令和元年度 学士論文

デザインレシピに沿った
プログラミング環境に関する研究

東京工業大学 情報理工学院 数理・計算科学系
学籍番号 16B11711
能勢　純弥

指導教員
増原 英彦 教授

令和2年2月28日

概要

デザインレシピとは、プログラムを書くための一連の手順のことである。これは、プログラミングの初学者が与えられた問題にどうアプローチするべきか分からず、手が止まってしまう現象を解決するために提案された。具体的には、データ構造の分析やテストの作成などのステップからなる。本研究の目標は、デザインレシピのステップを忠実に踏ませるためのプログラミング環境を作成することである。その第一歩として、構造的再帰を用いた関数を定義するための環境を設計し、実装した。既存のプログラミング環境にない機能として、データ例やテストの網羅性を高めるための設問を設けたほか、関数定義の概形の自動生成を可能にしている。

謝辞

本研究を進めるにあたり、叢悠悠助教には、直接のミーティングや Slack 上でのやりとりなどで多くの時間を割いて、具体的な研究の進め方や論文の読み方および書き方などのたくさんのご指導をしていただきました。増原英彦教授には、進捗報告会やセミナーにおいて、ユーザの視点に立った際の使いやすさ、教育者の視点に立った際の教育効果のあるインターフェイス、ついてなどの様々な助言をしていただきました。伊澤侑祐さんをはじめ研究室の皆様には、自分の研究で忙しい中、実装に関してのアドバイスや本論文およびスライドの校正、セミナーでの様々な指摘、長時間に及んだ発表練習など、最後の最後までご指導をしていただきました。心より感謝いたします。

目次

図目次

第 1 章

はじめに

デザインレシピ [1] とは、プログラムを書くための一連の手順のことである。これは、プログラミングの初学者が与えられた問題にどうアプローチするべきか分からず、手が止まってしまう現象を解決するために Felleisen ら [1] によって提案された。デザインレシピは問題解決に必要なステップを詳細に教えてくれるため、初心者でもエディタが真っ白のまま止まってしまう、ということは少なくなる。

このデザインレシピの考えをプログラミングの授業に導入している大学は多数ある[*1]。例えば、 Felleisen は米国ライス大学およびノースイースタン大学において、デザインレシピの各ステップに沿ってインタラクティブに進めていく授業を行った。デザインレシピを学んだ学生は、これを学んでいない学生に比べて、良いプログラミングの習慣 (関数の目的を書く、テストを作成する、など) を持っていた、という報告もされている [2]。特にノースイースタン大学の方ではその後のオブジェクト指向のプログラミングの授業において、より多くの学生が A を獲得している。

大学の他にもデザインレシピは中高生向けのプログラミング教室 Bootstrap[*2]でも採用されている。Bootstrap では、WeScheme [3] という環境が使用されており、ユーザがブラウザ上で関数の型や入出力の例を入力できる (図 1.1)。これは、関数の型と入出力の例を書かないと、コーディングが始められない。

しかし、WeScheme はデザインレシピの全てのステップをサポートしているわけではなく、特にデータ構造の分析とそれによって関数の概形を得るステップが抜けている。これは、Bootstrap がユーザとして中高生を想定していて、彼らが書く関数も単純なもの（再帰を含まないもの）であることを仮定していることによる [4]。同じような動機から Ryu [5] は DRaCO というプログラミング環境 (図 1.2) を実装しているが、これも WeScheme のようにレシピのステップを部分的にサポートしたものとなっている。

そこで、本研究では、デザインレシピの全てのステップを忠実に踏ませるためのプログラミング環境を作ることを目標にする。対象言語は Scala とした。これは東京工業大学

[*1] https://github.com/racket/racket/wiki/Courses-using-Racket
[*2] https://www.bootstrapworld.org

