令和３年度 修士論文

Mio: プログラムデザインを支援するブロック学習環境

東京工業大学 情報理工学院数理・計算科学系
学籍番号 20M30284
能勢　純弥

指導教員
増原 英彦 教授

令和４年２月28日

概要

初学者のプログラミング能力函養を目的とするデザインレシピは、プログラムを作ることで問題を解決する一連の流れを複数のステップに分割した教育スキームである。デザインレシピを実際に活用した教育現場からは、デザインレシピの教育効果について、関数の目的を書く、テストを作成する、といった良いプログラミングの習慣を身につけさせることができたという報告が上がっている。現在のデザインレシピの実践方法には、教授者の指示が中心であり、学習者の作業も自由な形式での記述が多い。また、デザインレシピはデータの分析や関数の概形の作成などの実行できないステップを含んでいる。

そのため、デザインレシピに基づいた現在のプログラミング学習には、中間ステップの成果物に対するフィードバックを教授者が 1 つ 1 つ確認する必要があることや、教授者の指導なしでは学習者がステップを省略してしまうことが問題となる。また、実際にデザインレシピに従ってプログラミングをしていると、前後のステップを行き来して考えることがある。ステップ間での成果物には依存関係があるため、あるステップでの修正が与える影響について考える必要があるが、この影響をふまえて全体を修正することは、学習者のみでは難しいという問題もある。

そこで、本研究では従来教授者が手作業で行っていた支援を計算機によって行う学習環境を提案する。本提案のアイディアは主に 2 つである。1 つ目は、自由記述となっていた部分に対し決まった文法と言語を与えたこと、2 つ目は各ステップの成果物に対して機械的に妥当性を判断し手作業で行っていたフィードバックを機械で行うことである。前のステップが完了しないと次のステップに進ませない、といった制約をもたせ、デザインレシピの各ステップに沿う意義をシステムとして学習者に教授することが可能となる。これらの提案を実現するために、本研究ではデザインレシピに基づくブロック形式とコード形式のハイブリッドなプログラミング学習支援環境 Mio を実装する。Mio の特徴は、学習者が自由な形式で書いていたステップをブロックとして記述させる点である。厳格な文法は学習者の負担となってしまう懸念があるが、ブロック言語にすることよって、学習者が新たな文法を覚えることなく直感的に記述可能である。ブロック言語としては Blockly を用いている。また、前のステップの修正が全体に与える影響を学習者が考えられるようにするために、Mio では前のステップでの変化がその後の既に完了しているステップにどのよ

うな影響を及ぼすのかを、取り消し/再実行を内部で実行することによって矛盾を検出することで解決した。また、実際に関数定義をするコーディングの部分は、ブロックではなくテキストで実践させる。これは、、一般的なテキストベースの環境への移行をスムーズにするためである。

実際、自分でデザインレシピの実践を紙とペンを使う従来の方法などと本研究の Mio を用いた方法での比較を行った。結果として、Mio が従来の方法に比べてデザインレシピの実践への取り組みやすさに有意であるという実感を得られた。

謝辞

この研究は増原英彦教授、叢悠悠助教の数え切れないほどの支援によって成り立ったと考えております。出会ってから 3 年間、愛想を尽かすことなく最後の最後までご指導ご鞭撻していただいたこと、心の底より感謝いたします。多くの学生を抱えていながら、ヘルプサインを出したときにすぐ手を差し伸ばしてくれる先生方には感謝してもしきれません。

常々核心をつく質問で気づきを与えてくれ、こちらからの質問にはこれでもかというほど適切な回答をしてくれた増原先生には感謝とともに尊敬の念が止みません。教育者として、研究者として、人間として、知識や技術だけでない姿勢の部分で様々学ばせていただくことが多数ありました。

様々な環境変化があり、一度も研究室というリアルの場で全員が集合はしなかった 2 年間ではありました。しかし、そんなことを気にしない以上にオンラインでの密な関係で、ありとあらゆる場で多種多様なアドバイスをくれた研究室の皆様に本当に感謝いたします。特に伊澤さん・田辺さんのご意見はいつも鋭く適切で、本質とは何たるかという本質を学ぶことが多くありました。さらには自分の研究も忙しい中お時間割いていただいたことにも大変感謝いたします。お二人をはじめとする研究室の皆様の協力なくしてはこの論文は完成しませんでした。

オンラインを通し、様々な大学・企業・海外の方々との出会いもありました。本研究の鍵である Northeastern 大学の Felleisen 教授をはじめ、様々な出会い・経験をさせていただいたこと、大変感謝いたします。皆様が本研究にご興味をお持ちいただいたこと、大変研究のモチベーションとなりました。

そして何よりも、研究者としての 3 年間を楽しくしてくれたのは他の誰でもない叢先生だと思っております。毎日のように叢先生の部屋に通った日々は忘れません。コロナ禍でなければおそらくもっと毎日叢先生の部屋に通っていたのだと思うと少し悔しい気持ちもします。通えない代わりというわけではございませんが、どんな時間にも返信をいただき、感謝すると同時に先生の睡眠時間が心配になる時期もありました。英語をはじめとする拙い部分を 0 から補っていただいたこと、3 年間丁寧に丁寧にご指導いただいたこと、本当にありがとうございました。

またご縁がありましたら、その際は皆様何卒よろしくお願いいたします。

目次

第 1 章

はじめに

プログラミングは一部の人間だけのものである、という常識は古い考えとなりつつある。2020 年、日本では小学校でプログラミングが必修となった。文部科学省はプログラミングを必修とする意味について、「コンピュータを『魔法の箱』とせず主体的に活用する。」と述べている[*1]。プログラミングはそれ自体が目的ではなく、手段に過ぎないのである。しかし、ただの手段に過ぎないからといって「プログラムは動けば何でも良い、目的が達成できれば良い」と考えてはいけない。こう考える人は、魔法の箱の中身を理解するどころか、魔法のプログラムを自分で作ってしまっている。そんな魔法のプログラム職人は、より複雑なプログラムを作ろうとすると、大量の課題に出会い、思い通りにはいかず何をどう修正すれば良いのかも分からなくなる。

そのようなことを防ぐために、プログラミング教育では、良いプログラミングをするという考え方を忘れてはいけない。プログラミングは確かに目的のための手段・道具であるかもしれないが、それと同時にそれ自体に良し悪しがあるものでもある。その点、手書きの文字とよく似ている。文字は何かを伝達するのに使われる道具だが、汚い文字は伝達に支障が出る。書道コンクールに入賞するほどの字は書けなくても良いが、自分が書いた文字が自分や他人が読めなければ勉強や仕事など生活において苦労する。誰が見ても汚くないと思う字、良い字は書けるべきである。プログラミングでも同様、良いプログラミングが書ける教育が必要である。

1.1　デザインレシピ

Felleisen らは「良いプログラミング」とはソフトウェアを作る際にすべてのステップを見通して体系的にアプローチしていくことであると考え、デザインレシピを提案した [1]。デザインレシピとは、プログラムを作ることで問題を解決する一連の流れであり、問題の分析からコーディングまでの複数のステップから成る。

[*1] https://www.mext.go.jp/content/20200218-mxt_jogai02-100003171_002.pdf

デザインレシピを用いることで良いプログラミングができるようになると期待されている。なぜなら、デザインレシピには関数の目的を書く、テストを作成する、といったステップが盛り込まれており、これらはプログラミングを行う際の良い習慣である [2] からである。

実際、デザインレシピの効果については様々な報告が上がっている。例えば、 米国ライス大学ではデザインレシピの各ステップに沿って双方向な形式の授業を行なっており、その結果としてデザインレシピを学んだ学生は、学んでいない学生に比べて良いプログラミングの習慣を持っていた、という観察が得られている [3]。ほかにも、アメリカの複数の大学でデザインレシピを用いた授業を行ったところ、場合分けの不足などの単純なミスが減少したという結果も報告されている [4]。

デザインレシピは従来、手書き (図 1.1) などを用いた自由な形式で実践されている。さらにそれは教育者の監督下で行われることが多い。そのため、授業でデザインレシピを扱う場合、授業法は教育者に依存している。

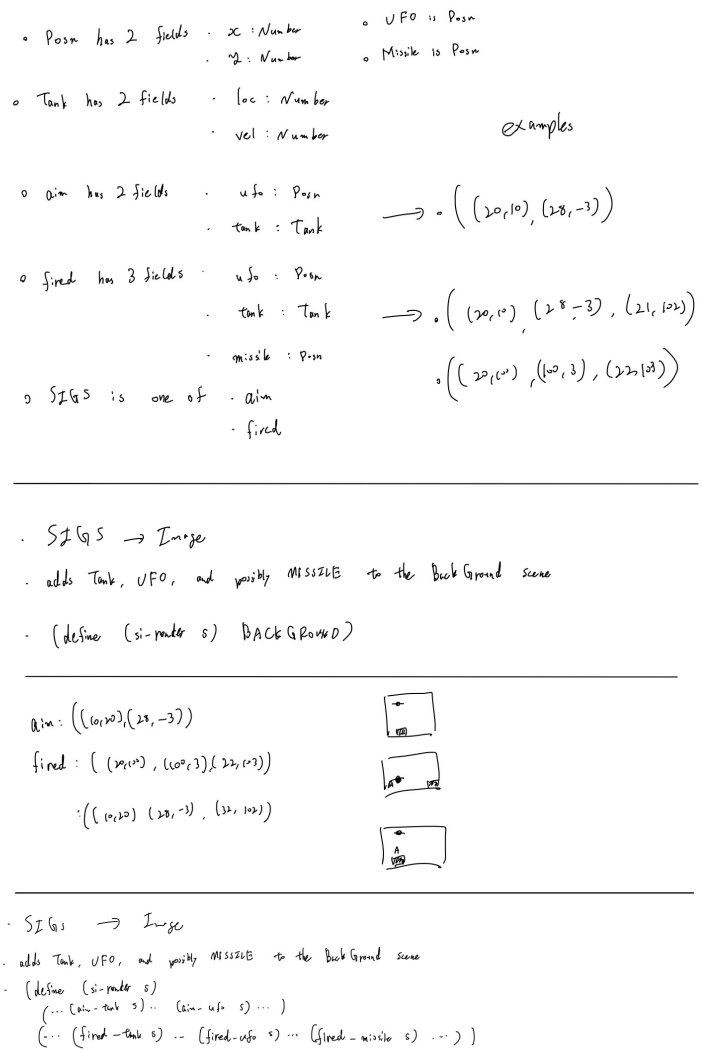

図 1.1: 手書きでのデザインレシピ実践例

そこで、本研究ではデザインレシピ実践をプログラミング環境にて支援する。

1.2　授業における教授者の役割

デザインレシピ実践における教授者の役割を知るために、デザインレシピの第一人者であり数々の大学でデザインレシピを教えてきた Felleisen に実際にデモ授業を実演してもらった。結果、教授者の役割としては以下のようなものが考えられる。

