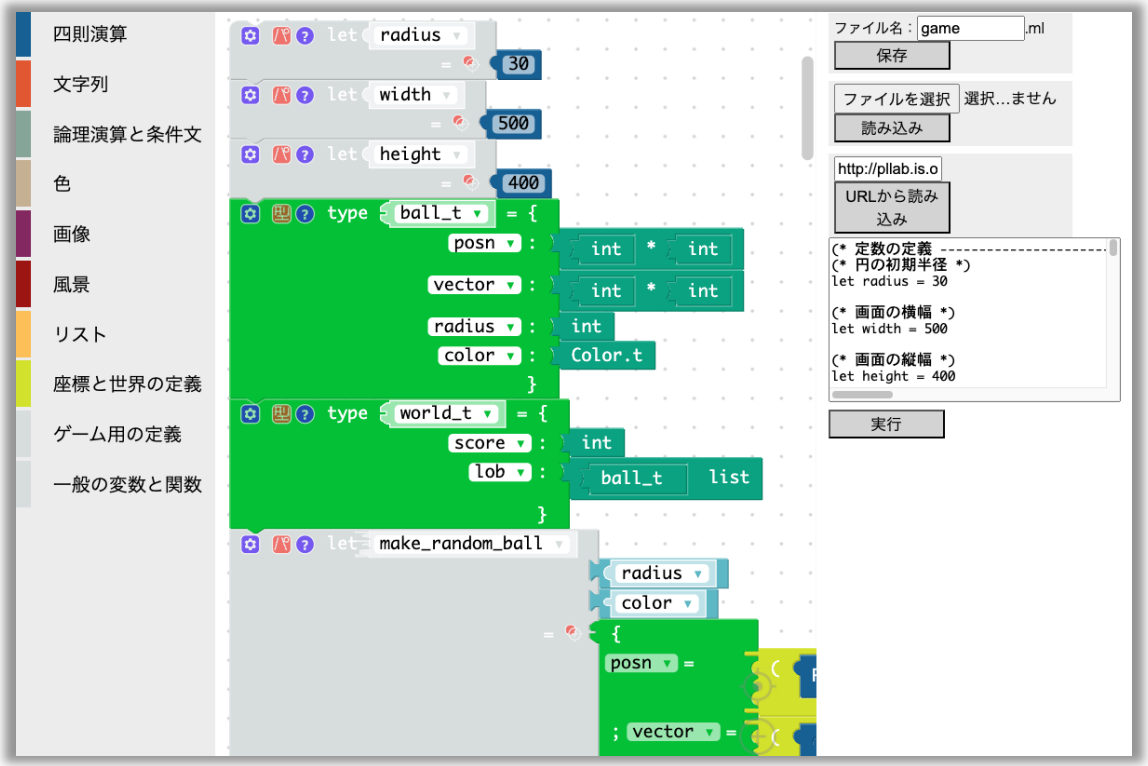
2025年3月5日
第27回プログラミングおよびプログラミング言語ワークショップ

# OCaml Blockly チュートリアルの改良と専用記述言語

田村優衣, 浅井健一
お茶の水女子大学

# OCamlのビジュアルプログラミングエディタ

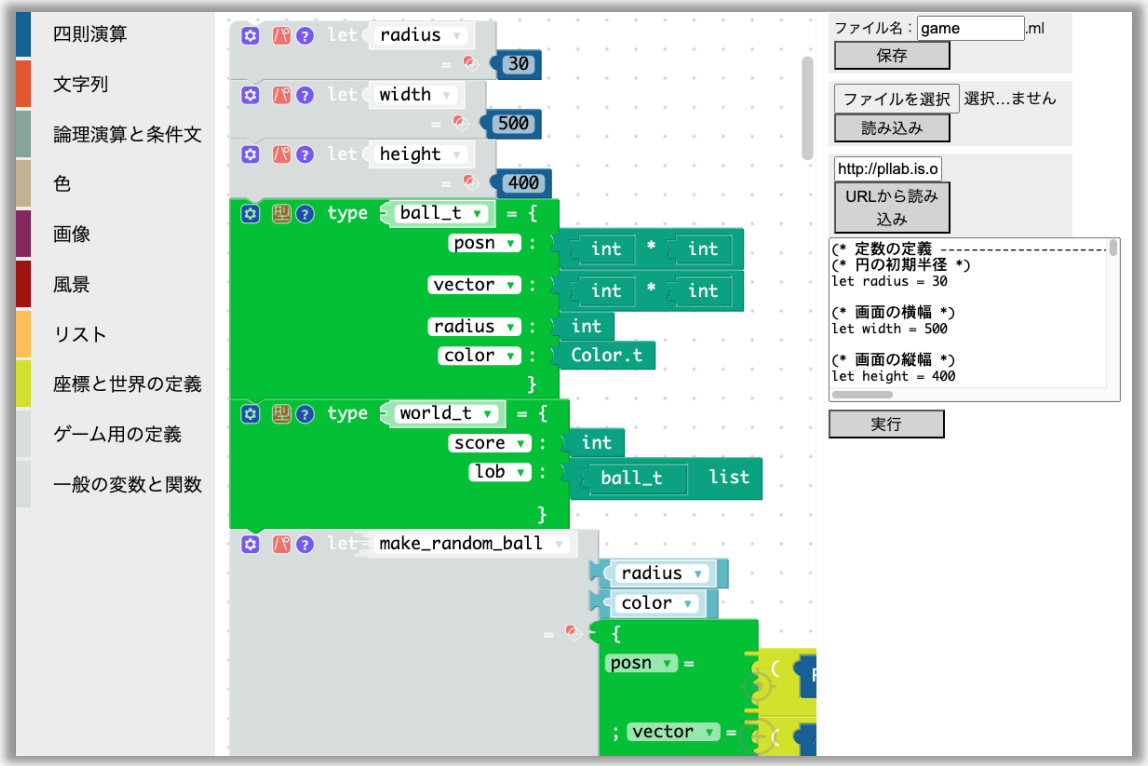
- コンパイルエラーとなるプログラムは構築できない
  - 本質的でない問題が起こりにくい
- 大学の授業で使用
  - 関数型言語（専門科目）
  - ゲームプログラミング入門（全学部）



