ライブプログラミング環境における ユニットテスト機能の設計と実現方法

今井 朝貴[1,a)]　増原 英彦[1,b)]　青谷 知幸[1,c)]

概要：ライブプログラミング環境はプログラム中の式をコード編集中に実行し、即座にその結果を表示する。この即時のフィードバックにより、プログラマは一時な式をプログラム中に書き、引数や定義を変更することで試行錯誤的なプログラミングを容易に行える。我々は試行錯誤的なプログラミング過程とテスト駆動開発の類似性に注目し、ライブプログラミングのためのユニットテスト機能を提案する。この機能は (1) 一時的な式とユニットテストケースを簡単な相互変換し、(2) 実行途中に得られた値を式に変換し、ユニットテストの期待値として利用可能にする。我々はこのテスト機能を Shiranui ライブプログラミング環境中に作成した。本発表では、このテスト機能の設計とインタプリタによる実現方式を紹介する。

キーワード：ライブプログラミング，ユニットテスト，デバッグ

Design and Implementation of Unit Testing Features for Live Programming Environments

Tomoki Imai[1,a)]　Hidehiko Masuhara[1,b)]　Tomoyuki Aotani[1,c)]

Abstract: Live programming environments execute expressions in a program during code editing, and instantly show their results. With this instant feedback mechanism, the programmer can easily perform exploratory programming by writing transient expressions, and changing parameters or program definitions. We focus on the similarity between the exploratory programming process and test-driven development, and propose a novel unit testing feature for live programming. The feature (1) seamlessly converts a transient expression and a unit test case, and (2) extracts an intermediate execution result into an expression so that it can be used as an expected value in a unit test case. We realized this feature in our live programming environment called Shiranui. In this presentation, we present the design of those features, and its interpreter-based implementation.

Keywords: Live Programming, Unit Testing, Debugging

1. はじめに

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ようなプログラミング環境のことである (図 1, 2)．これにより，プログラマはパラメータの変更や， `if` 文の追加などの試行錯誤的なプログラミングを容易に行うことができる．

これまでに，ライブプログラミング環境は絵を描くことや，音楽を作ることに使われてきたが [2], [3], [11]，近年になってより一般的なプログラムにも使われるようになってきている [6], [7], [8], [10], [14]．しかし，そのようなライブプログラミング環境はより実践的なプログラムを記述するための「プログラムの正しさを担保する仕組み」を持っていなかった．

本研究では，ライブプログラミング環境で行われる試行錯誤的なプログラミングと，テスト駆動開発の類似点に注目し，ライブプログラミングにより即時的なフィードバックを得つつ，プログラムの正しさを担保するようなシステ

[1] 東京工業大学
Tokyo Institute of Technology
[a)] imai.t.af@m.titech.ac.jp
[b)] masuhara@acm.org
[c)] aotani@is.titech.ac.jp

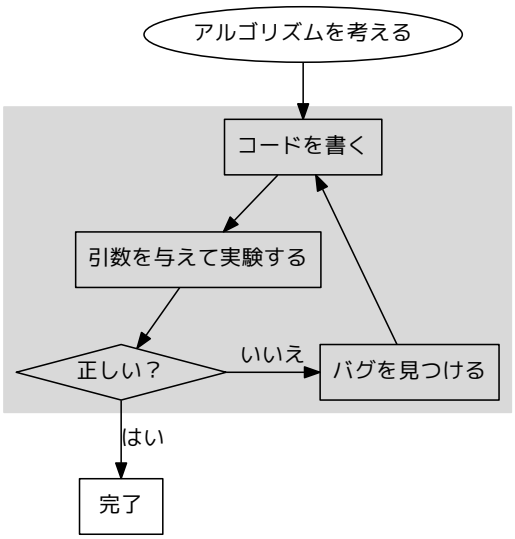


図 1: 通常の (ライブでない) プログラミングにおける行動．(グレーの部分はプログラミング中繰り返し行われる箇所)

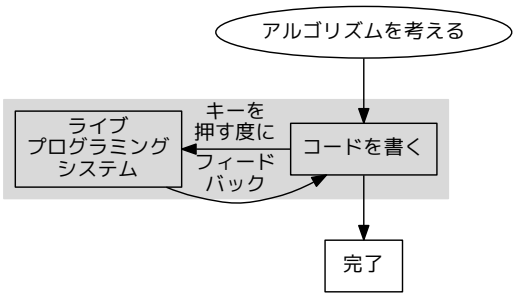


図 2: ライブプログラミングにおける行動の流れ．(グレーの部分はプログラミング中繰り返し行われる箇所)

ムを提案する．

2. 問題

ライブプログラミング環境を使って一般的な (絵や音楽に限らない) プログラムを作成する場合，まず (1) プログラムの動作をすぐに確認するためにこれから作る関数の関数呼び出しを書き (図 3 の 8, 9 行目)，次に (2) 関数本体を完成させる (図 3 の 1 ～ 7 行目)．

このような手順は，最初に関数の呼び出しを記述するという点で，テスト駆動開発 (test-driven development) に似ているが，ライブプログラミング環境では関数の計算結果の正しさをユーザが目視で確認していた．そのため，プログラムを変更した際の再確認は自動的に行うことができない．

一方で，計算結果を照合するための方法としてユニットテストプログラムを作成することが一般的であるが，そのようなテストプログラムは容易ではない．例えば，JUnit [1] を用いたユニットテストプログラムはソースコード 1 のようになる．4, 5 行目が実際にテストを行う部分であるが，“入力” を作成し，“期待される出力” を人手で計算し，記述する必要がある．これらを書く際には，そもそも実行されるべきプログラムが存在していないのでフィードバックを得ることができない．

```
1 func sum(n : Int) -> Int {
2     var r = 0                  (2 times)
3     for i in (0 ... n) {
4         r += i                 (10005 times)
5     }
6     return r/n                 (2 times)
7 }
8 sum(10000)                     5000
9 sum(3)                         2
```

図 3: Apple Swift のライブプログラミング環境の画面．画面の左側がコード，右側に文の実行回数や式の値が表示される．ソースを編集する度に再実行され，右側が更新される．

ソースコード 1: JUnit を用いたテストケース記述例

```
1 public class MathTest {
2   @Test
3   public void testSum() {
4     assertEquals(Math.sum(6, new int[]{1,2,3}));
5     assertEquals(Math.sum(9, new int[]{4,3,2}));
6   }
7 }
```

3. 本研究の貢献

本研究では，ライブプログラミング環境に対し，その利点を失うことがないユニットテスト機構を提案する．

本提案のアイデアは，“期待する動作” の記述をライブプログラミングの機能を使って支援することである．これにより，ライブプログラミング環境において毎回手動で確認する手間を減らし，かつ複雑な期待する値を書く手間も減らすことができる．また，実行時情報からユニットテストの入力を作成する機能により，入力を作成することも容易にする．

具体的には，以下の 3 つを提案する．

- FlyLine: 計算式を記述しておくと計算結果が編集の度に即座に更新されていく記法と，計算式と期待値を照合するユニットテストの記法を統一したもの．これにより開発者は (1) 計算式の記述を行なった後に (2) 関数定義を完成させ (3) 挿入された計算結果を見て正しいと確認したら (4) ユニットテストに変換する，といった一連の手順でライブプログラミングを利用しつつ，テスト駆動開発を行うことができる．(5.1 節)
- FlyMark: 関数定義の途中に式を記述し，その評価結果の履歴を表示する機能．特に履歴中の値から，それを得た呼び出し文脈やコードの実行範囲を表示することができる． (5.4 節)

- FlyMark 機能を使い，呼び出し文脈から値を取り出し，テストのための式を生成する機能．循環参照や関数オブジェクトを含むような複雑な値にも対応する．(5.7 節)

4. Shiranui 言語

本研究で提案する Shiranui では，ライブプログラミングの機能を設計するために独自の言語を設計した．Shiranui 言語は動的型付けな命令型言語であり，レキシカルスコープを持つ関数を第一級オブジェクトとして扱うことができる．

Shiranui 言語の言語としての基本的機能は JavaScript のサブセットと同一であるため，抽象構文のみを示すこととする．

Shiranui 言語は 5 章で述べる開発環境を実現するためのデータ用 DSL (4.3 節) 等の特別な機能を持つ．本章ではそれらについては，文法のみをここで示し，実際の使用方法については 5 章で述べる．

