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

tamtam@pllab.is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要　OCaml Blockly は，OCaml のビジュアルプログラミングエディタであり，直感的な操作によってプログラミングが可能である．また，画面上にガイドを表示することで，実際に動かしながら学べるチュートリアルが作成されている．チュートリアルは，初心者がツールに慣れるために不可欠であり，特に教育現場での活用を想定した場合，円滑に導入できる仕組みが求められる．しかし，現行の実装ではチュートリアルの作成方法が複雑であるため，活用が難しいという課題がある．本研究では，チュートリアル作成の簡易化を目指し，実装プログラムの改良やコンパイラの作成によって，チュートリアル作成のための領域特化型言語を作成する．これにより，プログラミングの知識がなくてもチュートリアルの作成が可能となり，教育現場で活用されることが期待される．

1　はじめに

OCaml Blockly [10] は，Google が提供するビジュアルプログラミングツール Blockly をベースとした，OCaml のビジュアルプログラミングエディタである．型システムや変数束縛を直感的なユーザインタフェースとして備えており，コンパイルエラーを起こすプログラムは組み立てられないようになっている．そのため，プログラミング初学者が陥りやすい本質的でない問題が起こりにくく，テキストベースで学ぶよりも簡単にプログラミングを学ぶことが可能である．

OCaml Blockly にはプログラミング未経験の大学生を対象にしたチュートリアルサイト [7, 8] が作成されている．サイトの内容は，全学部向けの授業をもとにしており，ブロックの操作方法を学びながら，ゲームプログラミングを学ぶことができる．図 1. のように，文章と画像による説明の後に，チュートリアルと練習問題が用意されていて，全て進めると図 2. の簡単なゲームが作れるようになる．チュートリアルや練習問題は，ボタンをクリックすると OCaml Blockly の画面に遷移し，実際に動かして学ぶことができる．

チュートリアルでは，後に詳しく説明する図 3 〜 7. のように，OCaml Blockly の画面に吹き出しやハイライトによるガイドを表示することで，次に行う操作を明確にしている．操作画面にガイドを表示するプログラミング教材として，Blockly Games [1] や Nintendo Switch のソフト [11] などがあり，このような形のプログラミング教材が年齢問わず，プログラミング初学者にとって取り組みやすい教材だと考えられる．

近年，小学校，中学校，高校の初等中等教育においてプログラミング教育が必修化され，多くの生徒がプログラミングに触れる機会を持つようになった．そのような状況では，チュートリアルのような取り組みやすい教材はますます重要になることが予想される．しかし，現在のチュートリアルの実装は，図 2. のチュートリアル専用となっている．特に，チュートリアルの制御構造が実装に埋め込まれており柔軟性が低く，また，チュートリアルのシナリオの記述もわかりやすいものとは言い難い．その結果，チュートリアルの新規作成や変更が非常に難しく，実質的に開発者以外がチュートリアルを作成するのは不可能である．

そこで，本研究では，OCaml Blockly のチュートリアル作成の簡易化を目指し，チュートリアルの作成に特化した領域特化型言語（以下， DSL）を開発する．それによって，プログラミングの知

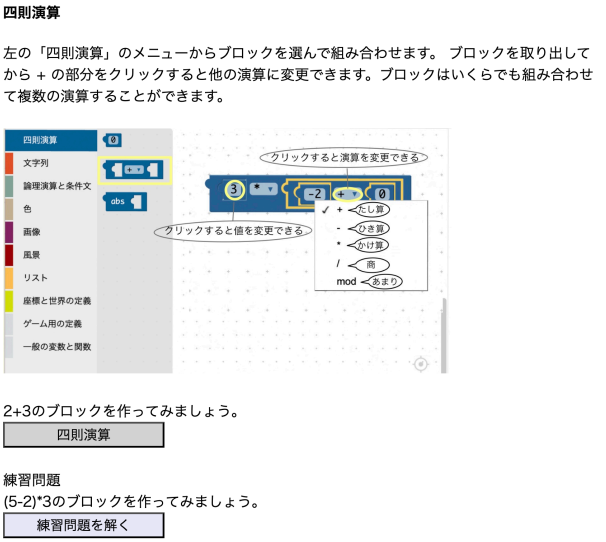


図 1. チュートリアルサイト

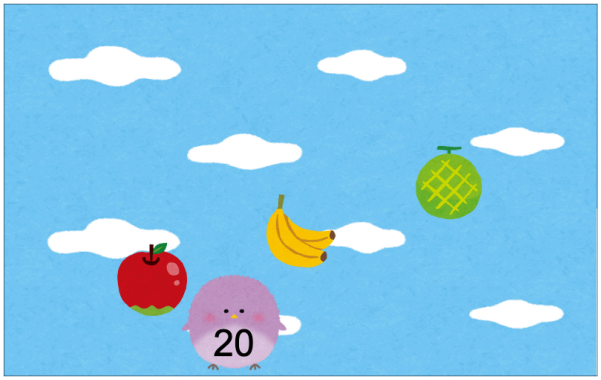


図 2. フルーツキャッチゲーム

識を持たない人でもチュートリアルの作成が可能となり，より広く教育現場でチュートリアルを活用できるようにすることを目指す．

一般的に， DSL の開発には，分析・実装・利用の 3 つのステップがある [5]．本研究では，チュートリアル作成という分野でどのような DSL が必要かを分析し，その最低限の語彙を特定する．その上でライブラリを使った内部 DSL と実装から独立した記述が可能な外部 DSL の両方を作成する．内部 DSL は，以前の実装と比べると十分にわかりやすいものとなっているが，チュートリアルの作成には実装を直接，変更する必要がある．一方，外部 DSL は完全に独立しており，プログラミングの知識がない人でも容易に記述できるものとなっている．

本研究は， DSL 開発において新たな知見を示すものではないが，チュートリアル作成という場面において実際に使用可能な DSL を実装することで， DSL 開発のひとつのケーススタディを与えるものとなっている．なお，本研究では，作成した DSL の利用についての考察は行っていない．現在，既存の 29 個のチュートリアル全てを，内部 DSL と外部 DSL の両方で再現できているが，本格的な利用については今後の課題とする．

