平成28年度 修士論文

ライブプログラミングが
生産性に与える影響に関する実証研究

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 15M37035

今井 朝貴

指導教員

増原 英彦

平成29年1月26日

概要

ライブプログラミング環境は，コードの編集中にプログラムを実行し，その結果を即座に表示することによってプログラミングを支援するようなプログラミング環境である．

一方で，それらがどのようにしてプログラマの生産性を向上するのかは明らかでなかった．これを明らかにすることで，今後ライブプログラミング環境の設計を議論する土台を作成することが本研究の目的である．

本研究では，ライブプログラミング環境での生産性向上に関する以下の 3 つの仮説を立てた．このうち 仮説 1 をライブプログラミング環境のユーザへの質問で，仮説 2 を実証実験により考察する．また，その過程でライブプログラミング環境でのユーザのふるまいを観察，文書化する．

1. ライブプログラミング環境を使っているとき，ユーザは実行結果をより頻繁に確認する
2. 頻繁に実行結果を確認することで，ユーザはフォルトをすぐに発見できる
3. 頻繁に実行結果を確認しフォルトをすぐに発見できる (仮説 2) ならば，全体としての生産性が向上する

質問を行う際には，ライブプログラミング機能の有無を切り替えられる環境を作成し，被験者にいくつかのプログラム課題を実装させた．その結果，アルゴリズムを表わす小さな関数に対しては，実行結果の確認頻度という点において通常の環境と大差がないことがわかった．これは，被験者がどの程度のプログラムを書いたら実行結果を確認するべきかを経験的に学んでおり，小さい関数の場合には途中で実行結果を確認すべきだと考えないからである．大きな関数を実装する課題においては，確認頻度が増加する被験者とそうではない被験者に分かれた．これは普段の確認頻度や，プログラミングスタイルに依存していると考えられる．

実証実験では，事前に作成したプログラミングの過程を再生し，さらに実行結果の確認タイミングを制御できるような実験環境を構築した．これを使い，フォルトが含まれるコードの作成過程を再生し，実行結果を頻繁に確認する場合とそうでない場合でそのフォルトに気づくタイミングが変化するか比較する．計測値からは読みとれないが，多くの被験者は事前の実験と同程度のプログラムであっても，適切な実行結果の確認タイミングを外部から与えた場合には仮説 2 は成り立つだろうと答えた．これは被験者らがよりライブプログラミング環境での経験を積み，適切な実行結果の確認タイミングを学習すれば小規模な関数であってもライブプログラミング環境の恩恵を受けられることを示唆する．

謝辞

本研究を進めるにあたり，数多くのアドバイスをくださった増原英彦教授，青谷知幸助教，研究室メンバ，本実験に参加いだたいた皆様に感謝いたします．

また，経済的，精神的に援助いただいた家族の皆様に深く感謝いたします．

目 次

第1章　はじめに

本章では，本研究で実証研究の対象とするライブプログラミング環境に説明し，その後，ライブプログラミング環境におけるプログラミングスタイルについて解説する．

1.1　ライブプログラミング環境

ライブプログラミング環境は，コードを書いている最中にその実行結果を即座に表示する [5]．これにより，従来のプログラミング環境にあった，コードを書く作業と実行結果の確認という作業の間の遷移をより簡単にすることでプログラミングを支援する [3] (図 1.1, 図 1.2)．

これまで，ライブプログラミング環境の多くは，絵の描画や音の操作などを対象としたものが多かった [13, 3]．しかし， 近年では，ライブプログラミング環境をアルゴリズムや，より大きなプログラムを対象とするものも増えている [9, 12, 11, 8, 15]．

1.1.1　例: Khan Academy の Live-editor

ライブプログラミング環境の一例として， Khan Academy による live-editor [2] を紹介する．Live-editor はブラウザ上で動作する JavaScript のためのライブプログラミング環境

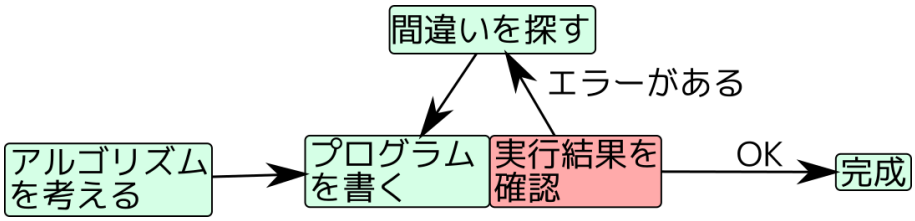

図 1.1: ライブプログラミング環境

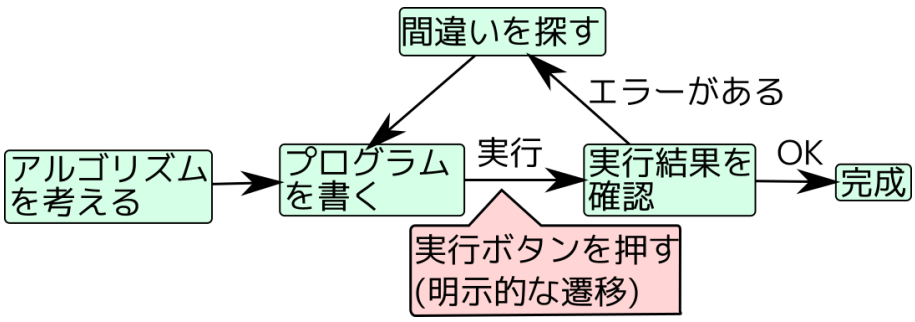

図 1.2: 従来の開発環境

図 1.3: live-editor のスクリーンショット

である．Khan Academy では，ライブプログラミング環境をコンピュータサイエンスのオンラインコースに使用している [7].

図 1.3 は live-editor のスクリーンショットである．左側がコードを記述するエディタであり，右側が実行結果が表示される領域である．Live-editor は，コードが変更された際のフィードバックとして，標準出力の更新と描画の更新が行なわれる．ユーザが左側のソースコードを変更する度にそれが自動的に再実行され，`println` の結果は右下の黒い領域に，`rect` などの描画関数の結果は右上の白い領域に表示される．

1.2 ライブプログラミング環境でのプログラミングスタイル

ライブプログラミング環境でプログラムを書く上では，ユーザは通常と異なるプログラミングスタイルを用いる必要がある．本節では，例を挙げてライブプログラミング環境でのプログラミングスタイルについて説明する．

例として，入力として整数のリストを受け取って， 0 以下の要素は 0 に， 大きい要素はそれを 2 倍したリストを出力する関数 `f` を考える．例えば， `f([1, 0, -4])` は `[2, 0, 0]` を出力として返す．

1.2.1 実験式を先に記述する

まず，ユーザは今から書こうとするプログラム，あるいは関数を実行する式 (実験式と呼ぶ) を書く必要がある．なぜなら，実験式がなければ実行時情報を得ることができないからである．

関数 `f` の例では， `f([1, 0, -4])` を関数 `f` 本体を記述する前に書く必要がある (ソースコード 1.1).

ソースコード 1.1: 実験式 f([1, 0, -4]) を本体を記述する前に記述しておく例

```
var f = function(list){
};
// 実験式，出力をするため println を使用する
println(f([1, 0, -4]));
```

ソースコード 1.2: return 文がないため，フィードバックを得られないプログラム

```
var f = function(list){
    var ret = new Array(list.length);
    for(var i=0; i<list.length; i++){
        ret[i] = list[i] * 2;
    }
};
// Return 文がなく， undefined が表示される
println(f([1, 0, -4]));
```

1.2.2 Return を先に記述する

単純なライブプログラミング環境の場合，実験式だけでなく， println や return などのフィードバックを返す式を先に書く必要がある．例えば，ソースコード 1.2 のように return 文がないコードでは，プログラマはコードを書いている間に何もフィードバックを得ることができず，ライブプログラミング環境の利点を活かせない．

なお，絵を描画するようなプログラムでは，必然的に ellipse などの画面に描画するフィードバックを返す関数を使用するため，この問題は起きづらい．

1.2.3 文法を正しく保つ

プログラムを書いている最中に，できる限りプログラムを文法上正しい形に保つ必要がある．なぜなら，文法が正しくない等の静的なエラーが発生している場合，そもそも実行することができず，フィードバックを得ることができないからである．

関数 f の例では，2 つ の if 文のうち，一方に文法エラーが発生したままもう一方を記述していても，フィードバックを得ることはできない (ソースコード 1.3)．このような場合には，コメントアウトしておく，間違ってはいるが文法上正しいコードをひとまず書いておく等の対策が必要となる．

1.2.4 区切りごとに動作を確認

ライブプログラミング環境は，段階を踏むようなプログラムでより有効に機能する．例えば，ソースコード 1.4 では，一つめの for 文を書いた時点で配列の前半の要素が正しく 2 倍されているかどうか確かめることができる．

また，条件分岐のようなものがあるときにも，その分岐ごとにプログラムの動作を確認することができる (ソースコード 1.5)．

ソースコード 1.3: 2 つの if 文のうち，片方に文法エラーがあるために実行ができない例

```
var f = function(list){
    var ret = new Array(list.length);
    for(var i=0; i<list.length; i++){
        if(list[i] >= 0){
            // 不完全な式
            ret[i] =
        }
        if(list[i] < 0){
            // フィードバックを得られない
            ret[i] = 0;
        }
    }
    return ret;
};
println(f([1, 0, -4]));
```

ソースコード 1.4: 2 つの for 文を含むコード

```
// リストの要素の半分を 2 倍，もう半分を 3 倍するコード
var f = function(list){
    var ret = new Array(list.length);
    for(var i=0; i<ret.length/2; i++){
        ret[i] = list[i] * 2;
    }
    for(var i=ret.length/2; i<ret.length; i++){
        ret[i] = list[i] * 3;
    }
    return ret;
};
println(f([1, 0, -4]));
```

ソースコード 1.5: 2 つの if 文を含むコード

```
var f = function(list){
    var ret = new Array(list.length);
    if(list.length === 0){
        return [1];
    }else{
    }
    return ret;
};
// [] のケースは先に確認できる
println(f([]));
println(f([1, 0, -4]));
```

第2章　目的

これまで，様々なライブプログラミング環境が提案されてきたが，それらがどのようにしてプログラマの生産性を向上するのか，通常のプログラミング環境とどの程度の差が発生するのかは明らかでなかった．

本研究では，ライブプログラミングがどのように生産性向上に寄与するのかを明らかにし，さらにライブプログラミング環境におけるユーザのふるまいを観察，文書化する．それにより，どのようにライブプログラミング環境を設計すれば良いのかを議論することを可能にする．

第3章　仮説

本研究では，ライブプログラミング環境が生産性向上に役に立つかを検証する．しかし，直接生産性が向上するかを測定した場合，どのような理由で向上するのかはわからない．

そこで，本研究では以下の 3 つの仮説を立て，検証する．

1. ライブプログラミング環境を使っているとき，ユーザは実行結果をより頻繁に確認する
2. 頻繁に実行結果を確認することで，ユーザはフォルトをすぐに発見できる
3. 頻繁に実行結果を確認しフォルトをすぐに発見できる (仮説 2) ならば，全体としての生産性が向上する

ライブプログラミング環境を使用している際には，ユーザは実行ボタンを押す等の明示的な行動なしに実行結果を確認することができる．その結果，実行結果を見るという行動を取りやすくなり，より頻繁に実行結果を確認するようになると考えられる (仮説 1)．

実行結果が更新されてかつユーザが意図しない動作を発見したとき，ユーザはフォルトがあることに気づく．フォルトがどこにあるのか推論する上で，以前に確認した実行結果と現在表示されている実行結果を照らし合せて推論することができる．さらに，ユーザがフォルトがあると疑うコードは以前動作を確認してから追加したコード片である．このコード片の大きさは動作確認を頻繁に行なった場合には小さくなり，コード片が小さくなるとフォルトを発見するのが容易になると考えられる (仮説 2)．

頻繁に実行結果を確認した際，フォルトを探す時間は減るが，一方で実行結果を確認するのに使用する時間は増加すると予想される十分に大きなプログラムでは，フォルトを探す時間を減らすことがより大きな影響があり，全体としてプログラムが早く完成すると考えられる (仮説 3)．

本研究では，仮説 1 はインタビューによって確かめ，仮説 2, 3 はこれを測定する実験を設計し確かめる．

第4章　予備実験

予備実験は本実験をどう設計するか，どのような課題を設定するかを検討するために行った．

4.1　予備実験1

最初の実験では，被験者にライブプログラミング環境とそうでない環境を使ってもらいその後のアンケートに答えてもらう．このアンケートの結果から次にどのような実験を行うべきかを考察する．

結果として，あらかじめ課題を細かい関数に分割した上でライブプログラミング環境を使った場合では，被験者のふるまいは通常のプログラミング環境とほぼ変わらないことがわかった．

4.1.1　対象被験者

被験者は，プログラミングの経験者を想定する．プログラミング言語として JavaScript を使用するが，その経験は問わない．

表 4.1 に示すように，実際の被験者は 6 名であり情報学を専攻する学部 4 年生，院生で構成されている．被験者 1 以外には，1.2 節で述べたように `return` 文を先に書くようにというアドバイスをしている．これは 被験者 1 での実験から得たフィードバックであるためである．

予備実験 1 では関数ごとにプログラムに近いアルゴリズムを与えていたため，それを JavaScript に翻訳するだけになってしまう可能性があった．そのため，被験者 5, 6 では関数ごとに与えていたアルゴリズムを取り除いて実験を行なった．その結果，被験者が意図しない動作を観測する回数は増えた．しかし，ライブプログラミングによる恩恵は変わらず限定的であるという結果だった．

被験者番号	学年	JavaScript 経験	プログラミング歴	アルゴリズム
1	M1	なし	4 年	与えられる
2	B4	すこし [1]	3 年	与えられる
3	D1	あり [2]	6,7 年	与えられる
4	M1	なし	5 年	与えられる
5	B4	あり	2 年 9ヶ月	与えられない
6	B4	なし	2 年 9ヶ月	与えられない

表 4.1: 予備実験 1 における被験者構成

[2]チュートリアルをやったことがある程度

[2]Greasemonkey スクリプトを書いたことがある程度

被験者番号	1 問目	2 問目	3 問目	4 問目
1	1 (L)	2 (N)	3 (N)	4 (L)
2	1 (N)	2 (L)	3 (L)	4 (N)
3	2 (L)	1 (N)	4 (N)	3 (L)
4	2 (N)	1 (L)	4 (L)	3 (N)
5	1 (L)	2 (N)	3 (N)	4 (L)
6	2 (N)	1 (L)	4 (N)	3 (L)

表 4.2: 予備実験 1 における課題の割り当て．かっこ内は使用する開発環境を表わし，ライブプログラミング環境を使う場合は L，そうでない場合は N と表記する．

4.1.2 実験手順

本実験は，以下のような順で進める．まず，本研究で使用する開発環境の紹介を行う．その後，簡単に JavaScript のチュートリアルを行う．ここでは， `for` 文 などの簡単な文法および絵を描画する関数を紹介する．その後，それぞれに振り分けられた課題を実行する．

