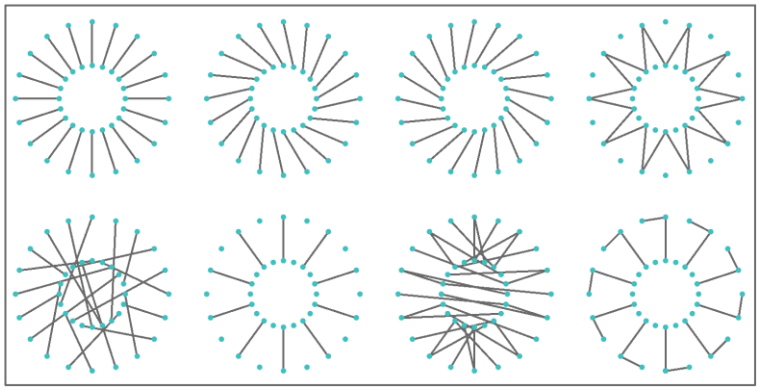


Essential Algorithms and Data Structures

for Computational Design in Grasshopper

First Edition

（日本語版）

Rajaa Issa

Robert McNeel & Associates

目次

はじめに

Essential Algorithms and Data Structures for Computational Design（コンピュテーショナルデザインのために不可欠なアルゴリズムとデータ構造）では，Grasshopper を使用して複雑な 3D モデリングアルゴリズムを開発する上で必要となる効果的な方法論を紹介しています．また，Grasshopper で採用されているデータ構造から，それらの構成，うまく扱うためのツールまで幅広く取り上げています．

この資料は，パラメトリックデザインに関心があるが，プログラミングのバックグラウンドがほとんどまたはまったくないデザイナーや設計者を対象としています．概念はすべて，Rhinoceros® (Rhino)のジェネレーティブモデリング環境である Grasshopper® (GH)を使用して，視覚的に解説しています．Grasshopper のユーザーインターフェースやツールについての初心者向けガイドを意図したものではないため，インターフェースと操作方法の基本的な知識があることを前提としています．その他のリソースや入門ガイドについては，www.rhino3d.com の「学ぶ」のページをご覧ください．

コンテンツは 3 つの章に分かれます．Chapter 1 は，アルゴリズムとデータについての解説です．パラメトリックな手段を用いたプログラムを作成し，うまく使いこなす上で役に立つ詳細な方法論について紹介します．また，データの型，ソース，およびそれらを処理する一般的な方法などのデータの基本的な概念についても紹介しています．Chapter 2 では，Grasshopper の基礎的なデータ構造を確認します．この章では，単一アイテムやリストの内容までを含みます．Chapter 3 では，Grasshopper におけるツリーデータ構造を詳細に確認し，より実用的なデザイン問題に応用してみます．すべての Grasshopper サンプルとチュートリアルは，Rhino のバージョン 6 で作成されており，ダウンロードしたデータに含まれています．

Rajaa Issa

Robert McNeel & Associates

Chapter 1: Algorithms and Data（アルゴリズムとデータ）

アルゴリズムおよびデータは，どのようなパラメトリックデザイン問題でも重要となる 2 つの要素ですが，アルゴリズムを記述はすることはありふれたことではなく，直感的なデザイナーにとって簡単ではないスキルが要求されます．アルゴリズムを用いたデザインプロセスは非常に論理的であり，設計意図とそれらを達成するための手順を明確に記述することが必要となります．この章では，クリエイティブなデザイナーや設計者が新しくアルゴリズムを用いたソリューションを開発するために役立つ方法論について紹介します．また，すべてのアルゴリズムにはデータの扱いが伴うことから，「アルゴリズムとデータ」は密接な関係にあると言えます．そのため，データの型とその処理の基本的な概念についても紹介していきます．

1_1: アルゴリズミックデザイン

アルゴリズミックデザインとは，**明確に定義した手順**を通じて**出力**を生み出すデザイン手法であると定義できます．その意味では，多くの人間の活動はアルゴリズム的です．例えばケーキを焼く場合，**レシピ**（明確に定義された手順）を使用して，**ケーキ**（出力）を作り上げます．**材料**（入力）または焼き方を変えると，異なるケーキが出来上がります．典型的なアルゴリズムを部分別に分析することで，ゼロからアルゴリズムを作成する上での戦略について確認してみましょう．

複雑さに関わらず，すべてのアルゴリズムには，**入力（Input）**，**主要プロセス（Key process）**，**出力（Output）**の 3 つの構成要素があります．ここで，Key process には，追加の入力とプロセスが必要になる場合もあります．

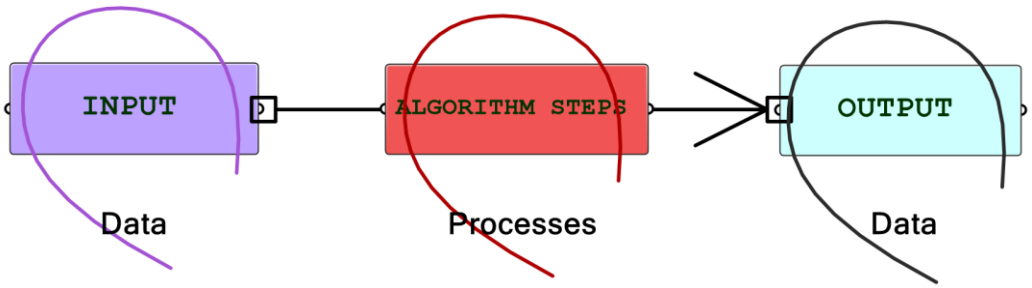


図 (1): アルゴリズムソリューションの構成要素.

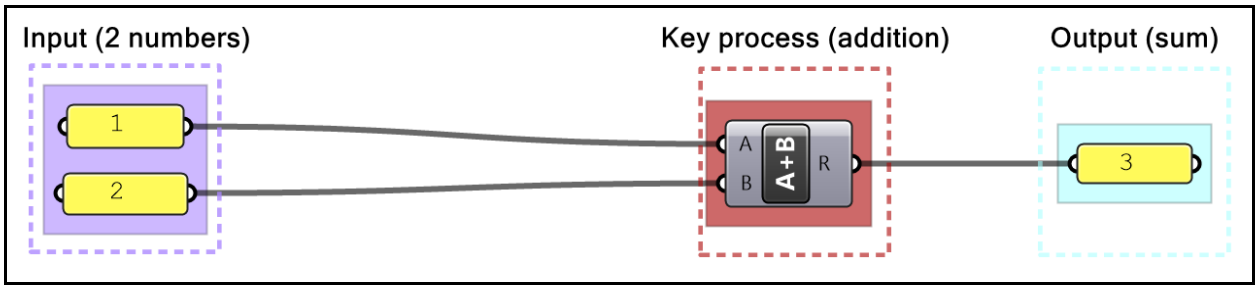
以降この資料では，アルゴリズムを整理してラベルを付け，3 つの要素をわかりやすく区分しています．また，色分けも一貫し，パートを視覚的に区別しています．このようにすることで，アルゴリズムの可読性が向上し，Input・Key process をすばやく識別し，Output を適切に抽出・表示することができるようになります．視覚的なヒントは，アルゴリズム的思考を円滑に進める上で大切です．

一般に，既存のアルゴリズムを読み取ることは比較的簡単ですが，ゼロから新しいアルゴリズムを構築するのは遥かに難しく，新しいスキルセットが必要です．既存アルゴリズムから読み方や修正方法を理解することも有効ですが，新しいアルゴリズムをゼロから構築するにはアルゴリズムデザインのスキルを養うことが不可欠です．

1_2: アルゴリズムの構成要素

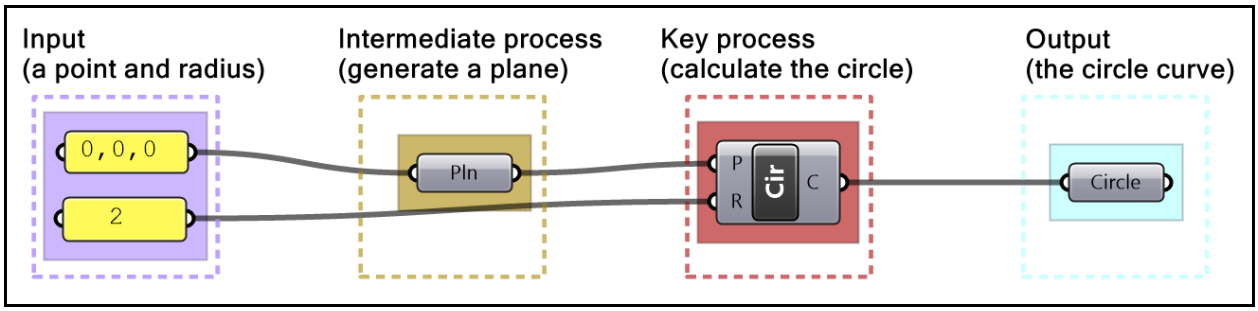
Grasshopper では，アルゴリズムの処理は左から右に流れます．一般には，左端に入力値やパラメータ，右端に出力を置きます．その間には 1 つ以上の主要プロセスがあり，場合によっては追加の入力と出力があります．アルゴリズムの 3 つの構成要素（Input，Key process，Output）を区分するためのわかりやすい簡単な例を見てみましょう．単純な足し算のアルゴリズムでは，2 つの数値（Input），合計値（Output），および数値を受け取って結果を出力する 1 つの Key process が含まれます．以降，Input には紫色，Key process には茶色，Output には水色を使用します．また，複数のパーツをグループ化してラベルを付け，アルゴリズムを左から右に整理します．

Example 1-2-1:
2 つの数値を足し算するアルゴリズム



アルゴリズムには，中間プロセス（Intermediate process）が含まれる場合があります．例えば，中心と半径（Input）を使用して円（Output）を作成する必要があるとします．このとき，円を作成すべき平面が不明なので入力が十分ではありません．この場合，追加情報，つまり円の平面を生成する必要があります．これを中間プロセスとし，以降，薄茶色を使用してラベルを付けます．

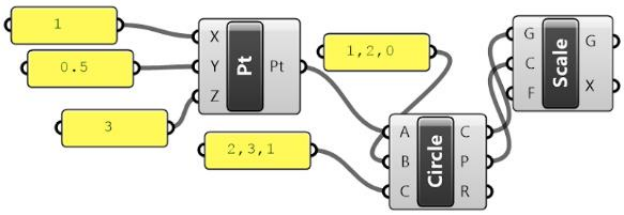
Example 1-2-2:
中心と半径から XY 平面上に円を生成するアルゴリズム



体裁を整えずに記述されているアルゴリズムは，解読や修正が困難になります．他の人が理解し，デバッグし，使用しやすくするためには，アルゴリズムの整理・ラベル付けに時間を使うことも非常に重要です．

Tutorial 1-2-3: 既存のアルゴリズムを読み解く

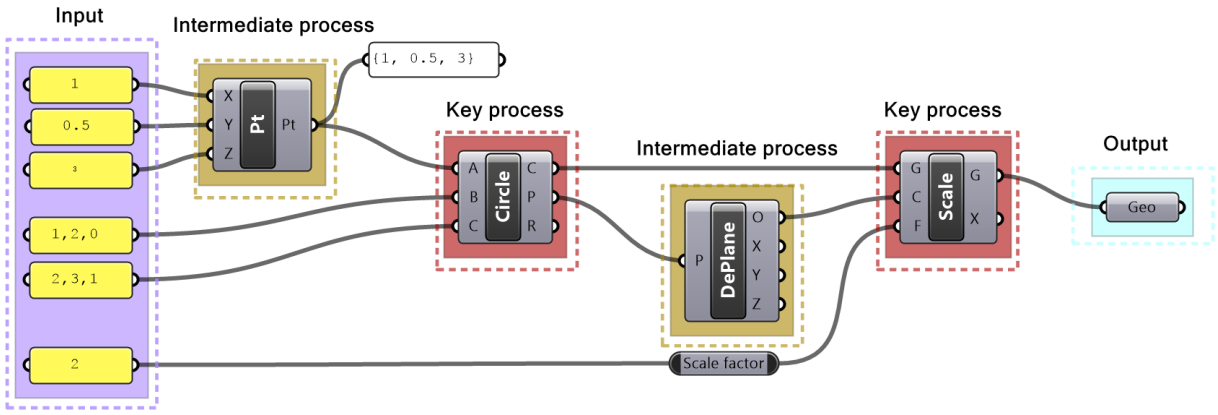
以下の定義を仮定し，どのようなアルゴリズムなのか説明を記述します．入力，主要プロセス，出力を判別してから，すべてのパートにラベルを付け，色分けします．アルゴリズムを修正して，読みやすくします．



Solution

アルゴリズムが何を意味するのかを理解するには，左側で入力をグループ化し，右側で出力をまとめてから，プロセスを順序毎に整理すると良いでしょう．次に，ソリューションを左から右に順に確認して，それが何をするかを推測します．Grasshopper では、各ステップの出力をプレビューして確認することができます．

チュートリアルの例は，与えられた 3 つの点を通る円の 2 倍の大きさの円を作成することを意味しています．点の内 1 つは，X，Y，Z，別々に入力を用意した 3 つの座標値から生成されます．



1_3: アルゴリズムの設計（4 つのプロセス）

アルゴリズムを設計する方法を一般化する前に，ケーキを焼くというような日常の生活でよく出会うアルゴリズムについて検証してみましょう．既にケーキのレシピがある場合は，シンプルに推奨される材料を入手し，それらを混ぜ，器に注ぎ，予熱したオーブンに一定時間入れてから，取り出します．レシピが十分に文書化されているほど作業は簡単です．ケーキを焼くのがさらに上手になったら，レシピを修正するのも良いでしょう．新しい材料（チョコレートやナッツなど）を追加し，異なる道具（カップケーキコンテナ）を使用できるかもしれません．

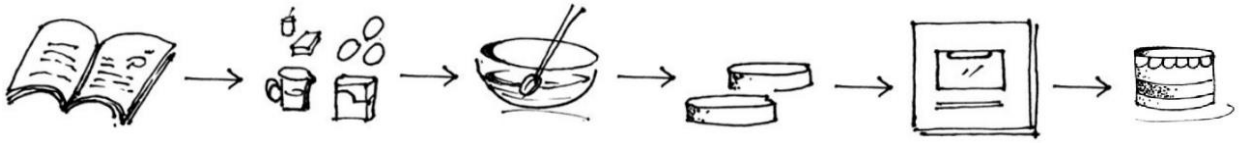


図 (2): 既存レシピに従った手順

設計者がアルゴリズムを作成するとき，通常，既存のアルゴリズムを探して，目的に合わせて変更しようとします．これも良い始め方ではありますが，既存のアルゴリズムを使うとなかなかうまくいかず，時間がかかる場合もあります．また，既存のアルゴリズムにはそれぞれ個性があるため，デザインの決定に影響を与え，創造性を制限する可能性もあります． もし，固有の課題がある場合は，大変であるとはいえ，新しい解決方法をゼロから作成することも選択肢として必要です．

ケーキの例に戻りますが，ケーキを焼く作業は，従うレシピも焼いた経験もない場合，遥かに困難なものになります．材料とプロセスを予測する必要があるため，それらを把握するまでの最初の数回の試みは，おそらく良い結果にならないでしょう．一般に，新しいレシピを作成するときは，逆からプロセスを辿ることが必要です．作りたいケーキの画像から始めて，材料，道具，手順を推測します．考え方としては以下のようになるでしょう．

- ケーキは焼く必要がある．だからオーブンと焼き時間が必要だな．
- オーブンに入れるのは，ケーキ生地でそれを入れる容器も必要だ．
- 生地は材料を混ぜて作れば良いな．

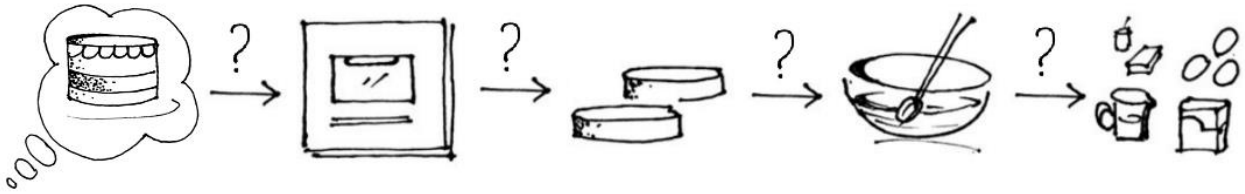


図 (3): 新しいレシピをゼロから作成する手順

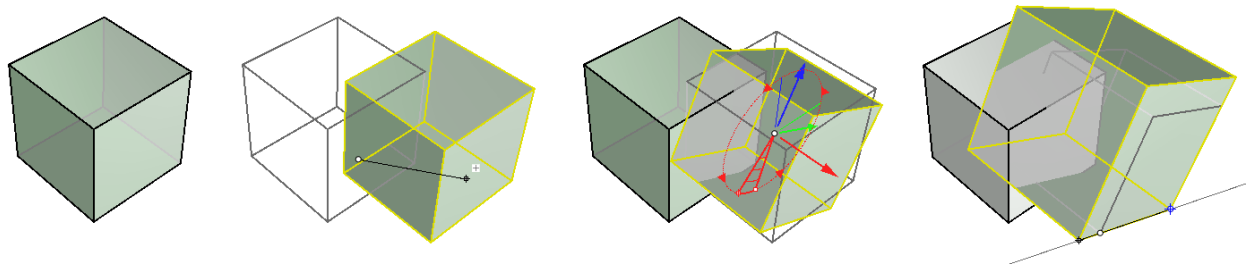
同様の方法論で，パラメトリックアルゴリズムをゼロから設計できます．ただし，新しくアルゴリズムを作成できることは「スキル」なので，根気や練習，および開発に掛かる時間も必要であることを忘れないでください．

3D モデリングおよびパラメトリックデザインにおけるアルゴリズム的思考

3D モデリングでも，アルゴリズム的な考え方がある程度必要となりますが，必要とされる手順とデータは既に決まっている場合が多くあります．例えば，3D モデラーを使用して立体モデルを設計する場合，次のような手順があります．

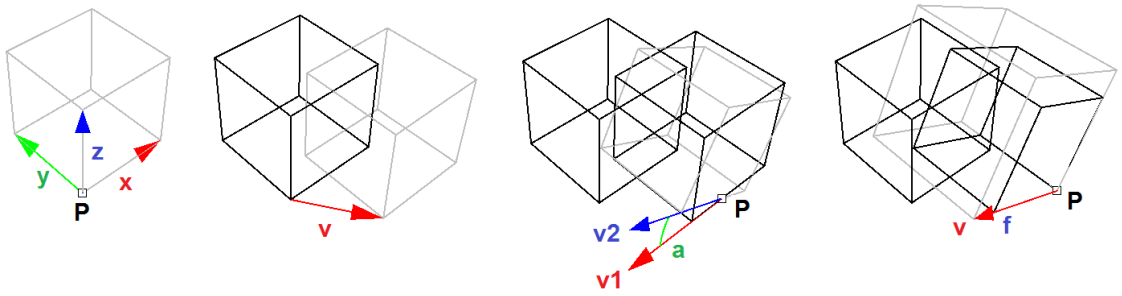
1- 作りたい形状について考えます（例：交差した数個のボックスの集合体）．
2- それを実現するためのコマンド，または一連のコマンド群を特定します（例：***Box*** コマンドを数回実行，いくつかのボックスを ***Move***，***Scale***，***Rotate*** して，ジオメトリを ***BooleanUnion*** する）．
これで完了です！

最初のボックスの基点，幅，高さ，スケール係数，移動方向，回転角度などのデータはコマンドによって要求され，設計者は事前に準備する必要はありません．また，最終出力（ブール演算の結果）が直接操作可能になり，ドキュメント内のオブジェクトとして可視化されます．



図(4): ビジュアルをベースとした機能や補助ツールを用いてジオメトリを作成・操作する対話的な 3D モデリング

アルゴリズムを用いた手法は，対話的ではなく，データとプロセスを明確に定義することが必要です．ボックスの例では，ボックスの向きと寸法を定義しなければなりません．コピーするときはベクトルが必要で，回転するときは平面と回転角度を定義する必要があります．



図(5): アルゴリズミックな方法では，ジオメトリ，ベクトル，変換などの明確な定義が必要

アルゴリズムの設計

アルゴリズム設計には，幾何学，数学，プログラミングの知識が必要です．幾何学と数学の知識は，***Essential Mathematics for Computational Design***[1]でカバーできます．プログラミングスキルに関しては，設計の意図を，ジオメトリを処理する論理的手順として定式化する能力を付ける時間と練習が必要です．最初はどんなアルゴリズムでも，次の 4 段階のプロセスを考えると良いでしょう．

1- 出力したいものを明確にする	**Output**
2- 出力を実現する主要な手順を明確にする	**Key processes**
3- 初期値とパラメータを調べる	**Input**
4- 中間プロセスを定義し，不足データを穴埋めする	**Intermediate input + processes**

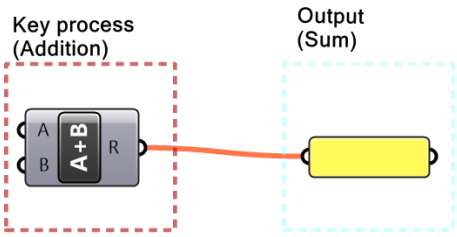
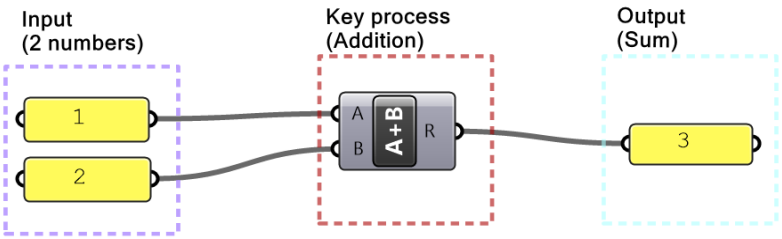
これら 4 段階の観点から考えることが，アルゴリズム設計スキルの上達の鍵です．方法論を説明するために簡単な例から始め，徐々に複雑な例に応用していきましょう．

Example 1-3-1: 2 つの数値の足し算

上記 4 プロセスの考え方を活用して，2 つの数値を足し算するアルゴリズムを作成します．

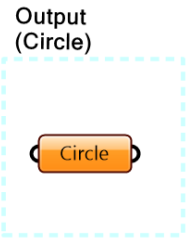
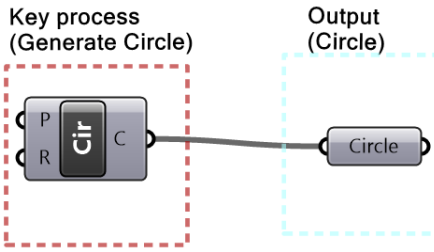
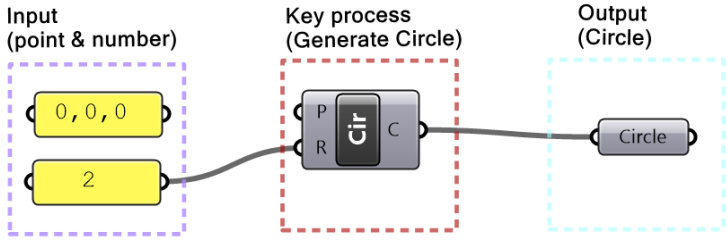
Step 1: Output: 2 つの数値の合計値(Sum). 合計値を表示するために ***Panel*** を用います.	Output (Sum)

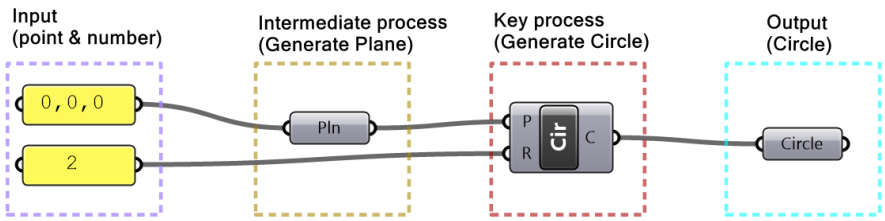
[1] Issa, Essential Mathematics for Computational Design, 4th edition, 2019.
PDF とサンプルファイルが無料でダウンロードできます.
https://www.rhino3d.com/download/rhino/6/essentialmathematics
（日本語版：https://www.applicraft.com/wp/wp-content/uploads/2020/01/TheEssentialMathematics_4thEdition2019-ja.zip）

Step 2: Key process: 足し算. 2 つの数値を入力し，合計値を出力する ***Addition*** コンポーネントを用います.	
Step 3: Input: 2 つの数値. 入力値が書かれた ***Panel*** を用います.	

Example 1-3-2: 1 つの円の作成

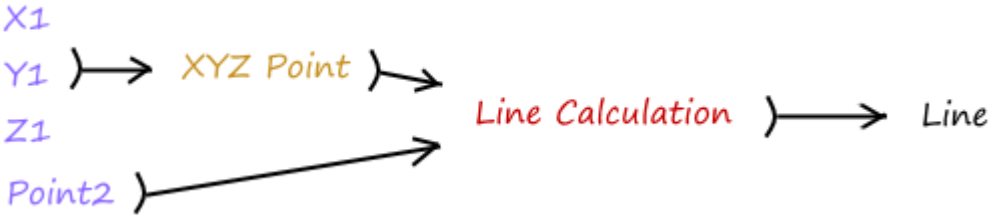
4 プロセスの考え方から，与えた中心点と半径から 1 つの円を作成します.

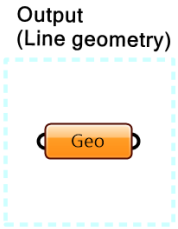
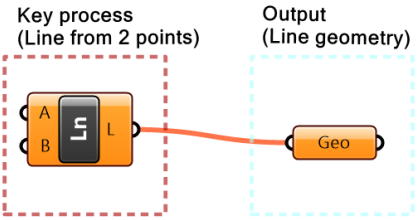
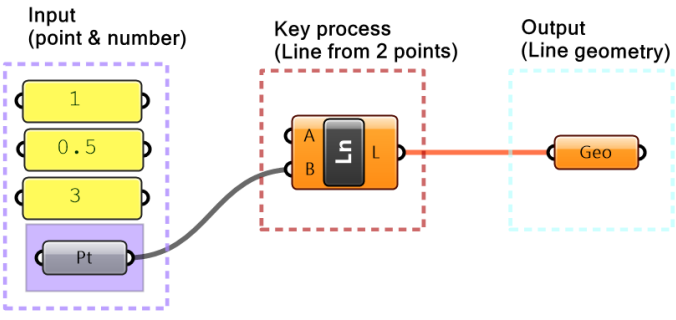
Step 1: Output: 円(Circle). 出力結果を格納するために ***Circle*** パラメータを用います.	
Step 2: Key process: 半径から円を生成する主要なプロセスを明確にします. GH の ***Circle*** コンポーネントを用います.	
Step 3: Input: 指定された入力値（中心と半径）を用います．半径の値を ***Circle*** コンポーネントに繋ぎます.	

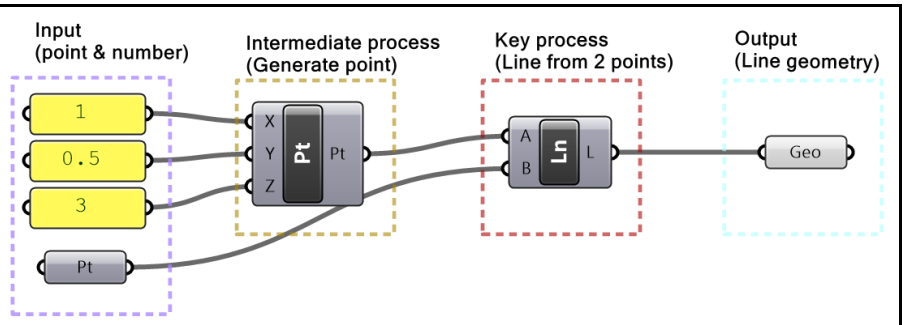
Step 4: Intermediate process: 円を定義するには中心点と，円を配置する平面が必要です．XY 平面に平行な平面に円を置くと仮定し，中心点は原点とします.	

Example 1-3-3: 線分の作成

4 プロセスの考え方から，2 点から線分を生成するアルゴリズムを作成します．1 つの点は Rhino から参照され，もう 1 つの点は 3 つの座標値（x = 1 ，y = 0.5 ，z = 3 ）を使用して作成されます.



Step 1: Output: 線分(Line)ジオメトリ．出力を格納するため，***Geometry*** パラメータを用います.	
Step 2: Key process: 2 点から線分を生成する主要プロセスを明確にします．GH の ***Line*** コンポーネントを用います.	
Step 3: Input: 指定した入力値（参照した点と 3 つの座標値）を用います．1 つの点を線分の終点（端子 B）につなぎます.	

Step 4: Intermediate process: 座標値を点として使用する前に，点を生成する必要があります.	

より複雑なアルゴリズムでも，問題を分析し，考えられる解決手段を調べ，可能な限りそれらを細分化することで，アルゴリズムの管理・読み取りが容易になります． 本資料では，さらに複雑なアルゴリズムを解く場合でも，4 プロセスやその他の手法を引き続き活用していきます.

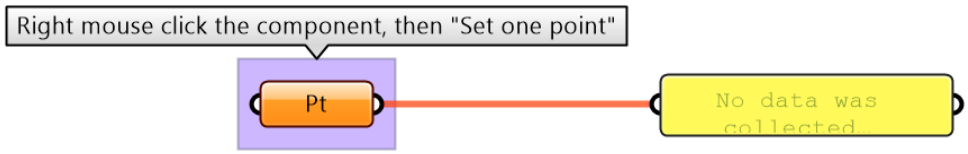
1_4: データ

データとは，コンピュータに保存され、プログラムによって処理される情報のことです． データはさまざまなソースから収集できます．データには多くの形式があり，構造の定義が明確であるため，効率的に使用することが可能です．すべてのスクリプト言語のデータには共通点もありますが，違う点もあります．本資料では，GH に特有のデータとデータ構造について解説します.

1_5: データ ソース

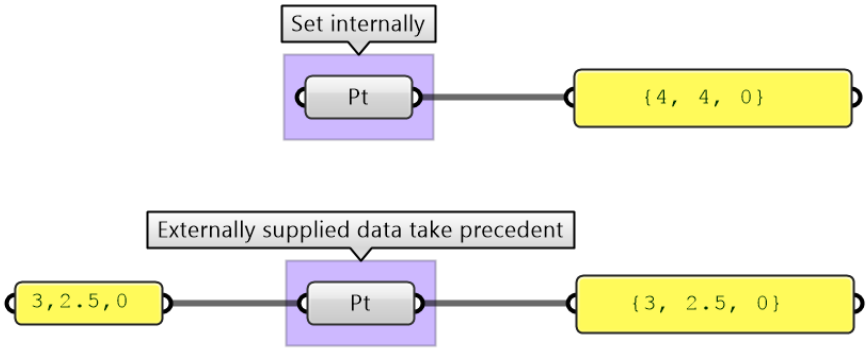
GH では，プロセス（所謂コンポーネントと呼ばれるもの）にデータを受け渡す方法が，大きく 3 つあります．内部（internal），参照（referenced），外部（external）です.

Grasshopper におけるデータ ソース
1- 内部的（Internally）にデータを set する データは，パラメータのインスタンスとして内部的に設定（set）できます．set すると，外部入力によって手動で変更または上書きされない限りその値のままです．この方法は，定数のような設定後にデータ変更の必要がない場合に適します．データは GH ファイル内に保存されます. 
2- 参照（Referenced）データ データは，Rhino または外部ドキュメントから参照できます． 例えば，Rhino で作成した点オブジェクトも参照可能で，Rhino で点を移動すると，GH の参照データも更新されます（コンポーネントの右クリックメニューから Set）．GH ファイルは Rhino ファイルとは別に保存されるため，GH ファイルが Rhino のデータを参照している場合には，データが欠損しないよう，GH ファイルと同時に Rhino ファイルも保存し，開く必要があります.



3- 外部（Externally）からデータを与える

データは，それより前のプロセスの出力から入力ことができます．この方法は，動的なデータやデータをパラメトリックに制御する場合に最適です．パラメータに外部から入力されたデータは，内部値または参照値よりも優先されます．



1_6: データ型

すべてのプログラミング言語は，割り当て可能な値や，操作やプロセス上で使用可能かどうかを，データ型で識別します．一般的なデータ型としては，整数（***Integer***），数値（***Number***），文字列（***Text***），ブール値（***Bool*** : true または false）などがあります． GH では，それらは Params> Primitives タブに含まれます．

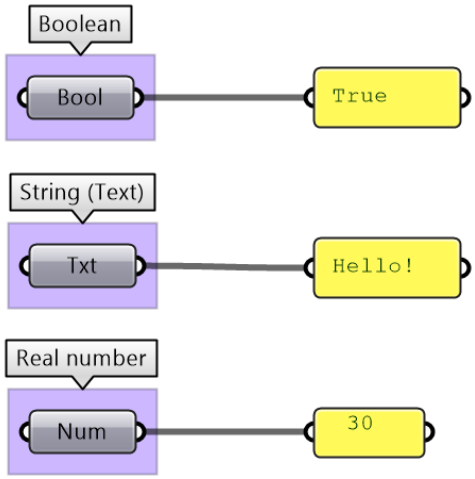


図 (6): すべてのプログラミング言語に共通のプリミティブなデータ型の例.

GH は，点（***Point*** : 座標を表す 3 つの数値），線分（***Line*** : 2 点間），曲線（***NURBS Curve***），曲面（***NURBS Surface***），ソリッド・ポリサーフェス等（***Brep***）などの一般的に 3D モデリングで使用されるジオメトリタイプをサポートしています．すべてのジオメトリタイプは，GH の Param> Geometry タブに含まれます．

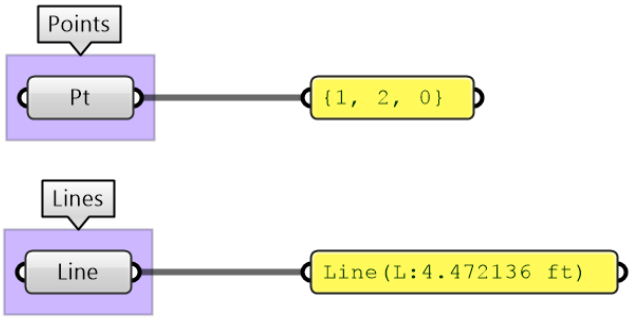


図 (7): ジオメトリデータの型の例.

その他の数学的な型としては，ドメイン（***Domain***），ベクトル（***Vector***），平面（***Plane***），変換マトリックス（***Transformation Matrix***）があります．これらは通常の 3D モデリングではあまり使用しませんが，パラメトリックデザインでは非常に一般的です．GH には，これらの型のデータの作成・分析・利用に役立つツールが豊富に揃っています．NURBS カーブやサーフェスのようなジオメトリの種類や数学の知識をしっかり理解するには，ぜひ著者による「***Essential Mathematics for Computational Design***」を参照してみてください.

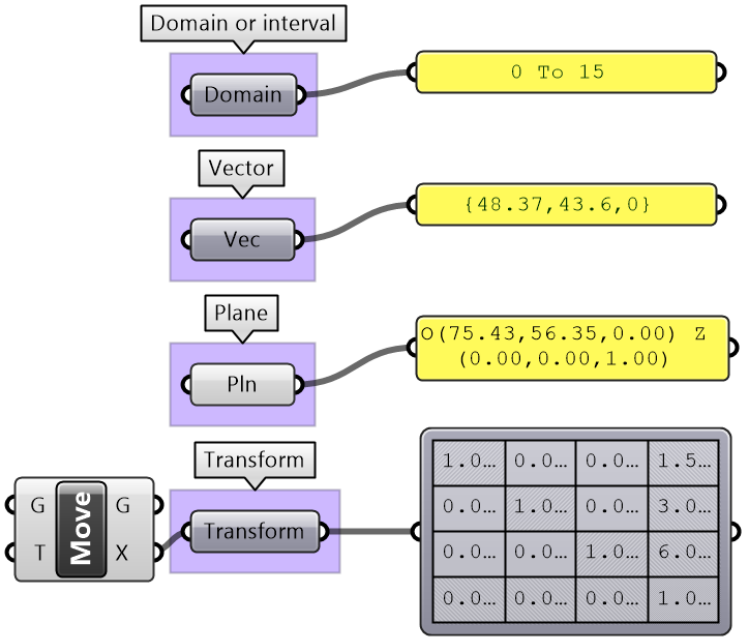


図 (8): コンピュータグラフィクスで一般的なデータ型の例.

GH のパラメータコンポーネントを使うと，データをある型から別の型に変換可能です（キャストやキャスティング等と言います）． 例えば，文字列を数値に変換する必要がある場合は，文字列データを ***Number*** パラメータに繋げば良いです．変換できない場合はエラーとなります.

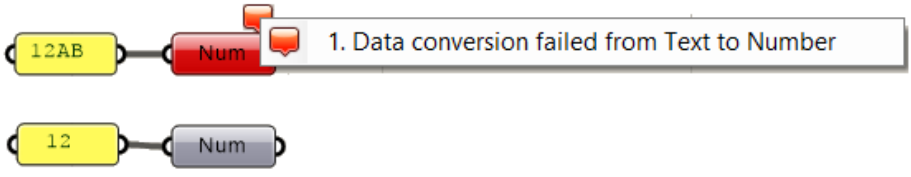


図 (9): Grasshopper のパラメータコンポーネントによるデータ変換 (キャスティング) .

GH コンポーネントは，可能な場合，入力を適切なタイプに内部的に変換します．例えば，***Addition*** コンポーネントに Text で入力すると，GH は Number として読み取ろうとします．コンポーネントが複数の型を処理できる場合，コンポーネントは変換せずに入力タイプのまま使用します．例えば，数式の等号・不等号は，Number だけでなく Text の比較もできます．このような場合は，混乱を避けるために目的に合った型を使うよう心掛けましょう．

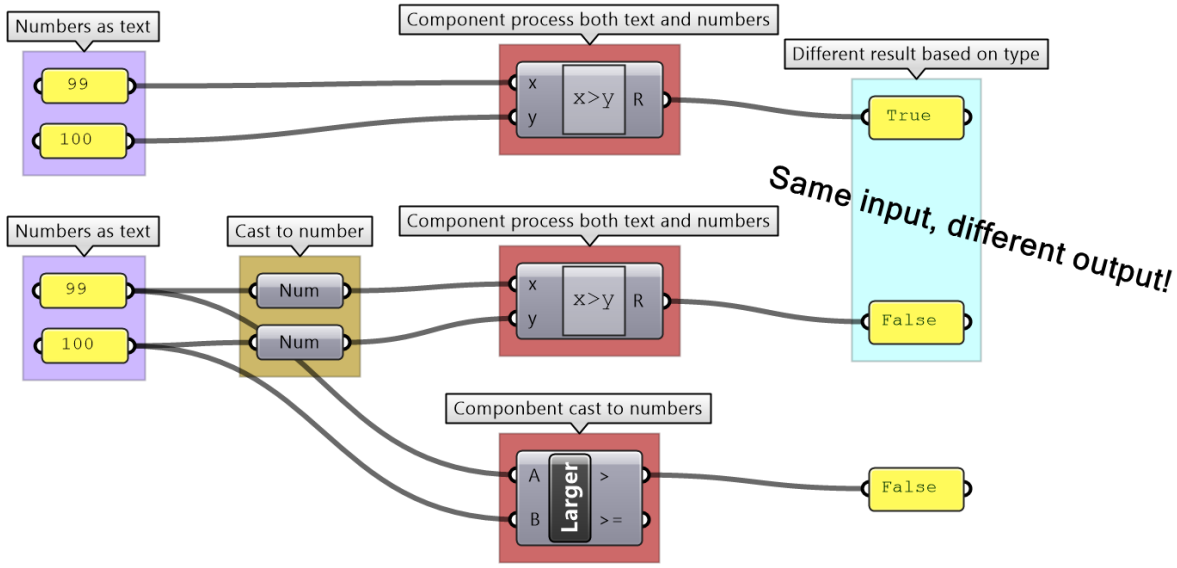


図 (10): 一部の演算は，複数の型を処理可能です．コンポーネントが複数の型を処理できる場合（GH の ***Expression*** など）は，特に目的の型にキャストしておくようにしましょう．

GH コンポーネントが無効な入力（null データや誤った型）を単に無視する場合があります．そのような場合，予期しない結果になり，バグを見つけるのが難しくなります．各コンポーネントの出力を使用する前に確認することが非常に重要です．

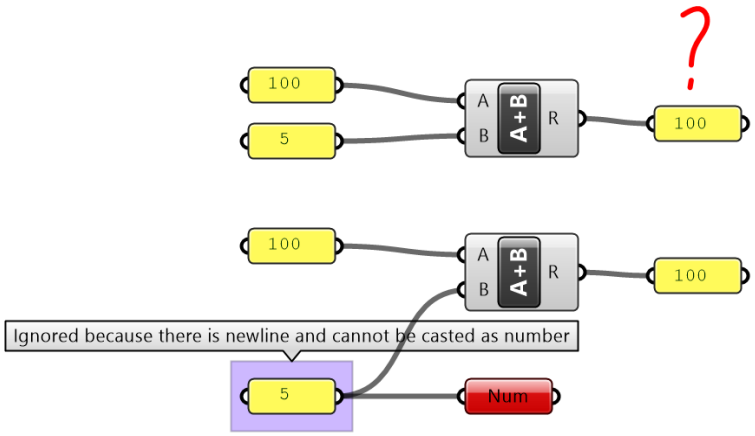


図 (11):無効な入力は無視され，デフォルト値が使用されます．例えば，***Panel*** コンポーネント内の数値は文字列として解釈される可能性があるため，***Addition*** コンポーネントへの無効な入力となる可能性があります．

1_7: データの処理

アルゴリズムの設計では，多くのデータの演算と処理を利用します．この資料では，6 つのカテゴリ（数値演算，論理演算，分析，ソート，取捨選択，マッピング）に焦点を絞って紹介します．

1_7_1: 数値演算

数値演算には，四則演算，三角法，多項式、複素数などの演算が含まれます．GH には，豊富な数値演算機能があり，それらは主に ***Math*** タブに含まれます．GH での演算の実行には，主に 2 つの方法があります．まず，加算（***Addition***），減算（***Subtraction***），乗算（***Multiplication***）などの特定の演算用に設計されたコンポーネントを使用する方法です．

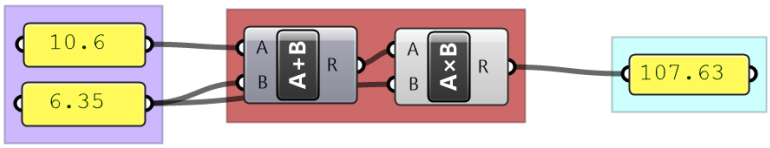


図 (12): GH の数値演算コンポーネントの例.

もうひとつが，***Expression*** コンポーネントを使用する方法で，1 つの数式で三角関数などの複数の演算子や関数を組み合わせて実行することができます．

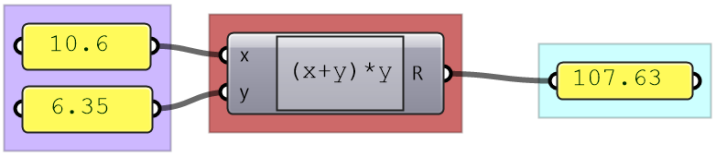


図 (13): 複数の演算を組み合わせて実行することができる GH の ***Expression*** コンポーネント.

複数の演算がある場合は，***Expression*** を用いると，より見やすく，ミスが起きにくくなるでしょう．

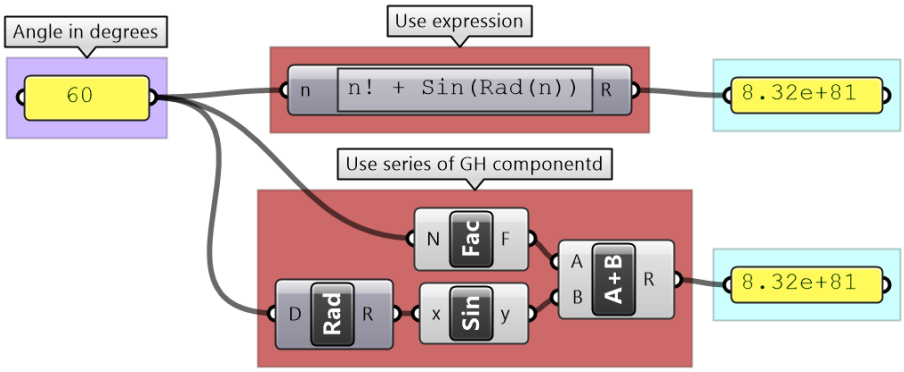


図 (14): 複数の演算を実行する場合，***Expression*** コンポーネントを用いると，メンテナンスがより楽になります.

Expression は，数式の形式によっては，入力を文字列として扱うこともできます．

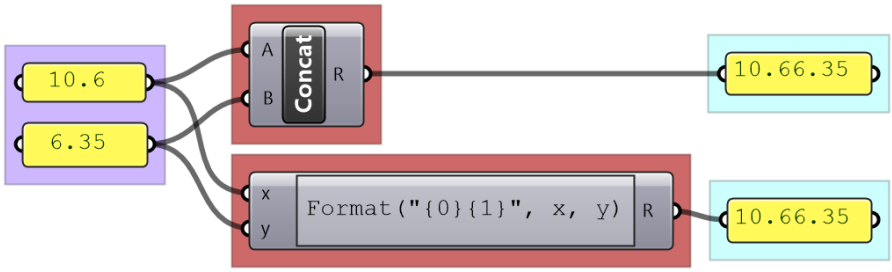


図 (15): ***Expression*** は，文字列の処理やフォーマット化に用いることもできます.