# OCamlのビジュアルプログラミングエディタ

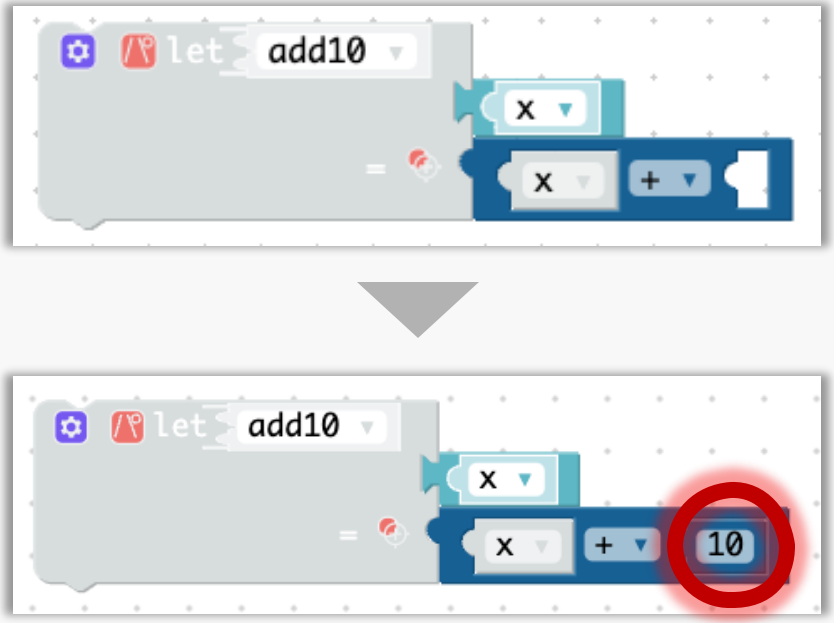
- コンパイルエラーとなるプログラムは構築できない
  - 本質的でない問題が起こりにくい
- 大学の授業で使用
  - 関数型言語（専門科目）
  - ゲームプログラミング入門（全学部）