4.1 データ型

Shiranui 言語では，以下のものを組み込みのデータ型としてサポートしている．

- 整数型
- 文字列型
- 真偽値型
- リスト型
- 参照型
- 関数オブジェクト型

このうち，リスト型と参照型は変更可能な値である．ここでの変更可能な値とは，オブジェクトが生成された後に，その値が変化するものを指す．

また，Shiranui 言語は動的型付けな言語であり，例えば整数型とリストの足し算のような型エラーは実行時に検出される．

4.2 文法

本節では，Shiranui 言語の文法を抽象構文により説明する．Shiranui 言語の構成要素は主に式と文である．特殊な文法については，5 章で説明する．

4.2.1 式

図 4 に Shiranui 言語の式の抽象構文を示す．ほとんどの構文の意味は JavaScript 等のものと同じである．

構文 `\id(v1, v2 ...) { stmts }` は仮引数が `v1, v2 ...` で，本体が `stmts` な関数オブジェクトを生成する．`id` は関数の識別子で，4.3 節のデータ用 DSL から関数を参照する際に使用される．

Shiranui 言語では， ML 言語と同様に明示的な参照を使用する．`ref e` により，明示的な参照を初期値 `e` で作成し， `!e` で中身の値を取り出し，`e <- e` で参照の中身を変更する．

データ用 DSL は関数オブジェクト等のデータを文字列化するための DSL で，4.3 節で詳細に述べる．

4.2.2 文

図 5 に Shiranui 言語の文の抽象構文を示す．式と同じく，ほとんどの構文の意味は JavaScript 等のものと同じである．`for v in e { stmts }` の意味は， `e` を評価した結果のリストに対し，各要素を `v` に束縛して `stmts` を順に評価するという意味である．FlyLine は本研究で提案するテスト機能のための構文であり，その文法と意味は 5 章で説明する．

4.3 データ用 DSL

5 章で述べる Shiranui の開発環境は，実行中に現われるデータを文字列として表示すること (文字列化という) を要求する．そこで，Shiranui では，循環を含むデータや関数オブジェクトを復元可能な文字列として表現するために，データ用の領域特化言語 (DSL) を設計し実装した．

4.3.1 文法

データ用 DSL の抽象文法を図 6 に示す．

データ用 DSL はホスト言語上で式が期待される場所に書くことができる．DSL は<|で始まり|>で終わり，この中は専用のパーザで解釈される．

データ用 DSL 内部に現われる式を DSL 内部式と呼ぶ．また，データ用 DSL 内には変数があり循環の解消等に使用される．DSL 内部式は次の 3 種類が存在する．

- データを表わす即値
- 変数定義
- 変数

データを表わす即値は，ホスト言語とほとんどのデータは同じように書くことができる (ソースコード 2)．しかし，例えばリストの場合には各要素を書き下す文法しか許されず，また各要素は DSL 内部式として解釈される．各要素が DSL 内部式として解釈されることによって，リスト全体を変数に代入しておき内部の要素でそれを参照するといったことが可能になる．

ただし，関数オブジェクトの文法はホスト言語の文法とはまったく違う．これについては 4.3.3 節で述べる．

ソースコード 2: ホスト言語と同様に即値を書くことができる

```
<|1|>        ;
<|"string"|> ;
<|true|>     ;
<|[1,2,3]|>  ;
<|ref 1|>    ;
```

(式)

e	::=	n	(数値リテラル)
	|	*str*	(文字列リテラル)
	|	`true` | `false`	
	|	v	(変数参照)
	|	e *op* e | `-`e	(演算式)
	|	e`(`e`,...)`	(関数呼出)
	|	`ref` e | `!`e | e`<-`e	(参照生成，取り出し，代入)
	|	`[`e`,...]`	(リスト生成)
	|	`[`e`..`e`)`	(半開区間リスト)
	|	`[`e`..`e`]`	(閉区間リスト)
	|	`\[`v`](`v`,...){`s`...}`	(関数オブジェクト)
op	::=	`+` | `-` | `*` | `/` | `%`	

図 4: Shiranui 言語の式

(文)

s	::=	e`;`	(式文)
	|	`{`s`...}`	(ブロック)
	|	`if` e`{`s`...}` [`else` `{`s`...}`]	(条件文)
	|	`return` e`;`	(return 文)
	|	`for` v `in` e `{`s`...}`	(for 文)
	|	`let` v `=` e`;`	(変数宣言)
	|	`mut` v `=` e`;`	(let v = ref e; の糖衣構文)

(全体)

p	::=	s^*	(文 の繰り返し)

図 5: Shiranui 言語の文

(データ用文)

d	::=	`<|`g`|>`	
g	::=	x	(変数)
	|	x `=` g	(変数定義)
	|	n	(数値リテラル)
	|	*str*	(文字列リテラル)
	|	`[`g`,...]`	(リスト)
	|	`ref` g	(参照)
	|	`$(`v`->`g`,...)`v	(関数オブジェクト)

図 6: Shiranui 言語のデータ用 DSL

変数定義は，ソースコード 3 のように記述する．この変数は<|から |>の中で有効な大域変数として振る舞う．

ソースコード 3: DSL 内の変数定義．変数 a にデータ [1,2,3] を代入する

```
<|a=[1,2,3]|>;
```

変数は，ソースコード 4 のように記述する．このとき，定義していない変数を使用すると実行時エラーが発生する．

ソースコード 4: DSL 内の変数．変数 a をそのまま書くと変数として解釈される

```
<|a|>; // -> error
<|a=[a,1]|>;
```

4.3.2 循環を含むデータ

循環を含むデータは，例えば有向グラフをデータとして持つ時に現われる．DSL では，そのようなデータに対し，DSL 内部の変数を使うことで循環を表現する．

例えば，ソースコード 5 は，図 7 のデータを表わしている．

ソースコード 5: 循環を持つデータ

```
<|a=[1,a]|>;
```

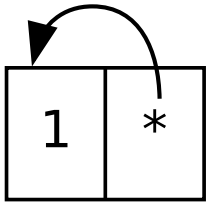


図 7: 循環を含むデータ

4.3.3 関数オブジェクトの文字列化

Shiranui 言語の第一級オブジェクトの関数は，レキシカルスコープを持つ．よって，関数オブジェクトはラムダ式とレキシカル変数の束縛状況のペアで表現することができる．

4.3.3.1 ラムダ式の文字列化

Shiranui 言語において，ラムダ式の文字列化は以下の二通りが考えられる．

(1) ラムダ式をコピー& ペーストする

(2) プログラマがラムダ式に識別子を割り振り，その識別子を使用する

Shiranui では 2 の方法を使用する．文字列化した文字列化した関数オブジェクトが，ラムダ式の変更に追従できる

ようにするためである．1 の方法では，プログラム中のラムダ式が変更された場合に，テスト中の文字列化した関数オブジェクトはその変化に追従することができない．

4.3.3.2 レキシカル変数の束縛状況の文字列化

例えば，ソースコード 6 の fact という関数オブジェクトを文字列化すると，ソースコード 7 の式となる．

ソースコード 6: 階乗を計算する fact

```
let fact = \lambda_id(n){
    if n = 0 {
        return 1;
    }
    return n * fact(n-1);
};
```

ソースコード 7: 6 の fact を文字列化した結果

```
<|a=$(fact->a)lambda_id|>;
```

$() で囲まれた内部 fact->a が環境を表わし，lambda_id がラムダ式を表わしている．また，関数オブジェクトを文字列化する際には，ラムダ式内部で現われる自由変数のみをレキシカル変数の束縛状況として文字列化する (図 8)．

```
mut a = 0;
mut unused = -1;

let f = \id(){
    return a;
};
```

```
<|$(a->ref 0)id|>
```

図 8: 関数 f は変数 unused を使用しない為，束縛状況に含めない

4.3.3.3 もうすこし複雑なデータの文字列化

関数オブジェクトを文字列化するもうすこし複雑な場合として，ソースコード 8 のような場合が考えられる．f,g は互いに shared という変数を参照する状態である．Shiranui は，循環を解決するのと同じ方法でこれも正しくソースコード 9 のように文字列化することができる．