多くのコンポーネントの Number 入力端子では，オプションで数式を記述できるようになっており，その数式の結果を入力として用いることができます．例えば，***Range*** コンポーネントには N（ステッ

プ数）入力があります．「N」を右クリックすると，数式（***Expression***）を設定できます．端子名に関わらず常に「x」を使用して，入力値を表します．

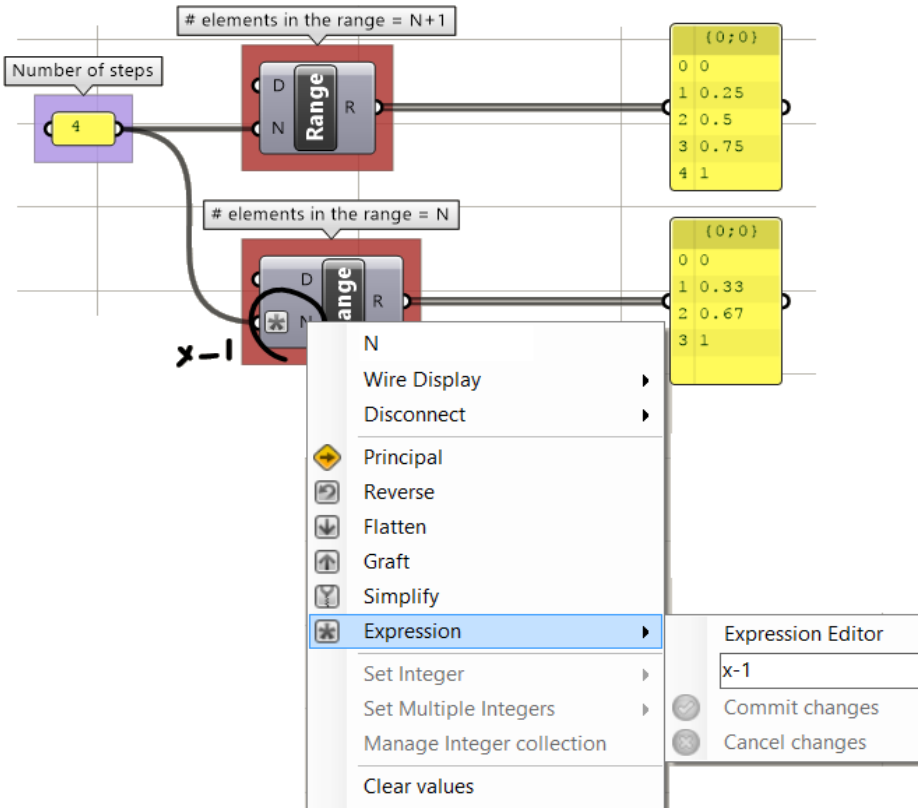


図 (16): 入力パラメータ内で数式の設定が可能です．変数「x」で入力値を表します．

1_7_2: 論理演算

GH の主な論理演算には，比較，集合などの演算が含まれます．

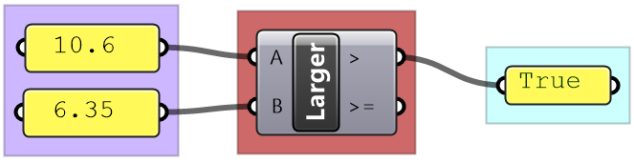


図 (17): GH には論理演算を行ういくつかのコンポーネントがあります．

論理演算は，データの条件付きフローを作成するためにも利用できます．例えば，半径が 2 つの値の範囲に収まる場合にのみ球を描画したい場合，半径が制限内にない場合に半径の入力を除外するロジックを作成する必要があります．

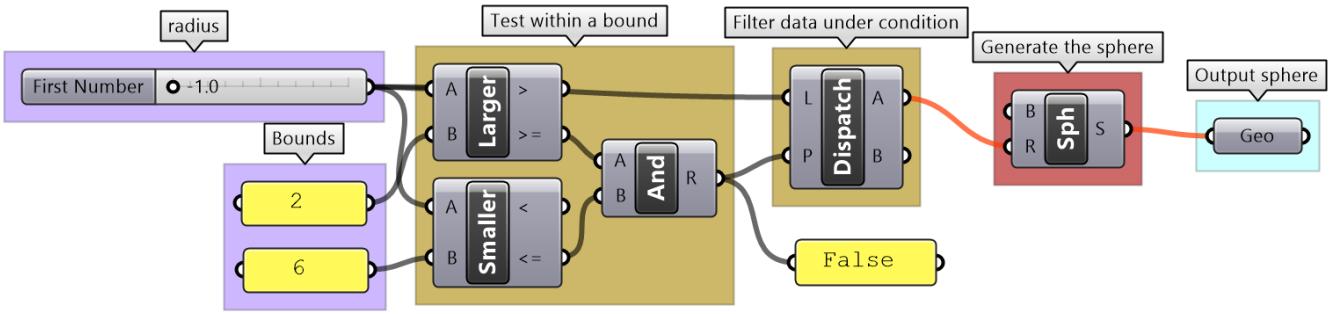


図 (18): 論理演算を用いたデータフローの制御

1_7_3: データ分析

GH には，データを調べて，プレビューするためのツールがたくさんあります．***Panel*** は，データとその構造の詳細を表示するためにも使用できます．***Parameter Viewer*** はデータ構造を見るためだけに用います．ほかの分析コンポーネントとしては，データをグラフでプロットする ***Quick Graph*** や入力した数値群の上限・下限を抽出する ***Bounds*** などがあります．

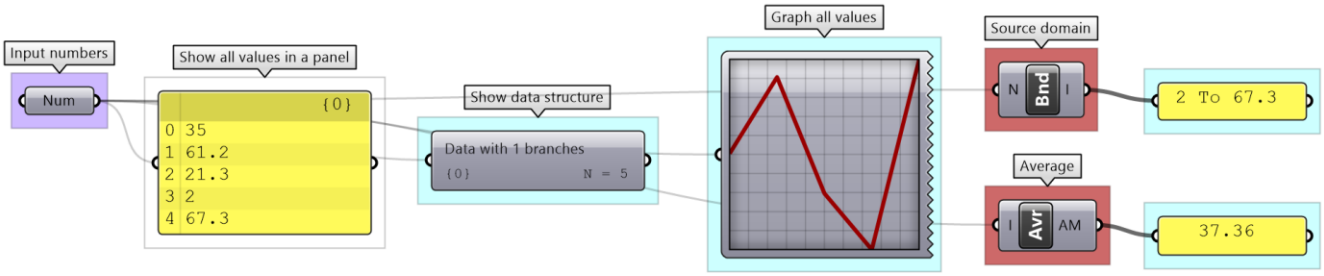


図 (19): GH でデータを分析するいくつかの方法.

1_7_4: ソート

GH には，数値データとジオメトリデータを並べ替えるためのコンポーネントがあります．***Sort List*** コンポーネントは，K（Keys）端子に入力した数値リストを昇順に並べ替えます．追加で ***Reverse List*** を使えば降順にもできます．***Sort List*** を使えば，例えば曲線の長さなどいろいろな Keys を入力することにより，ジオメトリの並べ替えも可能です．GH には，点を座標でソートするための ***Sort Points*** などの特定のジオメトリのソートのために設計されたコンポーネントもあります．

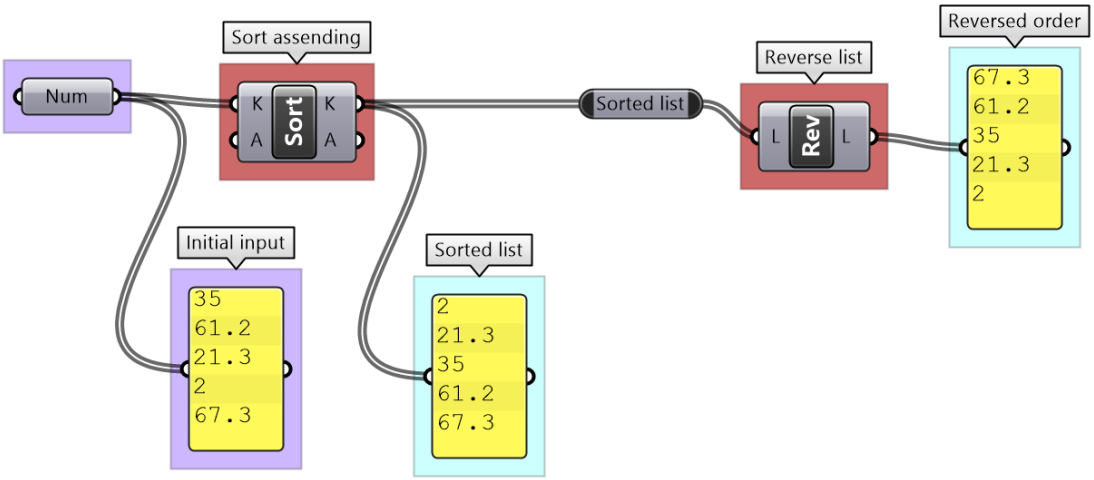


図 (20): GH での数値の並べ替え（ソート）

1_7_5: 取捨選択

3D モデリングでは，特定のオブジェクトまたはオブジェクトのグループをインタラクティブに選択できますが，アルゴリズム設計ではそのようにはできません．GH では，データ構造内の場所の指定，あるいは何かしらのパターンによってデータを取捨選択します．例えば，***List Item*** コンポーネントでは，インデックス番号に基づいて要素（アイテム）を選択します．

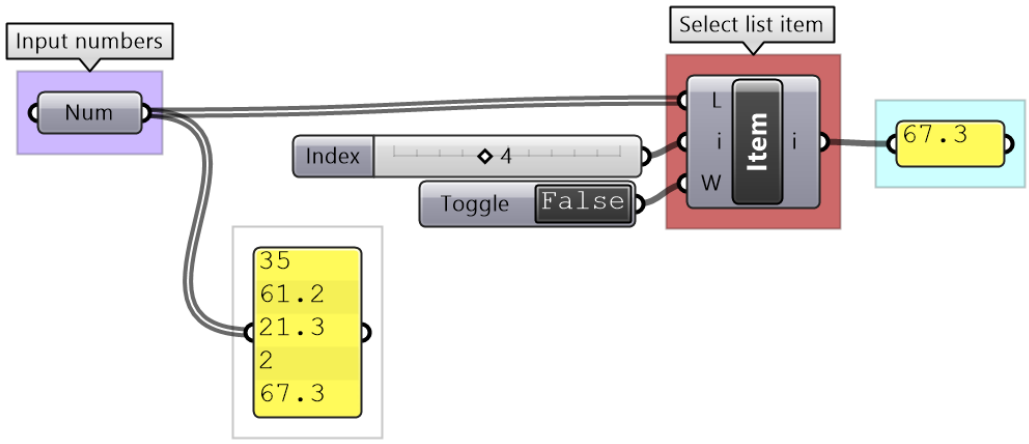


図 (21): GH でリストからアイテムを選択する.

Cull Pattern コンポーネントは，繰り返しパターンを利用してデータの一部を抽出します.

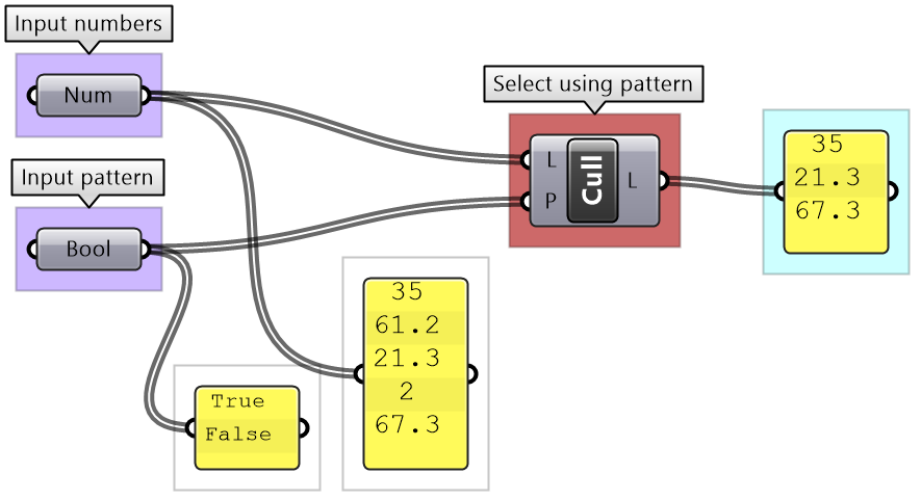


図 (22): リスト内のアイテムを 1 個おきに選択する例.

例からもわかるように，特定アイテムの選択や，Cull コンポーネントを使用すると，データの部分的なリスト（サブセット）が生成され，残りは破棄されます．しかし，データの一部をリストから分離・演算し，元のリストに再結合したい場合も多々あるでしょう．これは GH でも可能ですが，より応用的な処理が必要になるため，Chapter 3 の高度なデータ構造の内容で解説します.

1_7_6: マッピング

マッピングとは，指定された数値の範囲に対して，あるセット内のそれぞれの数値を新しいセットに線形に置き換えることを指します．GH には，***ReMap*** と呼ばれる線形マッピングを実行するコンポーネントがあり，一連の数値を元の範囲から新しい範囲にスケーリングできます．これは，アルゴリズム上の要件や制限に合ったドメインに数値の範囲を合わせるのに役立ちます.

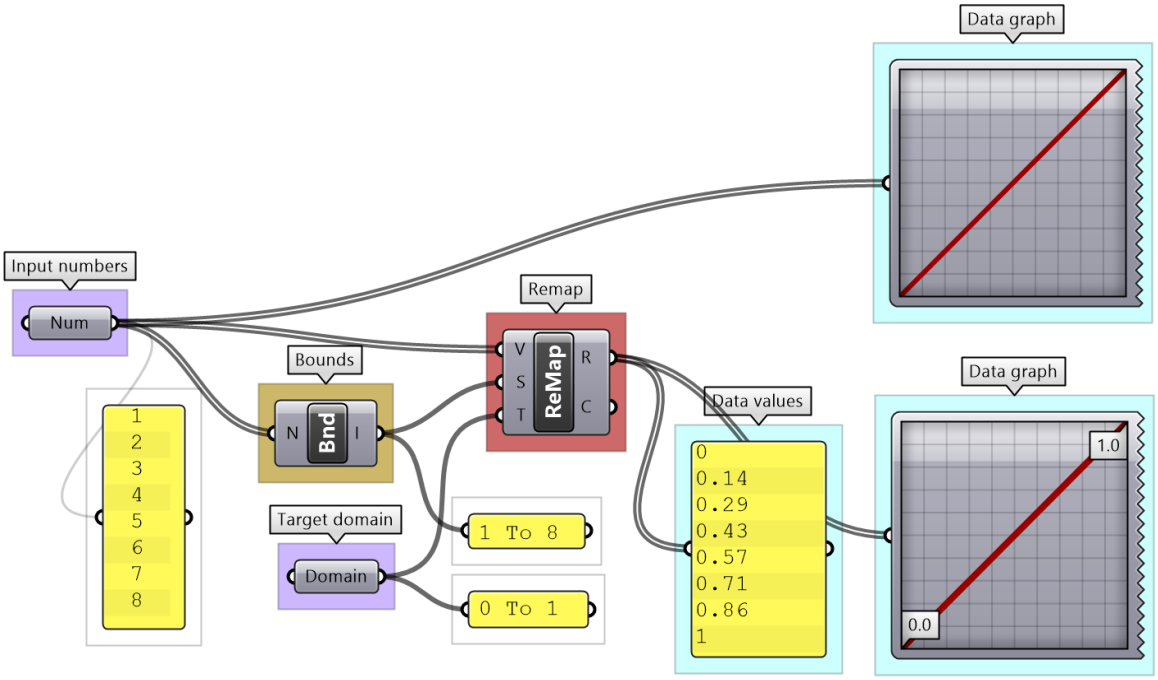


図 (23): GH での数値の線形リマッピングの例.

マッピングはデータ変換の一種です．例えば，角度の単位を度からラジアンに変換する場合もマッピングです（GH コンポーネントは，ラジアンで角度を受け入れるよう設計されています）．

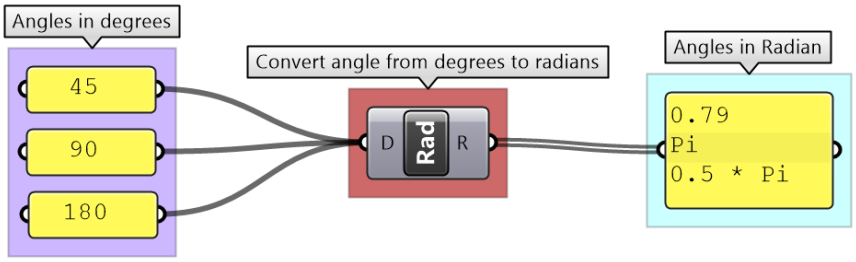


図 (24): 度からラジアンへの角度の変換.

パラメトリック曲線には「ドメイン」（曲線上の点として評価するパラメータ範囲）があります． 例えば，特定の曲線のドメインが 12.5〜51.3 の場合，12.5 は曲線の始点を表します．多くの場合，一貫したパラメータを使用していくつかの曲線を評価することが必要になります．それぞれの曲線のドメインを統一した範囲に再定義（Reparameterize）すれば，この問題は解決できます．そのときに一般的に使うドメインは「0〜1」です（この処理を正規化と言います）． GH コンポーネントの Curve の入力端子には，ドメインを 0〜1 に再定義する ***Reparameterize*** オプションがあります.

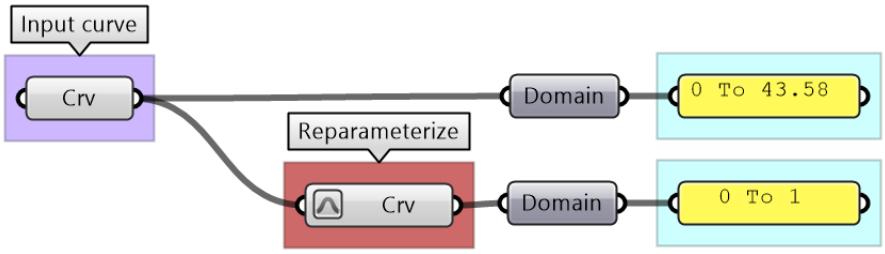
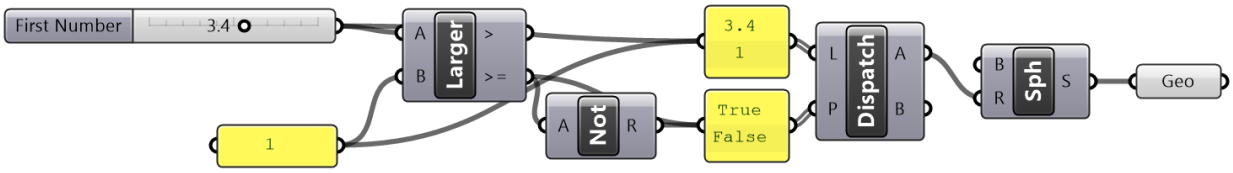


図 (25): Curve のドメインの正規化（0〜1 に変換）． GH の Reparameterize 入力オプションを使います.

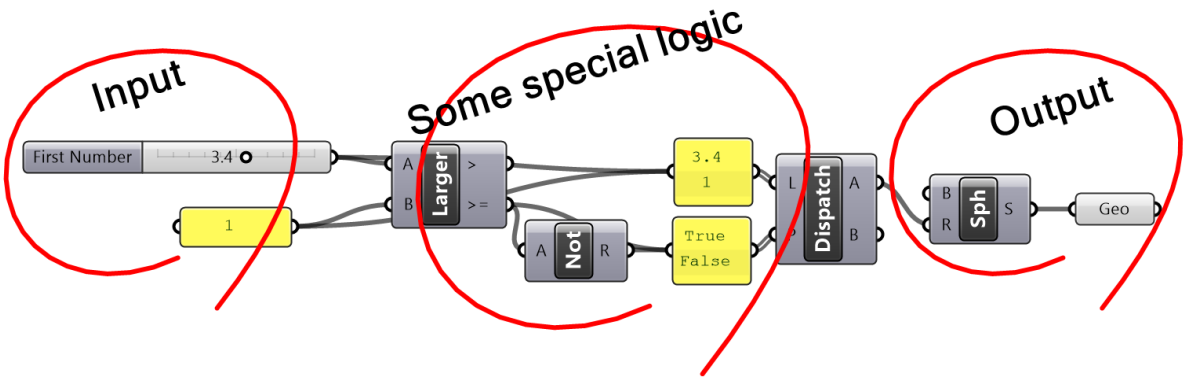
Tutorial 1-7-1: フローの制御

以下のアルゴリズムの目的は何でしょうか？注釈と色を使ってそれぞれのパートの目的を記述してみましょう.



アルゴリズムの分析

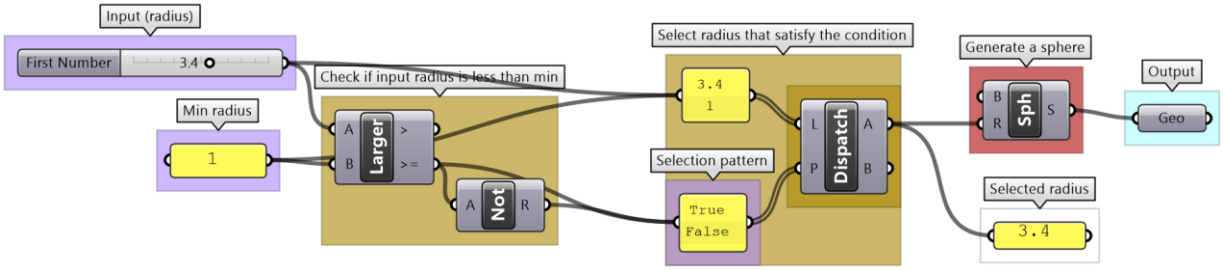
球の出力があり，半径の入力とそれを処理するいくつかのロジックがあります.



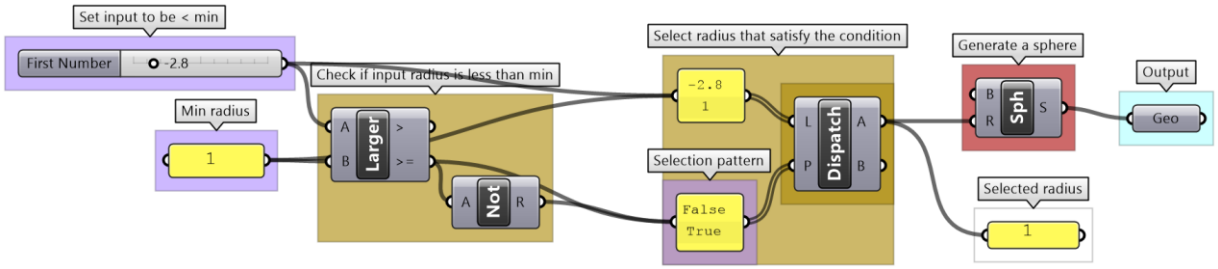
アルゴリズムへ注釈と色付け

出力を確認し，処理の手順を見ると，球の半径が 1 未満にならないように意図されていることがわかります.

・半径 を 1 より大きくしてテスト



・半径 1 以下でテスト



Tutorial 1-7-2: データの処理

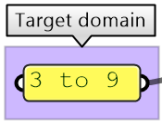
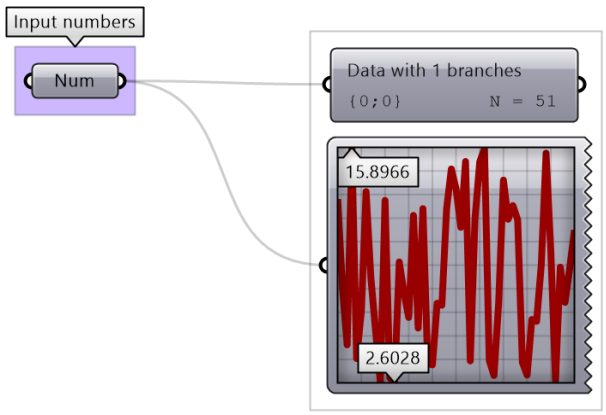
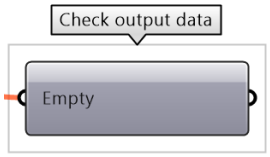
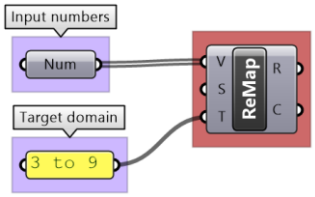
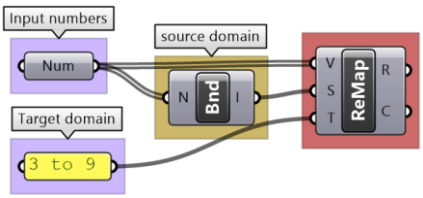
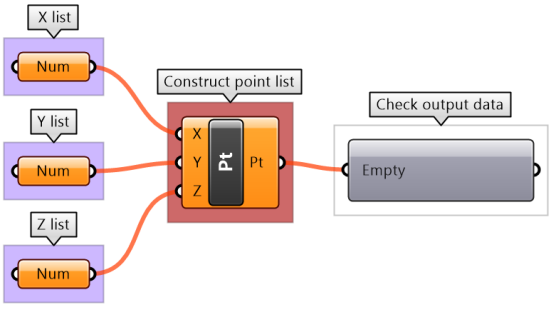
点座標のリストを指定して，以下の操作を行います.

1- データの内容を把握するためにリストを分析します.

2- 既存ドメインを新しいドメインにマッピングし直します.

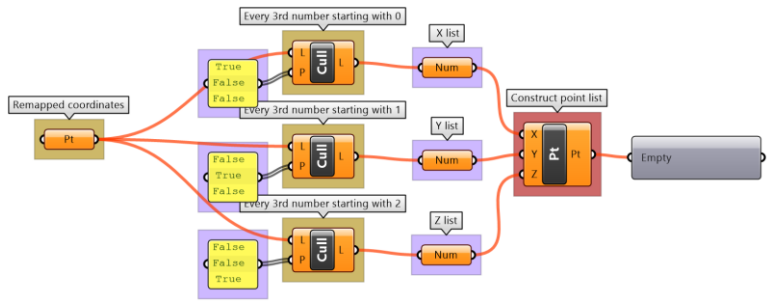
3- ドメイン 3～9 にマッピングし直した点座標成分を *Point* オブジェクトとして生成するアルゴリズムを記述します.

Note　入力リストの内容は，最初の 3 つの数値が順に 1 番目の点の x，y，z の座標成分で，次の 3 つの数値が 2 番目の点の座標成分となっています．

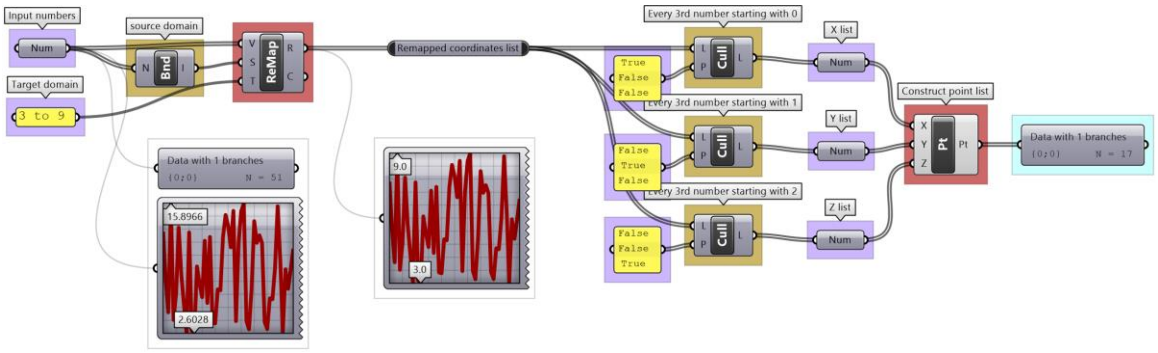
アルゴリズムの分析	
51 個の数値（17 個の点×3 つの座標成分）が入ったリストがあります． ***QuickGraph*** で，2.60～15.89 の範囲のグラフが表示します．値が無作為に分布していることがわかります． もう一つの入力は，ターゲットドメインです． 	
アルゴリズムを解くための 4 プロセス	
Output 点オブジェクトのリスト．	
Key Process #1 座標のリマップ ***ReMap*** コンポーネントを用いて，座標を既存のドメインから新ドメインにマッピングし直します．	
Intermediate Process #1 既存ドメインの入力が足りないので ***Bounds*** コンポーネントを使って抽出します．	
Key Process #2 点の生成 ***Construct Point (Pt)*** コンポーネントを使って座標成分から点を生成します．	

Intermediate Process #2
リストから X，Y，Z，それぞれの座標成分を抽出します．リストからそれぞれの成分を抽出するのに適したパターンを ***Cull Pattern*** コンポーネントに入力します．

Cull への入力は，Process #1 でリマップした点です．



全体像



1_8: アルゴリズミックデザインの落とし穴

効率的で読みやすくデバッグしやすいような洗練されたアルゴリズムを書くことは簡単ではありません．ここまでは，色分けとラベル付けを活用したスタイルでアルゴリズムを記述する方法を説明しました．また，アルゴリズムの開発に役立つ 4 プロセスの考え方を紹介しました．これらのガイドに従えば，バグを最小限に抑え，スクリプトの可読性を向上することにもつながります．次は，誤った結果，または意図しない結果を生むよくある問題をいくつか挙げます．

1_8_1: 無効または不正な型の入力

GH では，入力の型が正しくないか無効な場合，コンポーネントの色が赤またはオレンジに変わり，エラーの警告とどのような問題が生じているかに関するフィードバックを表示します．これも役に立ちますが，コンポーネントがデフォルト値を割り当てている，あるいは代替値で計算していた場合，誤った入力に気付けないことがあります．常に入力を二重チェックするようにしましょう（Panel または Parameter Viewer をつなげ，入力にラベル付けします）．間違った型の使用を避けるには，正確さを保つためにも，意図した型に変換しておくことをお勧めします．

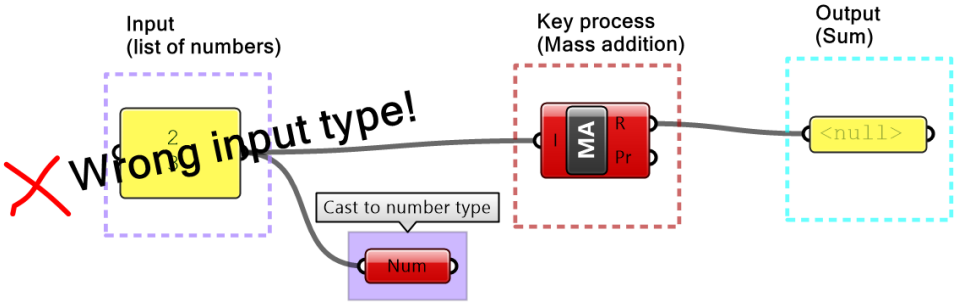


図 (26): 誤った入力型のエラー結果.

1_8_2: 意図しない入力

中間プロセスを介したり，複数ユーザーでアルゴリズムを作成していると，入力が意図しないものに変わってしまうことがよくある傾向にあります．すべての主要な入力と出力をプレビューして正しいかどうかチェックすることは大変有効です．***Panel*** コンポーネントは非常に用途が広く，すべての型の値をチェックするのに役立ちます．また，範囲外の値の入力や，不要な値の入力を防ぐ仕組みを作ることも良いでしょう．

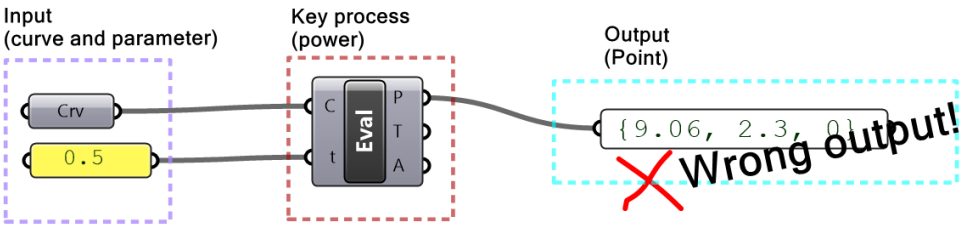


図 (27): 意図しない入力が原因の誤り．曲線ドメインは 0〜1 で 0.5 を使えば中点が得られる，とは限りません．

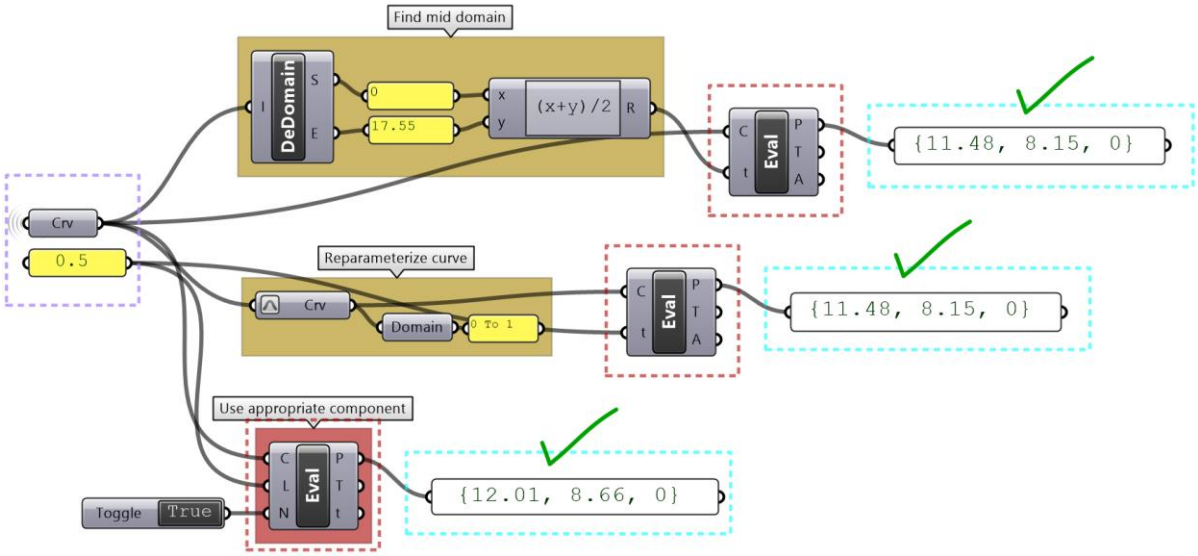


図 (28): 曲線の中点を評価するために適した解法の例．

1_8_3: 処理の順序が不明瞭

処理の流れが見やすくなるようアルゴリズムを縦や横に整理するようにしましょう．また，コード作成を続ける前に，各出力をチェックし，期待通りになっているか確認も必要です．ミスを防ぐテクニックもあります．例えば，複数の数値や処理を含む場合には ***Expression*** を使用するなどです．以下では，いくつかの好ましくないコンポーネントの組み方の例を強調して記載しています．

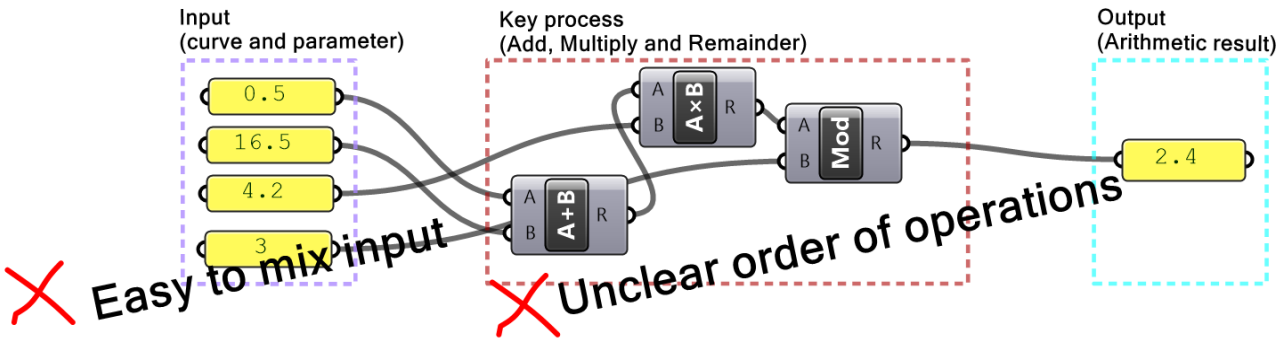


図 (29): 十分整頓されていないと入力を混同しやすい．

以下は，同じコードをミスが起きにくくなるように書き換える方法を示しています．

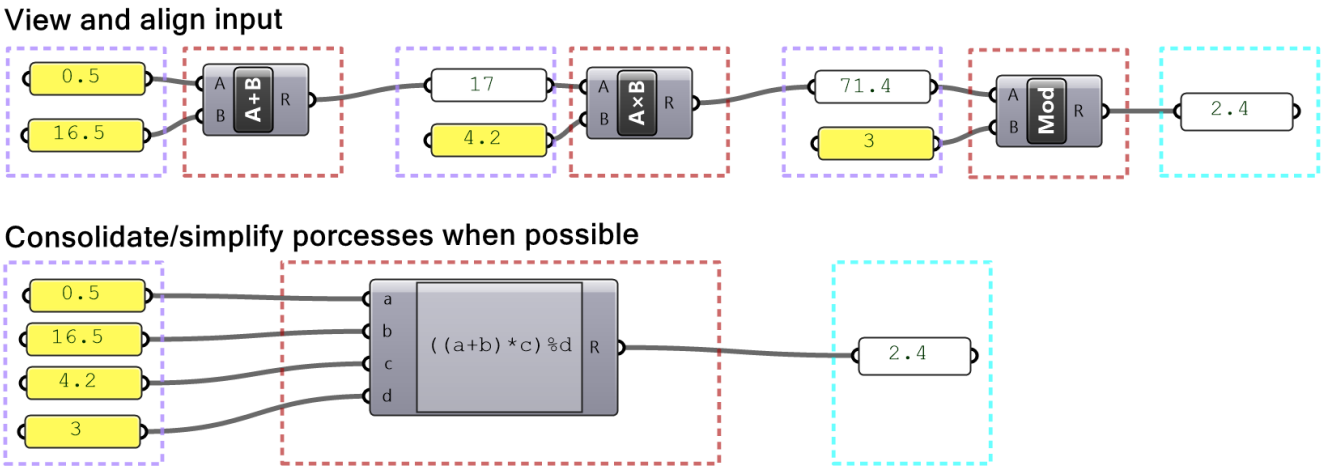


図 (30): プロセス毎に入力を並べる，あるいは ***Expressions*** を用いるのが最適です．

1_8_4: データ構造の不一致

GH において，同じプロセスまたはコンポーネントで入力されるデータ構造の不一致を防ぐことは特に難しく，データ構造の不一致があると，計算量が増大しメモリオーバーとなる可能性があります．コンポーネントにつなぐ前に，すべての入力（自明なもの以外）のデータ構造をテストすることが重要です．また，さまざまなシナリオで必要なマッチングを調べることも大事です（データマッチングについては後で詳しく説明します）．

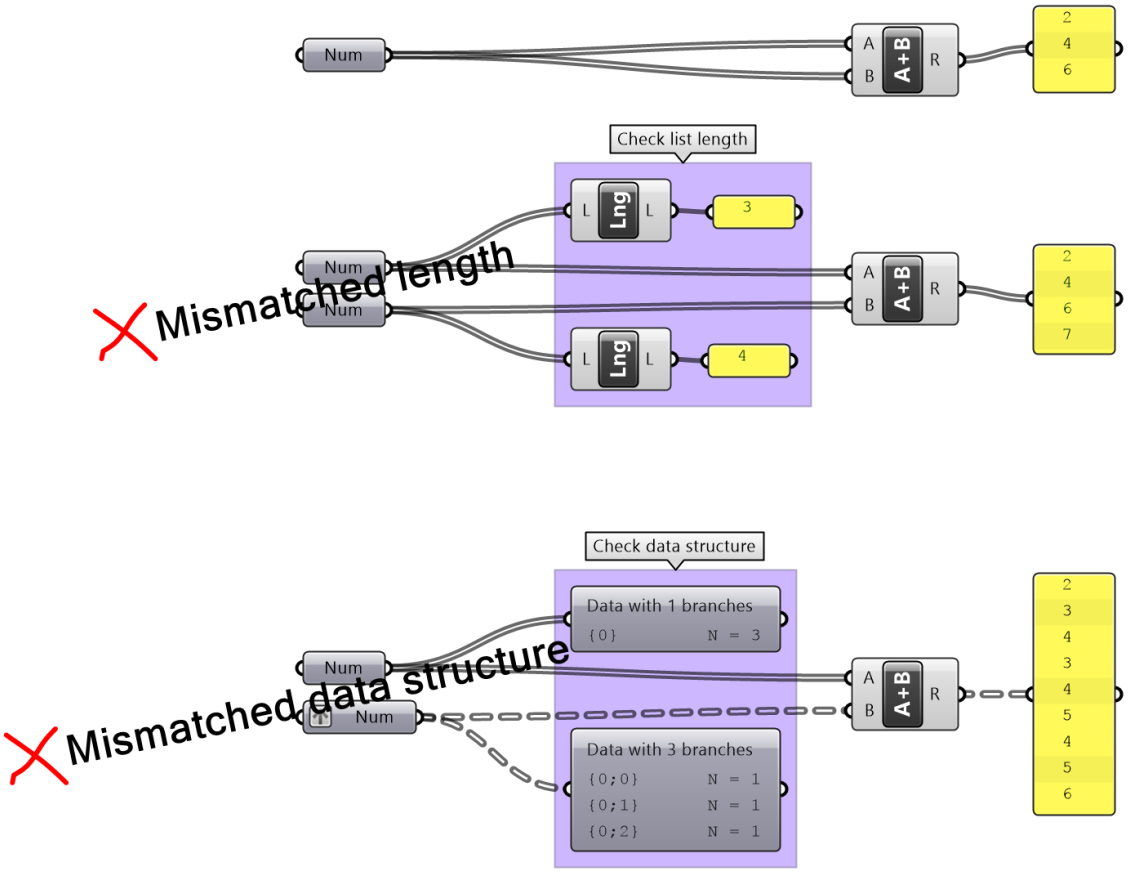


図 (31): 入力のデータ構造が一致しない場合，誤った出力の原因となります．

1_8_5: 長い処理時間

一部のアルゴリズムは計算に時間がかかり，処理が完了するまで待たなければなりませんが，場合によって待機時間を最小限に抑える方法があります．例えば，開発の初期段階では，完全なデータセットを処理する前に，アルゴリズムのテストのために，より小さいデータセットを使用すると良いでしょう． また，可能な場合はアルゴリズムを段階的に分割して，時間のかかる部分を分離して無効にすることもできます．また，アルゴリズムを書き直して最適化し，時間を節約することができる場合も多々あります．GH の ***Profiler*** 機能では処理時間を表示できるので，処理に時間がかかりすぎる，あるいはクラッシュしてしまう場合は，計算を再度実行する前に，計算を無効にし，クラッシュの原因となった入力を切断するようにしましょう．

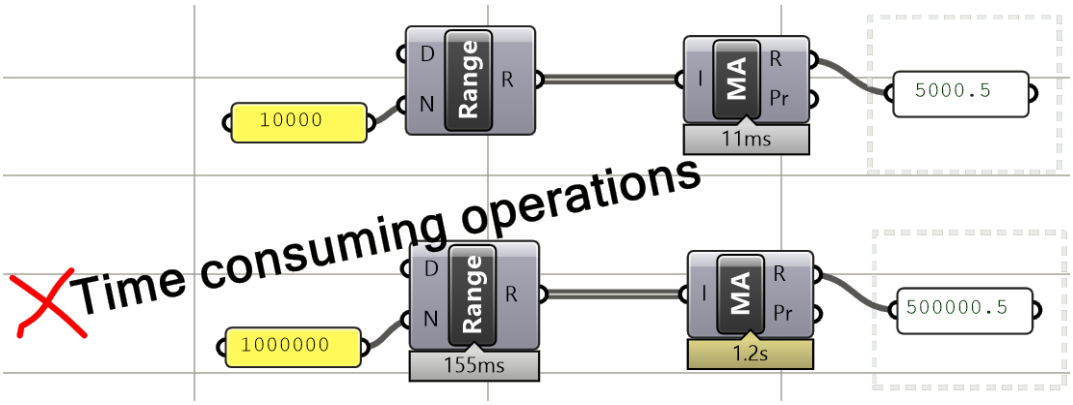


図 (32): GH の Profiler ウィジェットは処理時間を計測します.

1_8_6: 整理が不十分

整理が不十分な定義は，デバッグ・理解・再利用・修正が難しくなります．最初は余分に時間が掛かりますがそれでも，体裁を整えて定義を記述することの重要性はどれだけ言っても言い過ぎるということはありません．色分け，ラベル付け，変数への意味のある命名，繰り返し処理のモジュール化，入出力のプレビュー，を常に心掛けましょう．

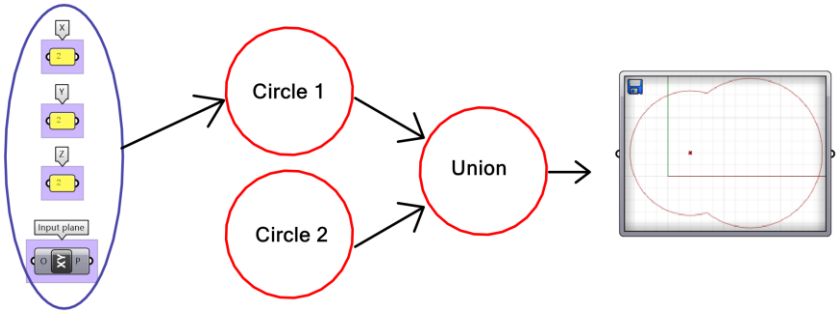
図 (33): ビジュアルプログラミングにおいて，整理が不十分だとコードの解読やデバックが難しくなります.

1_9: アルゴリズムのチュートリアル

1_9_1: 円の合成

4 プロセスの考え方から，2 つの円を合成するアルゴリズムを設計します．円は両方とも XY 平面に配置されています.
最初の円 Cir1 は，中心 C1=（2,2,2），半径 R1= 3～6 のランダムな数.
2 番目の円 Cir2 は，中心 C2 が C1 から正の X 軸方向に R1 ずれた位置，半径 R2 = R1 * 1.2.

問題とアルゴリズムの流れの分析



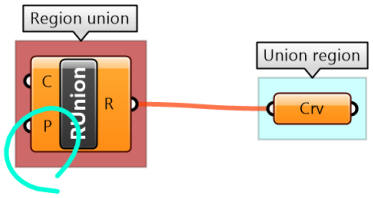
アルゴリズムの手順

Output

合成した領域の外周のカーブ

Key Process: 2 つの円の統合

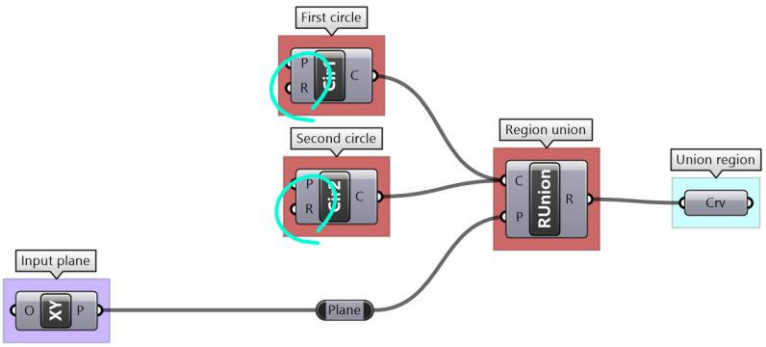
指定した平面上のカーブを合成する ***Region Union*** コンポーネントを使います.



Region Union への入力

必要となる入力を特定します.

