平成 31 年度修士論文

お茶の水女子大学大学院 博士前期課程

人間文化創成科学研究科 理学専攻 情報科学コース

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

著者氏名 : 松本 晴香（学籍番号 1740664）

指導教官 : 理学部 情報科学科 准教授 浅井 健一

平成 31 年 3 月

要旨

本論文では，Google の提供するビジュアルプログラミングツール Blockly をベースとして，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た OCaml エディタ OCaml Blockly を実装した．Blockly では，構文木に対応したブロック型のオブジェクトを組み合わせることでプログラミングをするため，シンタックスエラーを防ぐことはできるが，セマンティックスにまで踏み込んでいないために型エラーなどの起きるプログラムを組み立てることができてしまう．本論文では，Blockly の直感的なプログラム体験を維持しつつ，不正なプログラムを構成することを制限したユーザインタフェースを提案する．let 多相を含む OCaml の静的型システムを実装し，結果の型や型変数を形や色として出力し，変数束縛やスコープ範囲を視覚的に表現した．また，ブロックと OCaml コードとの相互変換をサポートすることで，通常のテキストベースのプログラムへの導入となることが期待される．本論文では，OCaml Blockly の設計と実装についての詳細や，実使用に向けた課題について述べ，その考察を示す．

キーワード： ビジュアルプログラミングツール，OCaml，JavaScript，Blockly

Abstract

We present OCaml Blockly, OCaml visual programming editor based on Blockly. Blockly is a JavaScript library for creating visual block programming languages. Blockly makes it impossible to cause a syntax error since constructed set of blocks always represents an abstract syntax tree. However, Blockly does not deal with semantic errors, so it allows users to create program which does not compile.

OCaml Blockly provides UI which prevents users from constructing blocks with any compile errors. Blocks in OCaml Blockly change their colors and shapes using their data types inferred by its HindleyMilner type system. OCaml Blockly visually represents information of variables such as their types, bindings, and scopes. Additionally, we supports bidirectional converter between OCaml code and blocks in OCaml Blockly. It helps beginners get used to text-based coding. We show the design and implementation of OCaml Blockly and discuss future tasks toward practical uses.

Keywords: Visual programming editor, OCaml, JavaScript, Blockly

目 次

第1章　序論

関数型言語の初学者が関数型言語を学び始めたとき，数々の本質的でない問題に陥りやすい．変数に何らかの値を再代入しようとしてシンタックスエラーを起こしたり，Int 型と Float 型を持つ値を足し合わせようとして型エラーになってしまうことがある．関数型言語のシンタックスや静的型付けに慣れていない初学者にとって，テキストエディタは入力の自由度が高すぎるため，エラーの原因を特定するのはしばしば困難である．

プログラミング初学者がプログラミング学習のために用いるツールの 1 つにビジュアルプログラミング環境がある [1, 2, 3]．ビジュアルプログラミング環境では，テキストではなく，視覚的なオブジェクトを組み合わせることでプログラミングを行う．例えば，近松らの [4] は，変数や構文をノードで表し，ノードを線で繋いで値の入出力の関係を制御することによって，Haskell プログラムをグラフとして表現する．

一方で，Google の提供する Blockly [5] はブラウザにおけるビジュアルプログラミング環境構築のライブラリとして配布されているオープンソースプロジェクトであり，完全にクライアントサイドのみで動く．ブロックの描画や移動，組み立てなどといったユーザインタフェースに関わる実装はライブラリ側が全て行なっているため，ブロックに表示する文字や入出力の関係を定義するだけで，カスタマイズされたブロックを持つビジュアルプログラミング環境を作ることができる．実装言語は JavaScript であり，ブロックの描画には SVG や CSS が活用されている．また，Blockly の標準のブロックであれば，ユーザが組み立てたブロック群から，JavaScript，Python などといったスクリプト言語に変換することができる．

本研究では，既に洗練されたユーザインタフェースが用意されていることや，構文木と一対一対応をしていて，テキストを連想しやすいことから，Blockly をベースとして OCaml ビジュアルプログラミング環境を実装し，これを OCaml Blockly と名付けた．OCaml Blockly では，不正なプログラムを表すブロックを組み立てることをユーザインタフェースによって制限する．また，各ブロックが持つ型を形や色で表して，視覚的に出力する．関数型言語の初学者が，OCaml Blockly の元で OCaml プログラ

ミングを行うことで，以下の利点を得られることを期待している．

- OCaml のシンタックスに慣れる．
- 自身が組み立てようとするプログラムが不正なものであることに瞬時に気づくことができ，テキストベースよりもストレスが少なく，OCaml プログラミングの本質を優先的に学習できる．
- 静的型付けを理解する．

また，ブロックによるプログラミングによって OCaml の言語仕様に慣れ親しんだのち，シームレスにテキストベースによるプログラミングに繋がるように，本研究ではブロックと OCaml コードの相互変換の実装を行った．

本論文の構成を以下に示す．まず，**第 2 章**で Blockly 固有の用語を説明し，**第 3 章**にて関連研究とその問題点について分析し，その問題点を踏まえて本研究が将来的に達成すべき目標と，本論文で達成するべき目標を示す．次に，**第 4 章**で本論文が実装した主な機能を紹介し，その各機能の実装の詳細を**第 5 章**にて説明する．**第 6 章**でブロックと OCaml コードとの相互変換を行うためのシステムの構成を述べる．**第 7 章**で本システムの現状をまとめ，最終的な目標の達成のために実現されるべき課題を整理し，考察する．最後に，**第 8 章**にて本論文をまとめ，今後のさらなる展望について言及する．

第2章　Blocklyでの用語

本研究はBlocklyをベースにして行うため，次節以降には，Blocklyのユーザインタフェース固有の用語が出現する．それに先立ち，主要となる用語と概念を本節で共有する．

コネクタ：ブロックとブロックを接続する切り欠きを表すオブジェクト (図2.1)．コネクタには4種類あり，値の入力を受け取る入力コネクタ，値の出力を表す出力コネクタ，上向きまたは下向きに付くステートメントコネクタがある．図2.1の右にあるブロックが示すように，ブロックの移動時には，ユーザがブロックを接続しやすいように，近くにある接続可能なコネクタがハイライトされる．

フライアウトメニュー：ブロックを新しく生成するためのメニュー (図2.2)．定義されたブロックが並んでおり，ドラッグをすると新しいブロックをそのまま得ることができる．