1. 開発環境の紹介
2. JavaScript のチュートリアル
3. 課題の実施
4. アンケート，インタビューの実施

また，被験者は画面を録画し，意図しない動作やフォルトを発見した場合にはマイクに向かって発言をする．これは後に解析を行うためである．

また，プログラムを書くのにかかった時間は気にしないことをあらかじめ被験者に伝えた．これは被験者がプログラムをはやく完成させるために，実行結果の確認を省略することを防ぐためである．

4.1.3 開発環境

対象とするライブプログラミング環境として，1.1.1 節で紹介した live-editor [2] を用いた．

本実験では， live-editor に対して，ライブプログラミング機能を有効，無効にする拡張を行なった．これにより，ライブプログラミング環境であるかだけが差分である二つの環境で実験を行うことができる．

図 4.1 がライブプログラミング機能が有効な状態で，実行ボタンが表示されていない．この状態では，ユーザが左側のプログラムを編集すると自動的にそれが実行され右側に結果が表示される．

図 4.2 は，ライブプログラミング機能を無効にした状態で，実行ボタンが表示されている．ユーザは実行結果を更新する際には手動で実行ボタンを押す必要がある．また，文法エラー等の静的な検査はライブプログラミング機能が有効のときと同様に即座に行われる．

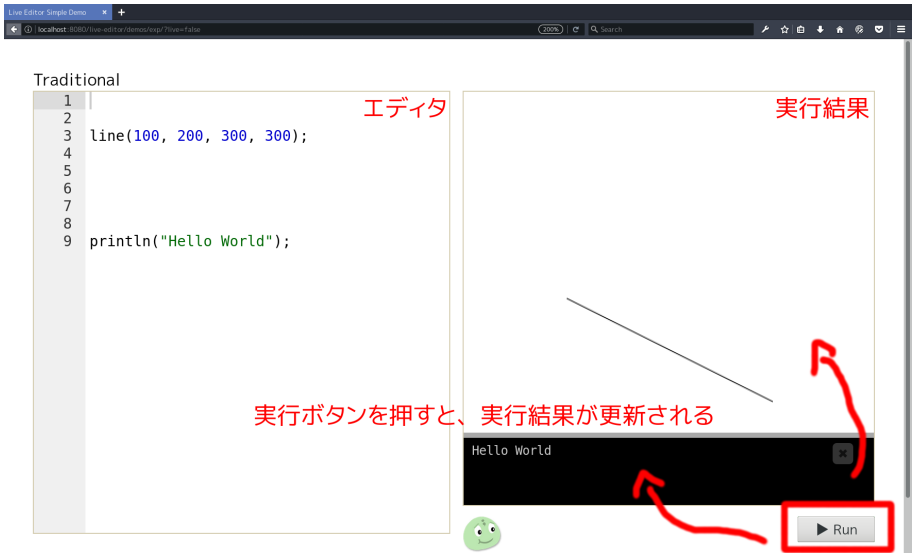

図 4.1: ライブプログラミングでない状態の開発環境

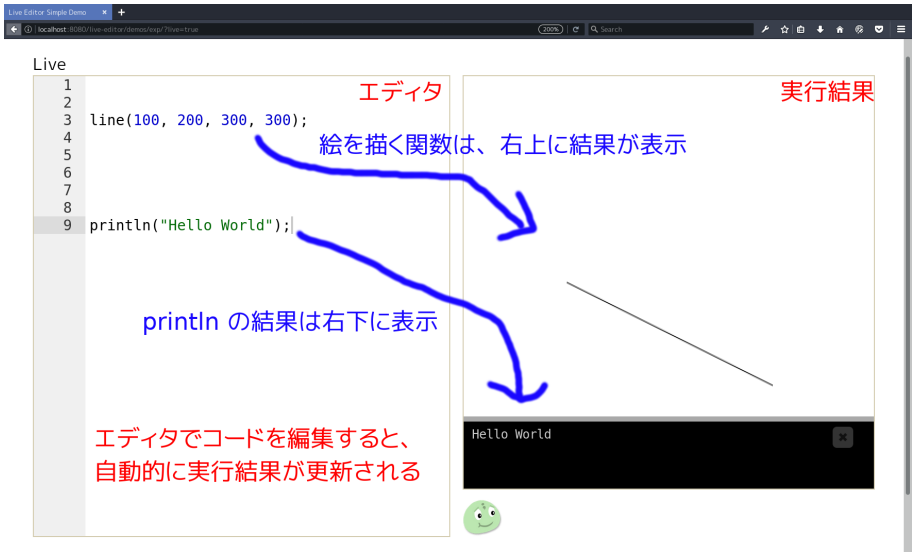

図 4.2: ライブプログラミングである状態の開発環境

4.1.4 課題設定

課題プログラムをあらかじめ複数の関数に分割し，その中身を実装するというテスト駆動開発形式で行った．そのような形式を選択した理由は二つある．一つめの理由は，プログラムの分割方法が被験者ごとに異なることによって結果が左右されることを防ぐためである．また，二つ目の理由は，テスト駆動開発形式がライブプログラミングにおける開発方法と相性が良いと経験的に考えられるからである [6]．

それぞれの関数は，あらかじめ以下の情報が与えられている．

- 何をする関数なのか
- 関数の実行例
- 関数の定義 (名前，仮引数)

- アルゴリズム

これらの情報は，ソースコード 4.1 のような形式で与えられ，被験者は関数の本体を埋める．関数を実装する際には，まず関数の実行例をコード中にコピーし，アルゴリズムを見ながら関数本体にコードを記述する．これにより，関数本体を記述中に実行結果が正しいかどうかを確認することができる．

課題は 4 つあり，アルゴリズムを実装する課題を 2 つ，絵を描画する課題を 2 つ用意した．アルゴリズムを記述する課題では，実行例が関数呼び出しとその結果 (文字列) で与えられる．絵を描画するプログラムの実行例は関数呼び出しとその描画結果の画像が与えられる．この際には被験者が目視で正しさを確認する．

ソースコード 4.1: 関数ごとに与えられる雛形

```
1 // 1. 整数を受け取る.
2 //    それが偶数ならば true, そうでないなら false を返す
3 // 例:
4 //   is_even(2) -> true
5 //   is_even(3) -> false
6 var is_even = function(n){
7 };
```

4.1.4.1 課題 1: 絶対値の和

入力として，数値の配列を受け取りその絶対値の和を計算するアルゴリズムを記述する．関数として，sum，calc_abs， map_abs が定義されている (雛形ソースコード 4.2)．なお，被験者は JavaScript 組み込みの map などの関数は使うことは許されていない．

4.1.4.2 課題 2: 4 つ以上の連続する値の除去

入力として，数値の配列を受け取り 4 つ以上連続で続く値を 0 で置き換えるアルゴリズムを記述する．関数として，consecutive_length，replace_zeros， remove_consecutives_from が定義されている (雛形ソースコード 4.3)．

この課題は課題 1 と同様にアルゴリズムを記述する課題であるが，複雑な for 文等の使用が想定される．

4.1.4.3 課題 3: ゆきだるまの描画

図 4.3 に示すような画像を出力するプログラムを記述する．関数は ground，sun，snowman の 3 つに分けて定義されている (雛形ソースコード 4.4)．

これはライブプログラミング環境が得意であると考えられる典型的な課題である．このようなプログラムは ellipse などの描画関数により直線的に記述され，さらに座標などのパラメータが即値としてプログラム内に現れるためフィードバックを得やすいからである．

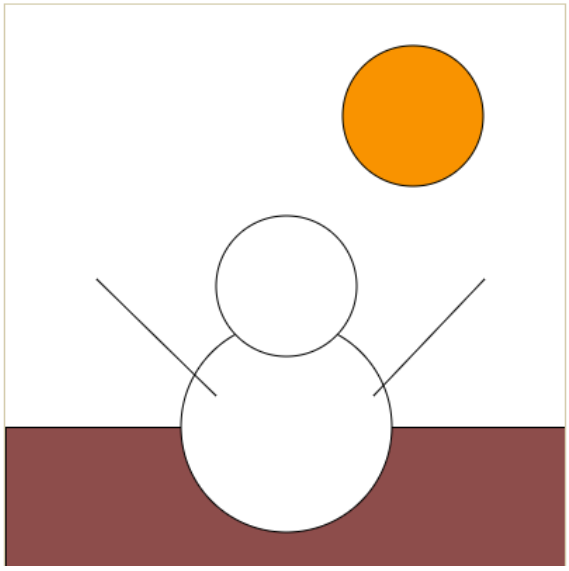

図 4.3: 課題 3 の出力

4.1.4.4　課題 4: 棒グラフの描画

数値の配列を受け取り，それを表わす棒グラフ (図 4.4 を描画するプログラムを記述する．関数 `oneBlock`，`nBlocks`，`chart` があらかじめ定義されている．また，ブロックの高さ `BLOCK_HEIGHT` および 幅 `BLOCK_WIDTH` もあらかじめ与えられる．

この課題は絵を描画する課題が，パラメータの微調整が存在しない．そのため，課題 3 と比べライブプログラミング環境の恩恵を受けづらいと考えられる．

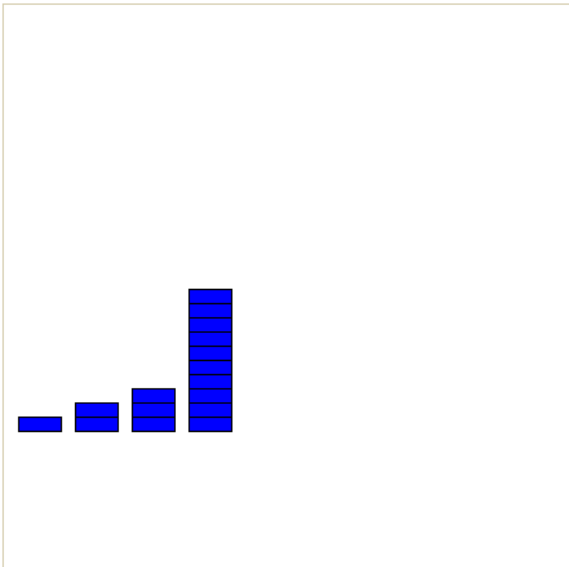

図 4.4: 入力として `[1, 2, 3, 10]` を受けとった際の課題 4 の出力

4.2 予備実験 1 の結果

実験で得られたアンケート，および被験者を観察した結果について述べる．このアンケートは筆者が直接，被験者に質問した回答である．

4.2.1 質問 1: 更新され続ける実行結果が邪魔になることはあるか？

実際の質問 “ライブプログラミング環境では，実行結果は自動的に更新されて表示されます．この表示が集中を乱したり，邪魔に感じたことがありましたか？はい，の場合は，特にどういう状況でそう感じたか教えてください．”

被験者らの返答 ほとんどの被験者と “いいえ” と答えた．理由として画面に大きな変化がないことや，自分が見たいときだけ見るからというものが挙げられた．また，被験者 2 のみエラー表示が気を引くと答えた．これは，例えば `for` 文のかっこを閉じていないために発生するエラーなどがそうである．

4.2.2 質問 2: アルゴリズムの記述，絵を描くプログラムの記述において，ライブプログラミングが役に立つと感じたか？

実際の質問 “課題のプログラムには「計算」するものと,「絵を描く」よりのものが二つありました．この二つで，ライブプログラミングが役に立つと感じましたか？”

被験者らの返答 アルゴリズムより課題に関しては，実行ボタンを押さなくて良いのが楽であったがそれ以上の恩恵はなかったとのコメントが多かった．

絵を描く課題では，被験者 1, 4 は微調整で役に立つと報告した．被験者 2, 3 はアルゴリズムを記述する課題と変わらず，実行ボタンが押すのが楽になった程度であると返答した．

4.2.3 質問 3: 課題プログラムの分割方法に違和感はあったか？

実際の質問 “課題の関数はあらかじめ分割して定義してありました．これについて特に違和感はありませんでしたか？自分が書くとしたら違うやり方をする場合には教えてください．(例: 全部一つの関数に書く，等)”

被験者らの返答 被験者 1, 2 が “特になかった” と報告した．また，残り 2 名の被験者は普段ならば `map` などを使うとコメントした．被験者 3 は書いてある通りにやったため，分割に関しては特に何も考えていなかったと報告した．

4.2.4 質問 4: ライブプログラミング環境で意図しない動作は早期に発見できそうか？

実際の質問 “今回，私達の想定は「ライブプログラミングにより，プログラマは意図しない動作を早期に発見，コードの誤り発見も容易になる」でした。今回の実験中で、ライブでないプログラミング環境を使っているとき、プログラムが意図しない動作をしたことがあり

ましたか? あった場合、そのとき、ライブプログラミング環境であればより早く気づくことができただろうと思いますか?”

被験者らの返答　被験者 1 は，意図しない動作があったがライブプログラミング環境でもそれを早く気づくことはないだろうと回答した．この理由として， return 文を最後に書いていたことを挙げた．

被験者 2, 3 は，何度か実行ボタンを押すのを忘れることがあったと報告した．

被験者 3, 4 は，意図しない動作があり，もしライブプログラミング環境を使っていれば実行ボタンを押す数秒程度は省略できたと回答した．このグループは 被験者 1 と違い，実験前に return 文を先に書くようにアドバイスし，実際にその通りにプログラムを記述したが，被験者 1 とほとんど差は見られなかった．被験者 3 は，その理由としてほとんどの関数を完全に記述してから実行していたので，変わらなかったということを挙げた．

被験者の何人かは，実行ボタンを押すことの手軽さについて言及した．今回のライブでない環境において，実行ボタンを押すことは簡単であったため，それほど差が出なかったのではないかと予想していた．

4.2.5　質問 5: ライブでない環境での意図しない動作は発見が遅れるか？

実際の質問　“今回の実験中で、ライブプログラミング環境を使っているとき、プログラムが意図しない動作をしたことがありましたか? あった場合、ライブでないプログラミング環境であれば気づくのが遅れただろうと思いますか?”