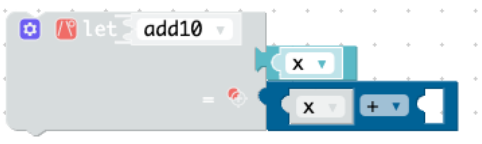
本論文の構成は次の通りである．まず，2 節で OCaml Blockly のチュートリアルについて例を挙げて紹介し，現行の実装の詳細とチュートリアル作成における課題を述べる．3 節で DSL の設計を示し，4 節で DSL 実装の準備，5 節で DSL の実装について述べる．6 節で関連研究について触れ，最後に 7 節で本研究の成果と現状，今後の課題について述べる．

2 OCaml Blockly チュートリアルの概要

DSL を開発する前に，現状を把握し，作成する DSL に必要な要素を抽出するため，まずは既存のチュートリアルを詳しく説明する．

2.1 チュートリアルの例

例として，四則演算ブロックの右側に 10 のブロックを接続して関数 add10 を完成させるチュートリアルを用いる．この例では，画面を読み込むと右のブロックが表示され，以下のチュートリアルが開始する．



1. 画面中央に「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。」という吹き出しが表示される．(図 3)
2. 四則演算メニューにハイライトが付き,「四則演算をクリック」という吹き出しが表示される．それに従い，四則演算メニューを開く．(図 4)
3. 整数ブロックに赤いハイライトが付き,「整数ブロックをメインワークスペースへ」というガイドが出る．整数ブロックをメインワークスペースにドラッグすると，四則演算ブロックの右側のコネクタに水色のハイライトが付き,「ブロックをドラッグしてはめる」という吹き出しが表示される．整数ブロックを四則演算ブロックの右側に接続する．(図 5)
4. 画面中央に「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。」という吹き出しが表示される．
5. 手順 3. で操作した整数ブロックに赤いハイライトが付き,「10 に変更」という吹き出しが表示される．ガイドに従って，ブロックの値を 10 に変更する．(図 6)
6. 画面中央に「チュートリアルクリア」という吹き出しが表示される． (図 7)

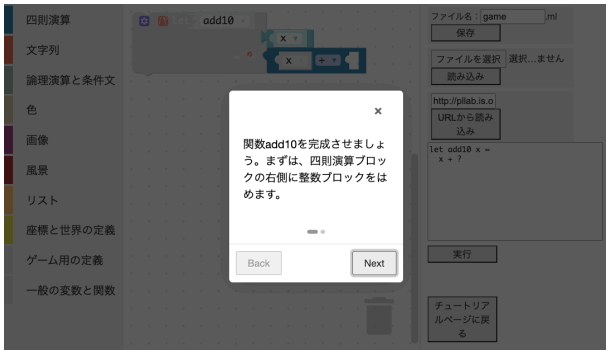


図 3. 画面中央に吹き出しを表示

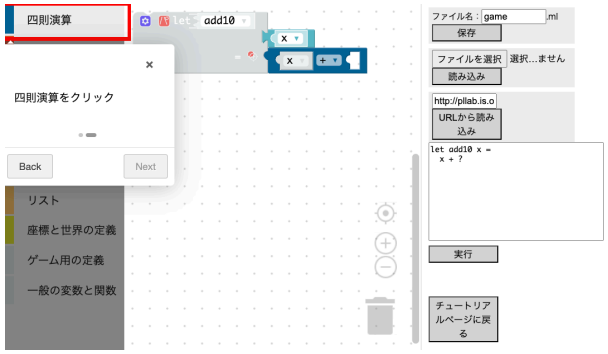


図 4. メニューを開く

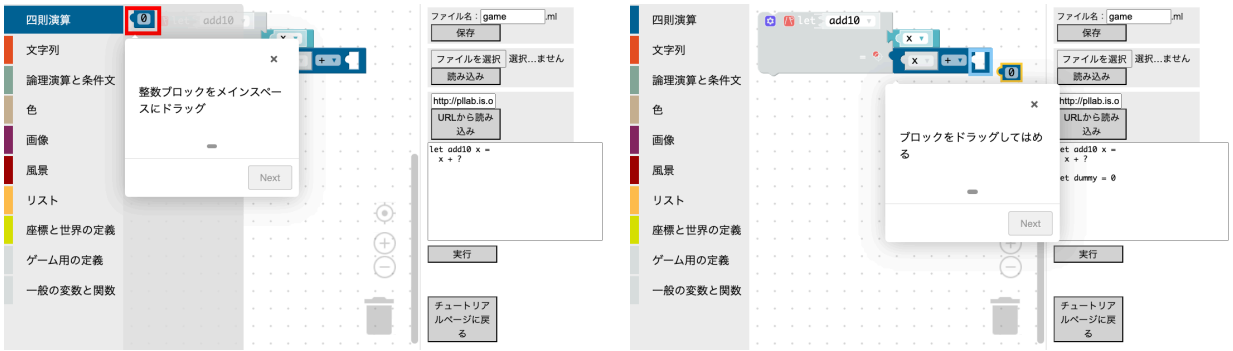


図 5. メニューからブロックをドラッグし，任意のコネクタに接続

2.2 現行の実装

主な使用言語は，JavaScript である．特に，吹き出しやハイライトの表示には，Intro.js [2] というライブラリを使用している．Intro.js を使うことで，吹き出しに表示するテキストやハイライトする要素を与えるだけで，簡単にステップ式チュートリアルが作成できる．チュートリアルの実行は 3 つのステップで行う．