チュートリアルサイト

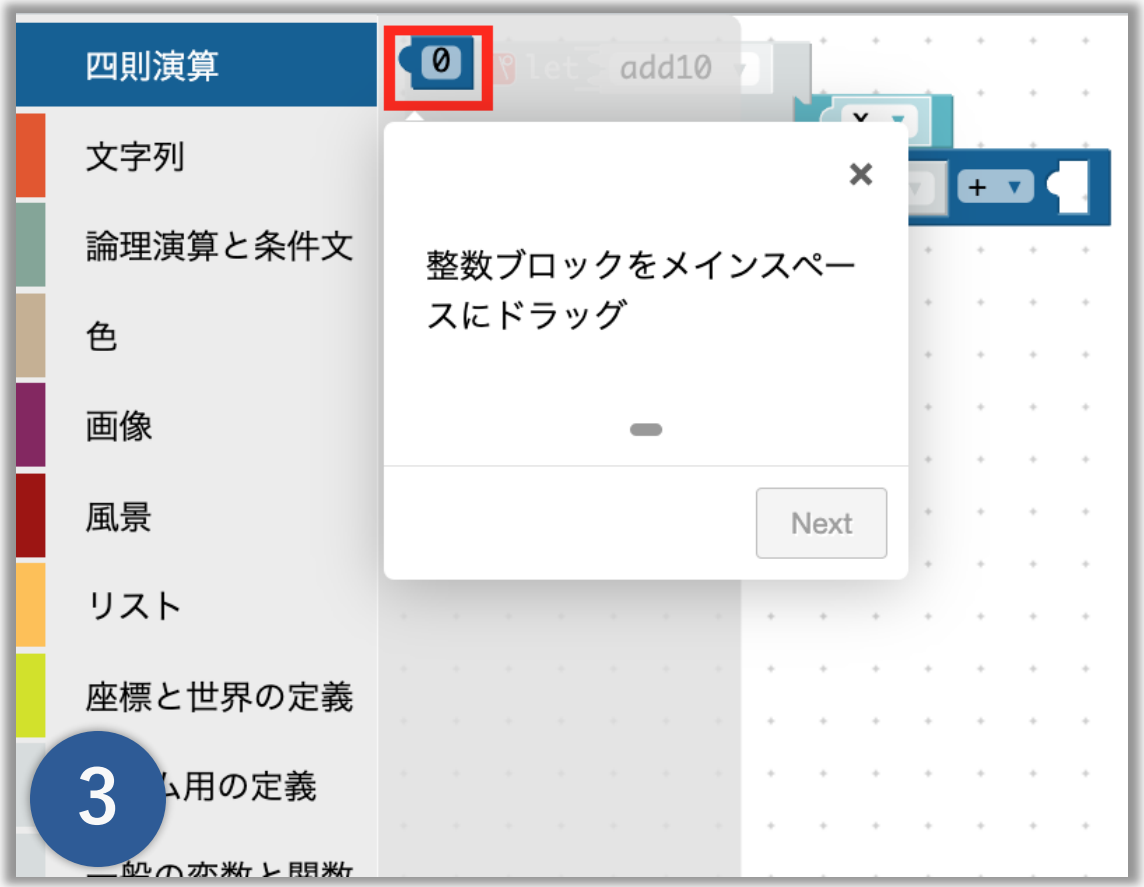


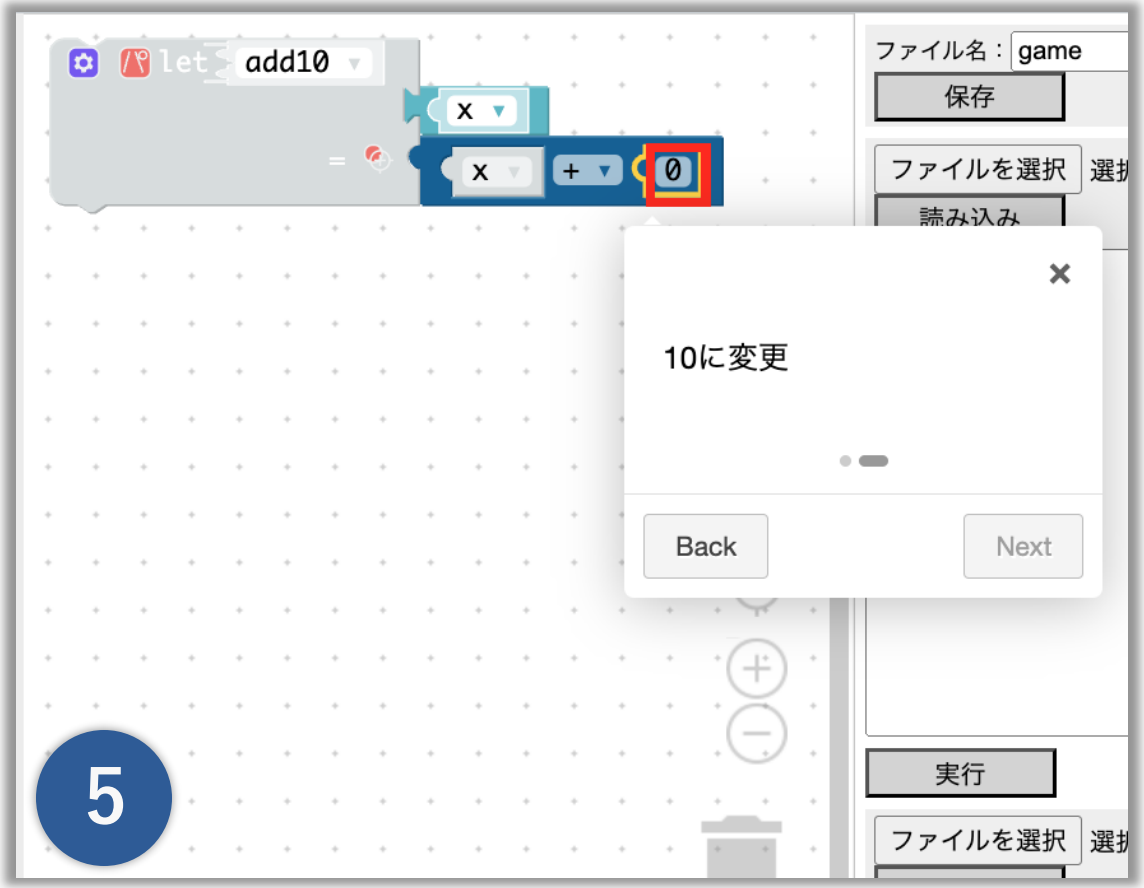
# OCaml Blockly の画面にガイドを表示

- **関数add10: 整数を受け取って10を足して返す**
- **デモ: 関数add10の完成**
  1. 「関数add10を完成させましょう。...」
  2. 四則演算メニューを開く
  3. 整数ブロックを四則演算ブロックの右側に接続
  4. 「値を10に変更します。...」
  5. 整数ブロックの値を10に変更
  6. チュートリアルクリア