被験者らの返答　被験者 1, 2 は，特に意図しない動作を発見しなかった．

被験者 3, 4 は，いくつかの意図しない動作に遭遇したものの，ライブプログラミングでなかったとしても変わらないだろうと回答した．これは実行結果を確認するタイミングがライブでない環境であっても変わらないという理由からである．

4.2.6　発生した誤り

後の実験のため，発生した被験者の誤りを列挙する．

4.2.6.1　実行ボタンの押し忘れ

ライブプログラミング環境で課題を行なった後，そうでない環境でプログラムを記述するとき，実行ボタンを押すのを忘れたために実行結果が更新されず，正当なプログラムであるにもかかわらず間違いだと誤認することがあった．

4.2.6.2　for 文 でのインデックス間違い

特にアルゴリズムを与えないグループで，for 文内部，およびインデックスの更新式等での間違いが観測された．

4.2.6.3 break のし忘れ

例えば，ソースコード 4.6 のように，for 文や while 文などで本来 break 文を書くべきところで書かないことがある．

4.2.6.4 return するものの反転

真偽値を返すような関数を実装する際に，本来返すべき値と反対の値を返してしまう場合がある．

4.2.6.5 計算式の誤り

式として記述した計算式が間違っていることがある．これはアルゴリズムを与えないグループでよく観察された．

4.2.6.6 関数の引数の順番の誤り

定義した関数の引数の順番を間違うことがある．このような場合，被験者は関数の本体が間違っていると誤認する場合もあった．

4.2.6.7 関数の引数が不十分

定義した関数の引数のうち，呼び出し元でそのいくつかが与えられない場合がある．このようなとき， JavaScript では足りない引数に undefined という値が入り，そのまま実行される．これによって，被験者は関数の本体が間違っていると誤認する場合があった．

4.3 予備実験 1 の考察

予備実験 1 において，被験者はライブプログラミング環境を使っても，実行ボタンを押す手間が省けることのみを利点と感じることがわかった．被験者はライブプログラミングであっても，そうでなくても同じタイミングで実行結果を確認していると予想される．その理由として，予備実験 1 で与えた課題では，既に細かく関数が分割されており，一つの関数を完成させることが容易であったことが挙げられる．これによりプログラムの実行結果を確認するタイミングは，関数を完成させた直後，あるいは完成後に編集した後のみになる．

被験者によると，絵を描画するという課題のほうがアルゴリズムを記述する課題よりライブプログラミングが役に立つと感じられるようである．この理由の一つは，絵を描画するような関数，例えば ellipse のような関数は，独立に実行されることである．例えば，ellipse(300, 300, 300); ellipse(400, 400, 400); のようなプログラムの場合，一つめの ellipse の呼び出しを記述した時点でプログラムの実行結果を確認できる．さらに，絵を描画するという課題にはパラメータの調整という作業が存在することが挙げられる．絵を描画する課題であるがほとんどパラメータが存在しない課題 4 においては，課題 3 ほどライブプログラミングの利点が活かせなかったという意見がある．

また，この実験ではアルゴリズムの正当性を確かめるために目視で確認する必要があり，そのコストが大きいために実行結果の確認をできるだけ少なくしようと考える可能性があ

る．これはユニットテスト機構のように正当性を色で表示したり，そのテストケースが何を表わしているかを表示することにより改善されると考えられる．これについては予備実験 2 で追試する．

4.4 予備実験 2

予備実験 1 の結果を踏まえアルゴリズムを記述する課題のうち，関数の一部が独立に実行可能でその実行結果をすぐに確認できるようなものについて同様に実験を行い，観察とインタビューを行う．

ここでの被験者は，予備実験 1 の被験者のうち 4 名である (表 4.3)．このうち，被験者 1, 4, 5 には 1.2 節で述べた方法を紹介した．課題は 2 つ用意し，表 4.4 のように振り分けた．

被験者番号	学年	JavaScript 経験	プログラミング歴
1	M1	なし	4 年
2	B4	すこし [3]	3 年
4	M1	なし	5 年
5	B4	あり	2 年 9ヶ月

表 4.3: 予備実験 2 における被験者構成

被験者番号	1 問目	2 問目
1	1 (L)	2 (N)
2	2 (L)	1 (N)
4	2 (N)	1 (L)
5	1 (N)	2 (L)

表 4.4: 予備実験 2 における課題の割り当て．かっこ内は使用する開発環境を表わし，ライブプログラミング環境を使う場合は L，そうでない場合は N と表記する．

4.4.1 課題設定

課題のプログラムは，関数一つに条件分岐を多量に含み，一部だけ実行して確かめることができるものとする．

また，予備実験 1 と違い，与える雛形にあらかじめテスト用のコードを含んでおり，被験者は自分自身で値の正当性を確かめる必要がないものとする．

4.4.1.1 課題 1

ソースコード 4.7 は課題 1 で与えられる雛形である．この課題では，入力として配列 2 つを受けとり，それらについてある条件ならばある操作をするという `if` 文を 4 つ書くことを要求する．

4.4.1.2 課題 2

ソースコード 4.8 は課題 2 で与えられる雛形である．この課題も課題 1 と同様に，入力として配列 2 つを受けとり，それらについて 4 つの `if` 文を要求する．

4.4.2 結果

被験者 2 は課題 1 においてプログラムを最後まで書いてから確認する傾向があった．また，ある箇所で文法エラーが発生していても気にせず，他の箇所のコードを書くことが多く，実行結果を確認できないことが多いようだった．課題 2 からは，実行結果をこまめに観測したほうが全体として早く終わることを学習し，こまめに実行するようにしたと述べた．

被験者 1 は，ライブプログラミング環境を用いた時のほうが小刻みに実行結果を確認したと報告した．予備実験 1 と違い，予備実験 2 の対象は大きなプログラムであったこと，実行ボタンを押す際にはまとまったプログラムを実行したいという理由が挙げられた．また，被験者 1 は実験中にラップトップのキーボードが故障し，急遽外付けのキーボードを使用したため，実行ボタンを押す行為が大変になったと報告した．ただし，今回の実験では実際に発生した意図しない動作については，ライブプログラミングであってもそうでなくても気づく早さはそれほど変わらないだろうと予想した．

被験者 4 は，事前に伝えた方法とまったく違うスタイルである関数型プログラミングにより課題をこなした．`map` や `reduce` 等を使用する方法でプログラムを記述していたため，あまりライブプログラミングの恩恵が受けられなかったと報告した．これはそれぞれの高階関数に対し，面倒であるという理由でテストを記述しなかったこと，関数を必要になったときに文法エラーをそのままに他の関数を実装したことなどが理由として挙げられる．さらに，プログラムの明確な区切りが `if` 等の条件分岐のみになってしまい，細かいフィードバックを得ることができなかった．

被験者 5 は，テストケースに与えられた場合ごとに `if` 文を書いていき，その都度実行結果を確認した．これは事前にプログラミングスタイルを紹介したためだと考えられる．

4.4.3 考察

被験者 2 は事前にプログラミングスタイルを紹介していなかったため，ライブプログラミング環境で被験者自身のプログラミングスタイルをそのまま適用した．しかし，その後の課題 2 において，ライブプログラミング環境でないにもかかわらず，ライブプログラミング環境よりも実行頻度を増やすことで生産性が向上したと考えられる．これにより，被験者 2 の場合は実行頻度が少ないライブプログラミング環境よりも，実行頻度が多いライブでない環境のほうが生産性が高いと予想される．

被験者 1, 5 では，ライブプログラミング環境においてどのようなプログラミングが有効かを事前に教えることによって被験者の実行結果の確認頻度が増加することを観測した．

また，ライブでない環境を使用した際にはより関数定義を行う頻度が増えることが観測された．これはライブプログラミング環境を使用した際には逐次的なプログラムを書いたほうがフィードバック等の面で利益があるが，そうでないときには関数定義をして分割したほうが利益があると判断すると思われ，実際に被験者 1 はそれに同意した．

また，今回の課題ではプログラムの小さな区切り，例えば一つの `for` 文等に対するフィードバックを得た被験者は少なかった．その理由として，被験者の習熟度やプログラムの書き

方が挙げられる．被験者 1, 2 の例では，そのようなコードが存在していたことを後に指摘すると，それには気づかなかったという回答があった．

ソースコード 4.2: 課題 1 の雛形

```
// 制限時間: 45分
// 課題: 整数の配列が与えられる. すべての要素の絶対値の和を計算しなさい.


// 1. 数値を受けとってその絶対値を返す関数
// アルゴリズム:
//   もし入力が 0 より大きいならば入力を,
//   そうでないならば入力に -1 を掛けた数を返す
// 例:
//   calc_abs(-3) -> 3
//   calc_abs(3)  -> 3
//   calc_abs(0)  -> 0
var calc_abs = function(n){
};

// 2. 数値のリストを受けとって, その要素の絶対値を要素とするリストを作る
// アルゴリズム:
//   1. 出力として, 入力と同じ長さのリストを作る
//   2. 要素の数だけ以下を繰りかえす
//     i. ある要素の位置を x として, 出力の x 番目に 入力の x 番目の絶対値を代入する.
//        (calc_abs を使う)
// hint: n の長さの配列は new Array(n) で作る
// 例:
//   map_abs([-1, 2, 3]) -> [1, 2, 3]
//   map_abs([]) -> []
var map_abs = function(arr){
};


// 3. 数値のリストを受けとって, その要素の和を計算する
// アルゴリズム:
//   1. 出力として cnt を 0 とする
//   2. すべての要素を cnt に足しあわせる
// 例:
//   sum([1,2,3]) -> 6
//   sum([-1, 2]) -> 1
//   sum([]) -> 0
var sum = function(arr){
};

// 4. 数値のリストを受け取って, その要素の絶対値の和を計算する
// アルゴリズム:
//   1. 数値のリストから, その絶対値のリストを作る
//   2. その絶対値のリストの和を計算する
// hint: map_abs と sum を使う
// 例:
//   abs_sum([-1, 2, 3]) -> 6
//   abs_sum([-1, -2])   -> 3
//   abs_sum([]) -> 0
var abs_sum = function(arr){
};
```

ソースコード 4.3: 課題 2 の雛形

```
1  // 1. 配列とあるインデックスが与えられる.
2  //    そのインデックスの要素と同じ要素がいくつ続くかを計算する
3  // アルゴリズム:
4  //   1. 返り値を ret とする
5  //   2. 与えられたインデックスから最後まで順に要素を見ていき,
6  //      与えられたインデックスの要素と同じだったら ret に 1 を加算,
7  //      違ったらそこで ret を返す
8  //   3. 最後まで違う要素がなければ, ret を返す
9  // 例:
10 //   consecutive_length([1, 1, 1, 1], 0)    -> 4
11 //   consecutive_length([1, 1, 0, 1], 0)    -> 2
12 //   consecutive_length([1, 1, 3, 3, 4], 2) -> 2
13 var consecutive_length = function(arr, index){
14 };
15
16 // 2. 配列とあるインデックス (index), 長さ (length) が与えられる
17 //      そのインデックスから長さ分の要素を 0 に置き換えた(新しい)配列を返す
18 // アルゴリズム:
19 //   1. 配列をコピーする (.slice(0)を使う)
20 //   2. コピーした配列の index から index+length-1 番目を 0 に置き換える (繰り
       返しを使って)
21 // 例:
22 //   replace_zeros([1, 1, 1, 1], 0, 4)    -> [0, 0, 0, 0]
23 //   replace_zeros([1, 1, 0, 1], 3, 1)    -> [1, 1, 0, 0]
24 //   replace_zeros([1, 1, 3, 3, 4], 2, 2) -> [1, 1, 0, 0, 4]
25 var replace_zeros = function(arr, index, length){
26 };
27
28 // 3. 配列とあるインデックスが与えられる.
29 //    もし, そのインデックスから 4 つ以上連続で同じ数が続いていれば, それを 0
       に置き換えた配列を返す
30 // アルゴリズム:
31 //   1. いくつ連続しているかを consecutive_length を使って計算する
32 //   2. もし, 4 つ以上連続しているならば, replace_zeros をつかって置き換える
33 //      そうでないならば, 入力の配列をそのまま返す
34 // 例:
35 //   remove_consecutives_from([1, 1, 1, 1], 0) -> [0, 0, 0, 0]
36 //   remove_consecutives_from([1, 1, 1, 1], 1) -> [1, 1, 1, 1]
37 //   remove_consecutives_from([1, 1, 1, 1, 1, 1], 1) -> [1, 0, 0, 0, 0, 0]
38 var remove_consecutives_from = function(arr, index){
39 };
40
41 // 4. 配列が与えられる.
42 //    配列中の連続する 4 つ以上の同じ数を 0 で置き換える
43 // アルゴリズム:
44 //   1. 返り値を ret とし, 初期値を引数とする
45 //   2. i を 0 から 配列の長さ-1 として,
46 //      ret を ret の i 番目から 4 つ以上連続した数を除去したもの (
       remove_consecutives_from を使う) でくりかえし上書きする
47 //   (0 かどうかチェックしない. 0 をもう一度 0 で上書きしても問題ないため)
48 // 例:
49 // remove_consecutives([1, 1, 1, 1]) -> [0, 0, 0, 0]
50 // remove_consecutives([1, 1, 1, 1, 1]) -> [0, 0, 0, 0, 0]
51 // remove_consecutives([1, 1, 1, 1, 2, 2, 2, 2]) -> [0, 0, 0, 0, 0, 0, 0, 0]
52 // remove_consecutives([1, 1, 2, 2, 1, 1]) -> [1, 1, 2, 2, 1, 1]
53 // remove_consecutives([1, 2, 2, 2, 2, 1]) -> [1, 0, 0, 0, 0, 1]
54 // remove_consecutives([2, 2, 2, 2, 3, 2 ,3, 3, 3, 3, 3, 4]) -> [0, 0, 0, 0, 3,
       2, 0, 0, 0, 0, 0, 4]
55 var remove_consecutives = function(arr){
56 };
```

ソースコード 4.4: 課題 3 で与えられる雛形

```
1  // 1. 地面をかく
2  // 参考画像2, 地面単体を参照
3  // アルゴリズム:
4  //   1. 塗り潰し色を茶に変更
5  //   2. 下部に四角を描く. ただし, x座標は 0, 幅は 400 (画面幅) にする
6  // 例:
7  //   ground() -> 参考画像2 の通り
8  var ground = function(){
9  };
10
11 // 1. 雪だるまをかく
12 // 参考画像3, 雪だるま単体を参照
13 // アルゴリズム:
14 //   1. 塗り潰し色を白に変更
15 //   2. 下の円をかく
16 //   3. 上の円をかく
17 //   4. 左手をかく
18 //   5. 右手をかく
19 // 例:
20 //   snowman() -> 参考画像2 の通り
21 var snowman = function(){
22 };
23
24
25 // 1. 太陽をかく
26 // 参考画像4, 太陽単体を参照
27 // アルゴリズム:
28 //   1. 塗り潰し色をオレンジに変更
29 //   2. 円をかく
30 // 例:
31 //   sun() -> 参考画像2 の通り
32 var sun = function(){
33 };
```

ソースコード 4.5: 課題 4 で与えられる雛形

```
// 課題: 整数の配列が与えられる. それを表わす棒グラフを描く.
// 棒グラフは小さいブロックをいくつか使うことで描く.
// 参考画像3 chart([1, 2, 3, 10], 10, 300)の結果 を参照


// 定数
var BLOCK_WIDTH = 30;
var BLOCK_HEIGHT = 10;

// 1. 左上が x, y で幅が BLOCK_WIDTH, 高さが BLOCK_HEIGHT の四角を描く
// 参考画像1, oneBlock(10, 10)の結果を参照
// アルゴリズム
//   1. rect 関数を使って左上が x, y で幅が BLOCK_WIDTH, 高さが BLOCK_HEIGHT の四角を描く
// 例:
//   oneBlock(10, 10) -> 参考画像1
var oneBlock = function(x, y){
};

// 2. 縦にブロックを n 個描く.
//    ただし, 一番下のブロックの下の y 座標が ground,
//    左の x 座標が (引数の) x とする.
// 参考画像2, nBlocks(3, 90, 50) の結果を参照
// アルゴリズム:
//   1. 一番上のブロックの上の座標を計算する
//      ground から, ブロックの高さのブロックの数倍を引いたものである
//   2. 1 で計算した座標からブロックの高さをずらしながら n 個のブロックを描く
// 例:
//   nBlocks(3, 90, 50) -> 参考画像2
var nBlocks = function(n, x, ground){
};


// 3. 与えられた配列を表わす棒グラフを座標 x , 下を ground にして描く
// 参考画像3 chart([1, 2, 3, 10], 10, 300)の結果 を参照
// アルゴリズム:
//   1. 塗り潰し色を青に変更する
//   2. nBlocks を使って配列の各要素を描く
//      その際の x 座標は BLOCK_WIDTH + 10 ずらしながら定義する
// 例:
//   chart([1, 2, 3, 10], 10, 300) -> 参考画像3
var chart = function(arr, x, ground){
};
```