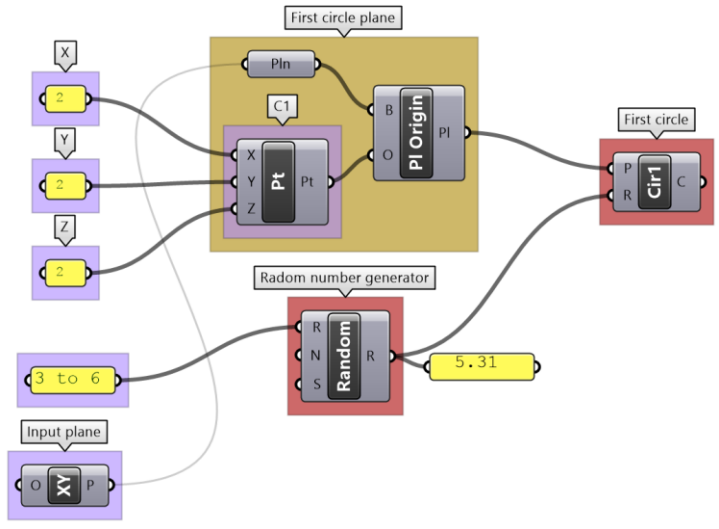
Region Union のための平面を与えます. 2 つの円はそれぞれ平面と半径が必要です. 平面の中心は円の中心になります.

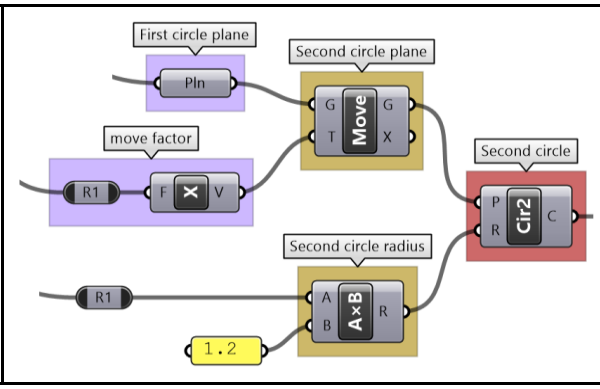
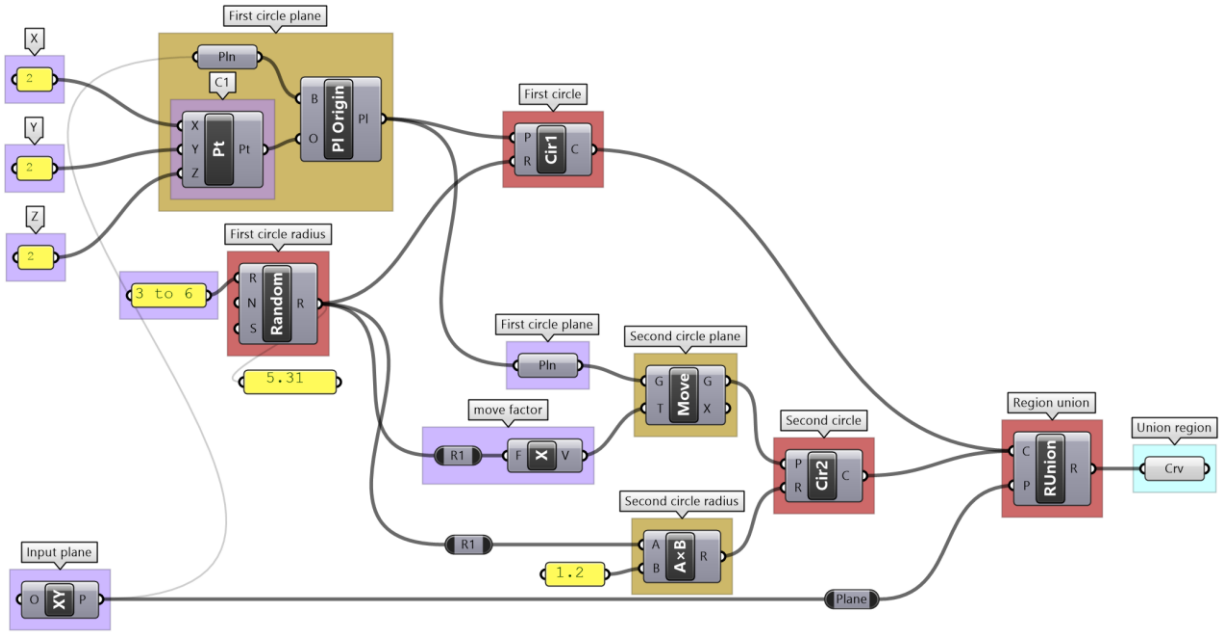


Intermediate process:
1 つ目の円の平面と中心点の生成

与えた座標から中心点を設定します. ***Plane Origin*** コンポーネントと ***XY-Plane***（デフォルトの中心点は原点）で平面を生成します.

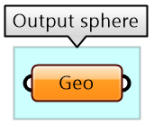
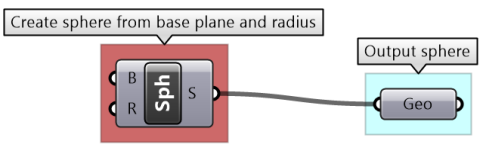
半径は 3〜6 の間からのランダムな数値です. ***Random*** コンポーネントを使います.



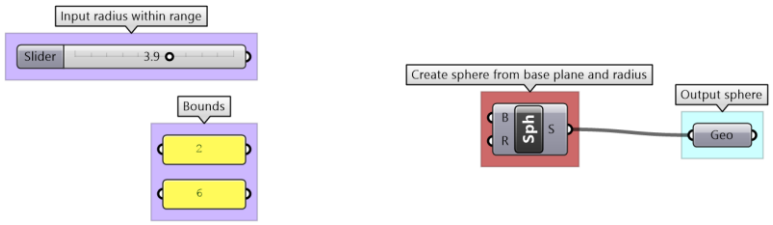
Intermediate process: **2 つ目の円の平面と中心点の生成** 1 つ目の円の平面を X 軸方向に 1 つ目の半径分移動して 2 つ目の円の平面を求めます. 2 つ目の円の半径は，1 つ目の半径を 1.2 倍して求めます.	
全体像 	

1_9_2: 球の大きさを制限

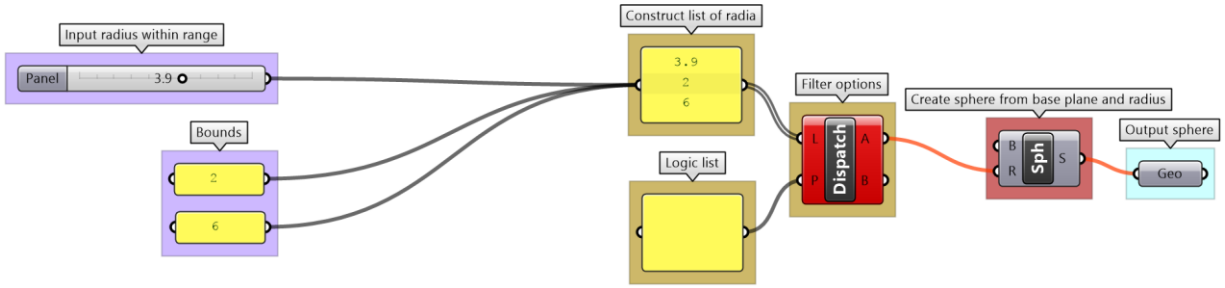
4 プロセスの考え方から，半径が 2〜6 に収まる球を描画します．入力半径が 2 未満の場合は，半径を 2 に設定し，入力半径が 6 より大きい場合は，半径を 6 に設定します．確認のために，0〜10 に設定した ***Number Slider*** を使用して半径を入力します．整理，色分け，ラベル付けも忘れずに行います.

アルゴリズム作成のための 4 プロセス	
Output 球（Geometry パラメータを使用）.	
Key Process: 球の生成 ***Sphere*** コンポーネントで球を生成します.	

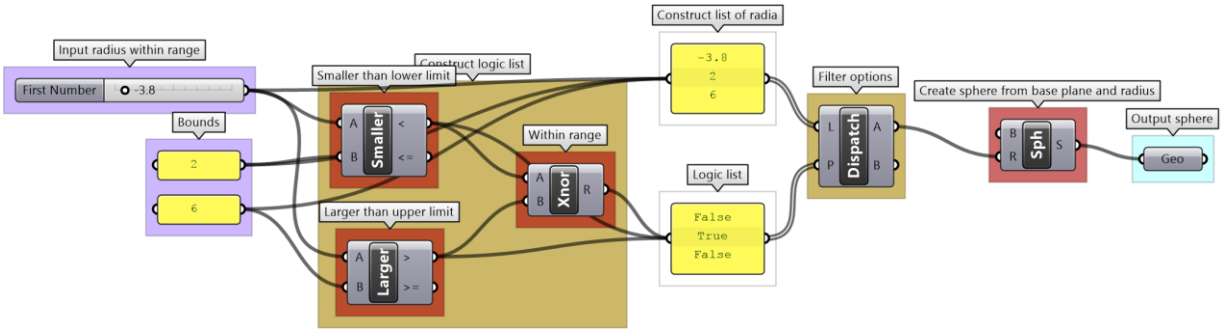
Input
1- 半径パラメータ(0 - 10).
2- 半径の下限・上限は， 2 と 6.



Intermediate processes #1
半径の取捨選択用パターンを生成するロジックを作成します．半径はスライダーの数値，下限，上限から成るリストになっています．パターンのリストは，正しい半径値を選べるように生成します．



Intermediate processes #2
選択のロジックでは，スライダーの値が下限・上限の間にあるかチェックします．間にある場合はその値が選ばれ，小さい場合は下限の値が，大きい場合は上限の値が選ばれます．



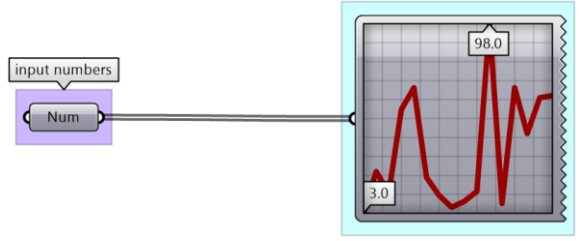
1_9_3: さまざまなデータ処理

次のように Number パラメータに埋め込まれた数値を考えます.
1-上限・下限と分布を確認して入力を分析する
2-データとその構造を表示する
3-偶数を抽出する
4-数値を降順に並べ替え
5-ソートされた数値を（100 から 200）に再マッピング

Solution

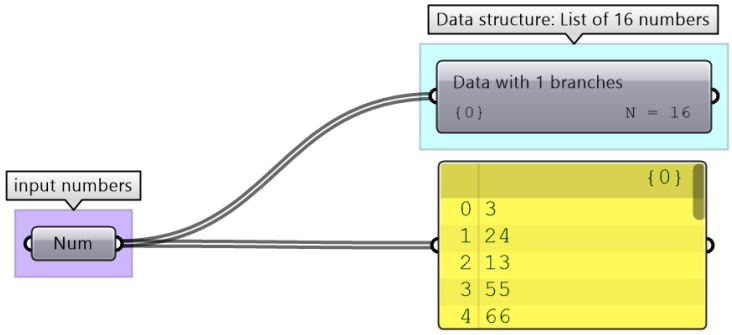
1- 入力の上限・下限・分布の分析

QuickGraph で見ると，入力した数値は，3～98 の間で無作為に分布していることがわかります.



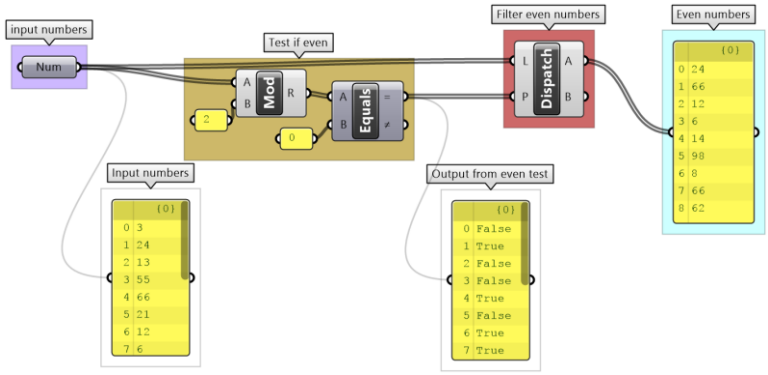
2- 入力のデータ構造と値の分析

Panel と ***Parameter Viewer*** で見ると，リストには 16 個のアイテムがあることがわかります.



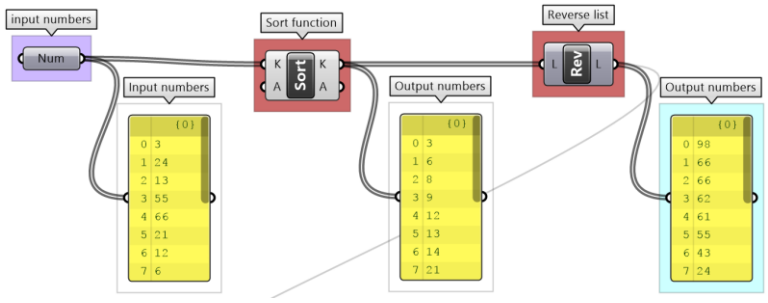
3- 偶数の抽出

数値が偶数かどうかをテストするロジックを作成し（余りなしで2で割り切れる），***Dispatch*** を使用して偶数を抽出します.



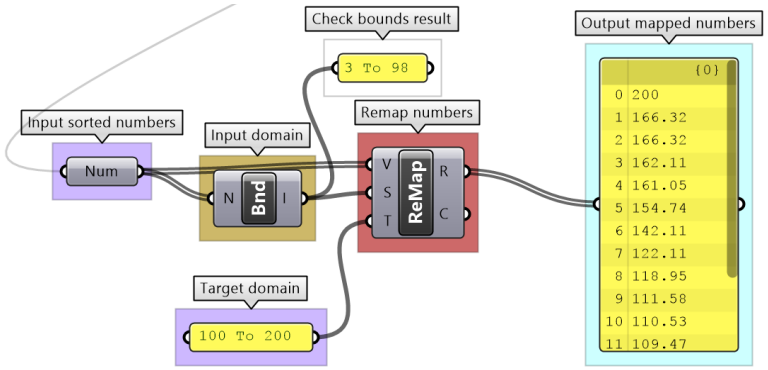
4- 番号を降順に並べ替え

Sort List は数値を昇順に並べ替えます．***Reverse List*** を追加することで降順にします.



5- 100-200 に再マッピング

入力をチェックし，***Remap*** コンポーネント でリストの数値を 100-200 に変換します.

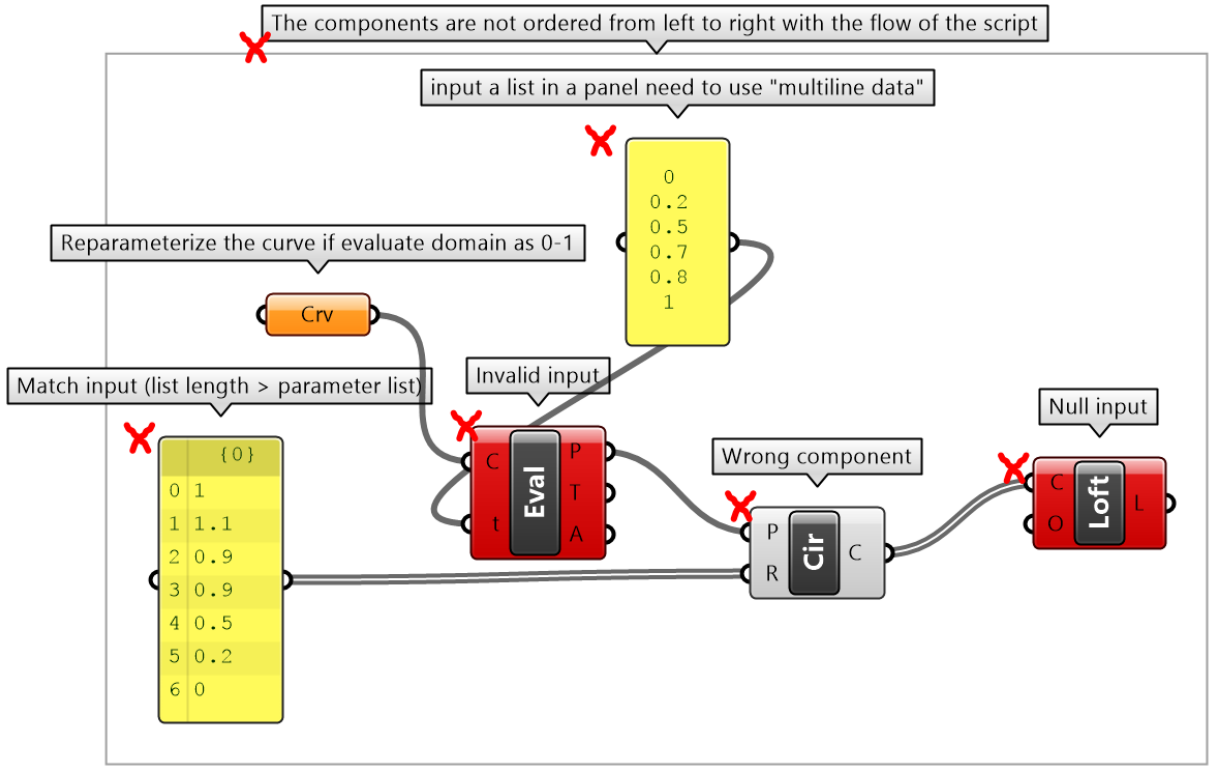


1_9_4: 落とし穴の回避

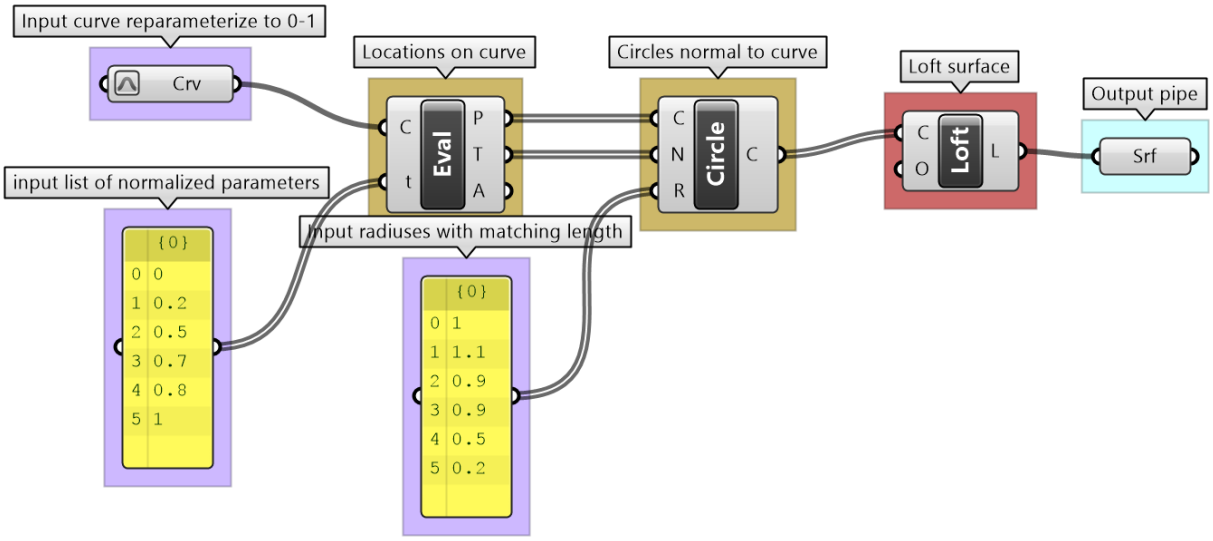
次のアルゴリズムが何を意図しているかを分析し，意図した通りに機能するのを妨げている誤りを特定してから，それらを修正するように書き直します．アルゴリズムの流れ，ラベル，色分けを活用して整理します.

Solution

問題点の確認:



エラーの修正とアルゴリズムの書き直し:



Chapter 2: Introduction to Data Structures（データ構造入門）

すべてのアルゴリズムでは，入力データを処理して，出力として新しいデータセットを生成します．データは，効率的にアクセス・処理できるように，明確に定義された構造内に格納されます．これらの構造を理解することは，アルゴリズム設計を成功させるための鍵になります．この章では，Grasshopper の基礎的なデータ構造について詳しく説明します．

2_1: 概要

GH には，単一のアイテム，アイテムのリスト，アイテムのツリーという 3 つの異なるデータ構造があります．GH コンポーネントは，入力データ構造に基づいて異なる方法で実行されるため，使用する前にデータ構造を把握することが不可欠です．GH でデータ構造の判別に役立つツールは，***Panel*** と ***Param Viewer*** です．

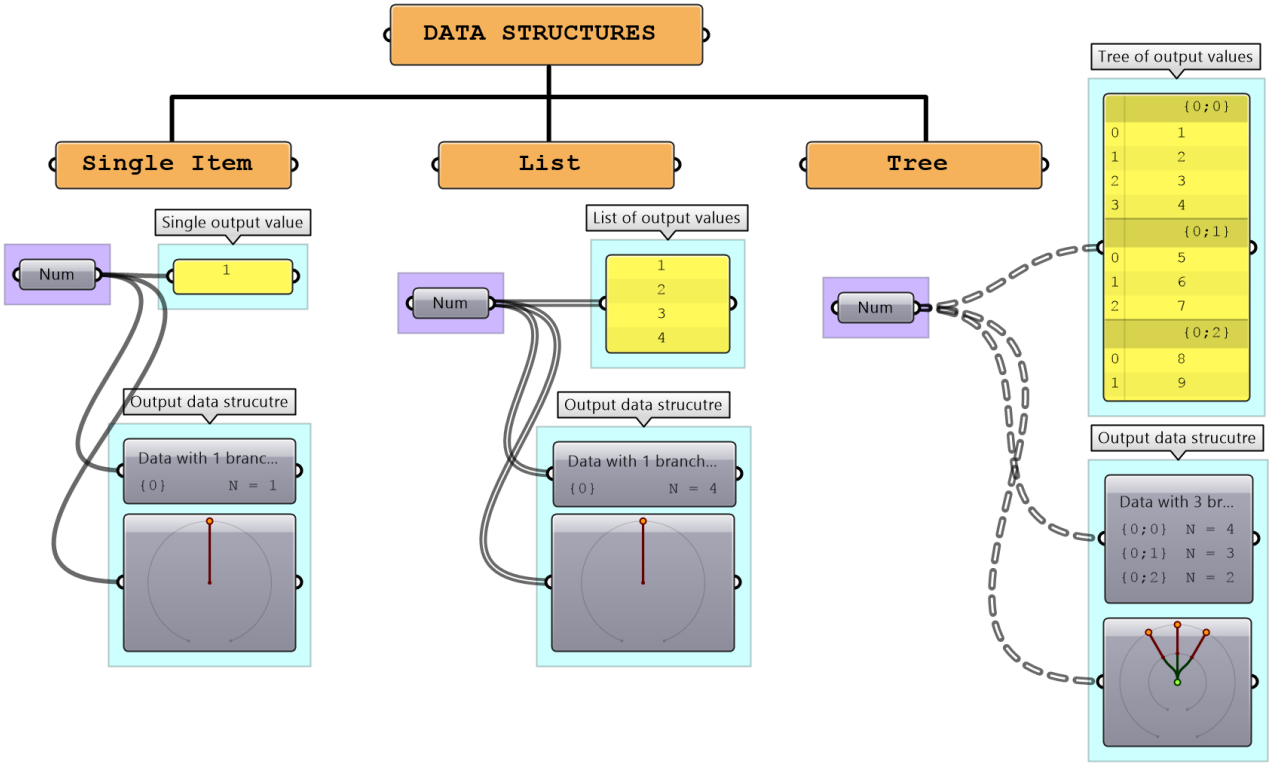


図 (34): GH のデータ構造.

GH の処理は，データ構造のタイプに基づいて異なる方法で実行されます．例えば，***Mass Addition*** コンポーネントは，リスト内のすべての数値を合計して 1 つの数値を生成しますが，ツリーを入力すると，それぞれのブランチ（枝分かれ）内の合計を表す数値のリストを生成します．

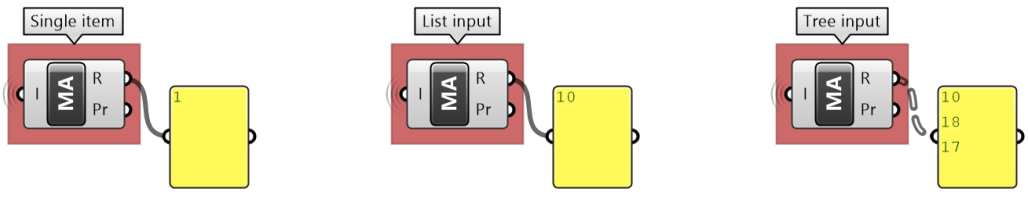
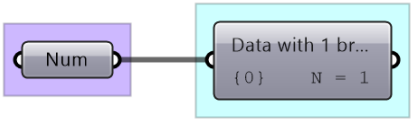
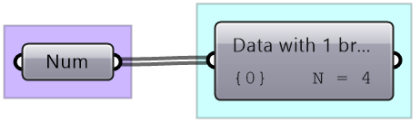
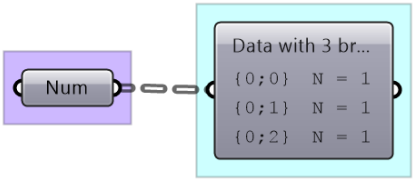


図 (35): データ構造により振る舞いの変わるコンポーネント． 図(34)の出力を合計した結果.

Display>Draw Fancy Wires を ON にしていれば，GH のコンポーネント間でデータを接続するワイヤーでデータ構造のタイプを視覚的に判別できます．単一のアイテムのワイヤーは単純な実線ですが，リストを接続するワイヤーは二重線として描画されます．ツリーデータ構造からのワイヤーは，二重の破線です．これらは，データ構造をすばやく特定するのに非常に役立ちます．

データ構造の表示	例
Item（アイテム）：単一ブランチの単一アイテム **ワイヤー表示**：単純な線	
List（リスト）：単一ブランチの複数アイテム **ワイヤー表示**：二重線	
Tree（ツリー）：複数ブランチとブランチ毎のアイテム **ワイヤー表示**：二重破線	

2_2: リスト生成

GH でデータのリストを生成する方法はたくさんあります．ここまでは，数値リストを Parameter または Panel（複数行データを含む）内に直接埋め込む方法を見てきましたが，リストを生成するための専用のコンポーネントもあります．例えば，数値リストを生成するものとして、***Range***，***Series***，***Random*** の 3 つの主要コンポーネントがあります．

Range コンポーネントは，最小値と最大値の範囲（所謂，ドメイン）をステップ数で等分した数値リストを作成します（リスト内の値の数は，ステップ数に 1 を加えたものと等しくなります）．

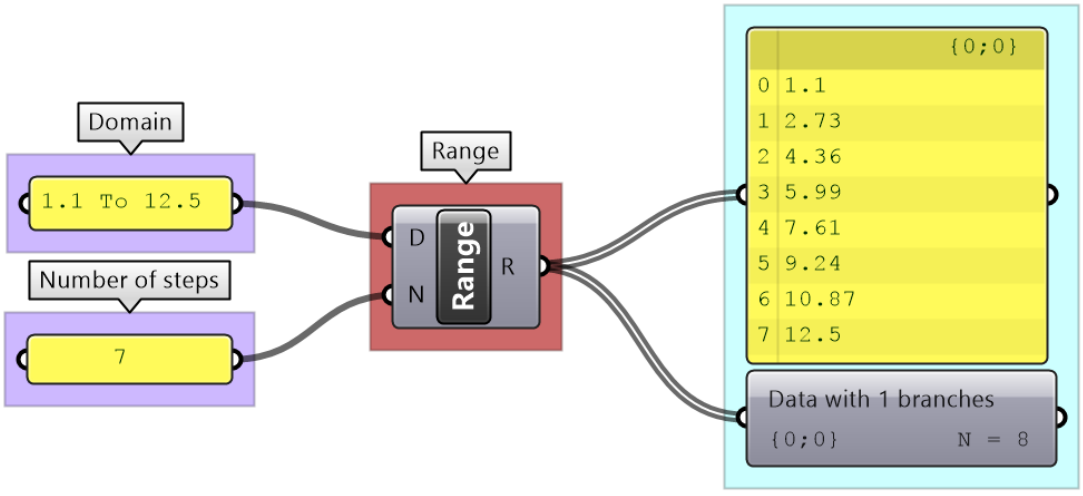


図 (36): GH の ***Range*** コンポーネントで 8 つの数値のリストを生成.

Series （数列）コンポーネントでも，等間隔の数値リストを生成しますが，このコンポーネントでは，初項（S）と間隔（N），要素の数（C）を設定します．

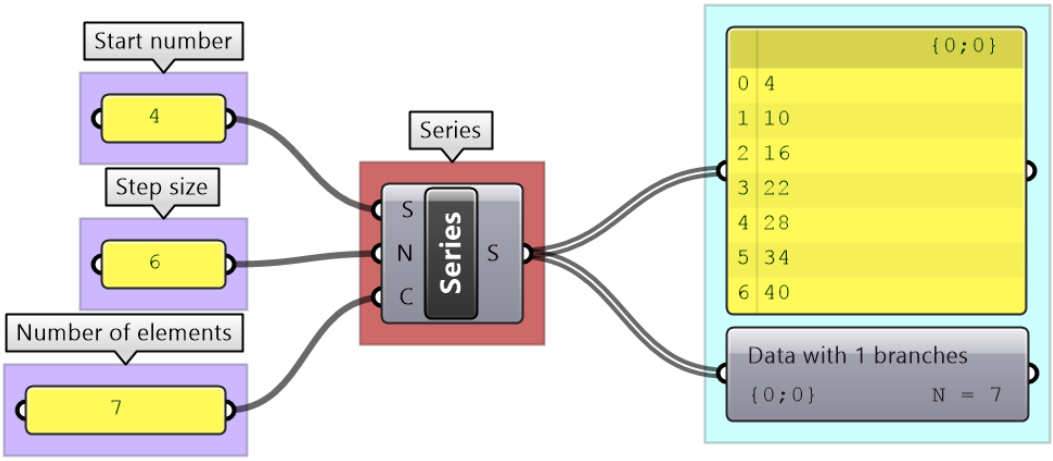


図 (37): GH の ***Series*** コンポーネントで 7 つの数値のリストを生成.

Random コンポーネント では，入力したドメイン（R）と要素数（N）からランダムな数値リストを生成します．同じシード番号（S）を指定すれば，いつも同じランダム数を得ることができます．

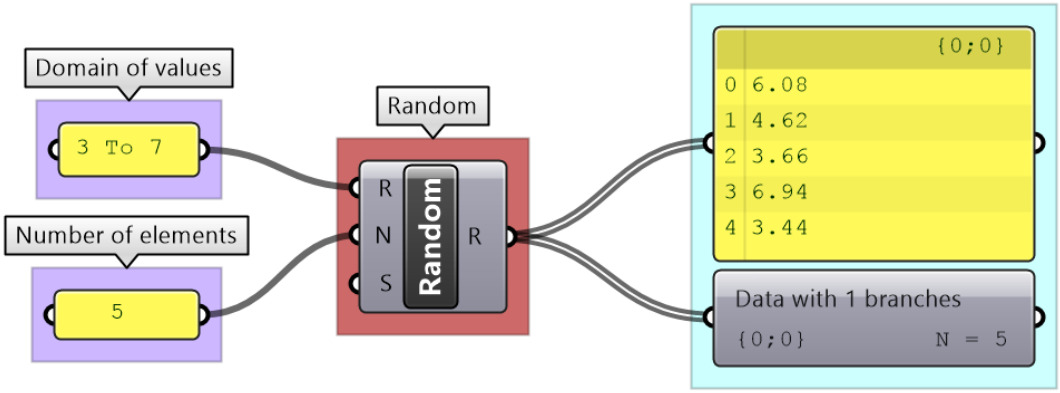


図 (38): GH の ***Random*** コンポーネントで数値リストを生成.

Divide Curve のような一部のコンポーネントでは，出力結果がリストとなることがあります（***Divide Curve*** の場合は，出力は，点，接線，パラメータのリストです）．***Panel*** コンポーネントを使用してリストの値をプレビューし，***Parameter Viewer*** を使用してデータ構造を調べます。

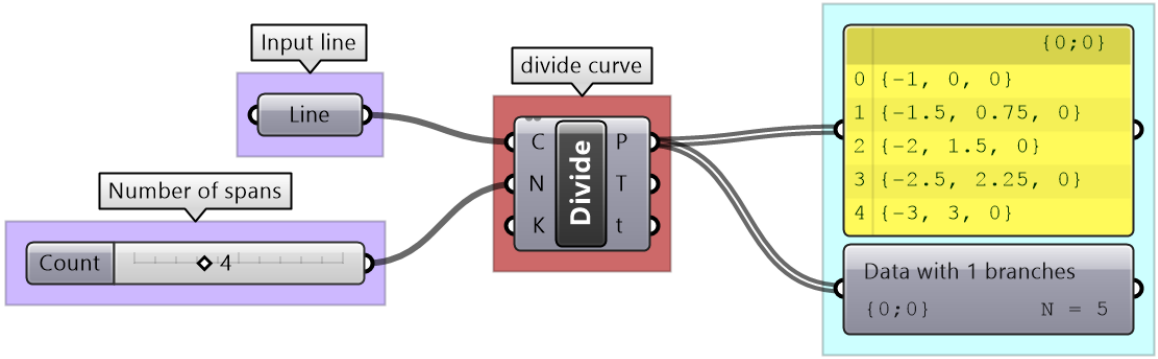
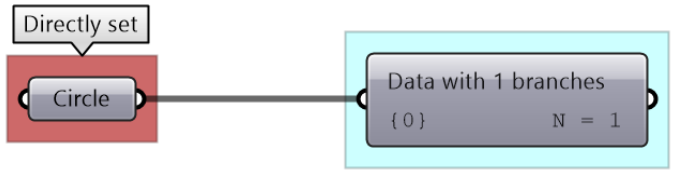
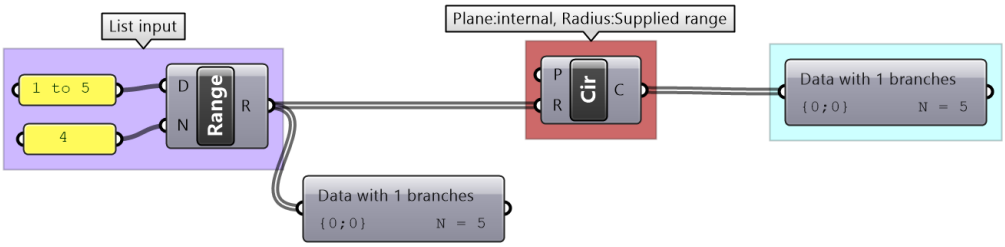
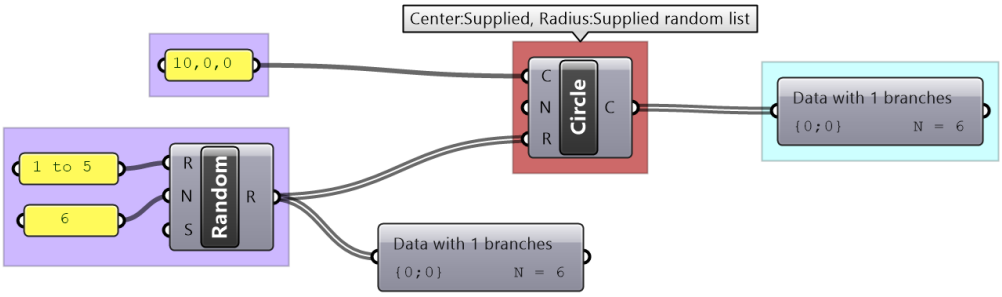
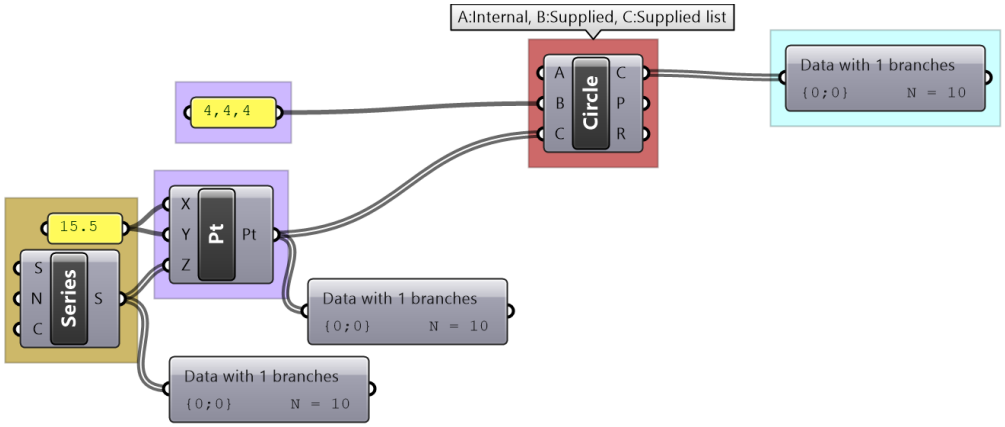


図 (39): ***Divide Curve*** に単一の入力 (curve) を与えるとリストの出力が生成されます.

2_2_1 リスト生成のチュートリアル

円を作成する 4 つの異なる方法を見比べてみます．さまざまなデータソースとデータ構造を使用します．

解説	Grasshopper solution
Parameter に直接，円を設定（Set）．	
平面は，内部的にデフォルトで設定された XY 平面を用います． ***Range*** コンポーネントで半径のリストを生成します．	
中心点（C）を一つ与えます． 法線（N）は内部のデフォルト値を用います． 半径のリストは，***Random*** コンポーネントを用います．	
3 点を通る円を作成します． A: 内部的に値を Set します． B: 1 つの点を入力します． C: ***Series*** コンポーネントで Z 座標のリストを作成し、点のリストを入力します．	

2_3: リスト処理

GH には，リスト操作やリスト管理のためのコンポーネントが幅広く用意されています．ここでは，最も一般的に使用されるものをいくつかを確認します．***List Length*** コンポーネントでは，リストの長さが確認でき，***List Item*** コンポーネントでは特定のインデックスのアイテムを抽出できます．

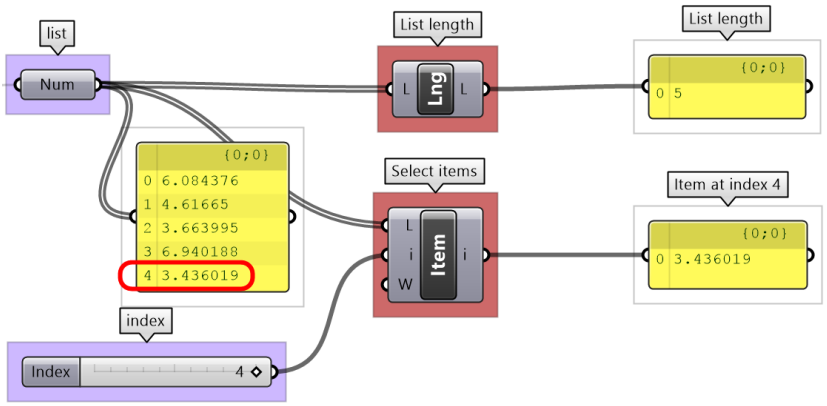


図 (40): GH のリスト操作の例.

Reverse List では，リストが反転され，***Sort List*** では，リストが並べ替えられます.

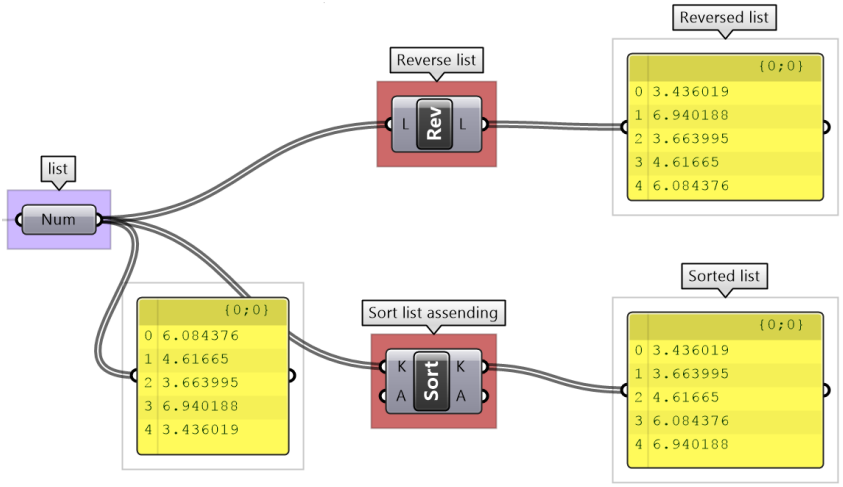


図 (41): GH のコンポーネントでリストの反転や並べ替えができます.

Cull Patterns や ***Dispatch*** のようなコンポーネントでは，パターンに基づいてリストを分割し，リストを部分的に抽出できます．これらはデータの流れの制御や部分的な抽出に非常に良く用いられます.

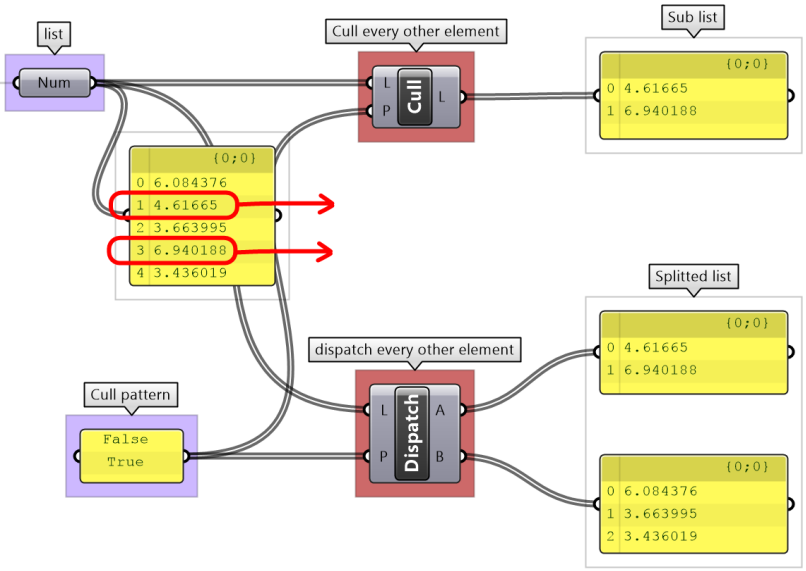


図 (42): ***Cull Pattern*** や ***Dispatch*** などでリストの一部を除外.

Shift List では，ステップ（S）に入力した値に従ってリストをずらします．これにより，複数のリストを特定の順序で一致させることができます．

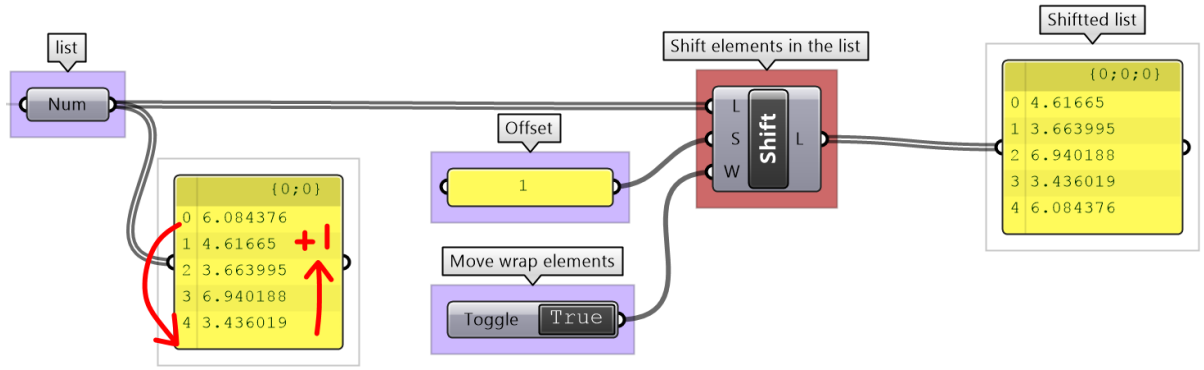


図 (43): GH の ***Shift*** のよる操作.

Subset は，入力したインデックス範囲に基づいて，リストの一部を抽出する別の例です．

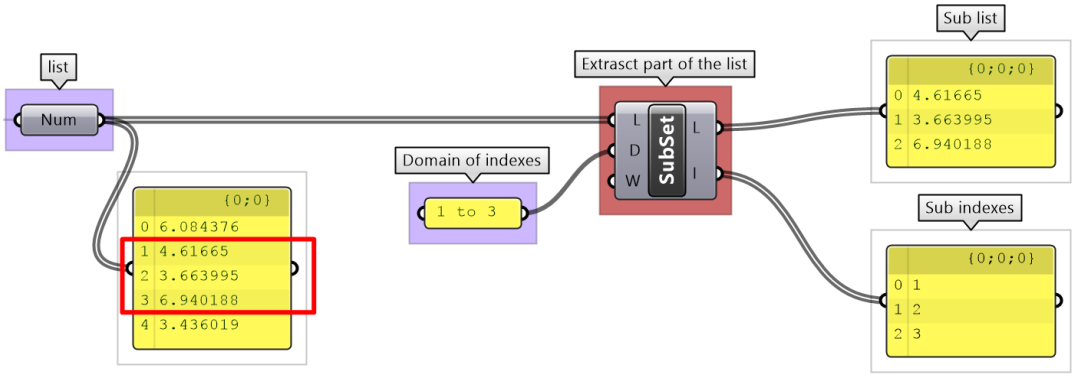
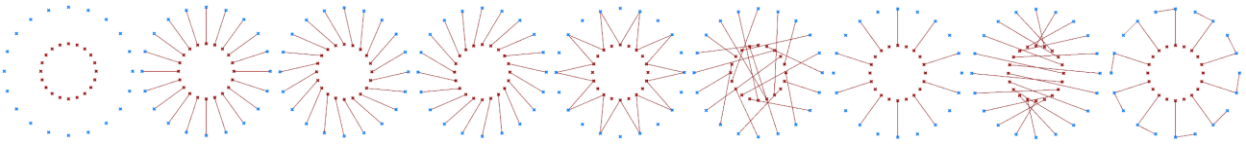
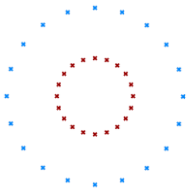
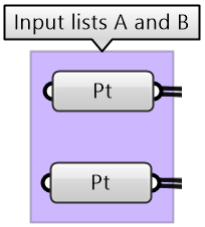


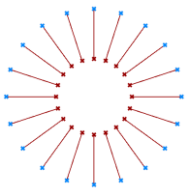
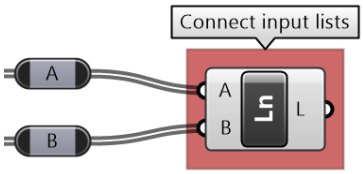
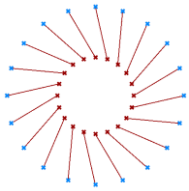
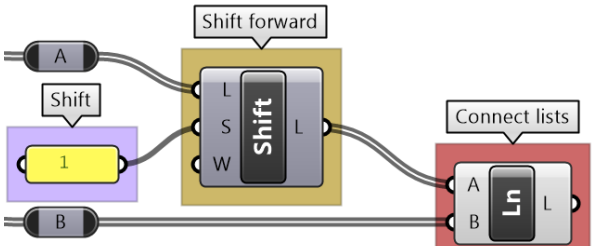
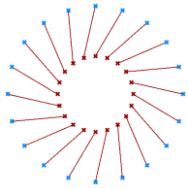
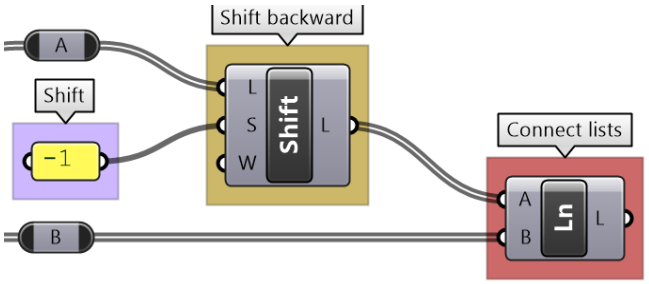
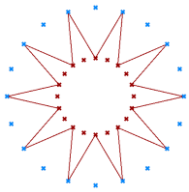
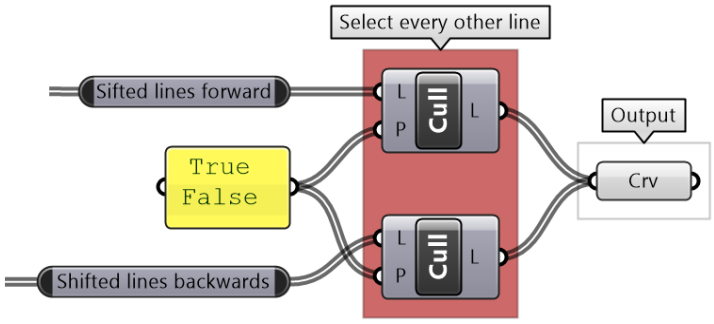
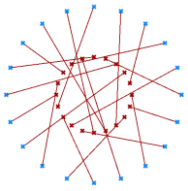
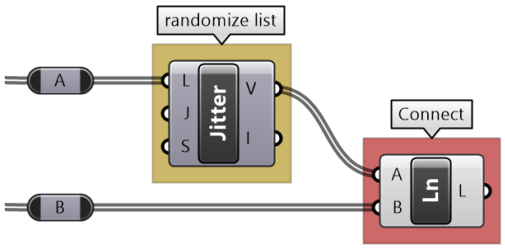
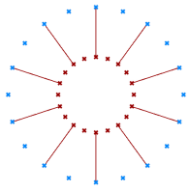
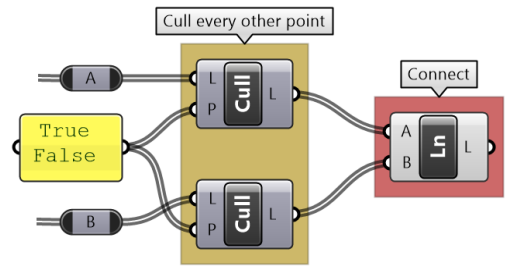
図 (44): インデックスの範囲を指定してリストのサブセットを抽出した例.