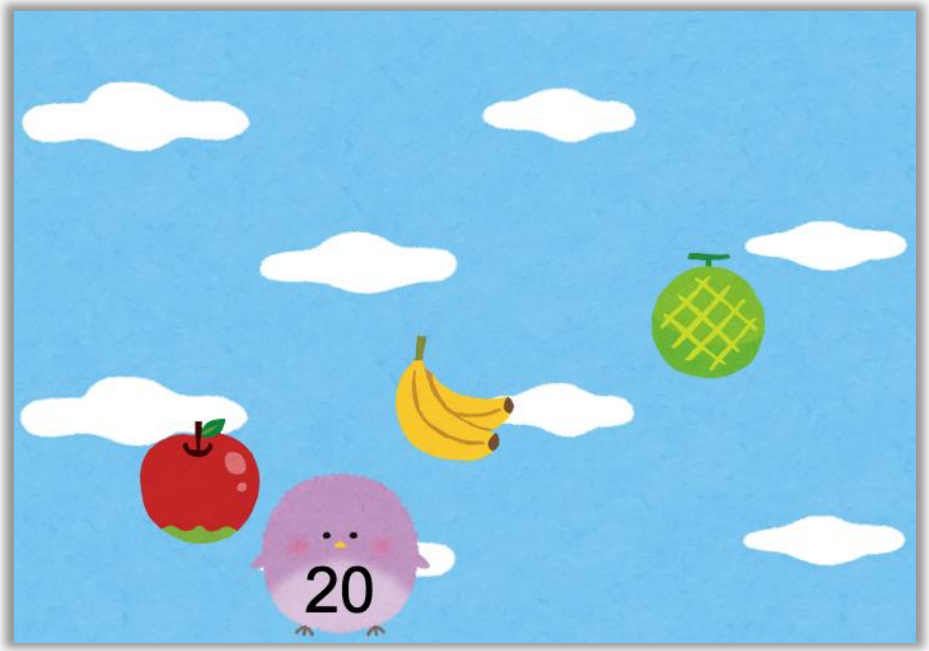
# 行う操作が明確・ひとりで学べる環境



# OCaml Blockly チュートリアルを活用する

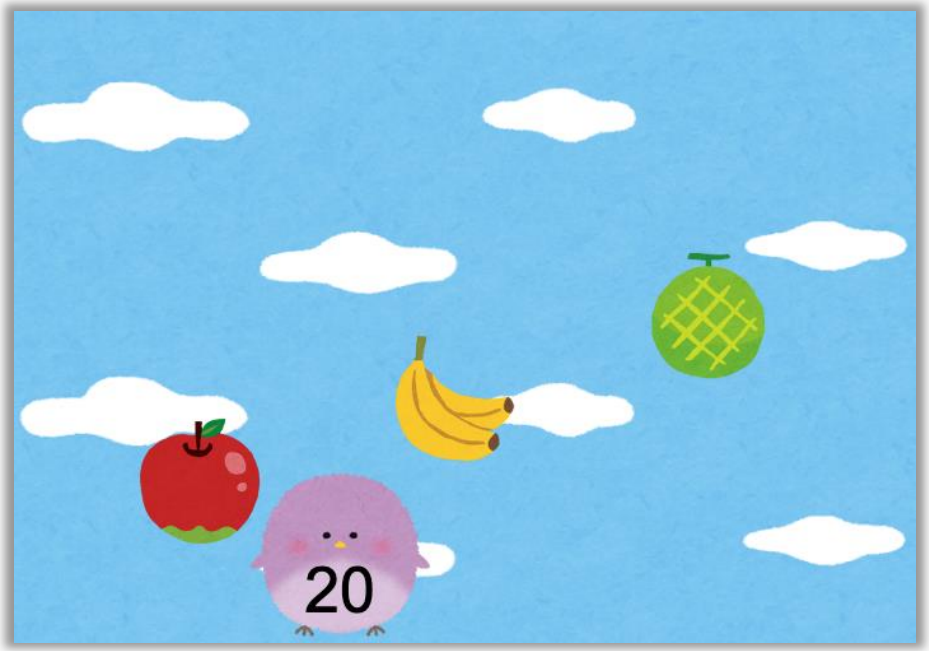
- **初等中等教育におけるプログラミング教育の必修化**
  - 多くの学生に指導する教員の負担
  - チュートリアルは自立学習が可能
- **現状**
  - フルーツキャッチゲームの作成のみ
  - 簡易的な実装により、新規作成が困難



# OCaml Blockly チュートリアルを活用する

- **初等中等教育におけるプログラミング教育の必修化**
  - 多くの学生に指導する教員の負担
  - チュートリアルは自立学習が可能
- **現状**
  - フルーツキャッチゲームの作成のみ
  - 簡易的な実装により、新規作成が困難