図 6. 値を変更

図 7. チュートリアルクリア

初期ブロックの表示　チュートリアルで，画面を読み込んだ時に表示されるブロックを初期ブロックと呼ぶ．初期ブロックは，該当のチュートリアルで学ぶことにフォーカスするために使用される．例えば，前節の関数 add10 を完成させるチュートリアルでは，画面を読み込むと四則演算ブロックの右側が空いたブロックが表示される．そのため，整数ブロックをドラッグして四則演算ブロックに接続する操作だけに着目できる．OCaml Blockly には，コードを渡すとブロックへ変換し表示する関数があるので，チュートリアル開始前に，実装プログラム内に記述された初期ブロックのコードを関数に渡し，ブロックを表示する．

初期ブロックの blockId リストを作成　OCaml Blockly では，ワークスペース上の全てのブロックに固有の Id（blockId）を与えている．チュートリアルでブロックをハイライトする時などに blockId を用いるため，実行時に blockId をリストで管理している．チュートリアルで初期ブロックを使用するためには，blockId をリストに追加しておく必要があるので，チュートリアル開始前に blockId を取得してリストに追加する．2.1 節の例では，初期ブロックの四則演算ブロックをチュートリアルで使用するので，その blockId をリストに追加する．

チュートリアルを開始　チュートリアルの内容は，図 8. のようなレコードを並べたリスト形式で記述されている．実装プログラムには，ガイドを表示する関数（レコードのリストに対するインタプリタ）が作成されている．チュートリアル操作のいくつかについては，対応する関数が作成されているが，操作によっては他の操作の中に埋め込まれた形で実装されている．チュートリアルを開始すると，操作を表す各レコードに対して，対応する関数を実行する形でチュートリアルは進む．それぞれの関数では，対応するプロパティがあればガイドを表示し，なければ何もせずに次の関数に進む．

図 8. のようなレコードに対して行われる処理では，プロパティが重要である．各プロパティの意味は次の通りである．

text: Intro.js の Options というオブジェクトの書き方をする．図 3 のように，背景を暗くして画面の中央に表示したい文章をここに書く．

category: 操作するメニューを指定する．値は OCaml Blockly 画面左のメニューの何番目かを示す．四則演算メニューはメニューの最初にあるので，category の番号は 0 である．図 4 のようなガイドを表示する．

block: 操作対象のブロックを指定する．値はメニュー内のブロックの何番目かを示す．四則演算メニューでは，0 が整数ブロックである．図 5. の左のようなガイドを表示する．

id: 新たにメニューから出したブロックの blockId をリストの何番目に追加するかを表す．指定した番号にすでに要素が存在するとリストの上書きが発生するため，チュートリアル作成者が

```
1  [
2      {
3          "text": [
4              [{ "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。" }],
5          ]
6      },
7      {
8          "text": [
9              [],
10             [],
11             [],
12             [{ "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。" }],
13             [],
14         ],
15         "category": 0,
16         "block": 0,
17         "id": 1,
18         "target": [0, "B"],
19         "value": ["10", "10"],
20     },
21 ]
```

図 8. チュートリアルのリストの例 (関数 add10 の作成)

blockId リストを把握しておく必要がある．

target: ブロックの動かす先を指定することができる．1 つ目の値は blockId リストの番号で，2 つ目の値はコネクタ名を表す．ブロックを指定するために，blockId リストを把握しておく必要がある．コネクタ名は，OCaml Blockly で定められており，四則演算ブロックのコネクタ名は左が A で右が B である．図 5. の右のようなガイドが表示される．

value: 操作したブロックの値の変更を行う場合に書く．1 つ目の値は吹き出しに表示する文字列で，2 つ目の値は OCaml Blockly の実装での文字列である．例えば，四則演算ブロックの演算子は + がデフォルトだが，− に変更したい場合，`["-","MINUS_INT"]` と書く．図 6. のようなガイドが表示される．

これを踏まえて，図 8. の 15〜19 行目のプロパティは次のように処理される．

- `"category":0` より，四則演算メニューを開くガイドを表示する．
- `"block":0` より，前のステップで開いた四則演算メニューの 0 番目である整数ブロックをワークスペースにドラッグするガイドを表示する．
- `"id":1` より，ドラッグした整数ブロックを blockId リストの 1 番目に追加する．
- `"target":[0,"B"]` より，blockId リストの 0 番目である四則演算ブロックのコネクタ B（右側）に，メニューから出した整数ブロックを接続するガイドを表示する．
- `"value":["10","10"]` より，整数ブロックの値を 10 に変更するガイドを表示する．

チュートリアルの作成　チュートリアルの作成には，その内容を記述したレコードのリストに加えて，初期ブロックを表示する場合は OCaml コードと blockId リストが必要になる．これらはいずれも実装プログラムに記述する．現行の実装は，チュートリアルを新たに作成することは想定していないため，上記の情報を指定された配列に格納している．その上で，チュートリアル開始時に指定したチュートリアル番号に従って対応する情報を取り出してチュートリアルを実行している．

2.3 現行のチュートリアル作成の問題点

現行の実装も，チュートリアルの内容をレコードに記述して，それを実行しているため，レコードの部分だけを理解すればチュートリアルを作成できると考えられる．しかし，実際には以下のような問題があり，チュートリアルの作成が決して容易とは言えないのが現状である．

チュートリアルの操作ごとに独立していない　チュートリアルの動作は，レコード内のプロパティの有無によって変わるようになっている．例えば，value というプロパティは値の変更をしたいときに記述する．しかし，これが有効なのはメニューからブロックを取り出したときのみになっている．新たにブロックを取り出し，その値を変更することはできるが，すでに存在しているブロックの値を変更することは不可能である．このように，レコードの記述がチュートリアルの操作ごとに独立していないため，各操作の間の依存関係を熟知していないとチュートリアルを記述することはできなくなっている．

ワークスペース上のブロックを blockId リストの番号で指定する必要がある　ワークスペース上のブロックの指定は，全て blockId リストの何番目のブロックかによって指定する．したがって，チュートリアルのレコードを記述するためには，blockId リストの中にどのようなブロックがどのような順番で入っているのかを常に把握していなくてはならない．原理的には，番号が一意に与えられていれば常にブロックを指定できるが，実際にチュートリアルを記述する際に番号を管理するのは，かなりの認知的な負荷がかかる．