ここでの正しさとは，文字列化した関数オブジェクトを実行した際，戻り値がプログラム中に出現したときと同じであることを指す．グローバル変数に対する副作用等は保証していない．この制限は，Shiranui が想定する文字列化した関数オブジェクトの使用方法 (実験，テスト) を考えると問題ではない．

ソースコード 8: 複雑な循環を持つ構造

```
mut shared = 0;
let f = \f_id(){
    shared <- !shared + 1;
    return !shared;
};
let g = \g_id(){
    shared <- !shared - 1;
    return f();
};
```

ソースコード 9: 8 の g を文字列化

```
<|$(f->$(shared->a)f_id,shared->a=ref 0)g_id|>;
```

4.3.4 深さ優先走査と幅優先走査の比較

循環を含むデータを文字列化するときに，どこを変数とするかは一意ではない．変数の位置を決めるためには，データを走査する．このとき，幅優先走査を使用することで，変数定義が現れる場所が，データ構造の浅いところに現われる．変数の場所に関して利点がある為，Shiranui は，幅優先走査を採用している．

例えば，有向グラフをリストのリスト，内部のリストはノードを表わし, 最初の要素はノードの値で，残りの要素はエッジを表わすような構造で管理するとする．すると，二つのノードがあり，互いにエッジを張っているような場合には，図 9 のようなデータ構造となる．

表 9 のようなデータを文字列化することを考えると，ソースコード 10，11 の二通りに文字列化することができる．

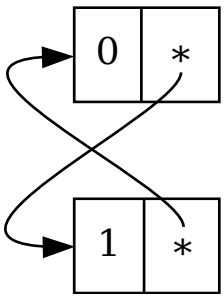


図 9: 複雑な循環データ

ソースコード 10: 深さ優先走査を使った文字列化

```
<|[a=[0,b=[1,a]],b]|>
```

文字列化のアルゴリズムは，最初に二度以上現われるデータを走査，次に，二度現われるデータを変数に置き換

ソースコード 11: 幅優先走査を使った文字列化

```
<|[a=[0,b],b=[1,a]]|>
```

えながら変換するものである．詳しくは 7.2 節で述べる．

ソースコード 10，11 の違いは，最初にどのように二度出現する値を探すかの違いである．ソースコード 10 は深さ優先走査を使ったものである．リストの 1 番目の要素を深く走査するため，変数 b の定義が 1 番目の要素に現われる．ソースコード 11 は，幅優先探索を使ったものである．リストの要素を平等に走査するため，変数 b の定義が二番目の要素に現われる．

5. 提案

本章では，提案手法を実装したプログラミング環境 Shiranui の設計について述べる．Shiranui 言語の 開発環境として，Emacs 上に Kasumi モードを設計，実装した．Kasumi はライブプログラミングとテストを統合する機能を備えた開発環境である．

実験とテストを統合したインターフェイスである FlyLine(5.1 節) は，実験を表わす IdleFlyLine からユニットテストである TestFlyLine に簡単に変換できるようになっている．これにより，ユーザは一度目視で確認した実験をテストにして，正しさの判定を自動化することができる．

インラインに printf デバッグを可能にする FlyMark(5.4 節) は，FlyLine ごとに出力を記録することで，実験やテスト毎に printf デバッグが可能である．また，値を表示するだけではなく，ジャンプすることで，値を表示したときの周りの変数の値も表示することができる．

また，複雑なテストパラメータを実行時の情報から取り出し，そのまま実験あるいはテストにする機能により，テストの入力パラメータを手で記述する手間を削減する．

実際に Shiranui を使ってプログラムを書く動画を以下に用意した．

- フィボナッチ関数:
  `https://www.youtube.com/watch?v=LZy2n3xJhb0`
- 継続渡しスタイルの階乗計算:
  `https://www.youtube.com/watch?v=4okfUkQ_HoA`

5.1 FlyLine

FlyLine は，Shiranui の開発環境の中心的な機能である．FlyLine は，任意の式の評価結果を表示しておく機能で，プログラムが 1 文字編集される度に結果が更新される．

FlyLine の抽象構文を図 10 に示す．FlyLine は “`op exp -> exp;`” という文法を持つコメントで，“`op`” は “`#+`” または，“`#-`” である．`op` が “`#+`” の FlyLine を IdleFlyLine，“`#-`” の FlyLine を TestFlyLine と呼ぶ．また，FlyLine 上

(FlyLine)

f	::=	`#+` e `->` e;	(IdleFlyLine)
	|	`#-` e `->` e;	(TestFlyLine，テストがパスした)
	|	`#-` e `->` e `|` e;	(TestFlyLine，テストがパスしない)

(式を拡張)

e	::=	d	(データ用 DSL)
	|	...	

(全体の拡張)

p	::=	$(s \mid f)^*$	(式，FlyLine の繰り返し)

図 10: FlyLine の抽象構文

```
1 #+ f(3) -> 1;
2
3 let f = \(n){
4     return 1;
5 };
```

図 11: f を実験する．
1 行目は f(3) の結果は 1 である，と読む

```
1 #+ f(3) -> 6;
2
3 let f = \(n){
4     if n = 1 {
5         return 1;
6     }else{
7         return n*f(n-1);
8     }
9 };
```

図 12: プログラムが変わると，IdleFlyLine の右辺を自動的に更新される (1 行目)

の->の左側を左辺，右側を右辺と呼ぶ．

開発環境は応答性を上げるために FlyLine を並列に評価する．それにより，ある FlyLine が時間のかかる計算をしている場合や無限ループに陥いった場合でも他の FlyLine は動作する．

5.1.1 IdleFlyLine

IdleFlyLine は，関数の挙動を実験するためのものである．IdleFlyLine は，`#+`から始まる行である (図 11 の 1 行目)．左辺は実験したい式であり，右辺は左辺を評価した結果である．IdleFlyLine はコードの編集があると，左辺を再度評価し右辺を更新する (図 12)．

IdleFlyLine の使用は，スクリプト言語が持つ Read-Eval-Print-Loop(REPL) で関数の実験をする作業 (図 13) に似ている．

しかし，S. McDirmid [7] によると，“First, REPLs (read-eval-print loop) enable the progressive input of top-level statements, displaying results as statements are written. However, REPLs are intended for quick experimentation and feedback is not available for method definitions.” とある．また，履歴は残るが，これまでの実験を即座に再利用することはできない．しかし，IdleFlyLine は関数の再定義に関してフィードバックを提供，実験を再利用することで，REPL より優れた実験環境を提供する．

また，コメントとして動作結果を記述するような方法も存在する [12] が，IdleFlyLine は，コメントと違い，動作することが保証されている．

5.1.2 TestFlyLine

TestFlyLine は，`#-`から始まる行である．左辺はテストし

```
> let f = \(n){return n;};
<|$()unknown|>
> f(3);
3
> f(5);
5
>
```

図 13: Shiranui の REPL

たい式，右辺は期待する値を表わす式である．TestFlyLine はテストしたい式を評価し，その評価結果が期待する値と一致する場合には緑の下線を引く．異なる場合には赤の下線を引き，右辺の右側に左辺の評価結果を表示する (図 14, 15)．これにより，TestFlyLine はユニットテストとして機能する．

```
#- f(3) -> 6;
#- f(4) -> 24;

let f = \(n){
    if n = 1 {
        return 1;
    }else{
        return n * f(n-1);
    }
};
```

図 14: 成功した TestFlyLine

```
#- f(3) -> 6 || 1;
#- f(4) -> 24 || 1;

let f = \(n){
    // !!!!BUG!!!!!
    return 1;
};
```

図 15: 失敗した TestFlyLine

評価中に 0 除算等のエラーが発生した場合には，図 16 のようにエラーメッセージを表示する．

5.2 FlyLine の評価順序と独立性

FlyLine はテストケースとして動作するよう，その評価順序はソースコードのどこに書いてあるかに依存しない．これは，テストケースの成否が実行順序に依存すべきでないからである．

また，FlyLine は環境を共有しない．これにより JUnit の setup メソッドのように，グローバル変数によってテスト環境を設定することになる．

5.2.1 評価順序

FlyLine は，ソースコード上の位置と関係なく，必ず最後に実行される．これは，FlyLine を実験，テストしたい対象の上に書くためである．もし，FlyLine をソースコード上の位置で実行することにすると，例えば，図 17 の FlyLine(1 行目) は変数 n がまだ定義されていないためエラーになる．

```
#- may_cause_error(0) -> 0 || "Division by 0";
let may_cause_error = \(n){
    return 4 / n;
};
```

