexaly Optimizer_12_5\examples\toy`

- LSP ファイル名を指定して、localsolver.exe を実行する。

例) `C:\Hexaly_12.5\bin>localsolver toy.lsp lsTimeLimit=1`

※LSP ファイル名 : toy.lsp。

※パラメタ lsTimeLimit は実行時間を 1 秒と指定。

※Hexaly Optimizer\_12\_5 のパスが切れていない場合、\bin フォルダに toy.lsp をコピーして実行させることができます。

## 2) LSP ファイルの例

ここでは、examples\toy の LSP ファイル:toy.lsp を示す。

Toy モデルは、ナップサック問題である。

品物が、8品あり、それぞれの重さと価値を以下に定義する。

重さ : 10、60、30、40、30、20、20、2 kg

価値 : 1、10、15、40、60、90、100、15 円

ナップサックには最大 102kg まで品物を入れることができ、価値が最大になるよう、どの品物を選べばよいか、その時の価値はいくらになるかが問題である。

LSP による定式化を以下に示す (toy.lsp ファイルの内容)。

```
/* toy.lsp */

/* Declares the optimization model. */
function model() {

  // 0-1 decisions
  x_0 <- bool(); x_1 <- bool(); x_2 <- bool(); x_3 <- bool();
  x_4 <- bool(); x_5 <- bool();
  x_6 <- bool(); x_7 <- bool();

  // weight constraint
  knapsackWeight <- 10*x_0 + 60*x_1 + 30*x_2 + 40*x_3 + 30*x_4 + 20*x_5 +
  20*x_6 + 2*x_7;
  constraint knapsackWeight <= 102;

  // maximize value
  knapsackValue <- 1*x_0 + 10*x_1 + 15*x_2 + 40*x_3 + 60*x_4 + 90*x_5 +
  100*x_6 + 15*x_7;
  maximize knapsackValue;
}

/* Parameterizes the solver. */
```

```
function param() {  
    lsTimeLimit = 1;  
}
```

※赤字は予約語。この例では、model() 関数で、データと制約式、目的関数を定義している。

※bool()が意志決定変数を意味し、bool()で定義された変数の0と1の値の組合せを高速に評価することで、最適解を求める。

## 2.2 実行結果

2.1 で示した toy モデル (toy.lsp ファイル) の実行結果を以下に示す。

```
C:\Hexaly_12.5\bin>localsolver toy.lsp  
Hexaly Optimizer 12.5.20240129-Win64. All rights reserved.  
Load toy.lsp...  
Run model...  
Run param...  
Run solver...  
  
Model: expressions = 38, decisions = 8, constraints = 1, objectives = 1  
Param: time limit = 10 sec, no iteration limit  
  
[objective direction ]: maximize  
  
[ 0 sec,  0 itr]:      0  
[ optimality gap   ]: 100.00%  
[ 0 sec, 1402 itr]:   280  
[ optimality gap   ]:    0%  
  
1402 iterations performed in 0 seconds  
  
Optimal solution:  
obj  =    280  
gap  =     0%  
bounds =    280  
  
C:\Hexaly_12.5\bin>
```

※ちなみに答えは、

$x_2(30, 15)$ ,  $x_4(30, 60)$ ,  $x_5(20, 90)$ ,  $x_6(20, 100)$ ,  $x_7(2, 15)$  の 5 個であり、重さの合計は 102kg、価値の合計は 280 円となる。

添付資料 4 に `toy2.lsp` として、`output function` を使って答えを出力した例を示す。

実行ログには、以下が示される。

—1323 iterations performed in 0 seconds

—Optimal solution:

- `obj` = 280 : 目的関数値 (最適解)
- `gap` = 0% : 上界値とのギャップ (最適解であることを示す)
- `bounds` = 280 : 上界値

計算終了時 (時間指定等で打ち切られた場合等) の最終解状態を、以下の 3 つで出力する。

- `infeasible` : 実行不可能解
- `feasible` : 実行可能状態
- `optimal` : 最適解

Hexaly Optimizer12.5 では、解空間が凸の場合には、解の上界または下界を計算し、最適解とのギャップを表示する。

### 3. Hexaly Optimizer による定式化

Hexaly Optimizer の定式化は意志決定変数を定義することから始まる。

意志決定変数には(bool)として定義した 0-1 変数、上下限を持つ整数変数(int)、上下限を持つ実数変数(float)、変数セット (list) として定義する変数セットの組合せからなり、意思決定変数の値を変化させ、超高速に解の探索を試行することで、大規模最適化問題を実用的に解くことが基本の考え方である。

bool 変数の数がたとえ 1000 万変数を超えても実用的な意味で解を求めることができる。bool 変数で定義した意志決定変数の組合せが解となる。解を構成する変数を一組だけ変化させ解を探索していく。ため、Hexaly Optimizer 用に定式化するためには、意思決定変数を使って、目的関数、制約関数を定義する必要がある。

ナップサック問題のように条件に合う品物を選ぶのが目的であれば、品物を選ぶか選ばないかを bool 変数として定義すれば良い。ある品物を選ばれた場合、bool 変数が 1 を取ることを想定すれば、品物の重さ及び価値を直接計算できるため、bool 変数の組合せで制約条件、目的関数を評価することができる。