本研究では，これらの問題を解決するために， DSL を作成する．

3 DSL 設計

この節では，現行のチュートリアルを分析して，必要な項目を整理し，最終的に作成する DSL を設計する．

3.1 チュートリアル操作の抽出

現行のチュートリアルを精査すると，チュートリアルの中で現れる操作は以下のようになる．

- 画面中央にメッセージを表示する
- 新しいブロックをメインワークスペースにドラッグする
- 新しいブロックを任意のブロックのコネクタに接続する
- 新しい引数ブロックを任意のブロックのコネクタに接続する
- レコードの作成にて，型ブロックを任意のフィールドに接続する
- ワークスペース上の変数や関数を使用する
- ブロックをゴミ箱に入れる
- メニューを開く
- 「パ」ボタンで，引数ブロックのダイアログを開く [閉じる]
- 「型」ボタンで，型ブロックのダイアログを開く [閉じる]
- 歯車ボタンで，Mutator を開く [閉じる]
- ブロックの値を変更する
- Mutator の要素数を変更する

- 世界の要素を変更する
- 変数名や関数名，レコード名を変更する
- レコードのフィールド名を変更する
- 実行ボタンを押す
- チュートリアルクリア

3.2 言語仕様の設計

DSL の設計には，汎用言語 (GPL) の機能を活用し，DSL をその言語内に埋め込む形で実装する内部 DSL と，パーサーやコンパイラを設計し実装する外部 DSL の 2 種類があり，その割合はほぼ等しい [4]．本研究では，まず実装プログラムに直接記述する内部 DSL を作成し，その後実装から独立した外部 DSL を作成する．

共通の設計　内部 DSL と外部 DSL に共通する設計として，次の 2 つがある．1 つ目はメニューやブロックの指定，2 つ目はライブラリ関数である．

メニューやブロックは番号ではなく名前で指定する．メニューの指定は，OCaml Blockly のメニューと同じものを用いる．また，ブロックの指定は，メニュー上のブロックとワークスペース上のブロックで異なるものを用いる．メニュー上のブロックには，実装プログラム内で名前を定める．この名前は，吹き出しで表示するものと同じである．ワークスペース上のブロックには，チュートリアルの記述内で名前を与え，これを「ブロック変数」と呼ぶ．ブロック変数は，アルファベットの大文字とアンダースコアで構成される．

このように，番号ではなく名前を使うことで，ブロックを指定する際の負荷を軽減する．特に，メニュー番号の代わりにメニュー名そのものを用い，チュートリアルの途中で現れるブロックにはユーザーが指定する名前を用いるようにする．

また，チュートリアルの実行処理を行うためのライブラリ関数を用意する．この関数は，表 1 のように，3.1 節で抽出した項目ごとに作成する．これにより，チュートリアルごとに独立した記述を可能にする．特に，各操作の間の依存関係は排除し，好きな順番で好きな操作を記述できるようにする．

内部 DSL の設計　内部 DSL は，ライブラリ関数を順番に呼び出す形で記述し，実装プログラムに直接組み込む．図 9. の `add10` のように，チュートリアルごとに 1 つの関数を定義し，その中にライブラリ関数を順に呼び出す形で記述する．そして，この関数をチュートリアル開始時に呼び出すようにする．

図 9. は TypeScript の関数になっているため，チュートリアルを追加・変更するには実装に手を加える必要がある．しかし，図 9. の内容は単にチュートリアルの操作を順に並べただけであり，ライブラリ関数の実装まで知る必要はない．なお，初期ブロックが必要な場合は，初期ブロックのコードを管理しているオブジェクトに記述しておく．

チュートリアルの指定は，チュートリアル名を用いて行う．例えば，チュートリアル名が `add10` であれば，`getTutorial["add10"]` でチュートリアルの関数，`initCode["add10"]` で初期コードを取り出すことができる．

外部 DSL の設計　内部 DSL では実装に直接，手を加える必要があったが，外部 DSL では必要な情報を 1 つのファイルにまとめ，そこに全てを記述するようにする．初期コードとチュートリアルの内容を，図 10. のように記述し，それをチュートリアルのページで読み込むことでチュートリアルを

ライブラリ関数	提供するガイドの内容
literal	画面の中央に吹き出しを表示する
dragToMain	新しいブロックをメインワークスペースにドラッグする
dragToTarget	新しいブロックを任意のブロックのコネクタに接続する
dragParam	新しい引数ブロックを任意のブロックのコネクタに接続する
dragTypeBlock	レコードの作成にて，型ブロックを任意のフィールドに接続する
useVariable	ワークスペース上の変数や関数を使用する
trash	ブロックをゴミ箱に入れる
openMenu	メニューを開く
open[close]ParamWin	「パ」ボタンで，引数ブロックのダイアログを開く [閉じる]
open[close]TypeWin	「型」ボタンで，型ブロックのダイアログを開く [閉じる]
open[close]Mutator	歯車ボタンで，Mutator を開く [閉じる]
renameVariable	変数名や関数名，レコード名を変更する
renameRecordField	レコードのフィールド名を変更する
changeValue	ブロックの値を変更する
changeMutator	Mutator の要素数を変更する
changeWorldItem	世界の要素を変更する
execute	実行ボタンを押す
complete	「チュートリアルクリア」の吹き出しを表示

表 1. ライブラリ関数の一覧

```
async function add10() {
  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();
}
```

図 9. 内部 DSL (関数 add10 の完成)

```
%{
(* 初期ブロックのOCamlコード *)
let[@ADD10] add10 x = (x + ?)[@PLUS]
%}
(* チュートリアルの内容 *)
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;;
```