図 16: 0 除算エラーの表示

```
#+ n -> 3;
let n = 3;
```

図 17: 1 行目の FlyLine はまだ定義されていない n を参照できる

5.2.2 独立性

FlyLine は，ソースコード中に複数記述される．もし，複数の FlyLine で環境を共有していたとすると，FlyLine の評価の順番等によって FlyLine の実験結果，テスト結果が変わってしまうことがある．そのような動作は，TestFlyLine がユニットテストとして機能することを難しくする．よって，そのような動作を防ぐために FlyLine は独立したインタプリタと環境で実行される (図 18)．

```
mut gm = 0;
let inc_gm = \(){
    gm <- !gm + 1;
    return !gm;
};

#+ inc_gm() -> 1;
#+ inc_gm() -> 1;
```

図 18: FlyLine は独立に動作する．独立でなければ，8 行目の FlyLine の右辺は 2 になる

5.3 FlyLine の選択

FlyLine は，ソースコード中に複数記述することができる．しかし，後述する 5.5 節のトレースブラウジングや，5.7 節のテストの作成等で，ある FlyLine に注目することが必要となる．FlyLine を選択すると，別の Emacs バッファに実行時の評価情報が表示される (図 19)．また，左辺の関数呼び出しの関数を表わすラムダ式の本体が緑にハイライトされる (図 19 の 5,6,7 行目)．

```
#+ f(0) -> 0;
#+ f(1) -> 1;
#+ f(2) -> 2;

let f = \(k){
    return k;
};
```

```
k at [69,70] = 1
```

図 19: 2 行目の FlyLine を選択

このとき，コード上で選択を行うと，対応する評価情報がハイライトされる (図 20)．ハイライト機能は，どの評価情報を見たいかを選択するときに使う．

```
#+ f(0) -> 0;
#+ f(1) -> 1;
#+ f(2) -> 2;

let f = \(k){
    let a = k;
    let b = k;
    return k;
};
```

```
k at [70,71] = 1
k at [85,86] = 1
k at [99,100] = 1
```

図 20: 評価情報のハイライト

5.4 FlyMark

FlyMark は，Shiranui で “printf デバッグ”[*1] を可能にする仕組みである．FlyMark は，#*から始まる行である (図 21 の 3 行目)．FlyMark は，関数の本体に記述する．FlyMark の出力は，複数のテストで独立するために，FlyLine ごとに独立しており，結果を見たいときには FlyLine を選択する．FlyLine を選択すると，自動的に FlyMark の右辺に評価結果が挿入される (図 22)．

```
1 #+ f(10) -> 3628800;
2 let f = \(n){
3     #* n -> ;
4     if n = 0 {
5         return 1;
6     }else{
7         return n*f(n-1);
8     }
9 };
```

図 21: 選択する前

```
1 #+ f(10) -> 3628800;
2 let f = \(n){
3     #* n -> 10,9,8,7,6,5,4,3,2,1,0;
4     if n = 0 {
5         return 1;
6     }else{
7         return n*f(n-1);
8     }
9 };
```

図 22: 選択した後，FlyMark の右辺が自動的に更新される (3 行目)

```
1 #+ f(10) -> 3628800;
2 #+ f(3) -> 6;
3 let f = \(n){
4     #* n -> 3,2,1,0;
5     if n = 0 {
6         return 1;
7     }else{
8         return n*f(n-1);
9     }
10 };
```

図 23: 違う FlyLine を選択した後，FlyMark の右辺が更新される (3 行目)

また，複数の FlyMark が同一の関数内部にあったとき，FlyMark はどこにいるかという表示を同期させる．図 24 では，4 行目で “2” を選択すると，8 行目の FlyMark も “2” のところに下線が追加される．

5.5 トレースブラウジング

本節では，5.3 節で述べた FlyLine の選択をした後の，評価情報を見る機構について説明する．このような評価情報を見ることを，トレースブラウジングという．トレースブラウジングは，FlyLine を使って実験，テストをした結果をより詳細に見るために行う．

トレースブラウジングは，関数呼び出し単位でトレースを扱う．現在注目している関数はコード中でハイライトされる．Shiranui は，注目している関数呼び出し内の変数の値，関数呼び出しの値を表示する．

[*1] プログラムに “printf” 関数を挿入し，変数の値を出力することで，プログラムの振舞いを調べるデバッグ方法

```
1 #+ f(3) -> 7;
2
3 let f = \(n){
4     #* n -> 3,2,1,0;
5     if n = 0 {
6         return 1;
7     }else{
8         #* n -> 3,2,1;
9         return n + f(n-1);
10    }
11 };
```

```
n at [58,59] = 2
n at [133,134] = 2
f at [137,138] = <|$()no_name|>
n at [139,140] = 2
f(n-1) at [137,143] = 2
```

図 24: 複数の FlyMark は同期する (4,9 行目)

5.5.1 Dive

Dive は，関数の呼び出しを辿っていくことでバグを見つけるような方法である．Dive は，FlyLine を選択した後に，注目している関数呼び出しの中の関数呼び出し式を選択して行う (図 25)．すると，選択した関数呼び出しを注目する (図 26)．

```
1 #+ f(10) -> 3628800;
2
3 let f = \i(n){
4     if n = 0 {
5         return 1;
6     }else{
7         return n*f(n-1);
8     }
9 };
```

```
n at [45,46] = 10
n at [97,98] = 10
f at [99,100] = <|a=$(f->a)i|>
n at [101,102] = 10
f(n-1) at [99,105] = 362880
```

図 25: 関数呼び出しを選択 (7 行目)

```
1 #+ f(10) -> 3628800;
2
3 let f = \i(n){
4     if n = 0 {
5         return 1;
6     }else{
7         return n*f(n-1);
8     }
9 };
```

```
n at [45,46] = 9
n at [97,98] = 9
f at [99,100] = <|a=$(f->a)i|>
n at [101,102] = 9
f(n-1) at [99,105] = 40320
```

図 26: n=10 のときの f(n-1) を注目した．右のバッファから n=9 とわかる

なお，FlyLine の選択とは，厳密には左辺の関数呼び出しに対する Dive である．

5.6 FlyMark

5.4 節で述べた，FlyMark を使うことでも関数呼び出しを注目できる．FlyMark の右辺は，右辺の評価履歴である．右辺を選択すると，右辺を表示したときの関数呼び出しに移動することができる．

5.5.1 節で述べた，Dive は，関数呼び出しを一つずつ辿ることしかできない．しかし，FlyMark を使うことで関数呼び出しを一気にジャンプすることができる．

図 27，図 28 では，他の関数内部の FlyMark を指定して一気にジャンプしている．

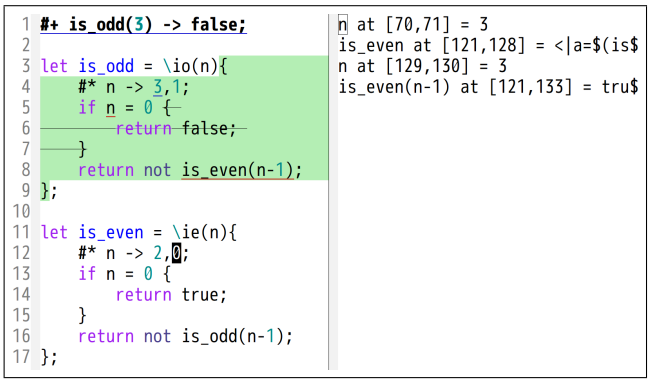


図 27: ジャンプ前

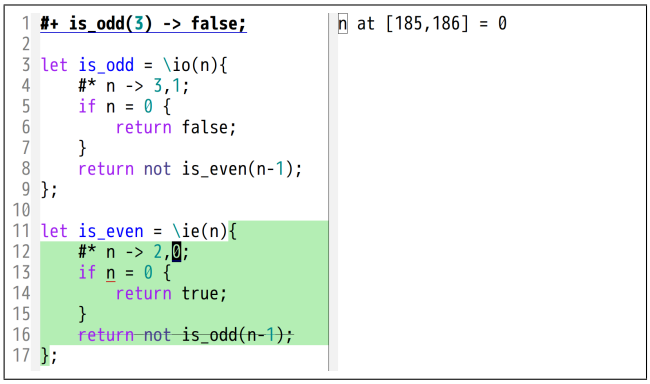
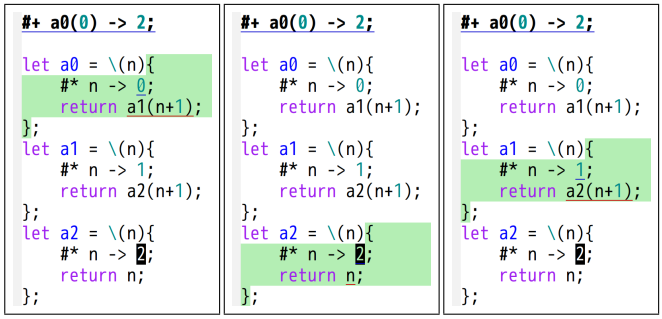