Hexaly Optimizer12.0 では、従来の MIP 問題のように、上下限のある整数変数を意思決定変数として定義することができる。最適化計算途中にマテリアルバランスを求め、どこで何をいくつ作る／使うかを答えとして求めることができる。この場合、拠点配置や設備投資を決める固定費問題として、生産数量等の変数を float 変数として定義し、工場数を制限する場合生産数量を決める意志決定変数の数を LSP で制約条件として定義すればよい（従来は bigM を使用して関連付ける必要があったが技巧的な定式化は不要である）。

**※ただし、既存の MIP 問題の定式化で工場制約と倉庫制約を別々の意思決定変数で定義していた場合には、解探索が必ずしも速くならないため、Hexaly Optimizer に適した問題定義が重要である。**

#### 3.1 LSP モデリングの考え方

以下の手順でモデリングを行う。

- 1) 意志決定変数(bool 変数及び float 変数等)を定義する。
- 2) 上記の意思決定変数をすべて使って、制約条件、目的関数を定義する。

この時、MIP のように、線形制約にこだわる必要はなく、制約条件、目的関数ともに、非線形表現が可能である。

### 3.2 LSP での bool 変数の定義

以下に典型的な問題毎に bool 変数の定義イメージを示す。

- ナップサック問題 :  $X_p$  ( $p$  : 品物)
- ルート選択問題 :  $X_r$  ( $r$  : ルート)
- 裁断計画問題 :  $X_{p,q}$  ( $p$  : 裁断パターン、 $q$  : パターンの使用回数)
- 人員配置問題 :  $X_{p,t,j}$  ( $p$  : 人員、 $t$  : 時間、 $j$  : ジョブ)
- 車両投入計画 :  $X_{c,p}$  ( $c$  : 車両、 $p$  : ポジション (順番))
- SCM :  $X_{t,i,j,k,p}$  ( $t$  : 期、 $i$  : 工場、 $j$  : ライン、 $k$  : 倉庫、 $p$  : 製品)
- スケジューリング :  $X_{t,i,j,p}$  ( $t$  : 時間等、 $i$  : 工場、 $j$  : ライン、 $p$  : 製品等)

,

Hexaly Optimizer では数百万以上の bool 変数を定義することができ、MIP の定式化と違い、オーダ単位に意思決定変数(bool)を定義することで、より、自然な形での定式化が可能となる。

Hexaly Optimizer12.0 は、最初に事前解析で実行可能解や上界値 (下界値) を求めるため、利用者が実行不可能性または実行可能性を考慮する必要はない。

### 3.3 LSP モデル

LSP モデルは、以下の要素から構成される。

- 意志決定変数 : `bool()`、`float` (下限値、上限値)、`int` (下限値、上限値)、`list()`
- 副生変数 : 任意の変数であり、プログラミングをわかりやすくすることができる。  
変数の定義には、`<-` を使用する。
- 制約 : `constraint` (予約語) で、制約条件を定義する。  
`constraint` 制約式で定義された値が実行可能性の判定で使用される。
- 目的関数 : `minimize` (予約語) または `maximize` (予約語) で目的関数を定義する。目的関数は複数定義可能であり、定義された順番に最適化を行う。目的計画法として利用可能である。

## 4. LSP 言語

LSP 言語は、最適化問題をモデル化し、モデルの検証及び解の検証を行うフェーズでの試行錯誤を行うのに最適な環境を提供することを目的として開発されている。

LSP 言語は、最新の関数型プログラミング言語である。関数型プログラミング言語の特長は、型推論を備えた言語であるため、Java や C 言語と異なり、コンパイラが自動的にデータの種別を推定するため、データの種別（型）をプログラマが指定する必要がない。その結果、プログラムの記述は Ruby など軽量言語のように簡潔である。

軽量言語では実現できないコンパイラによるプログラムのチェックが可能になる点にある。

LSP 言語の特徴は以下：

- －迅速に開発できる（開發生産性が良い。従来に比べて、1/5 から 1/2 の開発量）
- －バグを抑えやすい（コンパイラが型の間違い等を自動的にチェックする）
- －アプリケーションの性能を向上させやすい
- －簡潔かつシンプルなモデリング言語（できるだけ省略できるよう設計）
- ※大規模問題でも制約条件及びデータがそろっていれば、1 日でモデリングと実行が可能である。
- －作成(修正)←→実行が同時にできる（一つはエディタ、もう一つは DOS コマンドプロンプトの二つのウィンドウを操作しながら開発が可能である。
- －目的計画法のように目的を段階的に設定することができるため、モデルの開発及び解の検証を段階的に行うことができる。

添付資料 3 に lsp 言語と C++、Java、C# で記述した例を示す。

### 4.1 Function

LSP にはメインプログラムがなく、以下の 5 つの基本的なファンクションからなる。

`function model()` は必須であるが、その他は必要に応じて使用すればよい。

また、以下の基本的なファンクションを使用することができる。

- `input`: for declaring your data or reading them from files.
- `model`: for declaring your optimization model.
- `param`: for parameterizing the local-search solver before running.
- `display`: for displaying some info in console or in some files during the resolution.

- output: for writing results in console or in some files, once the resolution is finished.