ソースコード 4.6: 本来 break を書く必要があるが, 書き忘れてしまう例

```
// a のうち, b にも含まれるものを数える
var ans = 0;
for(var i=0; i< a.length; i++){
  for(var j=0; j<b.length; j++){
    if(a[i] === b[i]){
      ans++;
      // ここで break する必要がある
    }
  }
}
```

ソースコード 4.7: 課題 1 で与えられる雛形

```
// 1. もし，a の長さが 0 なら， b の要素の和を返す
// 2. もし，a に 0 が 3 つ以上含まれるなら， a と b が同じか判定 (:= 長さが同じ，各要素が同じ)
// 3. もし，a の最後の要素が 0 なら， a と b の要素を交互に含む配列を作る．
//    ただし，どちらかのほうが短い場合には，a[0], b[0], ... b[3], b[4] のように余った要素が最後に続く
//    (ex. op([1,0], [4,6,3]) -> [1, 4, 0, 6, 3])
// 4. それ以外の場合，a の要素のうち， b にも含まれるものを数える
// 例についてはテストを参照．
var op = function(a, b, n){
};

println("-- a の長さが 0 なら， b の要素の和を計算する");
test("a が空， b が空",
     op([], []),
     0);
test("a が空， b が空でない",
     op([], [1, 2, 3]),
     6);

println("-- a に 0 が 3 つ以上含まれていたら， a と b の等価性を判定する");
test("a と b が等しい",
     op([0,0,1,0], [0,0,1,0]),
     true);
test("a と b の長さが違う (a < b)",
     op([0,0,1,0], [0,0,1,0,0]),
     false);
test("a と b の長さが違う (b > a)",
     op([0,0,1,0, 0], [0,0,1,0]),
     false);
test("a と b の要素が違う",
     op([0,0,1,0], [0,0,1,1]),
     false);

println("-- a の 最後の要素が 0 だったら， a と b を交互に混ぜた配列を作る");
test("a と b の長さが等しい",
     op([1, 2, 3, 0], [4, 5, 6, 7]),
     [1, 4, 2, 5, 3, 6, 0, 7]);
test("a と b の長さが違う (a < b)",
     op([1, 2, 3, 0], [4, 5, 6, 7, 8, 9]),
     [1, 4, 2, 5, 3, 6, 0, 7, 8, 9]);
test("a と b の長さが違う (a > b)",
     op([1, 2, 3, 0], [4, 5]),
     [1, 4, 2, 5, 3, 0]);
test("a と b の長さが違う (b が長さ 0)",
     op([1, 2, 3, 0], []),
     [1, 2, 3, 0]);

println("-- それ以外の場合は， a の要素のうち， b にも含まれるものを数える");
test("b に含まれない要素がある",
     op([0, 1], [1]),
     1);
test("b に完全に包含されている",
     op([0, 1], [0, 1, 2]),
     2);
test("b に二度現れても，2回カウントはしない",
     op([0, 1], [1, 1]),
     1);
```

ソースコード 4.8: 課題 2 で与えられる雛形

```
1  // op 関数は， 整数の配列 a, b を受け取り，以下の操作をする．
2  // 1. もし，a の各要素の積が 10 以上なら，a と b の各要素の積の和 (内積) を作る
3  //    ただし，どちらかが短い場合は，足りない部分を 0 として計算する．
4  // 2. もし， a が b の部分配列 (:= b のある連続した要素が a と一致する) なら，
5  //    a と b を辞書式順序で比較して， a <= b なら， true，そうでないなら false を返す
6  //    辞書式順序 := 両方の配列を前の要素から比較して，どちらかが小さいならば，全体としてその配列が小さい．
7  //    ex. [1, 2, 3] <= [1, 2, 4]
8  //        [1, 2, 3, 4] >= [1, 2, 3] (途中でどちらかの要素が足りなくなったら，長いほうが大きい)
9  // 3. もし， a と b が同じ長さなら， b を反転した配列を返す．
10 // 4. それ以外のとき， a と b を連結した配列の偶数番目の要素を持つ配列を返す
11 // 細かい例についてはテストを参照．
12 var op = function(a, b){
13 };
14
15
16 println("-- もし， a の要素の積が 10 以上なら，a と b の各要素の積を各要素とした配列の和 (内積) を作る");
17 test("a と b の長さが同じとき",
18      op([1, 2, 3, 4], [3, 4, 5, 6]), 50);
19 test("a と b の長さが違うとき (a > b)",
20      op([1, 2, 3, 4], [3, 4, 5]), 26);
21 test("a と b の長さが違うとき (a < b)",
22      op([1, 2, 3, 4], [3, 4, 5, 6, 7]), 50);
23
24 println("-- もし， a が b の部分配列なら， a と b を辞書式順序で比較して， a <= b なら， true，そうでないなら false を返す");
25 test("a と b が同じとき (a と b の長さが同じとき)",
26      op([1, 2, 3], [1, 2, 3]),
27      true);
28 test("a < b の場合",
29      op([1, 2, 3], [1, 2, 3, 4]),
30      true);
31 test("a > b の場合",
32      op([1, 2, 3], [1, 1, 2, 3]),
33      false);
34 test("末尾に a が現れるとき",
35      op([3, 3], [4, 1, 3, 3]),
36      true);
37 test("a が空のとき， a はいつも b より同じか小さい",
38      op([], [1, 2, 3]),
39      true);
40
41 println("-- もし， a と b が同じ長さなら， b を反転した配列を返す");
42 test("b が 空でない時",
43      op([3, 1, 2], [1, 2, 3]),
44      [3, 2, 1]);
45
46 println("-- それ以外のとき， a と b を連結した配列の偶数番目の要素を持つ配列を返す");
47 test("b のほうが長いとき",
48      op([1, 2], [3, 4, 5]),
49      [2, 4]);
50 test("a のほうが長いとき",
51      op([1, 2, 3], [4, 5]),
52      [2, 4]);
```

第5章　本実験 1

予備実験 1 の結果から，細かい単位で正しく関数に分割した際にはあまりライブプログラミングが役に立つことはないと予想される．また，予備実験 2 の結果からはライブプログラミング環境に適したプログラミングを行えば，より頻繁に実行結果を確認することが予想できる．

本実験 1 では，もしライブプログラミングに適した方法でプログラムを書いたときに，実行結果の確認頻度が増加することによって，プログラム中の間違いに早く気づけるかどうかを測定する．

具体的には，あらかじめ録画されたプログラムを書く過程を見て，どの段階で間違いに気づくかを測定する．ただし，実行結果を確認できるタイミングを被験者ごと，課題ごとに変えることで実行結果の確認頻度を制御する．

5.1　対象被験者

予備実験 1, 2 と同じ被験者を対象とする (表 5.1)．よって，被験者はある程度ライブプログラミング環境および JavaScript を使用したことがある．

被験者番号	学年	JavaScript 経験	プログラミング歴
1	M1	なし	4 年
4	M1	なし	5 年
5	B4	あり	2 年 9ヶ月

表 5.1: 本実験 1 における被験者構成

5.2　開発環境

図 5.1 は，本実験 1 で使用する開発環境のスクリーンショットである．live-editor に対し，あらかじめ作られたプログラムを少しづつ再生できるような拡張を施した．次へと書かれたボタンを押すと，次のプログラムに遷移する．また，変更箇所はグレーにハイライトされるようになっている．

また，課題ごとにあらかじめ実行結果が更新されるタイミングが設定されている．次のプログラムに遷移した際に実行結果が更新されると，右側の実行結果が赤く枠取りされる．この機能により対照実験が可能となる．

図 5.1: 本実験 1 での開発環境

5.3 課題設定

5.2 節で紹介した開発環境を使い，複数箇所にフォルトがあるプログラムのコーディング過程を再生し，いつフォルトに気づけるかを測定する．被験者は実行結果が更新されたら必ず確認し，フォルトを発見したら画面下のテキストボックスにその旨を記載する．

対象とするプログラムは予備実験 2 と同一とする．これに関しては，既にどのようなフォルトが発生するかわかっているためである．また，それによって被験者は強い推論が働く可能性があるが，被験者 1, 5 によると，書き方が違う等の理由で強い影響はなかったとのことである．

5.3.1 課題 1

課題 1 のソースコード 5.1 には，合計で 5 つの課題がある．課題 1.2 と 1.3，および 1.2 と 1.4 は互いに実行結果に影響しあっている．

5.3.2 課題 2

課題 2 のソースコード 5.1 には，合計で 10 つのフォルトがある．このうち，フォルト 2.6 および 2.9 は JavaScript 特有のフォルトであり，かつフォルト 2.5，2.7 を修正することによって見つかるフォルトのため発見するのが難しいと考えられる．

5.4 本実験 1 の結果

本実験では，フォルトにいつ気づいたかを測定しまとめる．また，インタビューも行なったので，その結果も掲載する．

```
var op = function(a, b){
  // ※ count0 の計算は省略
  if(count0 >= 3){
    var ret = true;
    return ret;
  }
};
```

1. 返り値を定義

```
var op = function(a, b){
  var count0 = 0;
  if(count0 >= 3){
    if(a.length !== b.length){
      return true;
    }
    var ret = true;
    return ret;
  }
};
```

2. 長さが違う場合の if 文を記述

```
var op = function(a, b){
  // ※ count0 の計算は省略
  if(count0 >= 3){
    if(a.length !== b.length){
      return true;
    }
    var ret = true;
    for(var i=0; i<a.length; i++){
      ret = ret && a[i] === b[i];
    }
    return ret;
  }
};
```

3. 長さが同じ場合のコードを記述

図 5.2: 本実験 1 でのプログラム例

5.4.1 被験者の割り当て

被験者一人につき，二つの課題を割りふった．二つの課題のうち，一つは大きな if 文の最後のみで実行結果が更新され，もう一つの課題は，より細かい単位で実行結果が更新されるように設定されている (表 5.2)．

被験者番号	1 問目	2 問目
1	1 (L)	2 (M)
4	2 (M)	1 (L)
5	1 (M)	2 (L)

表 5.2: 本実験 1 における課題の割り当て．実行結果の確認タイミングが少ないものを L，多いものを M と記載する．

実際の例としては，引数として与えられた配列 a，b があったとき，a に 0 が 3 つ以上含まれていれば，a と b の同一性を判定するという課題がある．この課題では，図 5.2 のようにコーディング過程が分割される．この際，細かい単位で実行結果が更新されるグループは，1,2,3 の全てのコードで実行結果が更新される．そうでないグループは， 3 のコードのみで実行結果が更新される．

被験者	1.1	1.2	1.3	1.4	1.5	1.6
1	✓	△	✓	×	✓	✓
4	✓	✓	△*	△*	✓	×
5	✓	✓	✓	✓	✓	✓

被験者	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10
1	✓	△	△	✓	✓	△	△*	×	×	✓*
4	✓	✓	✓	✓	△	×	×	✓	×	×
5	✓	✓	✓	✓	×	×	△	×	×	✓

表 5.3: 本実験 1 における計測結果．✓ はフォルトが含まれるコードが出て，すぐにそれに気づいた場合，△ はすぐには気づかず，後で気づいた場合，× は最後まで気づかなかった場合．* は気づいたが，修正内容が間違っている場合

5.4.2 計測結果

表 5.3 が本実験 1 における計測結果である．フォルトをいつ発見したかを以下の 3 段階に分けて表記する．

1. フォルトが含まれるコードが現れてから次のコードに行く前にそのフォルトに気づいた場合
2. フォルトが含まれるコードが現れて，すぐには気づかず次のコードに進み，後でフォルトに気づいた場合
3. フォルトに最後まで気づかなかった場合

ただし，本実験 1 の場合では次に書かれるものをある程度予測する必要がある．例えば，図 5.3 では，必要である `break` が次の更新で書かれる可能性がある．しかし，書かかれない可能性もあり，それを被験者は前もって判断することができない．そのような場合のとき，被験者がマイクに向かって前もってその旨を発言していればそれを考慮してデータを修正している．

5.4.3 インタビュー

実験後，筆者が直接被験者に対していくつか質問を行なった．

5.4.3.1 プログラムの書き方に違和感はあったか？

被験者 1 は特になかったと報告した．また，被験者 1 は予測が必要な場合には，その旨を発言していた．

被験者 5 は，実験を開始した直後には最初に `return` を書くことに慣れなかったと報告した．

被験者 4, 5 は，コーディングの過程でその次に何をするのかが予測できなかったため，いつフォルトを発見したかが曖昧になっていると回答した．さらに，被験者 4 は自分のペースで次へを押し，区切りのいいところまで進めてから確かめていた．

図 5.3: break が必要だが，次にそれが現れるかどうか判断がつかない場合

5.4.3.2 実行結果を頻繁に確認したほうがプログラムの間違いに早く気づくと思うか？

被験者 1 は，他人のコードであるため，不等号の逆転などの間違いに早く気づいたと報告した．また，特に何も変更がない場合や自明な場合でも，実行結果が非常に頻繁に確認するように要求されたことが負担であったと述べた．また，実行結果が表示されていたとしても，この程度の規模であればコードの理解力には対した影響はないが，大規模であれば影響があったかもしれないと回答した．

被験者 5 によると，今回の実験は自分でプログラムを書いているわけではなく，他人のコードをじっくり読んで理解するためにその過程で間違いに気づいていた．そのため，今回の実験ではあまりそういう印象がなかった，しかし，実行結果があったためにプログラムの理解が容易になったと報告した．

被験者 1, 5 は，右側の標準出力が表示される箇所が更新される度，一番上までスクロールしてしまうために毎回負担であったと報告した．

被験者 4 は，プログラムが間違っていそうなときには実行結果を確認していたが，その他の場合はあまり確認していなかった．また，実行結果が更新されていなかったことにより，困ったことはあまりなかったと述べた．

5.4.3.3 自分がプログラムを書いていて，今回の実験ほど頻繁に確認をするか？

被験者 5 は，今回の実験ほどは確認をしないと回答した．普段はプログラムの区切りはもうすこし大雑把であり，プログラムが自明な場合，例えば値を return するだけのコードを書いた時点では実行結果の確認は行わないだろうと述べた．

被験者 1 もこれほどの確認はしないと報告した．しかし，今回の実験では，次へのボタンを押すとある程度のまとまった量のコードが出現し，これは普段自分がコードを書いてい

るよりも早い速度であり，普段以上に確認しているように感じたが実際にはこの程度で確認している可能性があるとも述べた．実際，予備実験 2 では 30 分から 40 分かかっていたが，それが 10 分程度に圧縮されている．

5.5 考察

フォルト 2.5, 2.6, 2.7 のように，複数のフォルト同士が複雑に関係しあう場合には，一度にそのフォルトを認識することは難しいことがわかった．これは，それらを全て一度に指摘している被験者がいないことから予想される．

また，本実験 1 での被験者のふるまいは，通常のプログラミングと大きく違った．通常のプログラミングでは，被験者はコードを考えてからそれをエディタ上に書き，実行結果を確認する．このとき，被験者は自分のコードがあっているかどうかを実行結果を見て判断する．しかし，本実験 1 では，コードを読み理解してから実行結果を確認する．このコードを理解をする段階で間違いに気づくことが多い．