図 28: 他の関数内部の FlyMark で一気にジャンプ

5.6.1 呼び出し元へのジャンプ

FlyMark を使ってジャンプした際には，呼び出し元での操作を見て正しさを検証しなければならないことがある．そのような場合には，呼び出し元に戻る機能を使って，呼び出し元へジャンプする (図 29)．



a0 にいる　2 を選択し a2 に移動　呼び出し元 a1 に移動

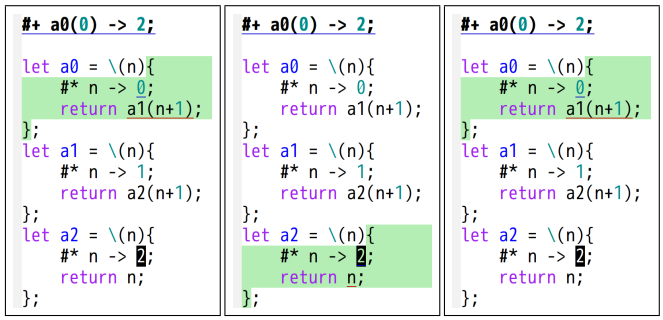
図 29: 呼び出し元へのジャンプ

5.6.2 ジャンプ元に戻る

FlyMark で一気にジャンプした場合には，元の場所に戻ることもできる (図 30)．Dive した後にジャンプ元に戻った場合，履歴が取り消されることを除いて呼び出し元へのジャンプと同じである．

5.7 テストの作成

テストを作ることはバグを埋めこまない為に重要である．Shiranui でテストを作るときには，TestFlyLine を記述する．



a0 にいる　2 を選択し a2 に移動　ジャンプ元 a0 に移動

図 30: ジャンプ元へ戻る

5.7.1 IdleFlyLine の変換

5.1.1 で述べた IdleFlyLine は，TestFlyLine に変換できる．IdleFlyLine を使って実験し，それが正しかった場合には，TestFlyLine に変換 (accept という) することでテストを増やすことができる (図 32)．実験結果が間違っていて，答えがわかっている場合には，ユーザが直接結果を入力する (decline という) もできる (図 33)．decline した場合には，カーソルが右辺に移動し，入力待ちの状態になる．

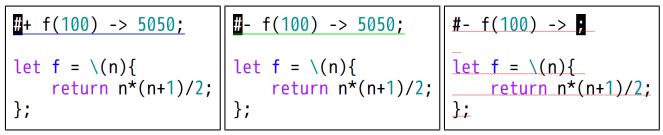


図 31: 変換前　図 32: accept 後　図 33: decline 後

また，ある関数がテストから正しいということがわかった場合には，IdleFlyLine を記述し，TestFlyLine に変換することにより，回帰テストを容易に作成できる．

5.7.2 実行結果の一部を編集

大きなデータ構造を扱うようなプログラムを記述している場合，実行結果と期待する値の一部だけが異なる場合がある．そのような場合には，一度 accept した後にその値を編集することにより，期待する値を作成することができる (図 34)．

5.7.3 途中の評価結果からテストを作成

Shiranui 開発環境では，実行時情報からテストを作成することができる．それにより，5.5 節のトレースブラウジング中に発見した，テストしたい関数呼び出しとその複雑な引数をそのままテストにすることができる (図 35, 36)．

6. 言語機能との連携

本節では，Shiranui 言語の持つ特別な機能と Kasumi の連携について述べる．

6.1 データ用 DSL

文法や，意味等については 4.3 節で述べた．データ用

```
// expected: [1, 2, 3]
#+ l123() -> [1,2];
let l123 = \(){
    return [1,2];
};
```

1. l123() は [1,2,3] を返すべきだが，[1,2] になっている

```
// expected: [1, 2, 3]
#- l123() -> [1,2];
let l123 = \(){
    return [1,2];
};
```

2. 一旦，[1，2] を期待する値として accept する

```
// expected: [1, 2, 3]
#- l123() -> [1,2, 3] || [1,2];
let l123 = \(){
    return [1,2];
};
```

3. [1，2] の最後に 3 を追加し，期待する値を作成

図 34: 実行結果の一部を編集して期待する値を作成

```
#+ f(4) -> 24;
let f = \id(n){
    if n = 0 {
        return 1;
    }
    return n * f(n-1);
};
```

```
n at [39,40] = 4
n at [82,83] = 4
f at [86,87] = <|a=$(f->a)id|>
n at [88,89] = 4
f(n-1) at [86,92] = 6
```

図 35: 関数呼び出しを選択

```
#+ f(4) -> 24;
#+ <|a=$(f->a)id|>(3) -> 6;

let f = \id(n){
    if n = 0 {
        return 1;
    }
    return n * f(n-1);
};
```

```
n at [68,69] = 4
n at [111,112] = 4
f at [115,116] = <|a=$(f->a $
n at [117,118] = 4
f(n-1) at [115,121] = 6
```

図 36: テストとして取り出す

DSL は，循環を含むデータおよび関数を文字列化するためのものだった．

6.1.1 循環を含むデータ

例えば，循環を含むデータは，図 37 のように，循環を含むグラフを扱うプログラムで出現する．この際に，5.7.3 節で述べた，実行時情報からテストを作る機能を使って，ユーザは，グラフのデータを手で書くことなく，テストを作成できる．

また，Shiranui のデータの文字列化は，ユーザ自身が読むことも想定し，読みやすい形で出力される．

```
#+ main() -> 1;

#+ <|$()hg|>(<|[a=[1,b],b=[2,a]]|>) -> 1;
let main = \m(){
    let cycle = make_cycle_graph();
    let ans = get_ans(cycle);
    return ans;
};

let get_ans = \hg(graph){
    return 1; // dummy
};
```

図 37: 循環を含むグラフをテストに

6.1.2 関数オブジェクト

関数オブジェクトは，関数の中に定義された関数として出現することがある．関数の中に定義された関数は，グローバルに定義された関数よりも実験しづらい．なぜなら，直接変数名を指定して呼び出すことができないからである．しかし，そのようなときでも，関数を文字列化できることを利用して，FlyLine の左辺に配置することができる．

```
1 #+ out(3) -> 9;
2
3 let out = \o(n){
4     let in = \i(a){
5         return n*a;
6     };
7     let ret = in(3);
8     return ret;
9 };
```

```
in at [96,98] = <|$(n->3)i|>
in(3) at [96,101] = 9
ret at [114,117] = 9
```

図 38: 中にある関数の呼び出しを選択 (7 行目)

```
1  #+ out(3) -> 9;
2  #+ <|$(n->3)i|>(3) -> 9;
3
4  let out = \o(n){
5      let in = \i(a){
6          return n*a;
7      };
8      let ret = in(3);
9      return ret;
10 };
```

```
n at [95,96] = 3
a at [97,98] = 3
```

図 39: 文字列化し，FlyLine の左辺に配置する (2 行目)

```
#+ use_closure_inside() -> 900;

// taken from use_closure_inside();
#+ <|$(i->ref 300)local_closure|>(3) -> 900;
#+ <|$(i->ref 300)local_closure|>(1) -> 300;
let use_closure_inside = \uci(){
    mut i = 300;
    // this closure can't be seen from outside
    let closure = \local_closure(n){
        return !i * n;
    };
    return closure(3);
};
```