図 10. 外部 DSL（関数 add10 の完成）

```
<チュートリアル>  :=  <ステップ>;; | <ステップ>;<チュートリアル>
<ステップ>        :=  literal <文>
                  |   <ブロック変数> = drag <ブロック名> to <接続先>
                  |   open <メニュー>
                  |   chage_value of <ブロック変数> to <値>
                  |   ...
<文>              :=  [ <文字列> ]
<ブロック名>      :=  整数 | 四則演算 | ...
<ブロック変数>    :=  <大文字と_(アンダースコア)>
<接続先>          :=  ( <コネクタ名> of <ブロック変数> )
<メニュー>        :=  四則演算メニュー  | ...
<値>              :=  数字 | <文>
```

図 11. チュートリアル内容の構文定義（一部）

開始する．初期ブロックは，先頭の %{ と%} の中に記述され，ブロック変数は OCaml の Attributes として記述される．チュートリアル内容は，図 11. の構文定義に従って記述される．

<ステップ>は，表 1. のライブラリ関数ごとに構文を定義している．プログラミング経験が少ない人でも容易に記述できるように，自然言語に近い文法にする．

設計の有用性　ライブラリ関数の作成により，チュートリアルごとに独立した記述が可能となった．また，ワークスペース上のブロックにブロック変数を与え，ブロック変数を用いて指定できるようになり，blockId リストを把握しなければならないという認知負荷が軽減されている．

4　実装の準備

DSL 実装の準備として，円滑に開発を行うため，実装プログラムの可読性や保守性を向上させる．

4.1　TypeScript と型定義

使用言語を JavaScript から TypeScript に変更し，型システムを活用してコードの明確さと可読性の向上を図った．TypeScript の型推論を正確に使用するには，どんな型でも代入を許す型，any 型を使わずに型付けする必要がある．そこで，Intro.js については公式の型定義ファイルを使用し，OCaml Blockly はチュートリアル実装に必要な範囲の型定義ファイルを作成した．明示的な型付けをすることで，プログラムの可読性と保守性を向上させることができた．

4.2　OCaml Blockly のメニューにあるブロックの管理方法

チュートリアルの DSL を実装するには，OCaml Blockly の各種のブロックの情報を管理する必要がある．現行の実装では，OCaml Blockly のブロックを図 12. のようなリストで管理していた．ブロックリストでは，OCaml Blockly のブロックの型 (`int_typed` 等) と吹き出しに表示するブロック名，フィールド名を管理している．フィールドは，各ブロックに保持されている値，例えば整数ブロックならその値（整数），四則演算ブロックなら選択されている演算子（+ や − など）を表している．

図 12. のようなリストで管理すると，要素を参照する際，常に何番目のブロックであるかを意識しなくてはならない．そこで，DSL において番号ではなく名前を使用してブロックを参照するよう

```
blocklst[0] = [
    ["int_typed", "整数", "INT"],
    ["int_arithmetic_typed", "四則演算", "OP_INT"],
    ["int_abs_typed", "abs"],
    ["random_int_typed", "乱数"]
]
```

図 12. ブロックリスト (四則演算メニューのみ抜粋)

にしたのと同様にして，実装でも番号ではなく名前を与えるようにした．ブロックのオブジェクトを定義し，各種情報に名前でアクセスできるようにした．例えば，図 13. の四則演算ブロックのオブジェクトは，図 14. のように表現される．

図 13. 四則演算ブロック

```
"四則演算": {
  index: 1,
  type: "int_arithmetic_typed",
  field: "OP_INT",
  input: {
    LEFT: "A",
    RIGHT: "B",
  }
},
```

図 14. ブロックオブジェクトの例

この変更は，単にデータの表現方法を変更しただけだが，実装の可読性は格段に高くなる．オブジェクトが管理するブロックの情報は次の通りである．

index: メニュー内で何番目か．メニュー上のブロックにハイライトをつける際に必要となる．

type: OCaml Blockly でのブロックの型．選んだブロックが正しいか判断する際に使う．

field: フィールド名．値を変更する場合にフィールド名が必要となる．

input: コネクタの情報．これは本実装で新たに加えた要素である．チュートリアル記述用の直感的なコネクタ名と実装プログラムで使用する OCaml Blockly でのコネクタ名の対応を管理する．四則演算ブロックの場合，OCaml Blockly のコネクタ名は左が A で右が B となっている．チュートリアル記述では，LEFT，RIGHT という直感的なコネクタ名を使用したいので，LEFT は A，RIGHT は B という対応を表している．

ここで，実装プログラムでブロック情報を参照するコードを比較する (表 2) ．ブロックリストでは，category と block のプロパティ値を用いてブロックを指定し，3 つ目の要素が 0 は型, 1 は名前, 2 はフィールド名を参照する．一方，ブロックオブジェクトでは，ブロックの名前を key として得たオブジェクトにて，ドット記法でプロパティにアクセスするため，参照先が明確である．

参照する情報	ブロックリスト (図 12)	ブロックオブジェクト（図 14）
メニューの何番目か	`block`	`getBlockObj[blockName].index`
ブロックの型	`blocklst[category][block][0]`	`getBlockObj[blockName].type`
ブロック名	`blocklst[category][block][1]`	`blockName`
フィールド名	`blocklst[category][block][2]`	`getBlockObj[blockName].field`

表 2. ブロック情報を参照するコード (`category`，`block` は整数, `blockName` は文字列である)

5　DSL の実装

内部 DSL と外部 DSL それぞれの実装について示す．実装プログラムの改良によって，ライブラリ関数を作成し，内部 DSL の実装を行う．その後，外部 DSL のためのコンパイラの作成を行う．

5.1　内部 DSL

内部 DSL の実装は，ライブラリ関数を作成すれば実現する．この節では，実際にチュートリアルの実行処理を行うライブラリ関数の作成やブロック変数の導入について詳細を示す．

5.1.1　チュートリアルの操作ごとに独立した関数

3.1 節で抽出したチュートリアルの操作ごとに，独立したライブラリ関数（表 1）を実装した．これにより，チュートリアル記述の可読性が大きく向上した．加えて，これはチュートリアルの実装の整理にもなった．特に，これまで複雑に絡み合っていた関数を独立させることで，関数間の依存関係を解消することができた．