一方で，理解をする際に実行結果を参照することが有用だとする被験者もいる．この際に発生する，正当かどうかよくわからないコードを実行結果を見て判断する行動は，通常のプログラミングでのコードの正当性を確かめる行動に類似していると考えられる．

ソースコード 5.1: 課題 1 の最終ソースコード．1.1 から 1.5 までの 5 つの課題がある

```
var op = function(a, b){
  if(a.length === 0){
    var ret = 0;
    for(var i=0; i<b.length; i++){
      ret += b[i];
    }
    return ret;
  }
  var count0 = 0;
  for(var i=0; i<a.length; i++){
    if(a[i] === 0){
      count0++;
    }
  }
  if(count0 >= 3){
    if(a.length !== b.length){
      return true; // ※ 1.1 返すべきものが反転している
    }
    var ret = true;
    for(var i=0; i<a.length; i++){
      ret = ret && a[i] === b[i];
    }
    return ret;
  }
  if(a[a.length-1] === 0){
    var ret = new Array(a.length + b.length);
    var len_min = a.length;
    if(a.length > b.length){
      len_min = b.length;
    }
    println(len_min);
    // a と b から それぞれ, len_min 個いれてから, 長いほうから残りを入れる
    for(var i=0; i<len_min; i++){
      ret[2*i] = a[i];
      // ※ 1.2 代入するインデックスが間違っている
      ret[2*i+1] = b[i+1];
    }
    if(a.length === len_min){
      // ※ 1.3 a からではなく， b から ret に入れる必要がある
      for(var i=0; i<a.length-len_min; i++){
        ret[2*len_min+i] = a[i+len_min];
      }
    }else{
      // ※ 1.4 b からではなく， a から ret に入れる必要がある
      for(var i=0; i<b.length-len_min; i++){
        ret[2*len_min+i] = b[i+len_min];
      }
    }
    return ret;
  }
  var ret = 0;
  for(var i=0; i<a.length; i++){
    for(var j=0; j<b.length; j++){
      if(a[i] === b[i]){ // ※ 1.5 b[i] ではなく， b[j]
        ret++;           // ※ 1.6 break のし忘れ
      }
    }
  }
  return ret;
};
```

ソースコード 5.2: 課題 1 の最終ソースコード．2.1 から 2.10 までの 10 つのフォルトがある

```
var op = function(a, b){
  var mula = 1;
  for(var i=0; i<a.length; i++){
    mula = mula * a[i];
  }
  if(mula >= 10){
    var len_min = a.length;
    if(a.length < b.length){ len_min = b.length; } // ※ 2.1 条件が反転している
    println(len_min);
    var ret = 0;
    for(var i=0; i<len_min; i++){
      ret += a[i] * b[i];
    }
    return ret;
  }
  var a_is_subarray_of_b = false;
  for(var i=0; i<=b.length-a.length; i++){
      var same = true;
      for(var j=0; j<a.length; j++){
          same = same && a[j] === b[i+j];
      }
      a_is_subarray_of_b = same || a_is_subarray_of_b;
  }
  if(a_is_subarray_of_b){
      var ret = true;
      for(var i=0; i<a.length; i++){
          if(a[i] < b[i]){
              ret = true;  // ※ 2.2 break が必要
          } else if(a[i] > b[i]){
              ret = false; // ※ 2.3 break が必要
          }
      }
      return ret;
  }
  if(a.length === b.length){
      var ret = new Array(b.length);
      for(var i=0; i<b.length; i++){
          ret[i] = b[b.length-i]; // ※ 2.4 b[b.length-i-1] が正しい
      }
      return ret;
  }
  var ret = new Array(Math.floor((a.length + b.length) / 2));
  var total_counter = 0; // a と b を結合した配列のインデックス
  for(var i=0; i<a.length; i++){
      if(total_counter % 2 === 0){ // ※ 2.5 total_counter % 2 === 1 が正しい
          // ※ 2.6 左辺は ret[Math.floor(total_counter/2)] が正しい
          // ※ 2.7 右辺は a[i] か, a[Math.floor(total_counter/2)] が正しい
          ret[total_counter/2] = a[total_counter/2];
      }
      total_counter++;
  }
  for(var i=0; i<b.length; i++){
      if(total_counter % 2 === 0){ // ※ 2.8 total_counter % 2 === 1 が正しい
          // ※ 2.9  左辺は ret[Math.floor(total_counter/2)] が正しい
          // ※ 2.10 右辺は b[i] が正しい
          ret[total_counter/2] = b[total_counter/2];
      }
      total_counter++;
  }
  return ret;};
```

第6章　本実験 2

本実験 2 では，本実験 1 と同様に事前に作成されたプログラミングの履歴を再生する．その過程で，フォルトが含まれるコードを表示されてから何秒後に指摘できるかを測定する．また，プログラム全体を確認する時間も仮説 3 のために記録する．

本実験 1 と違い，事前にアルゴリズムを与えた．事前にアルゴリズムを与えることで大まかなプログラムを予測させ，プログラムのみで理解する場合よりも実際のプログラミングに近くなると期待できる．

課題の大きさは，関数本体の行数が 10 行から 25 行程度の予備実験 1 で与えたものと同程度か，少し大きいものを対象とする．

6.1　実験環境

図 6.1 は本実験 2 で使用する実験環境である．この環境は事前にプログラミングの履歴 (プログラムの変化履歴) を記録して再生することができる．

被験者は右下のボタンを押すことにより次のプログラムに進む．このとき，前回からどこが変化したかはハイライトによって表示される (図 6.1)．

実行結果の確認タイミングを事前に設定することができる．次のプログラムに進んだとき，そこが事前に確認すべきタイミングであると設定されていた場合は右の実行結果が赤く枠取りされ，実行結果が更新される (図 6.2)．

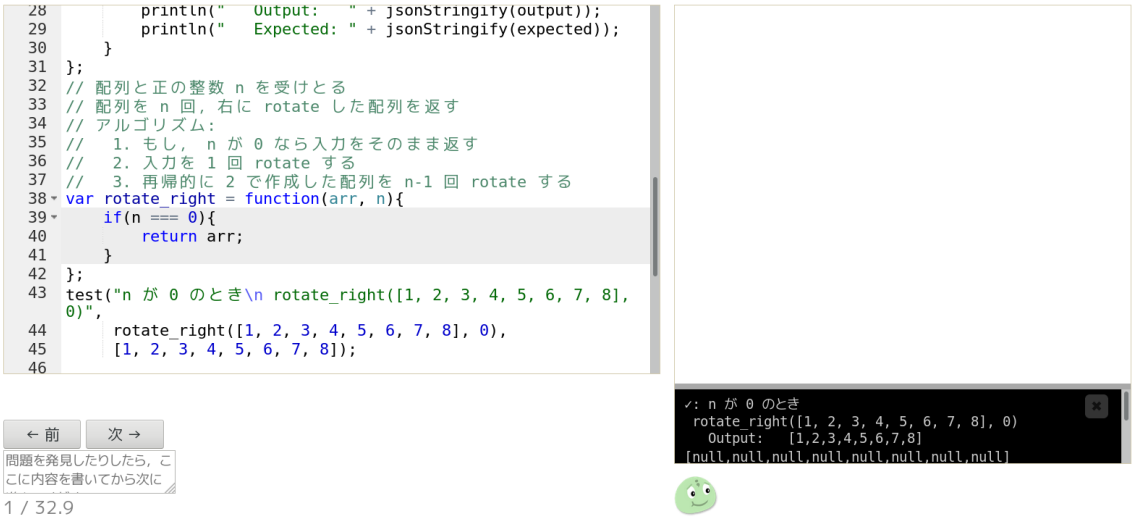

図 6.1: 本実験 2 の実験環境．次のコードを表示したとき，新しく追加されたコードが灰色にハイライトされる

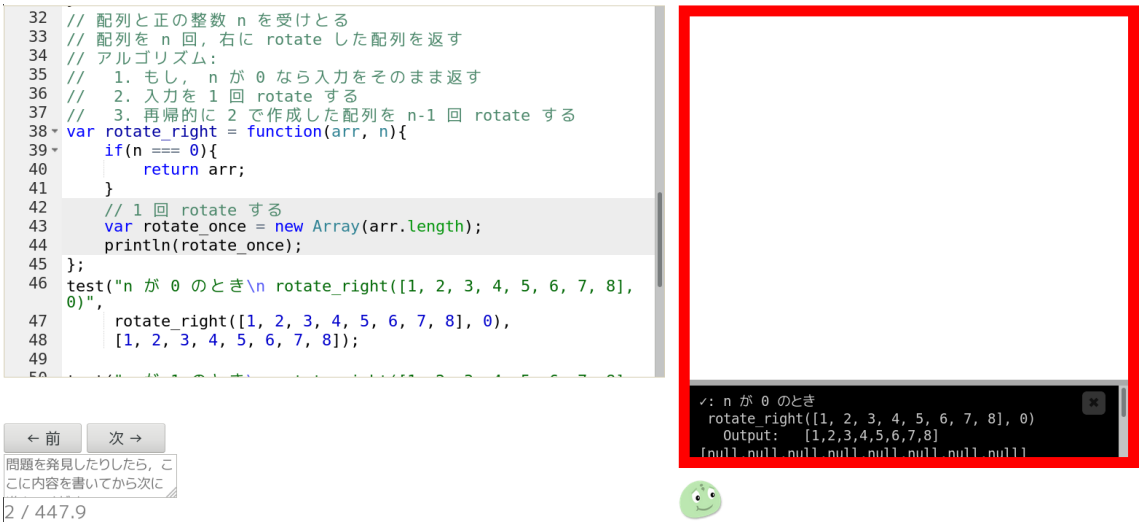

図 6.2: 本実験 2 の実験環境．実行結果が更新されたとき，右の実行結果が赤く枠取りされる

6.2 実験手順

被験者は実験中はスクリーンを録画し，マイクを有効にする．これは計測値を録画データから取り出すためである．

本実験 2 では，課題ごとに以下の手順を行う．このうち，過程 1 はアルゴリズムを理解する時間であるため，計測値からは除外する．

1. 課題プログラムの目的文，アルゴリズム，テストケースを理解する (図 6.3)
2. 課題プログラムのプログラミング過程を再生
   (a) コードを確認する (図 6.4，図 6.5)
   (b) もし，右の実行結果が赤く枠取りされた時，実行結果を確認する (図 6.6，図 6.7)
   (c) フォルトを発見した場合は，マイクに向かって報告する (図 6.8)
   (d) 次のコードに進む
3. これ以上フォルトがないと判断すれば終了 (図 6.9)

6.3 課題設定

対象とする課題は 6 つあり，プログラム中にそれぞれ 1 つのフォルトがある．被験者にはフォルトの個数を開示していないため，被験者はフォルトを発見した後もプログラムを確認する必要がある．

実際に実験する際にはフォルトの種類は以下の 3 つがある．

1. `if` 文の場合漏れ (課題 1, 4)
2. `break` あるいは `continue` 漏れ (課題 2, 3)

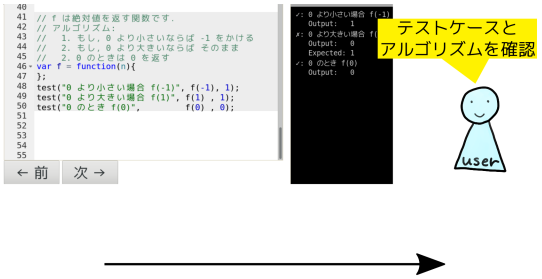

図 6.3: 目的文，アルゴリズム，およびテストケースを理解する

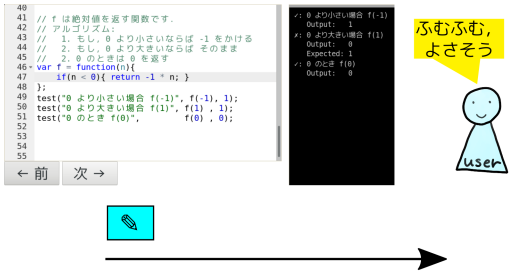

図 6.4: 次に進んでコードを確認する

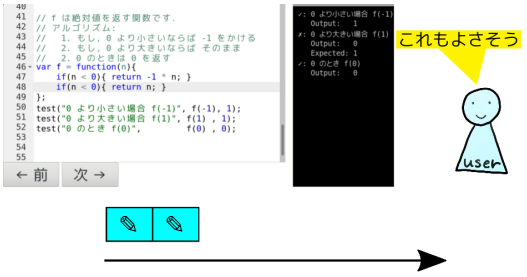

図 6.5: さらに次に進みコードを確認する．ここでフォルトが含まれるコードが発生するが，被験者は気づいていない

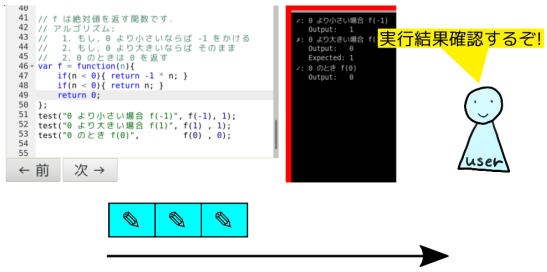

図 6.6: 次のコードに進んだとき，実行結果が更新される

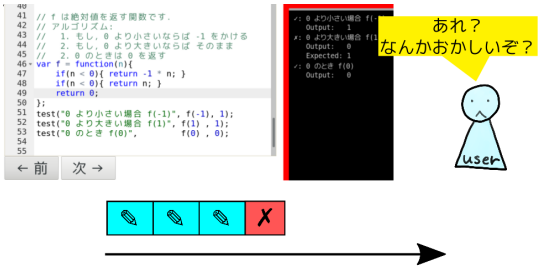

図 6.7: 被験者は実行結果を確認し，フォルトがあることに気づく

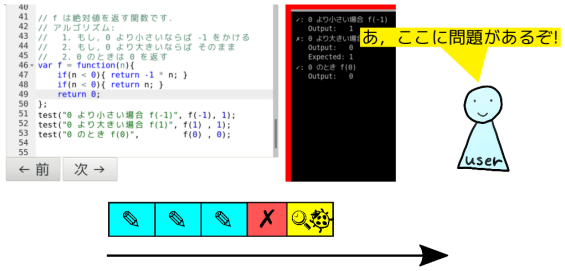

図 6.8: 被験者はフォルトを発見する

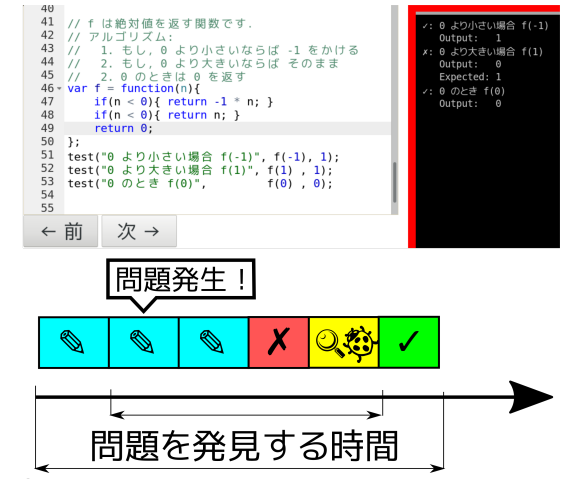

図 6.9: フォルトがもうなければそこで終了

ソースコード 6.1: 課題 1 のソースコード

```
// a と b を連結した配列の n 番目の要素を返す
// n は 0-based (0 から数える) とし，ない場合には -1 を返す
var op = function(a, b, n){
    if(n < a.length){
        return a[n];
    }
    if(n < a.length + b.length){
        return b[n-a.length];
    }
    return -1;
};

test("n が a の長さより大きく， a の長さ + b の長さより小さい",
     op([1, 2], [3, 4, 5, 6], 3),
     4);
test("n が a の長さより小さいとき",
     op([1, 2], [3, 4, 5, 6], 1),
     2);
test("n が a の長さ + b の長さと同じとき",
     op([1, 2], [3, 4, 5, 6], 6),
     -1);
test("n が 負のとき",
     op([1, 2, 3], [4, 5], -123),
     -1);
```

