OCaml 初学者のプログラミング学習調査

北川 舞, 浅井 健一

お茶の水女子大学

{kitagawa.mai,asai}@is.ocha.ac.jp

概 要 プログラミング初学者がどのようなプロセスを経てプログラムを書くのかということは、あまり理解されていない。そこで 2015 年から 2020 年における OCaml 初学者のプログラミング学習データを分析し、初学者のプログラミング行動について分析を行なった。コーディングの進捗状態、コンパイル結果ごとのエラー状態やエラー原因を探ることで、初学者がどのように学習を進めるのか、またどのようなプログラミング概念が初学者を悩ませているのか明らかにする。さらにビジュアルプログラミングエディタ OCaml Blockly が初学者に与えた影響についても、データに基づき考察する。ここで得られた知見は、今後のプログラミング教育において、改善すべき問題点や教育姿勢を考える際に有用であると考える。

1 はじめに

プログラミングはプログラムを書くフェーズと、実行しその結果を把握するフェーズの繰り返しによって行われる。プログラマはコンパイラが出力した実行結果やエラーメッセージに応じてプログラムに手を加えていくが、エラーメッセージを正しく読み解き適切なプログラムを書く作業は初学者にとって容易なことではない。

一連のプログラミング行動を分析し、初学者がどのようにプログラムを書くのか、またどのようにエラーメッセージに対応しどの部分に困難を感じているのか把握することは、プログラミング教育と教育的プログラミング環境の開発に役立つ。本論文では、まず関数型言語 OCaml の初学者がどのようなプログラミング行動をとるのか、またどのようにエラーを発生させ何に難しさを感じているのか、データから分析することを試みた。

また本研究室では、初学者のプログラミングを支援するいくつかのツールを開発している。型デバッガ [13]、OCaml ステッパ [7]、OCaml Blockly[10] などである。しかしこれらの教育ツールの有用性と、初学者のプログラミング行動がツールの有無によってどのように異なるのかについては、未だあまり分析されていない。そこで従来のテキストベースでのプログラミング行動と OCaml Blockly を用いたプログラミング行動に見られる違いについてデータに基づき考察した。現状データ収集の途中ではあるが、効率の観点で一定の効果があったと考えられることを示す。

本論文の構成は以下の通りである。まず 2 節で後半分析対象となる使用ツール OCaml Blockly の概要と全体の取得データについて説明する。3 節では OCaml 初学者のプログラミング行動を観察し、4 節で初学者が特に困難に感じている部分を把握する。3 節 4 節はテキストベースのプログラミング行動に関する分析である。5 節ではビジュアルプログラミング環境 OCaml Blockly を使用して学習している学生のプログラミング行動を従来のデータと比較分析することで、OCaml Blockly が学習に与える効果と影響を調べる。6 節で関連研究について触れ、7 節でまとめる。

2 分析対象

2.1 OCaml Blockly

OCaml Blockly は OCaml の構文と一対一対応のついたブロックを用いてプログラムを組むことができるツールである。不正なプログラムを表すブロックをユーザインタフェースによって制限し、各ブロックに各構文を表すキーワードを印字し、型情報を形で視覚的に表している。開発者はこのビジュアルエディタによる学習で初学者が得られるメリットとして、以下の 3 点を挙げている。

- OCaml の構文を覚えていなくても、Syntax error に陥ることなくプログラムが組める
- テキストベースよりもストレス少なく、OCaml プログラミングの本質を優先的に学べる
- 1 つ 1 つの式が持つ形が見えることで、型付け関係が理解できる (Type error に陥りづらい)

つまり初学者は従来のテキストベースでの学習に比べて、瑣末なエラーに悩まず学習を進められていることが期待できる。なお一つ一つの構文は開発者の地道な実装に寄るものであり、型変数を伴うヴァリアント型や例外処理、ユニット型を必要とする構文については現在のところ実装されていない。したがってデータ取得を行なった授業の第 9 週までの課題範囲を網羅するに留まっており、それ以降はテキストエディタを使用することになる。

2.2 取得データについて

分析にはお茶の水女子大学で毎年 15 週かけて行われる初歩的な関数型プログラミングの授業から、2015 年から 2020 年にかけて得られたデータを用いた (2020 年のみ授業は 14 回だった)。履修者は情報科学を専攻する学部 2 年生毎年約 40 名であり、この授業で初めて OCaml を学習する。履修者は毎年の傾向として、命令型言語である C 言語を用いたプログラミングを 1 年次に 1 年間テキストエディタを用いて経験しているものの、関数型言語の経験がない人がほとんどである。授業には、教師 1 名と複数の TA がついており、学生に対し折々にデバッグ等のサポートを行う。

2015 年から 2018 年、2019 年、2020 年では、それぞれ取得できたデータが異なる。15 年から 18 年において授業は大学計算機室にて行われ、授業全体を通じて本研究に参加した学生はテキストエディタで作業を行った。その際計算機室の端末で作業をしていた場合、プログラムをコンパイルするたびに実行端末、実行時刻と実行した OCaml ソースコードおよびそのバッファを記録された。データ数は 4 年間で計 124441 件となった。学生個人を特定するような情報は記録されておらず、また室内の端末を用いない場合はデータは取得されない。したがってその他教師側で得られる情報 (例えば、学生それぞれの提出物、成績) 等と今回取得されたデータを結びつけることはできず、また日を跨いで同一の学生のデータを特定することもできない。

19 年からは OCaml Blockly が授業に用いられるようになった。OCaml Blockly には当時実行環境が付与されておらず、学生はこのエディタで作成されたプログラムをテキストエディタにコピーして実行を行う必要があった。また 19 年は OCaml Blockly のアプリケーションから直接のデータ取得を行なっておらず、18 年までと同様テキストエディタからのデータ取得のみに頼ってデータを得ている。したがって取得データの中にはテキストエディタを用いて作成されたプログラムと OCaml Blockly を用いて作成されたプログラムが存在するが、どのデータがどちらに類するものであるかはわからない。なおこのビジュアルエディタの使用については個々の学生の意思が尊重されたため、学生全体のうちどの程度の学生が使用したかについて明確なデータは取得できていない。そうした 19 年の取得データ数は 1 年間で計 25390 件となった。

20 年は世情により、授業は完全オンラインで行われた。テキストエディタでのデータ取得が難しくなったため、これ以前は行っていなかった OCaml Blockly 上からのデータ取得を試みた。アプリケーション上でプログラムに変更が加わるたびに、その時刻とその際のビジュアルコードを表す XML およびビジュアルコードを変換した OCaml のテキストコードが取得された。ビジュアルコー

ドに穴がある場合 OCaml のテキストコード上では?で表される。なお 19 年と同様 OCaml Blockly の使用は個々の学生の意思によって選択でき、またテキストエディタによるデータ取得は本学計算機室端末で作業を行なった場合のみ可能であるため、OCaml Blockly を使用しなかった学生、また使用をやめた学生のデータは、授業全体を通して取得できていない。テキストエディタでのデータ取得と比べデータ粒度が細かいためデータ数は 1 年間で計 155824 件となったが、OCaml Blockly の使用限界上第 9 週までのデータのみであり、また 10 人程度のデータにすぎない。

3 学生のプログラミング行動

この節では 15 年から 18 年のデータからテキストベースでプログラミングを行う OCaml 初学者のプログラミング行動を分析し、少なくとも構文的に間違いのないプログラムを書くまでの過程を観察する。なおここではプログラマがどのような問題を解いているか、完成したプログラムが意味的に間違いのないプログラムとなっているかどうかは考慮しない。