■全ステップに従わせる　デザインレシピは全ステップに従うことで、良いプログラミングをするためのアプローチになる。デザインレシピの序盤のステップであるデータ定義やテンプレートは、書き方に厳格なルールはない。また、これらの序盤のステップははコーディングする前の作業である。しかし学習者が最終的に作成したいものはコーディングであるため、教授者の指示なしでは学生は序盤のステップ

を飛ばしてしまうことがある。よって教授者は学生が全ステップに従ってプログラミングするように指示する必要がある。

■フィードバックを返す　先にも述べた通り、中間ステップにはコメントなどの形式で書かれている場合も多い。それらの妥当性は機械で判断することはできず、教授者が手作業で見ることで、学生に中間成果の妥当性伝えることが可能となる。

1.3　デザインレシピをサポートするプログラミング環境

デザインレシピをプログラミング環境にて支援しようとした例はいくつかある。中高生向けのプログラミング教室 Bootstrap[*2]では WeScheme [5] という環境 (図 1.2) が使われている。WeScheme では学習者がブラウザ上で関数の型や入出力の例を入力することがき、関数の型と入出力の例を書かないとコーディングが始められない環境である。

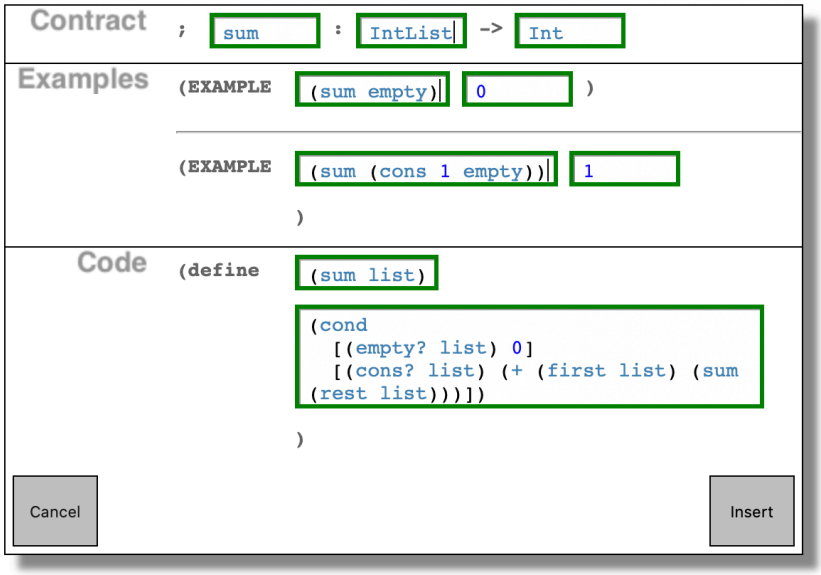

図 1.2: wescheme

また、Ryu [6] は DRaCO というプログラミング環境 (図 1.3) を開発した。DRaCO では関数定義を始める前に、契約文や目的文、テストケースなどを書くことが推奨されている。

[*2] https://www.bootstrapworld.org

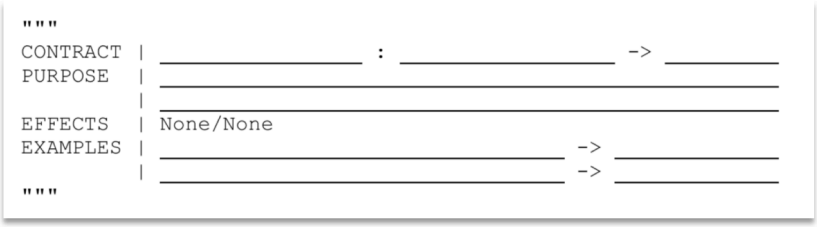

図 1.3: DRaCO

しかし、これらはいずれもレシピの一部のステップのみをサポートしている。また、中間ステップにおける成果物に対してのフィードバックなども特にない。

そこで、本研究の環境では、デザインレシピの全てのステップに従わせ、各ステップにおいてフィードバックを返すように設計する。前のステップが完了しないと次のステップに進ませない、といった制約をもたせ、デザインレシピの各ステップに沿う意義をシステムとして学習者に教授することが可能となる。

1.4　Mio

環境が教授者の代わりとなることも大切であるが、学習者が手書き以上に使いやすい環境である必要がある。デザインレシピ実践を手書きでやることのメリットとして考えられるのは、あるステップをやっている最中に前のステップに自由に戻れる点である。デザインレシピを実際に実践していると、前後のステップを行き来して試行錯誤することが多い。そのため、この手書きの良さは残したいと考えた。

そこで本研究では、教授者の役割を果たし、さらには手書きの良さを残す環境として Mio(図 1.4) を提案し実装した。

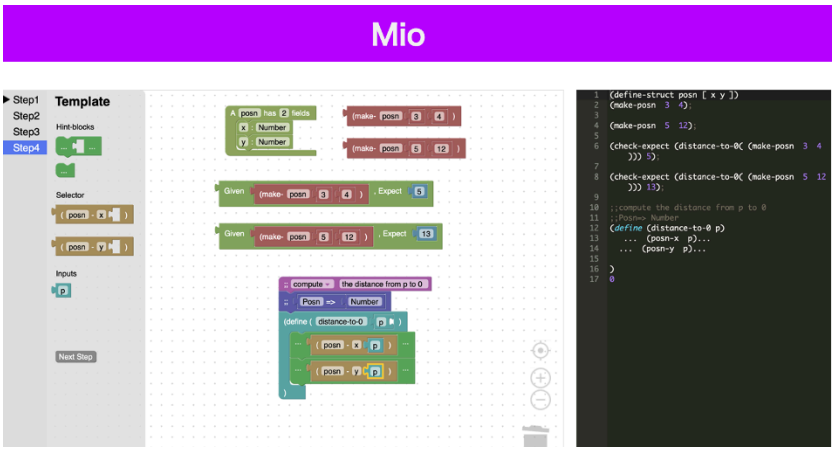

図 1.4: Mio 全体図

本研究のアイディアは主に 3 つである。1 つ目は、自由記述となっていた部分に対し決まった文法と言語を与えたこと、2 つ目は各ステップの成果物に対して機械

的に妥当性を判断し手作業で行っていたフィードバックを機械で行うこと、3 つ目は前後のステップを自由に試行錯誤できるようなシステムの構築である。

これらの提案を実現するために、Blockly [7] を用いた、デザインレシピに基づくブロック形式とコード形式のハイブリッドなプログラミング学習支援環境 Mio を実装する。Mio の特徴は、学習者が自由な形式で書いていたステップをブロックとして記述させる点である。厳格な文法は学習者の負担となってしまう懸念があるが、ブロック言語にすることよって、学習者が新たな文法を覚えることなく直感的に記述可能である。

本論文では第 2 章では、デザインレシピの各ステップの内容を分析する。第 3 章では Mio の機能について述べ、第 4 章では Mio の実装について述べる。第 5 章では評価について述べ、第 6 章では関連研究について述べる。第 7 章でまとめを述べる。

第 2 章

背景

2.1 デザインレシピ

デザインレシピ [1] とは、具体的には以下の 6 つのステップのことを言う。

- データ構造の分析・データ例の作成
- 目的・入出力の型の決定・ヘッダーの作成
- 入出力の例 (テストケース) の作成
- テンプレートの作成
- コーディング
- テストの実行

データ構造の分析や目的文などはコメントとして、テストなどはコードとして書く。ここでは、以下の問題を使って各ステップの作業を説明する。

問題

平面上のある点と原点との距離を返す関数を定義せよ。

■Step 1：**データ構造の分析・データ例の作成**　まずは、入力として与えるデータについての考察を行う。今から定義しようとしている関数の引数は 2 つの座標を持つ点である。このデータ構造を Racket の構造体として定義する。今回定義する座標を表す構造体を `posn` とする。そのため、2 つのフィールド (x 座標と y 座標) を持つデータ構造 `posn` は以下のように定義することができる。

```
(define-struct posn [x y])
```

Racket では構造体を定義すると、`make-`**<構造体の名前>**というコンストラクタ関数が自動で定義される。今回は `posn` を定義しているため、`make-posn` が自動で定義される。これを用いて、`posn` の具体例を以下のように作成する。

```
(make-posn 0 0)
```

```
(make-posn 0 5)
(make-posn 3 4)
(make-posn 5 12)
```

■Step 2：**目的・入出力の型の決定・ヘッダーの作成**　次に、関数の目的、名前、入力と出力の型およびダミーの出力を記述する。この関数の目的は与えられた点と原点の距離を求めることである。よって目的文として `Computes the distance of p to the origin`、適切な関数名として `distance-to-0` を与える。また、入力の型は `Posn` 、出力の型は `Number` である。ダミーの出力としては、例えば `0` を与えることができる。これらの情報を使うと、正しく動作はしないが実行可能なダミーの関数定義を書くことができる。

```
;Compute the distance of p to the origin
;Posn → Number
(define (distance-to-0 p)
  0)
```

■Step 3：**入出力の例 (テストケース) の作成**　次に、Step 1 で作成したデータ例を入力とする入出力の例を作る。今回、Step 1 で例を 4 つ作成したため、入出力の例も 4 つ作成する。

入力	出力
(0, 0)	0
(0, 5)	5
(3, 4)	5
(5, 12)	13

■Step 4：**テンプレートの作成**　このステップでは、Step 1 で確認した入力データの構造をもとに、関数本体の大まかな形を定める。テンプレートは、Step 2 の関数のヘッダーを書き換えることで作成する。今回は、入力 `p` は `posn` 構造体である。この構造体は `x` と `y` のフィールドを持っている。これらのフィールドが計算に使われる可能性があるため、セレクタ関数 `posn-x`, `posn-y` を用いて抽出する。これらのセレクタ関数は `posn` の定義時に自動で生成される。よって、`distance-to-0` のテンプレートは以下のようになる。

```
;Compute the distance of p to the origin
;Posn → Number
(define (distance-to-0 p)
```

```
(... (posn-x p) ...
 ... (posn-y p) ...))
```

■Step 5：**コーディング**　テンプレートを作成したら、”...” を具体的な計算に置き換えることで関数定義を完成させる。

```
;Compute the distance of p to the origin
;Posn → Number
(define (distance-to-0 p)
 (sqrt
  (+ (sqr (posn-x p))
     (sqr (posn-y p)))))
```

■Step 6：**テスト**　最後に、Step 3 で作成した入出力例をもとに `check-expect` を用いてテ ストを作成する。テストを実行することで、プログラムの動作を確認する。

```
(check-expect (distance-to-0 (make-posn 0 0)) 0)
>true
(check-expect (distance-to-0 (make-posn 0 5)) 5)
>true
(check-expect (distance-to-0 (make-posn 3 4)) 5)
>true
(check-expect (distance-to-0 (make-posn 5 12)) 13)
>true
```

テストの値が `false` になった場合は、Step 3 に戻って入出力例の正しさを確認した上で、Step 5 のコードを見直す。

第 3 章

デザインレシピに沿ったプログラミングのための環境 Mio の提案

今まで教授者の指示が中心であったデザインレシピを用いた授業を計算機で支援する環境が Mio である。計算機として支援する上で達成すべき目標を以下の 3 つとした。