図 40: 見えない関数をテスト

6.2 データのバージョンと履歴

Shiranui 言語には，リストや参照等の変更可能な値が存在する．それらの変更可能な値に対し 5.5 節の変数の値表示を行う際には，選択した関数呼び出しのときの値を表示することができる．7.1 節で述べる，実行時情報のバージョニング機能を使うことで，これを実現している (図 41)．

また，バージョニング機能は参照の中の参照のように，複数の変更可能データがネストしている場合にも機能する (図 42)．

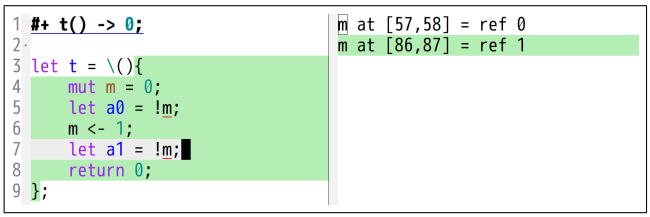


図 41: 評価した時のバージョンまで巻き戻す

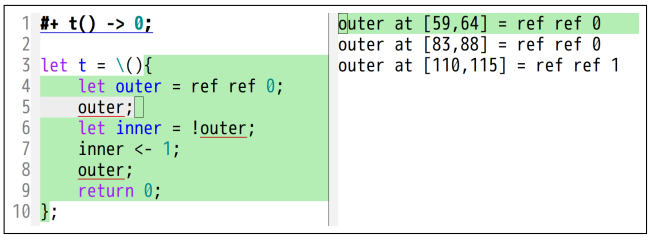


図 42: ネストした参照の巻き戻し

7. 実装

本章では，Shiranui の処理系の構造とアルゴリズムを述べる．

Shiranui のソースコードは，7200 行の C++と，900 行の Emacs Lisp で構成されている．実装はオープンソースソフトウェアとして，https://github.com/tomoki/Shiranui より入手可能である．

Shiranui の言語処理系は構文木を辿るインタプリタである．しかし，Shiranui 開発環境は実行時の式の評価結果等の情報を必要とする．よって，インタプリタは，式の評価結果を記録しながら実行する．その際に，変更可能な値については変更履歴も記録する．また，Shiranui 言語にはデータ DSL のような特別な機能がある．

Kasumi は Shiranui サーバと，Unix パイプで通信している．しかし，IdleFlyLine のようにソースコードを書き換えることがあると，実際に Emacs 上に表示されているソースコードと，サーバ上に存在する構文木データとの間に不一致が発生する．その不一致を補正する機能が Kasumi 側で必要となる．

7.1 実行時情報の保存

Shiranui 開発環境において，デバッグ時には実行時情報をユーザに提示することが必要となる．ここでいう実行時情報とは，式の評価結果，関数呼び出しの順番である．

Shiranui では，実行時情報を抽象構文木に保存している．抽象構文木に保存することにより，カーソルの位置の式の返り値を読み込む等の実装が容易になる．また，FlyLine ごとに独立して実行時情報を保存するために，抽象構文木を FlyLine の数だけ生成する．

7.1.1 変更可能な値の保存

インタプリタは構文木を辿るものであるため，式の評価結果を保存するのは容易である．しかし，保存する際に，式の評価結果として返り値のポインタを持っておくだけでは不十分である．なぜなら，Shiranui には，参照やリスト等の変更可能なデータが存在するからである．

Shiranui 言語中の変更可能データは内部的に変更履歴を記録するように定義されており，バージョン番号と記録された変更を適用 (flush)，巻き戻し (rollback) できるインターフェイスを持つ．データを更新する際には，更新を行うだけでなく，巻き戻しができるように履歴を記録し，バージョン番号を増加させていく．

また，式を評価して得た結果を記録する際には，その値とそこから到達可能な値のバージョンを記録する．これは参照の先にある参照等のように，ネストした変更可能データには対応するためである．関数オブジェクト (クロージャ) のために，環境にもバージョンを付けているため，それも保存する．この操作は，値に対する深さ優先走査で実装することができる．

7.2 データ用 DSL

4.3 節で述べたデータ用 DSL の値から文字列 (式) への文字列化， 文字列 (式) から値への 実体化の実装について述べる．

7.2.1 文字列化

文字列化は 2 パスで行う．

(1) 深さ優先走査を行い，循環を検出する

(2) 幅優先走査を行い，文字列化を行う．(1) で発見した循環する値を文字列化する際には，最初は値の前に変数をつけ，2 回目以降は変数のみを出力する．

7.2.2 実体化

実体化も 文字列化と同様に 2 パス で行う．

(1) 文字列式の構文木に深さ優先操作を行い，実際のデータ構造を生成する．変数定義を発見した場合にはその値を記録し，変数を発見した場合にはプレースホルダを設置しておく．

(2) (1) で生成したデータ構造に対し深さ優先操作を行い，データ構造上のプレースホルダをその変数が参照する値に置換する．

7.3 標準ライブラリ kagero.nui

標準ライブラリには，print や set といった，C++で定義された関数が存在する．実際には，system_call 関数のみが組み込み関数として直接定義されている．system_call 関数は文字列を受け取り，関数を返す．system_call で返ってくる関数は，Shiranui 上で定義された関数と違い，データを文字列化する際には環境に含まれず，無視される (ソースコード 13)．

ソースコード 12: kagero.nui の一部

```
let print = system_call("print");
let get = system_call("get");
```

ソースコード 13: 組み込み関数 p は文字列化時に無視される

```
#+ g -> <|$(free_func->$()ff)g|>;

let p = system_call("print");
let free_func = \ff(n){
    return n;
};
let g = \g(){
    return free_func(p);
};
```

7.4 開発環境実装

Shiranui と Kasumi は Unix パイプで通信している (図 43)．通信は非同期に行われるが，Emacs はシングルスレッドでの動作しかしないため，通信で送られてきたデータの処理をしている間はキー入力を受け付けない．

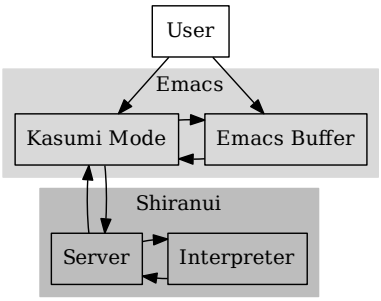


図 43: Kasumi の通信

7.4.1 サーバ実装

サーバは，別プロセスで実行されている Kasumi からの命令を受信し，インタプリタを動作させたり構文木からデータを読み取ったりする．

7.4.1.1 ソースコードの変更

Kasumi から，ソースコード変更通知を受け取った場合，以下のように動作する．

(1) 既に起動しているインタプリタ実行スレッドを終了
(2) シンタックスエラーをチェック
(3) FlyLine を除き，プログラムを実行
(4) FlyLine の数だけスレッドを生成し，FlyLine の左辺を実行
  (a) プログラムを再実行
  (b) FlyLine の左辺を実行
  (c) FlyLine の種類に応じて Kasumi に命令を送信
    (i) IdleFlyLine なら右辺に返り値を表示
    (ii) TestFlyLine なら成否を表示
  (d) トレースブラウジングのためのオブジェクトを生成

プログラムのトップレベルに現われる文は，(FlyLine の数)+1 回実行される．最初の実行は，トップレベルで実行時エラーが発生しないかどうかチェックするために行われる．

7.4.1.2 それ以外の命令

ソースコードの変更以外にサーバが受け取る命令は，例えば，5.5.1 節で述べた Dive がある．Dive は，FlyLine を選択した後に，関数呼び出しを選択し，選択した呼び出しを辿っていくものだった．

Dive は，Kasumi 側から現在のカーソル位置が送られ，以下のようにサーバ内で処理される．

(1) 現在 FlyLine を選択済であるか調べる
(2) 構文木を探索し，もっともらしい関数呼び出しを見つける
(3) 見つけた関数呼び出しを現在のトレース位置に設定
(4) トレース内の変数や関数呼び出しを探す
(5) 見つけたものについて，バージョンを使って復元した返り値，位置を返す

7.4.2 Kasumi モード

Kasumi は，バッファを after-change-functions[*2] を使って監視している．しかし，after-change-functions は，FlyLine のような Shiranui のサーバからの書き換えに対しても実行されてしまう．よって，単純に after-change-functions に追加した関数で Shiranui のサーバに実行命令を出すのでは無限ループに陥ってしまう．