ワークスペース：ブロックの組み立てやドラッグ移動などを自由に行うことのできる空間のこと．空間ごとの座標系や拡大縮小の制御，発火イベントの管理などを担っている．ワークスペースはブロックと同様に，SVGエレメントとしても実態を持っており，ブロックを表すSVGエレメントはこのワークスペースの空間を表したSVGグループの中に内包されている．各ブロックは必ず1つのワークスペースに属している．

メインワークスペース：フライアウトメニューから生成したブロックを自由に組み立てることのできる一番根本にあるワークスペースのこと (図2.2)．

図 2.1: Blockly 上でのブロックの例．各ブロックに外向きについている切り欠きの部分がコネクタである．

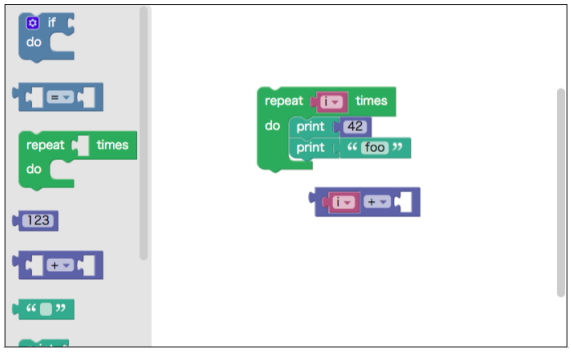

図 2.2: Blockly でのフライアウトメニュー，メインワークスペースを示したもの．左側の背景色が灰色の部分がフライアウトメニューであり，右側の背景色が白色の部分がメインワークスペースである．

第3章　関連研究

Blockly に型付けを行った関連研究に，Lerner らによる *Polymorphic Blocks* [6] がある．ターゲット言語は OCaml ではなく，擬似的な言語を対象にしている．Int 型や Bool 型といったプリミティブな型ごとにブロックのコネクタの部分の形を変え，Function 型や List 型といったパラメタ付き型についても再帰的に型の形を描画し，型変数については各コネクタを固有の色でハイライトすることにより，型変数を含めた型をわかりやすく視覚的に捉えられるようになっている (図 3.1)．型の視覚化を実現するために用いられているテクニックについて説明する．まず，ブロックの入出力を表す各コネクタに型表現を添付する．Int 型ならば，出力コネクタに Int の型表現が加えられ，List 型ならば，各入力コネクタに任意の型変数表現 α が，出力コネクタにはその型変数 α をパラメタとして持つ List 型表現が保存される．型変数にはそれぞれランダムに生成した色を持たせておく．ある 2 つのコネクタがお互いに接続するときには，それぞれのコネクタについた型表現の単一化を行う．あとはブロックの描画時に，コネクタに添付された型が指す表現ごとに別の形や色を描画すればよい．これにより，ユーザの動作に基づいて動的に変わる型をブロックの上に描画することを実現している．

しかし，[6] では，不正なプログラムの組み立てを許してしまう (図 3.2)．また，一度単一化された型はブロックを外しても元に戻らないことや，変数宣言の構文，let 多相 [7] をサポートしていないなど，本研究が求める関数型言語初学者向けのツールとして活用するためには不完全な部分が多く見受けられる．

他の関連研究としては，FunBlocks [8] がある．これは CodeWorld [9] という Haskell プログラミングの学習環境を Blockly 上に実現したものである．[6] と同様に型や型変数を視覚的に描画する方法を取っているが，let 多相やブロックを外したときの型の単一化の解除を行うなど，型システムは [6] よりも拡張されている．一方で，スコープという概念がないために図 3.2 と同じような不正なプログラムを表すブロックが組み立てられてしまう．また，CodeWorld 専用のプログラミング環境であるために，標準の OCaml エディタとして開発を行う本研究とは目的が異なる．

これらの点を踏まえて，本研究が OCaml Blockly を開発する上で最終的に達成し

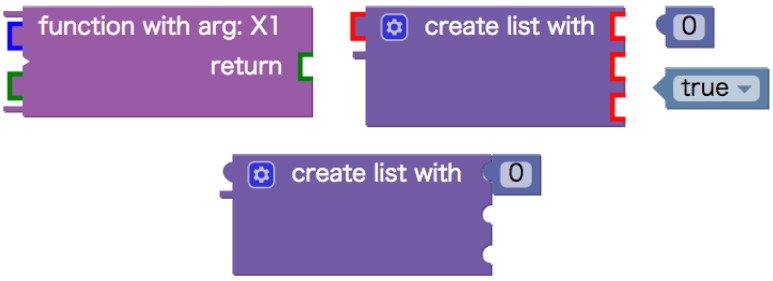

図 3.1: *Polymorphic Blocks* でのブロックの例．始め List 型のパラメタの型は決定していないため，型変数を意味するハイライトが赤色で示されているが (中)，Int 型のブロックを接続させると，ハイライトが消え Int 型の形に変更される (右)．

図 3.2: *Polymorphic Blocks* で組み立てた 2 つのラムダ式ブロック．左のブロックはラムダ式として正当だが，右のブロックでは入れ子になったブロックが表す変数 X1 が未定義なってしまう．

たい目標は以下の 2 点である．

本研究の掲げる最終的な目標

1. 「穴のないブロックを組み立てられた」ならば「コンパイルエラーの起きないプログラム」であることを保証する．
2. 関数型言語初学者向けの授業で使えるクオリティにする．

まず，1 つ目の目標により，コンパイルエラーの起きるプログラムを組み立てることを制限したユーザインタフェースを実装する．この保証による利点は第 1 章で述べた通りである．

続く 2 つ目の目標を達成するためにまず不可欠なことは，豊富な構文を揃えることである．また，実際に授業で使用するためには初学者の視点から見た「使いやすさ」「理解しやすさ」といった高いユーザビリティの達成は欠かすことができない．そのためには，本来の Blockly にある直感的でわかりやすいユーザ体験を維持するだけでなく，OCaml の言語仕様に特化したブロックのビジュアルを設計し，ブロックの表現を改造する必要がある．

一方で,「不正なプログラムを組み立てられない」という制約の元に，授業で使う OCaml の全ての言語仕様をブロックで表現し，なおかつ初学者が混乱しないユーザビリティを保つためには，多くの実装量が必要であることはもちろん，実際に授業でユーザに体験してもらい，フィードバックをもらう必要がある．そこで本論文では，最終的な目標を見据えた上で，その proof of concept となる目標を実現する．この目標は以下に示す通りで，それぞれの目標が最終的な目標のサブセットとなっていることに注意されたい．

本論文で実現される目標

1. シンタックスエラー，型エラー，Unbound value エラーが起きないことを保証する．
2. 元の Blockly の使いやすさを維持しつつ，OCaml に慣れるために必要な基礎的な構文を備える．