■各ステップの記述のサポート　計算機として各ステップでのやるべきことを明確にするサポートを行うべきであると考えた。学生がいくつかの中間ステップをスキップしてしまう原因の 1 つとして、書き方が決まっていない、という点が考えられる。先にも述べた通り、デザインレシピの序盤のステップであるデータ定義やテンプレートは、書き方に厳格なルールはなく、紙やペン・コメントなどを用いて学生が独自に記述していく必要がある。何をどのように書けば良いのかが決まっていないことは、学習者にとって大きな負担となる。そのため、厳格な文法が決まっているコーディングのステップから始めてしまうのである。

■中間成果物の自動検査　従来、教授者が手作業で行なっていた中間成果物の妥当性を自動でフィードバックする。学生にとってフィードバックが必要となる内容は以下のようなタイプに分別できると考える。

- 各ステップでの成果物の不足
- データ例の網羅性が不十分
- 前のステップとの矛盾点

成果物の不足の例としては、目的文を書かずに次のステップに進んでしまう、などが考えられる。また、データ例の網羅性が不十分である例としては、リストに対しての関数を定義している場合、データ例が空のリストだけなどの場合、十分なデータ例とは言えない。また、前のステップとの矛盾の例としては、シグネチャと関数定義で引数の数が異なる場合などが考えられる。ただし、これらはいずれもコメントや、実行できないコードの形で書かれている。そのため、計算機がこれらの妥当

性を判断して学生にフィードバックを返すことは難しい。

■前後のステップの行き来の自由　実際にデザインレシピに従ってプログラミングをしていると、前後のステップを行き来して考えることがある。しかし、データ型とデータ例などのようにステップ間での成果物には依存関係がある。そのため、すでに完了しているステップを修正した際には、その修正が与える影響について考える必要がある。しかし、その影響を踏まえて修正すべき箇所を全て学習者のみで判断するのは難しい。

3.1　Mio の方針

これらの目標を達成するために、まず自由に記述している部分を、決まった文法で書くことによって、機械的な処理ができると考えた。機械的な処理が可能になることで、各ステップの自動検査が可能となる。また文法を与えることで学習者にとっては各ステップの書き方が明確になり、取り組みやすくなり、さらには機械的な処理によって全ステップに強制的に従わせることが可能となる。

しかし、文法を新たに覚えることは学習者にとって負担である。そこで、ブロック言語を用いた環境を提案する。ブロック言語にすることよって、学習者が新たな文法を覚えることなく直感的に記述可能である。

また、前後のステップの行き来を容易にするために、前のステップの記述にさかのぼって修正しても後のステップの記述は維持される、しかし矛盾が起きる場合には指摘をする機能を提案する。

これを実現するために実装したのが Mio(図 3.1) である。Mio は、ブロック形式とコード形式のハイブリッドなプログラミング学習環境である。現在は Racket 言語に対応している。今まで紙とペンやコメントで書いていた実行されない Step 1∼4 をブロック形式で書かせている。一方、コーディング部分はテキスト形式で書かせる。コーディング部分はブロックではなくテキスト形式で書かせるのは、一般的なテキストベースの環境への移行をスムーズにするためである。

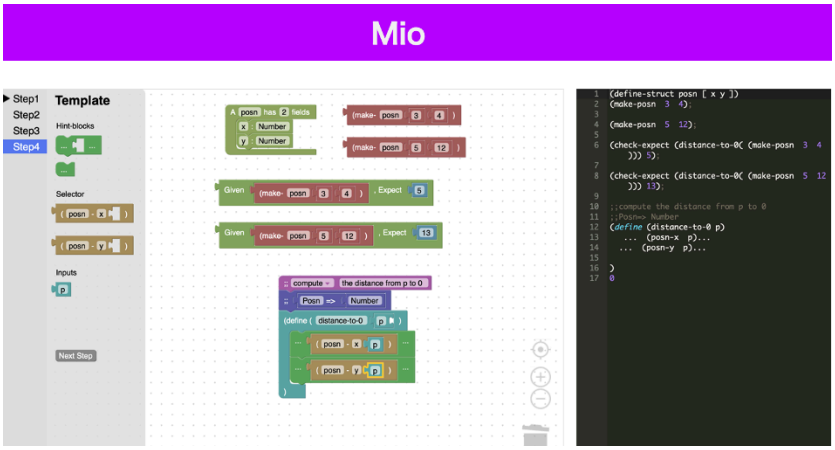

図 3.1: Mio 全体図

3.2　Mio の機能

Mio では具体的には、以下の 3 つのサポートを提供する。

- 各ステップのタスクの明示と補助
- 各ステップの中間成果に対するフィードバック
- 前後のステップの矛盾の有無

3.2.1　各ステップのタスクの明示と補助

Mio ではブロックを用いることで、各ステップで書くべき情報とその書き方を明確にしている。以下、第 2 章の問題を用いて、Mio における各ステップの作業を説明する。

問題：平面上のある点と原点との距離を返す関数を定義せよ。

■Step 1：**データ構造の分析・データ例の作成**　Ramsey は Step 1 をデータの分析とデータ例の作成の 2 つのパートに分け、それぞれを Step 1A, Step 1B として授業で扱った [8]。Mio でも同様に Step 1 をデータ構造の分析とデータ例の作成の 2 つにパートに分け、それぞれを Step 1-a、Step 1-b としている。

まず、Step 1-a として入力として与えるデータについての考察を行う。Racket では、点を表す構造体 `posn` を `(define-struct posn [x y])` のように定義した。Mio では、学習者は図 3.2 のように `define-struct` ブロックと `field` ブロックを用いて定義する。

図 3.2: Step 1-a(データ構造の分析)

ここで、Racket ではコンストラクタ関数 `make-posn` が自動で定義された。Mio でも Step 1-b のデータ例の作成のステップに進むと、Racket と同様に Make ブロックが自動生成される。これを用いて、学習者は `posn` の具体例 `(0,0)`, `(0,5)`, `(3,4)`, `(5,12)` を Mio では図 3.3 のように定義する。

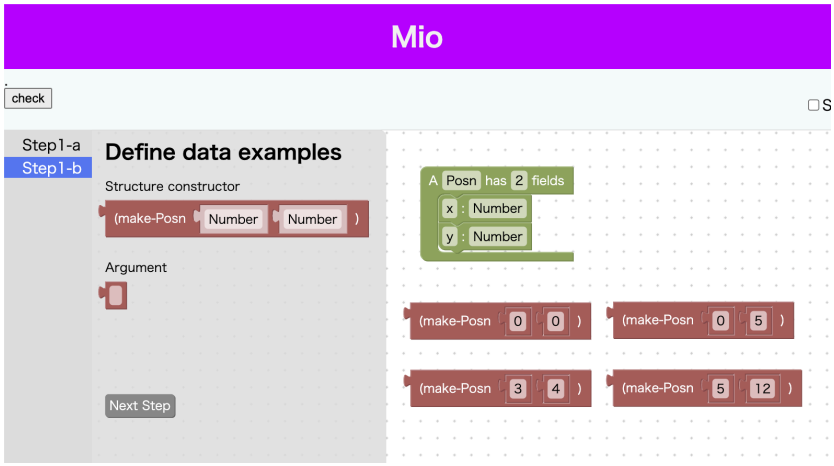

図 3.3: Step 1-b(データ例の作成)

■Step 2 : **目的・入出力の型の決定・ヘッダーの作成**　Step 2 では、学習者は図 3.4 のように関数の目的ブロック、シグネチャブロック、ヘッダブロックを用いて関数の目的文、契約文、ヘッダを定義していく。このうち、目的文ブロックは、先頭の動詞の候補として `compute`, `decide`, `create` を提供する (図 3.5)。

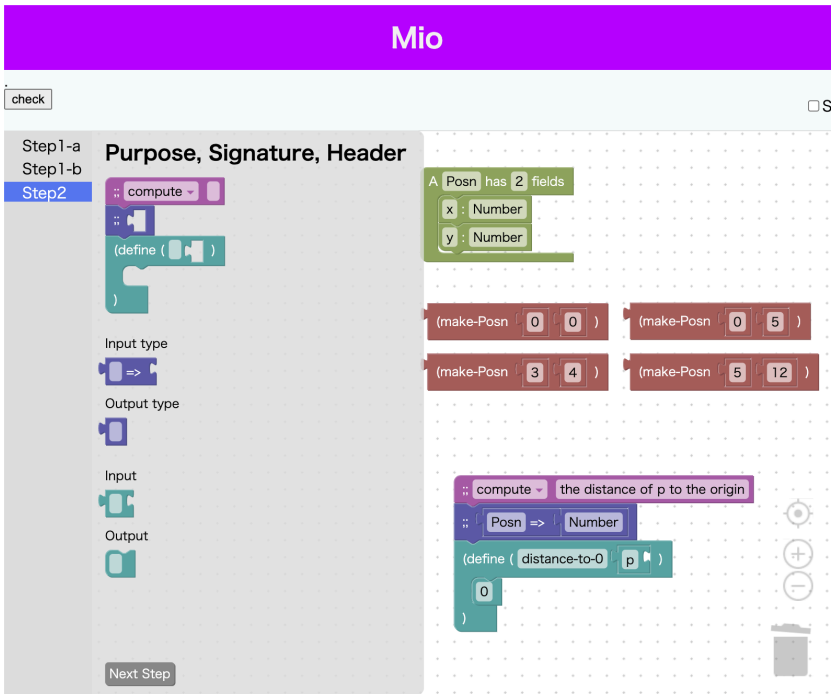

図 3.4: Step 2(目的・入出力の型の決定・ヘッダーの作成)

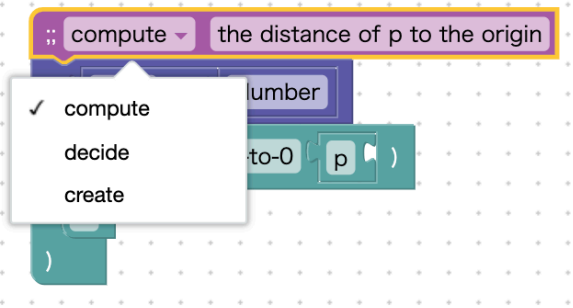

図 3.5: 目的文の動詞の選択肢

■Step 3：**入出力の例 (テストケース) の作成**　Step 3 では、学習者は Step 1 で作成したデータ例を入力とする入出力の例を作る。Mio では Step1 で作成したデータ例を学習者が利用できるよう、各データ例に対応するブロックが自動で生成される (図 3.6)。

図 3.6: Step 3(入出力の例 (テストケース) の作成)

■Step 4 : **テンプレートの作成**　このステップでは、Step 2 のヘッダブロックをもとにテンプレートを作成する。Racket では、`posn` を定義した際にセレクタ関数 `posn-x`, `posn-y` が自動で定義された。Mio でも同様に `posn-x` ブロックと `posn-y` ブロックが自動生成される。これらを用いて学習者は図 3.7 のように `distance-to-0` のテンプレートを作成する。