3. for 文の境界条件間違い (課題 5, 6)

課題ごとに実行結果の確認タイミングは以下の二通りである．これらはフォルトの種類ごとに片方が頻繁に確認する場合，もう一方が最後のみ確認する場合になるよう設定する．

- 頻繁に確認する場合 (for 文や if 文を記述する度に確認する場合)
- 関数を記述し終わった最後のみ確認する場合

6.3.1 課題 1: 連結した配列への要素アクセス

課題設定 入力として，整数の配列を二つと整数 n を一つ受け取る．それらを連結した配列の n 番目にアクセスする．ない場合は -1 を返す．(ソースコード 6.1)

期待する行動 一つ目の if 文を書いた時点で n が負の場合のテストケースが失敗する．これにより， if 文の条件式が間違っていることに気づくことができる．

6.3.2 課題 2: 配列のある要素の探索，配列の書き換え

課題設定 入力として，整数の配列と整数 x を受け取る．配列の要素のうち， x と一致する最初の要素から 3 つを 0 で置き換える． (ソースコード 6.2)

期待する行動 remove_from を計算する for 文を書いた後に，実行結果を確認するとその時点で remove_from が間違っているテストケースが存在する．そのため，直前の for 文が間違っていると推測できる．

ソースコード 6.2: 課題 2 のソースコード

```
// 整数の配列 arr と 整数 x を受けとる
// arr 内の x と一致する最初の要素から 3 つの要素を 0 で置き換えた配列を返す
// * x がない場合は何も置き換えない.
// * 3 つ削除できない場合は, 1 つ か 2つ できるだけ置き換える
// アルゴリズム:
//   1. どこから置き換えすべきかを計算する
//   2. 1 で計算したところから 3 つ分置き換える
//      ただし, 途中で配列の最後まで辿りついたらそこで終了
var remove_from_value = function(arr, x){
    var ret = arr.slice(0); // コピー
    var remove_from = -1;   // 3 つ削除を開始する位置
    for(var i=0; i<arr.length; i++){
        if(arr[i] === x){
            remove_from = i;
        }
    }
    println("remove_from = " + remove_from);
    if(remove_from === -1){
        return ret;
    }
    for(var i=remove_from; i<arr.length && i < remove_from + 3; i++){
        ret[i] = 0;
    }
    return ret;
};

test("x がない場合",
     remove_from_value([1, 2, 3], 4),
     [1, 2, 3]);
test("x があって, 3 つ置き換えられるとき",
     remove_from_value([1, 2, 3, 4, 5], 2),
     [1, 0, 0, 0, 5]);
test("x があって, 3 つ置き換えられるとき",
     remove_from_value([1, 2, 3, 4], 3),
     [1, 2, 0, 0]);
test("x が 2 つ以上あるとき",
     remove_from_value([1, 2, 2, 4], 2),
     [1, 0, 0, 0]);
test("x が 2 つ以上あるとき",
     remove_from_value([2, 2, 2, 2, 2], 2),
     [0, 0, 0, 2, 2]);
```

6.3.3 課題 3: ソートした配列のマージ

課題設定 入力として，整数の配列と整数 x を受け取る．配列の要素のうち， x と一致する最初の要素から 3 つを 0 で置き換える． (ソースコード 6.3)

期待する行動 while 文内で，最初の if 文を記述した時点では 2 つ目までテストケースが成功する．ここで，2 つめの if 文を記述すると 3 つ目までテストケースが表示されると期待する．しかし，実際には 2 つ目のテストケースが無限ループに陥り，表示されなくなる．そのため， 2 つめの if 文にフォルトがあると推測できる．

6.3.4 課題 4: 配列の要素の組み合わせ

課題設定 入力として整数の配列 arr と整数 n を受け取る． arr から n 個の要素を選ぶ方法を列挙する． (ソースコード 6.4)

期待する行動 with_arr_0 を計算するために再帰呼び出し combinations(n-1, arr.slice(1)) を追加した時点では，テストケースは全て実行される．しかし，without_arr_0 を計算するための再帰呼び出し combinations(n, arr.slice(1)) を記述すると，最初のテストケースのみが表示された状態になる．これにより，combinations(n, arr.slice(1)) が原因で無限ループに陥いっていることが推測できる．この式をよく確認すると，n が減っていないため，再帰の終了条件が満されないことがわかる．

6.3.5 課題 5: 配列の rotate

課題設定 入力として整数の配列 arr と整数 n を受け取る． arr から n 回右に rotate した配列を返す． (ソースコード 6.5)

期待する行動 rotate_once に代入する for 文を記述した時点で，表示された rotate_once の長さが期待されるより長くなっていることがわかる．そのため， for 文にフォルトがあることがわかる．

6.3.6 課題 6: 配列への挿入

課題設定 入力として整数の配列 arr ，整数 n と整数 x を受け取る． arr の n 番目に x を挿入した配列を返す．(ソースコード 6.6)

期待する行動 配列の前半部分に関する for 文を書いた時点では，表示されている ret は，期待通り前半部分が作成されていることが確認できる．しかし，後半部分に関する for 文を書くと ret が正しく表示されなくなる．このため， 2 つめの for 文がおかしいことが推測できる．

ソースコード 6.3: 課題 3 のソースコード

```
// ソート済みの整数のリスト a, b を受けとる.
// (全ての要素について, 前の要素と同じか大きいかが成り立つ)
// その二つのリストの要素全てを含み, かつソート済みのリストを返す
// アルゴリズム:
//   0. 返す配列を作る. (長さは a.length + b.length)
//   1. 今注目する a のインデックスを a_i,
//                  b のインデックスを b_i とする
//   2.1. もし, a_i も b_i も最後のインデックスなら終了
//   2.2  もし, a_i が最後のインデックスなら, b から一つ入れる
//   2.3  もし, b_i が最後のインデックスなら, a から一つ入れる
//   2.4  a[a_i] と b[b_i] を比べて, 小さいほうから一つ入れる
//        (同じなら a から入れる)
var merge_sorted = function(a, b){
    var ret = new Array(a.length + b.length);
    var a_i = 0; // a の注目するインデックス
    var b_i = 0; // b の注目するインデックス
    var r_i = 0; // 次に代入する ret のインデックス
    while(a_i !== a.length || b_i !== b.length){
        if(a_i === a.length){
            ret[r_i] = b[b_i];
            b_i++;
            r_i++;
        }
        if(b_i === b.length){
            ret[r_i] = a[a_i];
            a_i++;
            r_i++;
        }


        if(a[a_i] <= b[b_i]){
            ret[r_i] = a[a_i];
            a_i++;
            r_i++;
        } else {
            ret[r_i] = b[b_i];
            b_i++;
            r_i++;
        }
    }
    return ret;
};
test("a も b も空の場合",
     merge_sorted([], []),
     []);
test("a が空の場合",
     merge_sorted([], [1, 2, 3]),
     [1, 2, 3]);
test("b が空の場合",
     merge_sorted([1, 2, 3], []),
     [1, 2, 3]);
test("b の要素が全て a の要素より大きい場合",
     merge_sorted([1, 2, 3], [4, 5, 6]),
     [1, 2, 3, 4, 5, 6]);
// テストケース一部省略
```

ソースコード 6.4: 課題 4 のソースコード

```
// 整数 n と 整数の配列 arr を受け取る
// arr の部分集合のうち，要素が n 個のものを全て含む配列を作る
// (= arr から n 個を選ぶ方法を列挙する)
// ただし，最初に受けとる n は arr.length 以下として良い
// アルゴリズム:
//   1. n と arr から，作れる部分集合が一通りならばそれを返す
//   2. 再帰的に， arr[0] を使う場合と，使わない場合を計算する
//   3. arr[0] を使う場合と使わない場合の返り値同士を結合して，それを返す
var combinations = function(n, arr){
    // 作れる部分集合が一通りのとき
    if(n === 0){
        return [[]];
    }
    // arr[0] を使う場合と使わない場合を考える
    // ※ arr.slice(1) は arr の最初の要素を取り除いたリスト
    var with_arr_0    = combinations(n-1, arr.slice(1));
    for(var i=0; i<with_arr_0.length; i++){
        // ※ concat はリスト同士の結合
        with_arr_0[i] = [arr[0]].concat(with_arr_0[i]);
    }
    var without_arr_0 = combinations(n, arr.slice(1));

    return with_arr_0.concat(without_arr_0);
};
test("n = 0 のとき",
     combinations(0, [1, 2, 3]),
     [[]]);
test("n = 1 のとき",
     combinations(1, [1, 2, 3]),
     [[1], [2], [3]]);
test("n = 2 のとき",
     combinations(2, [1, 2, 3]),
     [[1, 2], [1, 3], [2, 3]]);
test("n = 3 のとき",
     combinations(3, [1, 2, 3]),
     [[1, 2, 3]]);
```

ソースコード 6.5: 課題 5 のソースコード

```
// 配列と正の整数 n を受けとる
// 配列を n 回，右に rotate した配列を返す
// アルゴリズム:
//   1. もし， n が 0 なら入力をそのまま返す
//   2. 入力を 1 回 rotate する
//   3. 再帰的に 2 で作成した配列を n-1 回 rotate する
var rotate_right = function(arr, n){
    if(n === 0){
        return arr;
    }
    // 1 回 rotate する
    var rotate_once = new Array(arr.length);
    for(var i=0; i<arr.length; i++){
        rotate_once[i+1] = arr[i];
    }
    rotate_once[0] = arr[arr.length-1];
    println(rotate_once);

    return rotate_right(rotate_once, n-1);
};

test("n が 0 のとき",
     rotate_right([1, 2, 3, 4, 5, 6, 7, 8], 0),
     [1, 2, 3, 4, 5, 6, 7, 8]);

test("n が 1 のとき",
     rotate_right([1, 2, 3, 4, 5, 6, 7, 8], 1),
     [8, 1, 2, 3, 4, 5, 6, 7]);

test("n が 2 のとき",
     rotate_right([1, 2, 3, 4, 5, 6, 7, 8], 2),
     [7, 8, 1, 2, 3, 4, 5, 6]);

test("n が 10 のとき",
     rotate_right([1, 2, 3, 4, 5, 6, 7, 8], 10),
     [7, 8, 1, 2, 3, 4, 5, 6]);
```

ソースコード 6.6: 課題 6 のソースコード

```
// 入力として，整数の配列 arr，正の整数 n，整数 x を受け取る
// arr の n 番目に x を挿入したリストを返す．
// n は 1-based (1 から数える) とし，配列の長さ+1 以下とする．
// アルゴリズム:
//   1. 返り値として，arr より 1 長い配列を作る (ret と呼ぶ)
//   2. ret の n 番目に x を入れる
//   3. ret の n 番目までに arr の n-1 番目までを入れる
//   4. arr の残りの部分を ret の n 番目以降に入れる
var insert_at = function(arr, n, x){
    var ret = new Array(arr.length + 1);
    ret[n-1] = x;
    for(var i=0; i<n-1; i++){
        ret[i] = arr[i];
    }
    for(var i=n; i<arr.length; i++){
        ret[i] = arr[i];
    }
    return ret;
};

test("2 番目に挿入する場合",
     insert_at([1, 2, 3, 4], 2, 9),
     [1, 9, 2, 3, 4]);

test("最初に挿入する場合",
     insert_at([1, 2, 3, 4], 1, 9),
     [9, 1, 2, 3, 4]);

test("最後に挿入する場合",
     insert_at([1, 2, 3, 4], 5, 9),
     [1, 2, 3, 4, 9]);
```

6.4 対象被験者

対象とする被験者は，予備実験および本実験 1 から表 6.1 に記載した被験者を採用した．また，表 6.2 に課題の順番および実行結果の確認タイミングが多い場合なのか少ない場合なのかを記載する．

被験者番号	学年	JavaScript 経験	プログラミング歴
1	M1	なし	4 年
3	D1	あり	6,7 年
4	M1	なし	5 年
5	B4	あり	2 年 9ヶ月
6	B4	なし	2 年 9ヶ月

表 6.1: 本実験 2 における被験者構成

被験者番号	1 問目	2 問目	3 問目	4 問目	5 問目	6 問目
1	4 (N)	1 (L)	2 (L)	6 (L)	5 (N)	3 (N)
3	1 (L)	4 (N)	2 (L)	3 (N)	5 (L)	6 (N)
4	6 (L)	2 (N)	5 (N)	4 (L)	3 (L)	1 (N)
5	2 (L)	6 (N)	5 (L)	3 (N)	1 (L)	4 (N)
6	1 (N)	4 (L)	3 (L)	2 (N)	6 (N)	5 (L)

表 6.2: 本実験 2 における課題の割り当て．かっこ内は使用する実験環境を表わし，頻繁に実行結果が更新される環境を使う場合は L，そうでない場合は N と表記する．

6.5 結果

表 6.3 に，フォルトがあるコードが現れてから何秒でそれに気づくか，最終的に何秒かかったのかをまとめた．図 6.10 に課題ごとにデータをまとめたグラフを掲載する．また，被験者ごとのデータを図 6.11 に載せる．

フォルト発見にかかった時間は，フォルトが含まれるコードが表示されてからそれを発見するまでの時間とする．最終的にかかった時間は，全ての時間からフォルトがどのようなものであるか，その修正方法を説明していた時間を引いたものである．この説明は本来のプログラミングでは必要ない行為なためである．

6.5.1 数値解析結果

本実験 2 で得た結果に対する数値解析について述べる．

図 6.12 は課題ごとにフォルトを発見するのにかかった時間をまとめたものである．頻繁に実行結果を確認した場合とのほうがフォルト発見にかかる時間がすくない場合もそうでない場合も存在し，また標本標準偏差も大きいため，この結果から結論を出すことはできない．

図 6.13 は課題ごとのプログラム全体を確認するのに使用した時間をまとめたものである．このグラフでは，頻繁に確認した場合とそうでない場合での差がフォルト発見にかかった時