3.1 コンパイルエラーの種類と分布

表 1. OCaml コンパイラのエラーメッセージから分類したエラー

エラータイプ	コンパイラのエラーメッセージ/発生条件	割合 (%)
Syntax error	Error: Syntax error ...	26.2
Unbound error: value	Error: Unbound value ...	14.5
Unbound error: record	Error: Unbound record field	9.3
Unbound error: constructor	Error: Unbound constructor ...	0.7
Unbound error: other	Error: Unbound ...	5.5
Illegal character error	Error: Illegal character ...	4.1
Literal error	(エラーメッセージ中のどこかに) literal	0.2
Type error	Error: This expression has type ...	26.9
Pattern matching error[1]	Warning 8: this pattern-matching is ...	1.9
Other error	その他のエラーメッセージ	8.3
Timeout	実行に 30 秒以上かかる	2.4

[1] この授業では Pattern matching の Warning もエラーとみなすようにした。

まずはじめに最も頻繁に遭遇するエラーを調べた。15 年から 18 年に実行されたプログラムを OCaml コンパイラの出力するエラーメッセージに基づき分類した。実行が正常に終了せず、表 1 に示したような特定のエラーメッセージが現れない場合は Other error に分類し、実行に 30 秒以上かかるプログラム (無限ループしてしまうプログラムを含む) は Timeout とした。表 1 の割合は、その結果発生した 67792 個のエラーの分布を示している。様々なエラーのうち、最も一般的な 4 つのエラーが全体のの 76.9%を占めていて、その内訳は Type error(26.9%)、Syntax error(26.2%)、Unbound error: value(14.5%)、Unbound error: record(9.3%) となった。

それ以下のエラーは、例えば Illegal character error(4.1%) や Pattern matching error(1.9%) が挙げられる。しかし前者はプログラム文中に適切な処理の施されていない日本語文字列が現れた特定の状況で出力されるメッセージであり、後者は不足したパターンの詳細についての説明がエラーメッセージ文中に記載されることから、比較的エラー原因が特定しやすい。これらのエラーと比べて様々な要因が考えうる Syntax error、Type error、Unbound error の主に 3 種類のエラーの頻度が高くなる結果となった。

なお 8.3%が Other error に分類されたが、この中身は様々である。今回はエラーの意味合いではなくあくまで表 1 に示したような文字列をエラーメッセージ中に見つけた場合にそれぞれのエラーに分類しているため、中には意味的に分類するならば、Syntax error であったり、Type error となるエラーも含まれていた。

3.2 コンパイル間の作業

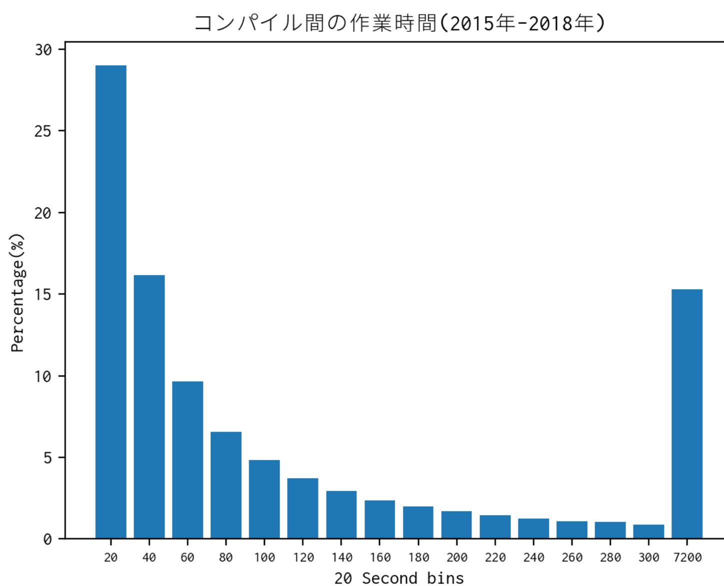

図 1. コンパイル間の作業時間 (2015 年-2018 年)

続いてコンパイルから次のコンパイルまでの間に費やしていた時間と、それらの実行が構文的なエラーのないプログラムを生み出す頻度を調査した。図 1 のヒストグラムの各バーは 20 秒ごとの時間くぎり内で再コンパイルされたプログラムの割合を表している。すべてのコンパイルの 84.7%は、直前のコンパイルから 300 秒以内に発生しているとわかった。

この結果は学生は再コンパイルするまでにほとんど時間をかけない傾向にあると示唆している。全体の再コンパイルまでの時間平均は約 300 秒となっており、学生が一つのプログラミングサイクルにさほど時間をかけないことがわかる。ところでこのデータからは、直前のコンパイル結果や変更を行なった直後のコンパイル結果については情報がない。学生はエラーのあるプログラムを修正しようとして書き換えを行いエラーのないプログラムに修正できたのかもしれないし、もともとエラーのないプログラムに書き足そうとしたのかもしれない。そこでプログラムを構文的なエラーが含まれている場合 (F) と構文的に正しい場合 (T) に分けて、それらのプログラムペアの割合とコンパイル間の経過時間を調べた。

表 2. プログラムペアの割合

		直前のプログラム	
		T	F
直後のプログラム	T	32.3%	15.0%
	F	10.0%	42.7%

実行結果別のプログラムペアは表 2 に示すように、4 つのうちのどれか 1 つに当てはまる。すべてのペアの 32.3%が構文的なエラーのないプログラム同士のペア (T → T) だったことがわかる。このペアは順調に課題を進めているないしは、同じ状態のプログラムを何度もコンパイルしている状態と考えられる。同様に 42.7%はエラーのあるプログラムを書き換えた直後の実行結果もエラーとなった (F → F)。これはエラーに悩み続けている状態である。F → T の場合は学生がどの程度の時間悩んだ末のことかわからないけれども、ついにエラーを修正できたタイミングのプログラムペア

を表す。T → F は学生がエラーのないプログラムが書けている状態から何か変更を加えて、その後最初にコンパイルした実行結果が構文的なエラーを含んでいた場合である。

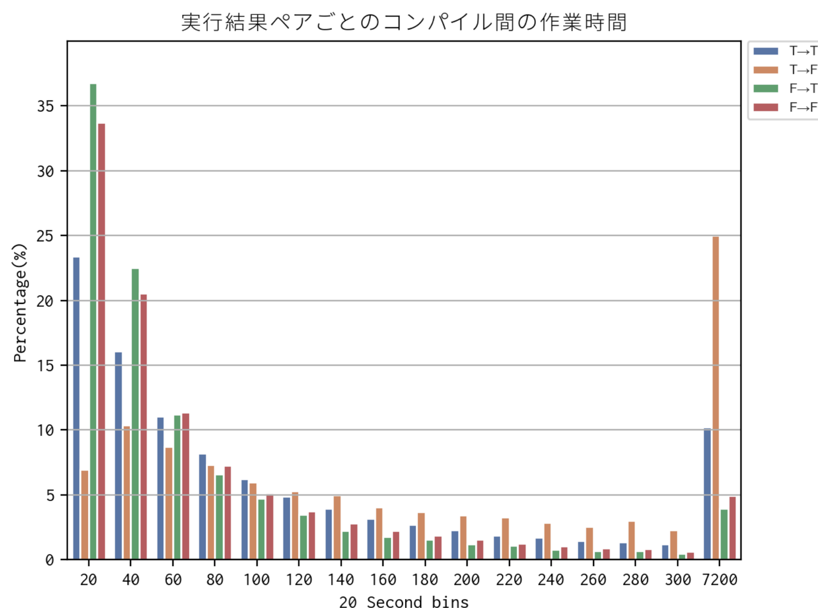

図 **2.** プログラムペアごとのコンパイル間の作業時間

再コンパイルまでの経過時間についてこれらのペアごとに割合を算出しグラフにした (図 2)。直前が構文的なエラーのあるプログラム (F) である時、学生は短時間で再びコンパイルする傾向にあり 1 分以内に再コンパイルする割合が高く見える。また直前が構文的に正しいプログラム (T) であるときの方が、それに続いて行うプログラム変更の作業に時間を費やす割合が高く見える。この時間に何が行われているかは正確にはわからない (夕ごはんを食べているかもしれない) が、他のパターンよりも何かしら作業をしている可能性が高い。