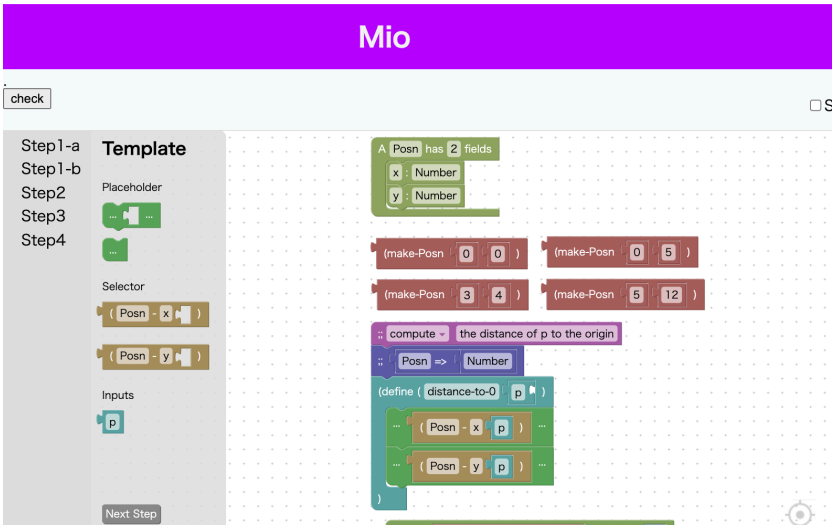

図 3.7: Step 4(テンプレートの作成)

■Step 5：**コーディング**　Step 5 以降はブロック形式ではなくテキスト形式で実装する。これは、Mio という教育用の環境からより一般的な環境への移行をスムーズにするためである。Show Code のチェックボックスにチェックを入れることで、ブロックで定義したものが全てテキストコードに変換されてテキストエディタに表示される (図 3.8)。

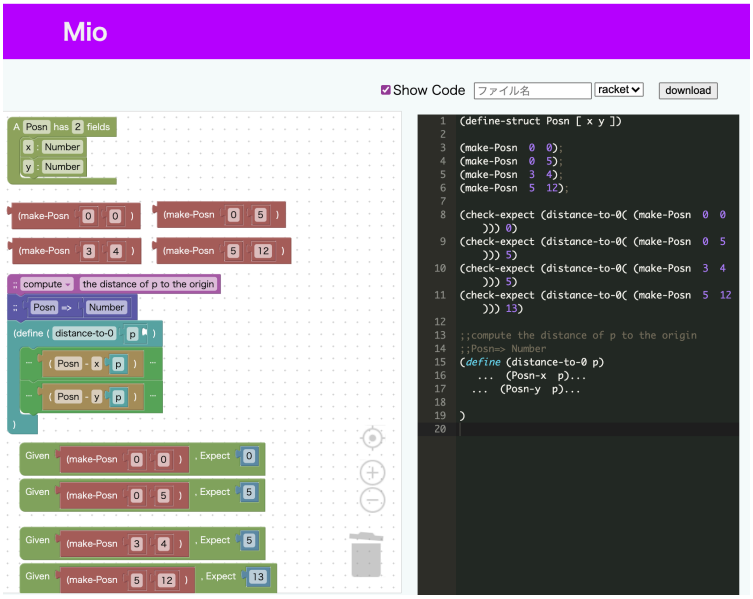

図 3.8: Step 5(コーディング) 実行前

テキスト形式のコードが得られたら、学習者は Step 4 で定めたテンプレートの … を具体的なコードに置き換えていくことで、図 3.9 のようにプログラムを完成させる。

```
13  ;;compute the distance of p to the origin
14  ;;Posn=> Number
15  (define (distance-to-0 p)
16    (sqrt
17      (+ (sqr (Posn-x  p))
18         (sqr (Posn-y  p))))
19  )
```

図 3.9: Step 5(コーディング) 実行後

■Step 6：**テスト**　最後に、Step 3 で作成したテストを実行する。現在の Mio では、テストの実行機能は搭載していないため、学習者がダウンロードボタン押してファイルを取得し、自分の環境でプログラムを実行する必要がある。

3.2.2　各ステップの中間成果に対するフィードバック

Mio では学習者が要求した時と、次のステップに進む時に、現在のステップの成果物に対して妥当性を判断する。妥当でないと判断された場合は次のステップには進むことができないようになっている。

各ステップでどのようなフィードバックが返ってくるのかを先ほどの例を使って説明する。

■Step 1：**データ構造の分析・データ例の作成**　まず、Step 1-a では、全ての入力欄が埋まっているかを確認する。もし埋まっていない状態で次のステップに進もうとすると、それを警告するフィードバックを返す。たとえば、図 3.10 のようにフィールドの型が書かれていない時は、該当の不備があるブロックをハイライトで明示し、フィールドの型が足りていないというメッセージを表示する。

図 3.10: Step 1-a の失敗例

Step 1-b でも同様に、不備がある場合は警告する。たとえば、図 3.11 のようにデータ例に引数が足りない場合は、データ例のブロックをハイライトし、引数がないこというメッセージを表示する。図 3.12 のようにデータ例の引数に値を書き忘れている場合は、データ例の引数のブロックをハイライトし、引数の入力がないというメッセージを表示する。

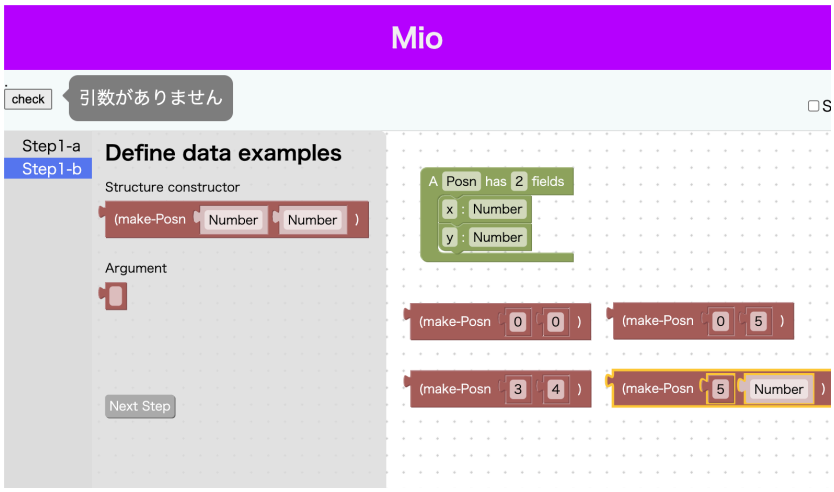

図 3.11: Step 1-b の失敗例 1

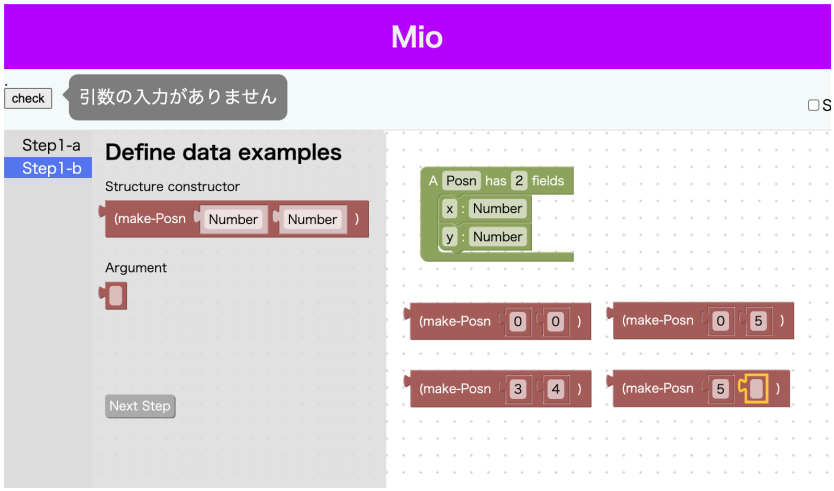

図 3.12: Step 1-b の失敗例 2

■Step 2: **目的・入出力の型の決定・ヘッダーの作成**　Step 2 では、Step 1 と同様に図 3.13 のように不備をチェックする他に、入力されているものの矛盾も警告する。例えば、図 3.14 のようにシグネチャブロックとヘッダブロックにおいて入力数が一致していない場合は、関数のヘッダをハイライトした上で、シグネチャとヘッダの入力数が矛盾していることとそれぞれの個数をメッセージとして表示する。

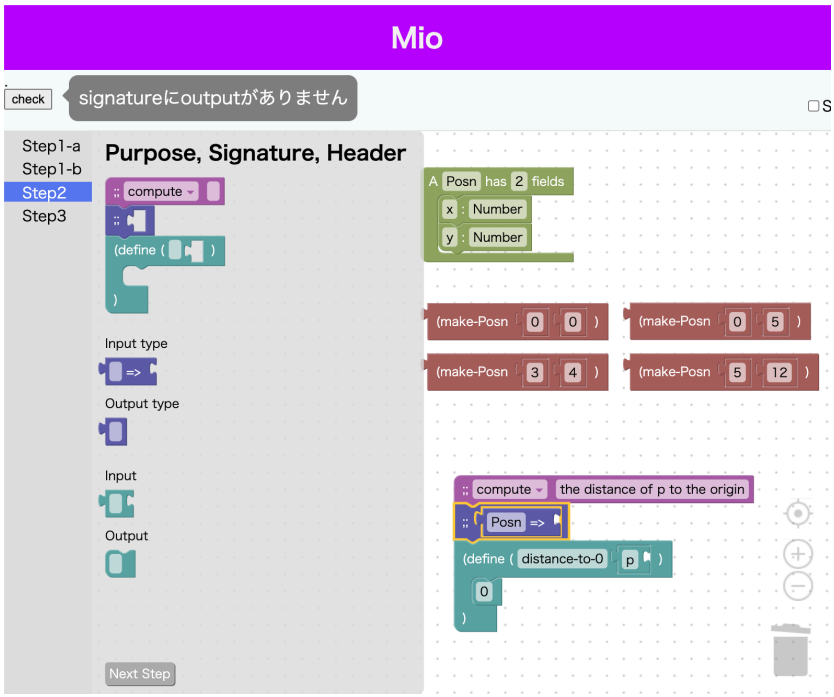

図 3.13: Step 2 の失敗例 1

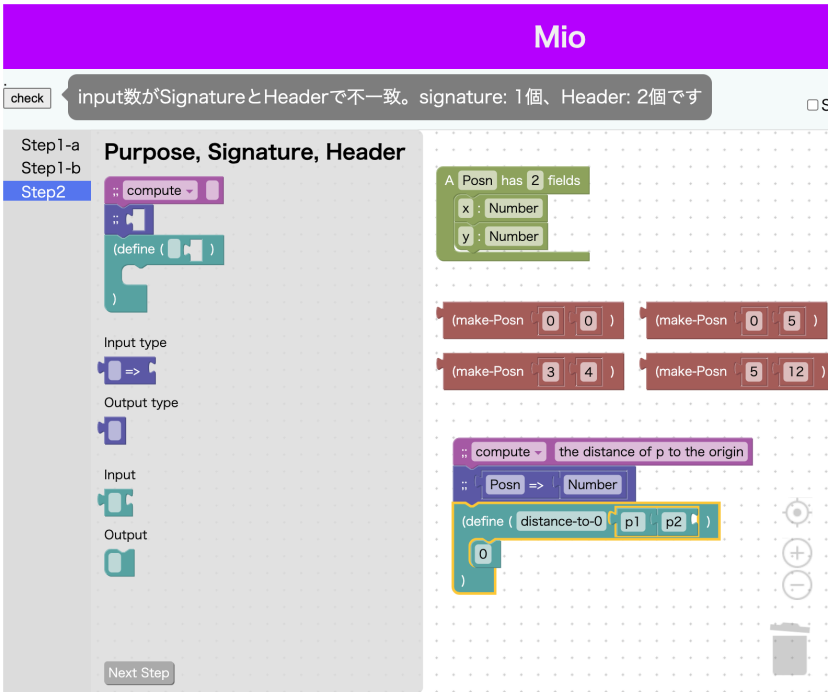

図 3.14: Step 2 の失敗例 2

■Step 3: **入出力の例 (テストケース) の作**　Step 3 でも同様に不備をチェックするが、別のフィードバックの例として、Step 3 で入出力の例を作成する際に、