一概に「コンパイルエラー」と言ってもエラーの種類には様々なものがあるが，主なものは上で述べた 3 つである．本論文では，この 3 つのエラーを起こすプログラムを構成することができないようなユーザインタフェースを構築する．

構文のカバー範囲については，OCaml を使用言語とした「プログラミングの基礎」[10] の前半部分で扱われるような，再帰関数，match を用いたリストのパターンマッチなどを目標とし，それらを「不正なプログラムを組み立てられない」という制約の元に実装する．

なお，[6] で行われた，型をブロックのコネクタの形や色で表現するアイデアや一部の実装は継承する．実装は，本来の Blockly に合わせて JavaScript で行う．

第4章　主な機能

前節で宣言した通り，本論文で防ぎたいコンパイルエラーの種類は以下の 3 つである．

- シンタックスエラー
- 型エラー
- Unbound value エラー

シンタックスエラーについては，ブロックと構文木が一対一対応しているため，完成したブロックにおいては起き得ない[1]．型エラーについては，ブロックの各コネクタに型表現を割り振るという第 3 章で述べた [6] のアイデアを参考にして，let 多相や型の単一化のリセット機能を実装し，型推論を拡張する．接続させると型推論が失敗するようなコネクタをユーザが接続させようとしたら，ブロックの接続を拒絶すればよい．最後に，Unbound value エラーを防ぐために，変数ブロックの生成や，変数ブロックを含んだブロックのスコープを管理する．上に示した 3 つ以外を理由としたコンパイルエラーについては，第 7 章にて議論する．

本研究で行った主な実装のうち，以下の 4 つの機能をそれぞれ画像を交えて説明する．

1. 変数束縛
2. スコープお砂場
3. let 多相を含んだ型システム
4. メッセージ出力

実装方法については，次節にてそれぞれ詳しく説明する．

[1] ブロックにまだ接続されていないコネクタがある場合は別である．

4.1　変数束縛

変数スコープや変数束縛の関係は，関数型言語初学者にとって，理解しにくいものの 1 つと言える．多くの命令型言語では，変数宣言文と同列にある閉じ括弧 } が出現するまでの文がスコープとなるのに対し，関数型言語では，OCaml や Haskell の let のように，in 内の式が終了するまでがスコープとなる．このようなスコープの概念に馴染みの無い初学者は，例えば「`(let foo = ... in ?); foo`」のような，スコープ外で `foo` にアクセスするようなミスをしがちである．

初学者が OCaml におけるスコープの概念を理解しやすいように，変数参照は必ず変数宣言から生成させる．これによって未定義の変数を参照することが原理的に起きえなくなる．また，一度生成した変数参照はその参照先を配置された位置によって変えない．これによりユーザの想定していなかったシャドウイングによってプログラムの意図が変わることを防ぐ．変数参照を表す変数参照ブロックを以降，単に「変数ブロック」と呼ぶ．変数ブロックの生成の流れを図 4.1 に示し，変数ブロックをドラッグしたときの様子を図 4.2，4.3 に示した．

例えば，図 4.2 でユーザが組み立てようとしているプログラムは，左から順に「`(let foo = ? in ?); foo`」,「`let foo = foo in ?`」,「`let foo = ? in foo`」であるが，初めの 2 つはプログラムとして正当でない．よって，ドラッグ中のブロック全体を不透明にすることによって，目的の位置にブロックを移動できないこと，あるいは目的のコネクタにブロックを接続できないことをユーザに伝える．

図 4.3 は，ユーザがどこから変数ブロックを生成したかによって変数ブロックの扱いが異なることを示した例である．組み立てようとしているプログラムはどちらも「`let foo = ? in let foo = ? in foo`」であり，OCaml のプログラムとしてはどちらも正当だが，ユーザが意図した変数を正しく参照することのできない左のケースでは，コネクタとの接続を拒否する．

なぜそこにブロックを置くことができないのか，接続できないのかといった理由を説明したエラーメッセージの出力も行っており，これについては第 4.4 節にて紹介する．

正しく束縛されない変数ブロックが存在したままユーザがその場にブロックを落とした場合は，ブロックが生成されたばかりならばブロックを削除し，そうでないならドラッグ開始前の位置，状況に戻らせる．また，変数をホバーしたときに関連した全ての変数がハイライトされるようになっており，束縛関係を把握しやすいようになっている (図 4.4)．スコープに従ったアルファ変換 (変数名の変更) も行うことができる．

4.2　スコープお砂場

変数ブロックや変数ブロックを含むブロックをスコープが有効な場所にしか置けないという制約を設けると，ブロックを自由な場所に置き，好きな順番でブロックを組み立てられるという，本来の Blockly にあった利点を損なってしまう．これを避けるため，本研究では，スコープチェックのついた遊び場を設け，これを「スコープお砂場」と命名した．使用例を図 4.5 に示す．

スコープお砂場は新しい変数が加わりうる各入力コネクタに紐づいており[2]，コネクタの隣にあるアイコンをクリックするとポップアップの形で表示される．所属先のコネクタから参照できる変数の一覧がお砂場内のフライアウトメニューから見られるようになっていて，そこから変数ブロックの生成も行える．ユーザはこのスコープお砂場でブロックを適当に組み合わせたのち，目的のブロックと接続をすればよい．スコープお砂場は，入れ子にしても使うことができる．

このお砂場にはスコープチェックがついており，所属先のコネクタから参照することができない変数を含むブロックはお砂場に移動することができない．例えば図 4.5 の右下のお砂場にある「`foo`」ブロックはもう一方のお砂場に移動できるが,「`bar + ?`」のブロックは移動することはできない．ブロックが不透明になり，ブロックをそのままリリースしてもブロックの移動はキャンセルされる．

また，お砂場を所有するブロックが移動する際は，お砂場内のブロックもスコープチェックの対象である．例えば，図 4.5 の「`let bar = ? in ?`」を表すブロックを動かして「`let foo = ? in ?`」の 2 つ目のスコープから外した場所に置くことはできない．お砂場内に変数ブロック「`foo`」が存在しているためである．

4.3　let 多相を含んだ型システム

let 多相を含む型システムを実装した．また，型の単一化の解除をサポートした．例えば，リストコンストラクタのブロック「`? :: ?`」に int 型ブロックを入れて「`3 :: ?`」とすると，出力コネクタの型が`'a list` から `int list` へと更新されるが，`int` 型ブロックを外すと再び`'a list` 型に戻る．