これらの観察から、学生がエラーメッセージを読みプログラムを編集し再びコンパイルするという一連の作業に対し、あまり時間を費やさないことがわかる。またエラーを修正する作業に比べて、構文的に正しいプログラムから新たなプログラムを書き足す作業の方により時間をかける傾向にあるとわかった。この結果は Java による先行研究 [8] の結果と類似しており、言語によらず初学者のプログラミング行動の一傾向を表していると考えられる。

さらにプログラムペアごとにどの程度のコード変更を行ったのかを調べた (図 3)。そもそもほとんどの実行で大したコード変更は行われていないことがわかる。F → T のペアでもこの傾向は観察されることから、学生が遭遇する最も一般的な構文的エラーは、最終的な修正には大規模な変更が必要なエラーではないことが多いと考えられる。F → F のペアが全体の半数近くを占めていることを考えると、エラーの修正方法がわからないうちはコードをあまり書き換えずさっさと実行してしまいそのうちコンパイルに成功するのではないだろうか。10word 以上の変更があったプログラムの割合でそれぞれ最も高くなるのは T → F の場合となっており、比較的真っ当なコード変更が行われるのはコンパイルに成功した直後のタイミングだったと考えられる。

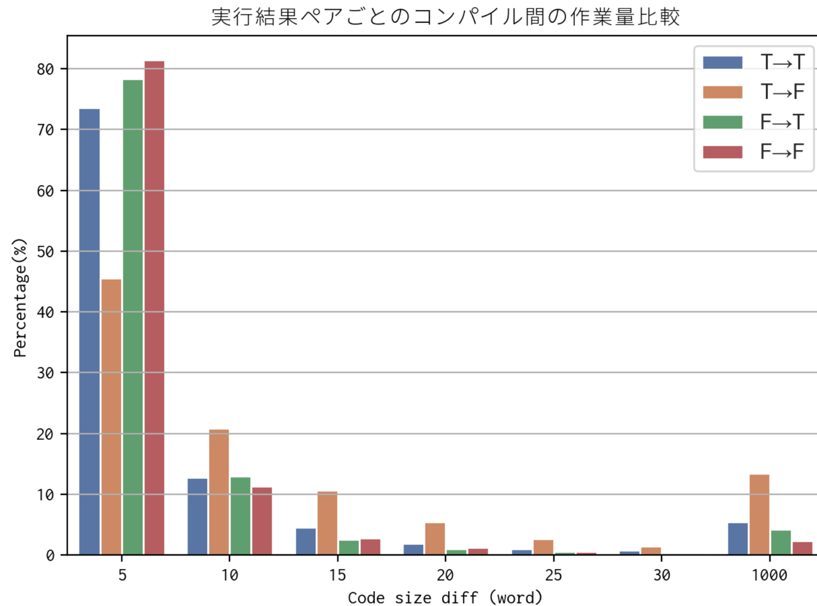

図 **3.** プログラムペアごとのコンパイル間作業量

4　初学者のエラー発生原因

初学者がプログラミングに苦戦する箇所を理解するためには、主なプログラミング概念の難しさと、その難しさが授業課題に取り組んでいく中でどのように変化するかを理解する必要がある。この節では「プログラミングにおいて難しかった箇所」を「エラーの修正に時間を要した箇所」と考えて調査を行なった。

4.1　エラー修正時間

学習できているのか疑問が残るところではあるが、何度発生したとしてもすぐに修正できるエラーは学生にとって負担ではない。修正に時間のかかるエラーが学生のプログラミングを難しくする原因となっている可能性が高い。そこでまずエラーの修正に要する時間を計算することとした。

表 **3.** 学生のプログラミング行動の流れ (一例)

id	プログラムタイプ	作成時間	備考
a	Ok	5:00	エラーが発生していないプログラム
b	Syntax Error	1:30	
c	Ok	2:00	b で発生したエラーを修正したプログラム
d	Type error	0:30	
e	Type error	5:00	
f	Ok	2:30	d から悩んでいたエラーを修正したプログラム

例えば表 3 のような流れでプログラミング行動をとった学生がいたとする。最初に作ったプログラム a ではエラーが発生しなかったものの、その次に作ったプログラム b では Syntax error が発生した。このエラーは次のプログラム c ではエラーが発生していないので、c のプログラムを作成する際に修正されたと考えられる。したがって b で発生した Syntax error の修正時間は、c のプログラム作成時間である 2:00 となる。一方エラーが簡単に修正できない場合もままある。d で Type error が発生した後 e でも引き続き Type error となっている。この際このエラーが d と全く同じ原因で発

生しているかどうかは機械的に判断することができないが、同種の間違いを犯し続けているという点に注目する。f でエラーのないプログラムが書けているので、ここでは d から f にかけて継続して発生していた Type error の修正時間は、e と f のプログラム作成時間の合計と考えて、この Type error の一群の修正に 7:30 を要したと計算することとした。

このようにして 15 年から 18 年にかけて各エラーにかかった修正時間を計算し、その週ごとの平均の推移を示したのが図 4 である。3.2 節のコンパイル間の作業時間から予想された通り、値はさほど大きくはならなかった。突出している値として第 3 週第 4 週第 9 週の Pattern matching error が挙げられるが、このエラーは件数が少ないため平均を計算する際に外れ値が大きく影響したと考えられる。それ以外の箇所では全ての週において各種類のエラー修正時間平均は 5 分以下となっている。逆に言えば修正に 5 分以上かかったエラーは修正に時間を要したエラーであると言える。

Illegal character error の修正時間平均の値は徐々に短くなっており、学生がエラーに慣れ修正がスムーズになっていく様子が反映されている。しかしそれ以外のエラーについてはその限りではない。これらの修正時間の増減は各週で扱うプログラミング概念の難易度によると考えられる。