## 4.2 演算子

LSP モデルの中で、自由に使用できる。とくに、目的関数、制約条件の記述に利用でき、非線形制約、非線形目的関数として利用可能である。演算子には、以下の種類がある。詳細は一覧表を参照されたい。

- 算術演算子 (sum、min、max、sin、cos、log、exp 等)
- 論理演算子 (not、and、or、xor)
- 関係演算子 (==、!=、<=、>=、<、>)
- 複合演算子 (if、array+at)

## 4.3 言語機能

### 1) 変数定義

変数の定義の例を示す。以下はすべて有効である。

```
a = true;    // a = 1
b = 9;       // b = 9

c = a + b;   // c = 10
c = a * b;   // c = 9

c = a == b;  // c = 0
c = a < b;   // c = 1
```

### 2) 配列定義

Hexaly Optimizer の配列は map で定義することができる。

Map は、値とキーを併せ持ったデータ構造になっている。キーは、整数であり、かならずしも連続的である必要はない。値は、どんなタイプでも可能であり、キーに対応させるために、[ブラケット]表記法を用いる。

```
a = map("z", 9); // a[0] = "z", a[1] = 9
a = {"z", 9};    // a[0] = "z", a[1] = 9
a["a"] = "abc"; // a[0] = "z", a[1] = 9, a["a"] = abc
```

### 3) 条件判定

条件判定は、if 文を使用する。記述形式は、以下：

```
if (C) S_true; else S_false;
```

または、`? :` で簡潔に記述することもできる。

```
if (1 < 2) c = 3; else c = 4;
c = 1 < 2 ? 3 : 4;

if (0) c = "ok";
if (true) c = "ok";
if (2) c = "error"; // ERROR: invalid condition

c = 0 * 9; // c = 0
if (c) {
    a = "L";
    b = 0;
} else { // executed block
    a = "S";
    b = 1;
}
```

### 4) 繰り返し

繰り返しには、while と for がある。

While は、以下で記述する。C が真である限り、S が実行される。

```
do S; while (C);
```

for は、以下で記述する。v が V にある限り、S が実行される。

```
for [v in V] S;
```

また、キーと値がセットの場合には、以下で記述する。

```
for [k,v in M] S;
```

```
for [i in 0..2] a[i] = i + 1; // a[0] = 1, a[1] = 2, a[2] = 3
s = 0; for [v in a] s = s + v; // s = 6
s = 0; for [k,v in a] s = s + k + v; // s = 9
```

```
for[i in 0..9]
  for [j in i+1..9]
    for [k in j+2..9]
      a[i][j][k] = i + j + k;
for[i in 0..9][j in i+1..9][k in j+2..9] // compact
  a[i][j][k] = i + j + k;

for[i in 0..9][j in i+1..9][k in j+2..9]
{
  a[i][j][k] = i + j + k;
  b[i][j][k] = i * j * k;
}
```

## 5) 繰り返し演算

繰り返し演算は、以下で記述する。

```
for [v in V] a[v] = f(v);
```

LSP では、以下の省略形で記述可能である。

```
a[v in V] = f(v);
```

```
for[i in 0..9][j in i+1..9][k in j+2..9]
  a[i][j][k] = i + j + k;

a[i in 0..9][j in i+1..9][k in j+2..9] = i + j + k;

x[i in 0..n-1][j in 0..m-1] <- bool();.
```

## 6) 関数

LSP では、任意に関数を定義できる。関数の値は、0 (false) または 1 (true) でも良いし、数値でも良い。LSP プ

ログラムは通常、function 間で共通の変数定義

(global) になっているため、function 内でローカルに使用したい場合には、local の宣言子でローカル変数であることを定義する必要がある。

```
function isEven(v) {
  if (v % 2 == 0) return true;
  else return false;
}

function computeSumOfEvenNumbers(a,b) {
  local total = 0;
  for [v in a..b : isEven(v)]
    total = total + v;
  return total;
}
```

#### 4.4 主なエラーメッセージ

コマンド・ラインで絶対に必要な引数は、lsp ファイルの名前です。もし lsp ファイルが利用できないならば、エラーを出す。また、コマンド・ラインの他の全ての引数 (パラメータ等) は、フォーマット identifier=value を持たなければならない。

- <f> doesn't exist or is not accessible. // LSP file
- Invalid argument format for <arg>. Expected format : identifier=value.

LSP は型を強く意識した言語であり、関数のパラメータには正しい型が必要である。

- Function <f> cannot handle argument of type <t>. Argument of type <t2> is expected.
- Function <f> takes <x> argument(s) but <y> were provided.
- Function <f> : <T> expression expected for argument <i>.

同様に、型チェックで不適切な型の場合には、エラーメッセージを出力する。

- Cannot apply <opName> operator on type <T>.
- Cannot apply <opName> operator between types <T1> and <T2> .

Cannot cast <T1> to <T2>.

- Cannot apply ternary operator '?' on given operators : incorrect argument type.
- Cannot cast 'nil' to <T>. A variable or a map element may not be assigned. いくつかの関数は引数を持つ。もし、引数の数が合わない場合には、エラーを出力する。 :

- Function <f> takes at least <x> argument(s) but <y> were provided.
- Function <f> takes at most <x> argument(s) but <y> were provided.