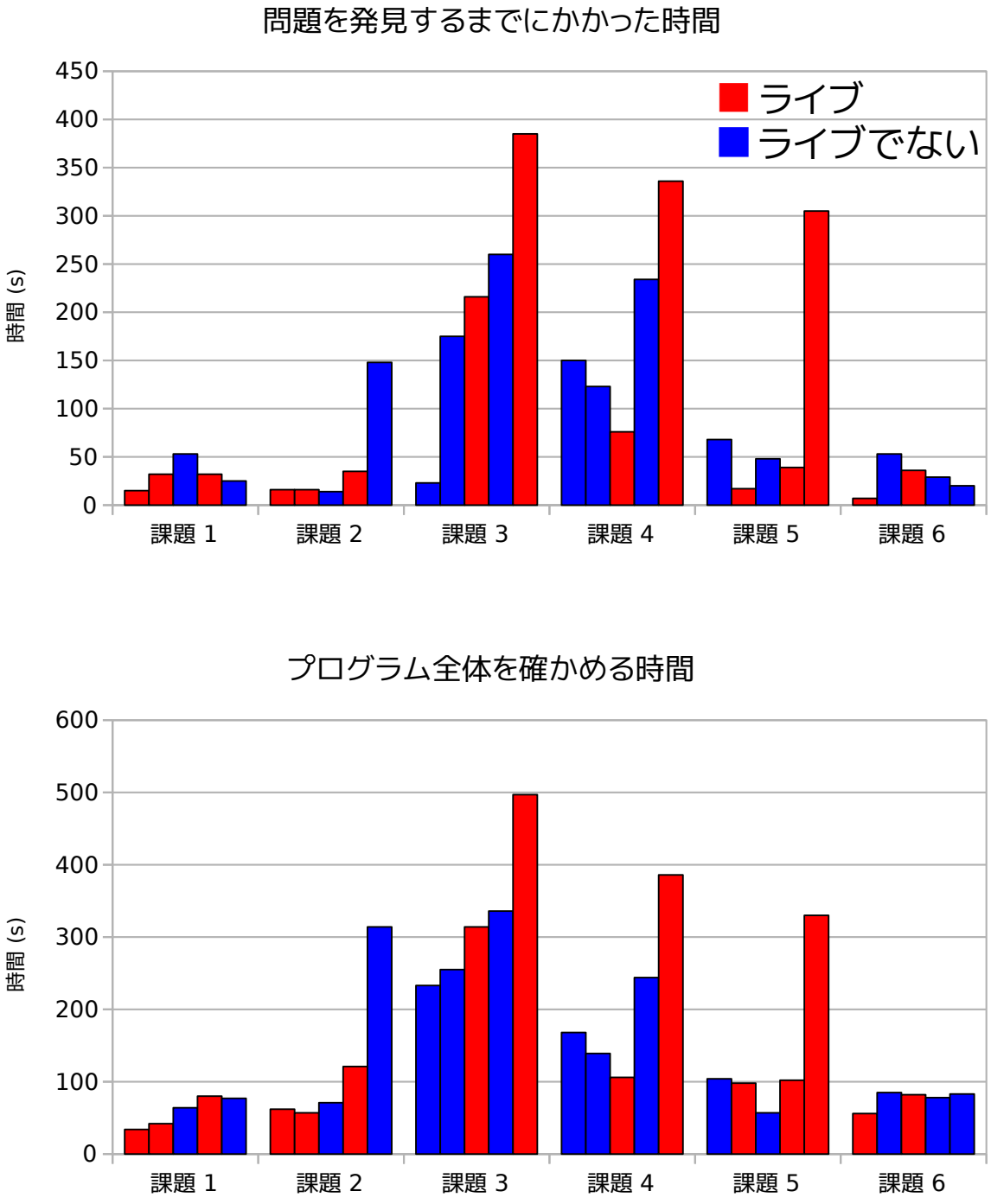

図 6.10: 本実験 2 における結果のグラフ．課題ごとに棒グラフは左から被験者 1, 3, 4, 5, 6 の結果

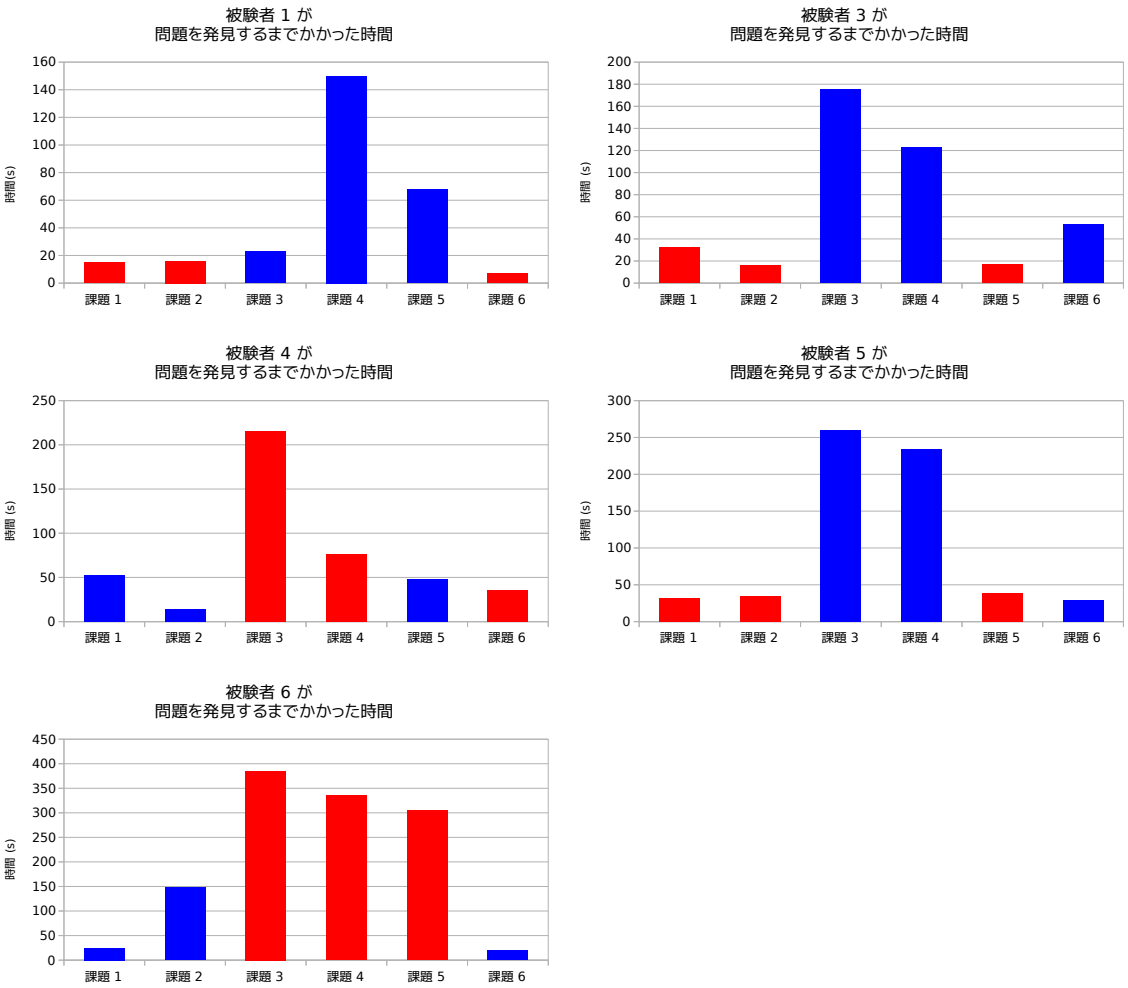

図 6.11: 本実験 2 におけるデータを被験者ごとにまとめたグラフ.

	課題 1			課題 2			課題 3		
被験者	環境	発見	最終	環境	発見	最終	環境	発見	最終
1	L	15	34	L	16	62	N	23	233
3	L	32	42	L	16	57	N	175	255
4	N	53	64	N	14	71	L	216	314
5	L	32	80	L	35	121	N	260	336
6	N	25	77	N	148	314	L	385	497

	課題 4			課題 5			課題 6		
被験者	環境	発見	最終	環境	発見	最終	環境	発見	最終
1	N	150	168	N	68	104	L	7	56
3	N	123	139	L	17	98	N	53	85
4	L	76	106	N	48	57	L	36	82
5	N	234	244	L	39	102	N	29	78
6	L	336	386	L	305	330	N	20	83

表 6.3: 本実験における集計データ．課題ごとに実験課題，フォルトを発見するまでの時間，最終的にかかった時間をまとめた．実験環境は確認タイミングが多い場合は L，そうでない場合は N と記載する．時間の単位は秒である．

間での差より小さい．これはフォルトそのものはすぐに発見した場合でも，それ以外に時間がかかっているからである．また，ほとんどのグラフは図 6.12 のグラフと同じ傾向を示している．これはそもそも課題ごとに早くフォルトを発見できた被験者は全体を確かめるのも早いことや，課題 4 のように最初からフォルトが存在していたもののそれが最後まで指摘されなかったものが含まれるためである．

6.5.2 インタビュー結果

予備実験と同様に，実験後にインタビューを行なった．

6.5.2.1 質問 1: 頻繁に確認したほうがフォルトに早く気づくと感じたか？

被験者 1, 3, 5, 6 は，質問に対し肯定的な返答をした．被験者 6 は，今回の実験ではそのような例が多かったと答えた．また，被験者 1, 5 は確認しようと思ったタイミングで表示されていないことがあったと述べた．

被験者 4 は，ほぼ実行結果を最後のみしか確認せず，必要なときには表示されていたため，特にそうでもなかったと返答した．

6.5.2.2 質問 2: 普段，どの程度の実行結果を確認するか？ (実験の環境と比較して)

被験者 1, 4 は，関数やプログラムを完成させてから実行すると述べた．その理由として，普段の対象が GUI であるため，実行や確認のコストが大きいことを挙げた．

被験者 5 はライブプログラミング環境を使っている場合は実験環境の多い場合とと同程度，そうでないならばもうすこし少なくなると述べた．

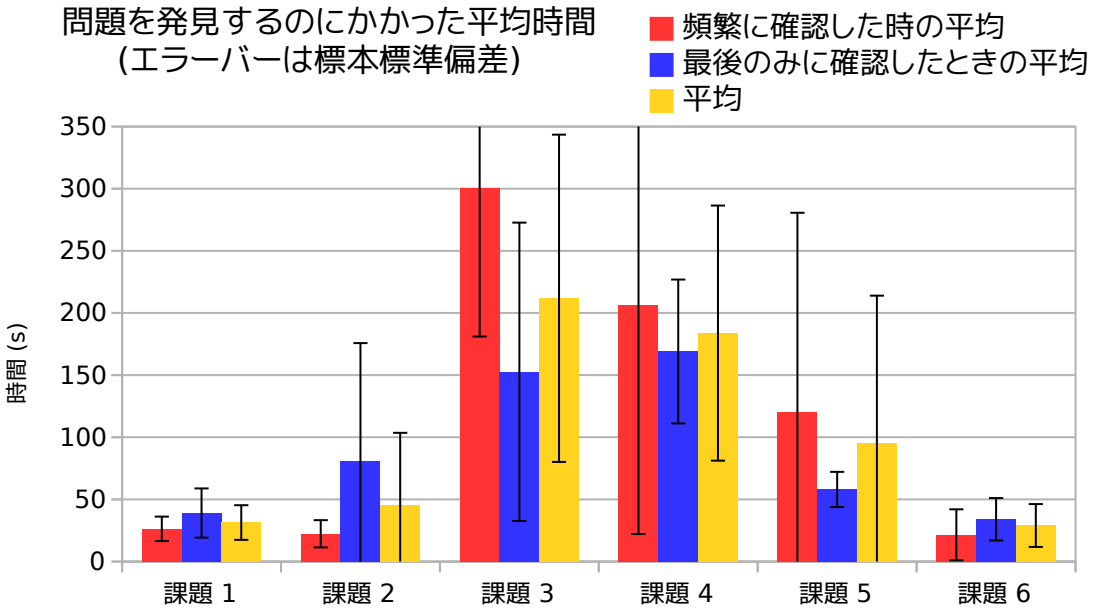

図 6.12: フォルトを発見するのにかかった平均時間とその標本標準偏差

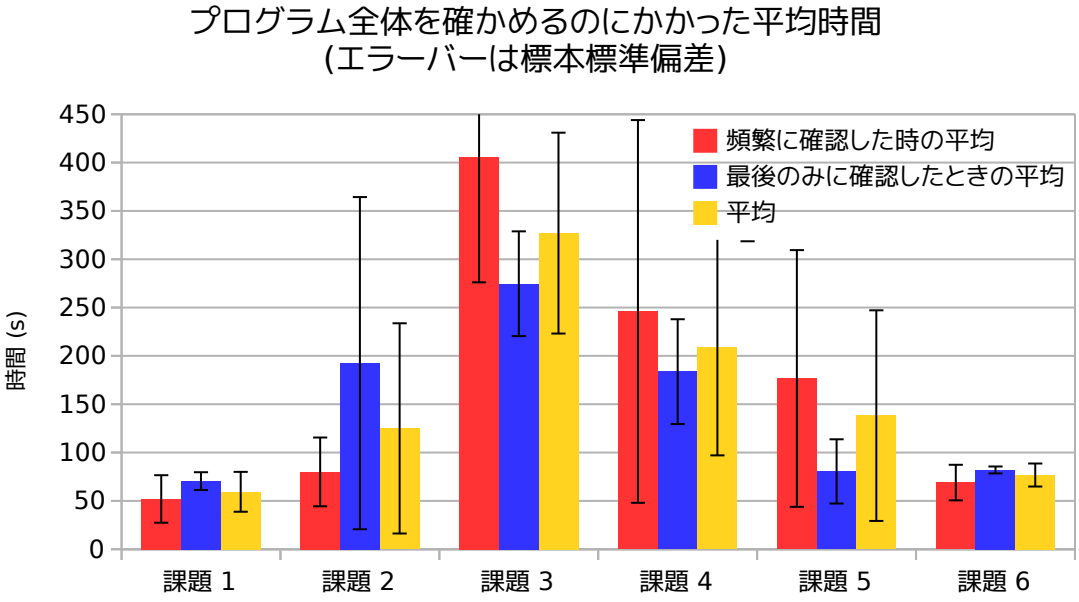

図 6.13: プログラム全体を確かめるのにかかった平均時間とその標本標準偏差

被験者 6 は，条件分岐を書いて実行結果を確かめる場合もあると述べた．ただし，実験環境の多い場合よりは少なく，例えば for 文を書いて実行などはしないとコメントした．

6.5.2.3　その他，気づいたこと

課題の難易度については，少し難しいという意見が多く，プログラムを修正することなくその間違いを指摘するのは困難であることが理由として挙げられた．

ほぼ全ての被験者は，プログラムの書き方について違和感はないと述べた．被験者 4 は，これまでの実験を通して慣れたとコメントした．

6.6　考察

予備実験と比較して今回の実験では，ライブプログラミングに関して肯定的な意見を得ることが多かった．予備実験と本実験 2 の大きな違いは，実行結果の確認タイミングが与えられるか，そうでないかである．本実験 2 で与えた確認のタイミングは，筆者が実際に課題を解く際に確認したものを元にしているため，ライブプログラミング環境に慣れていない被験者と比較するとより適切なものだったと考えられる．この事実は被験者がライブプログラミング環境での経験をより多く積めば，その恩恵を受けられることを意味する．

テストケースの成否や，どこまで表示されたかが変わっても被験者が気づかない，あるいは読み飛ばしてしまう場合があった．これは，そもそもテストケースがどのような意図を持って書かれたものかを意識していないためだと考えられる．これはテストケースは事前に与えたものであり，テストケースを書くことによって得られるような推論 [4] が働かないためだと考えられる．