**目標: チュートリアル作成の簡易化**
**手法: DSLの開発**



# DSL開発

**分析**

- 現行のチュートリアルを理解
- チュートリアル作成における問題の特定
- DSLの設計（内部DSLと外部DSL）

**実装**

- ライブラリの構築（内部DSLの実装）
- 外部DSL用コンパイラの作成

**利用**

- 既存のチュートリアルの再現
- 評価、考察は未着手

# 1. 分析

- **現行のチュートリアルについて**
  - 実装について
  - 実行時の処理
- **チュートリアル作成の問題点**
- **DSLの設計**
  - 問題点を踏まえた設計のポイント
  - 提案する内部DSL
  - 外部DSLの仕様

# 現行のチュートリアルについて

- **実装**
  - JavaScript
  - ガイドの表示: Intro.js
    - JavaScriptのライブラリ
    - ウェブサイトのガイドツアーの作成
- **記述（図）**
  - 初期ブロックのOCaml コード
  - レコードのリスト（内容）
- **実行の流れ**
  1. 初期ブロックの処理
  2. チュートリアル開始

```
initcode[40] = "let add10 x =\n x + ?";
introlst[40] =
  [
    {
      "text": [
        [{ "intro": "関数add10を完成させましょう。まずは、四則演算
          ブロックの右側に整数ブロックをはめます。" },],
      ]
    },
    {
      "text": [
        [],
        [],
        [],
        [{ "intro": "値を10に変更します。整数ブロックをクリックして
          キーボードから入力します。" }],
        [],
      ],
      "category": 0,
      "block": 0,
      "id": 1,
      "target": [0, "B"],
      "value": ["10", "10"],
    },
  ]
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

デモのプログラム

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

ワークスペース上の
letブロックを取る

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

変数名が add10 の
ブロックを抽出

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

コネクタ名 EXP1 の
ブロックを得る

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

得られたブロックのidを
リストに格納

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# ブロックを表示 → blockIdリスト作成

- コードからブロックに変換（OCaml Blocklyの機能）
- blockId: ワークスペース上のブロックのid
  - ブロックを指定する時など必要 → リストで管理
  - チュートリアルごとに、プログラムを記述

デモのチュートリアルの番号は40

```
else if (n == 40) {
    letblock1 = Blockly.mainWorkspace.getBlocksByType("let_fun_pattern_typed", true);
    letblock2 = Blockly.mainWorkspace.getBlocksByType("letstatement_fun_pattern_typed", true);
    letblock = letblock1.concat(letblock2);
    add10 = letblock.filter(x => x.getField("VAR").getText() == "add10");
    plus = add10[0].getInputTargetBlock("EXP1");
    initidlst = [plus.id];
}
```

# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する

# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

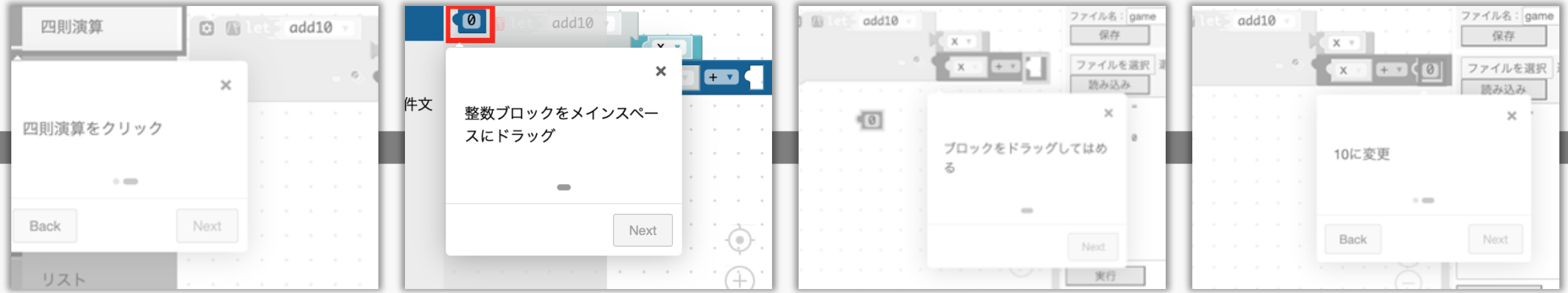
1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する




# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する



# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する




# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する

# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する




# レコードごとに一連の処理を適用

```
{
  "category": 0,
  "block": 0,
  "id": 1,
  "target": [0,"B"],
  "value": ["10","10"],
}
```

1. 四則演算メニュー(メニューの0番目)を開く
2. 整数ブロック(四則演算メニューの0番目)をメインスペースへ
   この時、blockIdをリストの1番目に追加する
3. 四則演算ブロック(idlstの0番目)の右側(コネクタB)に接続
4. 整数ブロックの値を10に変更する

# レコードのリストの記述が必要

1. **チュートリアルの操作ごとに独立していない**
   - 1つのレコードに複数の指示
   - 操作間の依存関係（例: 値の変更はブロックを取り出した直後）
2. **ブロックを指定する時に認知的負荷がかかる**
   - ワークスペース上のブロックを「blockIdリストの何番目か」で指定
   - リストの中身を把握する必要がある

# チュートリアル作成の問題を解決する

## レコードのリストの問題点

1. チュートリアルの操作ごとに独立していない
2. ブロックを指定する時に認知的負荷がかかる

## 設計

1. 操作ごとに独立したライブラリ関数を作成する
2. ワークスペース上のブロックに名前を与える
3. 初期ブロックに名前を与える

# チュートリアルの実行処理を行う

- **操作ごとに処理をまとめて関数化**
  - 直感的な関数名
- **操作間の依存関係を排除**