Step 1 で作ったデータ例を全て使用していない状態がある。例えば、図 3.15 では、(make-posn 5 12) が入出力の例として使われていないので、(make-posn 5 12) のブロックをハイライトし、この例が使われていないことをメッセージとして表示する。

図 3.15: Step 3 の失敗例

■Step 4: **テンプレートの作成**　Step 4 では不備のチェックのほかに、セレクタブロックを全て使っているかのチェックも行う。図 3.16 のように、posn の y 座標を取り出す (posn-y ...) ブロックが使われていないため、テンプレートのブロックをハイライトし、使われていないセレクタブロックがあるというメッセージを表示する。

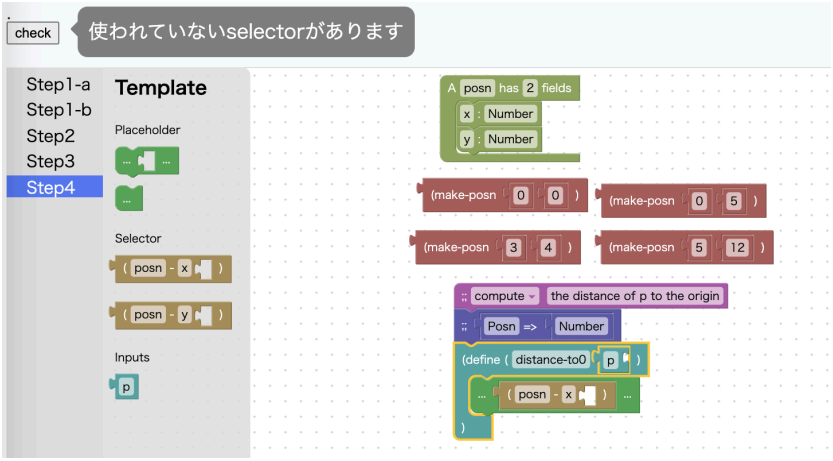

図 3.16: Step 4 の失敗例

■Step 5: **コーディング**, Step 6 : **テスト**　Step 5,Step 6 で Mio 側が提示するフィードバックは現在ない。Step 6 で行うテストが Step 5 に対するフィードバックにあたる。

3.2.3　前後のステップの矛盾の有無

次のステップに進む時には、現在のステップが完了してから進む必要があるが、前のステップには自由に戻ることができる。その際、前のステップを変更した時にそれ以降のステップに影響がある場合は、それらに学習者が気づけるように環境側でサポートする。

例として、図 3.17 のように Step 2 の途中で、Step 1-a の修正を行う場面を考える。具体的には Step 1-a でデータ構造の定義に新たにフィールドブロックを加える。

図 3.17: Step 2 の途中

この状態で、Step 1-a に戻り、field を 1 つ追加する (図 3.18)。

図 3.18: Step 1-a に戻って修正

フィールドを追加した上で Step 2 に戻ろうとすると、図 3.19 のように、Step 1-b のデータ例ブロックにフィールドが自動的に追加される。また、すでに作成したデータ例が妥当でないという警告が出る。

図 3.19: Step 2 を再開

第 4 章

実装

Mio の実装は主に Blockly [7] を用いている。Blockly は、ブロックエディタの UI と、ブロックの情報をテキストとして生成するためのフレームワークである。Blockly を用いることで、各ステップに対応したオリジナルのブロックを定義することが可能になる。また、今までのブロックの情報をもとに新たなステップのブロックを定義することや、中間ステップでの成果物を評価することができる。

4.1 オリジナルのブロックの定義

Mio では、各ステップに対応したオリジナルのブロックを JSON 形式で定義している。この節では具体例を用いてブロックを定義している際に与えている情報を説明する。

例えば、define-struct ブロック (図 4.1) はソースコード 4.1 の 1 行目から 12 行目のように定義される。3 行目の `type` でそのブロックの種類を定義する。5 行目 6 行目で図 4.1①部分を表している。具体的には、5 行目の `message0` の `%1` や `%2` はブロックに対する引数の場所を表し、6 行目の `args0` は引数の種類や名前などの情報を表す。同様に、8,9 行目の `message1`,`args1` で図 4.1②を表している。

また、field ブロック (図 4.2) はソースコード 4.1 の 15 行目から 26 行目のように定義される。17 行目 `type` で種類を定義し、19,20 行目 `message0`,`args0` で図 4.2③の部分を表している。また、23 行目の `previousStatement` は図 4.2④の部分のこのブロックの前 (上) に置けるブロックの種類を表している。同様に 24 行目の `nextStatement` は図 4.2⑤の部分のこのブロックの次 (下) に置けるブロックの種類を表している。

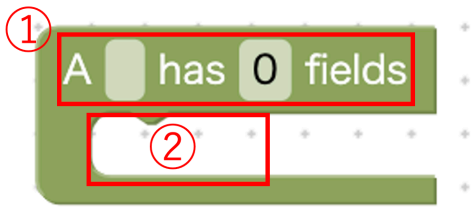

図 4.1: define-struct ブロック

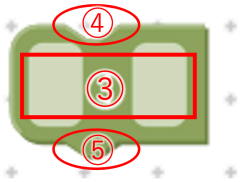

図 4.2: field ブロック

ソースコード 4.1: Step 1-a ブロックの定義

```
1  //define_struct の定義
2  Blockly.defineBlocksWithJsonArray([{
3    //ブロックの種類
4    "type": "define_struct",
5
6    //ブロックに表示されるメッセージと引数
7    "message0": "A %1 has %2 fields",
8     //引数の情報
9    "args0": [{"type": "field_input", "name": "struct_name"},
10           {"type": "field_number", "name": "struct_arg_number"}],
11
12   "message1": "%1",
13   "args1": [{"type": "input_statement", "name": "field",
14               "check": "struct_field"}],
15
16    //色の指定
17   "colour": 80 }]
18 );
19
20
21 //field の定義
22 Blockly.defineBlocksWithJsonArray([{
23   "type": "struct_field",
24
25   "message0": '%1: %2',
26   "args0": [{"type": "field_input", "name": "struct_field"},
27          {"type": "field_input", "name": "struct_field_type"}],
28
29   //前のブロックの種類の指定
30   "previousStatement": "struct_field",
31   //後のブロックの種類の指定
32   "nextStatement":"struct_field",
```

```
33
34   "colour": 80 }]
35 );
```

また、Step 1-b 以降は、以前のステップの情報をもとにブロックが作られるため、ソースコード 4.1 のようなブロックの定義を、次のステップのブロックを作るための関数の中で行っている。

4.2　ブロックのデータ処理

Mio ではブロックの情報をもとにフィードバックを自動生成している。この節では具体例を用いてブロックの情報の処理の流れを説明する。

■Step 1-a　ソースコード 4.2 は Step 1-a で作られた成果物に対してフィードバックを返す関数 check_Step 1a の一部である。3 行目で関数 getBlocksByType を用いて、type が define_struct のブロック全てを取得する。それぞれのブロックの中の struct_name という名前のフィールドの値を Blockly のプリミティブ関数 getFieldValue を用いて取得し、それが空かどうかを確かめる。空の場合、alert_blink という与えられたブロックを点滅させ、メッセージを表示する関数でフィードバックを返す。

ソースコード 4.2: Step 1-a へのフィードバック (一部)

```
1  function check_Step1a(){
2    //省略
3
4    //全define_struct ブロックを取得
5    for (const struct_block of getBlocksByType("define_struct")) {
6
7      //define_struct ブロックの定義名の field が空かチェック
8      if(struct_block.getFieldValue("struct_name").length==0){
9
10       //空であればブロックの点滅・メッセージを表示
11       alert_blink(struct_block, "struct の名前がありません")
12       break
13     }
14    }
15  //省略
16 }
```

■Step 3　Step 3 で作った入出力の例に対するフィードバックを返す関数 check_Step 3 には、Step 1-b で作ったデータ例を全て使っているかの確認が含ま

れている。ソースコード??はその一部である。大まかな流れとしては、Step 3 で作った入出力の例に使われている入力例 (図 4.3①) のリストを作り、Step 1-b で作ったデータ例 (図 4.3②) がそのリストの中にあるかを順番に確認していく。

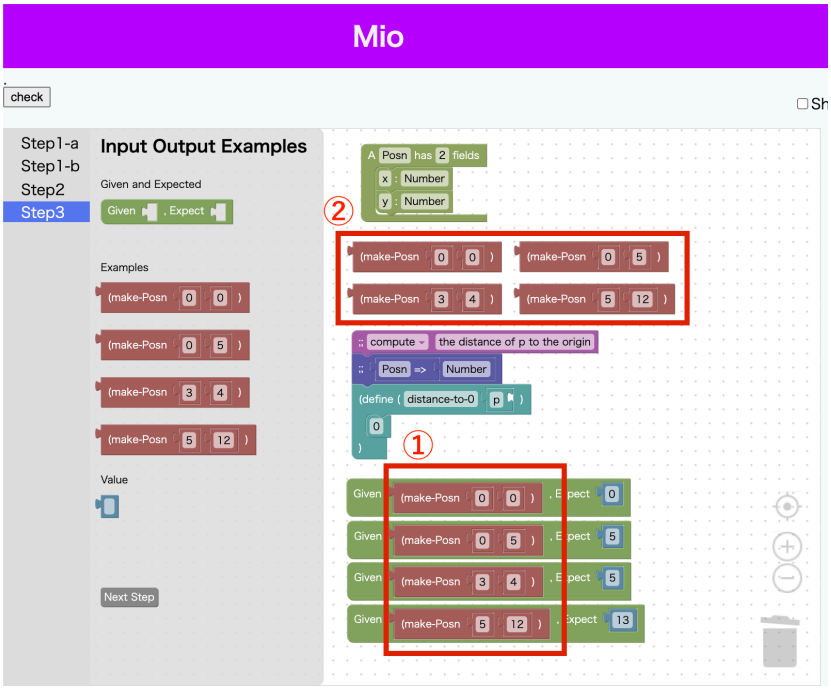

図 4.3: Step 3

まず 7 行目から 21 行目を用いて Step 3 で作った入出力の例に使われている入力例のリスト `inputs_of_io_examples` を作成している。より具体的には、11 行目で入出力の例のブロックを全て取り出し、12 行目から 19 行目は 1 つ 1 つの入出力の例に使われている入力をリストにして、`inputs_of_io_examples` に結合していく。11 行目の `getDescendants()` はそのブロックの中にある全てのブロックをリストとして返すプリミティブ関数である。この中から、13 行目で `filter` を用いて入力例のみを取り出す。そして、16 行目で取り出したものを `map` を用いて全てテキスト形式のコードに変換する。コードへの変換というのは、例えば図 4.4 のようなブロックは `(make-posn 5 12)` というテキストに変換される。これは、後で Step 1-b のデータ例のブロックとの比較をするためである。Blockly では同じ種類・同じ値のブロックでも別のブロックとして認識する。そのため、種類や値が等しい 2 つのブロックを同じブロックであると判別するには、Block をテキスト形式のコードに直す必要がある。