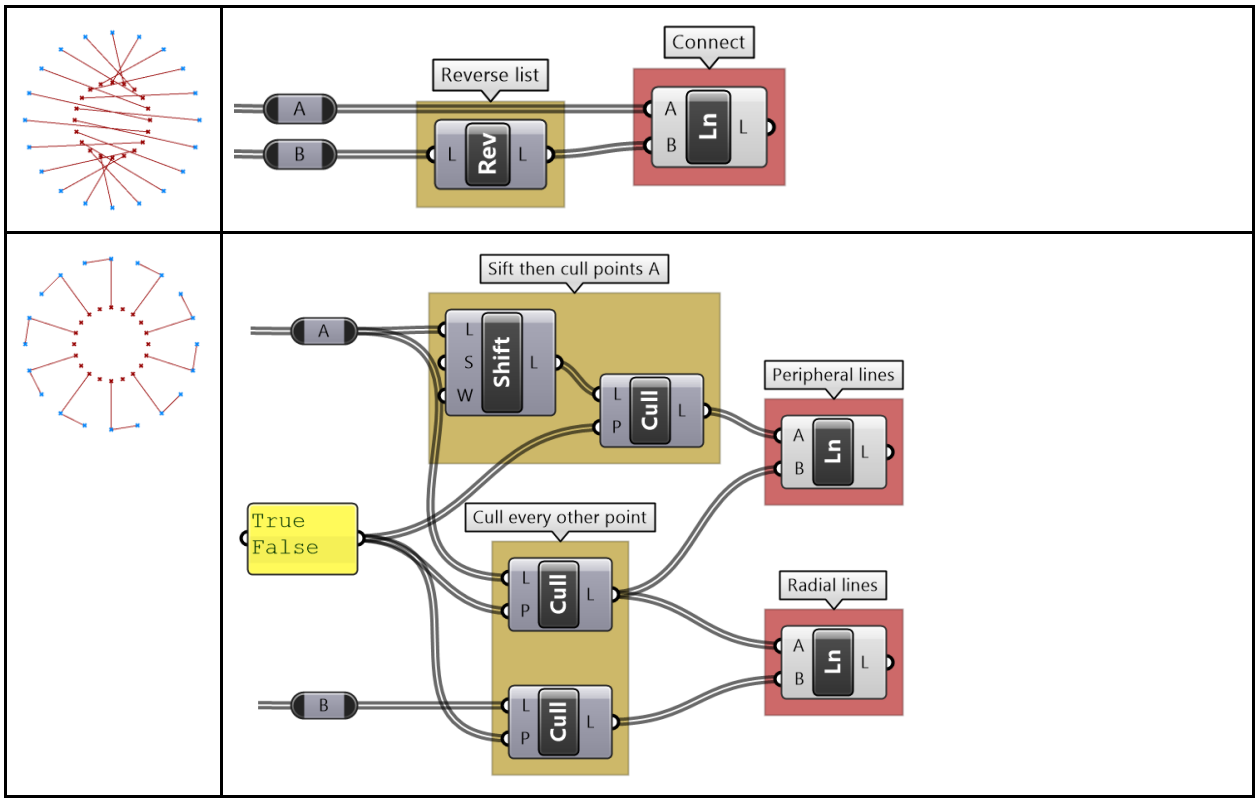
2_3_1 リスト処理のチュートリアル

2 つの点のリストを使って，次のようなイメージを生成します．



出力イメージ	Grasshopper solution
	

	 Connect input lists A B A B Ln L
	 Shift forward A Shift 1 B L S W Shift L Connect lists A B Ln L
	 Shift backward A Shift -1 B L S W Shift L Connect lists A B Ln L
	 Select every other line Sifted lines forward True False Shifted lines backwards L P Cull L L P Cull L Output Crv
	 randomize list A B L J S Jitter V I Connect A B Ln L
	 Cull every other point A True False B L P Cull L L P Cull L Connect A B Ln L



2_4: リストマッチング

すべての入力が，単一アイテムの場合，またはアイテム数が同じ単純なリストの場合，データがどのようにマッチングするか（組み合わされるか）は簡単に予想できます．マッチングは，対応するインデックスに基づきます． ***Addition*** で，GH でのリストのマッチングを確認してみましょう．

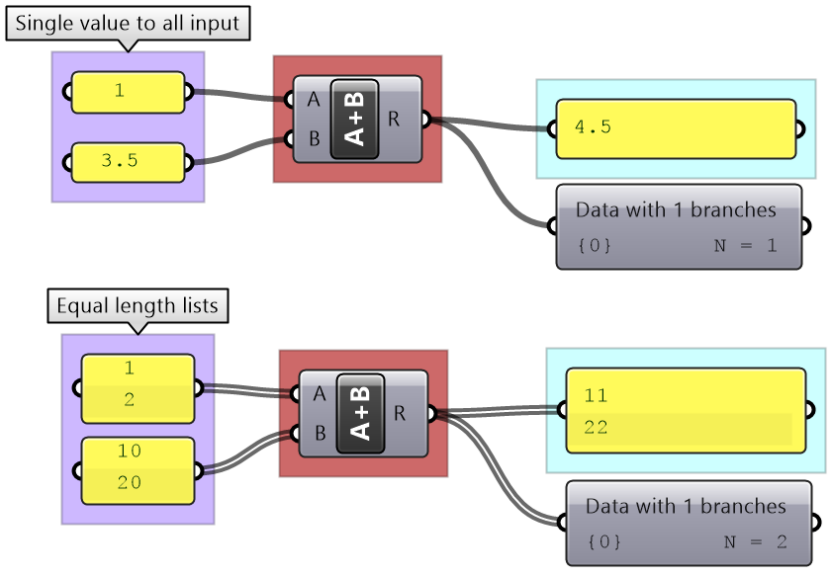


図 (45): リストの長さが同じ場合はインデックスに基づいてマッチングされます．

入力データの中に異なる長さのリストがある場合があります．この場合，GH では短いリストの最後のアイテムが再利用され，長いリストの以降のアイテムと組み合わされます．

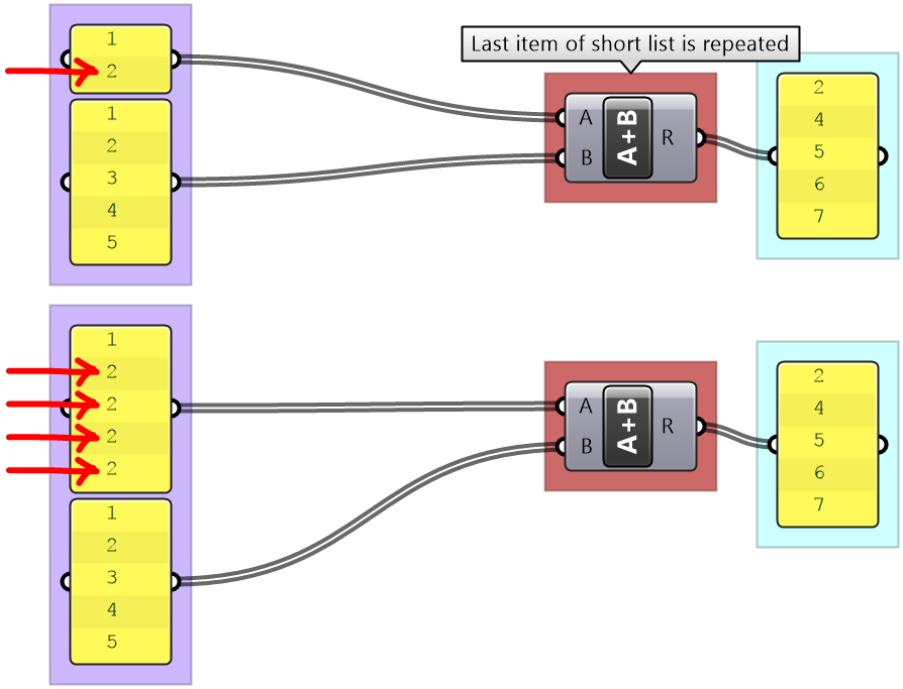


図 (46): GH のデフォルトのリストマッチングでは，短いリストの最後のアイテムが使い回されます.

GH では，任意のデータマッチング方法に切り替えられるよう，***Long***，***Short***，***Cross Reference*** というコンポーネントが用意されています．***Long*** は，デフォルトのマッチング方法と同じです．つまり，長いリストの長さに合わせるために，短いリストの最後のアイテムが繰り返されます.

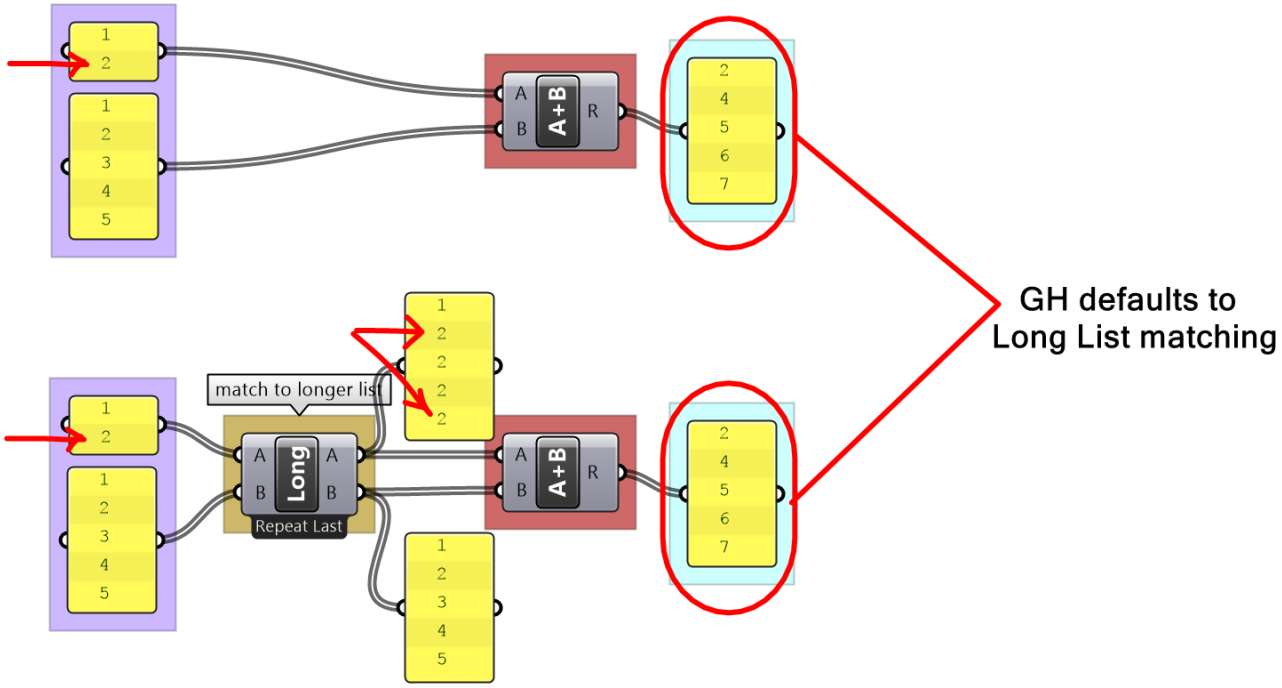


図 (47): ***Long*** によるリストマッチングは GH のデフォルトのマッチング方法です.

Short によるリストマッチングでは，長いリストを切り捨て，短いリストの長さに合わせます．過剰分のアイテムはすべて無視され，結果のリストの長さは短い方のリストと同じになります.

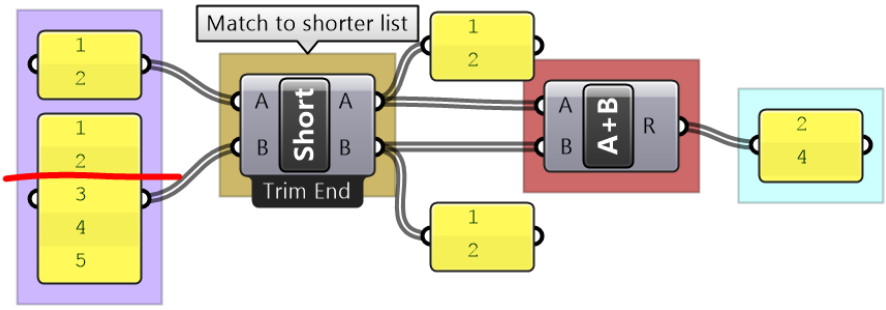


図 (48): ***Short*** によるリストマッチングでは，長い方のリストの余分なアイテムは省略されます.

Cross Reference は，最初のリストと 2 番目のリストの全アイテムをそれぞれマッチングします． 結果のリストの長さは，入力リストの長さの積と同じです．***Cross Reference*** は，入力データのすべての可能な組み合わせを生成したいときに役立ちます．図（49）からわかるように，入力の順序は結果の順序に影響します.

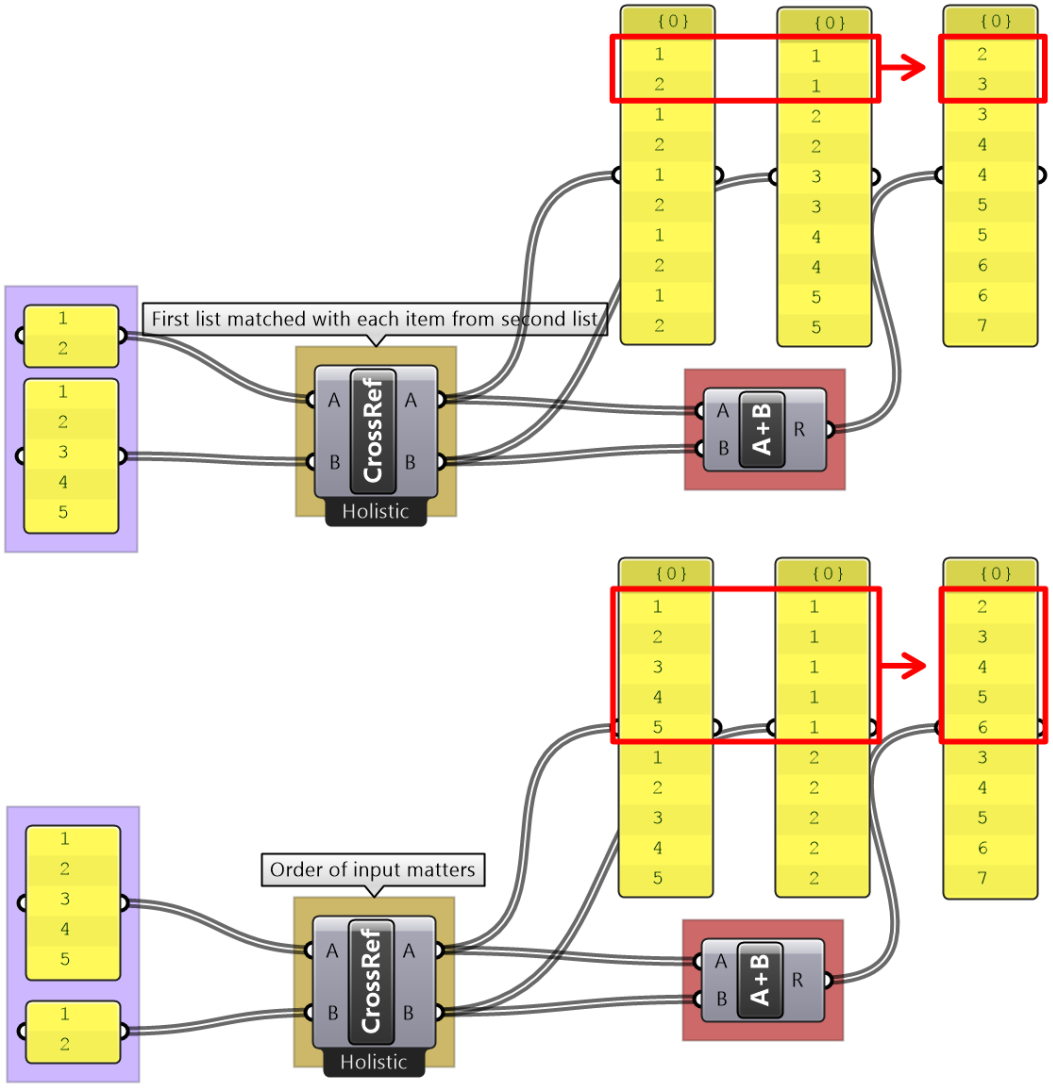


図 (49): ***Cross reference*** によるマッチングでは，すべての可能な組み合わせからリストを生成します.

目的の結果が生成されるマッチング方法がない場合は，目的に応じて長さが一致するようにリストを明示的にカスタマイズする方法もあります．例えば，長いリストの長さに一致するまで短いリストを繰り返したい場合は，以下の例のようにそれを実現するロジックを作成すれば良いです．

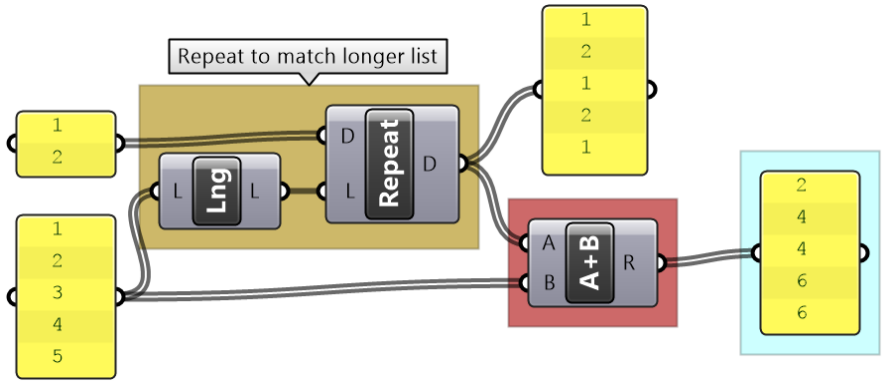
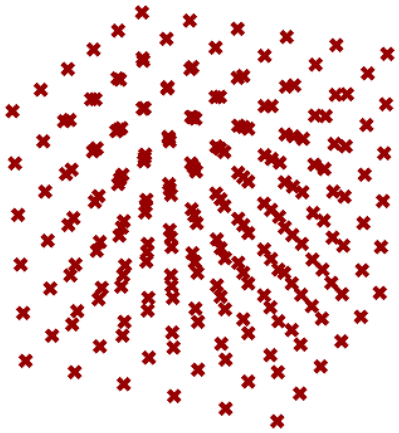
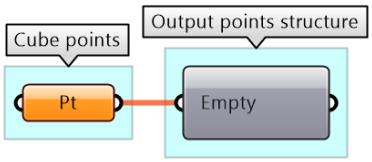
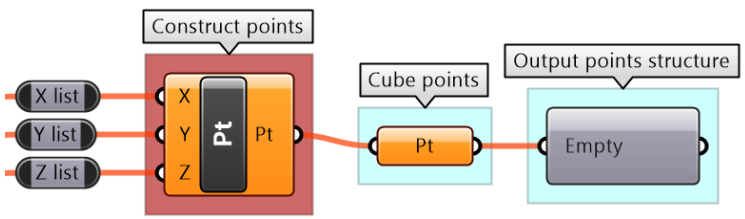


図 (50): その他のマッチングは，コンポーネントを組み合わせるなどして自作する必要があります．

2_4_1 リストマッチングのチュートリアル

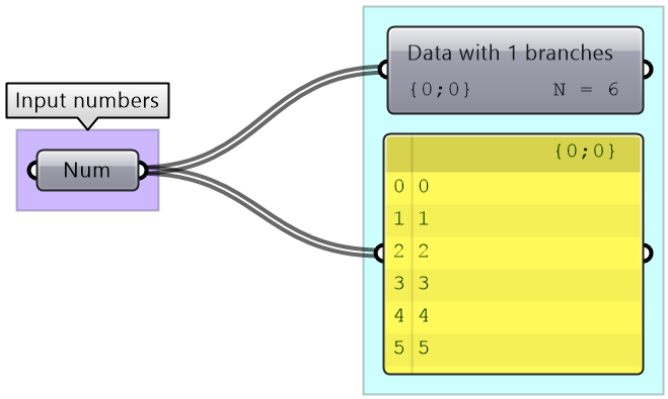
6 つの数値の入力リストを使って，図のような点群を生成します．



Solution	
Output: 全部で 6x6x6 = 216 点の点のリストを，XYZ の座標値のリストから作成します．	
Key process: ***Construct Point*** コンポーネントから点のリストを生成します．	

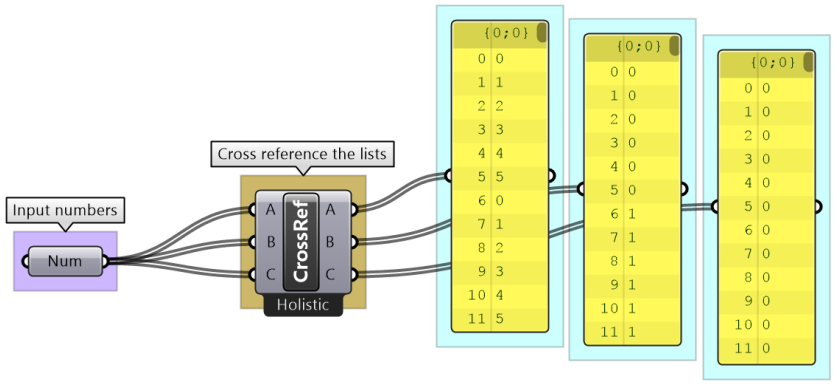
Input:
Parameter Viewer と ***Panel*** コンポーネントで入力を確認します.

与えられたリストは 6 つの数値でそれぞれの軸方向の座標成分を表します.

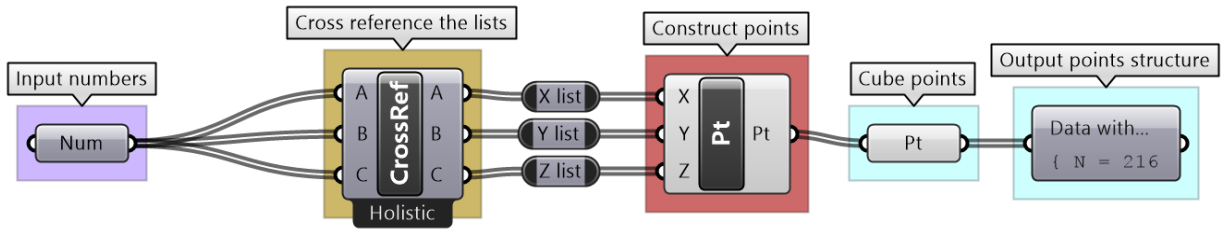


Intermediate process:
216 個の点を 3 軸方向にキューブ状に配置するためのすべての組み合わせを見つけます.

Cross Reference ですべての座標成分の組み合わせリストを生成します.



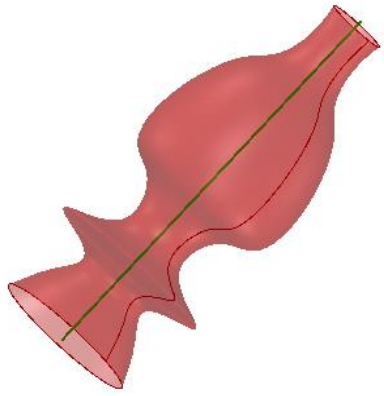
全体像

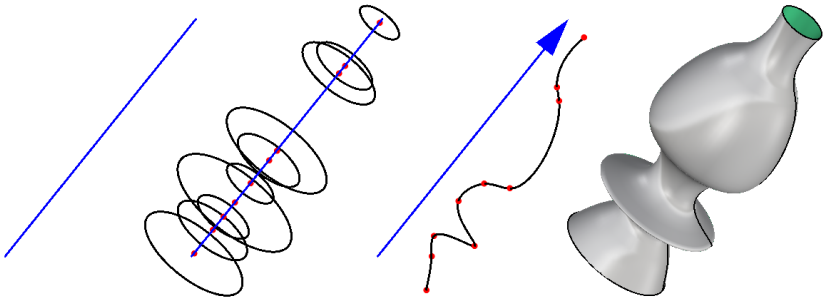
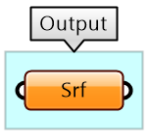
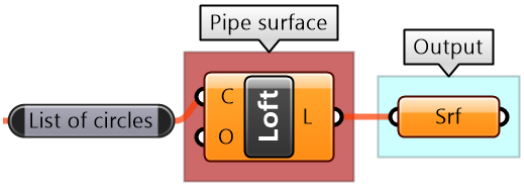
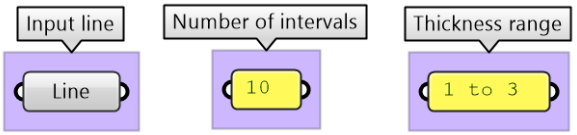
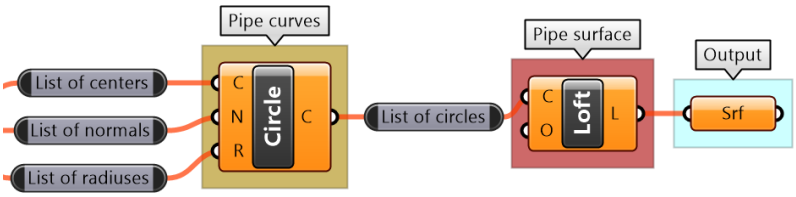


2_5: データ構造入門のチュートリアル

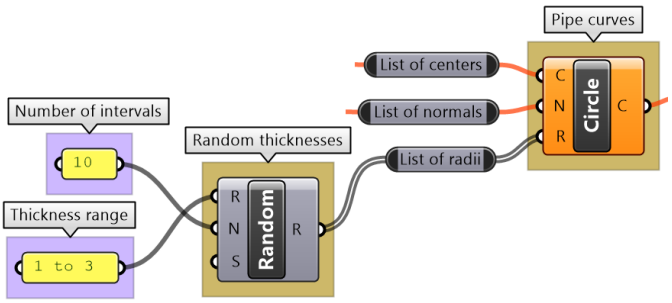
2_5_1: 太さが不均一なパイプ

カーブに沿って太さがランダムに 10 か所で変化する図のようなサーフェスを作成します. 太さの変化は 1～3 の間でランダムです.



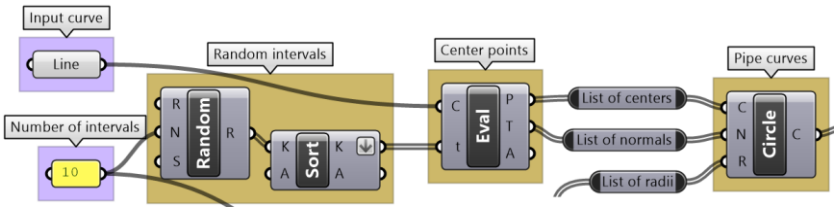
アルゴリズムの分析	
アルゴリズムを理解するには，3D モデリングの考え方が役立ちます．このサーフェスを生成するには 2 つの方法があります． 1-カーブに沿ってランダムな半径でランダムな位置に**円**を作成し，結果を**ロフト**． 2-**輪郭曲線**を軸に対して回転． 最初の処理は，以下の通りです． 1-ランダムな位置で線を分割． 2-その位置に**平面**を配置（直線が法線となる平面）． 3-**円**（または輪郭曲線上の点）を生成． 4-順番に円を選んで **Loft** します（または **Interpolate Curve** して **Revolve**）．	
アルゴリズムの手順	
Output: サーフェス．	
Key process: ***Loft*** コンポーネントでサーフェスを生成します．	
Input: ・直線 ・分割数 ・太さの範囲	
Intermediate process #1: ***Loft*** で複数の円からサーフェスを作成します．***Circle*** コンポーネントには，中心点，法線，半径を入力します．***Loft*** オプションはデフォルトで良いです．	

Intermediate process #2:
ランダムな半径のリストを作成します．***Random*** コンポーネントを使い，太さの範囲を入力します．

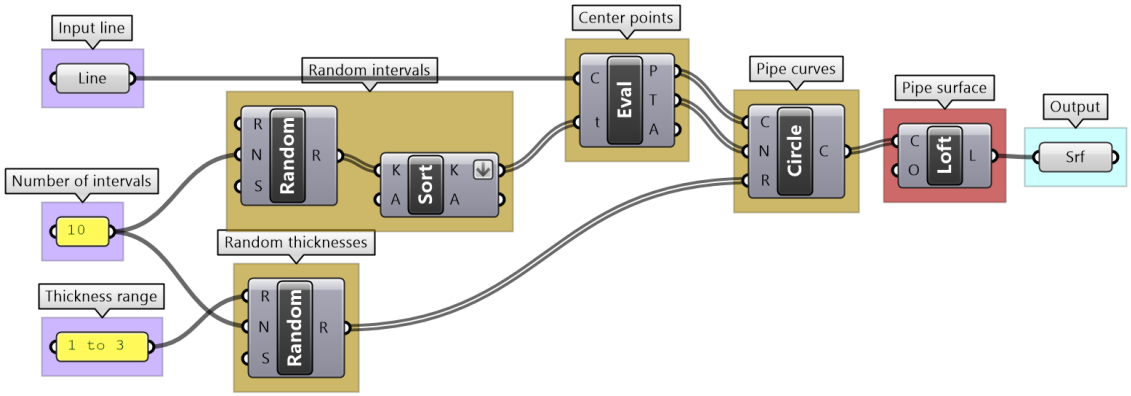


Intermediate process #3:
ランダムな間隔で直線を評価します．**Evaluate Curve** を使用して点と法線を抽出し，***Random*** を使用して曲線に沿ってパラメータを生成します．

問題点: ランダムなパラメータは順番に整列していません．***Evaluate Curve*** に入力する前に ***Sort List*** で並べ替えます．



全体像

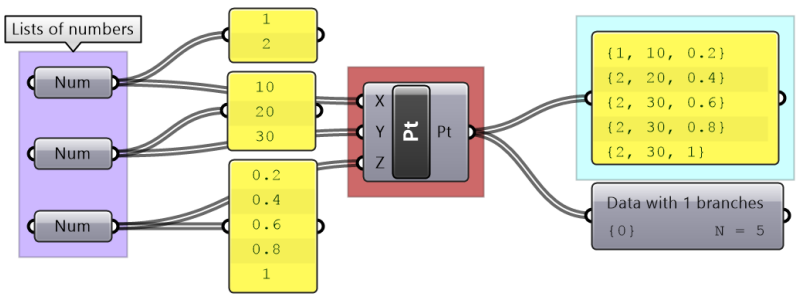


2_5_2: リストマッチングのカスタム

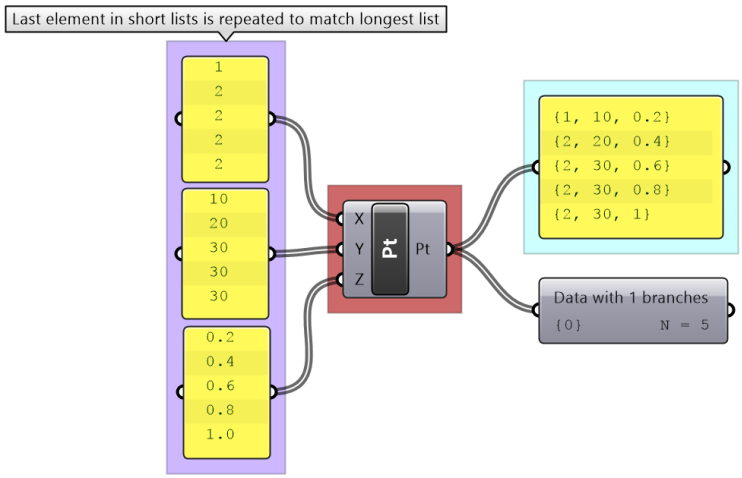
以下の例では，まずデフォルトの GH リストマッチングについて説明しています．次に結果を ***Short*** のマッチングと比較してから，短いリストのパターンを繰り返すようにマッチングをカスタマイズしてみましょう．例えば， [1,2]は，より長いリストの長さに合わせると，[1,2,1,2, ...]になります．

Solution

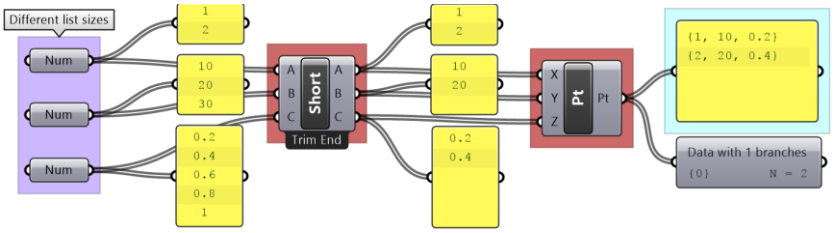
デフォルトの GH のマッチング:
マッチングを確認するため，リストを座標成分として ***Construct Point*** コンポーネントにつなぎ，結果を確認します．



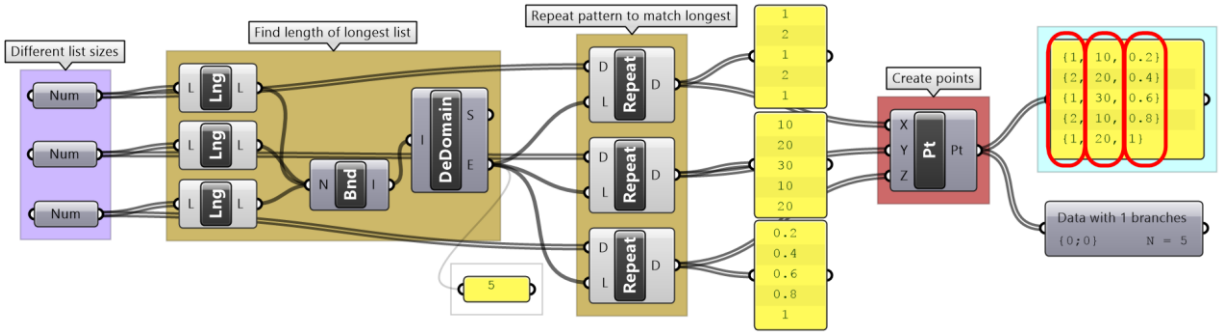
GH のデフォルトのマッチングの分析:
短いリストの最後のアイテムが，すべてのリストが同じ長さになるまで再利用され，アイテム同士はインデックス毎にマッチングします.



***Shortest List* (*Short*) のマッチング:**
長いリストの余分な値は省略され，すべてのリストは最短リストの長さに合わされます.

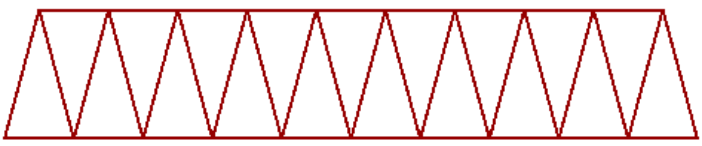
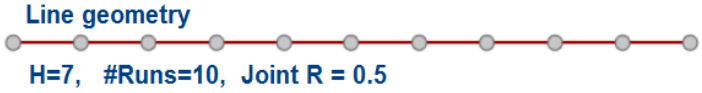


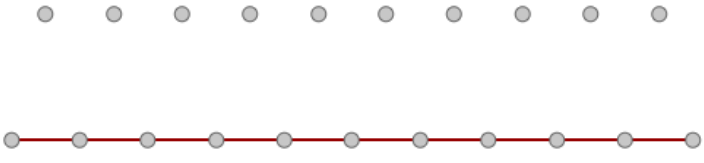
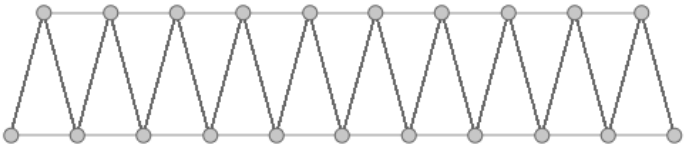
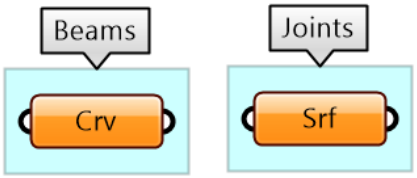
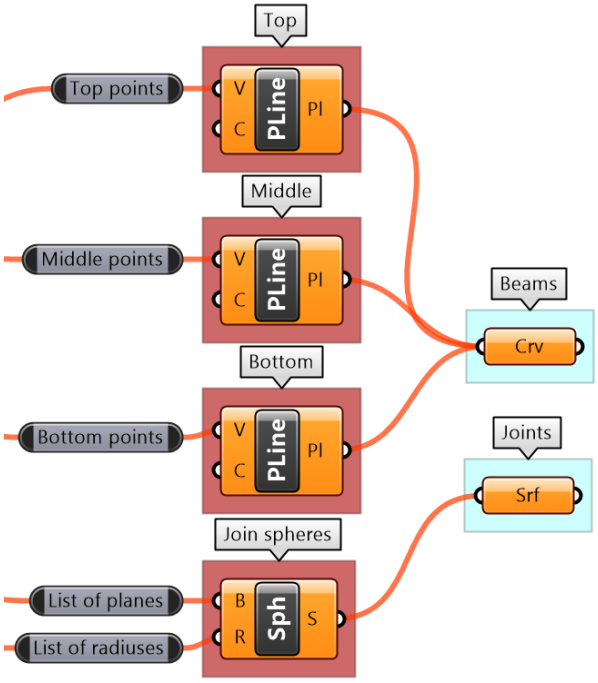
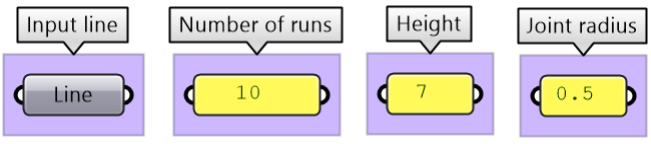
マッチングのカスタマイズ:
Repeat コンポーネントで最長リストの長さに合うまでアイテムを繰り返します.



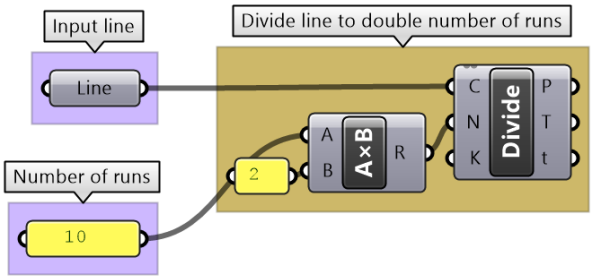
2_5_3: シンプルなトラス

図のようなシンプルなトラスを作成します．基準線，高さ，繰り返し数，ジョイント半径を指定します.

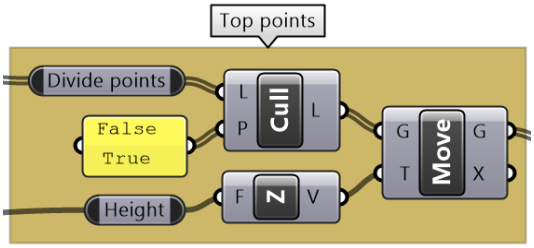
アルゴリズムの分析	
出力したいトラス形状を決めます.	
入力の定義 L= XY 平面の Line ジオメトリ H= 高さ R= 繰り返し数 J= ジョイント半径	

直線を 2×R 個に分割します.	
点を交互に，Z 方向に高さ H 移動します.	
下部の梁，上部の梁，および中間の梁のための 3 種類の順序付けした点のリストを作成し，それぞれをポリラインで接続します.	
GH のアルゴリズム実装	
Output: 曲線としての梁（ポリライン）と，球としてのジョイント（サーフェス）の 2 つの出力があります.	
Key processes: 上部・中間・下部の梁のためのポリラインを作成する必要があります. ***Polyline*** コンポーネントでそれぞれの点のリストを接続します. ***Sphere*** コンポーネントで ジョイントを作成します. 入力として分割点とジョイント半径を用います.	
Input: 直線，繰り返し数，高さ，ジョイント半径の 4 つの入力を与えます.	