関数を使う時、関数が未定義であれば、エラーメッセージを出力する。また、既存の関数と同じ名前の関数を定義するのもエラーである。

変数にかんしては、自由に再定義（再利用）可能である。ただし、局所変数だけは、同じ名前でも二回使うことはできない。もし、変数が値を持たない場合は、nil の値を持つ。

- Function <f> already defined.
- Function <f> undefined.
- Variable <name> already defined.

Input/output 関数は、指定されたファイルの入出力チェックを行う。

- File <f> cannot be opened.
- Cannot read from file <f>.
- Cannot write to file <f>.

数字または文字列を入力時に、プログラムとデータが一致しない（データ数、タイプ等が一致しない）場合およびファイルの最後まで読んだ場合にはエラーを出力する。

- Cannot convert the current token to int.
- Cannot convert the current token to double.
- End of file: no more line to read from file <f>.
- End of file reached.

文字列の操作では、文字列が空でないこと及びインデックスが許容範囲内であることが必要である。

- The given index for substring is out of range. Min value: 0, Max value: <len>.
- Number of characters for substring must be greater than 0.
- Search string is empty.

マップの制限として二つある。

- キーは整数または文字であること
- イタレーション中（連続して探索計算している間）は、マップを変更してはならない:

- 'nil' provided as key for a map. The key variable may not be assigned.
- Only types 'string' and 'int' are allowed for keys in maps.
- Cannot iterate on a modified map.

LSP モデルに対してパラメータで数値を指定する場合には、許容範囲の数値でなければならない。

- The objective bound must be an integer, a double or a boolean for objective <objIndex>
- The objective bound must be an integer or a boolean for objective <objIndex>
- The number of threads cannot exceed 1024.
- The annealing level size must be an integer between 0 and 9.
- Advanced parameter <key> does not exist.

モデルで式または変数を表現する場合には、<- で宣言する必要がある。局所変数を <- を使って宣言することは出来ない。

モデルには、必ず目的関数が必要。また、目的関数は数式で表現する必要がある。 マップ等は使用できない。制約式は、バイナリ表現でなければならない。

- Cannot assign Hexaly Optimizer expressions to local variables.
- At least one objective is required in the model.
- Only boolean expressions can be constrained.

- Only expressions with a value can be added in the objectives list.

setValue 関数は、意志決定変数 (bool 変数) にのみ初期値を与えることができる。

- The only allowed values are 0 or 1.

$x \leftarrow a[y]$  の数式表現では、マップとしてゼロから連続した整数キーが必要となる。また、バリューとして、キーの数ぶんだけ、数値データまたは LS 表現が必要である。

- All keys must be integers. Type found:  $\langle T \rangle$
- Values must be integers, booleans or expressions. Type found:  $\langle T \rangle$
- The first key must be 0. Key found:  $\langle \text{key} \rangle$
- Keys are not in a continuous range. Next key expected  $\langle \text{key1} \rangle$ . Key found:  $\langle \text{key2} \rangle$ .

探索の間に、変数値が制約を満足しない場合が発生することに注意されたい。例えて言えば、実行可能状態の時に、実行可能解を探索中に起きるケースがある。ゼロ割や配列オーバーフローが起きた時であり、割算の分母がゼロまたはインデックスが範囲外になったことを意味する。  $z \leftarrow x/y$  のような場合には、  $z \leftarrow x/\max(1,y)$  と表現することが望ましい。

- Division by zero.
- Index out of bounds for 'at' operator (index:  $\langle \text{indexId} \rangle$ , array size:  $\langle n \rangle$ )

■

以上

## 【付録 1】 演算子一覧表

### 使用可能な演算子と関数の表

以下の表では、各演算子が LSP 言語の名前で識別されている。Python、C++、C#、または Java では、これらの名前は各言語のコーディング規則と予約済みキーワードを尊重している。

- C++ と Java では、決定には "Var" という接尾辞が付く (boolVar、floatVar、intVar、setVar と listVar)
- C# では、すべての関数が大文字で始まる

	Function	Description	Arguments type	Result type	Arity Symb
Decisional	bool	Boolean decision variable with domain {0,1}	<i>none</i>	bool	0
	float	Float decision variable with domain [a, b]	<i>2 doubles</i>	double	2
	int	Integer decision variable with domain [a, b]	<i>2 integers</i>	int	2
	interval	Interval decision variable with domain [minStart, maxEnd]	<i>2 integers</i>	interval	2
	list	Ordered collection of integers within a range [0, n - 1]	<i>1 integer</i>	collection	1
	set	Unordered collection of integers within a range [0, n - 1]	<i>1 integer</i>	collection	1
Arithmetic	sum	Sum of all operands	bool, int, double	int, double	n >= 0 +
	sub	Substraction of the first operand by the second one	bool, int, double	int, double	2 -
	prod	Product of all operands	bool, int, double	int, double	n >= 0 *
	min	Minimum of all operands	bool, int, double	int, double	n > 0
	max	Maximum of all operands	bool, int, double	int, double	n > 0
	div	Division of the first operand by the second one	bool, int, double	double	2 /
	mod	Modulo: $\text{mod}(a, b) = r$ such that $a = q * b + r$ with $q, r$ integers and $r < b$ .	bool, int	int	2 %
	abs	Absolute value: $\text{abs}(e) = e$ if $e \geq 0$ , and $-e$ otherwise	bool, int, double	int, double	1