フォルト発見の手掛かりとなるテストケースの変化があっても読み飛ばしてしまうことがある．テストケースが最初から与えられているため，その時点でのプログラムでどこのテストケースまで実行されるべきかを推論できなかったためと考えられる．

計測値からは，頻繁に実行結果を確認した際にはフォルトに早く気づくことができる，という仮説に対して結論を述べることができない．これはフォルトを発見する際に，どうやって発見したかを区別していないことが大きな原因である．実験中観測された発見方法は 1. コードのみを見てすぐに気づく場合と 2. 実行結果を見てフォルトに気づく場合， 3. 実行結果を見たが，フォルトをすぐに発見できない場合の 3 つがある (表 6.4)．これらの発見方法の間でフォルトを発見する時間に大きな差がある．さらにコードを見て気づくという行動は本来のプログラミングにはない可能性が高い．なぜならプログラマが自分でコードを書いて際には，そのコードは正しいと信じているからである．

また，この実験では実行結果が更新された場合でも，それを確認しない被験者もいた．そのような場合のデータは，実行結果の確認頻度が変わらないため，参考にすることができない．よって，確認したかどうかのチェックボックス等の実験環境中に実行結果の確認をある程度強制する仕組みが必要であった．

フォルトの発見方法	かかる時間
コードのみを見てすぐ気づく	小
実行結果を見てフォルトに気づく	中
実行結果を見たがフォルトの箇所を特定できない	大

表 6.4: 本実験 2 におけるフォルトの発見方法

第7章　制限と今後

実際にプログラムを書く際には，関数を書く前にどのようなテストケースが必要かを考える必要がある．本研究の実験では，そのテストケースは既に与えられていた．このテストケースを考えるという過程でもライブプログラミングの利点がある可能性があるが，本実験 1, 2 でそれを測ることはできていない．テストケースを書く際には，まだ関数が存在しておらず，フィードバックとなる実行結果を得ることができない．よって，そのような測定を行う場合には，本実験で使用した単純なライブプログラミング環境ではなく，テストを書くのをサポートした環境が必要になる．

さらに，自身でテストケースを考えてからプログラムを書く場合には，プログラムを書いている最中にどのテストケースが成功するべきなのかを推論することが容易になる．そのような推論はライブプログラミング環境でプログラムを書く上で有用であると考えられ，追試験をする価値がある．

本実験ではライブプログラミングによって，実行結果を観測する頻度が増えると予想していた．予備実験 2 でそのように報告する被験者もいたが，これを数値として計測はできていない．これは視線のトラッキングをする等によって計測できると考えられる．

本実験 1 では，プログラムを被験者が書いていないため，普段のプログラミングとは違う条件だった．また，間違いを発見した場合でも，それを直すことができないため，残りのプログラムから間違いを探すときにライブプログラミングの支援をうまく受けられないことがわかった．より現実的な実験を行うには，被験者がプログラムを書いている最中に実行結果を観測するタイミングを変更することが必要だと考えられる．しかし，被験者ごとにプログラムの書き方やアルゴリズムが変わるため，比較するのが難しい．

本実験 1, 2 では，実行結果を確認することなしにフォルトを発見した場合と，実行結果を確認してフォルトを発見した場合の区別がついていない．筆者が実験の後にインタビューしたり，録画されたビデオを確認することによって，いくつかの場合は区別できるが，より正確に測定するには被験者がそれを明示的に宣言するなどの必要がある．

本実験 1 では，フォルトがあった際にユーザがどのように振る舞うかを観察した．しかし，実際のプログラミングでは，フォルトがないコード中にフォルトがないことを確かめることも必要である．その過程でもライブプログラミング環境は有用である可能性がある．これは本実験 2 で計測したプログラム一つを終了するまでの時間からある程度読みとれるが，より正確に計測する方法を考える必要がある．

実行結果を確認するタイミングについてより考察や測定が必要である．実行結果を確認するべきタイミングや確認したいと思うタイミングというものをより正確に説明することができれば，開発環境が自発的に確認を促すなどが可能になり，よりライブプログラミング環境を設計するのに役立つと考えられる．

本研究では，プログラム全体を完成させる時間と使って生産性を定義した．しかし，生産性をプログラマが感じるストレス等の数値として直接現れないものを使って定義することもできる．このような指標を使った場合でもライブプログラミング環境が生産性を向上すると

言えるのかは考察の余地がある．また，数値に直接現れない場合でも，脳波を使ったり [17]，心拍数を計測するなどで数値として計測できる場合もある．

よりデータ数を増やし，統計的に有意かどうかを確かめることは今後の課題とする．

第8章　アイデア

本章では，プログラマが実行結果を確認をより頻繁に確認するようになる開発環境等を述べる．

- プログラマがより頻繁に実行結果を確認することに慣れる
- より実行結果の確認を容易にする
- プログラマが頻繁に確認するように強制する

8.1　プログラマが慣れる

プログラマはどの程度コードを記述したら確認するかという閾値を持っている．予備実験より，この閾値はプログラムの種類等に依存し，ライブプログラミング環境を使った場合でもあまり変わらなかった．この閾値は慣れの影響が大きいと考えられる．

例えば，図 8.1 に示すようなアプリケーションを使ってプログラマに習慣を付けることが考えられる．このアプリケーションは決められた秒数ごとに実行結果を確認することを勧める音声を出力する．筆者が実際にこのアプリケーションを動かしながらプログラムを実装し，ある程度実行結果を確認する習慣が付くように感じられた．一方で，長時間に渡ってこのアプリケーションの音声に従いながらプログラムを実装するのはストレスが大きかった．このストレスの原因は，アルゴリズムを考えている時でも実行結果を確認するように要請されたことである．

8.2　実行結果の確認を容易にする

本研究では，ライブプログラミング環境が実行結果の確認を容易にするため，プログラマはより頻繁に実行結果を確認するだろうと予想した．本研究で使用した live-editor [2] では，

図 8.1: 決められた秒数ごとに音声を流すアプリケーション

情報	プログラマが使用可能か	コンピュータが使用可能か
全体のコード	✓	✓
コードの変化	✓	✓
テストケースの成否変化	×	✓
前回の確認からの実行結果変化	×	✓

表 8.1: 実行結果の確認タイミングを判断するのに使用できる情報

プログラマは実行結果を表示するためには `print` 文などを使用する必要があった．一方で，Apple Swift [11] や Bret Victor の提案 [1] などのように自動的に途中の実行結果を表示するような環境も提案されている．これらの機能が本当に実行結果の確認を増加させるのか，あるいはそれ以外の効果があるのかを調べる必要がある．

8.3 実行結果の確認を強制する

8.1 節および 8.2 節で述べたアイデアでは，プログラマが実行結果をより多く確認するようにするものだった．本節では，実行結果の確認タイミングをプログラマではなく，コンピュータに判断させることについて考える．もしコンピュータのほうが適切なタイミングを判断できるのであれば，プログラマ自身が判断するよりも生産性が向上する可能性がある．一方でプログラマがストレスを感じることもありえる．

このアイデアは本実験 2 で使用した開発環境を拡張したものである．本実験 2 では，被験者はあらかじめ設定された実行結果の確認タイミングに従った．この実行結果の確認タイミングは筆者が適切であろうと考えたものであったが，これをコンピュータが計算するように拡張する．

ライブプログラミング環境が実行結果の確認タイミングを計算する際，使用できる情報を表 8.1 にまとめた．実行しないとわからない情報はプログラマが確認タイミングを判断する場合は使用できないものであり，コンピュータのほうがより良いタイミングを判断できると考えられる．このうち，前回確認した時からの実行結果の変化は Patrick Rein らによって似た研究が行われている．しかし，彼らの方法ではプログラマ自身が実行結果の確認をしているため，前回いつ確認したかはプログラマが自身で指定する．

このアイデアを実装する際には，以下のことが難しいと考えられる．

- どの程度強制すれば良いのか．強制が強すぎた場合，プログラマがストレスを感じると考えられる．
- 書きかけのプログラムをどう扱えば良いのか．プログラマがどのテストケースを意識して書いているのかを推論するにはどうしたら良いのか．
- どのように実行結果を計算すれば良いのか．機械学習等を用いる場合，どのような教師データを使用するのか．

第9章　関連研究

Patrick Rein らは，ライブプログラミング環境がどの程度の早さで実行結果を更新し，ユーザに提示することができるかについてベンチマークを行っている [10]．しかし，その速度がユーザにどのような影響を与えるのかについては述べていなかった．本研究の延長として，どの程度の早さならばユーザはライブプログラミングの恩恵を受けることができるのかを調べることが考えられる．

Scratch はブロックベースのプログラミング言語を使用したライブプログラミング環境である [13]．ブロックベースのプログラミング言語では，必ず文法は正しい状態を保つことができる．本研究で対象としたプログラミング言語はテキストベースであったため，被験者によっては文法エラーがある状態でプログラムを書きすすめてしまい，フィードバックを得られない場合があった．本研究の言語をブロックベースなものに置き換えることで，どの程度プログラマの負担が軽減できるかは考察する価値があると考えられる．

筆者らが過去に提案した Shiranui は，ライブプログラミング環境に適したユニットテスト機構を提案した [6]．このユニットテスト機構ではテストケースを独立かつ並列に実行することにより，あるテストケースの実行に時間がかかる場合でも他のテストケースからフィードバックを得ることができる．本実験 2 では，実行時間が無限にかかるケースがあり，そこからフォルトを推論することができるように課題が設計されていた．このとき，Shiranui と同様に並列に実行することでより情報量を増やすことができるだろうと考えられる．

Apple による XCode Playground [11] は，`println` のように明示的な出力をしなくても，全ての式に対しその結果を表示するようなライブプログラミング環境である．プログラマはフィードバックを得るための式を書く必要がないため，手軽に実行結果を確認することができるだろうと予測される．このような環境の場合，明示的な出力を書く必要がないことが重要なのか，あるいは実行ボタンを押す必要がないことが重要なのかを検証する必要がある．

本研究で行なった実証実験は，デバッガーの実験とエラーを見つけるまでの時間をはかるという共通点があるが，コードの編集が含まれるかどうかという違いがある．デバッガーの実験では Kouhei Sakurai らの研究 [14] のように，最初にエラーを含むコードが与えられ，そこからエラーを発見する時間を計測する．それに対し，本研究ではプログラムは少しづつ与えられ，エラーが含むコードが表示されてからエラーを発見する時間を計測する．

Hidetake Uwano らはフォルトを探す際の視線の移動のパターン及びフォルト発見に要した時間を計測した [16]．この実験はコードの編集や，実行結果の確認等が含まれないが，ライブプログラミング環境における実験に拡張ができると考えられる．

第10章　結論

ライブプログラミング環境が生産性を向上させることの理由づけとして，以下の 3 つの仮定を立て，本実験では仮定 1，2 を考察した．

1. ライブプログラミング環境を使っているとき，ユーザは実行結果をより頻繁に確認する
2. 頻繁に実行結果を確認することで，ユーザはフォルトをすぐに発見できる
3. フォルトをすぐに発見できるならば，全体としての生産性が向上する

実験は予備実験と本実験で構成される．予備実験では，ライブプログラミング環境と，実行ボタンがある通常のプログラミング環境を用意し，被験者にいくつかの課題を実装させインタビューを行なった．本実験では，事前に作成されたプログラミングの過程を再生し，かつ実行結果を確認するタイミングを制御できる環境を用意し，被験者にフォルトが含まれるコードの作成過程を再生させ，フォルトを発見するタイミングを測定した．

以下に予備実験から考えられることを述べる．

アルゴリズムを対象とし，かつ事前に用意された小さい関数を実装する場合，ライブプログラミング環境であってもそうでなくても実行結果の確認頻度に差は出ないことがわかった．これは，被験者がどの程度のプログラムを書いたら実行結果を確認するべきかを経験的に学んでおり，小さい関数の場合には途中で実行結果を確認すべきだと考えないからである．

アルゴリズムを対象とした比較的大きなプログラムでは，実行結果の確認頻度が増加する被験者もそうでない被験者もいた．これはどのようなプログラミングスタイルでコードを書くのか，元々経験的に学んだ確認頻度がどれくらいであるかに依存する．

ライブプログラミング環境を使用していることがプログラミングスタイルに影響を与えることが観測された．ライブプログラミング環境を使用している際には，より逐次的なプログラムを書いている傾向があり，そうでない環境を使っている場合には，関数を分割する傾向があった．これは逐次的なプログラムのほうがライブプログラミング環境と相性が良いことを被験者が経験的に学んだものと考えられる．これは，ライブプログラミングが絵を描画する等の直線的になりやすいプログラミングで応用されてきたことと関連がある．

以下に本実験から考えられることを述べる．

予備実験と同程度かすこし大きいプログラムであっても，事前に適切な実行結果の確認タイミングを外部から与えた場合には，被験者らは頻繁に実行結果を確認することでフォルトを早期に発見できると述べた．これは仮説 2 そのものであり，被験者らがライブプログラミング環境での経験を積み，適切な確認タイミングを学習すればライブプログラミング環境での恩恵をより受けられることを意味する．

参考文献

[1] Inventing on Principle.

[2] Khan Academy. GitHub - Khan/live-editor: A browser-based live coding environment. https://github.com/Khan/live-editor. Accessed 2017-1-11.

[3] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's Alive! Continuous Feedback in UI Programming. *SIGPLAN Not.*, 48(6):95–104, June 2013.

[4] Chetan Desai, David Janzen, and Kyle Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bull.*, 40(2):97–101, June 2008.

[5] Chirstopher Micahel Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[6] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. Making Live Programming Practical by Bridging the Gap Between Trial-and-error Development and Unit Testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 11–12, New York, NY, USA, 2015. ACM.

[7] Resig John. Redefining the Introduction to Computer Science. http://ejohn.org/blog/introducing-khan-cs/. Accessed 2015-2-5.

[8] Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM.

[9] Sean McDirmid. Usable live programming. *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software - Onward! '13*, pages 53–62, 2013.

[10] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. How Live Are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop*, PX/16, pages 1–8, New York, NY, USA, 2016. ACM.

[11] Apple Inc. Swift - Overview - Apple Developer. https://developer.apple.com/swift/. Accessed 2015-2-1.

[12] Kodowa Inc. Light Table. http://lighttable.com/. Accessed 2015-2-5.

[13] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: Programming for All. *Commun. ACM*, 52(11):60–67, November 2009.

[14] Kouhei Sakurai and Hidehiko Masuhara. The Omission Finder for Debugging What-should-have-happened Bugs in Object-oriented Programs. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1962–1969, New York, NY, USA, 2015. ACM.

[15] Gustavo Soares, Emerson Murphy-Hill, and Rohit Gheyi. Live Feedback on Behavioral Changes. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 23–26, Piscataway, NJ, USA, 2013. IEEE Press.

[16] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research &Amp; Applications*, ETRA '06, pages 133–140, New York, NY, USA, 2006. ACM.

[17] 中川 尊雄, 亀井 靖高, 上野 秀剛, 門田 暁人, 鵜林 尚靖, and 松本 健一. 脳活動に基づくプログラム理解の困難さ測定. コンピュータ ソフトウェア, 33(2):78–89, 2016.