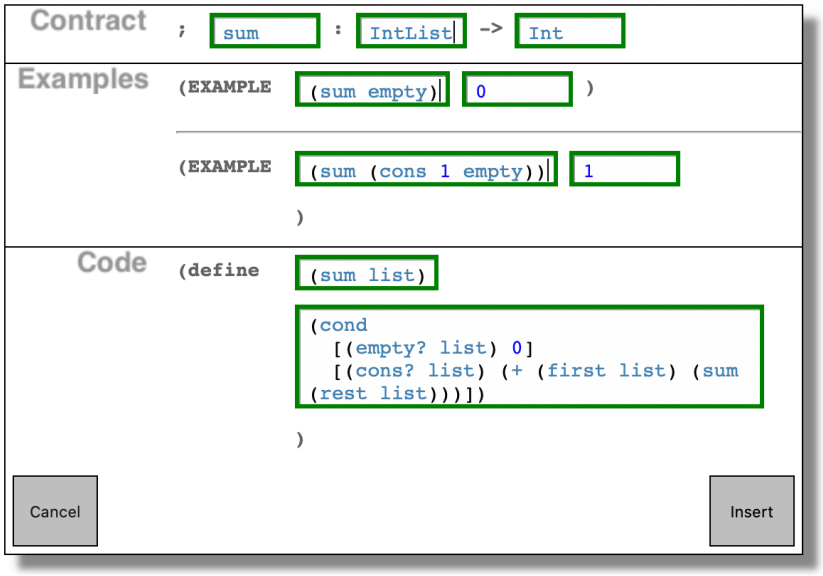

図 1.1　WeScheme の例

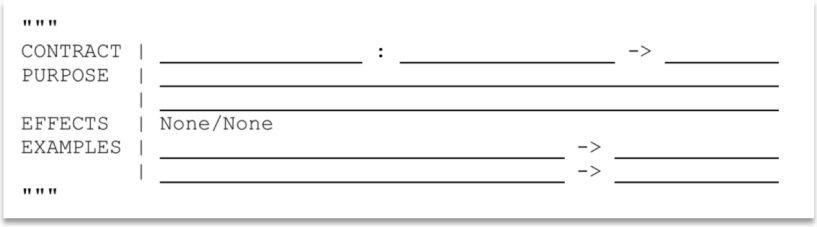

図 1.2　DRaCO の例

の学部生向け講義においてユーザー実験を行いたいためである。デザインレシピはデータ構造やアルゴリズムのパターンによって異なる定義を持つが、現在は最も基本的なものである構造的再帰のデザインレシピをサポートできている。

本論文の構成は以下の通りである。まず第 2 章で構造的再帰のデザインレシピを説明する。次に第 3 章で、環境がない状態でデザインレシピを学ぶ際の問題点と、それを解決するためのサポートを具体例を用いて説明する。第 4 章と第 5 章では今後行う予定である拡張として、より複雑なデザインレシピのサポートとフィードバック機能の導入について述べる。第 6 章で関連研究を紹介し、第 7 章でまとめる。

第 2 章

デザインレシピとは

デザインレシピとは、具体的には以下の 6 つのステップのことを言う。

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

データ構造の分析や目的文などはコメントとして、テストなどはコードとして書く。具体例を使って各ステップで行う作業を見てみよう。ここでは受け取った整数のリストの値の合計値を返す関数を考える。

■Step 1：**データ構造の分析**　まずは、入力として与えるデータについての考察を行う。今から定義しようとしている関数の引数は整数の入ったリストである。ここで整数のリスト (`List[Int]` 型) は以下の 2 種類のパターンがある。

```
// 空のリスト Nil，整数を 1 つ以上持つリスト x :: xs
```

ここで 2 つ目の整数を 1 つ以上持つリストでは、先頭要素を取り除いた残りも整数のリストになっている。つまり、再帰的な構造になっている。これを踏まえて、それぞれのパターンのデータ例を作成する。

```
// * 空のリストの例 Nil
// * 整数を 1 つ以上持つリストの例 1 :: Nil，1 :: 2 :: 3 :: Nil
```