	Function	Description	Arguments type	Result type	Arity	Symb
	dist	Distance: $\text{dist}(a, b) = \text{abs}(a - b)$	bool, int, double	int, double	2	
	sqrt	Square root	bool, int, double	double	1	
	cos	Cosine	bool, int, double	double	1	
	sin	Sine	bool, int, double	double	1	
	tan	Tangent	bool, int, double	double	1	
	log	Natural logarithm	bool, int, double	double	1	
	exp	Exponential function	bool, int, double	double	1	
	pow	Power: $\text{pow}(a, b)$ is equal to the value of $a$ raised to the power of $b$ .	bool, int, double	double	2	
	ceil	Ceil: round to the smallest following integer	bool, int, double	int	1	
	floor	Floor: round to the largest previous integer	bool, int, double	int	1	
	round	Round to the nearest integer: $\text{round}(x) = \text{floor}(x + 0.5)$ .	bool, int, double	int	1	
	scalar	Scalar product between 2 arrays.	array	int, double	2	
	piecewise	Piecewise linear function product between 2 arrays.	array, int, double	double	3	
Logical	not	Not: $\text{not}(e) = 1 - e$ .	bool	bool	1	!
	and	And: equal to 1 if all operands are 1, and 0 otherwise	bool	bool	n >= 0	&&
	or	Or: equal to 0 if all operands are 0, and 1 otherwise	bool	bool	n >= 0	
	xor	Exclusive or: equal to 0 if the number of operands with value 1 is even, and 1 otherwise	bool	bool	n >= 0	
Relational	eq	Equal to: $\text{eq}(a, b) = 1$ if $a = b$ , and 0 otherwise	bool, int, double	bool	2	==
	neq	Not equal to: $\text{neq}(a, b) = 1$ if $a \neq b$ , and 0 otherwise	bool, int, double	bool	2	!=
	geq	Greater than or equal to: $\text{geq}(a, b) = 1$ if $a \geq b$ , 0 otherwise	bool, int, double	bool	2	>=
	leq	Lower than or equal to: $\text{leq}(a, b) = 1$ if $a \leq b$ , 0 otherwise	bool, int, double	bool	2	<=
	gt	Strictly greater than: $\text{gt}(a, b) = 1$ if $a > b$ , and 0 otherwise. In case of intervals: $\text{gt}(a, b) = 1$ if $\text{start}(a) \geq \text{end}(b)$ , and 0 otherwise.	bool, int, double, interval	bool	2	>

	Function	Description	Arguments type	Result type	Arity	Symb
	lt	Strictly lower than: $lt(a, b) = 1$ if $a < b$ , and 0 otherwise. In case of intervals: $lt(a, b) = 1$ if $end(a) \leq start(b)$ , and 0 otherwise.	bool, int, double, interval	bool	2	<
Conditional	iif	Ternary operator: $iif(a, b, c) = b$ if $a$ is equal to 1, and $c$ otherwise	bool, int, double	bool, int, double	3	?:
Set related	count	Returns the number of elements in a collection.	collection, interval, array	int	1	
	indexOf	Returns the index of a value in a collection or -1 if the value is not present.	collection, int	int	2	
	contains	Returns 1 if the collection contains the given value or 0 otherwise.	collection or interval, int	bool	2	
	partition	Returns true if all the operands form a partition of their common domain.	collection	bool	$n > 0$	
	disjoint	Returns true if all the operands are pairwise disjoint.	collection	bool	$n > 0$	
	cover	Returns true if all the operands form a cover of their common domain.	collection	bool	$n > 0$	
	array	Creates an array of fixed or variadic size.	bool, int, double, array, list, set	array	$n \geq 0$	
	at	Returns the value in an array or a list at a specified position.	array, list, int	bool, int, double	$n \geq 2$	[]
	find	Returns the position of the collection containing the given element in the array.	array, int	int	2	
	sort	Returns the array sorted in ascending order. When used with two arguments, the array is sorted based on the values returned by the lambda.	array, lambda	array	1 or 2	
Interval related	start	Returns the start of a non-void interval.	interval	int	1	
	end	Returns the end of a non-void interval.	interval	int	1	
	length	Returns the length of a non-void interval, equivalent to $end(interval) - start(interval)$ .	interval	int	1	
Other	call	Call a function. It can be used to implement your own operator.	bool, int, double	double	$n > 0$	

## 【付録 2】 Hexaly Optimizer12.5 変更事項

### リリースノート

- 新しい演算子 *INTERSECTION* が追加された。
- 新しい演算子 *DISTINCT* が追加された。
- *SORT* 演算子は、キー関数をオプションの 2 番目として受け入れるようになった。
- C# API が macOS arm64 と Linux arm64 で使用できるようになった。

### モデリング演算子

#### Intersection

交差演算子は 2 つのオペランドを受け入れる。この 2 つのオペランドは、配列またはコレクションである。この演算子は、両方のオペランドに存在する値であり、この演算子を使用すると、複数の値をセットまたはリストに含める。