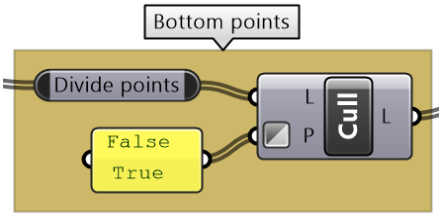
Intermediate process #1
繰り返し数の 2 倍の数で曲線を分割します．***Divide Curve*** を使い，***Multiply*** で繰り返す数を 2 倍します．



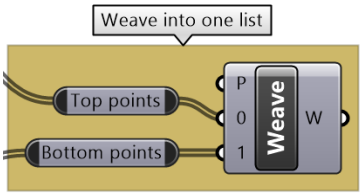
Intermediate process #2
上部の点を作成するには，すべての分割点のリストから 1 つおきに点を選択し，高さの量だけ垂直方向に移動します．
Cull Pattern で点を抽出し，***Move*** で垂直にずらします．



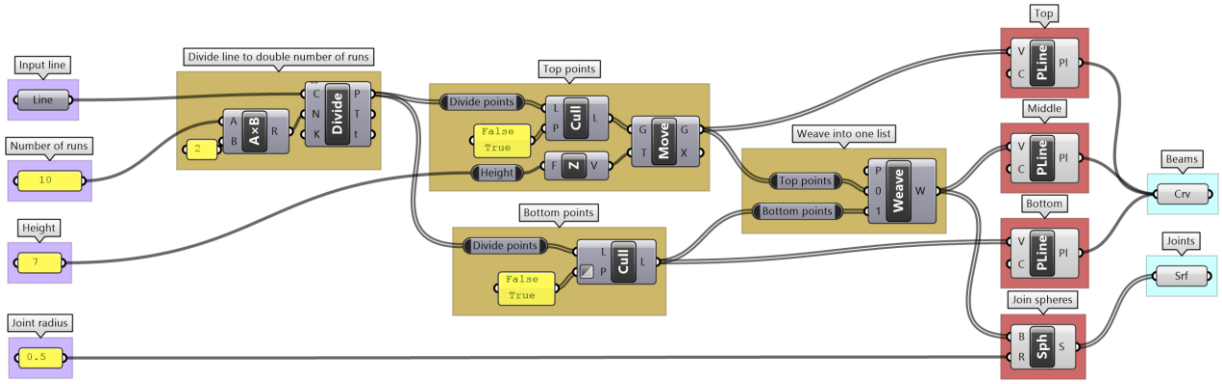
Intermediate process #3
下部の点を作成するには，上部の点を抽出に使用したパターンの反転パターンで，1 つおきに点を抽出します．
Cull Pattern で点を抽出します（Pattern 入力で ***invert*** オプションを設定します）



Intermediate process #4
中間梁の点を作成するために ***Weave*** で上部と下部の点を編み込みます．

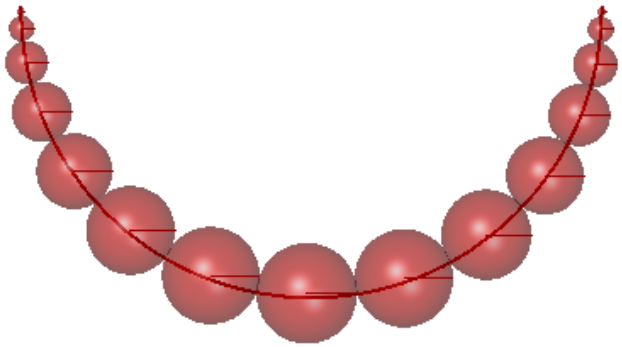


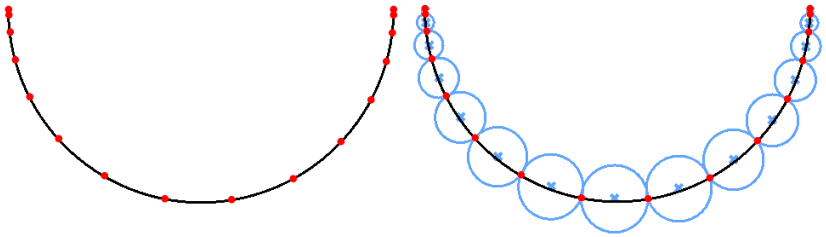
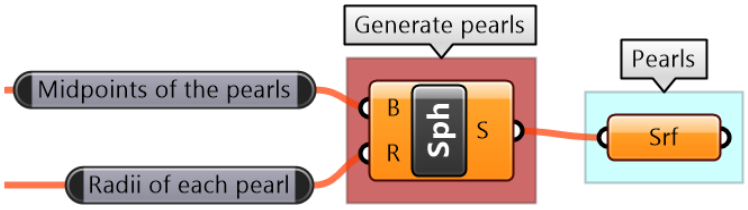
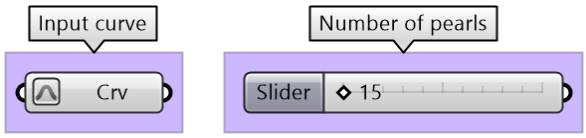
全体像



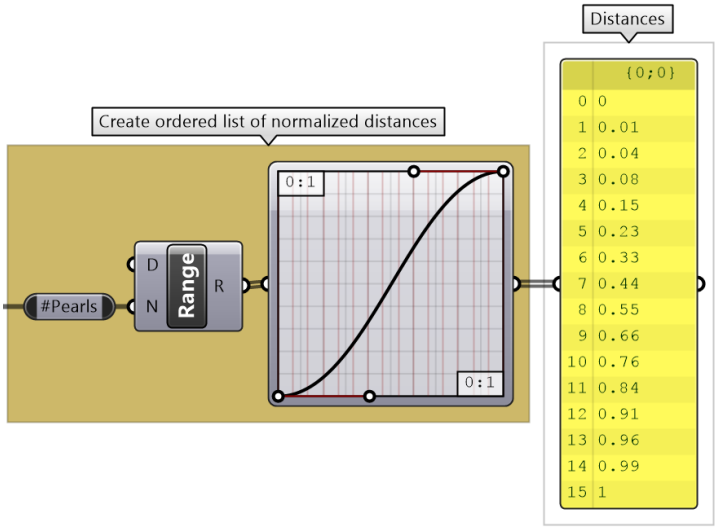
2_5_4: 真珠のネックレス

画像のように，真ん中に最も大きな真珠が 1 つあり，端に向かって徐々に真珠が小さくなるネックレスを作成します．真珠の数を 15 から 25 の間でパラメトリックに変更できるようにします．

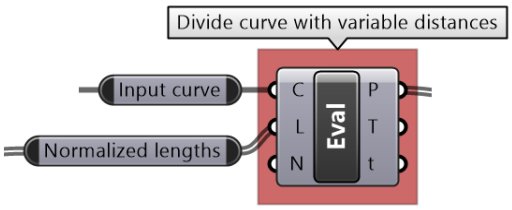


アルゴリズムの分析	
ネックレス作成のワークフローは，一般に次のようになります. 1- 曲線を長さが徐変するセグメントに分割します（中央が最も広く、端に向かって狭くなります）. 2- 各セグメントに分け，それぞれの中点を見つけます. 3- 長さの半分を半径として使用して，中心に球を作成します.	
アルゴリズムの手順	
Output: サーフェス.	
Key process: ***Sphere*** コンポーネントでサーフェスを生成します.	
Input: ネックレスの曲線, 真珠の数のパラメータ（ユーザーが変更可能）	

Intermediate process #1:
Range で等距離の数値リストを作成します．長さが徐辺するようにする必要があるので，***Graph Mapper*** コンポーネントを使用して間隔を制御します．



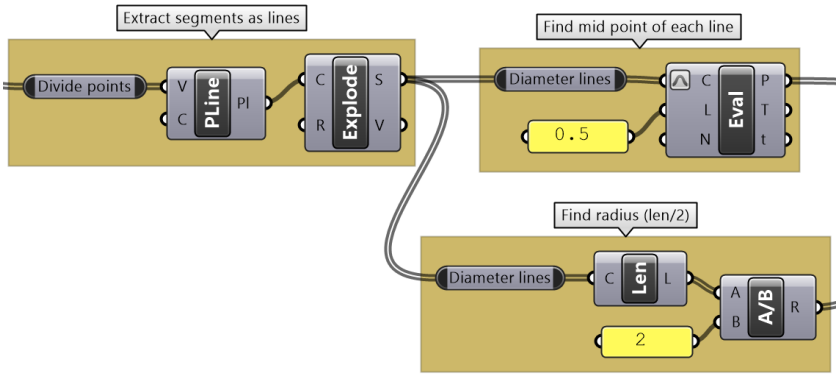
Intermediate process #2:
曲線の始点から距離は正規化（パラメータは 0〜1 の間）されているため，***Evaluate Length*** で分割点を見つけることができます．



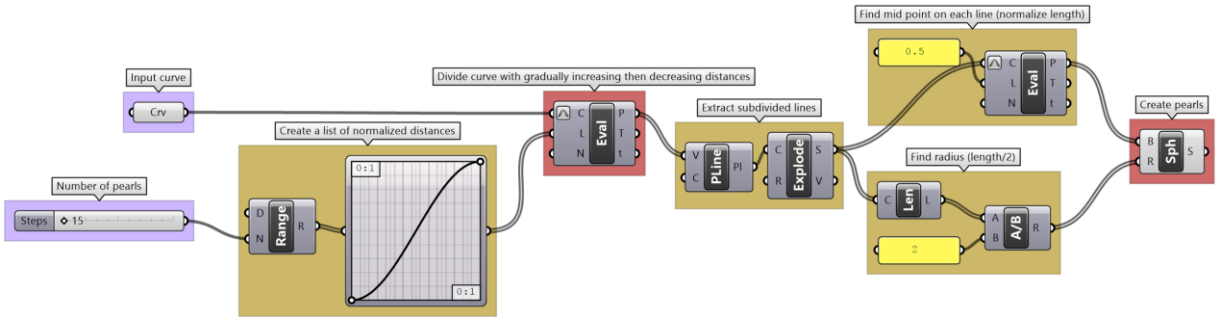
Intermediate process #3:
セグメントを生成します．***Polyline*** と ***Explode*** で点をセグメントに分割します．

中心点は，セグメントの中点として計算されます．***Evaluate Length*** で中間の長さを評価します．

半径は，各セグメントの長さの半分の値として計算されます．***Length*** と ***Division*** を使用します．



全体像



Chapter 3: Advanced Data Structures（データ構造応用）

この章では，GH のデータ構造の応用，つまりツリー構造とそれらを生成および使いこなすためのさまざまな方法について解説します．ツリー構造をいつどのように使用するか理解し，それらの活用・処理方法に効率良く慣れることを目指します．

3_1: Grasshopper のデータ構造

3_1_1 イントロダクション

プログラミングには，データの格納・アクセス方法を規定するための多くのデータ構造があります．最も一般的なデータ構造は，変数，配列，入れ子（ネスト）になった配列です．ほかには，データの並べ替えやマイニングなど，特定の目的に最適化されたデータ構造などがあります．しかし，Grasshopper では，データ格納のための構造はたった 1 つだけで，それが「**データツリー（Data Tree）**」です．ちょっと待って，じゃあこれまでに学んだ**単一アイテム（Item）**や**リスト（List）**は？と思われるかもしれません．GH では，これらもツリーのひとつです．単一アイテムは，1 つの要素（葉）が入った 1 本のブランチ（枝）を持つツリーであり，リストは，複数の要素が入った 1 本のブランチを持つツリーです．すべてのデータを 1 つの統一したデータ構造に収めるのは，実用的には非常に洗練されていますが，それと同時に，ユーザーはデータ構造が処理と処理の間でどのように変化するか，それが意図した結果にどのように影響するかについて，注意する必要があります．この章では，GH のデータツリーについて詳しく解説します．

3_1_2 データツリーの処理

Panel や ***Parameter Viewer*** でデータ構造が確認できます．それらをいろいろなコンポーネントにつないで，データの格納方法を見てみましょう．まず，単一アイテム入力からです．***Parameter Viewer*** には 2 つの表示モードがあり，1 つはテキスト，もう 1 つはグラフです．単一アイテムの入力は，アイテムを 1 つだけ持つ 1 本のブランチに格納されていることがわかります．

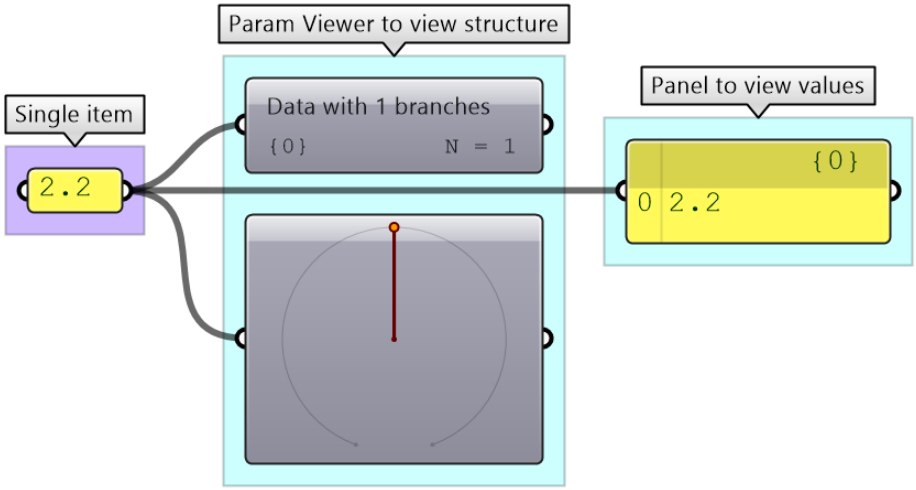


図 (51): GH のデータ構造は，さまざまな方法で確認できます．

Parameter Viewer には，図（52）に示すように，各ブランチのアドレス（「パス」と呼ばれます）とそのブランチのアイテム数が表示されます．

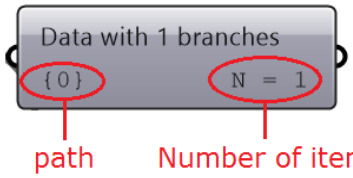


図 (52): ***Parameter Viewer*** は，各ブランチのパスのアドレスとアイテム数を示します.

リストは，図（53）のように 1 つのブランチを持つツリーとして表現されます．また，図（54）のように，3 つのアイテムを 3 つの異なるブランチに格納することもできます.

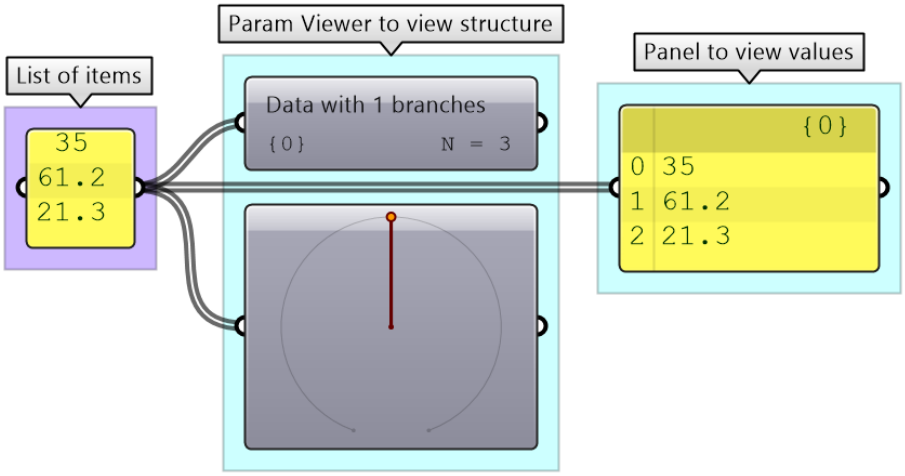


図 (53): リストは，複数アイテムの入った 1 本のブランチを持つツリーです.

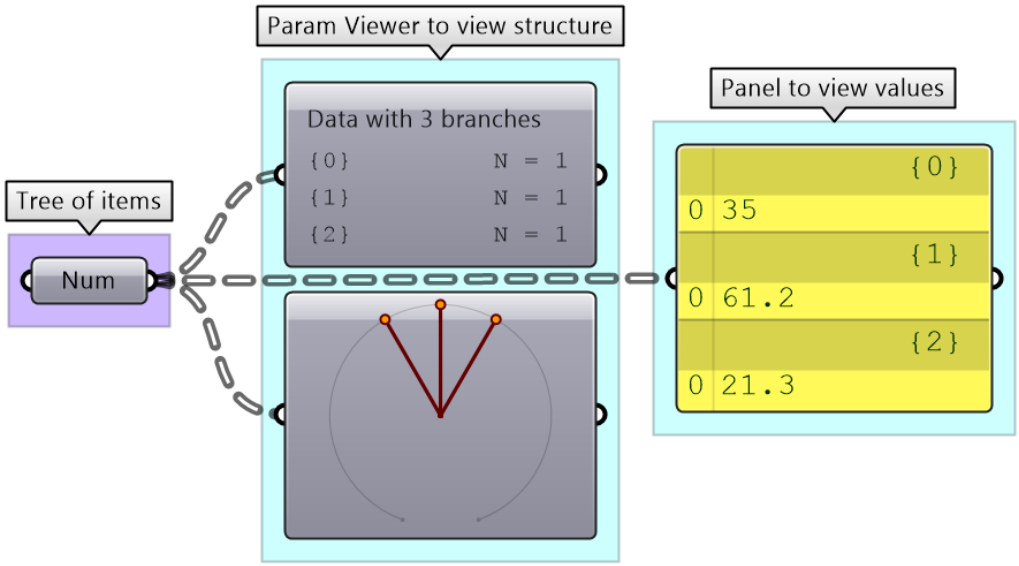


図 (54): ツリーは，1 本以上のブランチとそれぞれのブランチに 1 つ以上のアイテムを持ちます.

GH のデータ構造を理解する鍵は，次の問いに答えられるようになることです．**3 つの数値を，1 つのブランチにまとめて格納したときと 3 つの各ブランチに格納したときの違いは何か？**
データ構造は，入力値をどのようにマッチさせるかを GH コンポーネントに伝えます．言い換えると，コンポーネントは，入力データの構造によってデータの処理方法が変わる場合があるということです．次の例は，値のセットは同じで構造が異なる場合に，結果にどう影響するかを示しています.

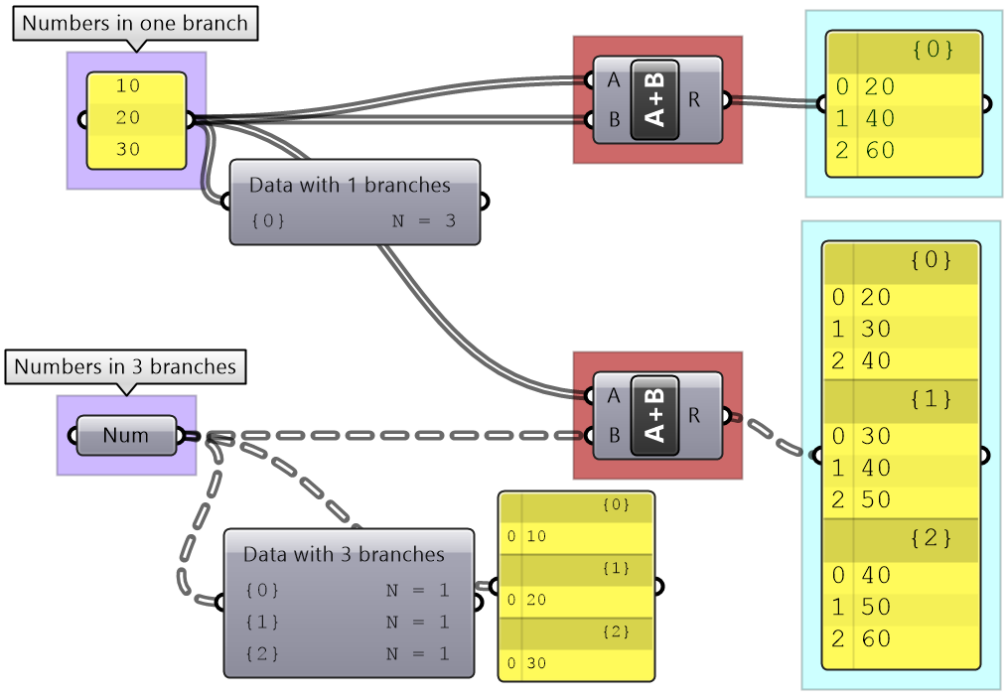


図 (55): 同じ値のセットでもデータ構造が異なると出力に影響します.

データツリーのマッチングについては後で詳しく説明しますが，GH コンポーネントでは，データ構造によって結果が大幅に変わる可能性があることに注意が必要なことはおわかり頂けたでしょう．これは，プログラミング言語で 1 つの統一したデータ構造を使用する際に出くわす複雑さの 1 つです.

3_1_3 データツリーの表記

データツリーを理解するための最初のステップは，GH におけるツリーの表記を知ることです．例からわかるように，ツリーはブランチから構成され，各ブランチはいくつかのアイテム（葉）を持ちます．各ブランチのアドレス，つまりパスは，「；（セミコロン）」で区切られた整数で表され，「{ }（波括弧）」で囲まれます．各アイテムのインデックスは「[ ]（角括弧）」で囲まれます．この図は，ツリー内の要素のアドレスの内訳を示しています.

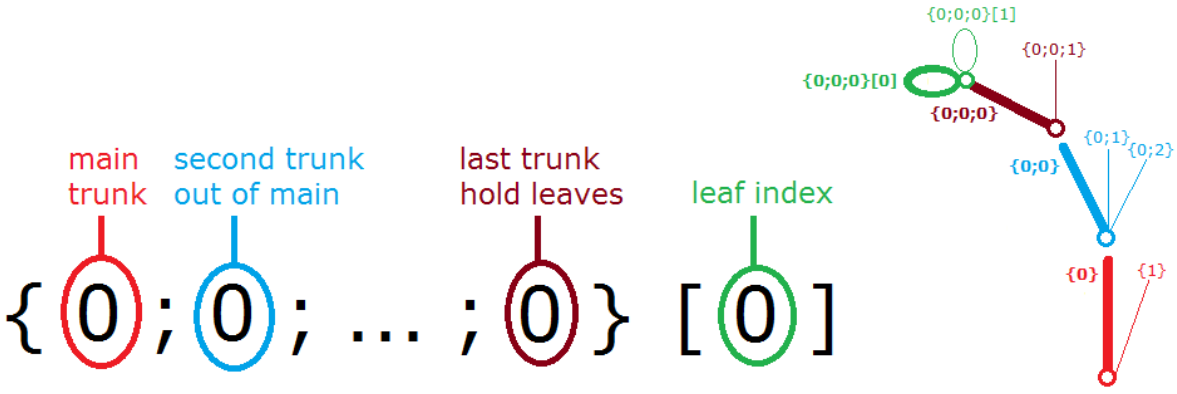


図 (56): 要素のアドレスは，枝（ブランチ）のアドレスと，枝に付く葉（アイテム）のインデックスから成ります.

さまざまなツリー構造とそれらが ***Paramster Viewer*** および ***Panel*** でどのように表示されるかの例を以下に示します.

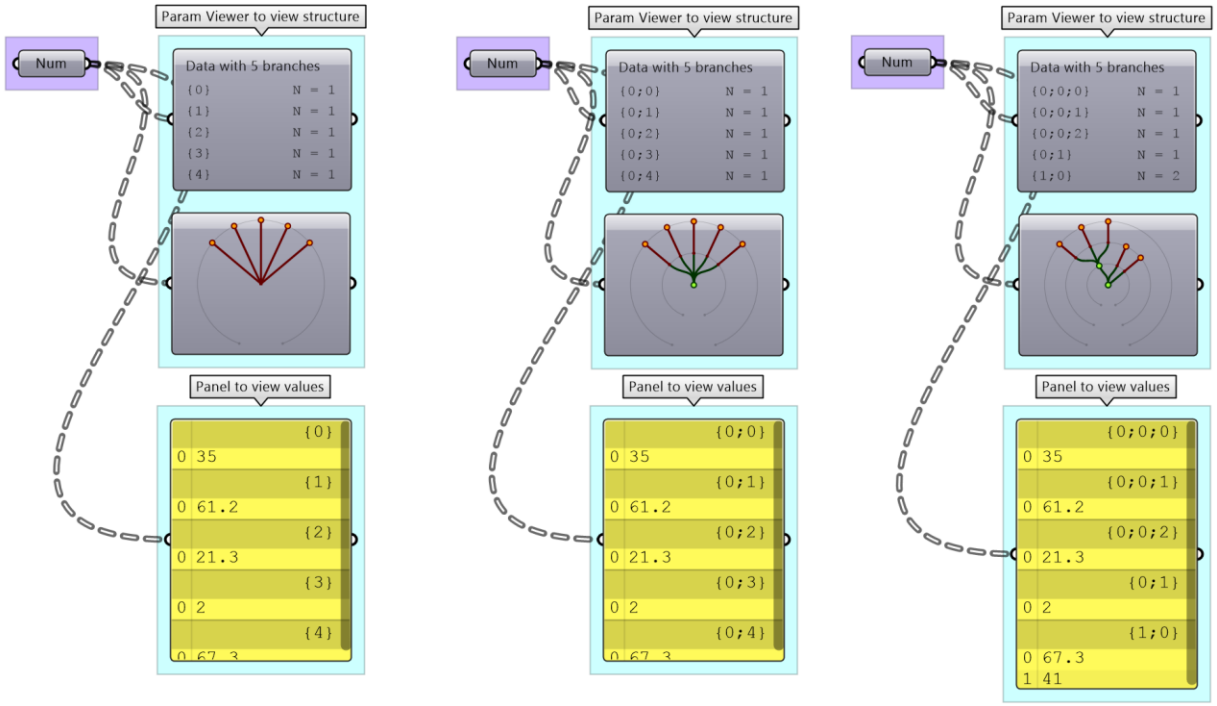
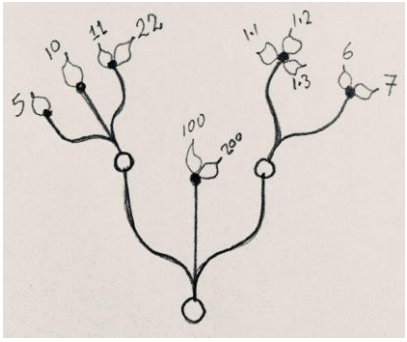
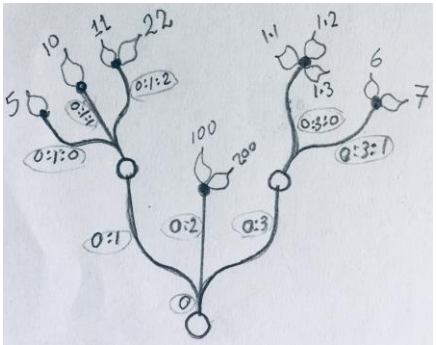


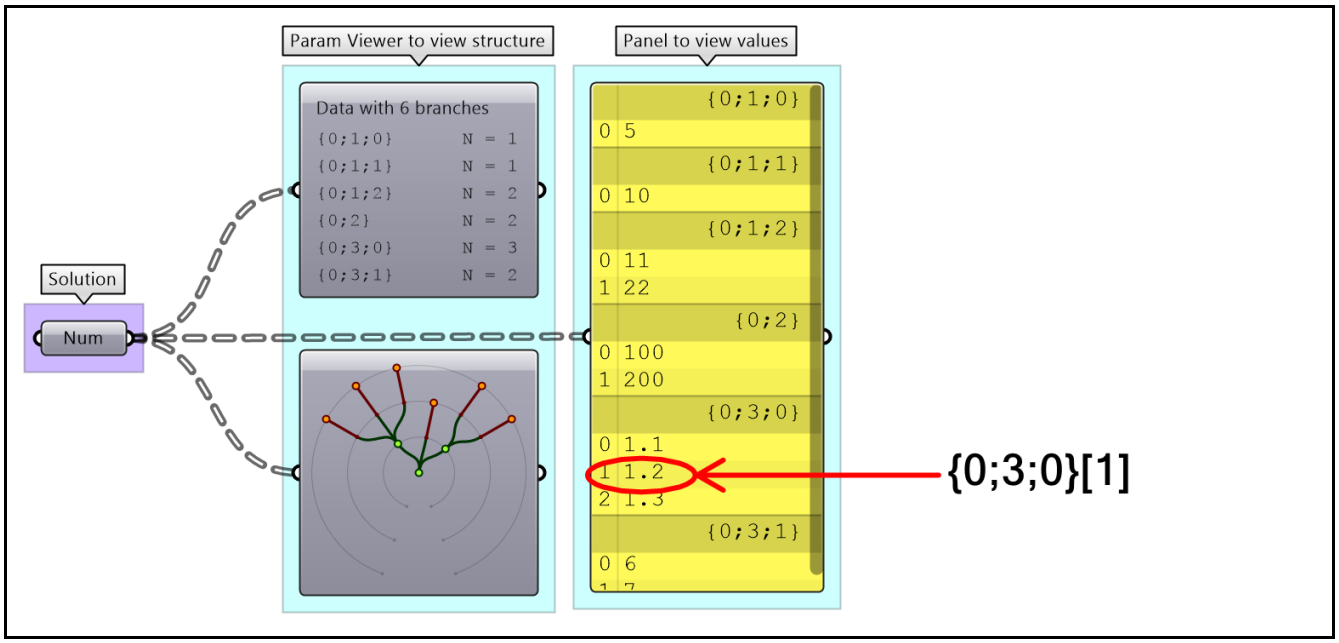
図 (57): 異なる構造で保持されている同じ値のセット．左：5 本の幹（5 本の木）にそれぞれ 1 つのアイテム．真ん中：1 本の幹（1 本の木）に 5 本の枝があり，各枝は 1 つのアイテムを保持．右：2 本の幹（2 本の木）があります．最初の幹には 2 本の枝があり，最初の枝はさらに 3 本の子枝に分岐し各小枝が 1 つのアイテムを保持，2 番目の枝は 1 つのアイテムを保持． 2 番目の幹は 2 つのアイテムを保持.

3_1_1 データツリーのチュートリアル:

Number パラメータを用いて，***Param Viewer*** で画像のようになる図示した数値を持つツリーを作成します．次に，***Panel*** 上にアイテム「1.2」への完全なアドレスを注記します．枝と葉の順序は常に左から右へ時計回りになることに注意です.



Solution	
	「**1.2**」のパスは次の通り． **{ 0 ; 3 ; 0 } [1]** Note: メインの幹から出る 3 つのブランチは，0:1, 0:2, 0:3 となっていますが，0:0, 0:1, 0:2 となる場合もあり，いずれも正しいです.



3_2: ツリー生成

データツリーを生成する方法はたくさんあります．直接的に作成もできますが，ほとんどはいくつかのプロセスの結果としてできるものです．これが出力を使用する前に出力のデータ構造を常に確認する必要がある理由です．GH パラメータ内でデータを入力し，直接データ構造を設定することが可能です．一度設定すると，変更が比較的難しいため，入力を変更しない場合に最適です．以下は，パラメータ内に直接データツリーを設定する方法の例です．

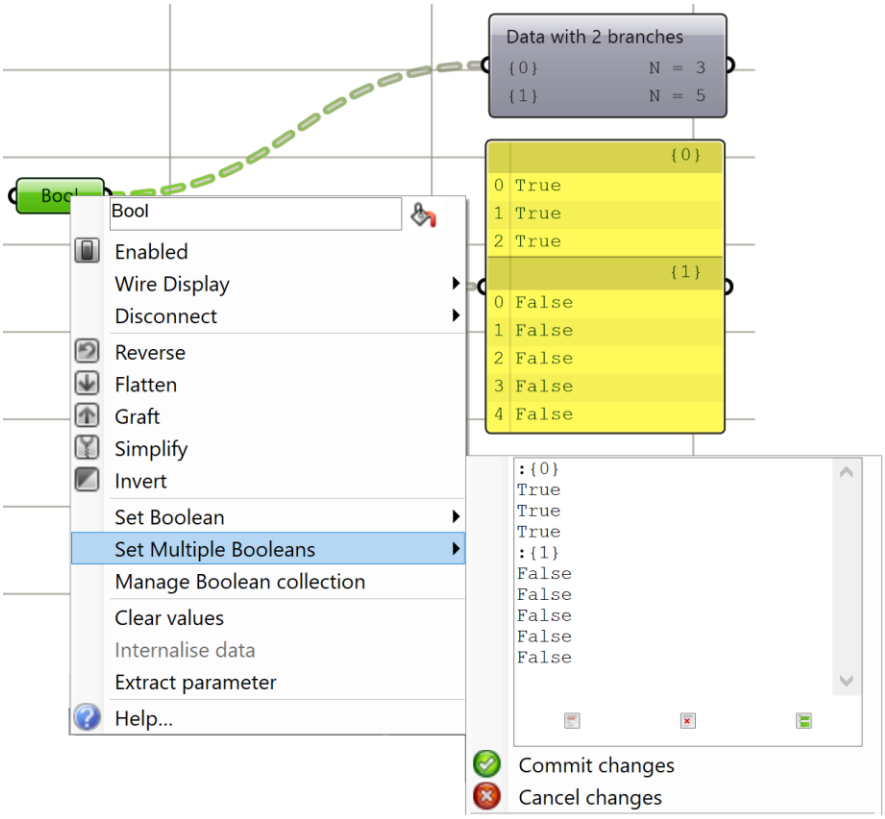
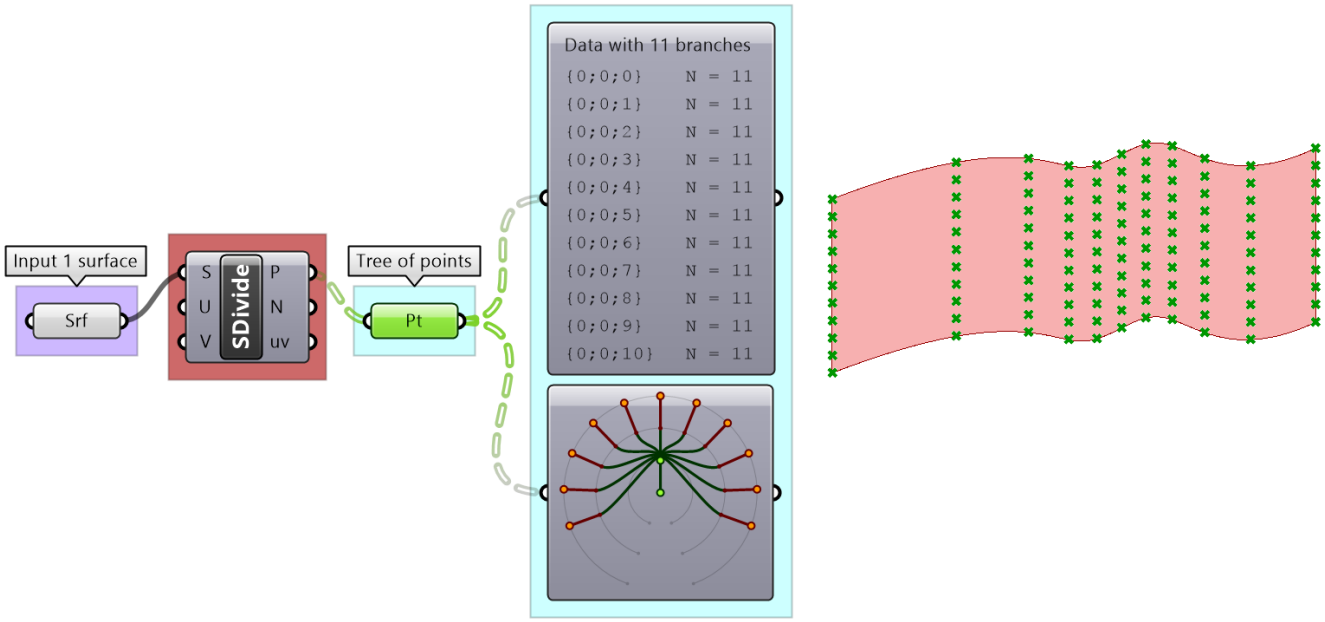
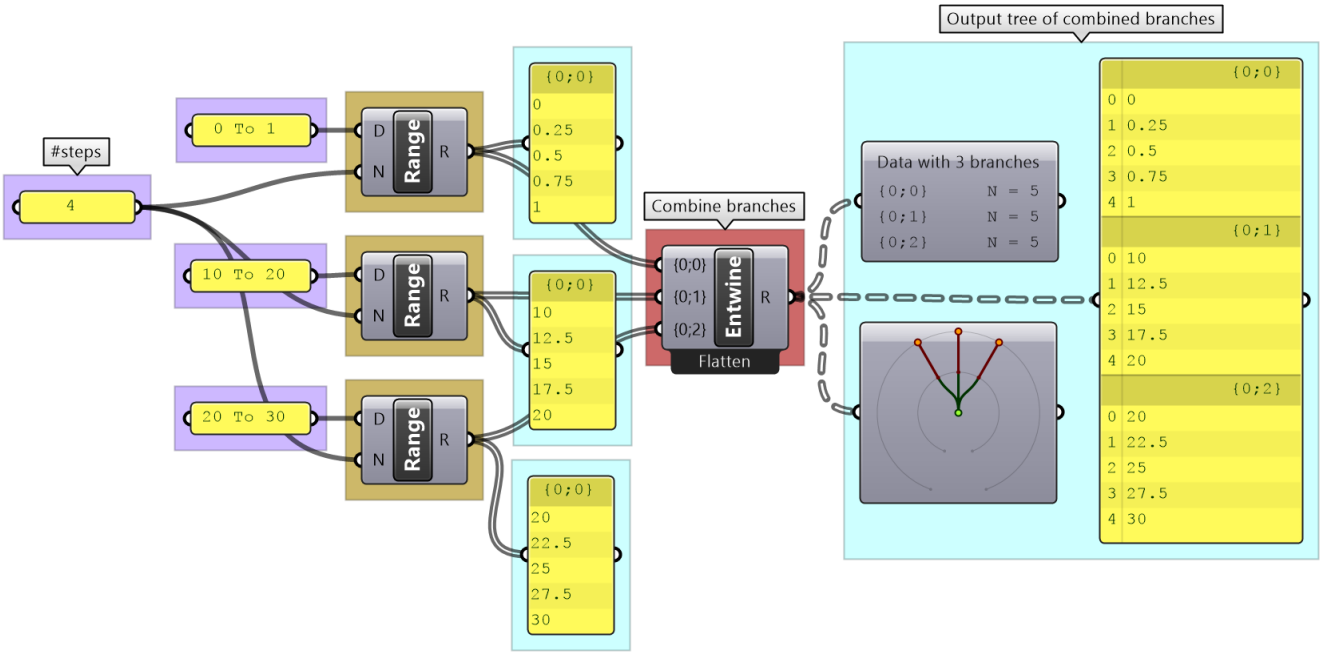


図 (58): データツリーを直接パラメータ内に設定する方法.

Grid や ***DivideSrf*** のようなデータツリーを生成するコンポーネントや，***Entwine*** のようなリストを組み合わせてツリー構造にするコンポーネントなどいろいろなものがあります．また，リストを生成するコンポーネントはすべて，入力がリストの場合、ツリーが作成されます． 例えば，***DivideCrv*** に複数のカーブを入力すると、点のツリーが得られます．数値のリストを生成するすべてのコンポーネント（***Range*** や ***Series*** など）も，リストを入力するとツリーを生成します．

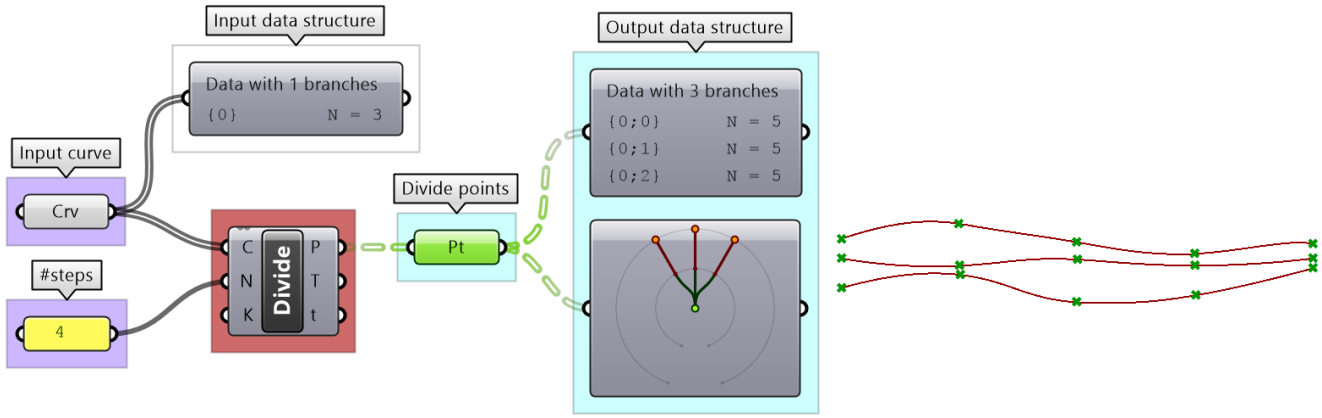


図(59): ***SDivide*** は 1 つの入力（サーフェス）で 1 つのデータツリー（点グリッド）を出力します．



図(60): ***Entwine*** コンポーネントは， 複数のリストを結合し，ツリー構造にします．

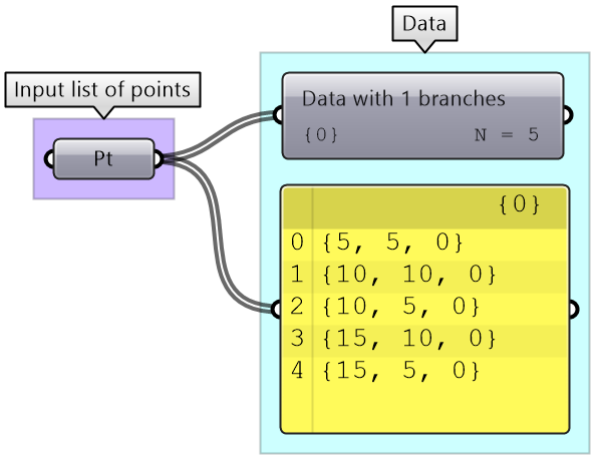
ツリーを生成する最も一般的なケースの 1 つは，カーブのリストを分割して点のグリッドを生成する場合でしょう．このとき，入力は複数のカーブのリストで，出力はツリーとなります．



図(61): ***Divide*** コンポーネントは，リスト（カーブ）を分割し，ツリー構造（点グリッド）を生成します.

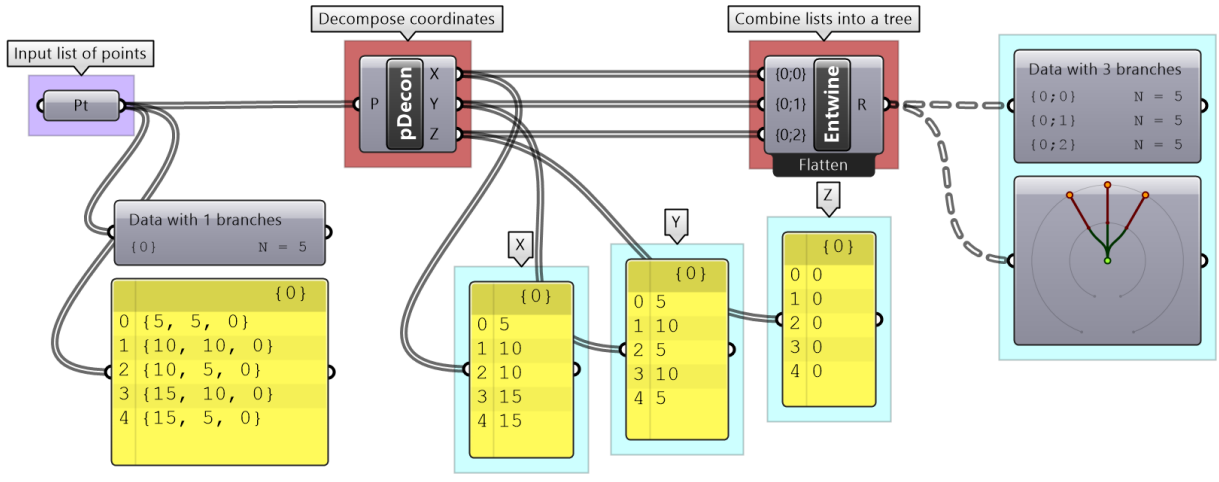
3-2-1 ツリー生成のチュートリアル

次のような点のリストを仮定し，それぞれの座標成分を格納する 3 つのブランチを持つ数値ツリーを作成します.



Solution

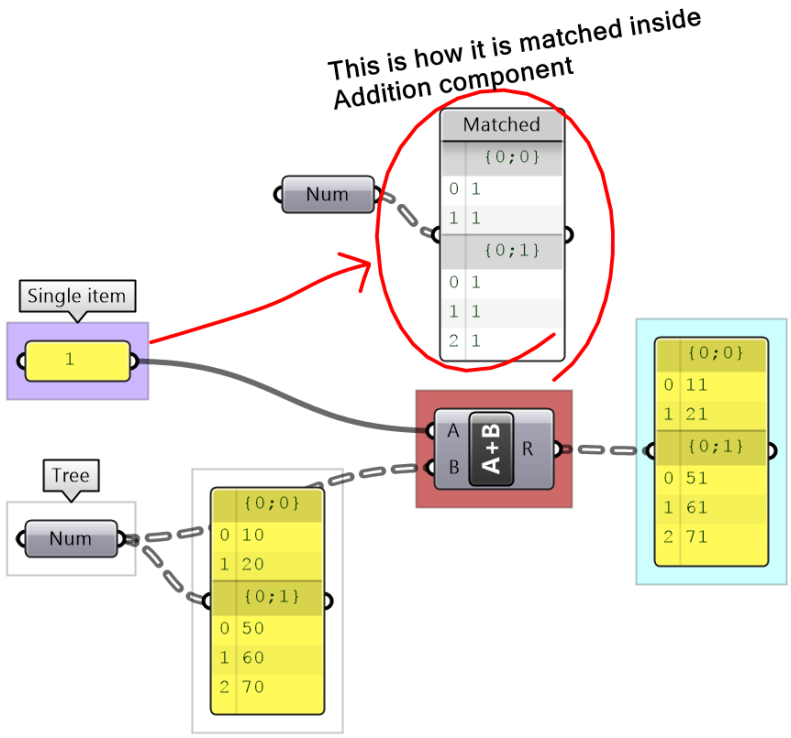
Discussion: 入力する各点は，3 つの数値（座標成分）で 1 つのアイテムとなるデータです．各座標成分を個別のリストに分離し，それらを 1 つのデータ構造に結合すれば良いです．したがって，まず入力点を分解（***pDecon*** コンポーネントを使用）してから，リストを 1 つの構造に結合（***Entwine*** コンポーネントを使用）します.



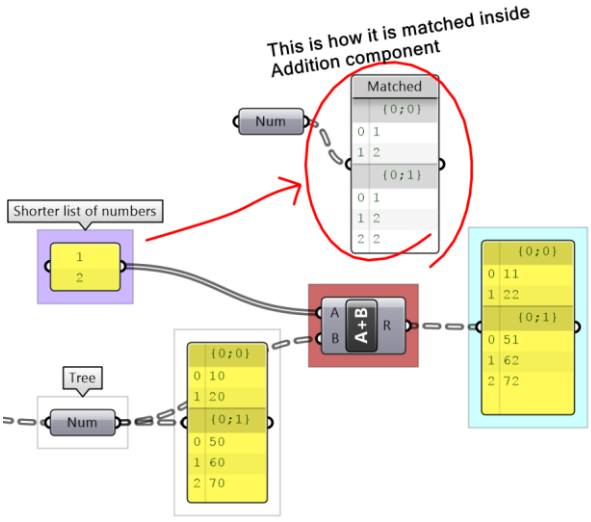
3_3: ツリーマッチング

リストのマッチング方法として，***Long***，***Short***，***Cross Reference*** について説明しました．ツリーでも同様の規則に従って，短いブランチの最後のアイテムを再利用して長さが一致するように展開します．また，1 つのツリーのブランチが少ない場合は，最後のブランチが繰り返し利用されます．以下に一般的なツリーマッチングの例を示します．

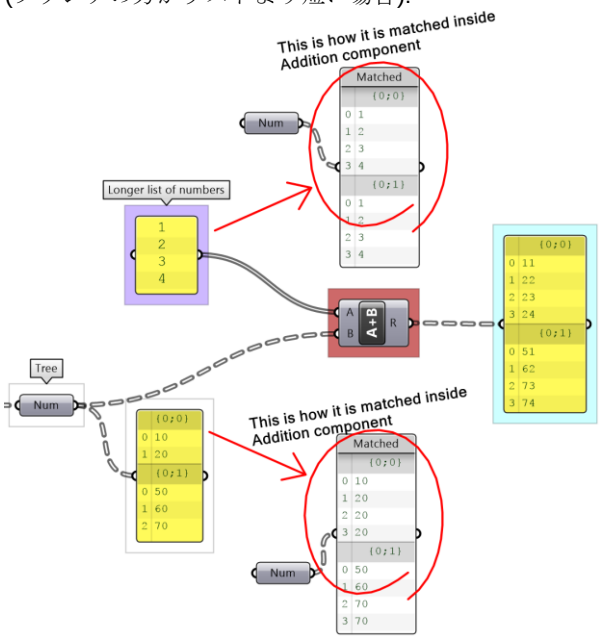
ツリーと 1 つのアイテムのマッチング:



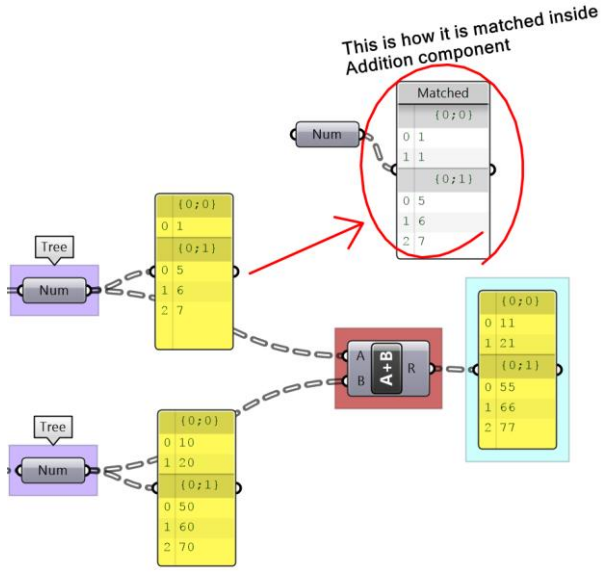
ツリーと短いリストのマッチング
(ブランチの方がリストより長い場合):



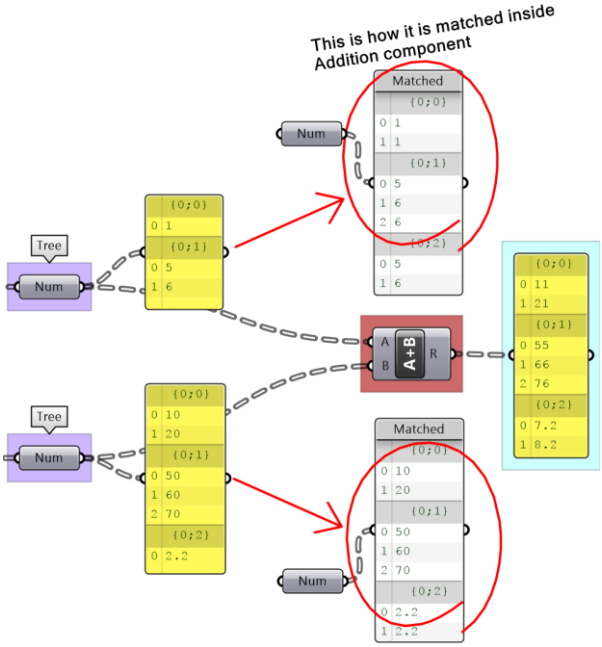
ツリーと長いリストのマッチング
(ブランチの方がリストより短い場合):



同じ数のブランチを持つ 2 つのツリーのマッチング:

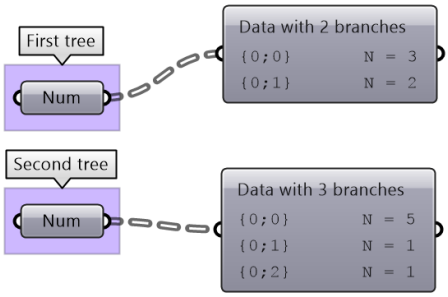


違う数のブランチを持つ 2 つのツリーのマッチング:



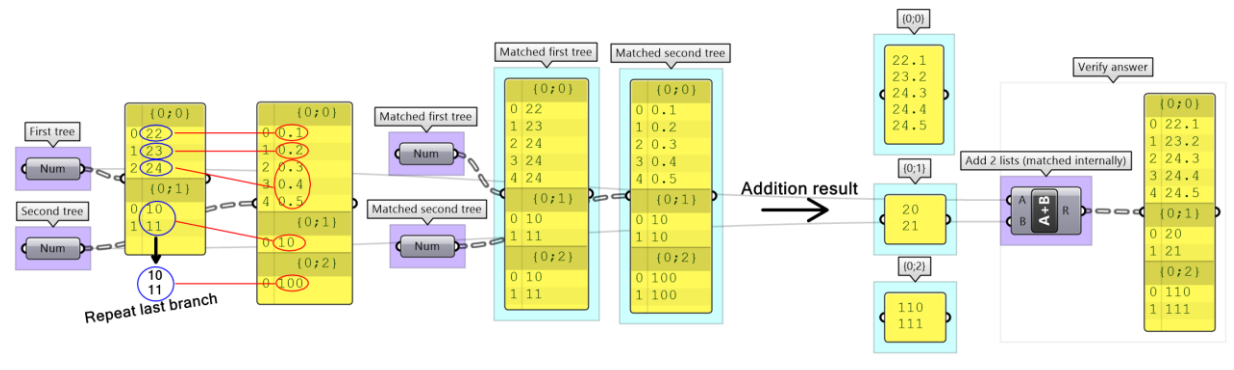
3_3_1 ツリーマッチングのチュートリアル

次の 2 つの Number の構造を確認し，それらを加算した結果と構造を予測してみましょう（デフォルトの GH のマッチングを使用）． ***Addition*** コンポーネントを使用して答えを確認します.