同じ変数の型は，例えば図 4.5 の変数「`bar`」のように，スコープお砂場に置かれたブロックであっても一貫していなければならないとした．例えば，スコープお砂場の

[2]図 4.5 では，in の直前のスコープにもお砂場のためのアイコンが表示されているが，これは let に引数を追加する場合や rec フラグを有効にした場合に必要になる．

外で宣言された変数を参照する変数ブロックが，スコープお砂場内で `float` 型として扱われたならば，元の変数宣言も `float` 型であると決定される．

4.4　メッセージ出力

不正なプログラムを組み立てられないユーザインタフェースを設けると，なぜ不正なのかをまだ理解できない初学者が困惑してしまう可能性があり，本研究が達成するべき「理解しやすさ」が不十分になってしまう．これを回避するため，本システムではドラッグ中のエラーを書いたツールチップの表示を行った (図 4.6)．なぜユーザの意図した場所にブロックを置くことができないのか，なぜ目的のブロックと接続できないのかといった理由を見ることができる．エラーのフォーマットは，本来の OCaml のエラー出力になるべく従っている．

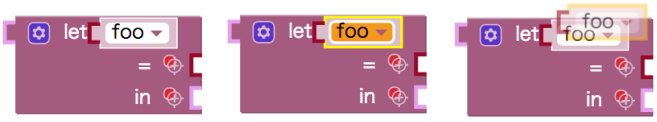

図 4.1: 変数ブロックを生成する流れ．変数宣言を囲うように描画されたブロック (左) にカーソルを合わせて (中) ドラッグする (右) と，変数ブロックの生成を行うことができる．

図 4.2: 変数ブロック「`foo`」を生成後にドラッグしたまま動かしたときのブロックの様子．正しく変数ブロックが束縛されない場合はブロックを不透明に表示する．

図 4.3: 外側の変数宣言部分から生成した変数ブロック (左) と，内側から生成した変数ブロック (右) を内側の in 以下に接続させようとしたときの比較．変数ブロックのコネクタの色にも注目されたい．

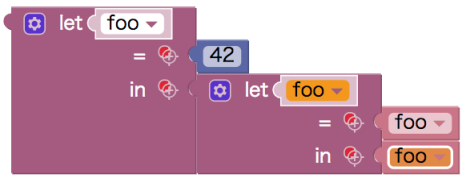

図 4.4: 右下の変数ブロックをホバーしたときの反応．束縛変数，束縛されている変数がオレンジ色にハイライトされる．

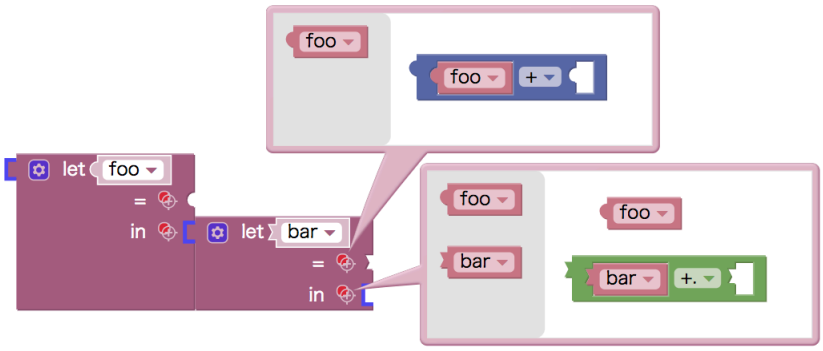

図 4.5: スコープお砂場の例．ライフルのスコープを模したアイコンをクリックするとポップアップが現れる．スコープで参照できる変数ブロックの一覧が見れ，ブロックの生成も行える．スコープお砂場内で，変数ブロックをどんな型として扱うかによって，変数宣言のブロックの型も変化する．

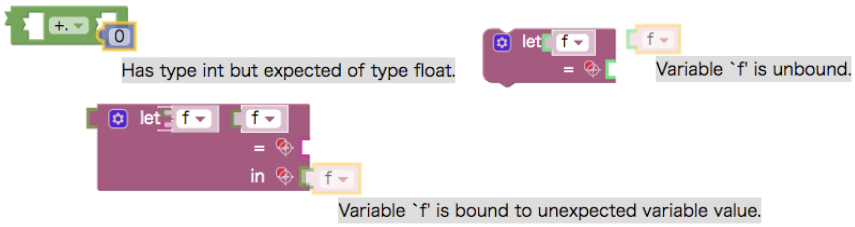

図 4.6: エラーツールチップの例．

第5章　実装方法

前節では，OCaml Blockly が 4 つの主な機能をユーザから見た視点で紹介した．本節では，それぞれについて実装方法を詳細に説明する．本研究は Blockly に加え [6] もベースにして行っているため，それぞれの実装も交えて説明を行う．

5.1　変数束縛

まず，Blockly での変数ブロックの実装について触れる．Blockly では，新しい変数の追加をフライアウトメニュー内にあるメニューボタンから行う．既に存在する変数名の追加は行えない．変数を追加すると，図 5.1 のような変数フィールドを持った数種類のブロックがフライアウトメニューに追加される．変数フィールドとは，図 5.1 の変数名 i を囲う薄いピンク色の部分であり，クリックすると変数名の変更などが行えるブロック内のエレメントである．各変数フィールドは，内部的に変数インスタンスを保持している．例えば，図 5.1 に示した 3 つの変数フィールドは，1 つの変数インスタンスを内部的に共有している．そうすることで，変数名を変更したときに，同じ変数を参照する変数フィールドの変数名が同期的に更新される．しかし，この Blockly での変数の扱いは，関数型言語での immutable な変数やシャドウイングをわかりやすく表現するためには相性が悪い．Blockly にはスコープという概念がなく，全ての変数はグローバルな変数として扱われているため，同名の複数の変数を扱うことができないためである．

図 5.1: 本来の Blockly にて，ブロック内に変数フィールドを持つ 3 つのブロックの例．

束縛変数とその束縛関係を表現するには，変数インスタンスの種類を「宣言される変数」と「宣言される変数を参照する変数」に分けると都合がよい．参照する変数は，

自身がどの変数を参照しているのかを保持し，変数名を取得したいときや変更したいときはそれに問い合わせれば，変数名が変更されたときの各変数フィールドの更新を元の Blockly と同様にシンプルに行うことができる．

実装の詳細を説明する．1 つの変数は，1 つの束縛変数宣言と，0 個以上の変数参照と関連付けられる．この変数宣言と変数参照を表現するため，以下の 2 つのクラスを定義した．