```
{
   "category": 0,
   "block": 0,
   "id": 1,
   "target": [0, "B"],
   "value": ["10", "10"],
},
```

- 四則演算メニューを開く
- メニューから整数ブロックを取り出し、四則演算ブロックの右側に接続
- 整数ブロックの値を10に変更する

```
openMenu("四則演算");
dragToTarget("NUM", "整数", ["PLUS", "RIGHT"]);
changeValue("NUM", 10);
```

# ブロック変数を導入

- **チュートリアル作成者が決める**
  - メニューからブロックを取り出すときに指定

```
{
   "category": 0,
   "block": 0,
   "id": 1,
   "target": [0, "B"],
   "value": ["10", "10"],
},
```

- 四則演算メニューを開く
- メニューから整数ブロックを取り出し、四則演算ブロックの右側に接続
- 整数ブロックの値を10に変更する

```
openMenu("四則演算");
dragToTarget("NUM", "整数", ["PLUS", "RIGHT"]);
changeValue("NUM", 10);
```

# ブロック変数の導入

- **チュートリアル作成者が決める**
  - メニューからブロックを取り出すときに指定
- **初期ブロックにブロック変数**

```
{
   "category": 0,
   "block": 0,
   "id": 1,
   "target": [0, "B"],
   "value": ["10", "10"],
},
```

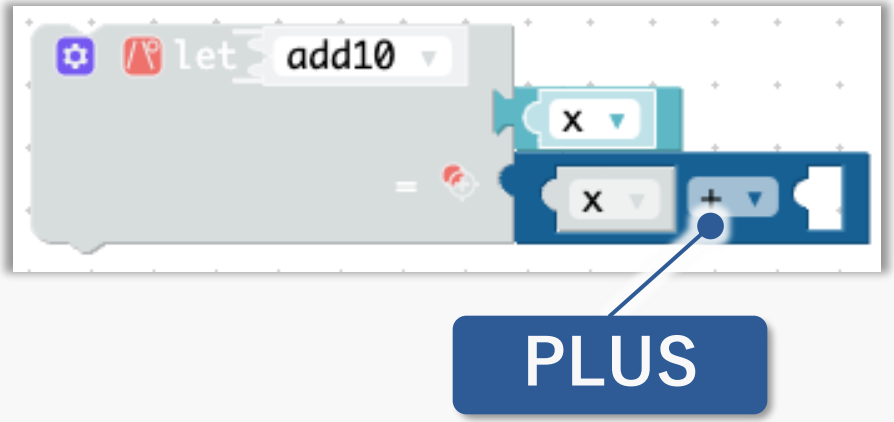
- 四則演算メニューを開く
- メニューから整数ブロックを取り出し、四則演算ブロックの右側に接続
- 整数ブロックの値を10に変更する

```
openMenu("四則演算");
dragToTarget("NUM", "整数", ["PLUS", "RIGHT"]);
changeValue("NUM", 10);
```

# コードにブロック変数を記述する

- **OCaml の Attributes**
  - 型チェックの影響を受けずに追加情報を付与
  - 書き方：式[@ブロック変数]
- **例：「関数add10の完成」のコード**
  - 式は丸括弧で囲む

```
let add10 x = (x + ?)[@PLUS]
```



# ライブラリ関数を順番に呼び出し

- **ホスト言語の機能を活用**
  - チュートリアルごとに1つの関数を定義
  - 初期ブロックのコードは配列に記述

**チュートリアル作成の問題点を解決**