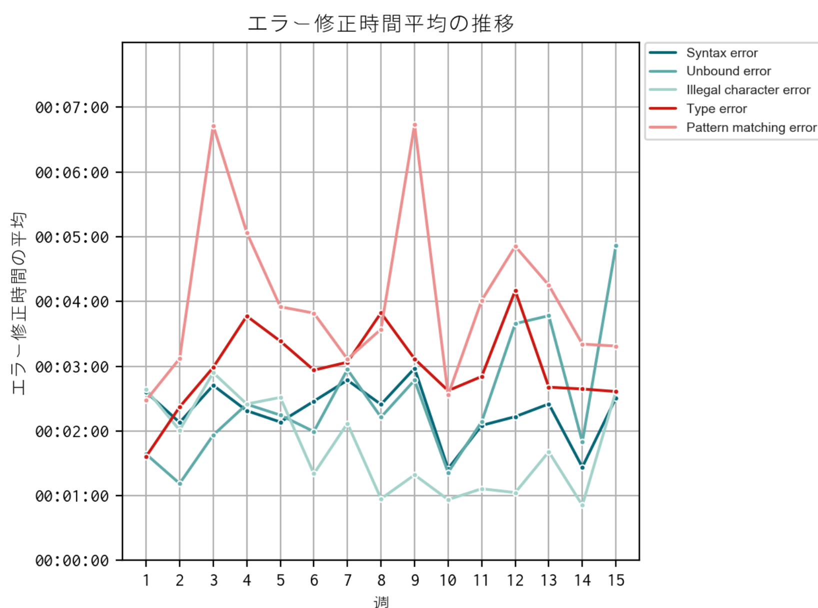

図 4. エラー修正時間平均 (2015 年-2018 年) の推移

4.2 授業課題の概要

授業では初歩的な OCaml の構文と関数型プログラミングの概念を学習しながら、ダイクストラ法を用いた最短経路問題を解くプログラムを完成させる。各回では OCaml の構文やデータ構造とアルゴリズム、再帰関数や高階関数といった関数型言語の特徴的な要素を学習しながら、いくつかの課題となる関数を完成させる。18 年までのデータを用いて、これらの学習できるプログラミング概念と課題数等のデータをまとめたものが表 4 であり、エラー発生件数を集計したのが図 5 となっている。各週それぞれのエラー件数から算出したそれぞれのエラータイプの割合 (あるエラータイプの発生件数/その週のエラー発生件数全体) が示してあり、かっこの中に示しているのは、4.1 節の計算方法から算出した修正に 5 分以上かかったエラーにおけるそれぞれエラータイプの割合 (あるエラータイプで 5 分以上かかったエラーの発生件数/その週の 5 分以上かかったエラー発生件数全体) である。なおエラー件数から算出したそれぞれのエラータイプの割合が 2.0%以下の時はごくわずかであるとして記載していない。

実行回数およびエラー発生回数中央値はその週の授業日から 1 週間の間に取られたユーザごとのデータの中央値となっている。各週に新しい課題が追加されるので、基本的にある週のデータはその週の課題を解くものであったが、課題発表後から授業全体の終了期日まで再提出を認めているので、他の週の課題を解いている学生もいた。今回の分析では同じ時期のプログラミング行動であるとして、特段の区別は行なっていない。なおこの授業では、デザインレシピ [6] という考え方を大切にして授業が行われている。これは作成しようとしている関数の目的、また予想される実行例を予め整理してからプログラムを書くべきである、という考え方である。

表 4. 各週の授業課題と実行回数 (2015 年-2018 年合算)

週	問題数	実行回数 中央値 (回)	エラー発生 回数中央値 (回)	Syntax Error(%)	Type Error(%)	Unbound Error(%)	Illegal Error(%)	Pattern Error(%)	プログラミング概念
1	3	10	4.5	21.9(26.6)	27.5(20.3)	16.5(12.5)	22.2(25.0)	6.5(14.1)	関数定義、四則演算、文字列結合
2	5	21	9	24.2(27.4)	40.0(49.1)	12.7(9.4)	20.8(12.3)	—	if 文、関数適用
3	7	17	10	35.4(31.3)	26.7(33.0)	14.9(15.0)	18.3(14.5)	—	タプル、レコード、match 文
4	5	21	11	21.5(16.0)	33.3(41.2)	27.7(21.9)	4.9(10.2)	6.4(3.5)	リスト構造、再帰関数 (1)
5	5	23	13	28.5(20.9)	33.2(44.0)	23.2(18.7)	3.3(3.3)	—(2.2)	再帰関数 (2)、挿入法
6	4	21	13	32.5(25.2)	28.0(36.6)	25.7(23.2)	4.2(1.6)	—(4.7)	ダイクストラ法
7	7	21	13	28.8(22.1)	25.9(35.3)	33.0(30.7)	3.5(1.8)	—(1.3)	高階関数 (List.map)
8	3	18	12	34.5(27.0)	20.1(42.1)	23.4(21.2)	—	2.2	List モジュール (List.fold_right)
9	6	19.5	10	23.0(14.8)	25.3(29.7)	32.5(26.3)	2.6(1.1)	—(2.3)	アキュムレータ (List.fold_left)
10	7	24	12	19.5(14.4)	38.0(45.3)	17.7(10.4)	2.2(0.6)	—(2.4)	木構造
11	7	25	14	30.0(22.3)	25.8(38.1)	28.9(24.1)	—(1.2)	—(2.5)	例外処理
12	11	20	12	24.2(18.1)	18.8(24.5)	39.6(34.4)	—(0.6)	2.4(3.7)	モジュール、赤黒木
13	6	22	14	25.0(19.5)	17.2(18.3)	44.7(47.5)	—(0.3)	—(2.9)	副作用命令
14	3	20	10	21.6(12.9)	39.5(49.7)	22.5(18.4)	—	—(2.0)	参照型 (ref)
15	3	10	7	27.0(19.1)	18.6(17.1)	43.5(52.8)	—(1.5)	—(1.5)	ヒープ構造

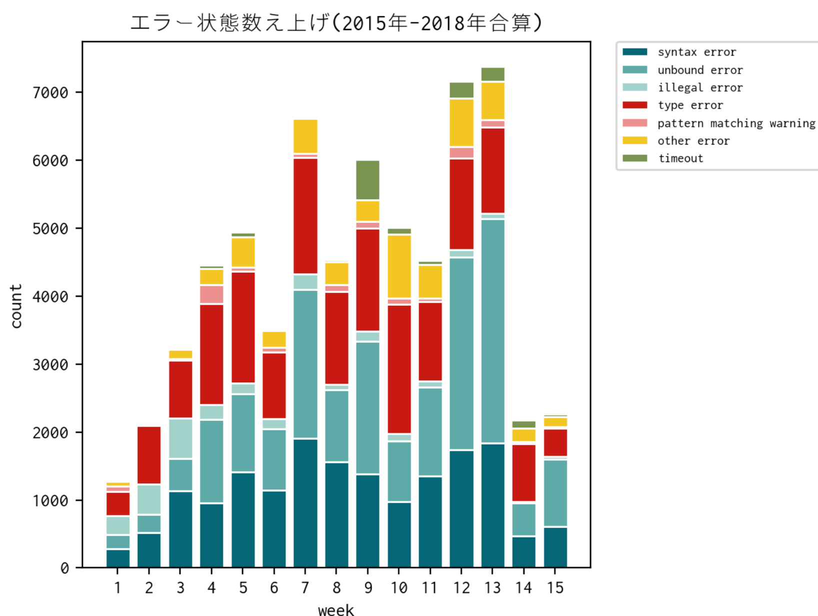

図 **5.** エラー発生件数の週ごと推移 (2015 年-2018 年合算)

4.3　データの分析結果

第 1 週と第 15 週

第 1 週と第 15 週はそれぞれ授業全体の導入部分、補足的部分であったことから、結果にいくつか特徴が見られた。まず実行回数中央値とエラー発生回数中央値の値がともに小さい。それぞれ授業全