`BoundVariableValue` 束縛する変数，宣言される変数．

`BoundVariableValueReference` 束縛される変数，宣言された変数を参照する変数．

Blockly の変数フィールドは束縛変数フィールドに拡張され，それぞれの束縛変数フィールドは，それが変数宣言であるか変数参照であるかによって，2 つのうちどちらかのクラスのインスタンスを持つ．

これらのクラスを用いて，スコープチェックを以下のように実装した．変数宣言を持つブロックは，スコープを作るコネクタを引数として，スコープ内で参照できる変数の一覧を返す関数を定義する．この関数を用いて変数環境を更新して，スコープ内のブロックに存在する全ての変数参照が変数環境にあるかどうかを調べれば良い．この作業を再帰的に行えば，ネストしたブロック全体のスコープチェックが行える．ユーザがある 2 つのコネクタを接続しようとしたときは，接続を仮定したスコープチェックを行い，失敗した場合は接続を拒絶する．

次に，束縛変数フィールドの描画について説明する．変数宣言か変数参照かどうかによって束縛変数フィールドの描画の仕方を変える．変数宣言の場合は，ブロックの形を模した SVG を背景に描画し，変数宣言部分から変数ブロックの生成を行えることを表現する．そして，実際にドラッグされたときにその変数への参照を持つブロックを生成する．束縛変数フィールドをホバーしたときに，関連した全ての変数フィールドをハイライトする機能は，変数宣言が自身を参照する変数の一覧を保持することによって行える．変数宣言に問い合わせて関連した全ての変数を取得したら，それら変数を内包する各フィールドの背景色を変える．

5.2　スコープお砂場

スコープお砂場のようなメインワークスペースとは独立した作業空間を実現するには，まずワークスペースをもう 1 つ作ればよい．そうすることによって，メインワー

クスペースとは分離した別のブロックを組み立てる世界を作ることができる．

スコープお砂場内でのスコープチェックはメインワークスペースとは異なる．メインワークスペースでのスコープは空の変数環境であるが，お砂場でのスコープは所属先のスコープで使える変数を全て含んだ変数環境である．このお砂場が持つスコープは，お砂場を所有するブロックを上位から辿っていくことにより取得する．これは，構文木を上から辿って変数環境を更新していくインタプリタと似た要領である．お砂場が入れ子になっている場合は，親のお砂場が持つスコープも同様に取得する．ユーザがブロックをお砂場の上にドラッグしてきたときには，移動中のブロック内の自由変数が全てお砂場のスコープで有効かを調べる．そうでない場合は，お砂場内で参照できない変数を含んでいるときなので，ブロックのお砂場への移動を拒絶する．

また，スコープから参照できる変数ブロックを並べたフライアウトメニューをお砂場に実装した．これによって，ユーザはお砂場内で使用できる変数の一覧を知ることができ，また，変数参照ブロックの生成も直接そこから行うことができる．スコープお砂場を所有するブロックの移動によって，お砂場の暗黙的な変数環境が変わるときは，即時的にフライアウトメニューを更新する．

お砂場にブロックを持ちよって自由に組み立たり，その後に別のワークスペース内のブロックと接続させたりするには，ブロックが異なるワークスペース間を移動できるようにする必要がある．本来の Blockly では，ブロックの自由な組み立てはメインワークスペースでしか行われないため，ワークスペースを越えたブロックの移動は想定されていない．このため，本システムではブロックのワークスペース間の移動を実装した．実装の方法は主に 2 つあると考えられる．

1 つの方法はブロックの SVG を移動先のワークスペースの SVG グループに移動させ，ブロックオブジェクト内の古いワークスペースへの参照を全て更新することである．この方法はあまり現実的でない．なぜならワークスペースと繋がりを持つのはブロックのみでなく，コネクタやフィールドなどといったブロック上に存在する不特定多数のコンポーネントも，ワークスペースを参照し合ったり，イベントリスナーを登録したりするなどしてワークスペースに依存しているためである．それら全てのワークスペースへの依存を完全に正しく更新するのは煩雑であり，将来の拡張のたびに変更が必要な，技術的負債となりうる．

もう 1 つの方法は移動先に新しいブロックを作り直し，古いブロックを削除することである．この方法は， Blockly に元々あるブロックのエンコード，デコード機能をうまく使うことによって，比較的容易に確実な実装をすることができる．

XML を利用したブロックの再構築

Blockly では，ブロックを XML にエンコード，XML からブロックにデコードできるようになっている．このブロックと XML の双方向の変換は，従来の Blockly にていろいろな場面で活用されている．例えば，デモページを作るとき，フライアウトメニューにどの種類のブロックを備えるかは XML で指定することになっている．指定する XML を入れ子にすれば，接続した複数のブロックをフライアウトメニュー上に並べることもできる．

本研究では，このブロックと XML の双方向変換を利用して，ワークスペース移動時のブロックの再構築を行なった．移動したいブロックを XML にエンコードし，その XML から新しいブロックをデコードして得る．このとき，気をつけるべきことが 2 つある．1 つはブロック内の変数の参照関係が新しいブロックでも保たれていること，もう 1 つは古いブロックにあった全ての型表現を新しいブロックへと移し替えることである．

まず，作り変えの後も変数の参照関係を保つために，エンコード時にはブロック内の変数参照がどの変数を参照していたかという情報も XML に落とす必要がある．この変数の参照関係のエンコードは，自由変数であるものに対してのみ行う．そうでないと，新しいブロック上の変数が古いブロック上にある変数を参照してしまうことになり，それはワークスペース間の移動後に削除されてしまうからである．エンコードされなかった変数の参照関係は，ブロックを組み立て終えたのちに，各スコープの変数環境に従って自動で復元する．

次に必要なことは古いブロックにあった型表現を全て新しいブロックの同じ位置のコネクタに移し替えることである．これは型変数の固有の色をブロックの作り直しの前と同じ色に保つためであり，型表現の移し替えを行うことで，ブロック上の型変数が既に何か具体的な型に単一化されていたとしても，接続を解除したときに元の色に戻れるようにすることができる．

なお，XML をデコードして新しいブロックを組み立て直す最中はブロックに対するスコープチェックを無効にする．ブロックは深さ優先で組み立てられていくが，例えば「`let x = ... in x + 42`」の足し算ブロック部分「`x + 42`」を組み立てる最中に，x がスコープにない変数になってしまうからである．また，ブロックに付属したスコープお砂場も同様に新しいブロックへと移動させるが，その説明はここでは割愛する．