#### Distinct

Distinct 演算子は、1 つまたは 2 つのオペランドを受け入れる。オペランドが 1 つの場合、演算子 配列またはコレクションを受け取り、順序付けされていない個別の値のセットを配列内に返す。オペランドが 2 つある場合は、コレクション、間隔、または配列とラムダ関数を呼び出し、順序付けされていない個別の値のセットを返す。各要素で呼び出された関数によって返されるすべての値の中で、指定されたコレクションまたは間隔であり、この演算子は、スケジューリングをモデル化するために使用できる。機械で生産される製品の種類が限られている問題に使用可能。

#### Sort

sort 演算子は 2 つのオペランドを受け入れるようになった。ラムダ引数が指定されている場合 第 2 引数として、入力配列をソートするためのキーとして使用される。

#### Array

配列演算子は、リストを第 1 オペランドとして受け取り、ラムダ関数を 第 2 オペランド。ラムダ関数は、list を呼び出し、配列を返す。

#### Nary 演算子

N 演算子 (max、min、sum、prod、and、or、xor) は、新しいタイプのオペランドを処理する。範囲とコレクションに加えて、nary 演算子は配列を次のように受け取ることができるようになった。第 1 オペランドとラムダは、第 2 オペランドとして機能する。最後に、彼

らオペランドは、コレクションまたは 1D 配列の 1 つだけを取ることができるようになった。オペレーターは、は、単一オペランドの各要素に適用される。

## API の変更

### Python

- 演算子を追加した。
- 交差式を作成するために追加された。
- 演算子を追加した。
- 個別の式を作成するために追加された。
- 値が指定された位置はコレクションである。

### C++

- 演算子を追加した。
- 交差式を作成するために追加された。
- 演算子を追加した。
- 個別の式を作成するために追加された。
- 値が指定された位置はコレクションであり、コレクション値を取得する新しいメソッドは与えられた位置である。

### C#

- 演算子を追加した。
- 交差式を作成するために追加された。
- 演算子を追加した。
- 個別の式を作成するために追加された。
- 値が指定された位置はコレクションであり、コレクション値を取得する新しいメソッドは与えられた位置である。

### Java

- 演算子を追加した。LSOperator. Intersection
- 交差式を作成するために追加された。LSModel. Intersection
- 演算子を追加した。LSOperator. Distinct
- 個別の式を作成するために追加された。LSModel. distinct
- 値が指定された位置はコレクションであり、コレクション値を取得する新しいメソッドは与えられた位置。LSArray. isCollectionLSArray. getCollectionValue

## API 要件の変更

### Python

- Python 3.4 と Python 3.5 のサポートを終了した。

### C#

- macOS arm64 および Linux arm64 との互換性が追加された。
- macOS x64 および Linux x64 には .NET 6.0 (またはそれ以上) が必要になった。

## プラットフォームの依存関係の変更

### Linux

- libc 2.28 (またはそれ以上) および libstdc++ 6.0.25 (またはそれ以上) が必要ある。

## 【付録 3】 サンプル 2 (lsp と実行結果)

### 付録3. 1 lspプログラム (toy モデルに関数:output を追加)

```
/****** toy2.lsp *****/
function model()
{
  nbProducts = 8;
  value = {1,10,15,40,60,90,100,15};
  // 0-1 decisions
  x[i in 0..nbProducts-1] <- bool();
  // weight constraint
  knapsackWeight <- 10*x[0] + 60*x[1] + 30*x[2] + 40*x[3] + 30*x[4] + 20*x[5] +
20*x[6] + 2*x[7];
  constraint knapsackWeight <= 102;
  // maximize value
  knapsackValue <- 1*x[0] + 10*x[1] + 15*x[2] + 40*x[3] + 60*x[4] + 90*x[5] +
100*x[6] + 15*x[7];
  maximize knapsackValue;
}

function output()
{
  println("Selected Products:");
  for [i in 0..nbProducts-1 :x[i].value == 1]   println("#"+i+" ("+"value[i]+")");
}
```

### 付録3. 2 lsp実行結果(toy モデルに関数:output を追加)

```
C:\Hexaly Optimizer_12_5\bin>Hexaly Optimizer toy2new.lsp
Hexaly Optimizer 12.5.20240129-Win64. All rights reserved.
Load toy2new.lsp...
Run model...
Run solver...
```

```
Model: expressions = 38, decisions = 8, constraints = 1, objectives = 1
Param: no time limit, no iteration limit
```

```
[objective direction ]: maximize
```

```
[ 0 sec,  0 itr]:      0
[ optimality gap   ]: 100.00%
[ 0 sec, 2136 itr]:   280
[ optimality gap   ]:    0%
```

```
2136 iterations performed in 0 seconds
```

```
Optimal solution:
```

```
obj  =    280
gap  =     0%
bounds =    280
```

```
Run output...
```

```
Selected Products:
```

```
#2 (15)
#4 (60)
#5 (90)
#6 (100)
#7 (15)
```

```
C:\Hexaly Optimizer_12_5\bin>getValue(x[i])"
```