これらのライブラリ関数を時系列順に呼び出すと，OCaml Blockly での操作を待たずに先に進んでしまう．正しい操作をしてから次の関数に進みたいため，async/await を使って同期的な処理を行えるようにした．通常，async/await は非同期処理を扱うための仕組みだが，適切に使用することで処理を 1 つずつ順番に実行することが可能である．await を使うと，その処理が完了するまで次の処理には進まないため，あたかも同期処理のように動作する．

表 1. のうち，OCaml Blockly で操作するガイドを表示する関数（literal と complete 以外）には async/await を用いて，正しい操作をした後に次のライブラリ関数の処理に進むようにした．これにより，OCaml Blockly での操作を待つことができるようになった．

5.1.2　ブロック変数とブロック環境の導入

これまで，blockId のリストの番号を指定することでハイライトするブロックを指定していたが，リストの何番目がどのブロックの blockId か，というのを覚えておくのは不便である．そこで，ブロック変数という形でワークスペース上のブロックに名前を与え，ブロック環境でブロック変数とそのブロックの情報の対応を管理する．ここでのブロックの情報は，blockId と 4.2 節で述べたブロックオブジェクトである．新しいブロックをワークスペースに出すライブラリ関数では，引数としてブロック変数を受け取り，ブロック情報とともに環境に追加する．

例えば，図 9. の 4 行目 `dragToTarget("NUM","整数",["PLUS","RIGHT"])` では，ドラッグした整数ブロックに `NUM` というブロック変数を与えている．この時，ブロック環境には, `NUM` を key として，操作した整数ブロックの blockId と `getBlockObj["整数"]` で得られたオブジェクトを追加する．

また，初期ブロックの四則演算ブロックに `PLUS` というブロック変数を与えることで， 接続先を `["PLUS","RIGHT"]` と書くことができる．従来のチュートリアル記述 (図 8) で四則演算ブロックの右側を指すときは，18 行目のように `[0,"B"]` と記述している．これは，四則演算ブロックの blockId がリストの 0 番目であること，四則演算ブロックの右側のコネクタは B であることを把握している必要があった．しかし本研究では，リストの何番目かを覚える必要はなく，チュートリアル作成者がブロック変数を決めることができる．また，4.2 節のブロックオブジェクトによってコネクタ名も直感的なものになった．

図 15. のように，OCaml Blockly のメニュー画面で既にブロックが接続されている時は，全てのブロックに対するブロック変数をライブラリ関数の引数で指定する．このように，全てのブロック変数をチュートリアル作成者が決められるようにすることで，ワークスペース上のブロックを体系的に管理することが可能になった．

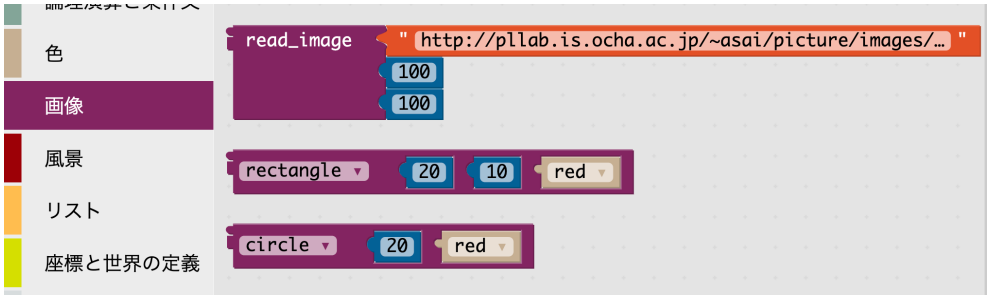


図 15. 最初からブロックが接続されている例

5.1.3 初期コードからブロック変数の取得

ワークスペース上のブロック管理を，ブロック変数とブロック環境に変更したため，2.2 節で述べた，初期ブロックに対して blockId リストを作成する部分も変更が必要である．そこで，初期ブロックにブロック変数を与える仕組みを実装した．初期ブロックを表示する場合は，チュートリアル開始前に OCaml のコードをブロックに変換して表示しているので，OCaml コードに Attributes の形でブロック変数を書くようにした．OCaml の Attributes は，コードに型チェックの影響を受けずに追加情報を付与する機能であり，[@と] で囲われた中に記述する．これを参考に，OCaml の文法に従ってブロック変数の書き方を定めた．例えば，図 16. のような変数 score を定義するブロックで，グレーの変数定義ブロックに「SCORE」，青の整数ブロックに「NUM」というブロック変数をつけるには，図 17. のような書き方をする．

図 16. 変数定義ブロックと整数ブロック

```
let[@SCORE] score = (100)[@NUM]
```

図 17. ブロック変数の書き方

そして，ブロック変数付きの OCaml コードを抽象構文木にした後，木をトラバースして JSON 形式で出力するコンパイラを作成した．チュートリアル実装で必要な情報を持つレコード型 block_t (図 18) を定義し，抽象構文木からレコードに変換した後，JSON 形式で出力する．コンパイラは，OCaml で作成しており，JSON 形式への変換は Yojson [3] というライブラリを使用している．Yojson は，OCaml で JSON の処理を行うためのライブラリで，リスト，レコード，オプション型などの OCaml のデータ型を簡単に JSON 形式へ変換できる．作成したコンパイラで，図 17. をコンパイルして出力すると，図 19. が得られる．

OCaml コードから JSON を得られたら，TypeScript プログラムで JSON の処理を行う．チュートリアル開始前に，初期ブロックのブロック変数をブロック環境に全て追加することで，その後のチュートリアルで初期ブロックの使用が可能になる．

5.1.4 スコープお砂場に対応

OCaml Blockly の機能の 1 つであるスコープお砂場では，そこで定義されているブロックを使うことができる．図 20. のように，既にブロックが接続されているコネクタに新たなブロックを接続すると，元のブロックがスコープお砂場に押し出される．（中の自由変数がスコープ外に出るのを防ぐため.）現行のチュートリアルでは，スコープお砂場に押し出されたブロックを指定する方法がないため，そのブロックを使ったチュートリアルを続行することができなくなってしまう．