```
initCode["デモ"] = "let add10 x = (x + ?)[@PLUS]";
async function demo() {
  literal("関数add10を完成させましょう。まずは、四則演算ブロックの右側に整数ブロックをはめます。");
  await openMenu("四則演算");
  await dragToTarget("NUM", "整数", ["PLUS", "RIGHT"]);
  literal("値を10に変更します。整数ブロックをクリックしてキーボードから入力します。");
  await changeValue("NUM", 10);
  complete();
}
```

# 自然言語に近い文法

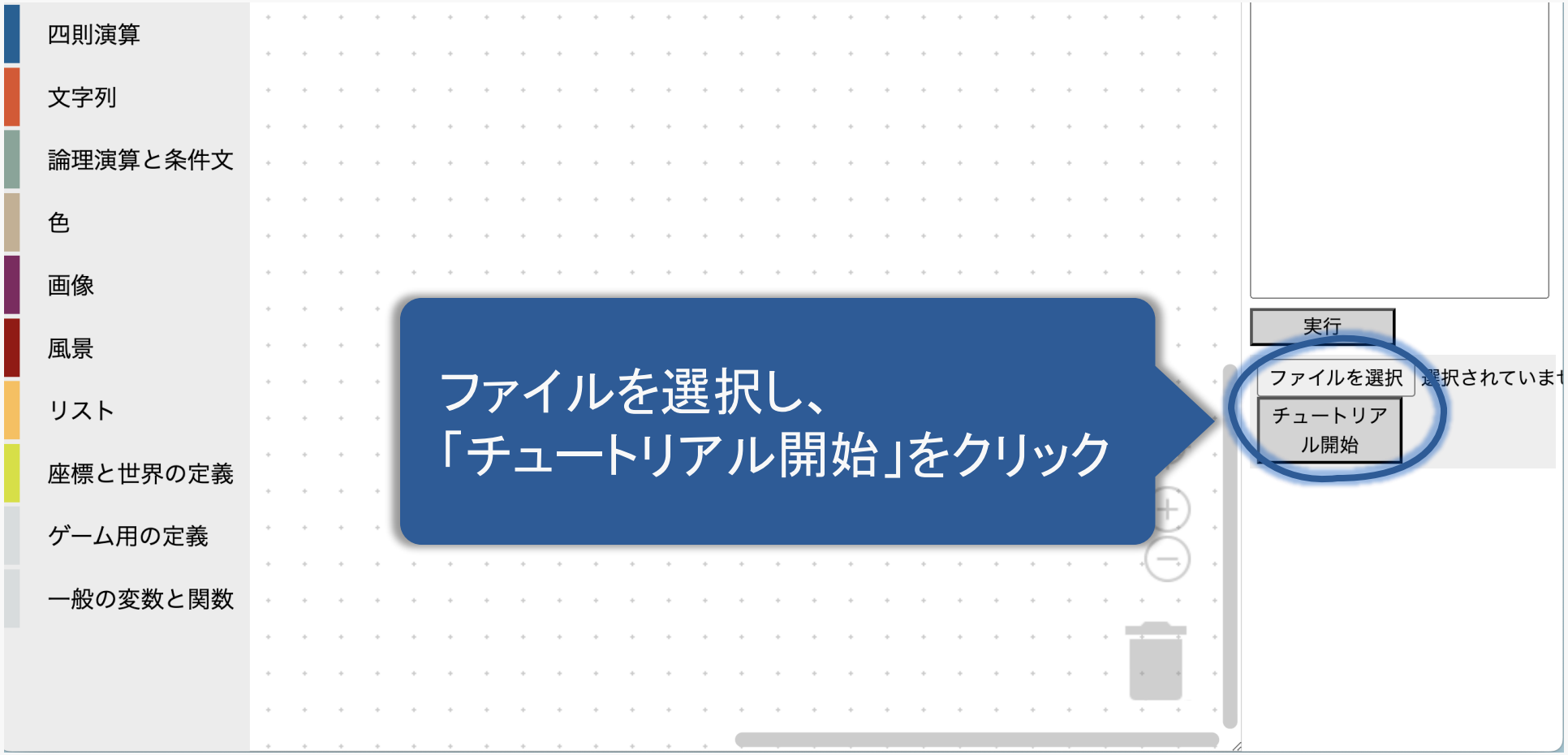
- **サイトにファイルを渡すだけで実行可能**
  - ファイルを1つ用意すればチュートリアルが作れる