■Step 2：**目的・入出力の型の決定**　次に、関数の目的、名前、入力と出力の型およびダミーの出力を記述する。この関数の目的は与えられた整数の全ての合計を求めることである。よって適切な名前として `sum` を与える。また、入力の型は `List[Int]` 、出力の型は `Int` である。ダミーの出力としては、例えば `0` を与えることができる。これらの

情報を使うと、正しく動作はしないが実行可能なダミーの関数定義を書くことができる。したがって、Step 2 が終了した段階でコードは以下のようになっている。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int =
  0
}
```

■Step 3：入出力の例 (テスト) の作成　次に、Step 1 で作成したデータ例を入力とするテストを作る。今、例は 3 つあるのでテストも 3 つできる。よって、Step 3 が終了した段階でコードは以下のようになっている。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int = {
  0
}

// テスト
val test1 = sum(Nil) == 0
val test2 = sum(1 :: Nil) == 1
val test3 = sum(1 :: 2 :: 3 ::Nil) == 6
```

■Step 4：テンプレートの作成　このステップでは、Step 1 で確認した入力データの構造をもとに、関数本体の大まかな形を定める。その際、以下の 3 つの原則に従う。

1. 入力のパターンに対応した match 文を挿入する。
2. それぞれのパターンの部分項は使うものと仮定する。
3. 再帰的な部分があるかを確認し、ある場合はその部分に再帰呼び出しを入れる。

こうして作られたものを**テンプレート**と呼ぶ。sum の場合、Step 1 で確認した通り入力のパターンは 2 パターンあるので、match 文は 2 つの case を持つ。さらには、空でない場合の時に先頭要素を取り除いた残りの部分が再帰的な構造を持つので、その部分には再帰呼び出しを入れる。したがって、sum のテンプレートは以下である。

```
def sum(list: List[Int]): Int = {
  list match {
    case Nil =>  ...
    case x :: xs => ... x ... sum(xs) ...
  }
}
```

ここで、テンプレートの段階では「どのようなデータを処理するのか」にのみ着目し、「どのように答えを求めるのか」を考えないことに注意しよう。sum の場合、リストが空であるケースは明らかに 0 を返したいが、ここでは '...' としている。このように、関数の構造と具体的な計算を切り分けることによって、初学者がやるべきことが明確になる。よって Step 4 が終了した段階でコードは以下のようになっている

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int = {
  list match {
    case Nil =>  ...
    case x :: xs => ... x ... sum(xs) ...
  }
}

// テスト
val test1 = sum(Nil) == 0
val test2 = sum(1 :: Nil) == 1
val test3 = sum(1 :: 2 :: 3 ::Nil) == 6
```

■Step 5：コーディング　そして、Step 4 で定めたテンプレートの … を具体的なコードに置き換えていくことで、プログラムを完成させる。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int = {
  list match {
    case Nil =>  0
    case x :: xs => x + sum(xs)
  }
}

// テスト
val test1 = sum(Nil) == 0
val test2 = sum(1 :: Nil) == 1
val test3 = sum(1 :: 2 :: 3 ::Nil) == 6
```

■Step 6：テスト　最後に、Step 3 で作成したテストを実行することで、プログラムの動作を確認する。まず、エラーが起きた場合は、エラーの種類に応じてプログラムを見直す。

1. コンパイルエラー：型の不一致によるもの。エラー箇所が関数定義内にある場合は Step5 に戻りコードを見直す。テストでエラーが起きている場合は入力あるいは出力の型を見直す。
2. 実行時エラー：場合分けの不足による場合や、テストの入力が不適切な場合がある。

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

第 3 章

デザインレシピに沿ったプログラミングのための環境の提案

3.1　デザインレシピを用いた学習の課題

Ramsey [6] は米国タフツ大学においてデザインレシピに沿ったプログラミングを実際に教え、そのなかでの学生の様子を調べた。彼は、学生がデザインレシピに従うなかで起こしやすい問題や間違いとして、以下のような点をあげている。