Solution

Key solution idea: 2 つの入力ツリーには，異なる数のブランチから成り，それぞれのブランチは異なる数のアイテムを持ちます．ブランチが少ないツリーの最後のブランチは，長いツリーのブランチの数と一致するまで再利用され，対応するブランチ同士は，短いブランチの最後のアイテムを再利用することでマッチングします.



3_4: ツリーからデータを抽出

GH には，ツリーからブランチやアイテムを抽出するのに役立つコンポーネントがあります．ブランチまたはアイテムへのパスがわかる場合は，***Branch*** と ***Item*** コンポーネントを使用できます．正しいパスを入力できるように，入力の構造を確認する必要があります．

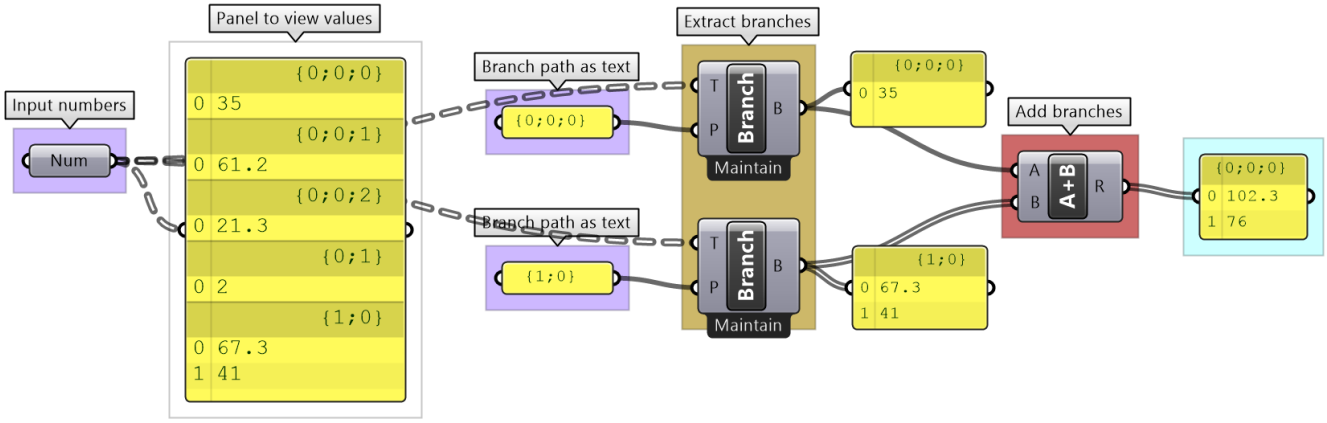


図 (62): ツリーからブランチを抽出.

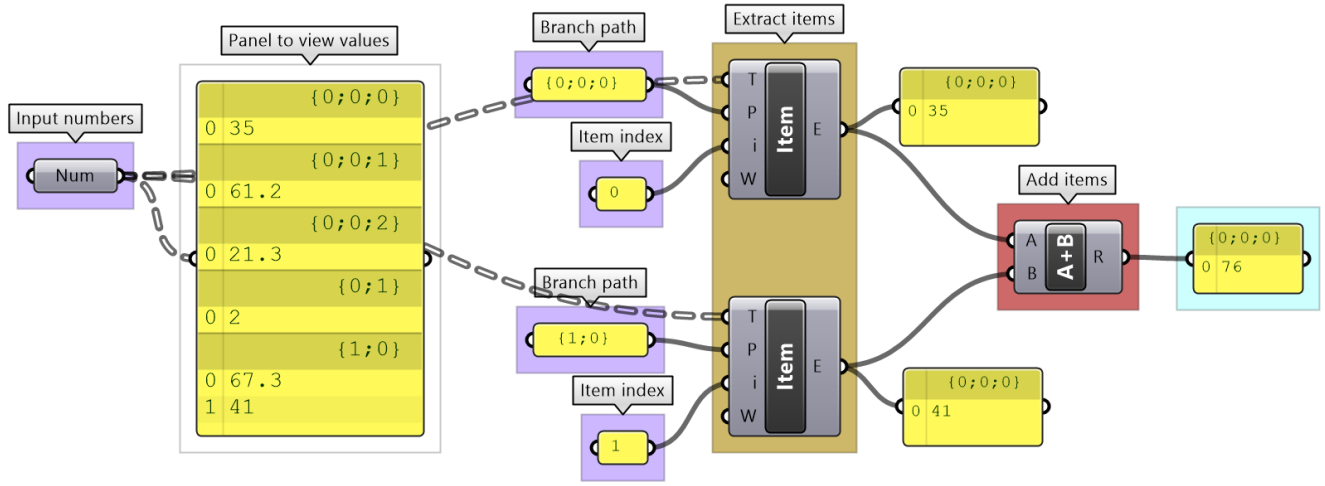


図 (63): ツリーからアイテムを抽出.

構造が変化する可能性があることがわかっている場合，または直接パスを入力したくない場合は，***Param Viewer*** および ***List Item*** コンポーネントを使用して動的にパスを抽出できます．

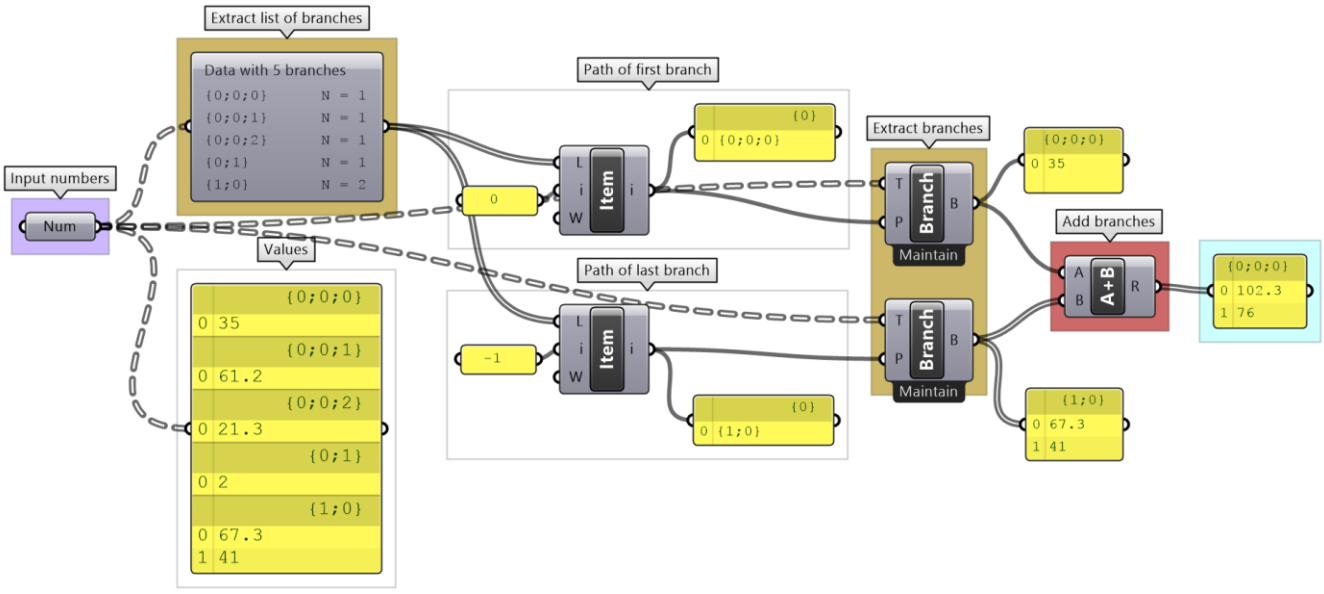
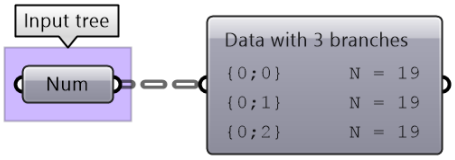


図 (64): データのパスを動的に抽出する方法の例.

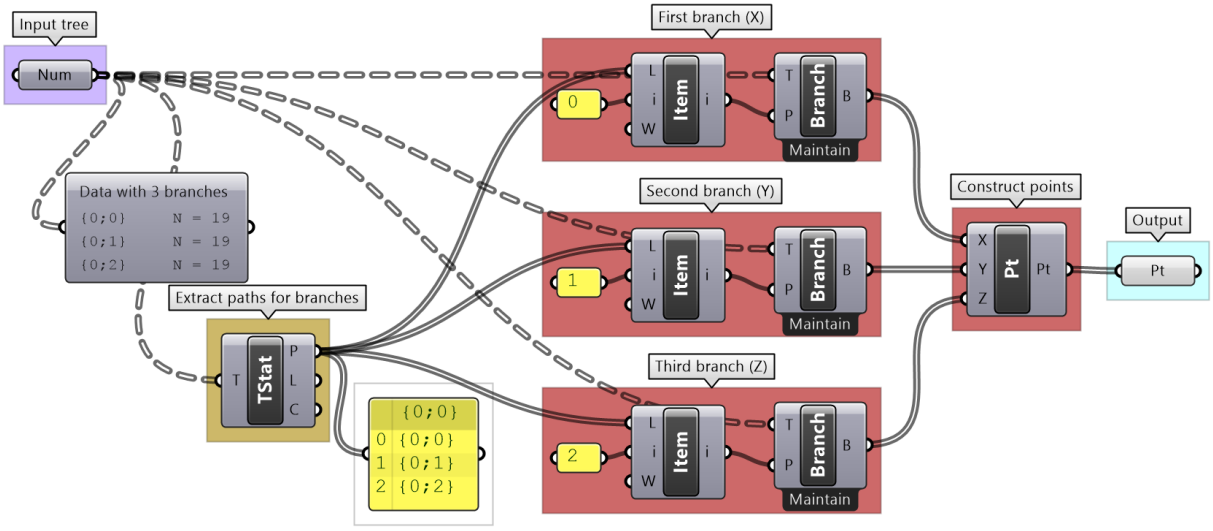
3_4_1 ツリーからデータを抽出のチュートリアル

次のツリーには，点リストの各座標成分（x、y、z）を表す 3 つのブランチがあります．そのツリーを使ってこれらの点リストを作成してみましょう．



Solution

Key solution idea: X，Y，Z 成分を表す 3 つのリストを入力として使用して，点のリストを作成します．入力ツリーの 3 つのブランチを分離できたら，それらを ***Construct Point*** コンポーネントにつなぎます．



3_5: 基本的なツリー処理

基本的なツリー処理は，いろいろな場面で用いられ，ほとんどのアルゴリズムで必要になるかもしれません．これらの処理のされ方，そして出力への影響を理解することはとても重要です．

3_5_1: ツリー構造の可視化

データマッチングで見てきたように，要素のセットが同じでデータ構造が異なる場合は，異なる結果を生成します．GH には，データ構造を確認する方法が 3 つあります．テキストまたはダイアグラム形式の ***Parameter Viewer*** と ***Panel*** です．

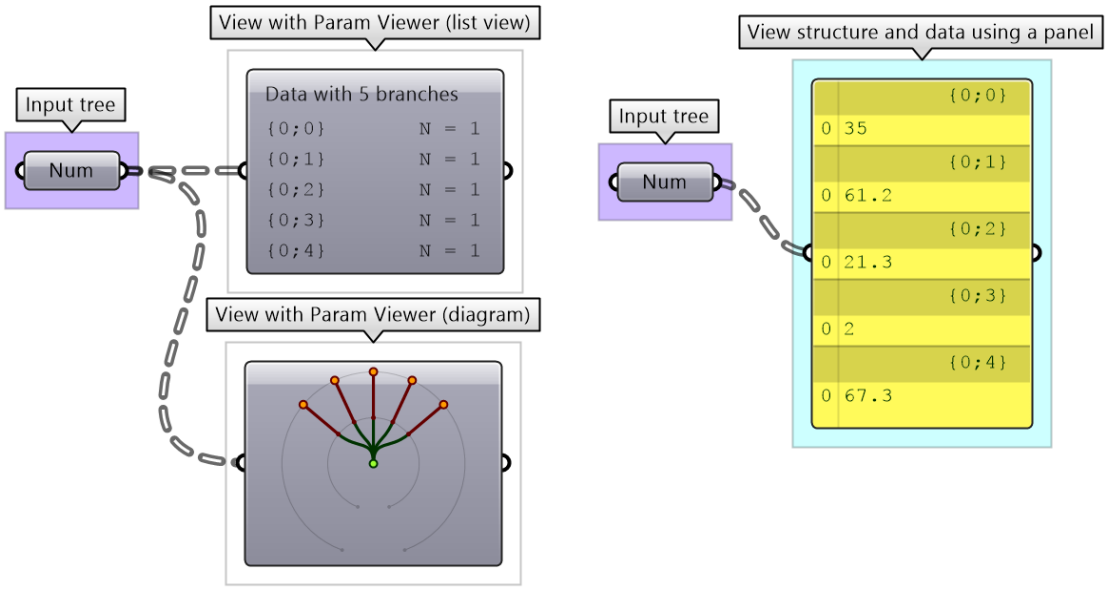


図 (65): ***Parameter Viewer*** や ***Panel*** コンポーネントを用いて，ツリーを可視化．

ツリー情報は，***TStats*** コンポーネントを使用して抽出できます．すべてのブランチへのパスのリスト，各ブランチのアイテムの数，ブランチの数を抽出できます．

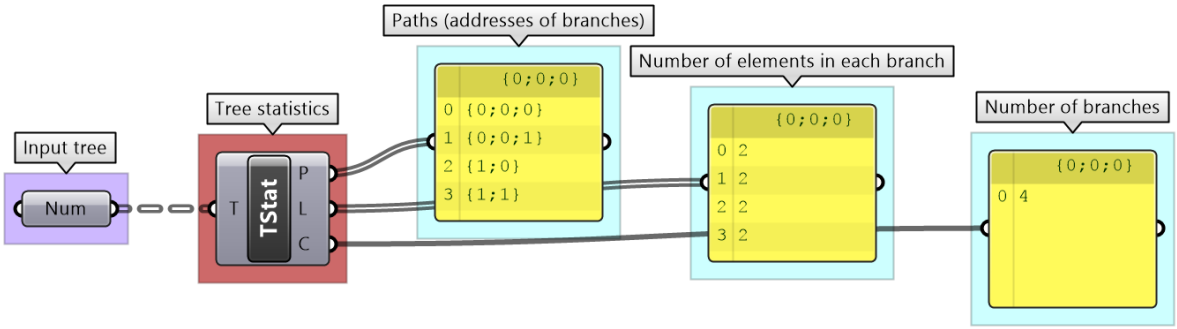
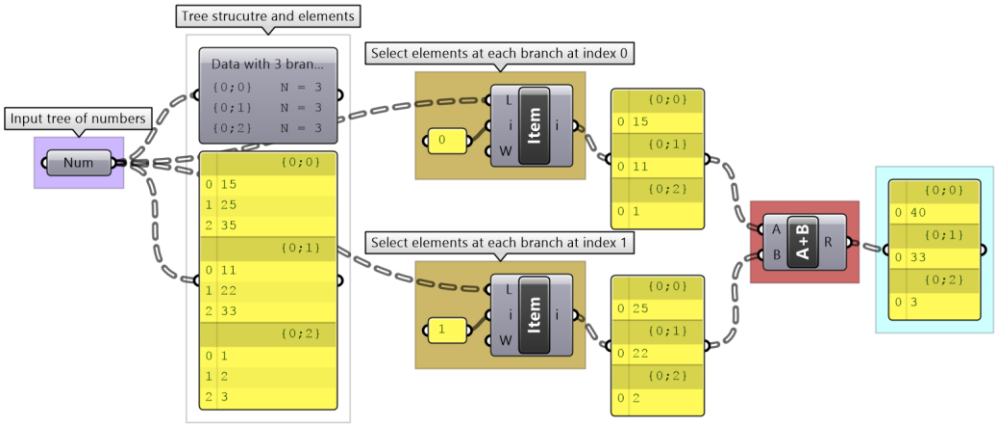
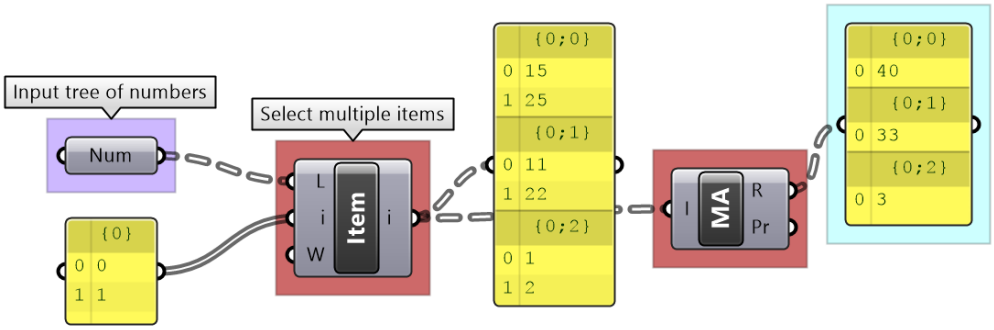
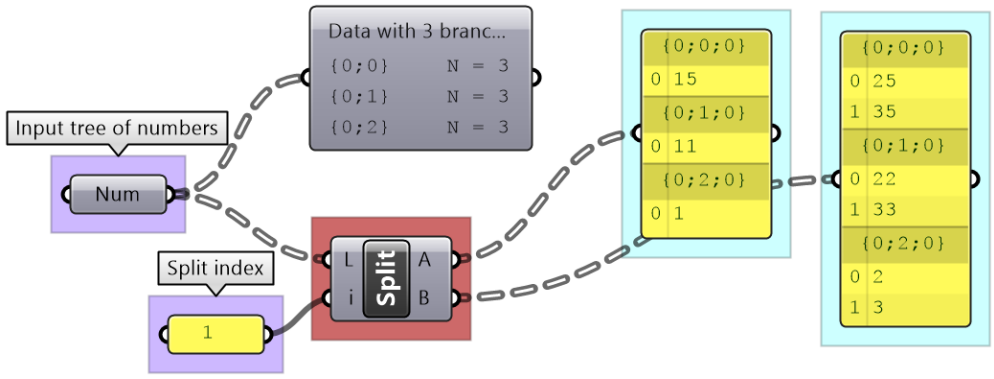
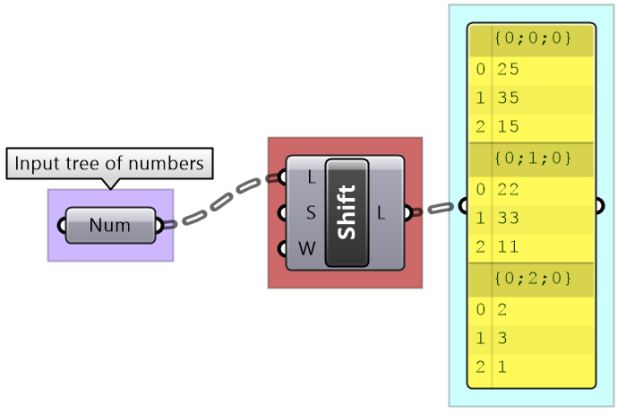
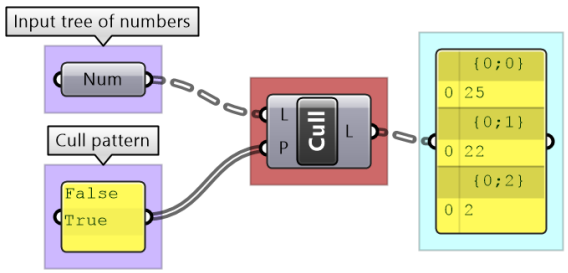


図 (66): ***TStats*** コンポーネントを使ってツリー構造の情報を抽出．

3_5_2: ツリー構造におけるリスト処理

ツリーは，ブランチのリストであると考えることができます．ツリーでリスト処理を行う場合，各ブランチは個別のリストとして扱われ，処理は各ブランチに個別に適用されます． 結果のデータ構造を予測することは難しいので，処理を適用する前後に，入出力の構造を確認することは常に大事になります．ツリーでリスト処理がどう機能するかを確認するために，単純なツリーをさまざまなリスト処理に適用し，出力とそのデータ構造を調べます。

処理	ツリーに適用するリスト処理の例
List Item それぞれのブランチの特定のインデックスのアイテムを抽出.	
List Item 複数インデックスを指定してツリーの一部を抽出し，***Mass Addition*** などでまとめて実処理を実行.	
Split List 特定のインデックスでブランチを分割し，2 つのツリーを生成.	
Shift List それぞれのブランチのアイテムをシフト.	

Cull Pattern それぞれのブランチのアイテムを取捨選択.	

3_5_3: Graft でリストをツリー化

場合によっては，リストをツリーにして，各アイテムを個別のブランチに配置する必要があります．Graft（接ぎ木）処理は，異なる深さのブランチを持つ複雑なツリーを処理できます．

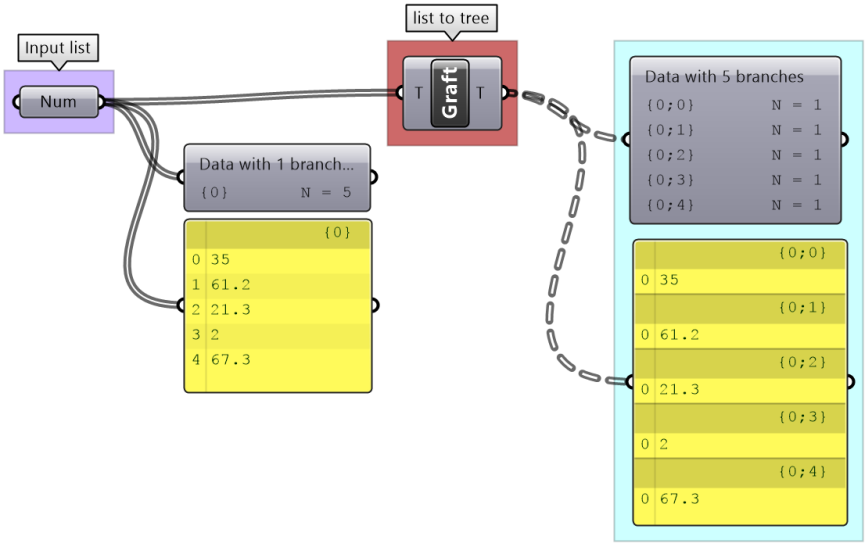


図 (67): ツリーを Graft 処理すると各アイテムの入った新しいブランチが生成されます.

単純なリストをツリーにするなど，データ構造を複雑にすることは直感的にはわかりづらいかもしれませんが、特定のマッチングを実現しようとする場合，Graft は非常に便利です．例えば，1 つのリストのそれぞれのアイテムを 2 つ目のリストのすべてのアイテムと加算したい場合，加算プロセスに入力する前に，最初のリストを Graft する必要があります．

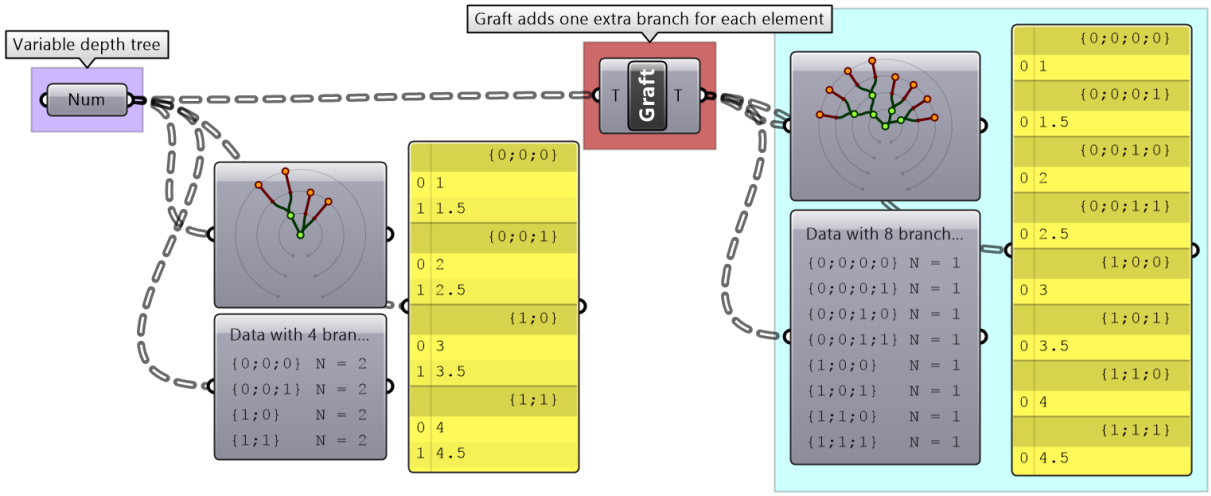


図 (68): 複雑なツリーの Graft.

3_5_4: Flatten でツリーをリスト化

また，ツリー構造を単純なリストに変換する必要がある場合もあります．これは，Flatten（平坦化）処理で実現できます．各ブランチのデータが抽出され，1 つのリストに順番にまとめられます．

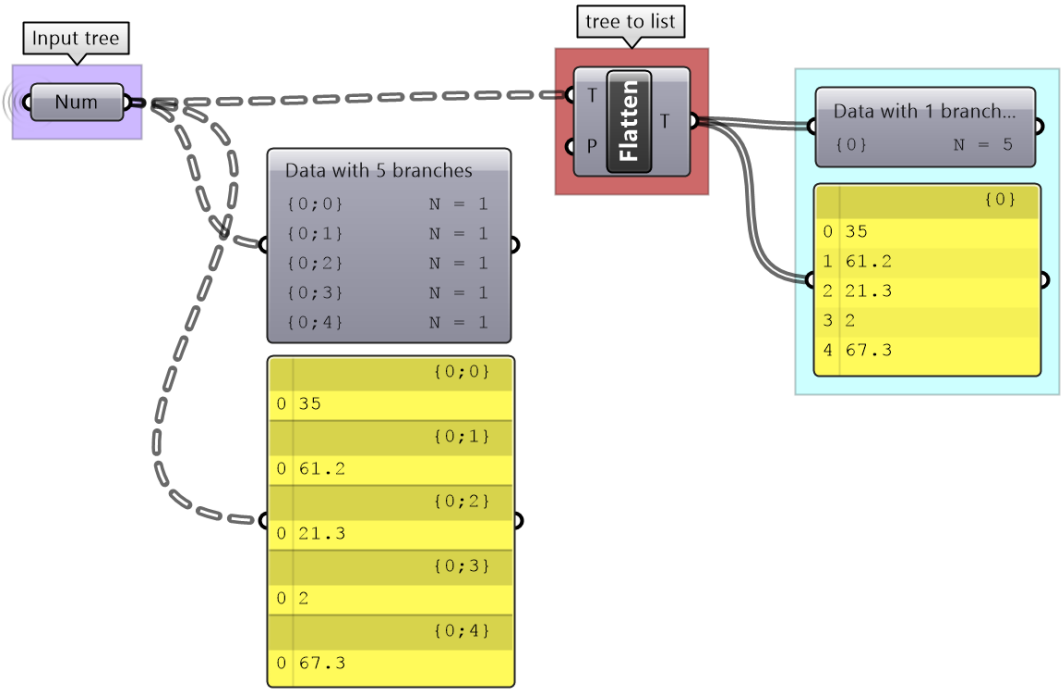


図 (69): Flatten 処理では，ツリーのすべてのアイテムが 1 つのリストに平坦化されます．

Flatten は，複雑なツリーも処理できます．最小のパス番号から順にブランチを取得し，すべてのアイテムを 1 つのリストに配置します．

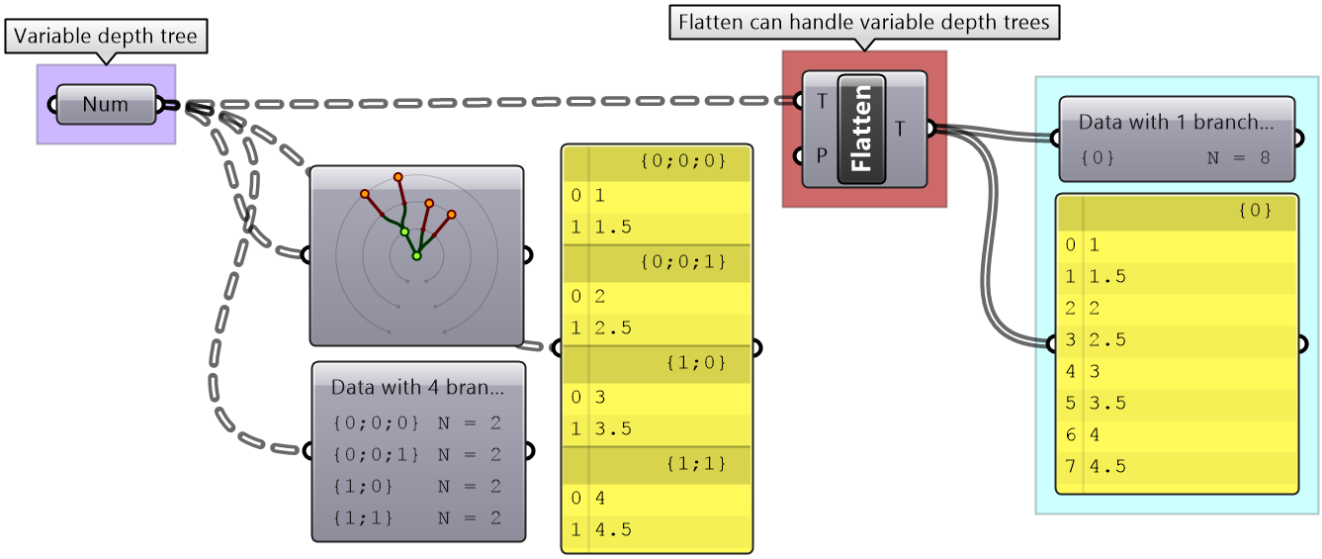


図 (70): 複雑なツリーの Flatten 処理．

3_5_5: データの結合

各リストが新しいツリーのブランチになるように複数のリストを結合することができます．これは，単に 1 つの大きなリストが作成されるリストの ***Merge*** とは異なります．

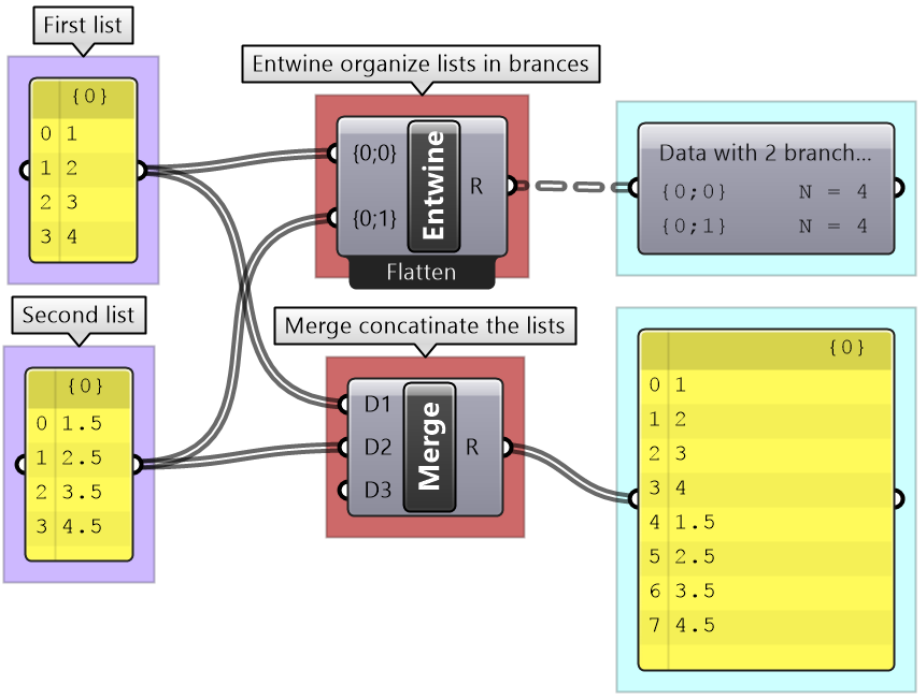


図 (71): ***Entwine*** と ***Merge*** は，リストをツリーもしくはより大きいリストに結合します.

3_5_6: データ構造の Flip

ツリーを反転（Flip）してブランチとアイテムの組み合わせを変更することが合理的な場合があります．これは，点が行と列で編成されているグリッドなどで特に便利です（2 次元配列構造と同様）．Flip すると，ブランチ間で対応するアイテム（ブランチ内で同じインデックスを持つ）が 1 つのブランチとしてグループ化されます．例えば，2 つのブランチと各ブランチに 4 つのアイテムを持つデータツリーは，4 つのブランチと各ブランチに 2 つのアイテムを持つツリーに Flip できます.

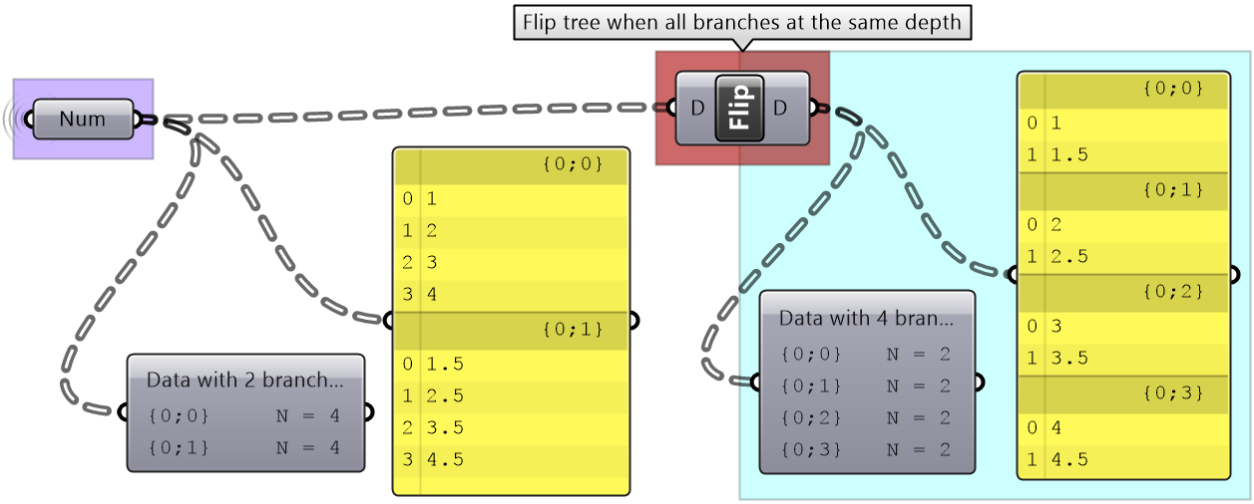


図 (72): ***Flip Matrix*** でデータツリーの構造を反転（行と列を入れ替え）します.

ブランチ内のアイテム数がそれぞれ異なる場合，Flip したツリーの一部のブランチには「null」値が入ります.

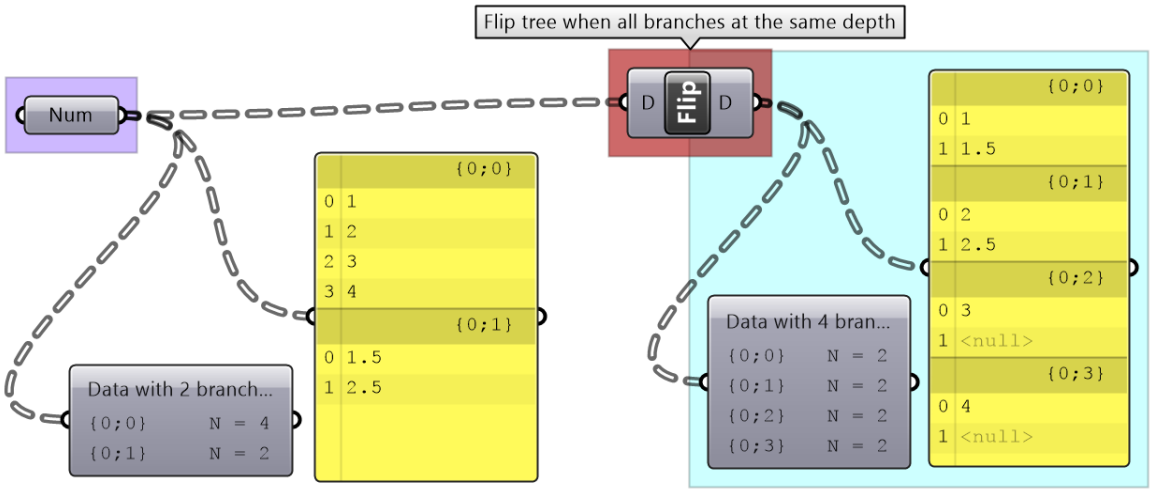


図 (73): 異なる長さのブランチを持つツリーを Flip すると「null」が追加されます.

Flip Matrix では，異なる深さのブランチを含むものは扱えません（単純に解が存在しないからです）．

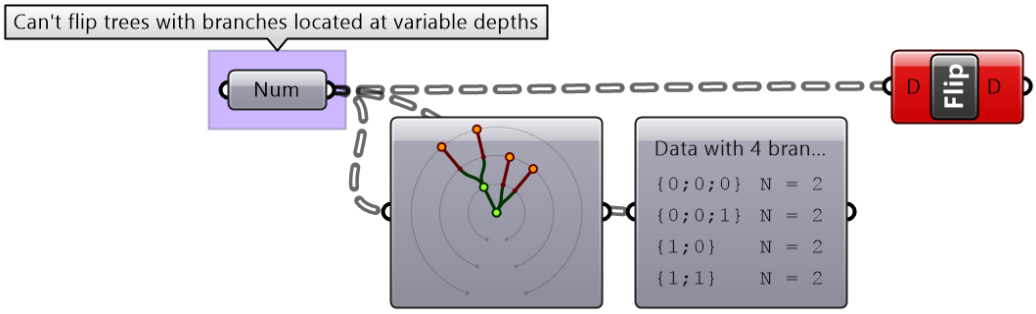


図 (74): 入力ツリーに異なる深さのブランチが含まれる場合，***Flip Matrix*** はエラーになります.

3_5_7: Simplify でデータ構造を簡素化

複数のコンポーネントを介してデータを処理すると，データ構造が不必要に複雑になる可能性があります．最もよくあるのは，パスアドレスの前か後ろにゼロが追加される形式です．複雑なデータ構造を一致させるのは大変です．***Simplify Tree*** の処理は，空のブランチを削除するために使用します．null 要素や空のブランチを削除し，複雑さを軽減するために，***Clean Tree*** や ***Trim Tree*** などいろいろなコンポーネントがあります．***Explode Tree*** を使用すると，すべてのブランチを個別のリストとして抽出することもできます.

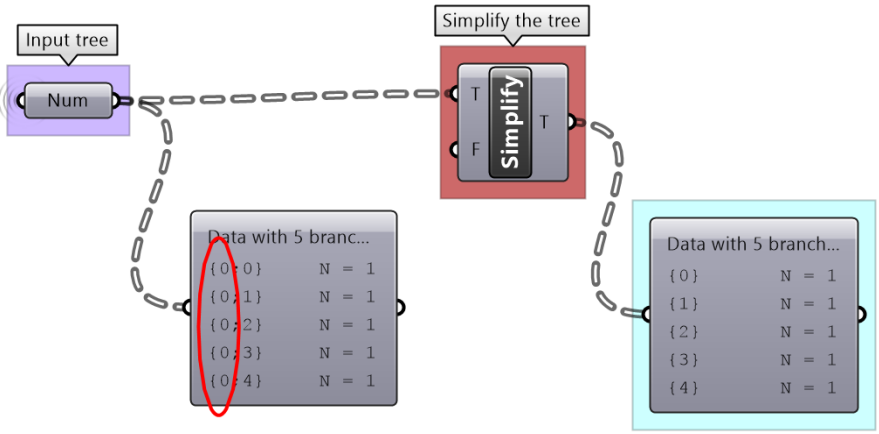
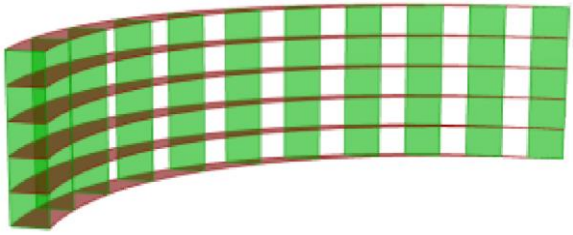
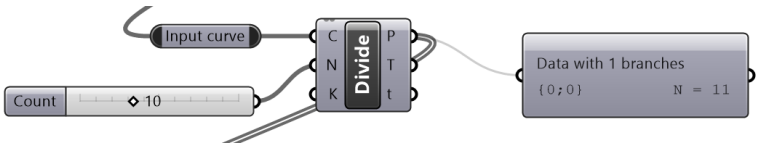
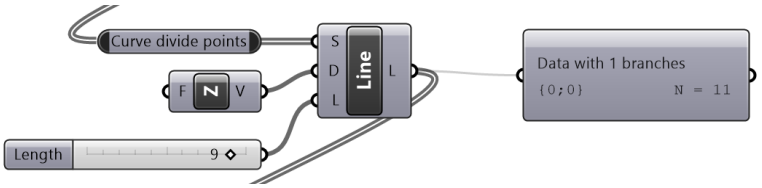
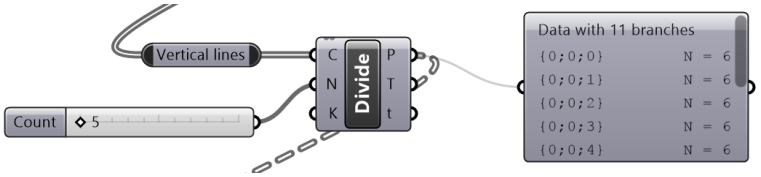
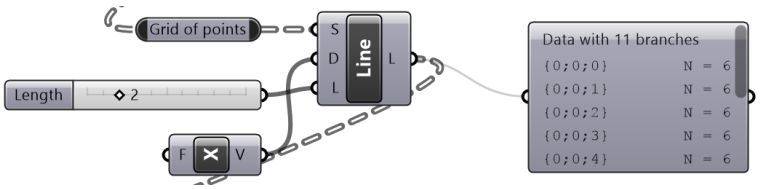
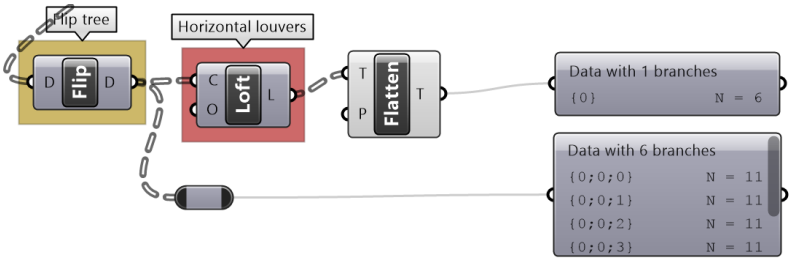


図 (75): データに適用する処理が増えるほど，パスの複雑さが増します．***Simplify*** により空のブランチを削除できます.