図 4.4: Step 1-b 例

23 行目から 34 行目で、Step 1-b で作った例が 7 行目から 21 行目で作った入出力の例に使われている入力例のリスト `inputs_of_io_examples` に入っているかを 1 つずつ確認していく。より具体的には、25 行目で全ての例ブロックを取り出す。27 行目は親ブロックがないブロックに絞っている。親ブロックというのは、外側のブロックのことを指す。例えば、図 4.5①には親ブロックが存在しないが、図 4.5②には親ブロックが存在する。そして 29 行目で実際にブロックが `inputs_of_io_examples` のリストに入っているかを確認し、入っていない場合は、関数 `alert_blink` を用いて使われていないブロックを点滅させ、メッセージを表示する。

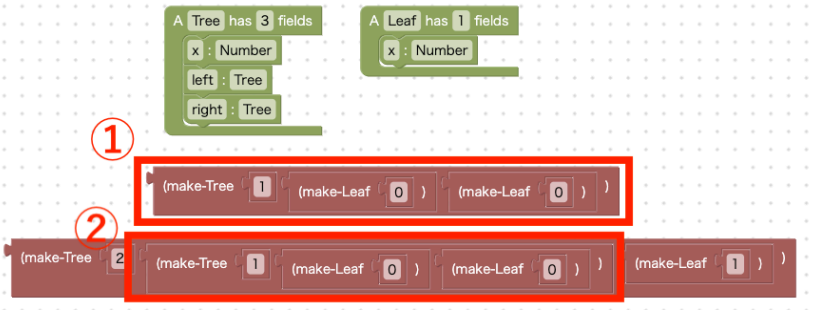

図 4.5: Step 1-b 親があるブロックとないブロック

ソースコード 4.3: Step 3 へのフィードバック (一部)

```
1  function check_Step3(){
2    //省略
3
4    //定義されている構造の種類数
5    var number_of_structure = getBlocksByType("define_struct").
         length
6
7    //入出力の例に使われている入力例を入れるためのリスト
8    var inputs_of_io_examples =[]
9    //入出力の例に使われている入力例をリストに入れる
10   for (let n =1; n <= number_of_structure; n ++) {
11     for(const io_ex of getBlocksByType("io_examples")){
12       inputs_of_io_examples.push(
13         io_ex.getDescendants(true).filter(function(descendant_io){
14           //対象のブロックのみに絞る
15           return descendant_io.type== "struct_example"+String(n)
16          }).map(function(input_io){
17           //コードに変換する
18           return JSON.stringify(String(Blockly.blockToCode(
                 input_io)))
```

```
19          }))
20      }
21   }
22
23   //例ブロックが全て使われているかの確認
24   for (let m =1; m <= number_of_structure; m ++) {
25    for (const ex of getBlocksByType("make_struct"+String(m))){
26      //親ブロックが無いブロックに限定
27      if(ex.getParent() == null){
28       //例ブロックがリストの中にあるかを確認
29       if(!include(inputs_of_io_examples, JSON.stringify(String(
             Blockly.blockToCode(ex))))){
30        alert_blink(struct, "この例が使われていません")
31       }
32     }
33    }
34   }
35  //省略
36 }
```

4.3　前後のステップの矛盾の有無の検査

この節では、前後のステップの矛盾の有無の検査のアルゴリズムについて説明する。前のステップを修正した時に、その修正が後のステップに影響を与えている場合、前後のステップで矛盾が起きることが想定される。そのため前のステップを修正した際には、その修正が与えた影響を考えて後のステップを修正する必要がある。しかし、その修正が与える影響全てを学生が把握するのは困難である。なぜならば、各ステップの関係は複雑であり、プログラムが大きくなるにつれて各ステップの依存関係はより複雑なものになるためである。

これらを解決するための方法としていくつか考えられる。例えば、前のステップの修正をした際には、その後のステップを全て書き直させる方法である。これは修正が必要な可能性のある全てのステップを考え直すので確実に矛盾は起きないと言えるが、あまりにも学習者の負担が大きい。また、各ブロック同士の関係を予め計算機側が把握しておき、関係のあるブロックのみを書き直させる方法も考えられる。これは全てを書き直す方法よりは修正が少なくて済むが、実際に書き直す必要がないブロックまで書き直させる可能性は高い。またこの方法は今後環境の機能を拡張していく際に、1 つ 1 つブロックの関係性を全て見直す必要があるため、実装面での負担も大きい。他にも、環境側は指示せずに学習者自身に修正点を探し出させる方法もあるが、これは矛盾点が見逃される可能性が高い。

そこで本研究では前のステップの修正をしても後のステップの記述は残し、矛盾

が起きた場合のみ指摘する方法を取る。この方法を実現するために、実装には取り消し/再実行の機能を用いている。取り消し/再実行とは、ブロックの操作を記録し、それを巻き戻したり再生したりできる機能である。この機能を用いて、前のステップの修正をした際に、後のステップの作業を内部で自動で再実行し、その時に矛盾が起きた場合のみ指摘するという方法である。

3.2.3 節では、Step 2 の関数のヘッダーを作っている途中で Step 1-a でのデータ定義を修正すると、その修正がすでに完了している Step 1-b に与える影響を自動検出する例を示した。この具体例を用いて、この機能のアルゴリズムと内部の動きを説明する。

まず、ブロックの動きを保存するリストを次のステップに行くタイミングで、各ステップごとに用意する。例えば図 4.6 のように Step 2 の途中の状態の時は、Step 1-a での操作が保存されているリストと Step 1-b での操作が保存されているリストの 2 つがある。それぞれを `Stack-1a`,`Stack-1b` と呼ぶ。

図 4.6: Step 2 の途中

そして Step 2 が途中の状態で、Step 1-a に戻ると、Step 2 での途中までの操作を保存しているリスト `Stack-currentStep` が作られる。Step 1-a に戻って図 4.7 のように field を 1 つ追加した後に Step 2 に戻ろうとしたタイミングで、`Stack-1a` に Step 1-a での field を 1 つ追加するまでの操作が追加される。

図 4.7: Step 1-a に戻って修正

そして、内部で `Stack-1a` の操作を全て取り消し、その操作を順番に再実行し、その後に Step 1-a のチェックを行う。チェックが通れば次のステップのブロックを再定義する。これを `Stack-1b`,`Stack-currentStep` でも順番に行う。結果、まず Step 1-a が終わった後の次のステップのブロックの再定義で、図 4.8 のように Step 1-b のデータ例ブロックにフィールドが自動的に追加される。そして `Stack-1b` の取り消し/再実行が終わった後の Step 1-b のチェックで、すでに作成したデータ例が妥当でないという警告が図 4.8 のように出る。

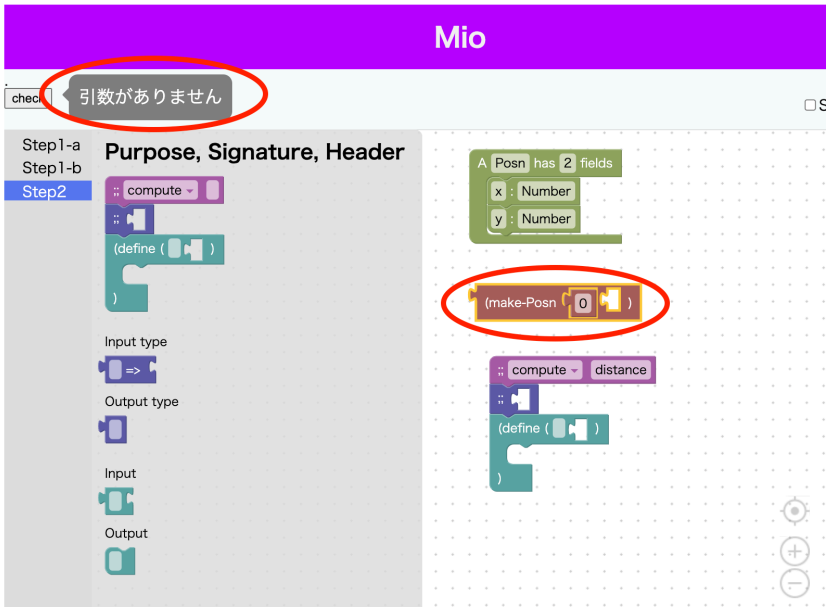

図 4.8: Step 2 を再開

第 5 章

評価

How to Design Programs (HtDP) [1] の例題を用いて自己評価を行った。

5.1 自己評価

自己評価の目的は、Mio が従来の実践方法に比べて役に立つか、という観点である。まずは対象として、複数の構造体を用いた関数を定義する例題を扱う。具体的には HtDP 第 6 章で扱われる UFO ゲームの準備についての問題である。これを扱った理由は、現状の Mio で実践可能な例題の中でより規模の大きい例題だからである。

問題の概要は、簡単なスペースインベーダーゲームをプレイするためのゲームプログラムを設計することである。プレイヤーは戦車 (`Tank`) を操作し、キャンバスの上から下へ降りてくる `UFO` の着陸を阻止するために、スペースバーを押してミサイル (`Missile`) を 1 発発射することができる。UFO がミサイルに衝突すればプレイヤーの勝ち、そうでなければ UFO が着弾してプレイヤーの負けとなる。この問題で必要なデータ構造は以下の 4 つである。括弧内はそのデータ構造が持つ引数である。

- `UFO` (`x` 座標と `y` 座標の 2 つ)
- `Missile` (`x` 座標と `y` 座標の 2 つ)
- `Tank` (位置 `loc` と速度 `vel` の 2 つ)
- `aim` (`UFO` と `Tank` の 2 つ)
- `fired` (`UFO` と `Tank` と `Missile` の 3 つ)

この例題を用いて、紙とペンを使う方法、既存のエディタ (DrRacket) を使う方法、Mio を使う方法の計 3 通りを用いて以下を行った。以下、紙とペンを使う方法と既存のエディタ (DrRacket) を使う方法の 2 つをまとめて従来の実践方法と呼ぶこととする。