スコープお砂場にチュートリアルを対応させるために，既存のライブラリ関数に機能を追加し，新たなライブラリ関数を作成した．まず，ブロックを接続する操作があるライブラリ関数では，ブ

```
type block_t = {
  blockName: string;
  blockVar: string option;
  field: field_t option;
  inputs: (input_t list) option;
  next : block_t option;
}
and field_t = {
  name: string;
  text: string;
}
and input_t = {
  name: string;
  child: block_t;
}
```

図 18. block_t 型

```
{
  "blockName": "変数定義",
  "blockVar": "SCORE",
  "field": {
    "name": "VAR",
    "text": "score"
  },
  "inputs": [
    {
      "name": "EXP1",
      "child": {
        "blockName": "整数",
        "blockVar": "NUM",
        "field": {
          "name": "INT",
          "text": "100"
        }
      }
    }
  ]
}
```

図 19. JSON 形式の出力

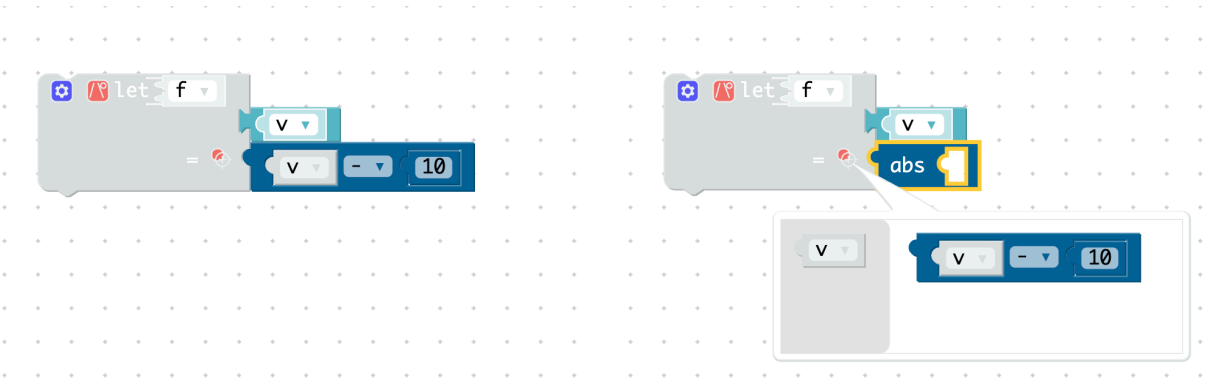


図 20. ブロックがスコープお砂場に移る

ロックがスコープお砂場に移動した場合に，移動したブロックの blockId をとってきてブロック環境に追加するように変更した．ブロック環境に追加する際にブロック変数が必要なので，関数の引数でもらってくる．そして，スコープお砂場に押し出されたブロックをチュートリアルで使うために，スコープお砂場からブロックを移動させるライブラリ関数 dragFromScope とスコープお砂場の開閉を行うライブラリ関数 open[close]ScopeWin を作成した．これにより，ブロックがスコープお砂場に押し出されることで使用不可となっていたチュートリアルを修正することができた．

5.2 外部 DSL

5.1 節で内部 DSL を作成したが，内部 DSL でチュートリアルを記述するにはどうしても実装に手を加える必要がある．ここでは外部 DSL を JSON 形式に変換し，それを読み込んでチュートリアルを実行できるようにする．このようにすると，チュートリアル作成者は実装を変更することなく，外部 DSL で記述するだけでチュートリアルを実行できるようになる．

外部 DSL で書かれたチュートリアル記述は，最終的に (1) 初期ブロックのコード，(2) 初期ブロックの JSON，(3) チュートリアルの JSON の 3 つにコンパイルされる．このうち (1) は OCaml Blockly にそのまま渡されてブロックに変換される．一方 (2) はブロック変数を抽出して blockId の環境を作るのに使われる．最後の (3) がチュートリアル本体で，この記述に沿って対応するライ

ブラリ関数を順番に呼ぶようにする．

外部 DSL のコンパイラの実装では，まず ocamllex と ocamlyacc を用いて字句解析と構文解析を行い，初期ブロックのコードとチュートリアル部分に分ける．前者は，初期ブロックのコード（1）として使用するとともに，5.1.3 節で作成した初期ブロック用のコンパイラに渡して，初期ブロックの JSON（2）を得る．そこから，チュートリアルの変数チェックに使用するため，ブロック変数を抽出しリストにする．後者は，チュートリアルの JSON（3）に変換する．その際，指定されたブロック変数が定義されている変数かどうかをチェックし，未定義のブロック変数が使われていた場合は，エラーを報告する．また，新しくブロック変数を付与する際に，同じ変数が既に使われている場合も，ブロック環境が上書きされてしまうため，エラーを報告する．

6　関連研究

本研究は， DSL を設計することで，ソースプログラムに埋め込まれてしまってる機能を誰でも利用できるようにする試みのひとつである．本研究では，OCaml Blockly のチュートリアルという特定のシステムを対象として，チュートリアルの作成をチュートリアルの実装から切り離すことで，容易にチュートリアルを作成できるようにした．本研究のように，プログラミング教材を作成するための DSL は見つけることができなかったが，自然言語を生かしたプログラミングを行う研究としては，高野ら [6] の研究がある．これは日本語で記述できる疑似コードを提案するもので，理解のしやすさとプログラミングの習得を両立させようというものである．

また，松島ら [9] はゲームのルールを宣言的に記述するだけで，ゲームが動くようなシステムを提示している．複雑なゲームのルールも，単純な小さなルールの組み合わせでできているという洞察に基づき，各ルールを特定の形で書くことで，そこから直接，ルールに対応する関数を導出している．一方，本研究におけるチュートリアルの操作は，ゲームのルールほど複雑ではない．そのため，あらかじめ各操作に対応する関数を用意しておくことで，比較的，簡単にチュートリアルを記述できるようになっている．