体の導入部分、補足的部分の内容となっているため課題が少ないことに加えて、問題も易しかったと考えられる (エラー発生件数も少なくなっていた)。それ以外の週においてこれらの値は落ち着いており、問題数にかかわらず課題のボリュームは同程度であったことがわかる。

第 1 週は最も基本的な OCaml プログラミングに学生が慣れていない様子がデータからもわかった。それは Syntax error に次いで、比較的修正が容易と思われる Illegal character error が修正に手間取るエラーとして割合を高く占めていることからも伺える。match 文学習以前であるにも関わらず、Pattern matching error が出たのは、次に示すような変数と文字列型の区別がついていないという初歩的な間違いによるものだった。

```
(* 目的: 名前の文字列を受け取ったら「こんにちは、〇〇さん。」という文字列を返す *)
let aisatsu "name" = " こんにちは、nameさん。"
```

また第 15 週に発生したエラーは、第 15 週の課題が影響しているというよりも、それ以前の課題の再提出に注力した結果であることがプログラミングデータの目視観察からわかった。

発生させがちなエラーと修正が難しいエラー

表 4 より 15 週間の中には発生件数割合の観点で見ると、Syntax error が最も多くなる週や Type error が最も多くなる週、時には Unbound error が最も高くなる週があるが、どの週においても一定レベルのこれら 3 種類のエラーが出ていることがわかる。この件数から見た割合と、修正時間から見た割合を比較するといくつかわかることがある。

まず、Type error 以外のエラー (Syntax error、Unbound error、Illegal character error、Pattern matching error) でそれぞれかっこの外と中の割合を比べると、ほとんどの週でかっこの中の割合が小さくなっていることがわかる。このことからこれらのエラーは、発生件数割合に対して苦労して修正するエラーの割合は小さくなっていることがわかる。

一方 Type error で同様の比較を行うと、第 1 週と第 15 週を除いて割合は全てかっこの中の方が高くなっている。発生件数割合のわりに修正に時間を要するエラーが多い様子がうかがえる。また修正に時間がかかったエラーのみで割合をみると、大半の週で Type error が最も修正に時間のかかるエラーとして割合を占めることがわかる。これを覆しているのは第 1 週第 15 週と第 12 週第 13 週のみである。第 1 週第 15 週がそれ以外の週と傾向が異なることはすでに確認した。第 12 週第 13 週について次に考察する。

Unbound error の割合が目立つ週

第 12 週第 13 週では Unbond error が最も割合が高くなっており、Type error は全週を通じて最も低い割合となっている。また第 7 週と第 9 週も、発生件数割合が Type error よりも Unbound error の方が高い結果となった。ただしこれらの週は、図 5 を見てわかるように全体を通じて最もエラー発生件数が多い 4 週であり、割合は小さいものの Type error も相当数出ている。つまり大量の Type error を上回って Unbound error を発生させる原因があった、ということである。

これはこれらの週の学習項目と作成関数が原因であると考えられる。第 12 週第 13 週ではモジュールと副作用命令をそれぞれ扱う。これらの概念はテストケースを書くのが難しい。前述の通りこの授業の中で学生はデザインレシピの考えに従って課題を進める。そこで学生はこの考え方でプログラムを書くことを実現すべく、実行例のテストケースを書いてから関数本体作成に取り組む、という流れをとるが、この流れによって Type error が発生する、ということがしばしば起こることがわかった (後述)。モジュールや副作用命令を扱う週では、この種の Type error が発生しづらいことが Type error の件数を相対的に減らす原因となったと考えられる。またこれらの 4 週に共通する点として、別の週に作成する関数を組み合わせて新たな関数を作成するということが原因としてあった。ファイル内の定義関数を把握できていない状態で不用意な実行を行い、Unbound error が発生しやすくなったと考えらえる。

一定レベルコード量が増加する、あるいは複数ファイルに跨って作業する必要が出てくると、全体を把握してプログラミングを行うことは初学者にとって難しい作業であるようだ。

Type error の発生原因

以上のことから初学者が最も苦労して修正を行うエラーは Type error であると考えられる。ではなぜ Type error の修正に手間取るのか、修正に時間を要したエラーを目視で確認することによって調査を行った。調査の結果、課題の内容によらず Type error の修正に手間取る主たる 2 つの原因が確認できた。1 つ目はデザインレシピの考えから学生が事前に作成するテストケースとその後学生が作成する関数の型の不一致によるものであり、2 つ目は関数適用や各種構文の各節 (例えば、`if` 文の `then` 節と `else` 節) での型の不一致によるものである。

前者はこの授業独特の課題作成順によって生まれる Type error である。例えば 2 つのベクトルの足し算を行う関数 `add_vector` を作成したいとする。デザインレシピにしたがって学生はまず、関数があると仮定し以下のように関数の目的とテストケースを作成する。

```
(* 目的: 2つの整数2次元ベクトルの足し算を行う *)
let add_vector vec1 vec2 = ... (* 作成前 *)
(* テスト *)
let test1 = add_vector (1, 2) (3, 4) = (4, 6)
let test2 = add_vector (-1, 5) (7, 3) = (6, 8)
```

この時点で学生は作成したい関数の目的を把握できている。その後学生は関数本体の定義に入るが、例えば `match` 文の構文を理解できていないと、次のような誤ったプログラムを作成することがままある。

```
(* 目的: 2つの整数2次元ベクトルの足し算を行う *)
let add_vector vec1 vec2 = match vec1 vec2 with
  (x1, y1), (x2, y2) -> (x1+x2, y1+y2)
(* テスト *)
let test1 = add_vector (1, 2) (3, 4) = (4, 6)
let test2 = add_vector (-1, 5) (7, 3) = (6, 8)
(* 発生するエラーメッセージ:
    Error: This expression has type 'a * 'b but an expression was expected of type
      'c -> (int * int) * (int * int) *)
```

この時関数 `add_vector` だけで見れば構文上のエラーはない。ただ本来ならば、整数のペアを想定していた引数 `vec1` が `vec2` を引数とするような関数になってしまっただけである。ところが学生はテストケースを事前に書いているため、このプログラム全体を実行した時、上記のようにエラーメッセージが出力される。結果として学生は作成関数の目的を明確にし、後々の意味的エラーの発生を防ぐプログラミングを行うことができるが、今回の調査からこの種の意味的エラーに悩むことはデザインレシピの活用により減っているものの、代わりに対処できて欲しい Type error の読み解き自体に学生が苦労していることが明らかになった。

後者は学生がプログラム全体の構造を把握できていないために発生する問題である。熟練者となれば `if` 文の `then` 節 `else` 節や `match` 文の各分岐の結果が同じ型にならねばならないことを理解できており、以下のようにその種の誤りゆえにエラーメッセージが出たとしても、解消に時間はかからない (`[]` の場合は適切にリスト構造を返すように書くことができているものの、`then` 節は int 型の値を返している)。

```
(* 目的：整数のリストを受け取ったら、 その中の負の数をすべて 0 にしたリストを返す *)
  let rec negative_zero lst = match lst with
      [] -> []
    | first :: rest -> if first >= 0 then first
      else if first < 0 then 0
      else negative_zero rest;;
(* 発生するエラーメッセージ:
    Error: This expression has type int but an expression was expected of type 'a
      list *)
```