## 【付録4】BNF Syntax(バックス・ナウア記法)

BNF の表記は次のような導出規則の集合である。

```
<symbol> ::= <expression with symbols>
```

左辺の<symbol>は単一の記号である。また、<expression with symbols> は記号列、または選択を表すバーティカルバー「|」で区切られた記号列であり、左辺の `<symbol>` の置換となるものを表している。なお、導出規則で使用された記号は「非終端記号」と「終端記号」に分類される。導出規則群の左辺に現れた記号は「非終端記号」と呼ばれ、いずれの導出規則の左辺にも現れなかった記号は「終端記号」と呼ばれる。

以下に LSP 言語の BNF Syntax を示す。

※<> ::= <>を省略してある。

```
identifier
```

```
  : simple_identifier  
  | contextual_keyword
```

```
contextual_keyword
```

```
  : 'pragma'  
  | 'as'  
  | 'from'  
  | 'extends'  
  ;
```

```
expression
```

```
  : ternary_expression  
  | lambda_expression  
  | table_expression  
  ;
```

```
expression_no_range
```

```
  : ternary_expression_no_range  
  | lambda_expression  
  | table_expression  
  ;
```

```
ternary_expression_no_range
```

```
  : or_expression_no_range  
  | or_expression_no_range  
    '?'ternary_expression_no_range  
    ':'ternary_expression_no_range
```

```

;

or_expression_no_range
  : and_expression_no_range
  | or_expression_no_range '||' and_expression_no_range
  ;

and_expression_no_range
  : equality_expression_no_range
  | and_expression_no_range '&&' equality_expression_no_range
  ;

equality_expression_no_range
  : relational_expression_no_range
  | equality_expression_no_range '=='
relational_expression_no_range
  | equality_expression_no_range '!='
relational_expression_no_range
  ;

relational_expression_no_range
  : additive_expression
  | relational_expression_no_range 'is' additive_expression
  | relational_expression_no_range 'is' 'nil'
  | relational_expression_no_range 'is' 'bool'
  | relational_expression_no_range 'is' 'int'
  | relational_expression_no_range 'is' 'double'
  | relational_expression_no_range '<' additive_expression
  | relational_expression_no_range '>' additive_expression
  | relational_expression_no_range '<=' additive_expression
  | relational_expression_no_range '>=' additive_expression
  ;

ternary_expression
  : or_expression
  | or_expression
    '?' ternary_expression
    ':' ternary_expression
  ;

or_expression
  : and_expression
  | or_expression '||' and_expression
  ;

and_expression

```

```

: equality_expression
| and_expression '&&' equality_expression
;

equality_expression
: relational_expression
| equality_expression '==' relational_expression
| equality_expression '!=' relational_expression
;

relational_expression
: range_expression
| relational_expression 'is' range_expression
| relational_expression 'is' 'nil'
| relational_expression 'is' 'bool'
| relational_expression 'is' 'int'
| relational_expression 'is' 'double'
| relational_expression '<' range_expression
| relational_expression '>' range_expression
| relational_expression '<=' range_expression
| relational_expression '>=' range_expression
;

range_expression
: additive_expression
| additive_expression '..' additive_expression
| additive_expression '...' additive_expression
;

additive_expression
: multiplicative_expression
| additive_expression '+' multiplicative_expression
| additive_expression '-' multiplicative_expression
;

multiplicative_expression
: unary_expression
| multiplicative_expression '*' unary_expression
| multiplicative_expression '/' unary_expression
| multiplicative_expression '%' unary_expression
;

unary_expression
: function_call_expression
| new_expression
| '+' unary_expression

```

```

| '-' unary_expression
| '!' unary_expression
| 'typeof' unary_expression
;

new_expression
: 'new' member_array_expression arguments
;

function_call_expression
: primary_expression
| super_expression
| function_call_expression '[' expression ']'
| function_call_expression '.' identifier
| function_call
;

member_array_expression
: primary_expression
| super_expression
| member_array_expression '[' expression ']'
| member_array_expression '.' identifier
;

super_expression
: 'super' '.' identifier
;

primary_expression
: assignment_identifier
| 'true'
| 'false'
| 'nan'
| 'inf'
| 'nil'
| string
| integer
| double
| '(' expression ')'
;

lambda_expression
: identifier '=>' block_statement
| function_arguments_declaration '=>' block_statement
| identifier '=>' lambda_body_expression
| function_arguments_declaration '=>' lambda_body_expression

```

```

    | 'function' function_arguments_declaration block_statement
    ;

lambda_body_expression
    : ternary_expression
    | lambda_expression
    ;

table_expression
    : '{' '}'
    | '{' table_list '}'
    ;

table_list
    : expression
    | table_key '=' expression
    | table_key ':' expression
    | table_list ',' expression
    | table_list ',' table_key '=' expression
    | table_list ',' table_key ':' expression
    ;

table_key
    : string
    | identifier
    | integer
    | '-' integer
    ;

function_call
    : function_call_expression arguments
    | function_call_expression variadic_arguments
    ;

arguments
    : '(' ')'
    | '(' function_argument_list ')'
    ;

variadic_arguments
    : variadic_compositor_list '(' function_variadic_list ')'
    ;

function_argument_list
    : expression
    | function_argument_list ',' expression