そこで，Shiranui では，FlyLine のようなサーバからの書き換えに対して，inhibit-modification-hooks を有効にして after-change-functions を抑制する．しかし，IdleFlyLine や FlyMark 等の，バッファ内の文字数が変化する変更は，Shiranui のサーバ内にある構文木のポイント [*3] と，バッファ内のポイントがずれることを引き起す．

ポイントがずれる例として，以下のような場合を考える．

```
1 #+ a -> 42;
2 #+ a -> 42;
3
4 let a = 42;
```

次に，4 行目の `let a = 42;` を `let a = 423;` に変更したとする．すると，Kasumi はその変更をサーバに送信し，サーバは，1 行目と 2 行目の IdleFlyLine の右辺を `423` に

[*2] https://www.gnu.org/software/emacs/manual/html_node/elisp/Change-Hooks.html
[*3] ポイントとは，Emacs Lisp の用語で，ソースコードの先頭から数えた文字のインデックスを指す．https://www.gnu.org/software/emacs/manual/html_node/elisp/Point.html

変更する命令を送信する．

IdleFlyLine の結果は，“どこに (where)”，“何文字削除して (remove_length)” ，“何を挿入するか (value)” の三つ組で表わされる．

1 行目と 2 行目のどちらの IdleFlyLine の結果が先かはわからないが，ここでは 1 行目，2 行目の順に来ると仮定する．1 行目の 42 は 8 文字目から始まり，2 行目の 42 は，19 文字目から始まる．また，42 の長さは 2 であるから，サーバからは，(where=8, remove_length=2, value=“423”),(where=19, remove_length=2, value=“423”) という順で送信される．

すると，ずれの補正をしなかった場合には，図 44 のようにソースコードが変更される．このように期待しない変更がされてしまうのは，1 行目の文字数が 1 増え，2 行目の変更の “どこに (where)” が 1 ずれた位置になってしまったことが原因である．

```
1 #+ a -> 42;
2 #+ a -> 42;
3
4 let a = 423;
```

ユーザのキー入力により，4 行目を変更した

```
1 #+ a -> 423;
2 #+ a -> 42;
3
4 let a = 423;
```

1 行目の IdleFlyLine の変更 (8,2,“423”) を受信

```
1 #+ a -> 423;
2 #+ a ->4232;
3
4 let a = 423;
```

2 行目の IdleFlyLine の変更 (19,2,“423”) を受信

図 44: ポイントのずれを補正しなかった場合，期待しない形にソースコードが変更されてしまう

7.4.2.1 変更の一時保存

inhibit-modification-hooks を有効にして，after-change-functions を抑制しても，ユーザによるバッファ操作があった場合には，抑制していた変更もサーバに送る必要がある．そこで，抑制した変更は，一時的に保存しておき，ユーザによるキー入力があったときにまとめておくるようにする．

例として，IdleFlyLine の送ってきた命令を処理する関数をソースコード 14 に示す．

kasumi-fix-point は，後述するサーバからソースコードへのずれ補正関数である

ソースコード 14: IdleFlyLine の変更を受け取る関数．after-change-functions を抑制，ずれの表を持つ

```
(defun kasumi-receive-idleflyline (value)
  (let* ((lines         (...))
         (target        (...))
         (where         (...))
         (remove_length (...))
         (value         (...)))
         ;; after-change-functions を呼ばない
         (inhibit-modification-hooks t))
    (save-excursion
      (progn
        ;; 右辺の先頭に
        (goto-char (kasumi-fix-point where))
        ;; 右辺の先頭から最後までを消す
        (delete-region (kasumi-fix-point where)
                       (+ (kasumi-fix-point where)
                          remove_length))
        ;; 変更を挿入
        (insert value)
        ;; 変更履歴を保存しておく
        (add-change (kasumi-fix-point where)
                    remove_length value)
        ;; "ずれ"を計算するための表を持つ
        (kasumi-add-diff (kasumi-fix-point where)
                 (- (length value) remove_length))

        ;; 青の線を引く
        (kasumi-put-idleflyline (...) (...))
```

7.4.2.2 ずれの補正

ソースコード上のポイント，例えばカーソル位置からサーバの構文木上の位置を計算するには，7.4.2.1 節の一時保存したデータを使用する．本章のはじめに使用したカーソルがずれる FlyLine の例では，(where=8, remove_length=2, value=“423”) というデータを受け取った時点の補正方法は表 1 となる．

サーバの位置 10 に対応するソースコード位置が二つ存在している．対応するソースコード位置がそもそも存在しないものがあるからである．よって，存在しないソースコード上のポイントを参照すると，ずれが発生してしまう．

複数の抑制した変更があっても表を複数利用することで解決できる．

しかし，FlyMark を利用したジャンプのようにサーバ上に存在しないデータを利用したいときには，FlyMark の右辺のインデックスを使う等のポイントを使わない方法で参照する必要がある．

8. 関連研究

YinYang [7] は，本研究と同様に，ライブプログラミン

位置 (サーバ)	文字 (サーバ)	位置 (コード上)	文字 (コード上)
8	‘4’	8	‘4’
9	‘2’	9	‘2’
10	‘;’	10	‘3’
10	‘;’	11	‘;’
11	‘改行’	12	‘改行’

表 1: (where=8,remove_length=2,value=“423”) を受け取った時のカーソルずれ補正

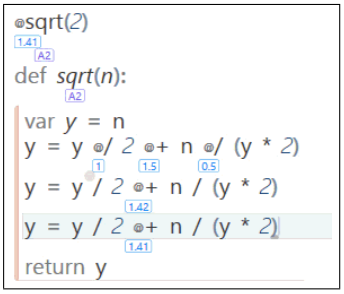


図 45: YinYang の probing (http://research.microsoft.com/en-us/people/smcdirm/liveprogramming.aspx)

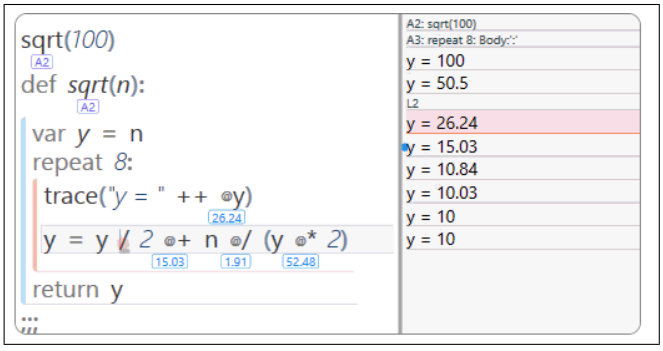


図 46: YinYang の Tracing (http://research.microsoft.com/en-us/people/smcdirm/liveprogramming.aspx)

グを実用的なものにする研究である．YinYang は，独自の言語と，開発環境を使用している．開発環境には，tracing (図 46) と probe (図 45) という機能を備えている．tracing は，Shiranui における 5.4 節の FlyMark と同等の printf デバッグを即座に反映にする仕組みで，同じように関数呼び出しの単位のトレースのジャンプにも使える．しかし，FlyLine のように関数呼び出しごとにログを分離することができない，どこで表示されたのかがすぐにわからないという欠点がある．長所として，右側のペインに一覧として表示されるために大域的なデバッグが行いやすいことがある．probe は，Shiranui における 5.5 節のトレースブラウジングおよび，変数の値表示と同等の機能である．しかし，テストを作成しようとした場合には手動で引数全てをコピーペーストして，関数呼び出しの式の組み立てる必要がある．また，筆者の知る限り，Shiranui が可能な関数オブジェクトや循環データの扱いについては考えられていない．

Apple Swift [8] は，実験のための環境として，Playground というライブプログラミング環境を備えている．Playground は，すべての文について評価した結果を別のペインに表示する．しかし，FlyLine にあたるようなテストの仕組みは持っていない．また，評価した結果はすべてに順序が付いているが，関数呼び出し単位等では表示することはできない．第一級オブジェクトとしての関数を言語でサポートしているが，復元可能な形で表示することはできない．よって，関数オブジェクトを実行時情報から取り出しテストに使う等のことができない．