5.3　let 多相を含んだ型システム

Blockly は型を文字列による名前で表現することによって，簡単な型チェックをサポートしている．各コネクタに型の名前を保存することができ，型が設定されている場合は，同じ名前の型を持つコネクタとしか接続できないようにすることで，単純な型付けを行なっている．

型を文字列で区別してしまうと List 型などのパラメタ付き型を表現することは難しいが，[6] では各コネクタに文字列の代わりに型変数などの型表現を加えることによってパラメタ付き型をブロック上に実装している．型の単一化が可能な型表現を持つコネクタ同士でしか接続できないようにし，接続する際には 2 つのコネクタが持つ型表現を単一化する．一方で，[6] では連結させたブロックを外したとき型の単一化が解除されないことや，構文に沿った型推論，let 多相が実装されていないなど，拡張が必要な部分がある．

型の単一化の解除

ユーザがブロックを外した際に型の単一化の解除を行うには，コネクタに添付した型変数の単一化の関係を全て削除したのち，構文の意味に沿ってもう一度型推論を走らせることが必要である．新しい型変数を振り直すことも単一化解除の 1 つの方法だが，本システムでは型変数を固有の色で表現しており，解除後にも同じ色を保つ必要があるため，型変数の振り直しは行わない．型付きブロックのクラス定義それぞれに，以下の 2 つインスタンス関数を定義する．

`clearTypes()` ブロックに存在する全ての型変数の単一化の関係を削除する．

`inferTypes(env)` ブロックの構文に沿って型推論を行う．引数 `env` は型環境であり，変数名をキーとして型スキームを値として持つオブジェクト．返り値としてブロックの出力の型を返す．

ある 2 つのブロックの接続が外れたとき，全ブロックのトップのブロックから `clearTypes`，`inferTypes` を順に実行すればよい．

let 多相の実装

let 多相を実装するには，ブロックの接続解除時だけでなく，接続時にもトップのブロックから型推論を実行する必要がある．値制限を設けているためブロックの付け替

えによって単相，多相との切り替えが起こりうるためである．また，スコープお砂場にあるブロックにも型推論を行うことを考えると，型推論を行うブロックの順番も重要である．束縛型変数を持つ型スキームの場合，let で宣言した変数の型スキームの解決は，必ずスコープお砂場にあるブロックに型推論を走らせるよりも前に行わなくてはならない．スコープお砂場に let で宣言した変数に束縛された変数ブロックに型推論を走らせるとき，その型スキームが必要になるからである．

5.4　メッセージ出力

第 4 章で述べた通り，本研究で防ぐ対象とするエラーは，シンタックスエラー，型エラー，Unbound value エラーの 3 つである．シンタックスエラーはブロック上では起き得ないとして，ドラック中のメッセージ出力は型エラー，Unbound value エラーの 2 つに対して行うとする．

ユーザになぜ期待通りの動作が行えないのかを説明するべきシーンは，以下の 2 つがある．

- 接続してしまうと型エラーあるいは Unbound value エラーが起きてしまうコネクタ同士を繋げようとしたとき．
- 変数を含んだブロックを変数環境に正しく束縛されず，Unbound value エラーが起きてしまう場所にドラッグしたとき．例えば，「`x; let x = 5`」など．

1 つ目におけるエラーを取得するためには，接続可能なコネクタを探索する処理を拡張する．元の Blockly には，近傍にあるコネクタの中から接続可能なものを探す処理がある．この接続可能かどうかの判断は OCaml Blockly のために拡張されたため，第 5.1 節で触れたようなスコープチェックや，型チェックが含まれている．近くのコネクタが全て接続可能でない場合，最も近いコネクタに対するエラーメッセージをツールチップに表示する．このエラーメッセージは，Unbound value エラー，型エラーに関する詳細をユーザに通知する．

2 つ目は，ブロックを接続させずに移動させる動作になる，すなわち，十分に近いコネクタが存在しない場合である．この場合は，ドラックされているブロックに対してスコープチェックを行い，正しく変数環境に束縛されない変数が存在するならば，そのエラーメッセージを取得し，表示すればよい．

第6章　ブロックとOCamlコードとの相互変換

本節では，ブラウザ上でのブロックと OCaml コードの相互変換の実現について説明する．組み立てられたブロックから OCaml コードを生成することで，本来のテキストによる OCaml プログラミングへのイメージが湧きやすくなる．逆に OCaml コードからブロックへの変換を実現することで，型や変数の束縛関係を視覚的に確認できることになり，テキストベースのプログラミングに移行しつつある初学者にとっての補助となることが期待できる．また， OCaml コードから複雑なブロックの組み合わせを復元できるので，OCaml Blockly の開発時に，特定の条件下の UI/UX をテストしたいときに使うこともできる．双方向の変換について，それぞれ説明を行う．

6.1　ブロックからOCamlコードへ

第 1 章で述べた通り，Blockly は標準のブロックに対して JavaScript や Python などのスクリプト言語へのコード生成をサポートしている．それらの実装と同じような手順で，本研究で新たに追加した OCaml ブロックに対する OCaml コードの生成を実装した．ブロックは構文木と一対一対応しているので，OCaml コードへの変換は比較的自明である．

6.2　OCamlコードからブロックへ

第 5.2 節で述べた通り，Blockly はブロックと XML のエンコード，デコードをサポートしている．つまり，OCaml コードをブロック表現の XML に変換することができれば，ブロックを生成することができる．OCaml コードをパーズするために，OCaml コンパイラのパッケージ compiler-libs を利用した．入力された OCaml コードから compiler-libs によって抽象構文木を得たのちは，木をトラバースして対応するブロックの XML を組み立てる．これらをビルドしたものを js_of_ocaml で JavaScript へと

変換すれば，ブラウザ上で OCaml コードからブロックへの変換が実現できる．変換するコードの例を図 6.1 に，コードをブロックに変換したものを図 6.2 に示す．

```
let rec pi_impl n d =
  if n > 0.0
  then n *. n /. d /. (d -. 2.0) *. pi_impl (n -. 2.0) (d -. 2.0)
  else 1.0;;
let pi n = 2.0 *. pi_impl (n *. 2.0) (n *. 2.0 +. 1.0);;
let pi_exp = pi 9000.
```