1. デザインレシピ Step 1∼4 の実践
2. 意図的な序盤のステップの修正

5.1.1　結果 1: Step 1∼4 の実践

実際にやってみた様子は図 5.1 である。まずは、この規模の問題を Mio で対応可能であるということが分かった。

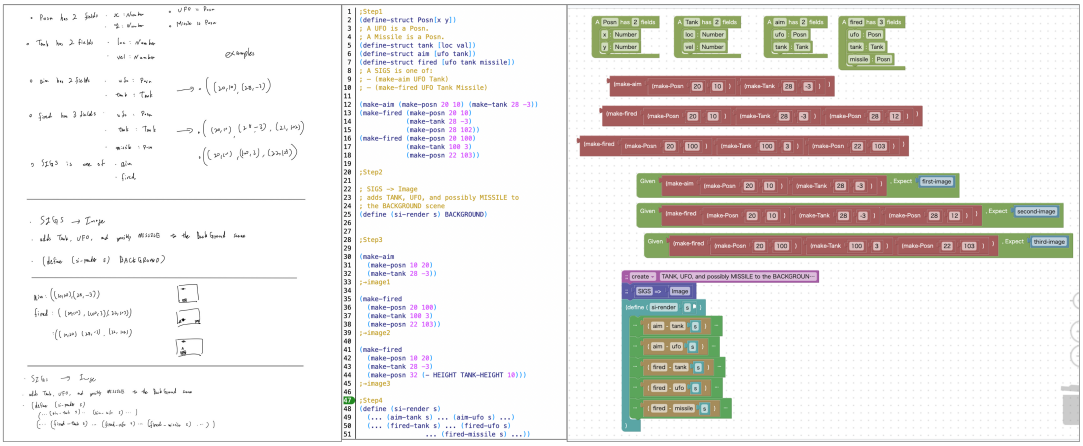

図 5.1: 実践の様子 (左から手書き・DrRacket・Mio)

実践して感じた Mio の恩恵として、ブロックであることによる有用性、学習者の負担の軽減が実感できた。一方で、見やすさの観点で課題を感じた。

■ブロックの有用性　まずは、ブロックであることによって何をするのかが明確に理解できることを実感できた。従来の実践方法は問題に取り組もうとした際に直面するのは白紙や真っ白なエディタである。そのため各ステップでやるべきことが何なのかを思い出す必要がある。一方で Mio ではブロックによって各ステップでのタスクが明らかである。

また、各ステップごと、さらにはブロックの種類ごとに色で分かれているため、どのステップがどこに書いてあるかがすぐに分かる。そのため、それぞれのステップの振り返りが従来の実践方法に比べてやりやすい。

■学習者の負担軽減　Mio でデザインレシピを実践する際は、その問題に対して考えなくてはいけない情報のみの記述で充分である。例えば、データ定義をする際、従来の実践方法では `"...has...fields, ...and..."` や `"(define-struct...)"` などを書く必要があるのに対して、Mio で書く必要がある情報はデータ定義の名前とフィールドの情報に限定されている。これは、全体を通して学習者の負担の大きな差になることを実感できた。

■見やすさに対する課題　複数の構造体を定義したりすると、データ定義とデータ例の対応などが見づらいと感じた。これは、色を自由に変更可能にしたり、対応するブロックを結ぶ線などを自由に記述できるようにするなどのアイディアで今後改善していけると考えている。

また、全てのステップを 1 つのスペースに自由に置けるようになっているが、スペースの中でステップごとにブロックが分別されていると、前のステップの修正なども見やすくなると考える。

このような見た目の部分が、学習者、特に初学者が使う上では大切な観点であると考えている。

5.1.2　結果 2: 意図的な序盤のステップの修正

Step 4 まで実践した後で、意図的に序盤のステップを修正した。修正の内容は、Step 1 で定義したデータ構造の 1 つに要素を追加するということである。具体的には、`Tank` の引数として `color` を追加した。これによって必要な修正箇所は、`Tank` の定義と、`Tank` が用いられているデータ例全てである。

修正後の様子は図 5.2 である。変更した箇所を赤い丸で囲っている。まずは Mio でもこのような修正ができることを確認できた。

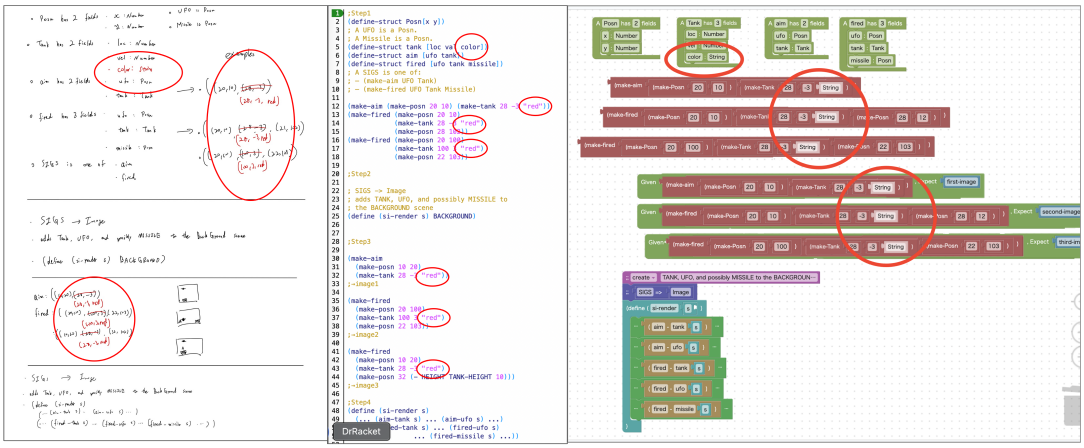

図 5.2: 修正後の様子 (左から手書き・DrRacket・Mio)

実践してみた結果、Mio の恩恵として、変更箇所の明示は学習者に役に立つことが実感できた。一方で、Mio でも把握できない修正すべき箇所があることが分かった。

■変更箇所の明示　図 5.2 でも分かる通り、変更しなくてはいけない箇所は多数の場所に散らばっている。従来の方法ではこれらを漏れなく目視で確認して把握する必要があった。一方で Mio ではデータ定義を変更した際にはその変更に矛盾する箇所を明示してくれる。これは修正する際に学習者の負担の大幅な軽減になることを実感できた。

■把握できない修正箇所　Mio では前後のステップで矛盾が起きる箇所を検出する。そのため、入力例を修正した際の出力の修正が必要かどうかは環境側では把握できないことが分かった。これに対する改善案は今後考えていく必要がある。

第 6 章

関連研究

6.1　デザインレシピを支援するプログラミング環境

第 1 章でも紹介したように、デザインレシピをプログラミング環境にて支援しようとした例はいくつかある。

■WeScheme　WeScheme [5] は、学校教育のために作られたプログラミング環境図 (6.1) で、授業や教室で使用されることを目的としている。実際、中高生向けのプログラミング教室 Bootstrap[*1]などで使用されている。Web 上で動く環境のため初期設定が不要である点や、Web 上にコードを保存できる点など、授業を効率行うための機能をいくつも備えている。そして WeScheme にはデザインレシピに従って書くという機能がある。これは、学習者がブラウザ上で関数の型や入出力の例を入力することがき、関数の型と入出力の例を書かないとコーディングが始められない。しかし、デザインレシピの他のステップへのサポートは WeScheme にはない。これは、使う対象が中高生ということもあり、主に単純なプログラムを書くことが想定されているためであると考えられる。また、Mio とは異なり、中間成果物があるかどうかの判断のみのため、成果物の内容に対してのフィードバックを返すことはない。

[*1] https://www.bootstrapworld.org

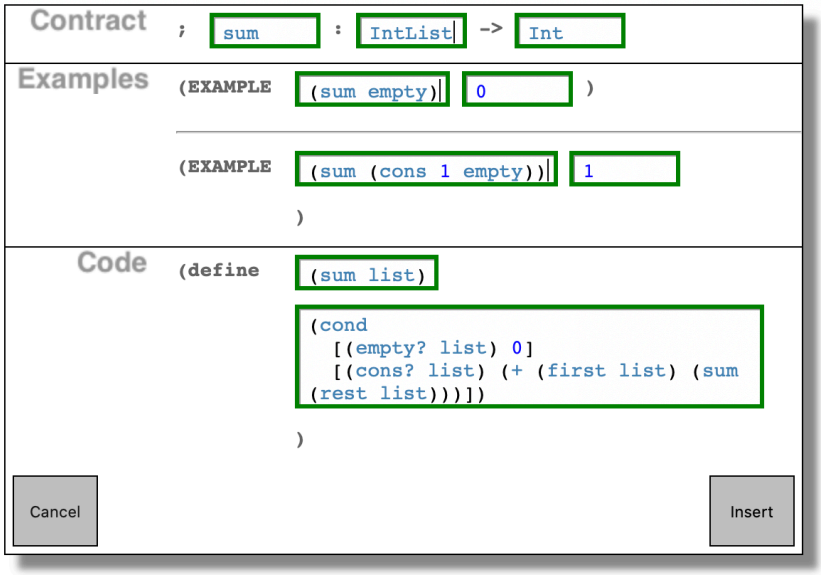

図 6.1: wescheme

■DRaCO　Ryu [6] の開発した DRaCO というプログラミング環境 (図 6.2) は、デザインレシピに従ったプログラミングを学生ができるようにすることが目的である。DRaCO では、プログラミングを始める前に契約文や目的文、テストケースなどを書くことが推奨されている。これも、推奨することのみで、全ステップに従わせる強制力や成果物への妥当性の判断などはない。そして WeScheme と同様、Mio とは異なり全ステップへのサポートはない。Ryu は DRaCO を用いて約 1 年に渡って実験を行った。プログラミングを教えるクラスで、DRaCO を用いたクラスとそうでないクラスに対して、テストの結果と学生の感想を統計的に分析している。その結果、Ryu はテストの結果に統計的に有意であると言い切れるほどの明確とした結果は出なかったものの、DRaCO を使ったクラスの方が点数が良かったことは事実としてあると述べている。これは母数が少なかったことも原因であると Ryu らは考えている。また、学生の感想の中で多くを占めたのは、慣れない環境を使うことへの戸惑いである。DRaCO や Mio のような学習をサポートする環境を教育に使う際には、そこへの戸惑いや不慣れさが学習の妨げにならないようにカリキュラムを考える必要がある。