7　まとめ

7.1　本研究の成果と現状

本研究では，OCaml Blockly のチュートリアル作成の簡易化を目指し，DSL を作成した．現行のチュートリアルの作成方法では,（1）チュートリアルの操作ごとに独立していない（2）ワークスペース上のブロックを blockId リストの番号で指定する必要がある，という 2 つの問題があった．これに対して，内部 DSL の実装として実装プログラムの改良を行った．ライブラリ関数の作成により（1）を解決し，ブロック変数を導入することで（2）を解決できた．2 つの問題が解消したことで，チュートリアルの流れに沿った直感的な記述が可能になったが，内部 DSL は実装プログラムに直接組み込む必要がある．そこで，内部 DSL の利点をそのまま保ちつつ，チュートリアル記述を実装プログラムから独立させ，チュートリアル作成に必要な情報をまとめて記述できる外部 DSL を作成した．外部 DSL では，新たな制御機構の追加が可能となっており，次の段階の改良を考えられ，更なる機能向上が期待できる．また，外部 DSL を記述したファイルを専用のサイトで読み込むだけでチュートリアルが実行できるため，教育現場での活用も容易になったと考えられる．

現行のチュートリアルサイトには 29 個のチュートリアルがある．現在これらの全てを，図 9. の内部 DSL と図 10. の外部 DSL の両方で再現できている．このことから，チュートリアルサイトの再現に必要なライブラリ関数は, 表 1. と 5.1.4 節で作成した dragFromScope, open[close]ScopeWin で全て実装できており，作成したコンパイラが期待通りに動いていることがわかる．

7.2 今後の課題

7.2.1 ユーザーが想定外の操作をした時の処理

想定外の操作というのは，ガイドと違うブロックを選んでしまった，値の変更の際に指定されたものと異なる値に変更してしまった，などである．現行のチュートリアルでこれらの間違った操作をした場合，原則最初からになってしまう．Intro.js でガイドが表示されていても，OCaml Blockly の画面は自由に操作可能なため，想定外の操作を予測するのは非常に困難である．よって，誤った操作の場合は最初に戻る，というのは理にかなっている．しかし，長いチュートリアルの場合は特に，最初に戻って既知の操作を再度行うのは効率が悪い．これを解決するために，適切なタイミングでチュートリアルに復帰できる仕組みが必要である．内部 DSL の実装にて作成したライブラリ関数は，チュートリアルの操作ごとに独立している．ライブラリ関数ごとに該当の操作に対して，起こりうる誤操作を精査し，それぞれの場合で処理の方法を考える．内部 DSL を作成してチュートリアルの制御構造が明らかになったことで，誤動作時に例外処理などの機能によって対応する可能性を検討できるようになった．

7.2.2 チュートリアルの矛盾を出力

チュートリアルの矛盾は，メニューからブロックを取り出す時にメニューが開いていない，つまり，dragToMain 関数の前に openMenu 関数がない，などである．ライブラリ関数によって，チュートリアルの順序が柔軟になったため，このような矛盾が起こり得る．メニューが開いていなかった場合，実装プログラムでブロックが見つからないというエラーが起き，OCaml Blockly 画面のコンソールに出力される．これをポップアップダイアログで表示することで，チュートリアルの誤作動の原因を明らかにする．

矛盾を検知するタイミングは,「ライブラリ関数の実行時」と「外部 DSL のコンパイル時」の 2 つがある．前者は，その矛盾が起こるまではチュートリアルが進んでしまう．ライブラリ関数を順に呼び出す形の内部 DSL では，チュートリアルが開始後に矛盾していたことがわかる．一方で，後者では矛盾を検知したらコンパイルエラーとすることで，チュートリアル開始後のエラー発生を減らせる．このような実行時エラーを防ぐ機能は，外部 DSL に対して各種の解析を行うことで対応できるようになっていくと期待している．

7.2.3 DSL の利用

作成した内部 DSL と 外部 DSL を用いて新たなチュートリアル作成を行い，追加すべきライブラリ関数の確認や記述性，表現力を確認する．現在研究室では，中学生向けに一次関数を学ぶサイトを作成している．このようなサイト向けのチュートリアルを作成することも視野に入れている．こうした経験を積み上げていくことで，さらに使いやすいものを目指していく．

謝辞

多くの有益なコメントをくださった査読者の皆様に感謝申し上げます．本研究は一部 JSPS 科研費 20K12107 の助成を受けたものです．

参考文献

[1] Blockly Games. `https://blockly.games/`.

[2] Intro.js. `https://introjs.com/`.

[3] Yojson. `https://ocaml-community.github.io/yojson/yojson/Yojson/index.html`.

[4] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, Vol. 71, pp. 77–91, 2016.

[5] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, Vol. 35, No. 6, pp. 26–36, 2000.

[6] 高野志歩, 田村みゆ, 富岡真由, 秋信有花, 倉光君郎ほか. 擬似コードから考える自然言語を活かしたプログラミング言語. 情報教育シンポジウム論文集, Vol. 2021, pp. 147–151, 2021.

[7] 柴田真琴. OCaml Blockly のチュートリアルサイト. `http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/24/kyozai/demos/tutorial/tutorial.html`.

[8] 柴田真琴. OCaml Blockly のチュートリアルサイトの理解度向上に向けた改善. 修士論文, お茶の水女子大学, 2023.

[9] 松島勇介, 上野雄大, 森畑明昌, 大堀淳. 宣言的記述からの関数型言語によるゲームプログラムの導出. 第 13 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, 3 2011.

[10] 松本晴香, 浅井健一. Blockly をベースにした OCaml ビジュアルプログラミングエディタ. 第 21 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, 2019.

[11] 任天堂. ナビつき! つくってわかる はじめてゲームプログラミング. `https://www.nintendo.com/jp/switch/awuxa/index.html`.