```
%{
let add10 x = (x + ?)[@PLUS]
%}
literal [関数add10を完成させましょう。まずは、四則演算ブロックの
右側に整数ブロックをはめます。];
open 四則演算メニュー;
NUM = drag 整数 to (RIGHT of PLUS);
literal [値を10に変更します。整数ブロックをクリックしてキーボード
から入力します。];
change_value of NUM to 10;
complete;;
```

**初期ブロックのコード（ブロック変数付き）**

**チュートリアルの内容（ライブラリ関数と対応）**

# 簡単に導入・修正もスムーズ



# 2. 実装

- **内部DSLの実装: ライブラリの構築**
  - ライブラリ関数の同期処理
  - ブロック変数を用いたブロック環境
  - コードからブロック環境を構築
- **外部DSLの実装:コンパイラの作成**

# チュートリアルの操作ごとに独立した関数

- **逐次処理の実装**
  - async/await を使用
  - OCaml Blockly で正しい操作が行われたら、次のライブラリ関数へ

| | |
|---|---|
| literal | 画面中央にメッセージを表示 |
| openMenu | メニューを開く |
| dragToTarget | 新しいブロックを任意のブロックのコネクタに接続 |
| changeValue | ブロックの値を変更する |
| complete | チュートリアルクリア |

逐次処理

# ワークスペース上のブロックの管理方法を変更

旧

**リスト** `idlst[0];`

- blockIdを前から順番に、直接格納する
- 番号による指定

新

**ブロック環境** `blockEnv.get(blockVar).id`

- ブロック変数をキーとして、blockIdなどを管理（key-value）
- 名前（ブロック変数）による指定

# 内部DSLの実装③ 初期ブロックのブロック変数

## コードからブロック環境の作成

OCaml コード（ブロック変数付き）

▼ 初期ブロック用コンパイラ

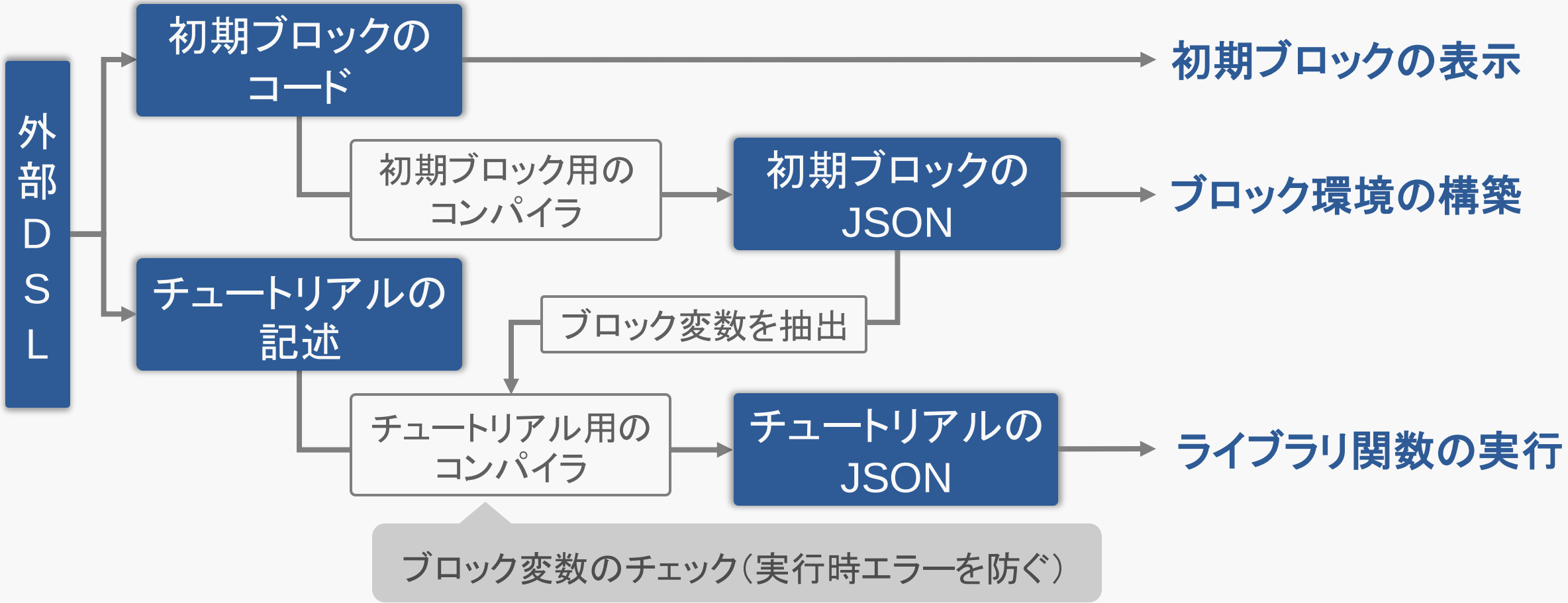
初期ブロックのJSON

▼ blockIdを取得し、環境に追加

ブロック環境の構築

```
{
  "blockName": "関数定義",
  "blockVar": "ADD10",
  "field": { "name": "VAR", "text": "add10" },
  "inputs": [
    {
      "name": "ARG0",
      "child": {
        "blockName": "変数パラメタ",
        "field": { "name": "VAR", "text": "x" }
      }
    },
    {
      "name": "EXP1",
      "child": {
        "blockName": "四則演算",
        "blockVar": "PLUS",
        "field": { "name": "OP_INT", "text": "ADD_INT" },
        "inputs": [
          {
            "name": "A",
            "child": {
              "blockName": "変数",
              "field": { "name": "VAR", "text": "x" }
            }
          },
          { "name": "B", "child": { "blockName": "?" } }
        ]
      }
    }
  ]
}
```

# DSL用のコンパイラ


外部DSL
初期ブロックのコード
初期ブロック用のコンパイラ
初期ブロックのJSON
初期ブロックの表示
ブロック環境の構築
チュートリアルの記述
ブロック変数を抽出
チュートリアル用のコンパイラ
チュートリアルのJSON
ライブラリ関数の実行
ブロック変数のチェック（実行時エラーを防ぐ）

# まとめ

- **成果と現状**
  - DSLを開発し、チュートリアル作成の簡易化を図った
  - 作成したDSLで既存のチュートリアル29個を再現
- **今後の展望**
  - エラーに対する制御機構
    - ユーザーが誤って操作した際のチュートリアル復帰
      - ライブラリ関数により誤操作の予測が容易に
    - 実行時エラーを未然に防ぐ
      - ブロック変数のチェックは実装済み
      - 例: ブロックを取り出す前にメニューを開いているか？
  - 作成したDSLの利用
    - 実際に使用して改善していく