図 6.1: ウォリス積を利用して円周率を計算する OCaml プログラム

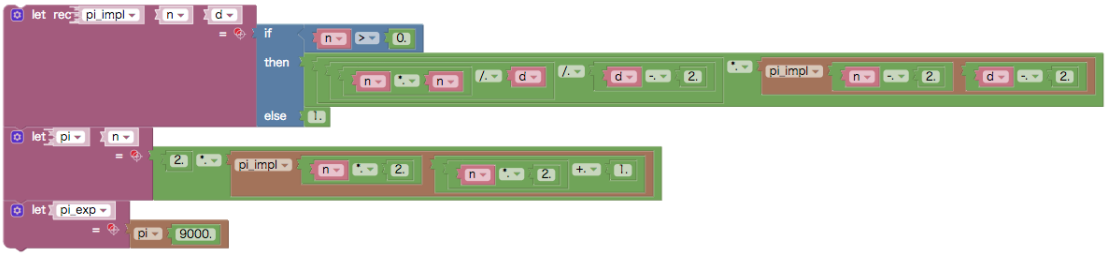

図 6.2: 図 6.1 に現したウォリス積を利用した OCaml プログラムをブロックに自動で変換したもの．型の整合性，変数束縛ともに反映されている．

第7章　現状と実使用に向けて

7.1　現状

シンタックスエラー，型エラー，Unbound value エラーが起きないという制約の元に，OCaml に慣れるために必要な基礎的な構文を備えた．現在 OCaml Blockly でサポートしている構文は，以下の通りである．使用例を図 7.1 に示す．

- `bool` 型，`int` 型，`float` 型，`string` 型．
- 2 つ組のみのタプル，リスト，ラムダ式，関数適用．
- 論理演算，数値演算，`fst`，`snd` などのビルトイン関数．
- リストのパターンマッチを行う `match` 文．
- `in` を持つ `let`，`in` を持たない `let`，`let rec`(どの `let` もコンテキストメニューから変更可．)．
- 1 引数のみのコンストラクタ定義．

7.2　実使用に向けて

第 3 章にて掲げた最終的な目標は以下の 2 つであった．

1. 「穴のないブロックを組み立てられた」ならば「コンパイルエラーの起きないプログラム」であることを保証する．
2. 関数型言語初学者向けの授業で使えるクオリティにする．

本論文では，この最終的な目標を見据えた上で，その proof of concept となる目標を達成することができたと考えている．今後は，現状の OCaml Blockly の元にさらなる実装やテストを重ね，最終的な目標を達成していきたい．そのために具体的に何を行なっていくか，どんな困難があるのか，といった点を本節で整理し，考察する．

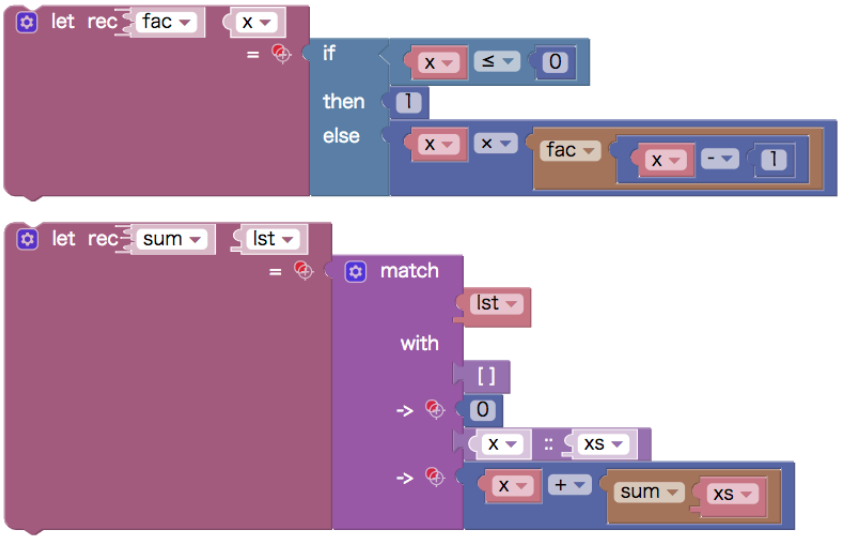

図 7.1: OCaml Blockly でブロックを組み立てた例．整数を受け取りその階乗を返す関数 fac (上) と，整数の入ったリストを受け取りその合計を返す関数 sum (下)．

コンパイルエラーの起きない制約

本論文では，シンタックスエラー，型エラー，Unbound value エラーを主要なコンパイルエラーとし，この 3 つのエラーを起こすプログラムの組み立てを制限したユーザインタフェースを実装した．Unbound に関するエラーとしては，以降レコード型などの構文を追加していくに当たって，Unbound field などといったエラーが起きうることが考えられるが，これは Unbound value エラーを防いだ実装と同じような要領で回避することができる．

一方で，標準の OCaml で起きうるコンパイルエラーはシンタックスエラー，型エラー，Unbound エラーの 3 つだけではない．例えば，「`let rec f = f`」というプログラムは，2 つ目の f の出現を指して，「`This kind of expression is not allowed as right-hand side of ‘let rec’`」というコンパイルエラーになるし，「`match ... with x :: x -> ...`」といったプログラムは 2 つ目のパターン変数 x を指して「`Variable x is bound several times in this matching`」というエラーになってコンパイルが失敗する．

これらのコンパイルエラーを起こすプログラムをブロック上で組み立てられないよう，OCaml Blockly 上でのエラーのチェックを拡張していく必要があるが，その方法は主に 2 つある．

1 つは，各種エラーを逐一対応していき，アドホックな変更を重ねることである．この方法は，数種類のエラーだけを対応するならば，手っ取り早く確実であるが，全てのコンパイルエラーを防ぐためには，スケールしない方法と言える．OCaml Blockly は OCaml という膨大な言語処理のとても局所的な部分を JavaScript 上で「再現」しているため，OCaml におけるコンパイルエラーと一貫性を保つのは困難である．

2 つ目の方法は，compiler-libs を利用して実際にコンパイルを行い，そのコンパイルエラーを流用することである．抽象構文木にブロック上の穴を表現するための構文を追加し，ブロックを抽象構文木に落せるようにして，コンパイルを行う．コンパイルエラーが起きたら，そのエラーを返す．このプログラムを js_of_ocaml でビルドすれば，ブラウザ上で用いることができる．この方法は，OCaml におけるコンパイルエラーとの一貫性を確実に実現できるが，お砂場を含むワークスペースにある全ブロックとテキストによる OCaml コードの対応関係を的確に保たなければいけない．

さらなる構文

新しい構文をただ単にブロックに追加するのは容易であるが,「不正なプログラムを組み立てられない」という OCaml Blockly の制約を維持しながらの構文の追加は自明でない．