```

```

;

function_variadic_list
: expression
| function_variadic_list ',' expression
;

variadic_compositor_list
: '[' filter_iterator ']'
| variadic_compositor_list '[' filter_iterator ']'
;

filter_iterator
: identifier 'in' expression ':' expression
| identifier ',' identifier 'in' expression ':' expression
| identifier 'in' expression
| identifier ',' identifier 'in' expression
;

range_iterator
: additive_expression '..' additive_expression
| additive_expression '...' additive_expression
;

statement
: block_statement
| assignment_statement
| local_assignment_statement
| local_statement
| if_else_statement
| for_statement
| while_statement
| dowhile_statement
| continue_statement
| break_statement
| modifier_statement
| throw_statement
| trycatch_statement
| with_statement
| function_call_statement
| new_statement
| return_statement
| super_constructor_statement
| ';'
;

```

```

block_statement
: '{' '}'
| '{' statement_list '}'
;

statement_list
: statement
| statement_list statement
;

assignment_statement
: identifier assignment_operator expression ';'
| assignment_identifier assignment_compositor_list
assignment_operator expression ';'
;

assignment_identifier
: identifier
| 'this'
;

assignment_operator
: '='
| '<-'
| '+='
| '-='
| '/='
| '*='
| '%='
;

assignment_compositor_list
: assignment_compositor
| assignment_compositor_list assignment_compositor
;

assignment_compositor
: '[' filter_iterator ']'
| '[' range_iterator ']'
| '[' expression_no_range ']'
| '.' identifier
;

local_assignment_statement
: 'local' identifier local_assignment_operator expression ';'

```

```

    | 'local' assignment_identifier assignment_compositor_list
local_assignment_operator expression ';'
;

local_assignment_operator
: '='
| '<-'
;

local_statement
: 'local' identifier ';'
;

if_else_statement
: if_condition statement
| if_condition statement statement
;

if_condition
: 'if' '(' expression ')'
;

for_statement
: 'for' for_compositor_list statement
;

for_compositor_list
: '[' filter_iterator ']'
| for_compositor_list '[' filter_iterator ']'
;

while_statement
: 'while' '(' expression ')' statement
;

dowhile_statement
: 'do' statement 'while' '(' expression ')' ';'
;

continue_statement
: 'continue' ';'
;

break_statement
: 'break' ';'
;

```

```

modifier_statement
    : modifier expression ';'
    ;

modifier
    : 'minimize'
    | 'maximize'
    | 'constraint'
    ;

throw_statement
    : 'throw' expression ';'
    | 'throw' ';'
    ;

trycatch_statement
    : 'try' statement 'catch' '(' identifier ')' statement
    ;

with_statement
    : 'with' '(' with_resource ')' statement
    ;

with_resource
    : identifier
    | identifier '=' expression

function_call_statement
    : function_call ';'
    ;

new_statement
    : new_expression ';'

return_statement
    : 'return' ';'
    | 'return' expression ';'
    ;

super_constructor_statement
    : 'super' arguments ';'
    ;

declaration_list
    : function_declaration

```

```

    | class_declaration
    | declaration_list function_declaration
    | declaration_list class_declaration
    ;

function_declaration
    : 'function' identifier function_arguments_declaration
    block_statement
    ;

function_arguments_declaration
    : '(' ')'
    | '(' function_identifier_list ')'
    ;

function_identifier_list
    : identifier
    | function_identifier_list ',' identifier
    ;

class_declaration
    : class_header '{' '}'
    | class_header '{' class_member_list '}'
    ;

class_header
    : 'class' identifier
    | 'class' identifier 'extends compound_name'
    | 'final' 'class' identifier
    | 'final' 'class' identifier 'extends compound_name'
    ;

class_member_list
    : class_member_list class_member
    | class_member
    ;

class_member
    : class_constructor
    | class_method
    | class_field
    | class_static_function
    | class_static_field
    ;

class_constructor

```

```

        : 'constructor' function_arguments_declaration block_statement
        ;

class_method
    : 'override' identifier function_arguments_declaration
block_statement
    | identifier function_arguments_declaration block_statement
    ;

class_field
    : identifier ';'
    | identifier '=' expression ';'
    ;

class_static_function
    : 'static' identifier function_arguments_declaration
block_statement
    ;

class_static_field
    : 'static' identifier ';'
    | 'static' identifier '=' expression ';'
    ;

pragma_list
    : pragma_statement
    | pragma_list pragma_statement
    ;

pragma_statement
    : 'pragma' simple_identifier ';'
    | 'pragma' simple_identifier simple_identifier ';'
    | 'pragma' simple_identifier integer ';'
    | 'pragma' simple_identifier double ';'
    ;

use_list
    : use_statement
    | use_list use_statement
    ;

use_statement
    : TOKEN_USE simple_identifier ';'
    | TOKEN_USE compound_name 'as' simple_identifier ';'
    | TOKEN_USE import_list 'from' compound_name ';'
    ;

```

```

import_list
  : simple_identifier
  | simple_identifier 'as' simple_identifier
  | import_list ',' simple_identifier
  | import_list ',' simple_identifier 'as' simple_identifier
  ;

compound_name
  : compound_name_part

compound_name_part
  : simple_identifier
  | compound_name_part '.' simple_identifier
  ;

header_section
  : pragma_list
  | pragma_list use_list
  | use_list
  ;

program
  : <EOF>
  | declaration_list <EOF>
  | header_section <EOF>
  | header_section declaration_list <EOF>
  ;

```

以上