しかし初学者にとっては、少し構造が複雑になると途端にどの部分の型がどのように関連付けられていれば良いのか想像するのが難しくなってしまう。その結果発生するこの種の Type error に悩まされている様子も多く観察された。

初学者にとってプログラムの構造やデータ構造を把握すること、またそれぞれのデータ構造を表す構文を適切に書き表すことが難しいポイントであることがわかる。困難を減らすためにはこれらの構造を初学者にわかりやすい形で可視化しプログラミングを補助することが有用であると考えられ、この点において特に OCaml Blockly が初学者の学習に有用である可能性がある。

5 OCaml Blockly の効果と影響

OCaml Blockly によって学生のプログラミング行動がどのように変化したか調査する。2.1 節で説明した OCaml Blockly の開発意図に沿って、ユーザの使用結果からこれらの目標を達しているかどうか、が調査の焦点となる。

5.1 課題解答時間の変化

OCaml Blockly を用いてプログラムを作成する場合、穴のないプログラムを完成させた時そのプログラムに構文的エラーは存在し得ない。エラーが出ないので、4 節で検討したように構文的エラーの発生具合から何かを分析することはできない。そこで課題を完成させるまでの時間について変化を調べた。課題が完成したかどうかは、各課題関数が境界事例のテストにおいて、模範解答と一致するかどうかで判定した。このシステムを以下チェックシステムと呼ぶ。

表 5. ある課題関数 A を解く際のプログラミング行動とデータ取得タイミングの違い (一例)

id	テキストエディタ (15-18 年)	ビジュアルエディタ (20 年)	作成時間	備考
a	—	○	0:00	課題関数名 A が初めてプログラム中に現れた
b	○	○	1:30	A を作り始めてから初めてコンパイル実行した
c	—	○	2:00	修正中
d	—	○	0:30	修正中
e	○	○	5:00	修正をして実行してみた
f	—	○	2:30	正しい関数 A が完成した
g	○	○	0:40	正しいプログラムを始めて実行した

表 5 は同じようなプログラミング行動をとった、テキストエディタで作業した学生とビジュアルエディタで作業した学生の作業例である。それぞれ丸となっているところが、データが取得されたタイミングを表している。テキストエディタでデータ取得を行っている時期の解答時間は、課題関数名がプログラム中に現れた後初めてコンパイル実行を行ったタイミング (b) から、チェックシステムに通過するプログラムが完成した後初めてコンパイル実行を行ったタイミング (g) までの時間を計測した。一方ビジュアルエディタでデータ取得を行っている時期の解答時間は、課題関数名がプログラム中現れたタイミング (a) の変更からチェックシステムを通過しうるプログラム書き換えのタイミング (f) までの時間とした。計測開始時点と終了時点が異なるため、20 年のビジュアルエディタでのデータが課題作成の凡そ正確な時間を測れているのに対し、18 年までテキストエディタ

でのデータはプログラムを完成させてからすぐにコンパイルした場合実際の作成時間よりも短めになり、(課題をまとめて作ろうとするなど) プログラムを完成させてからしばらくコンパイルしない場合実際の作成時間よりも長くなる。

OCaml Blockly を使って課題関数を作成できた第 9 週までの課題について、解答時間中央値を 15 年から 18 年までのデータと 20 年のデータで比較した結果が表 6 である。18 年までのデータにおける解答時間中央値は、テキストエディタで取得されたデータで計算された課題解答時間の中央値となっており、20 年のデータはビジュアルエディタで取得されたデータで計算された課題解答時間の中央値となっている。

なおテキストエディタでデータ取得を行っている際 1 回目の実行で課題が完成する可能性 (表 5 の (a) の時点で正しいプログラムが完成した場合) がありその時の解答時間は 0 秒となるが、これらのデータは実のところどれだけの作業時間を要して課題を解いたのか不明であるため除いて中央値を計算している。これはビジュアルエディタのデータ取得では解答時間 0 秒が現実的に不可能であることから、データの性質をできるだけ揃えるために行なった作業でもある。

18 年までのデータが作成時間を正確に表したものではないという前提の元ではあるが、表 6 からわかるようにほとんどの問題において、ビジュアルエディタを用いた 20 年の方の解答時間の中央値が小さくなっていることがわかる。第 2 週の `taikei` 関数、第 3 週の `person_t` の定義 `taikei_hyoji` 関数 `hyoji` 関数、第 4 週の `length` 関数は特に値の低下が見られ、OCaml Blockly が効果的に活用できたことがわかる。課題関数名が太字になっている箇所は、20 年の方が値が大きくなった問題となっている。一部の課題で作業時間が多少長くなっている様子が見られるが、OCaml Blockly から取得されたデータは 20 年のデータのみで 10 数人程度の学生のデータにすぎないので、今後データがそろうにつれてより正確な判断ができると考えられる。

5.2 OCaml Blockly から離れて以降のプログラミング行動に与える影響

ここまで 15 年から 18 年までのデータと 20 年のデータを使って分析を進めてきた。ところで 19 年のデータは OCaml Blockly とテキストプログラミングの中間に位置するデータとなっている。すなわち第 9 週までは一定数の学生が OCaml Blockly を使って課題を進め、第 10 週以降全員が OCaml Blockly が使えなくなりテキストエディタに移行し、その際のテキストプログラミングのデータも取られている (20 年は第 10 回以降のプログラミングデータは取れていない)。端から OCaml Blockly を使用しなかった学生のデータも含まれているので一概には言えないが、使用した学生もいるデータの統計結果が全編テキストプログラミングで課題を進めた学生群のデータと比較してどのような結果になるか確認することで、OCaml Blockly ユーザが使用後、テキストプログラミングに移行してからのプログラミング行動に与える影響の一端を確認することができると考えられる。

全体のエラー発生件数

OCaml Blockly が使えなくなった第 10 週以降のエラー発生件数を、15 年から 18 年までの平均と 19 年とで比較すると、第 11 週が若干増加しているものの各週同程度以下となっている (図 6)。2.1 節で述べている通り OCaml Blockly は各種エラーに悩まされることなく各種エラーを発生させる原因となる、構文や型の概念の理解を助けるツールとなっている。したがってツールの使用をやめた後にもエラーを発生させにくいプログラミング行動を取ることを期待した。結果として特別 Syntax error や Unbound error を出す傾向が弱まった、あるいは型の概念がきちんと身について Type error の割合が減少したということはなかったが、テキストエディタに慣れないために種々のエラーが多く発生したという印象は受けない。

課題解答時間中央値

表 6 と同様に、OCaml Blockly が使えなくなった第 10 週以降の課題解答時間を、15 年から 18 年の中央値と 19 年の中央値とで見るとあまり変化が見られない (表 7)。どの程度の学生が OCaml

表 **6.** 2015 年から 2018 年および 2020 年の課題解答に関する統計比較 (第 1 週-第 9 週)