Theseus [6] は，JavaScript や HTML 用の IDE である Brackets の拡張である．Theseus は，JavaScript の関数ごとに呼び出された回数や，引数，返り値を記録していく．また，呼び出しのコールスタックも記録することができる．これらは，JavaScript の動的な操作に対しては有効である．また，JavaScript の変更可能データについても深さを決めてコピーを行っている．これは，Shiranui 言語の採用しているバージョンよりもメモリ効率が悪く，また深さを決めている為，決まった深さを越えた場合にはまちがったデータを参照してしまう．また，テストに対してもサポートは筆者の知る限りはない．

xmpfilter [12] は，ruby のライブラリ rcodetools の一部である．xmpfilter は，Emacs や Vim のプラグインを使い，IdleFlyLine および FlyMark と似たインターフェイスを実現することができる．しかし，xmpfilter は即座に実行されるわけではなく，ユーザによる明示的な実行を要求する．TestFlyLine にあたるものは存在しない．また，FlyMark に当たるものも独立していない．また，トレースブラウジングのように，実行時情報をインタラクティブに表示する機能はない．

REX [4] は，回帰テストを，REPL の履歴をクラスタリングし自動生成するというものである．これは，Shiranui における，5.7.1 節の IdleFlyLine から TestFlyLine への変換に相当する．しかし，Shiranui では，IdleFlyLine は，自動的に再実行され，プログラムの挙動が正しいとユーザが確信したときに変換する．REX では，プログラムが正しくないとユーザが確信したときにもテストが生成されてしまう．また，5.1.1 節で述べたように，REPL は関数の再定義に関して，フィードバックを提供しない．筆者の知る限り，REX も同様に，テストは生成するものの，フィードバックを即座に提供することはしていない．

Soares らによる，Java のプログラムに対し，自動的に回帰テストを作成し，現在のプログラムと比較する研究 [14] がある．Soares らによる研究は，テストを作成することのみが目的ではなく，ユーザに振舞が変わるということを提示するのが目的である．Shiranui のように，ユーザが直接テストを書くことは想定していない．

9. 結論と今後の課題

9.1 結論

ライブプログラミングにおける実験と，伝統的なユニットテストの類似性に着目し，それらを統合するインターフェイスを設計，実現した．

それにより，ライブプログラミングの実験，伝統的なテストによるプログラムの正しさの自動検証，実行時情報からのテスト作成を同時に達成した．

以下に具体的にどのようなインターフェイスでライブプログラミングにテストを統合したかを述べる．

実験とテストを統合したインターフェイスである FlyLine(5.1 節) は，実験を表わす IdleFlyLine から，ユニットテストである TestFlyLine に簡単に変換できるようになっている．これにより，ユーザは一度目視で確認した実験をテストにして，正しさの判定を自動化することができる．

インラインに printf デバッグを可能にする FlyMark(5.4 節) は，FlyLine ごとに出力を記録することで，実験やテスト毎に printf デバッグが可能である．

実行時情報を実験やテストに変換する機能 (5.7 節) は，実験中に現われた複雑なデータ構造も簡単に実験やテストにすることができる．これにより，テストの作成にかかるコストを削減することができた．

実際に C++言語で記述したインタプリタと Emacs Lisp で記述した対話エディタを組み合わせたプログラミング環境を作成した．この処理系では， FlyLine ごとにスレッドを作成することにより， FlyLine を独立して実行し，すばやいフィードバックを可能にした．また，変更可能な値にバージョンをつけ，構文木の各ノードにプログラムの評価結果を保存していくことにより，実行結果の履歴表示や巻き戻しを可能にした．

9.2 今後の課題

以下では，本研究の今後の課題を述べる．

9.2.1 テストの最適化

現在の Shiranui は，コード書き換えの際に全ての FlyLine を再実行する．全てを再実行することは非常に時間がかかり，フィードバックを即座に提供することができなくなる．この問題に対し，書き換えの影響を受けるような FlyLine のみを実行したり，バグ発見に貢献している FlyLine を優先する等の改良が考えられる．

また，関連研究として，文献 [15] 等が挙げられる．

9.2.2 Shiranui 言語の型

Shiranui 言語は，静的型を持たない言語として設計されている．しかし，ライブプログラミングの実行結果の蓄積という利点を活かし，実行結果からの型推論，あるいは Gradual Typing [13] を導入することが考えられる．

9.2.3 複雑なデータ構造

現在，Shiranui は，データ構造としてリストをサポートしている．しかし，現実には，C 言語に見られる構造体や，Java 等のクラスといった複雑なデータ構造が必要となる．

それらの複雑なデータ構造に対しても，より良い可視化，ライブプログラミングのサポートを行うことが課題として挙げられる．

9.2.4 従来のライブプログラミングへの統合

Shiranui の仕組みは，これまで開発されてきたグラフィックスを描くようなライブプログラミング [2], [3], [11] にも入れることができる．これらの従来のライブプログラミングは，プログラムとアウトプットを一組しか持たない．よって，比較実験を行うことが難しい．例えば，絵を描くプログラムでは最適な線の太さを求める際に，変更前の線の太さを覚えておき比較する必要があった．この問題は，Shiranui の FlyLine のような仕組みを導入することにより解決できる．

また，マウスの動きを使うようなプログラムにおいては，マウスの動きを実行時の情報だと解釈し，取り出すことによって実験を自動化，容易にすることも考えられる．

9.2.5 既存言語に対する Shiranui の仕組みの導入

本研究では，Shiranui 環境を実現するために独自の言語である Shiranui 言語を作成した．しかし，独自の言語ではなく既存の言語に Shiranui の仕組みを導入することができれば，既存のプログラミング環境やライブラリを使用することができる．

現在，既存の言語に対する導入方法として二種類の手法を考えている．

まず，コード変換によりログ等を保存する方法である．この方法は Theseus [6] が採用している．しかしながら，この方法はコードそのものを書き換えてしまうため，既存のプログラミング環境やデバッガと相性が悪い．

次に，逆向き実行が可能なデバッガを使用して実現する方法である．そのようなデバッガには The GNU Debugger [9] や ocamldebug [5] 等がある．これらのデバッガはコードそのものを書き換えることなく逆向き実行が可能である．またそれ自体がブレイクポイントデバッガでもあるため，Shiranui の仕組みをさらに拡張し，ライブプログラミングとブレイクポイントデバッガの中間の環境が作成できると考えられる．

なお，ライブプログラミングとステップ実行デバッガの連携については，S. Burckhardt ら [2] によると，“Also, the code in event handlers and initialization bodies is not debuggable via live programming. Thus, a step-wise debugger is still useful and future work may look at how live programming and step-wise debugging can work together.” とある．

9.2.6 開発効率に関するユーザ実験

Shiranui が開発効率に貢献できるのかを実験するのも課題である．実験内容としては，ライブプログラミング機能を無効にした Shiranui と，有効にした Shiranui を使って，課題を解いてもらうことで比較することを考えている．課題としては，例えば二次方程式の解を導く等のものである．

参考文献

[1] Kent Beck and Erich Gamma. JUnit: A Cook's Tour. *Java Report*, 4(5):27–38, 1999.

[2] 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.

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

[4] Adrian Kuhn. On Extracting Unit Tests from Interactive Live Programming Sessions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1241–1244, Piscataway, NJ, USA, 2013. IEEE Press.

[5] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.02: Documentation and user's manual*. PhD thesis, Inria, 2014.

[6] 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.

[7] 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.

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

[9] Free Software Foundation Inc. GDB: The GNU Project Debugger. `http://www.gnu.org/software/gdb/`. Accessed 2015-2-25.

[10] Kodowa Inc. Light table. `http://lighttable.com/`. Accessed 2015-2-5.

[11] 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.

[12] rubikitch and Fernandez Mauricio. rcodetools — RubyGems.org — your community gem host. `https://rubygems.org/gems/rcodetools`. Accessed 2015-2-1.

[13] Jeremy G Siek and Walid Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[14] 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.

[15] Bastian Steinert, Michael Haupt, Robert Krahn, and Robert Hirschfeld. Continuous Selective Testing. In Alberto Sillitti, Angela Martin, Xiaofeng Wang, and Elizabeth Whitworth, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 48 of *Lecture Notes in Business Information Processing*, pages 132–146. Springer Berlin Heidelberg, 2010.