- Step 1 におけるデータ例の書き忘れ
- Step 2 における構文的に正しくない型表現
- Step 3 における網羅的でないテストの記述
- Step 4 における正しくないテンプレートの記述

それぞれについて、第 2 章で用いた `sum` 関数をもう一度例に説明し、それによって起こる影響を述べる。

■Step 1 におけるデータ例の書き忘れ　Sgtep 1 では、`Nil` の例として `Nil` を、`x :: xs` の例として `1 :: Nil` や `1 :: 2 :: 3 :: Nil` などを書かなくてはいけない。これらのデータの例を書き忘れてしまうと、Step 3 におけるテストの網羅性が低くなる。具体例を書かずに Step 3 に進んでしまうと、いくつかのパターンに対してのテストを作成し忘れてしまうためである。

■Step 2 における構文的に正しくない型表現　Step 2 では入出力の型を書かなくてはいけないが、型表現を構文的に間違えてしまうと、安全なプログラムは書けない。構文的に間違えた型表現というのは、例えば以下のように整数リストの型として `List` と書いてしまうことなどである。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List): Int =
```

```
  0
}
```

■Step 3 における網羅的でないテストの記述　Step 3 ではテストを記述する。以下は網羅性の低いテストの例である。

```
// テスト
val test1 = sum(1 :: Nil) == 1
val test2 = sum(1 :: 2 :: 3 ::Nil) == 6
```

この例のように Nil に対するテストを作成し忘れてしまい、空でないリストに対するテストのみしか書かれていないと、Nil に対するプログラムの動きが間違えていたとしても気づけない。

■Step 4 における正しくないテンプレートの記述　Step 4 ではテンプレートを記述する。以下は正しくないテンプレートの例である。この例では Nil に対する場合分けを書き忘れている。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int = {
  list match {
    case x :: xs => ... x ... sum(xs) ...
  }
}
```

テンプレートが正しくないまま Step 5 のコーディングに進むと、以下のようなコードを書いてしまう。

```
// 目的：整数リストの要素の合計を求める
def sum(list: List[Int]): Int = {
  list match {
    case x :: xs => x + sum(xs)
  }
}
```

しかし、これが Step 6 においてテストが通らなかったとき、学生は x + sum(xs) という具体的な計算部分が間違えていると思い、場合分けが不足しているという関数の構造におけるミスには気がつかない。

3.2　課題解決のための方法

3.1 節で述べた問題が起こる原因として、デザインレシピのステップに従わせることを、学生の自主性に任せて行ったことが考えられる。各ステップにおいて作業を省略してしまうことや、間違ったまま次のステップに進んでしまうことは、正しくないプログラムを書くことにつながる。

そこで、本研究では、レシピの全てのステップにおいて誘導を行う環境を設計し、実装する。この環境は、3.1 節で述べた問題を解決するだけでなく、各ステップにおいて具体的に何をしたら良いのか分からない学生の助けにもなる。

3.3　各ステップにおけるサポート内容

本研究の環境での、各ステップにおけるサポート内容を、第 2 章で用いた `sum` 関数を例に説明していく。

■Step 1　ここでの入力例は図 3.1 のようになる。まず、注目したいデータの型を入力し、そのデータのパターンを Scala の match 文に当てはまる形で書かせる。その後に、それぞれのパターンにおける具体例を書かせる。`sum` 関数において、注目したいデータは整数リストである。そして整数リストのパターンには `Nil` と `x :: xs` の 2 パターンがある。本研究では、これらを再帰的でないパターンと再帰的なパターンを分けて書かせることによって、ベースケースの書き忘れなどの単純なミスを回避する。さらに注意したいのは、再帰的でないパターンから入力させる、ということである。なぜなら、これらのパターンを考慮しないとデータの具体例を作ることができない上、関数も停止しなくなってしまうからである。また、ここで、再帰的なパターンにおいて、どの部分が再帰部分なのかを明示させる。`x :: xs` の場合、再帰部分は `xs` である。