3_5_8: 基本的なツリー処理のチュートリアル #1

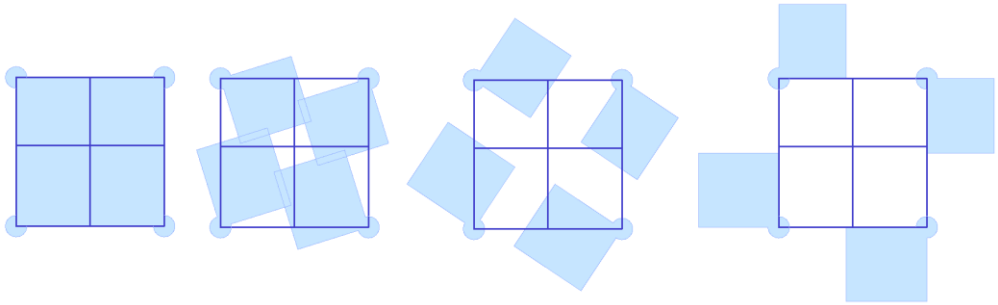
XY 平面上の 1 つの曲線を入力して，図のように水平ルーバーと垂直ルーバーを作成してみましょう．

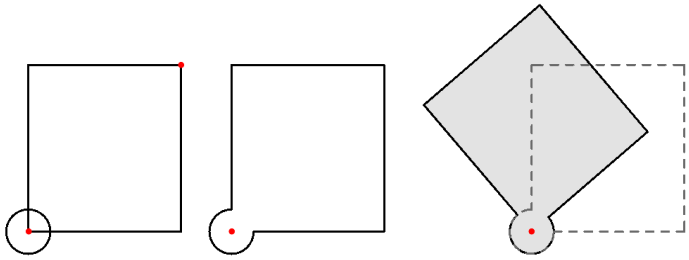
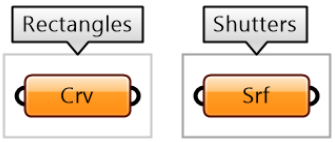
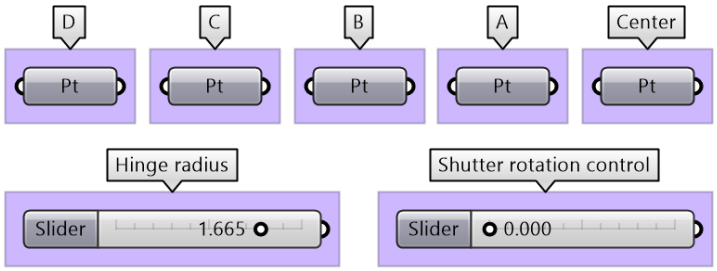
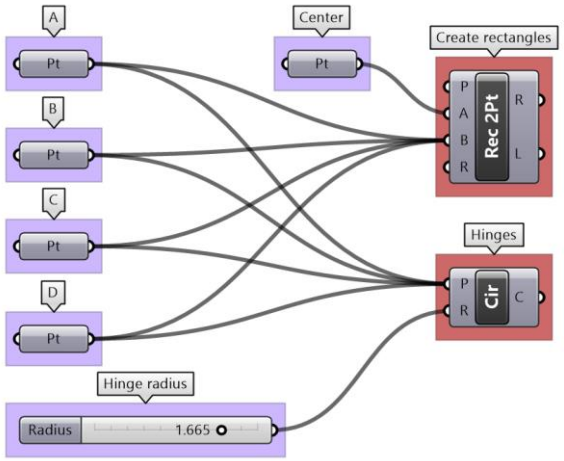


Solution	
入力カーブ データ構造: 1 つのアイテム（1 つのアイテムを持つ 1 本のブランチ）．	
カーブを分割して**点**を出力 データ構造: リスト（11 アイテムを持つ 1 本のブランチ）．パス先頭に 0 が追加されていることに注意．階層が増えたことを表します．	
Line SDL で各点の垂直線を作成 データ構造: リスト（11 アイテムを持つ 1 本のブランチ）．ここでは，パスの複雑さは変わっていません．	
垂直線を分割し，**格子状の点**を作成 データ構造: ツリー（各 6 つのアイテムを持つ 11 本のブランチ）．パス先頭に 0 が追加．	
Line SDL で各点に水平線を作成 データ構造:ツリー（各 6 つのアイテムを持つ 11 本のブランチ）．ここでは，パスの複雑さは変わっていません．	
ブランチの Line からロフト**サーフェス**生成 データ構造:ツリー（各 1 つのアイテムを持つ 11 本のブランチ）．パスの複雑さは変わりません．	
ツリーの行列を反転し，ブランチの Line からロフト**サーフェス**生成 データ構造:ツリー（各 1 つのアイテムを持つ 11 本のブランチ）．パスの複雑さは変わりません． **Flatten** でツリーをリストにすることができます．	

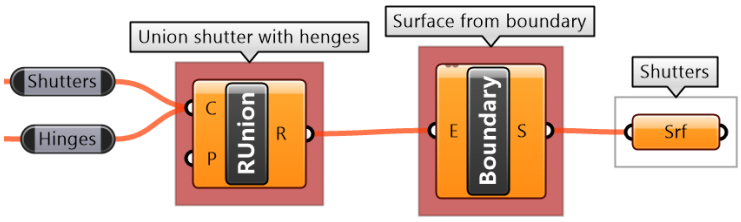
3_5_9: 基本的なツリー処理のチュートリアル #2

同一平面の 4 点のコーナーとヒンジ半径を入力し，回転パラメータにより，図のように開閉するシャッターを作成します.



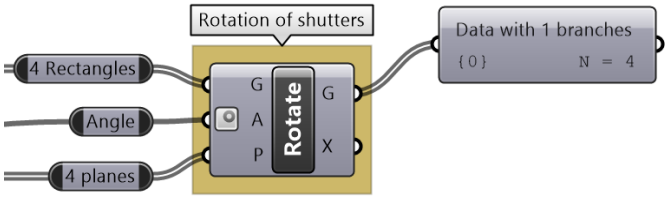
アルゴリズムの分析	
各シャッターは，長方形とヒンジの 2 つのパーツから成ります. 長方形とヒンジを結合し，ヒンジを中心に回転できるようにします. すべてのシャッターは，1 つの回転制御で同時に動かします.	
Grasshopper の実装	
Output ・シャッターのサーフェス ・フレームのカーブ	
Input ・4 つのコーナー点および中心点 ・ヒンジの半径 ・回転パラメータ	
Key processes 長方形とヒンジを作成します. ***Rectangle*** と ***Circle*** を使用.	

それらの曲線を ***Runion*** で結合し，それらを境界線として ***Boundary*** でサーフェス化します.



Intermediate process #1

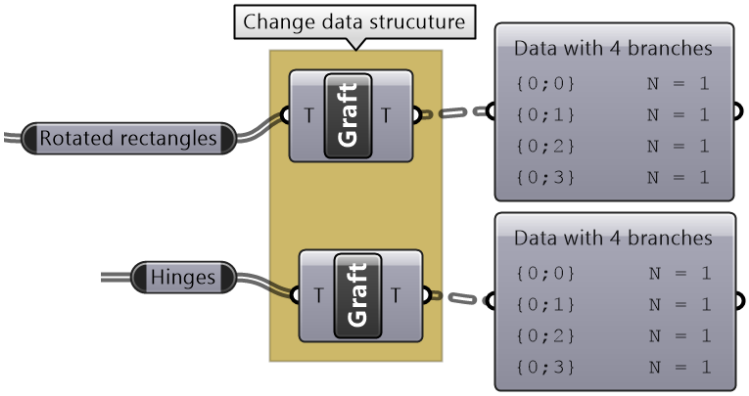
角度を入力して長方形を回転します.
Rotate を使用.



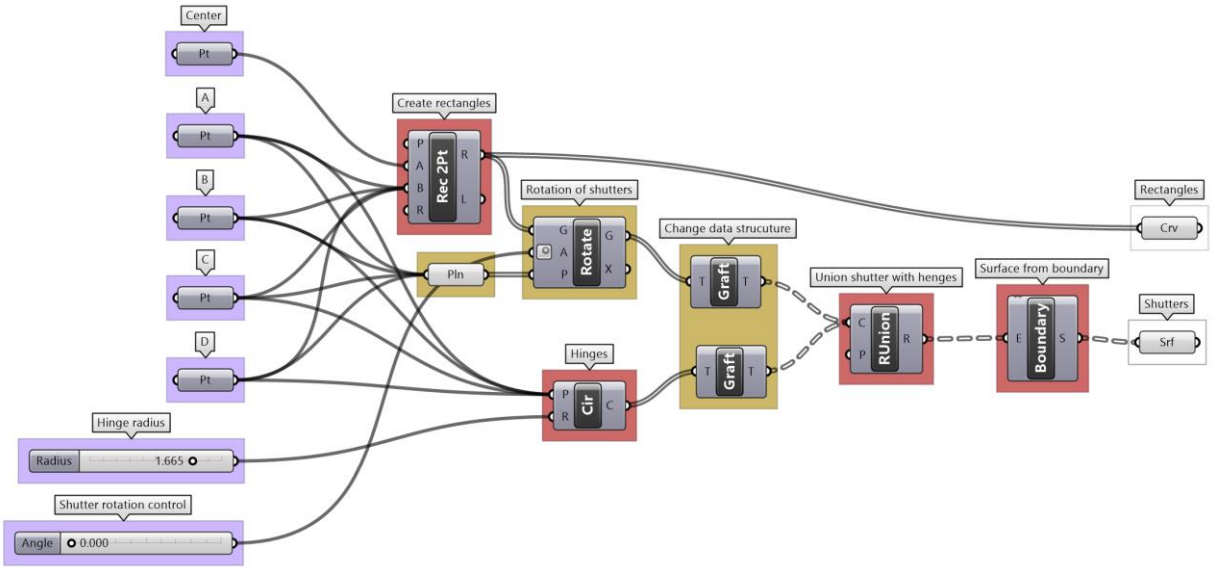
Intermediate process #2

Runion の前に，長方形とヒンジのデータ構造を適切にマッチングさせます.

Graft を使用して，長方形とヒンジが正しくペアになるようにします.



全体像



3_6: 高度なツリー処理

アルゴリズムが複雑になるにつれて，データ構造も複雑になります．特定の問題を解決するために必要となる 3 つの高度なツリー処理について見ていきます．これらは，データ構造を直接操作することでアルゴリズムをシンプルにするために活用されます．

3_6_1: Relative Item で相対アイテムを指定

最初の処理は，1 つのツリー内または複数ツリー間のアイテムの接続に関する一般的な問題の解決に役立つものです．点グリッドがあり，点を斜めに接続する必要があるとします．各点を，+ 1 のブランチの+1 のインデックスの点に接続します．例えば，ブランチ{0}，インデックス[0]の点は，ブランチ{1}，インデックス[1]の点に接続します．

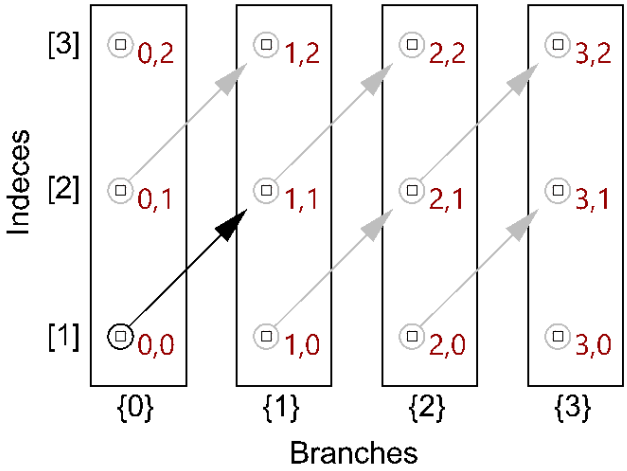


図 (76): + 1 ブランチ，+1 インデックスは，相対的に対角の位置関係にあるアイテムを指します．

GH では要素の相対的な位置関係（オフセット）を，「{ブランチのオフセット} [インデックスのオフセット] 」という形式のオフセット文字列で表現します．この例では，点を斜めに接続するオフセットは，{+1} [+ 1]です．GH で相対的にツリーの要素を使用する例を以下に示します．***Relative item*** コンポーネントは，オフセットで指定された方法に従い，2 つの新しいツリーを生成します．

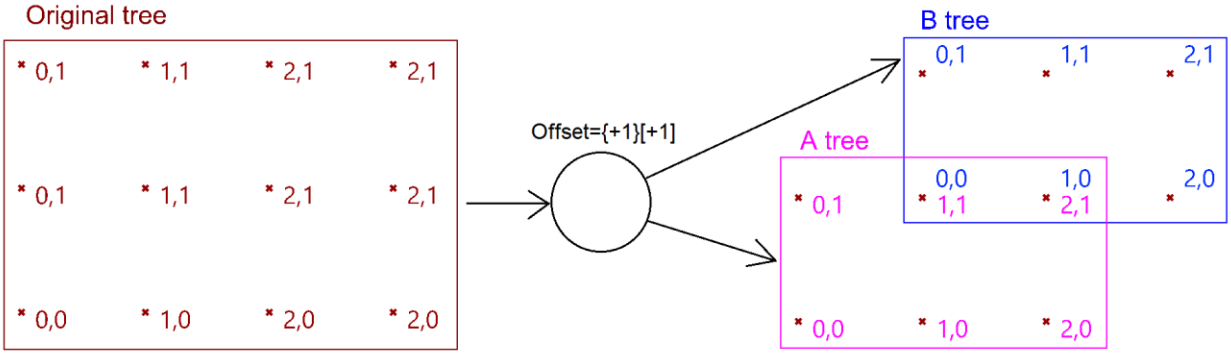


図 (77): オフセット{+1}[+1]の Relative Items は，元のツリーを分解し，対角が接続する 2 つの新しいツリーを生成します．

以下は GH の実装例です．***Relative Item*** コンポーネントを使用して，ツリーの相対アイテムを定義し，結果として得られた 2 つのツリーを直線で接続しています．

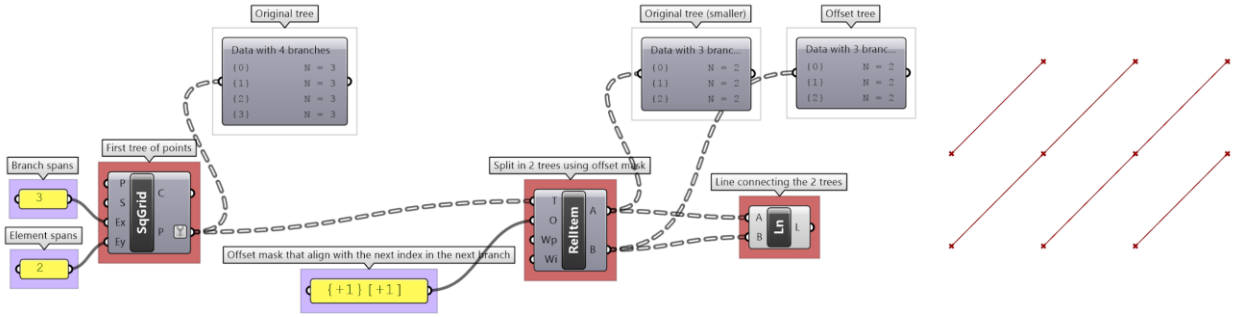
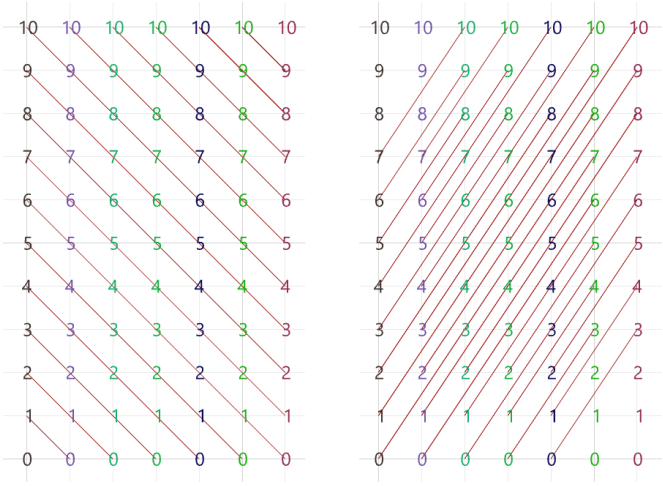
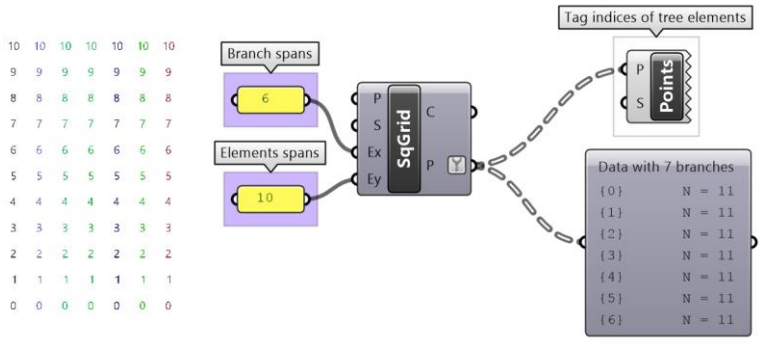
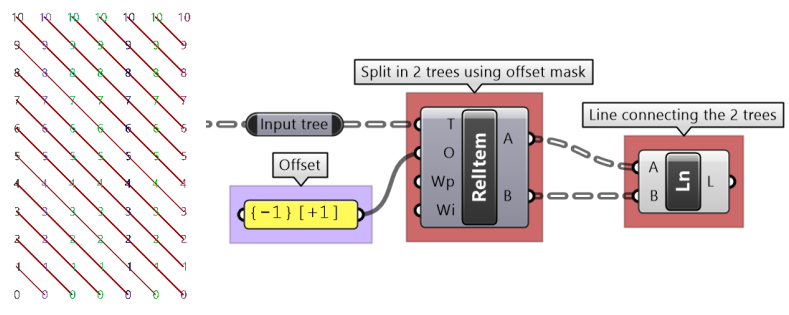


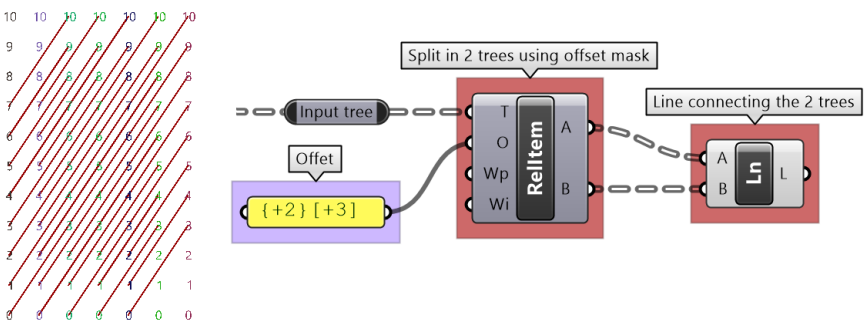
図 (78): GH の ***Relative Item***（オフセット{+1}[+1]の場合）.

3_6_1_1 Relative Item チュートリアル #1

7 つのブランチの正方形グリッドから，図に示すパターンを作成しましょう．各ブランチには 11 のアイテムがあります.



Solution	
{ブランチオフセット} [インデックスオフセット]を定義します	
グリッドを生成します.	
-1 ブランチの+1 インデックスのアイテムと接続する相対ツリーを作成します. 2 つの相対ツリーを接続して直線を作成します.	

オフセットを {+2}[+3] に変更すると，2 つ目のパターンが作成されます.	

1 つのツリーで相対アイテムを定義する方法を示しましたが，2 つのツリー間の相対アイテムを指定することもできます（ただし，2 つの入力ツリーのデータ構造に注意し，それらに互換性があることを確認する必要があります）．例えば，最初のツリーの各点を，同じインデックスでブランチが+1 の別のツリーの別の点に接続する場合，オフセットは{+1} [0]と設定します.

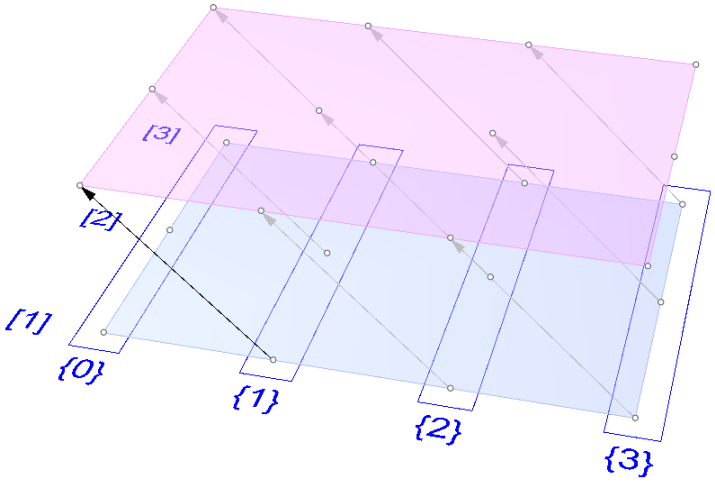


図 (79): ***Relative Items*** は，2 つのツリー間の相対的な接続を作成します.

Relative Items コンポーネントでは 2 つのツリーを入力し，オフセットに従ってアイテムが対応する 2 つのツリーを出力します.

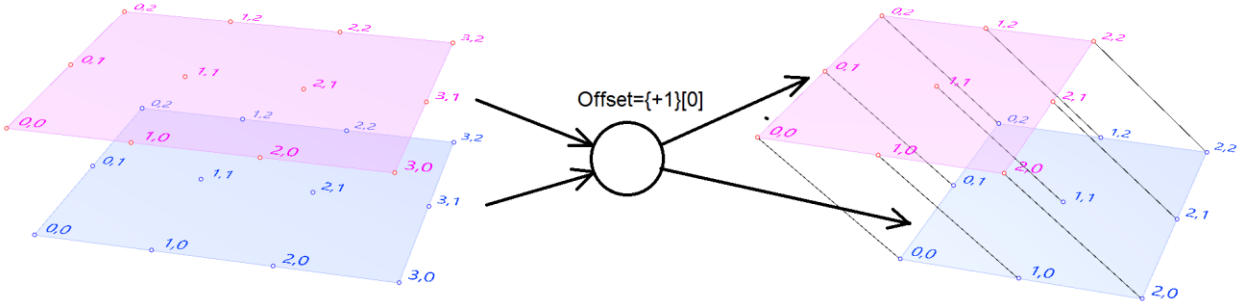


図 (80): ***Relative Items*** は，オフセットで指定した接続を実現する新しいツリーを生成します.

以下の GH 定義では上記を実現しています.

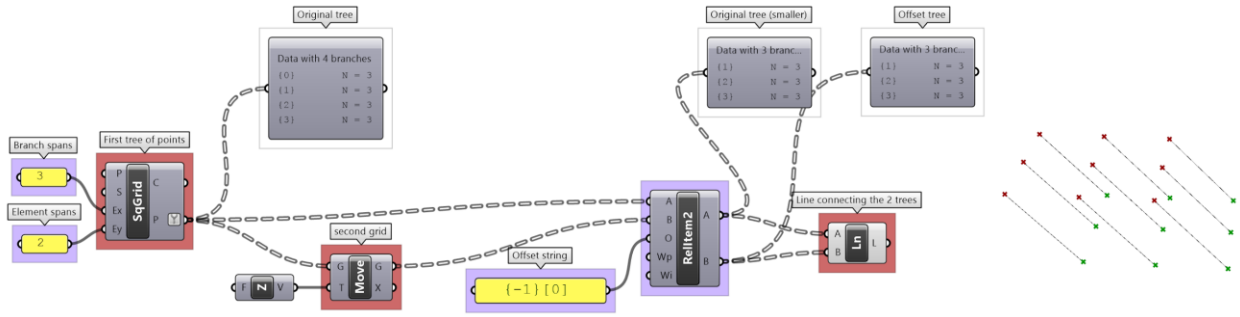
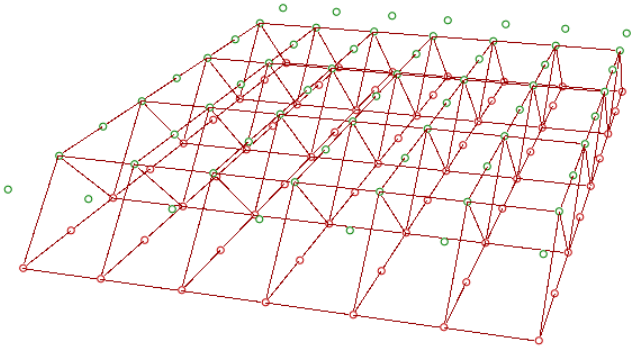
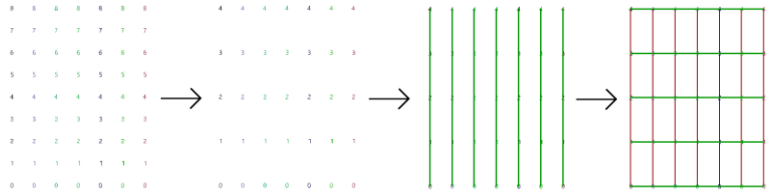
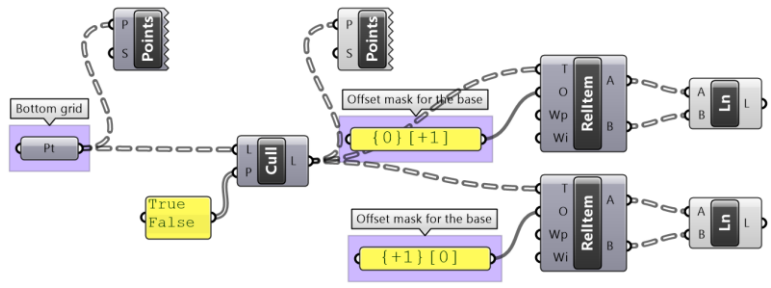
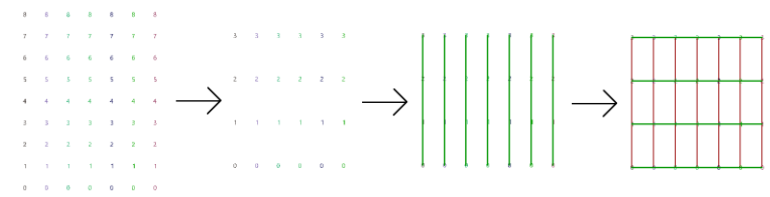


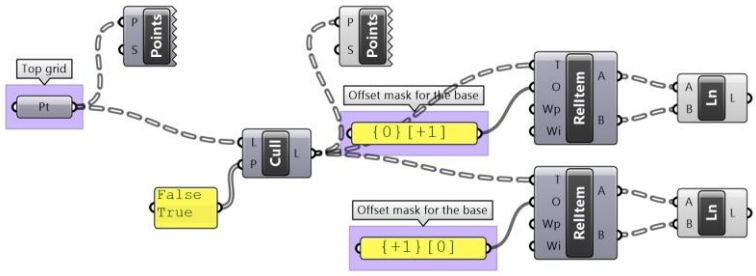
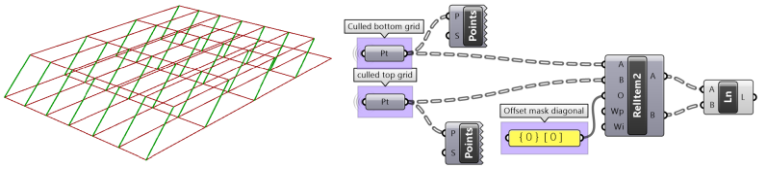
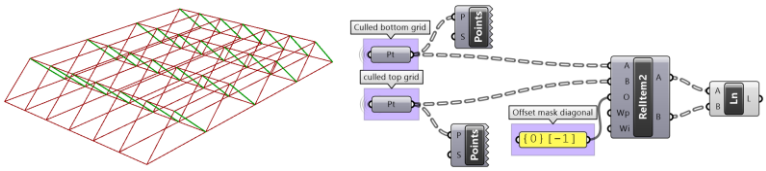
図 (81): GH での ***Relative Items*** の実装.

3_6_1_2 Relative Item チュートリアル #2

2 つの立体的なグリッド間の相対アイテムを使用して，図のような構造を生成してみましょう.



Solution	
下部ツリー内の接続	
ブランチの数は維持し，インデックスを1つおきに除外します（除外するインデックス 1，3，...）. ***Relative Item*** コンポーネントのオフセットを定義して，垂直接続と水平接続を作成します.	
Grasshopper 定義	
上部ツリー内の接続	
ブランチの数は維持し，インデックスを1つおきに除外します（除外するインデックス 0，2，...）. Relative Item コンポーネントのオフセットを定義して，垂直接続と水平接続を作成します.	

Grasshopper 定義	
2 つのツリー間の接続	
点を除外済みのグリッドを利用します．***Relative Items*** コンポーネントのオフセットを定義して，上部と下部をつなぐ接続の 1 つ目のセットを作成します：{0} [0]	
Relative Items コンポーネントのもう一つのオフセットを定義して，上部と下部をつなぐ接続の 2 つ目のセットを作成します：{0}[-1]	

3_6_2: Split Tree でツリーを自在に分割

ツリーの一部を選択したり，2 つに分割したりする GH の ***Split tree*** 機能は，非常に強力です．ツリーの出力を陽（Positive）に指定する文字列（Mask）を利用してツリーの分割が可能です．余ったものは陰（Negative）のツリーと呼ばれ，出力として与えられます．すべてのツリーはブランチとインデックスで構成されているため，Mask には，これらのツリー内のどのブランチとインデックスを分割するかに関する情報を含める必要があります．以下に Mask のルールを示します．

Split Tree の Mask: 構文とルール	
{ ; ; }	ツリーのブランチの Mask は，波括弧で囲みます．
[ ]	アイテム（葉）の Mask は角括弧で囲みます．すべてのアイテムを選択したい場合，省略するか，[*]を使用します．
( )	丸括弧はグループ化に用います．
*****	パス内の任意の整数．アスタリスクを使用すると，パスに関わらず，すべてのブランチを含めることができます．
?	任意の 1 つの整数．
6	特定の整数．

!6	特定の整数を除く値.
(2,6,7)	グループ内の特定の複数の整数.
!(2,6,7)	グループ内の特定の複数の整数を除く値.
(2 to 20)	範囲内に含まれるすべての整数 (2 と 20 も含む).
!(2 to 20)	範囲内に含まれるすべての整数を除く値.
(0,2,...)	繰り返しパターンに当てはまる整数．繰り返しパターンは，少なくとも 2 つの整数である必要があり，後の整数は前の整数よりも大きくする必要があります.
(0,2,...,48)	繰り返しパターンに当てはまる整数（上限あり）．オプションとして 3 つのドットの後ろに上限の値を記述できます.
!(3,5,...)	繰り返しパターンに当てはまらない部分の整数．繰り返しパターンは，左には拡張せず，右にのみ拡張されます．したがって，このルールでは，0, 1, 2, 4, 6, 8, 10, 12 と残りのすべての偶数を選択します.
!(7,10,21,...,425)	繰り返しパターンに当てはまらない部分の整数（上限あり）.
{ * }[(0 to 4) **or** (6,11,41) **]**	ブール演算子（and/or）を使用して，2 つ以上のルールを組み合わせることができます．この例では，ツリーのすべてのブランチの最初の 5 つのアイテムを選択し，さらに 7, 12, 42 番目のアイテムを選択します.

以下は，有効な分割 Mask の例です.

ブランチを指定して分割	
{ * }	すべてを選択（ツリー全体が Positive から出力，Negative は empty）
{ *; 2 **}**	3 番目のブランチの選択
{ *; (0,1) **}**	前から 2 本のブランチの選択
{ *; (0, 2, …) **}**	すべての偶数ブランチの選択
ブランチとアイテムを選択して分割	
{ * }[(1,3,...)**]**	すべてのブランチの奇数インデックスのアイテムを選択
{ *; 0 **}[**(1,3,...)**]**	1 本目のブランチの奇数インデックスのアイテムを選択
{ *; (0, 2) **}[**(1,3,...)**]**	1 本目と 3 本目のブランチの奇数インデックスのアイテムを選択
{*; (0,2,...) **} [** (1,3,...) **]**	偶数ブランチの奇数インデックスのアイテムを選択
{*; (0,2,...) **} [**(0) **or** (1,3,...)**]**	偶数ブランチの 0 および奇数インデックスのアイテムを選択

Split Tree は，点グリッドがあり，そのサブセットを変換したい場合によく用いられます．分割すると，元のツリーの構造が保持され，分割で除外された要素は null に置き換えられます．したがって，***Split Tree*** で変換を適用すると，再結合が簡単です．
7 つのブランチと各ブランチに 11 のアイテムを持つグリッドがあり，インデックス 1～3 と 7～9 のアイテムをシフトしたいとします．***Split Tree*** と Mask（{*} [(1,2,3) or (7,8,9)]）を用いてシフトしたい点を分離できます．Positive ツリーをシフト後， Negative ツリーを再結合します．

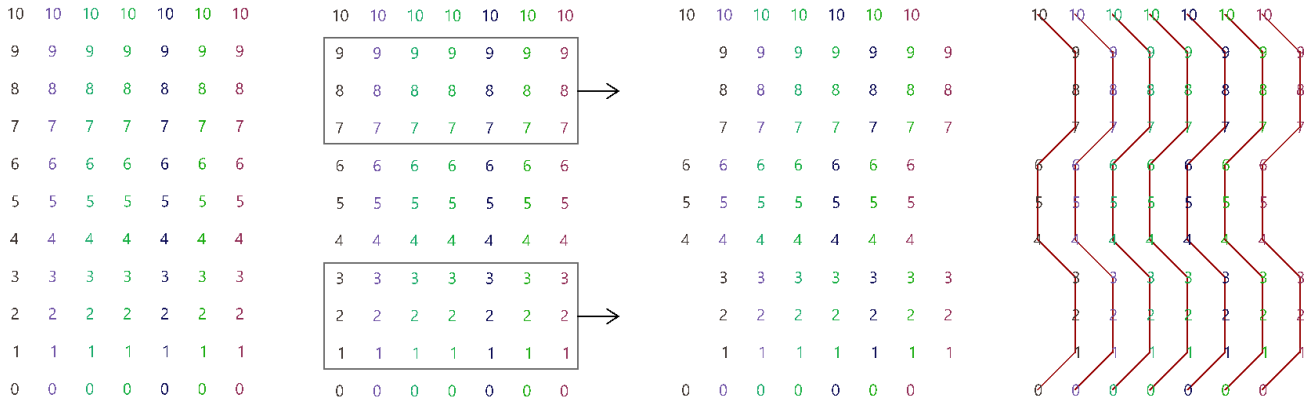


図 (82): ***Split tree*** では，ツリーの一部を処理して，元の構造に戻すことができます．

以下は，***Split Tree*** を使用して上記を実現する GH 定義の例です．

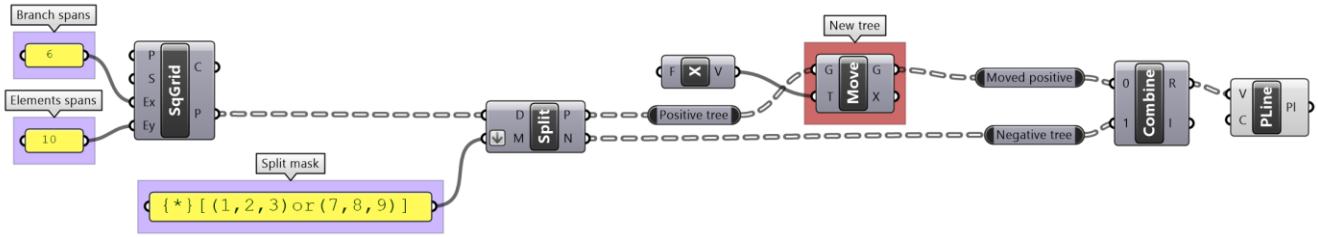
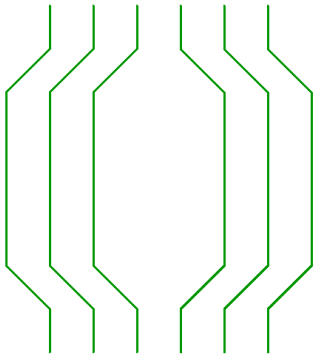


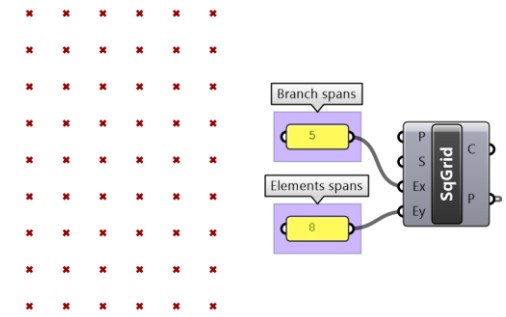
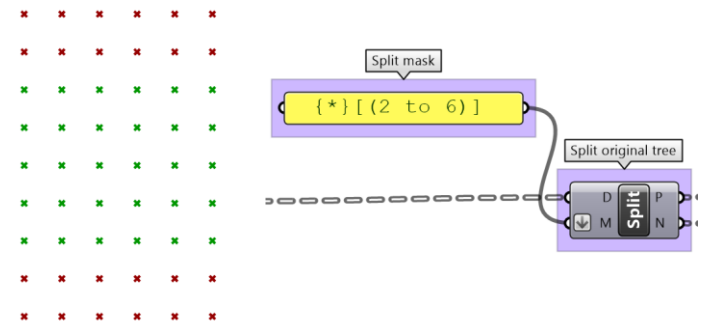
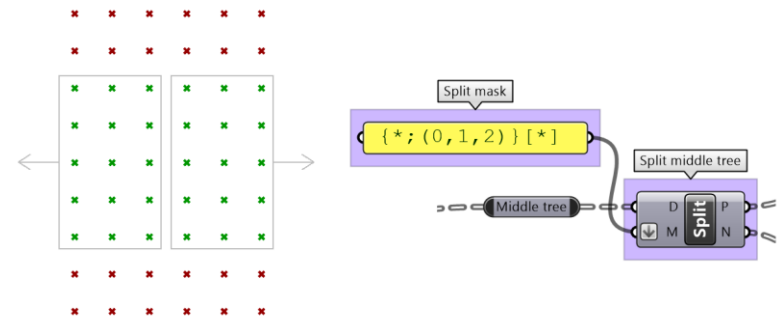
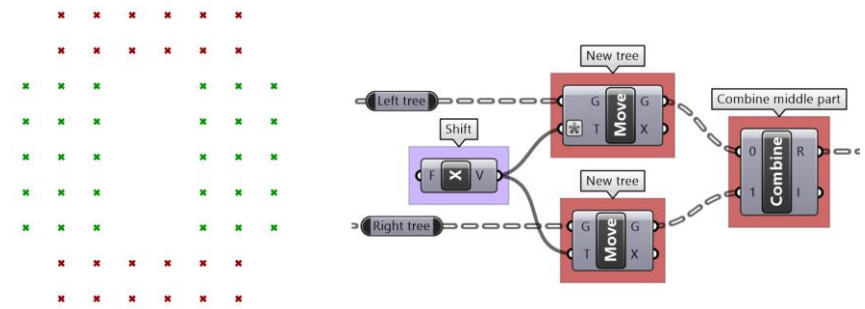
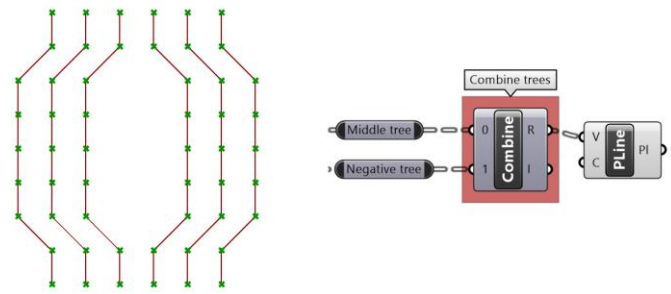
図 (83): ***Split tree*** の GH 実装例．

Relative Trees よりも ***Split Tree*** を使用するメリットの 1 つは，分割用の Mask の適用範囲が非常に広く，ツリーの目的の部分を分離するのが容易なことです．また，データ構造は Positive ツリーと Negative ツリーで元の構造が保持されるため，パーツの処理後にツリーを簡単に再結合できます．

3_6_2_1 Split Tree チュートリアル #1

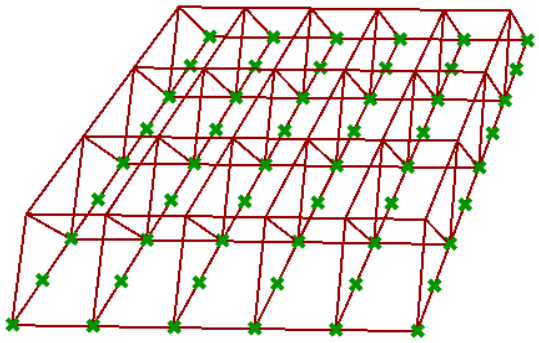
6x9 のグリッドを仮定し，***Split Tree*** を用いて以下のような形状を生成してみましょう．

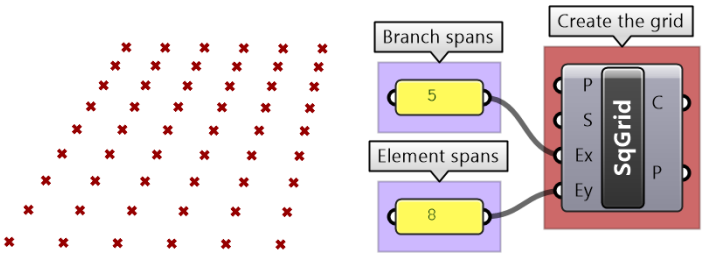
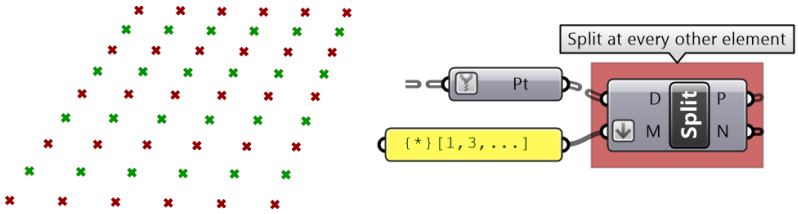
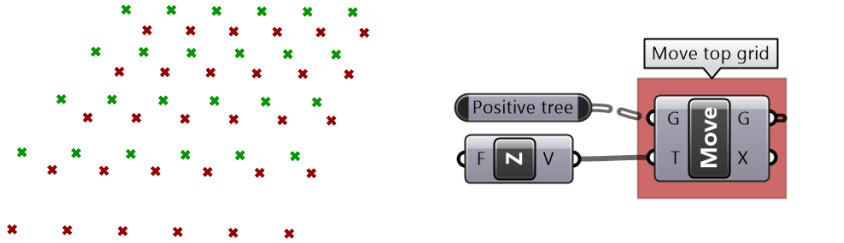
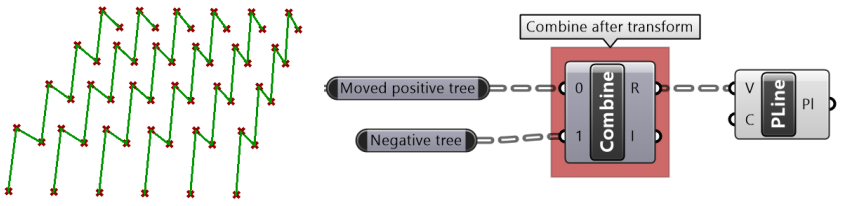
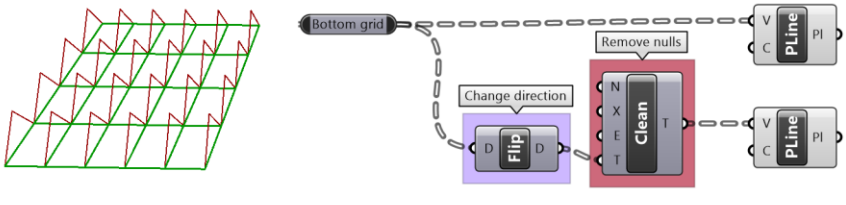


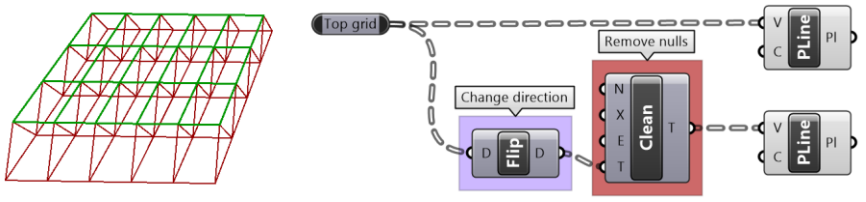
アルゴリズムの手順	
グリッドを生成します	
Split Tree で中間部分を分離します	
Split Tree で中間部分をさらに半分にします	
半分にした 2 つの部分をそれぞれ反対方向に ***Move*** し，処理後に再結合します	
中間部分と残りのツリーを再結合し，それぞれのブランチのアイテムでポリラインを作成します	

3_6_2_2 Split Tree チュートリアル #2

グリッドを仮定し，***Split Tree*** 機能を使用して図のようなトラス構造を作成してみましょう．



Solution	
6x9 のグリッドを作成します	
Split Tree でアイテムを 1 つおきに分けます	
Positive ツリーを鉛直方向に ***Move*** します	
Positive ツリーと Negative ツリーを再結合し，各ブランチのアイテムでポリラインを作成します	
Negative ツリーから下部のカーブを作成します	

Positive ツリーから上部のカーブを作成します	

3_6_3: Path Mapper でデータ構造を書き換え

GH でデータツリーなどの複雑なデータ構造を扱うときに，ツリー内の要素を簡潔にしたり，並べ替えたいと思うような場面があるでしょう．そのために，GH には ***Flatten***，***Graft***，***Flip Matrix*** のようないくつかの便利なコンポーネントがありますが，複数ツリーの処理やデータ配置に工夫が必要な場合，これらでは不十分な場合があります．そのような場合に，ツリー内の要素の再編成やツリー構造の変更に役立つ非常に強力なコンポーネントとして，***Path Mapper*** というコンポーネントがあります．これはエラーになりやすく，直感的には扱いづらいかもしれませんが，場合によっては解決策を見つけるための唯一の方法にもなり得ますので，ここで紹介しておきます．

Path Mapper は，Source パスと Target パスの間でデータをマッピングします．Source パスは固定で，入力ツリーによって決まります．ユーザーは Target パスのみ設定できます．Target パスの設定時は，以下の一覧の定数も使用できます．

item_count	現在のブランチ内のアイテムの数
path_count	ツリー内のパス（ブランチ）の数
path_index	現在のパスのインデックス番号

まず，***Path Mapper*** 内の組込の Mapping を使用して構文を理解することから始めましょう．

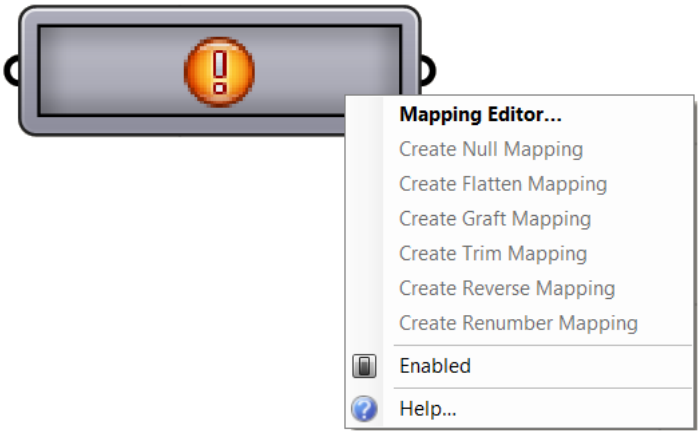
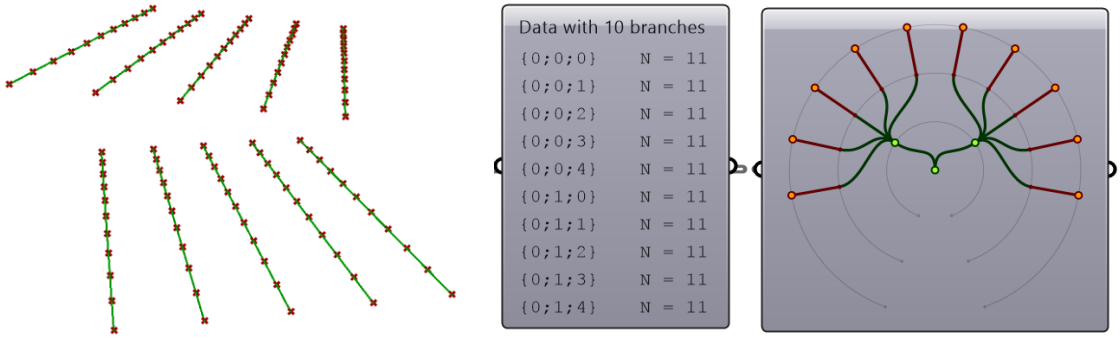
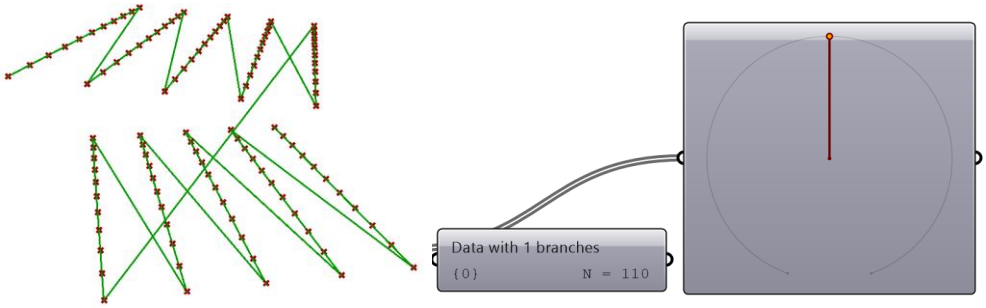
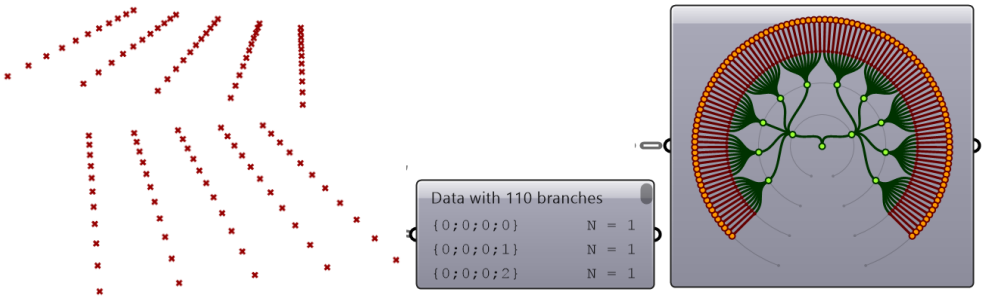
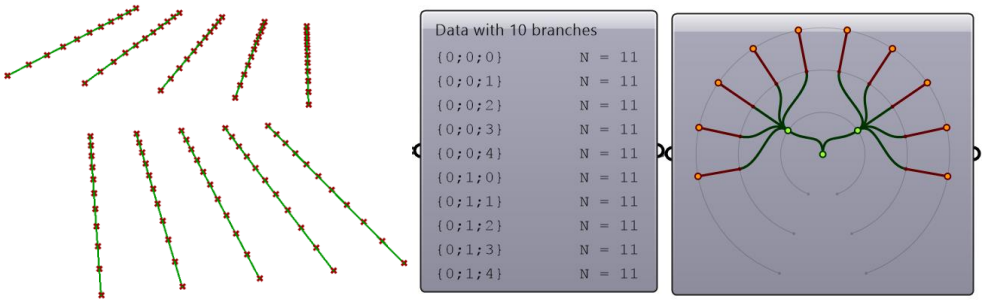
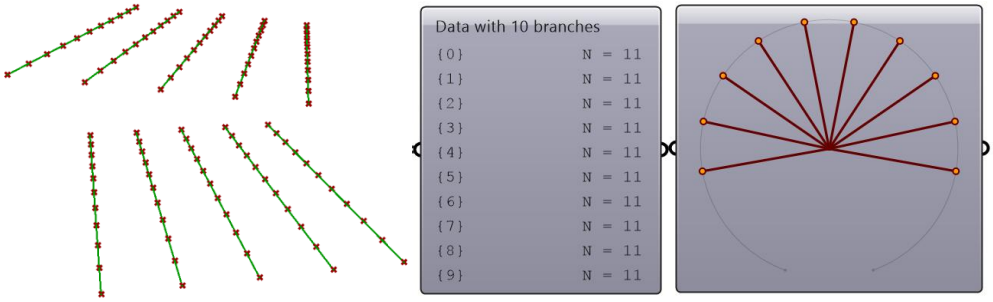


図 (84): ***Path Mapper*** の組込の Mapping.