週	課題関数名	エラー回数平均 /実行回数平均	解答時間中央値 (2015-2018)	解答時間中央値 (2020)	解答時間中央値 (2019)(参考値)	プログラミング概念
1	chocolate	0.389	0:06:44	0:04:50	0:11:31	関数定義, 四則演算 (int)
1	aisatsu	0.516	0:11:04	0:05:10	0:08:45	関数定義, 文字列結合
1	bmi	0.559	0:10:29	0:08:03	0:10:51	2 引数関数定義, 四則演算 (float)
2	**hanbetsushiki**	0.648	0:05:43	0:06:17	0:08:59	3 引数関数定義
2	kai_no_kosuu	0.578	0:11:18	0:04:49	0:14:03	if 文, 関数適用 (hanbetsushiki を使用)
2	taikei	0.644	0:14:44	0:05:36	0:12:29	if 文, 関数適用 (bmi を使用), 文字列
2	**shohizei**	0.506	0:08:11	0:09:42	0:11:07	四則演算
2	**choco**	0.293	0:13:03	0:18:10	0:16:40	if 文, 関数適用 (shohizei を使用), 文字列
3	add_vector	0.650	0:07:18	0:05:36	0:08:03	タプル, match 文 (複数)
3	person_t	0.656	0:14:21	0:02:42	0:11:53	データ型定義
3	taikei_hyoji	0.601	0:23:45	0:03:31	0:19:20	レコード, match 文, 文字列結合
3	manhattan	0.574	0:12:28	0:05:09	0:09:59	タプル, match 文 (複数)
3	ekimei_t	0.737	0:12:54	0:07:12	0:09:02	データ型定義
3	hyoji	0.529	0:35:48	0:09:57	0:21:12	レコード, match 文, 文字列結合
3	ekikan_t	0.757	0:12:40	0:06:53	0:08:17	データ型定義
4	contain	0.705	0:05:29	0:03:10	0:05:34	リスト, 再帰
4	count	0.627	0:11:11	0:08:04	0:10:05	リスト, if 文, 再帰 (スタックを利用)
4	length	0.661	0:09:39	0:02:41	0:08:12	リスト, 再帰 (スタックを利用)
4	romaji_to_kanji2	0.713	0:20:32	0:11:10	0:09:58	レコードのリスト, 再帰, タプル, 文字列
4	get_ekikan_kyori2	0.692	0:16:51	0:14:41	0:13:49	レコードのリスト, 再帰, 複雑な論理演算
5	negative_zero	0.662	0:07:11	0:01:07	0:05:12	if 文, 再帰 (リストを返す)
5	even	0.680	0:09:23	0:06:44	0:07:18	if 文, mod 演算, 再帰 (リストを返す)
5	insert	0.605	0:18:22	0:12:19	0:12:33	挿入法 (補助関数), 再帰 (リストを返す)
5	ins_sort	0.583	0:20:17	0:10:28	0:10:45	挿入法 (insert を利用), 再帰 (リストを返す)
5	seiretsu2	0.668	0:22:36	0:08:51	0:11:32	レコードのリスト, 挿入法
6	**add1_list**	0.596	0:02:13	0:04:25	0:04:50	再帰, (局所変数定義)
6	eki_t	0.764	0:15:23	0:08:34	0:11:52	データ型定義
6	make_eki_list2	0.721	0:14:28	0:09:27	0:11:47	複数のデータ型を利用, 再帰
6	shokika2	0.706	0:14:53	0:07:15	0:12:24	if 文, 再帰
7	**double**	0.695	0:04:50	0:05:12	0:05:33	List.map, (局所変数定義/匿名関数)
7	negative_zero	0.662	0:07:18	0:03:26	0:05:12	List.map(使用して再定義)
7	num_zero	0.677	0:07:40	0:06:08	0:05:42	List.map
7	koushin1	0.697	0:14:11	0:07:15	0:13:49	レコードのリスト, (List.map), 局所変数定義
7	koushin	0.649	0:13:45	0:07:00	0:13:39	レコードのリスト, 関数適用 (koushin1 を使用)
7	make_eki_list2, shokika2	0.702	0:19:55	0:11:33	0:16:48	List.map(使用して再定義)
7	**make_initial_eki_list2**	0.655	0:13:04	0:16:19	0:14:04	List.map
8	**add_pairs**	0.767	0:05:25	0:08:51	0:07:38	再帰, List.fold_right, (パターンマッチつき let 文)
8	saitan_wo_bunri2	0.697	0:23:35	0:18:58	0:18:33	レコードのリスト, 再帰, List.fold_right
9	**gcd**	0.648	0:06:21	0:08:26	0:04:42	再帰 (リストではない)

Blockly をどの程度使用したかわからないため不確かであるが、ビジュアルエディタから入っても、その後テキストエディタでプログラムを書きにくくなっている、という様子は見受けられなかった。

5.3 アンケート結果

まずはじめにどの程度の学生がどのように OCaml Blockly を使用してくれていたか、19 年と 20 年のアンケートの結果を示す。質問項目は以下のようにしそれぞれ 5 段階評価で答えてもらった。アンケートは受講者が任意で提出できるもので、19 年度は 20 名 20 年度は 30 名が解答した。

1. OCaml Blockly は役に立ちましたか。
2. OCaml Blockly を使用したことで、よりプログラミングしやすくなった (プログラムの構造を意識できるようになった。型を意識できるようになった) と感じますか。
3. 逆に OCaml Blockly を使用したことで、自分のプログラミング能力が下がった (OCaml Blockly が手放せない。テキストエディタでのプログラミング能力が育たなかった) と感じますか。
4. この授業は、自分のプラスになりましたか。

結果は表 8 のようになった。問 1 より、ブロックインタフェースは役に立った (4)、ある程度役にたった (3) と解答した学生が 9 割となっており、ほとんどの学生が OCaml Blockly を活用できたと解答した。使ってくれた学生はプログラムの構造を把握するのに有用であったと解答した一方で、

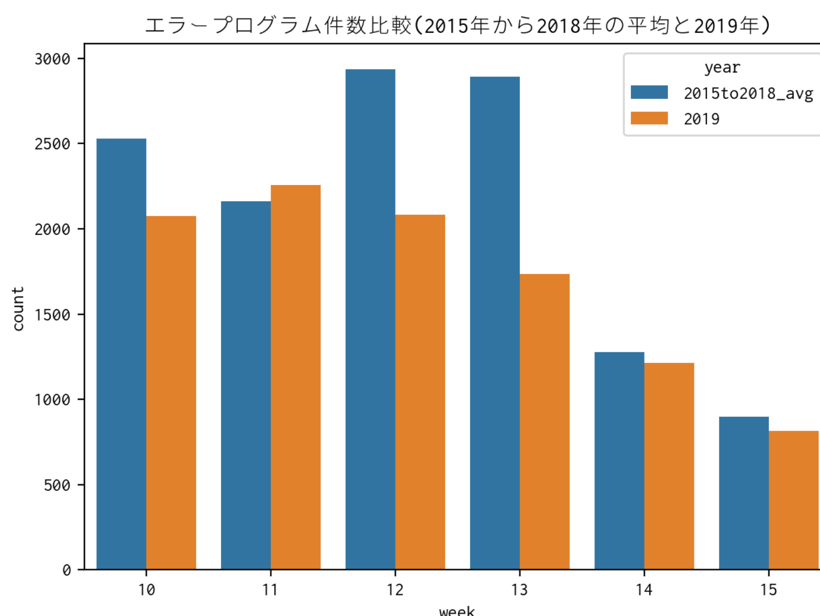

図 **6.** エラー発生件数比較 (2015 年から 2018 年の平均と 2019 年)