例えば，ユーザの動作によってあらゆる箇所のブロックが動的に変更された場合にどう対応するかという仕様を考えなければならない．コンストラクタを定義するブロックが変更されたとき，コンストラクタ呼び出しはどう変わるべきか，その引数に自由変数が入っていたらどうなるべきか．また，let 文の引数宣言を削除したときに，引数を参照していた変数ブロックはどうなるべきか．

他にも，ブロックのユーザインタフェースとどう合わせるのかを考える必要がある．例えば，変数を持つパターンブロックが match ブロックに接続されたときに，変数宣言としてどう見せるべきなのか，可変長の要素を持つブロックであれば，それをどうユーザに操作させるのが適切か，などと言った問題がある．これらの問題を考えると，構文と構文が表す概念を OCaml Blockly の世界に載せるには時間が必要である．

本論文では，ユニットテストで様々な組み合わせのプログラムを機能させ続けることに注意を払つつ，OCaml Blockly での各構文の仕様を決め，種類を少しずつ足していった．基礎的な構文が実現できた今，OCaml の授業で使用するために，次に追加する必要のある構文を以下にまとめた．

- 柔軟なコンストラクタ定義，レコード型.
- 可変長のタプル.
- 豊富なマッチパターン.
- List モジュール関数.

OCaml で様々なデータセットを扱うために，コンストラクタ，レコード型といった柔軟なユーザ定義のデータ型を実現したい．また，現状ではタプルは 2 つ組のみしかサポートしていないが，可変長のタプルを追加する．次は，関数型言語の重要な特徴の 1 つといえる match 文のパターンを豊富にすることである．現在は `x :: xs`，`[]` といった 2 つのパターンのみが使えるが，これを拡張し，例えば `(a, b)` や，再帰的なパターン `(a, b) :: xs` を表現できるようにする．最後は，List モジュールのサポートである．

これ以上の構文は，実際に授業のカリキュラムに合わせてテストをした際に，必要な構文があれば随時サポートしていく指針だ.

ユーザビリティの点から

使いやすさ，理解しやすさの達成するためには，まず第一に初学者にテストしてもらいフィードバックをもらう必要がある．フィードバックに基づき，OCaml Blockly のユーザインタフェースに関わる部分を改善していく.

一方で，著者が実際に使用してみたとき，実使用に向けて，使い勝手の点から設計や実装が必要だと考える点は以下の 2 つである.

- OCaml の構文に合わせたブロックのユーザインタフェースの改造.
- 型を視覚的に表示することによる限界.

まず始めに，OCaml の構文に沿ってブロックのユーザインタフェースを改造する必要がある．現在ブロックそのものに関するユーザインタフェースはほとんど変えていない．もともと Blockly はビジュアルプログラミング言語環境を構築するためのライブラリであり，1 つの言語に特化した設計にはなっていないが，どちらかというと命令型言語との方が相性が良く，関数型言語のような文より式が主な言語をそのまま載せると，元の Blockly にあった直感的なユーザ体験を損ないがちである．図 7.1 では，

let ブロックの左側の余白部分が大きいせいで，代入や出力の関係が捉えにくくなってしまっている．初学者にとって OCaml の言語仕様をより直感的に理解してもうらためには，OCaml の言語仕様に特化したビジュアルのデザインが不可欠である．

また，型を視覚的に表示することには限界がある．型によってコネクタの形を変える [6] のアイデアは，プリミティブ型に関してはとても有効である．一方で，Function 型などといったパラメタ付型を形で表すことは，型が入れ子になって複雑になるほど，逆に理解をさせにくくしてしまう．入れ子になる型は視覚的に表示することをやめて，ホバー時にツールチップの形で出力する，などといった策を取る必要がある．

第8章　結論

ビジュアルプログラミング環境構築のためのライブラリ Blockly をベースにして，OCaml プログラムを組み立てるビジュアルプログラミング環境，OCaml Blockly を開発した．束縛変数，let 多相を含む型システムを実装し，シンタックスエラー，型エラー，Unbound value エラーを起こすプログラムの組み立てを拒否するユーザインタフェースを構築した．変数ブロックを用いた自由なブロックの組み立てを行うことができる，スコープチェックの付いた空間，スコープお砂場を実装した．ユーザが理解しやすいエラー出力や，OCaml 言語に慣れるために必要な基礎的な構文のサポート，テキストベースのプログラミングへの導入となるブロックとテキストとの相互変換を行った．

実使用に向けた課題は第 7 章にてまとめたが，それとは独立して，OCaml Blockly には更に高度な拡張に向けた展望がある．例えば，Try OCaml を用いてブラウザ上での OCaml コードの実行をサポートすることが考えられる．他にも，本学の授業で既に用いられているデバッグツールである，型デバッガ [11] との融合も考えられる．型の整合性を壊すようなコネクタを接続させようとしたときに，ブラウザ上に型デバッガを起動させれば良い．そうすれば，本論文が第 4.4 節で行なった，2 つの型がなぜ合わないかのエラー出力だけでなく，ユーザの意図したプログラムの型が破綻した原因はどの箇所にあったのか，という型の間違いを発見することができる．

これから OCaml Blockly を実使用していくことに向けて，本システムをさらに拡張，改善していきたい．OCaml Blockly の実装は，https://github.com/harukamm/ocaml-blockly にて公開されている．

謝辞

丁寧にご指導してくださった浅井先生に感謝します．友人として近況を報告し合い，励まし合ったＳ線香花火氏，中川氏に感謝します．貴重な体験をさせてくれた両親に感謝します．修士学生の間に技術的な数々のことを教えて頂き，いつでも相談に乗ってくださったプログラマＳ氏に感謝します．

参考文献

[1] Viscuit. https://www.viscuit.com/.

[2] Scratch. https://scratch.mit.edu/.

[3] Snap! https://snap.berkeley.edu/.

[4] 近松万由子, 岩崎英哉, 中野圭介. 関数型言語の初学者のための haskell のビジュアルプログラミング環境. 第 20 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, 2018.

[5] Blockly. https://developers.google.com/blockly/.

[6] Sorin Lerner, Stephen R. Foster, and William G. Griswold. Polymorphic blocks: Formalism-inspired ui for structured connectors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pp. 3063–3072, New York, NY, USA, 2015. ACM.

[7] 五十嵐淳. プログラミング言語の基礎概念. サイエンス社, 2011.

[8] Funblocks. https://stefanj.me/funblocks/.

[9] Codeworld. https://github.com/google/codeworld.

[10] 浅井健一. プログラミングの基礎. サイエンス社, 2007.

[11] Kanae Tsushima and Kenichi Asai. An embedded type debugger. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, pp. 190–206, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.