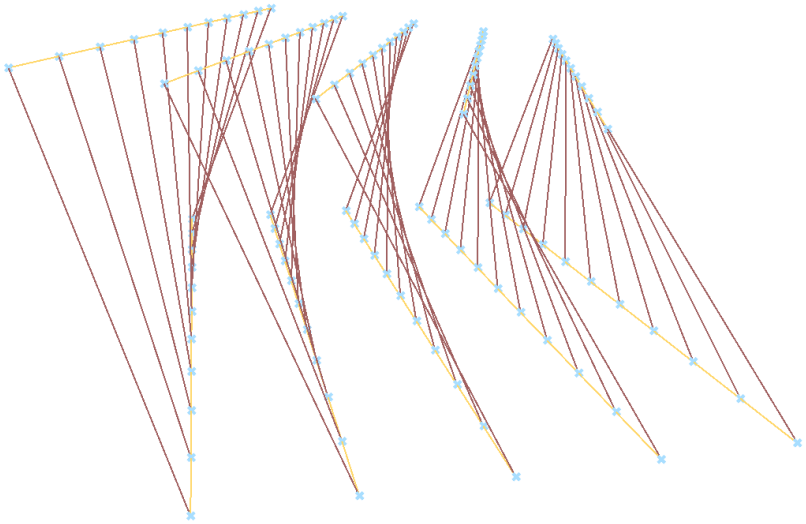
次の例では，入力ツリーとして，2つの点グリッド（2つのツリー）があります．***Polyline*** を使うと，ブランチ毎にポリラインが 1 つ作成されるのでデータ構造がわかりやすくなります．組込の Mapping を適用すると，構造と点の接続へどのように影響するかを確認してみましょう．

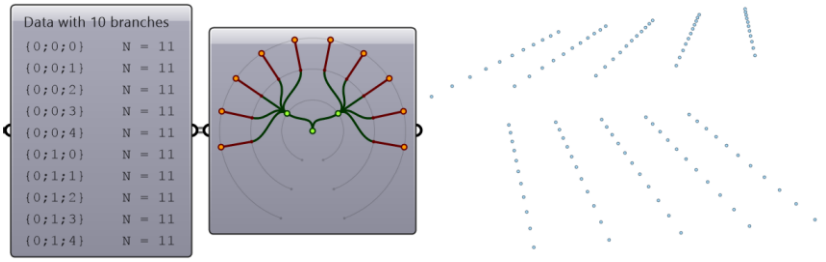
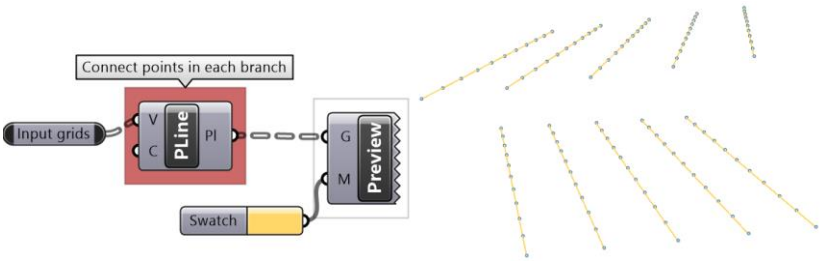
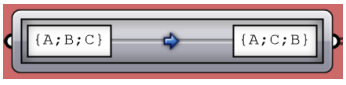
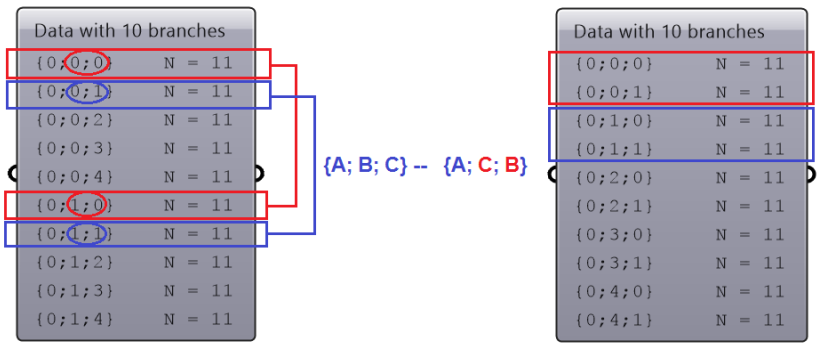


Path Mapper の組込 Mapping	
Null Mapping	何も変化しません.
Flatten Mapping （平坦化）	
Graft Mapping （接ぎ木）	
Reverse Mapping （反転）	
Renumber Mapping （再割り振り）	

3_6_3_1 Path Mapper チュートリアル #1

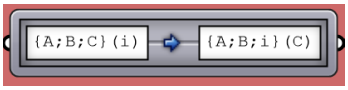
点のツリー構造を仮定し，以下のような接続を作成してみましょう．

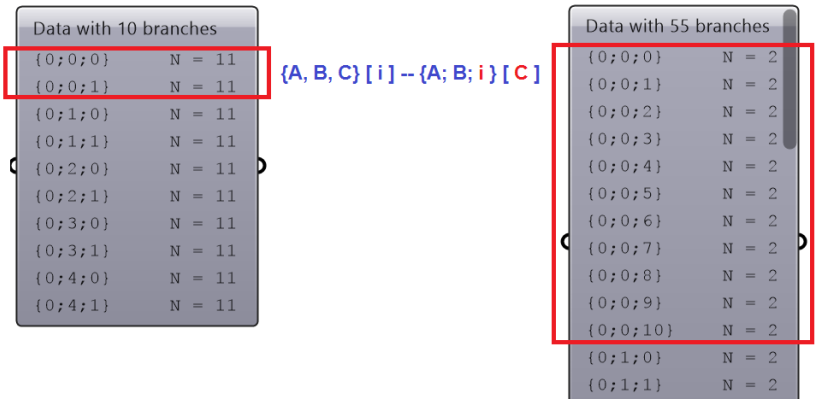


Solution	
入力は，2 つのツリー（それぞれに 5 つのブランチ，各ブランチに 11 のアイテム）で，合計で 10 のブランチがあります．	
Polyline を使ってブランチ毎のアイテムの接続を確認できます．	
垂直方向の接続を作成するには，2 つのツリー間で対応するアイテム毎にブランチを作成し，***Polyline*** でそれらを接続する必要があります． 1-ツリーのパスを分析します 2-目的のグループを生成するマッピングを考えます	
まず，2 つのツリー間で対応するブランチをグループ化します． これは，パスの後ろ 2 つの整数を入れ替えることで実現できます． 	

次に，5 本の根元のブランチをそれぞれ Flip します．ブランチにはそれぞれ 11 のアイテムがあるため，Flip すると，それぞれ 2 つのアイテムを持つ 11 のブランチが作成されます．合計 55 のブランチができます．

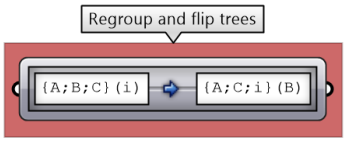
Flip するには，アイテムのインデックスとパスの最後の整数を入れ替えます．



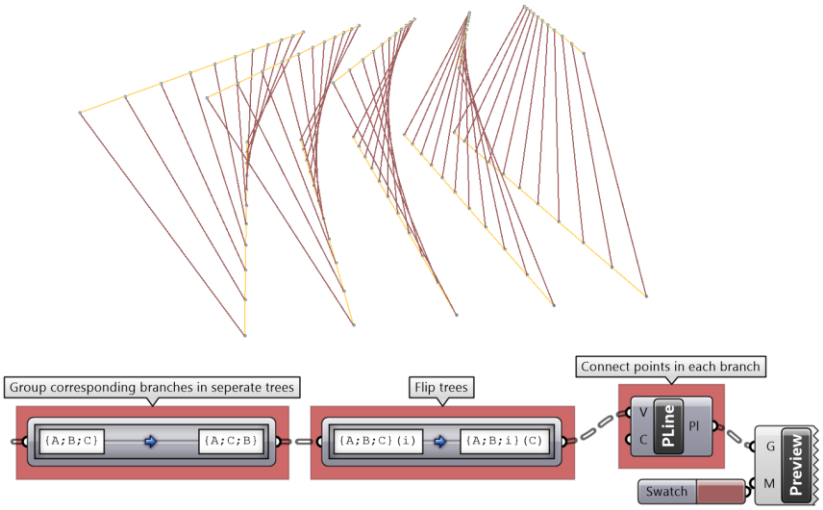


最後に，***Polyline*** で垂直な接続を作成します．

Note: 以下のように書けば，1 つのコンポーネントで 2 つのマッピングを合成することも可能です．

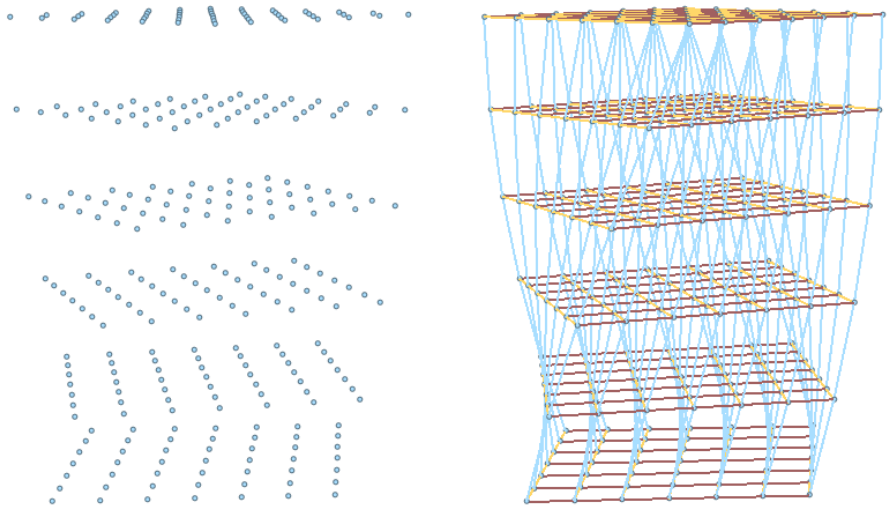


合成はいつも可能なわけではありませんが，可能な場合は，処理時間と容量を節約できます．

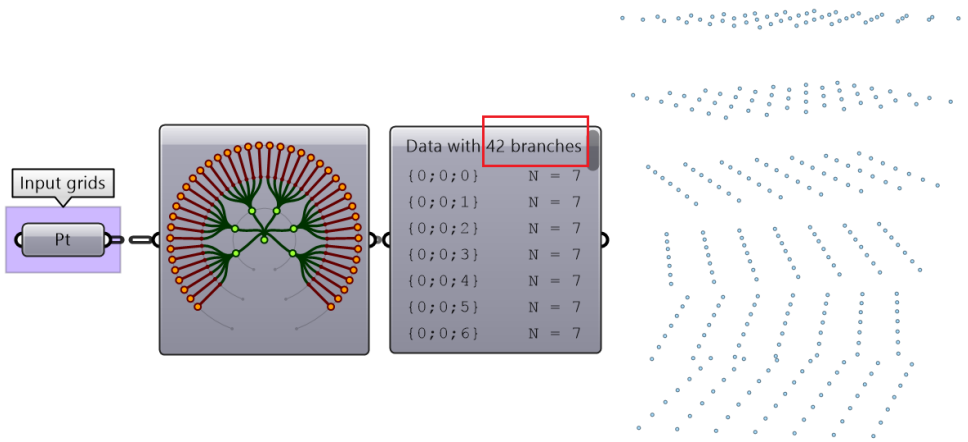
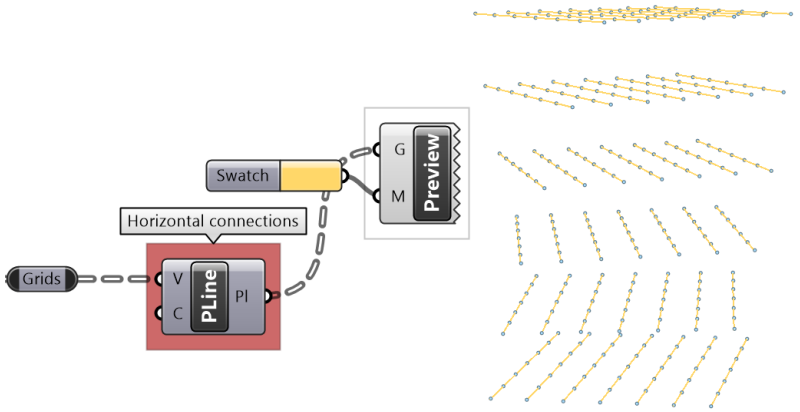
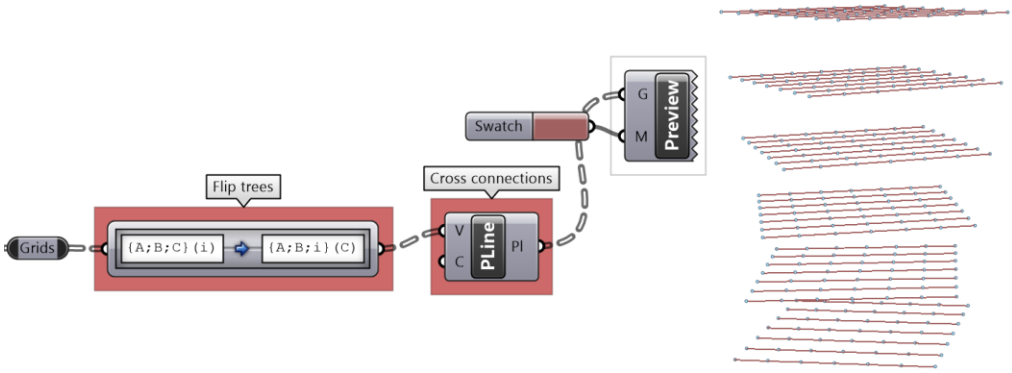
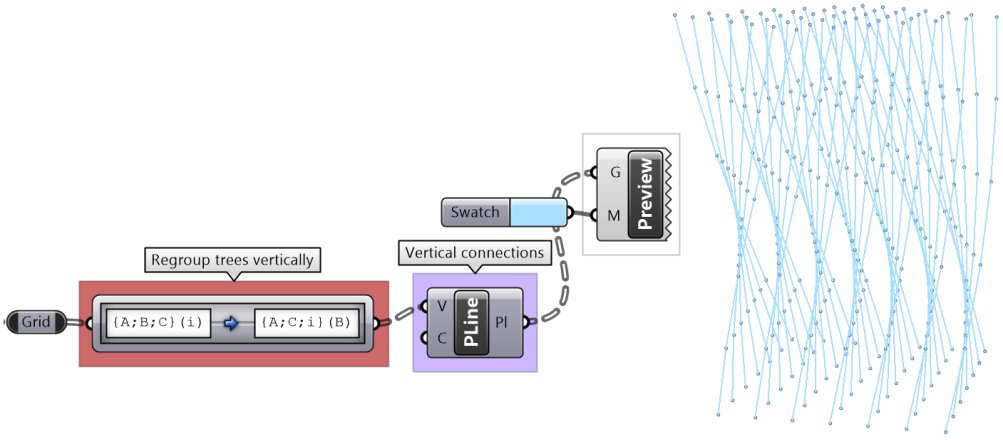


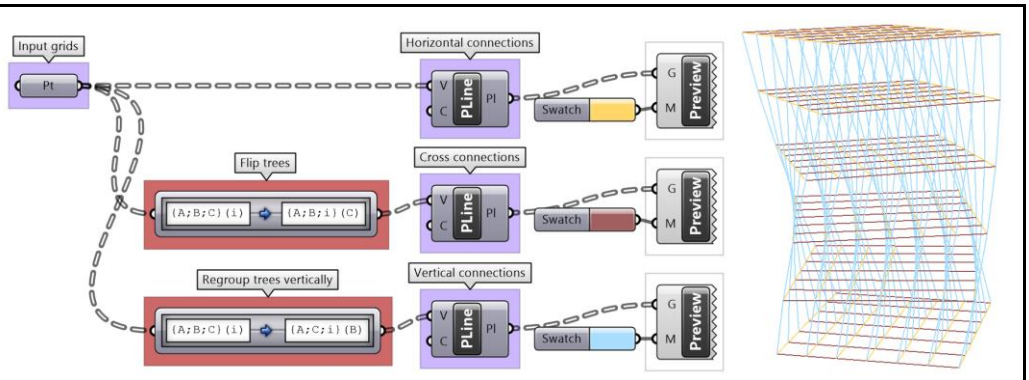
3_6_3_2 Path Mapper チュートリアル #2

点のツリー入力を仮定し，以下のような構造を作成してみましょう．



Solution

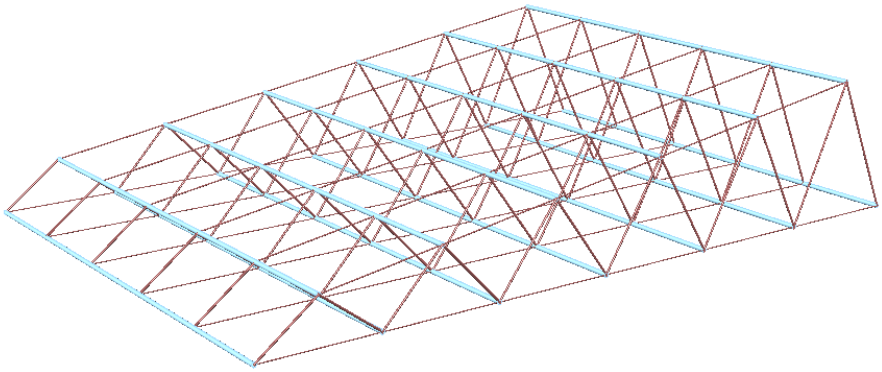
入力ツリーには 42 のブランチがあり，根元の 6 本のブランチにそれぞれ 7 本のブランチが付きます.	
Polyline を使ってブランチ毎のアイテムの接続を確認します.	
Path Mapper でブランチとアイテムのインデックスを入れ替え，ツリーを Flip します.	
Path Mapper でツリー内の該当するブランチのアイテムをグループ化し直します.	

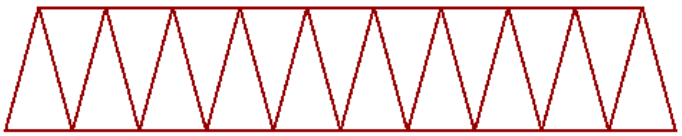
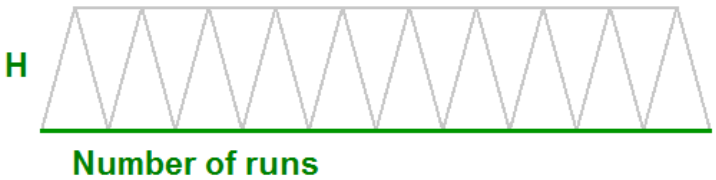
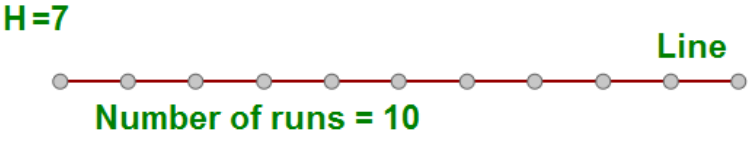
すべての接続を合わせれば完成です.	

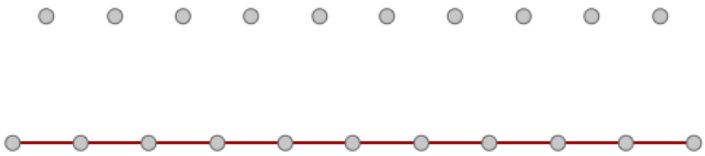
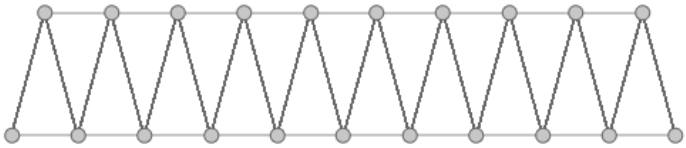
3_7: データ構造応用のチュートリアル

3_7_1: 傾斜する屋根

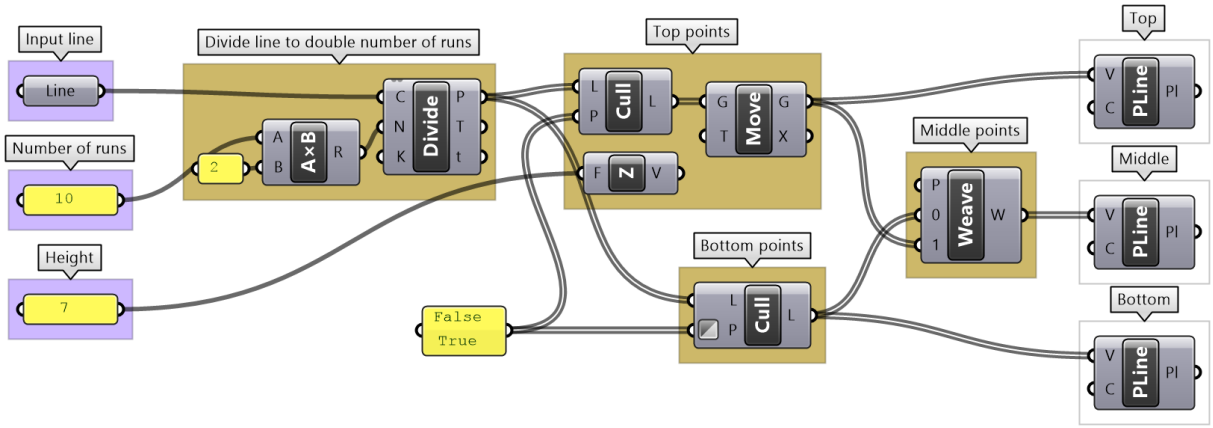
高さが徐々に変化する図のようなパラメトリックなトラス構造を作成してみましょう.



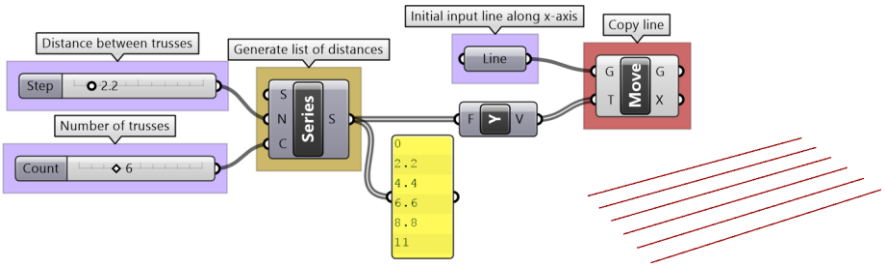
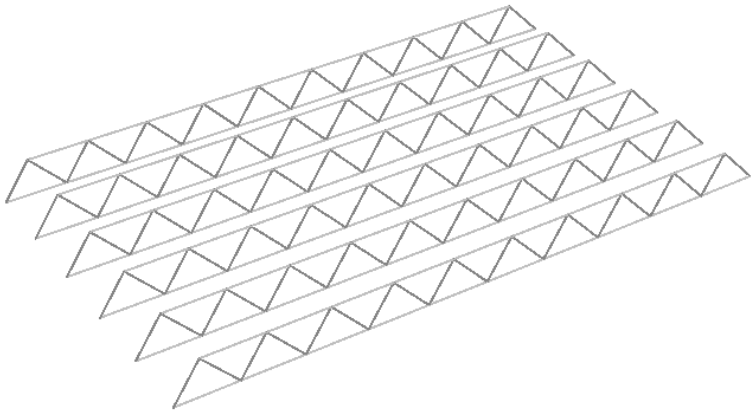
Solution	
アルゴリズムの分析：まずはシングルトラスで検証	
目的の出力となるシングルトラスを明確にします.	
初期値を定義します. 1- XY 平面上の基準線 2- 繰り返し数 3- 高さ	
アルゴリズムの手順を明確にします：	
入力値を準備します 基準線 L= Line 繰り返し数 R=10 高さ H = 7	
Line を 2*R で分割します	

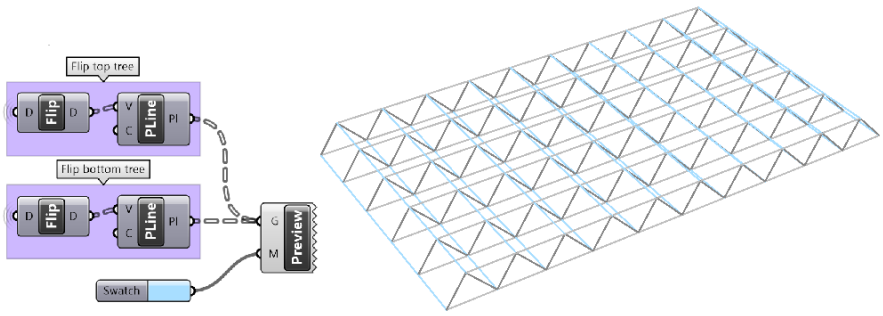
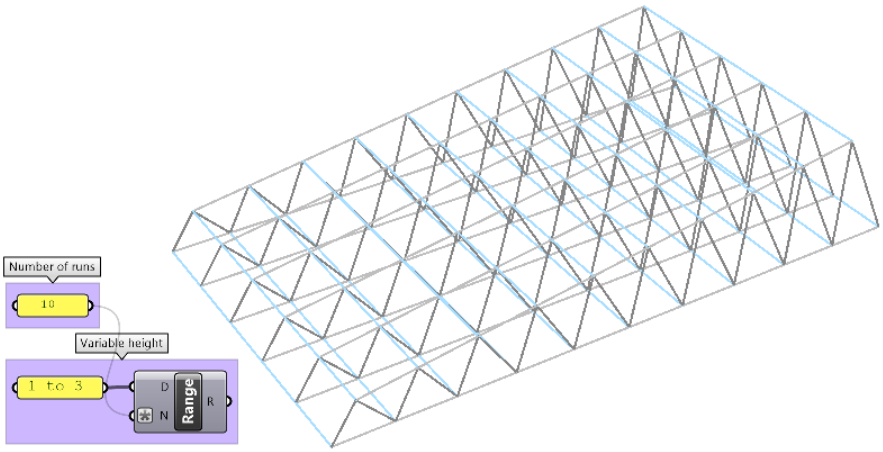
点を 1 つおきに Z 方向に高さ H 移動します.	
下部・上部・中央の梁用に整列した 3 つの点のセットを作成し，それぞれをポリラインで接続します.	

GH へのアルゴリズムの実装

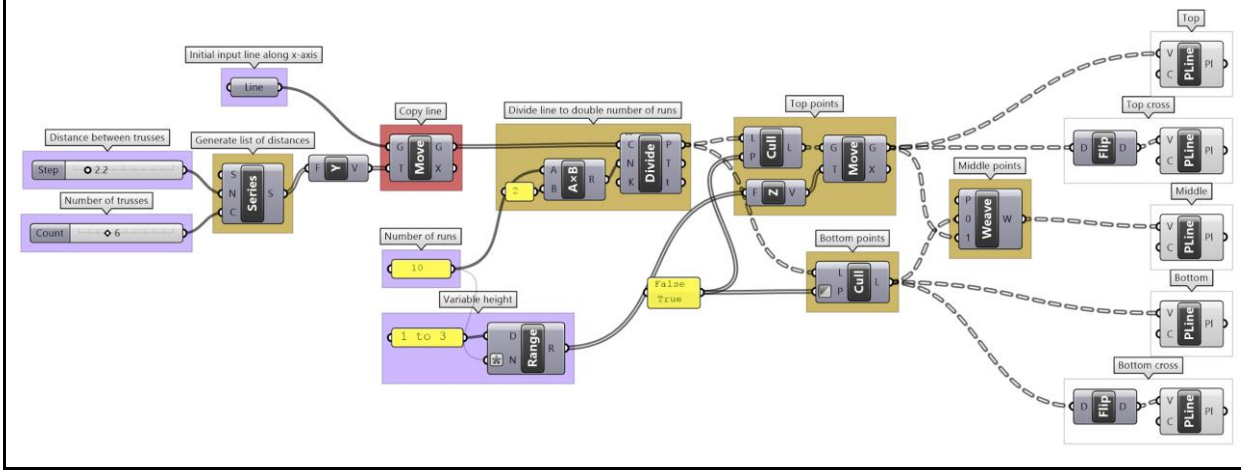


異なる高さの複数のトラスに拡張

初期値の Line を ***Series*** で Y 軸方向にコピーして，一連の基準線を作成します.	
単一アイテムの Line ではなく，Line のリストを入力として使用します. 3 つ（上・中・下）の点のリストではなく，ツリーができることに注意．このツリーは，トラスの数と等しい数のブランチを持つ点グリッドです.	

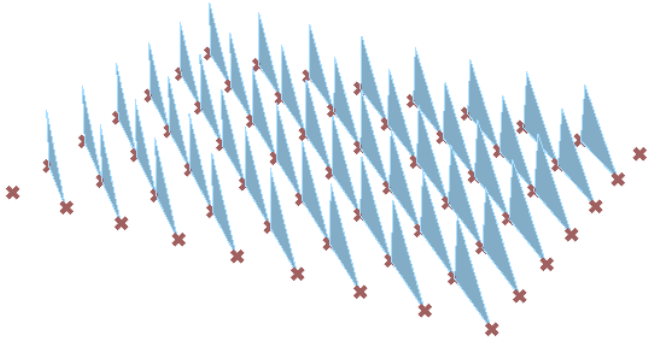
上部と下部のツリーを ***Flip Matrix*** で処理し，クロスする接続を作成します.	
徐変する高さを作成します.	

GH 実装の完成イメージ



3_7_2: 対角で三角形を生成

グリッドを仮定し，***Relative Item*** コンポーネントを使って対角で三角形を作成してみましょう．



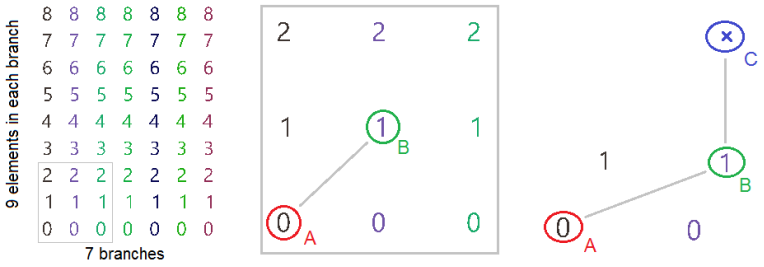
Solution

アルゴリズムの分析

三角形を生成するには，3 つのコーナー点が必要です．

2 点（A、B）のセットは，グリッド内にあります．B は A の対角の点です（相対インデックスは+1 ブランチ，+1 アイテム）．
3 つ目の点（C）のセットは，B を鉛直方向に移動させた点です．

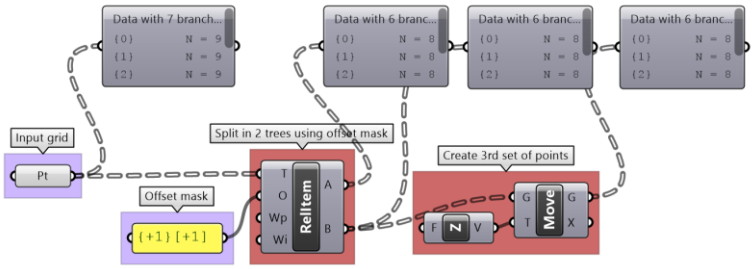
コーナー点をグループ化して接続し，その境界線でサーフェスを作成します．



GH への実装

Relative Item で A と B のセットを作成します（オフセットは {+1}[+1] ）．

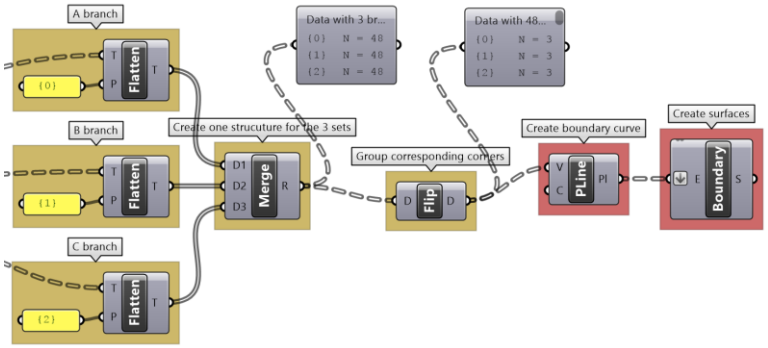
B のセットを Z 方向に移動します．



セット A・B・C の 3 つのブランチを合わせたツリーを作成します．

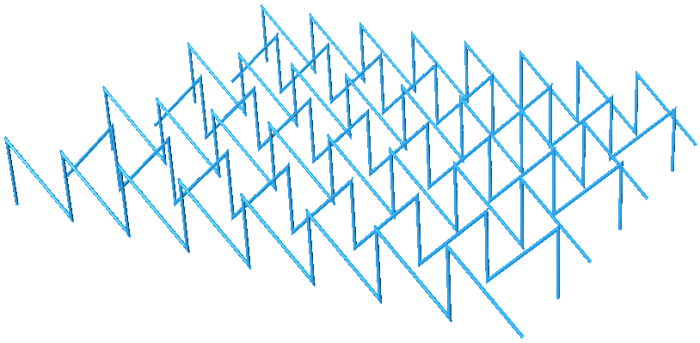
ツリーを Flip して，対応する点をグループ化します．

Polyline と ***Boundary*** でサーフェスを生成します．



3_7_3: ジグザグ構造

入力のグリッドを使用して，図のような構造を作成してみましょう．



Solution

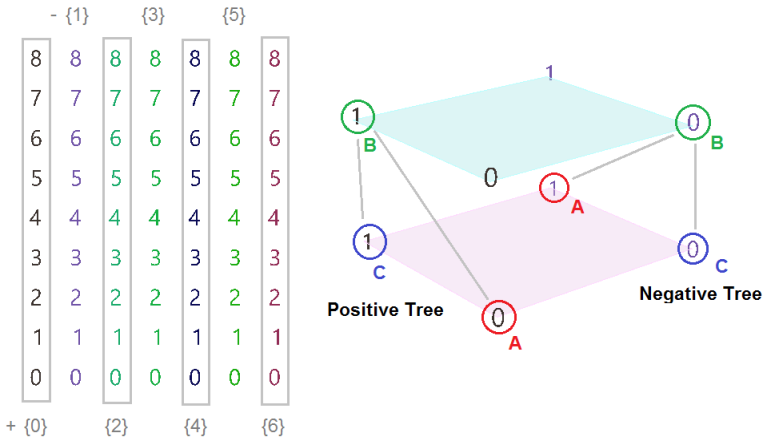
アルゴリズムの分析

ジグザグは方向が変わるため，グリッドを Positive と Negative の 2 つの部分に分割するのが最適です．

Positive ツリーで 3 組の点を特定し，整列します．

Negative ツリーのブランチのアイテムを反転し，3 組の点を特定し，整列します．

2 つのツリーを Merge して点をつなげてジオメトリを作成します．



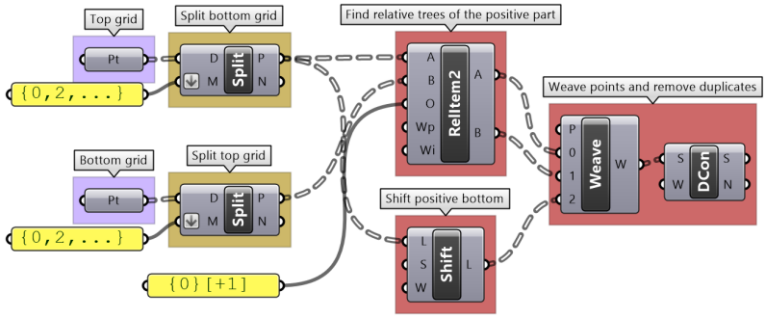
GH への実装

上部・下部グリッドそれぞれ，***Split Tree*** を使って，Positive ツリーと Negative ツリーを作成します（分割用の Mask は，{0,2,...}）．

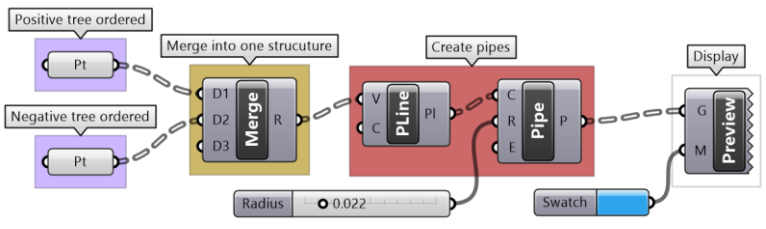
Relative Items を使用して，オフセット{0}[+1] で A と B のツリーを作成します．

Shift で C のツリーを作成します．

Weave でデータを合わせ，重複アイテムを削除します．

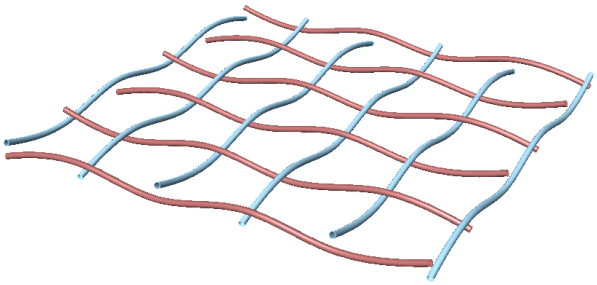


処理後の Positive ツリーと Negative ツリーを ***Merge*** して，***Polyline*** と ***Pipe*** を使用してジオメトリを生成します．



3_7_4: 糸の編み込み

長方形グリッドを初期入力として使用して，糸を平面状に編み込んだような形状を作成してみましょう．密度とサイズを調整できるようにします．Bonus：平面ではなく，曲面にも適用できるようにしてみましょう．



アルゴリズムの分析

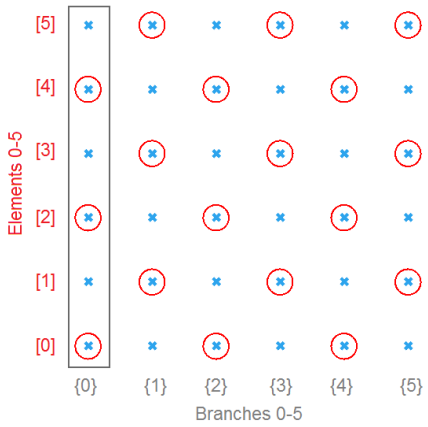
入力は，縦糸になる方向のブランチを持つ平面状の正方形グリッドです．

グリッドを，ブランチ毎にアイテムが交互になる2つのパートに分割します．

1 番目のパートを上に移動，2 番目のパートを下に移動して，再結合します．

各ブランチの点を通る曲線を描きます．

グリッドを Flip して，同様の処理を行い，横糸を作成します．

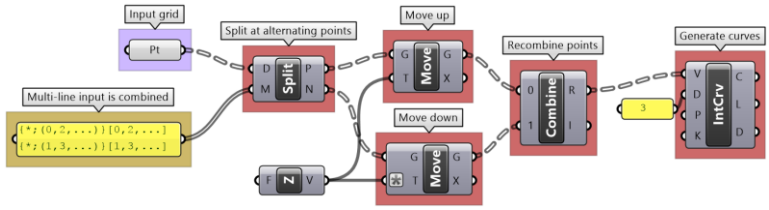


GH への実装

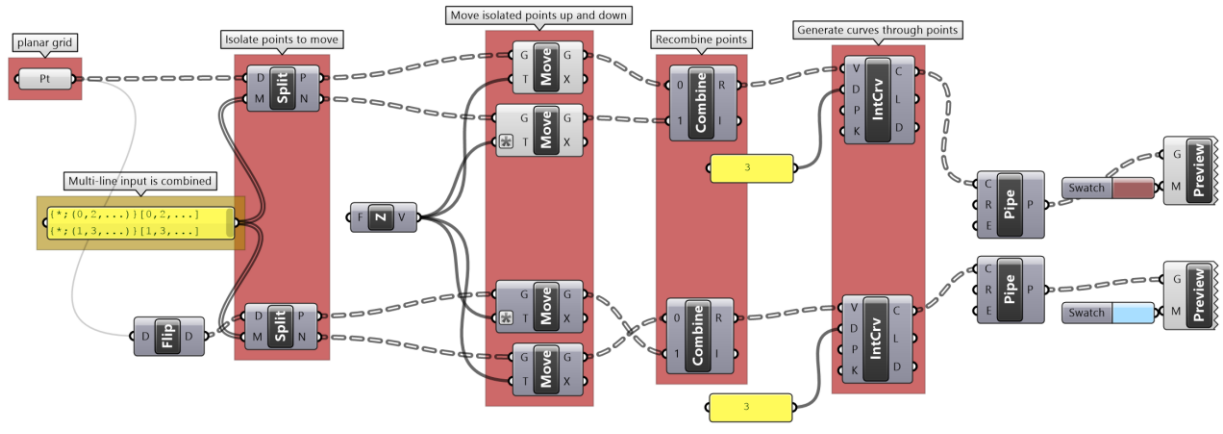
Split Tree で点を交互に分離し，上方向・下方向に ***Move*** します．

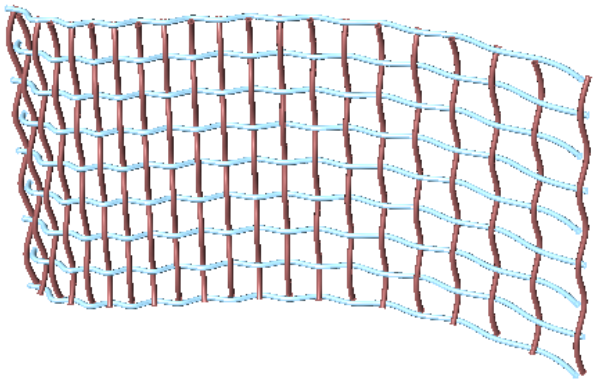
点を ***Combine*** し，***IntCrv*** で各ブランチの点をつないで曲線を作成します．

ツリーを ***Flip*** し，***Split***，***Combine***，***IntCrv*** を繰り返してもう一方向の曲線を作成します．



全体像





Bonus solution

Z 軸に対して点を上下移動するのではなく，各点におけるサーフェスの法線（Normal）方向を使用します.
Note：法線と点グリッドのデータ構造がマッチしているか注意しましょう.