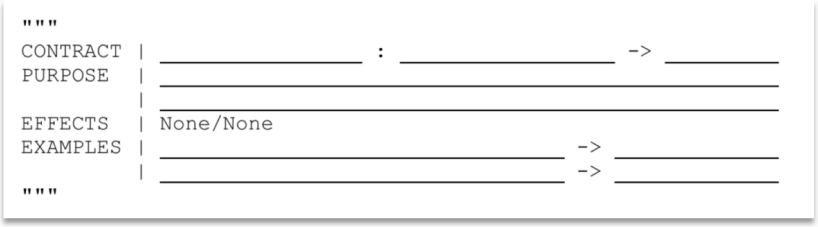

図 6.2: DRaCO

また、WeScheme や DRaCO はいずれもテキスト形式で書かせることが基本である。その点も、ブロックを用いている Mio とは異なる点である。

6.2　ブロックを用いたプログラミング環境

ブロック形式のプログラミング環境の代表例としては、Scratch [9] が挙げられる。Scratch(図 6.3) は、小中学生を主にターゲットとしており、プログラミングの初学者が最初に構文の書き方を覚える必要無く簡単に実行できることを目的とする。実際に Scratch は小学生のプログラミングへの心理的ハードルを下げた [10]、プログラミング学習へのモチベーションを上げた [11]、などの報告がある。ブロック形式であるというのは、初学者にとって心理的な面でも大きなメリットになるということが分かる。Scratch と Mio の異なる点は、Scratch はコーディングをブロックで行うように設計されたものであり、一方 Mio はコーディング以外の部分をブロックで組み立てるように設計したものである。

図 6.3: Scratch

6.3　ハイブリッド型プログラミング環境

Mio はデザインレシピのコーディング前のステップはブロックで、コーディング以降の部分はテキスト形式で実践するという点においてハイブリッドな環境である。Mio 以外にもブロックとテキストを共に使用するハイブリッドな環境は存在する。

Alrubaye ら [12] は pencil code を基にハイブリッドなプログラミング環境 (図 6.4) を作成した。これは、ブロック型のプログラミング環境で学んだ学生が、テキスト形式のプログラミグ環境へと移行する際にギャップを感じる問題を解決するために開発された。コーディングをする際にブロック (図 6.4 左側) をコーディング画面に持っていくことで、ブロックが自動でテキストコードに変換される。また、ブ

ロックを用いずにテキストとしてコードを書くこともできる。ここは Mio とは異なる点である。Mio では、ブロックで行う動作とテキストで行う動作を完全に分けている。これは Mio でブロックを使う目的が、コーディング以外の部分をデザインすることである。また、Alrubaye らはこのハイブリッドプログラミング環境を用いて実際に学生の授業で実験を行った。その中で、学生はブロックを構文を確認するためにも有効活用していたことが観察されている。これは Mio でブロックを使う利点と共通する部分である。

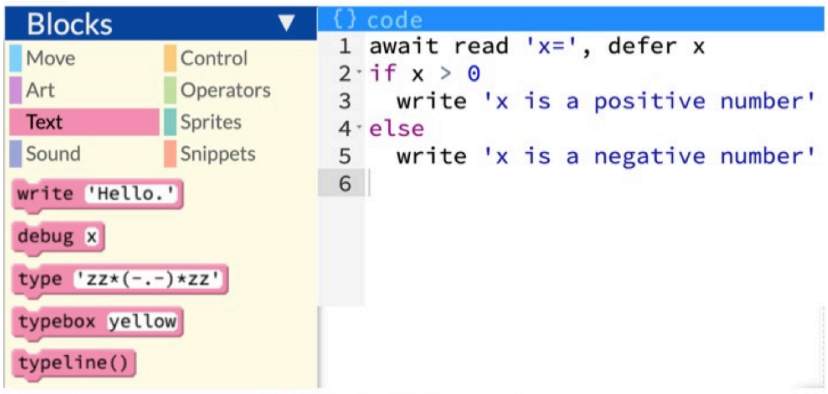

図 6.4: pencil-code の全体像 ([12] の figure1)

第 7 章

結論

7.1 まとめ

本論文では、デザインレシピに基づくプログラミング環境 Mio を提案した。全 6 ステップあるデザインレシピのうち、Step 1 から 4 まではデータ分析やデータ例などのコーディング前のステップである。Mio は、この Step 1 から 4 までをブロック形式で書かせ、それ以降のコーディングの部分はテキスト形式で書かせる、ブロック形式とコード形式のハイブリッドなプログラミング学習環境である。

ブロックを用いることで、自由記述となっていた部分に対し、学習者が新たな文法を覚える負担なく、決まった文法と言語を与えた。また、決まった文法を与えたことで、各ステップの成果物に対して機械的に妥当性を判断しフィードバックを機械で行うことができた。前のステップが完了しないと次のステップに進ませない、といった制約をもたせ、デザインレシピの各ステップに強制的に従わせることを可能とした。また、デザインレシピに基づいたプログラミング学習は、前後のステップを行き来することが考えられる。そこで、前のステップの修正が全体に与える影響を学習者が考えられるようにするために、Mio では前のステップでの変化がその後の既に完了しているステップにどのような影響を及ぼすのかを、取り消し/再実行を内部で実行することによって矛盾を検出することで解決した。

実際に How to Design Programs (HtDP) の例題を用いて自己評価を行った。Mio の恩恵として、ブロックであることによる有用性、学習者の負担の軽減が実感できた。一方で、見やすさの観点などの課題も発見できた。さらには、前後のステップの自由な試行錯誤の補助も充分に機能し、デザインレシピ実践の補助になることが実感できた。

7.2　今後の課題

7.2.1　ユーザ実験

自己評価では、各ステップのサポートと、前後のステップの試行錯誤の補助については実感できたが、フィードバックの有用性に関しては調べることができなかった。ここを重点的にユーザ実験を行う必要があると考えている。

ユーザ実験の対象者として、初学者、一通りデザインレシピについて学んだ学生、デザインレシピ実践に熟練した人の 3 種類を考えている。まず Mio は初学者を意識して作成している。そのため初学者にとって Mio が有用であるかどうかを知る必要がある。Mio はデザインレシピの実践に対して環境によるサポートを行なっている。そのため、従来の実践方法と比較した時の有用性を図るには、すでに実践したことがある人の意見も大切である。また、デザインレシピ実践に熟練した人はデザインレシピを実践していく中で今までも悩んだことが多くあると考えている。そこが Mio によってどの程度解決できるかを考え、さらには何が必要かを知る意味でも大切な対象者である。

実験方法としては、自己評価と同じく従来の実践方法と Mio で同じ問題を取り組んでもらう。その上で、各ステップのサポートや前後のステップの試行錯誤の補助が、従来の方法に比べてどの程度有用だと感じたか、またフィードバックの内容やタイミングが適切かどうか、などをアンケート形式で聞く。特にフィードバックに関しては、タイミングや内容などは学習者のレベルに合わせて変更可能にすると、より幅広い学習者に有意な環境になると考えられる。また、Mio の機能に依存しない評価項目も必要である。例えば、従来の実践方法と比べて手を煩わせる部分はあったか、などである。

7.2.2　様々な関数への対応

現状、構造体を用いた単純な関数にのみ対応しているが、HtDP [1] では他にも様々な複雑な関数のデザインレシピを紹介している。例えば、構造的再帰を持つデータ構造を用いる関数や、ネストしたデータ構造を用いる関数、アキュムレータを使用する関数などは、それぞれに適した各ステップのタスクがある。現状の Mio で用意しているブロックでは、それらのステップのタスクを行うには十分ではない。しかし、いずれもそれぞれのレシピに適したブロックを用意することで対応可能であると考えている。

例えば、構造的再帰を持つデータ構造を用いる関数では、図 7.1 のようなデータ定義が自己言及していることを明らかにして定義できるブロックや、図 7.2 のようにテンプレートに対して再帰呼び出しが起きうる場所に再帰関数の呼び出しが書けるようなブロックが必要である。

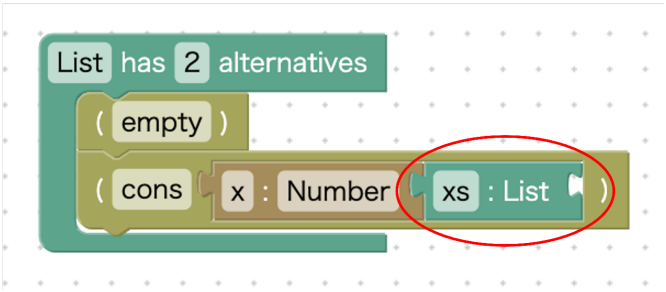

図 7.1: 自己言及を明示できるデータ定義ブロック

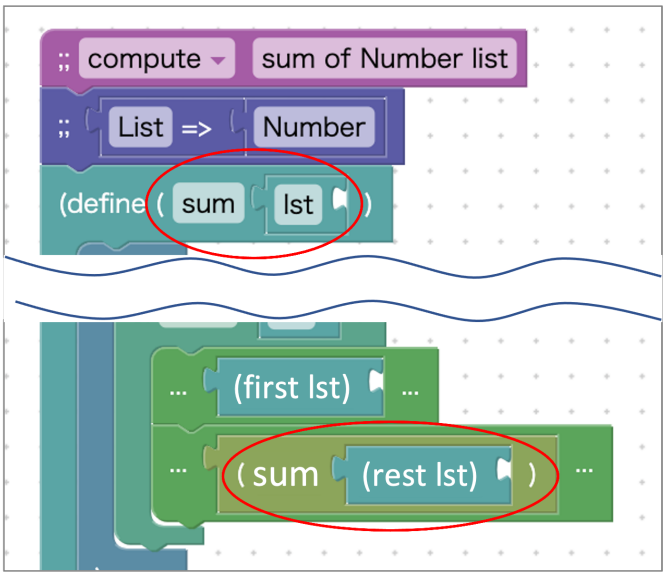

図 7.2: 再帰呼び出しを挿入できるテンプレートブロック

参考文献

[1] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: An introduction to computing and programming*. The MIT Press, 2001.

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The teachscheme! project: Computing and programming for every student. *Computer Science Education*, Vol. 14, No. 1, pp. 55–77, 2004.

[3] MATTHIAS FELLEISEN, ROBERT BRUCE FINDLER, MATTHEW FLATT, and SHRIRAM KRISHNAMURTHI. The structure and interpretation of the computer science curriculum s.

[4] Kathi Fisler. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*, pp. 35–42, 2014.

[5] Danny Yoo, Emmanuel Schanzer, Shriram Krishnamurthi, and Kathi Fisler. Wescheme: the browser is your programming environment. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pp. 163–167, 2011.

[6] Mike Dongyub Ryu. Improving introductory computer science education with draco, 2018. master's thesis, California Polytechnic State University.

[7] Blockly. `https://developers.google.com/blockly`, visited 2022-1-3.

[8] Norman Ramsey. On teaching *how to design programs*: Observations from a newcomer. *SIGPLAN Not.*, Vol. 49, No. 9, aug 2014.

[9] Scratch - imagine, program, share. `https://scratch.mit.edu/`, visited 2022-1-3.

[10] 山本利一, 鳩貝拓也, 弘中一誠, 佐藤正直. Scratch と wedo を活用した小学校におけるプログラム学習の提案. 教育情報研究, Vol. 30, No. 2, pp. 21–29, 2014.

[11] 岡崎善弘, 大角茂之, 倉住友恵, 三島知剛, 阿部和広. プログラミングの体験形式がプログラミング学習の動機づけに与える効果. 日本教育工学会論文誌, Vol. 41, No. 2, pp. 169–175, 2017.

[12] Hussein Alrubaye, Stephanie Ludi, and Mohamed Wiem Mkaouer. Comparison of block-based and hybrid-based programming environments in transferring programming skills to text-based environment. 06 2019.