一部の学生は「プログラムは自分の手で書きたかったため使用していない」と解答した。ビジュアルプログラミングエディタを使用したプログラミングがテキストエディタを使用した従来のプログラミングに対して「本格感」を欠くという印象を抱く学生は多い。また問 2 の結果から、ブロックインタフェースによってプログラムの構造やデータ構造を意識してプログラミングを行うことについては、強く感じる (4) あるいはある程度強く感じる (3) と回答した学生が 7 割という結果になった。プログラムの構造把握の一助となったことが伺える。しかしコネクタの形状の違いで型の概念を、あるいは利用できる変数ブロックを限定することでスコープの概念を理解させようとするなど「感覚的に」OCaml プログラミングを行えることを目指している OCaml Blockly であるが、一部の学生は「操作方法がわからず使用しなかった」と解答している。構文の追加や操作性の向上によってこれらの課題を改善していく必要がある。なお問 3 から学生の主観になるが、ブロックインタフェースがプログラミング能力を育むのに悪影響を及ぼしたと感じることはないとわかった。

表 **7.** 2015 年から 2018 年および 2019 年の課題解答に関する統計比較 (第 10 週-第 13 週)

週	課題関数名	解答時間中央値 (2015-2018)	解答時間中央値 (2019)
10	double_tree	0:05:42	0:04:05
10	search	0:17:43	0:17:21
10	insert	0:28:13	0:26:34
10	inserts	0:21:00	0:21:26
10	search(再)	0:19:59	0:19:32
10	map_tree	0:18:40	0:14:26
10	fold_tree	0:21:19	0:21:35
11	assoc	0:07:42	0:04:30
11	total_price2	0:26:21	0:23:35
11	total_price	0:11:01	0:09:41
11	get_ekikan_kyori2(再)	0:08:16	0:09:54
11	koushin(再)	0:16:49	0:17:06
11	romaji_to_kanji2(再)	0:15:33	0:17:44
12	totalprice	0:20:00	0:18:13
13	fac	0:13:19	0:16:23

表 **8.** OCaml Blockly 使用者アンケート (2019 年+2020 年)

問	(役に立った/強く感じる)	4	3	2	1	0	(使わなかった/全く感じない)
1		10+10	4+15	2+3	1+0	2+6	
2		7+4	6+18	5+12	0+0	2+0	
3		0+0	0+0	4+10	6+11	10+14	
4		15+26	3+8	1+1	0+0	0+0	

6 関連研究

初学者のプログラミング行動を分析する研究は様々行われている。例えば Jadud[8] は教育用プログラミング環境 BlueJ を使用してオブジェクト指向プログラミングを学習している Java 初学者の学生のプログラミング行動を観察し、学生が発生させやすいエラーがわずかな修正で済むこと、また発生しやすいエラーが何かを示し、開発しているプログラミング環境の改善に利用している。Blikstein[2] は時間とコードサイズでグラフを作成し、学生のプログラミング行動を可視化して分析を試みている。

また初学者が難しく感じるプログラミング概念は何か、という観点での研究も行われている。Cherenkova ら [4] はコンピュータプログラミングコース CS1 における学生のプログラミング行動から学生の苦手意識を分析しており、条件式とループに特に問題があったと指摘している。Berges ら [1] は項目反応理論を使うことで初学者の反応を分類し難易度を評価しており、榊原ら [12] は修正時間の変化の一次近似式からエラーの修正難易度を判断しており、エラーの種類毎のエラー総数と修正時間の平均には相関がないことがわかった。また手動で Syntax error の誤り原因を分析しているもの [5] もある。

OCaml Blockly のようなビジュアルプログラミング言語は様々開発されており、それらの教育実践評価には Logo と Scratch を使用した学生の反応調査 [9] やブロックプログラミング環境の有用性についてのログ分析 [11] などが挙げられる。ビジュアルプログラミング環境が学生のプログラミングモチベーションに与える影響を調査したもの [3] もある。

7 まとめと今後の課題

OCaml プログラミングの初学者がどのようにプログラミング行動を進めるのか把握した。学生は長時間エラーを前に悩み続けることは少なく、比較的短時間で再コンパイルする様子が確認できた。長い時間をかけて修正を行うエラーは Type error であることが多く、データ構造を表す構文やプログラム全体の構造が把握できていないことが原因で発生するものが一定の割合を占めた。この種の難点を解消するのに、OCaml Blockly は有効に働くと期待できる。2020 年度の OCaml Blockly を使用してプログラミングをした学生のデータと、従来のテキストプログラミングでの学生のデータを比較したところ、OCaml Blockly を使用した学生がある程度短時間で課題を済ませている様子が確認できた。また、一部の学生が OCaml Blockly を使用した 2019 年度の学生のテキストプログラミングに移行した後のデータを確認したところ、従来の最初からテキストプログラミングで作業している学生のデータに対して、プログラミング結果が悪化している様子は確認されなかった。

Type error が学生にとって難しいエラーであるということが改めて明らかになったので、その上で本研究室で開発された型デバッガがどのように学習に影響を与えているか考察していくことも検討したい。学生が OCaml Blockly を使用して学習を行った際に現れる効果と影響については、エディタの対応する構文が限られていること、データ数が少ないこと、明確に OCaml Blockly を使用したとわかる学生のその後のデータが取れていないことから曖昧な考察に留まっている。今後データを揃えていくことで OCaml Blockly の有用性を示していけると考えている。

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] Marc Berges and Peter Hubwieser. Evaluation of Source Code with Item Response Theory. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '15, pp. 51–56, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Paulo Blikstein. Using Learning Analytics to Assess Students' Behavior in Open-Ended Programming Tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, LAK '11, pp. 110–116, New York, NY, USA, 2011. Association for Computing Machinery.

[3] Jeffrey B. Bush, Monica R. Gilmore, and Susan B. Miller. Drag and Drop Programming Experiences and Equity: Analysis of a Large Scale Middle School Student Motivation Survey. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE '20, pp. 664–670, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Yuliya Cherenkova, Daniel Zingaro, and Andrew Petersen. Identifying Challenging CS1 Concepts in a Large Problem Dataset. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pp. 695–700, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. All Syntax Errors Are Not Equal. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '12, pp. 75–80, New York, NY, USA, 2012. Association for Computing Machinery.

[6] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2018.

[7] Tsukino Furukawa, Youyou Cong, and Kenichi Asai. Stepping ocaml. In Peter Achten and Heather Miller, editors, *Proceedings Seventh International Workshop on Trends in Functional Programming in Education, TFPIE@TFP 2018, Chalmers University, Gothenburg, Sweden, 14th June 2018*, Vol. 295 of *EPTCS*, pp. 17–34, 2018.

[8] Matthew C. Jadud. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, Vol. 15, pp. 1–25, 2005.

[9] Colleen M. Lewis. How Programming Environment Shapes Perception, Learning and Goals: Logo vs. Scratch. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pp. 346–350, New York, NY, USA, 2010. Association for Computing Machinery.

[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論文集, 2019.

[11] Yoshiaki Matsuzawa, Yoshiki Tanaka, and Sanshiro Sakai. Measuring an Impact of Block-Based Language in Introductory Programming. In Torsten Brinda, Nicholas Mavengere, Ilkka Haukijärvi, Cathy Lewin, and Don Passey, editors, *Stakeholders and Information Technology in Education*, pp. 16–25, Cham, 2016. Springer International Publishing.

[12] 榊原康友, 松澤芳昭, 酒井三四郎. プログラミング初学者におけるコンパイルエラー修正時間とその増減速度の分析. 情報教育シンポジウム 2012 論文集, Vol. 2012, No. 4, pp. 121–128, aug 2012.

[13] 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.