図 3.1　sum 関数を定義する際の Step1 の入力例

■Step 2　ここでの入力例は図 3.2 のようになる。このステップでは、関数の目的、名前、入出力の型を書かせる。この際、どの引数に対して場合分けが行われるのか、ということを確認させる。また、自動的に構文の正しさをチェックする。本環境では、実行を介さずに構文チェックを行うため、ダミーの出力例は不要となる。

図 3.2　sum 関数を定義する際の Step2 の入力例

■Step 3　ここでの入力例は図 3.3 のようになる。このステップでは、テストを作成させる。そのために、入力例、出力例を書かせるためのテキストボックスを表示する。ここで、Step 2 で場合分けを行うと指定された引数に関しては、Step 1 でのデータ例が入力例としてすでに表示された状態となる。Step 1 でパターンごとに書いたデータ例を用いることによって、テストの網羅性を高めている。なお、ユーザによるテストの追加と削除も可能である。

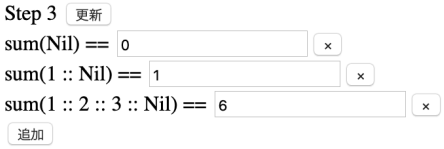

図 3.3　sum 関数を定義する際の Step3 の入力例

■Step 4　ここでの入力例は図 3.4 のようになる。このステップでは、テンプレートを作成させる。その際、テンプレートの正しさを判断する。正しければ Step 5 に進めるようになり、正しくなければどの部分が間違っているのかを表示する。例えば、パターンマッチのケースが不足していることや、使う式として書かれた式が不十分であるということを指摘する。この判断を行うために、環境側で正しいテンプレートの自動生成を行う。具体的には、Step 2 で場合分けを行うと宣言された引数に対してパターンマッチを挿入する。その場合分けは、Step 1 で与えられたパターンの情報に基づいて行う。各場合分けに、パターン変数と再帰呼び出しの結果を使う式として記述する。ここで、再帰呼び出しの引数は Step 1 で指定された再帰部分となる。

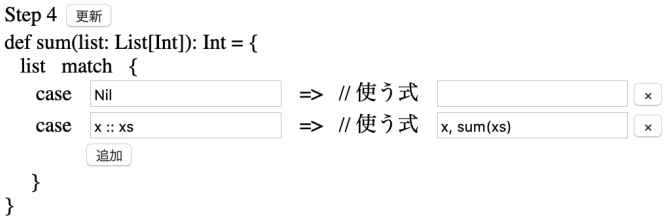

図 3.4　sum 関数を定義する際の Step4 の入力例

■Step 5, Step 6　ここでの入力例は図 3.5 のようになる。ユーザのテンプレートが正しいと判断されたら、その穴を埋めさせる。その結果を、コードとして表示すると図 3.6 のようになる。そして、コーディングが終わった後は、テストを実行する (図 3.7)。

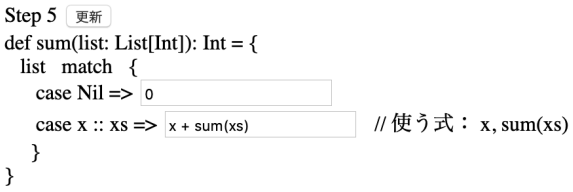

図 3.5　sum 関数を定義する際の Step5 の入力例

```
1  // 目的：整数リストの要素の合計を求める
2  def sum(list: List[Int]): Int = {
3     list match {
4          case Nil => 0
5          case x :: xs => x + sum(xs)  // 使う式: x, sum(xs)
6      }
7  }
8
9  // テスト
10 val test1 = sum(Nil) == 0
11 val test2 = sum(1 :: Nil) == 1
12 val test3 = sum(1 :: 2 :: 3 :: Nil) == 6
```

図 3.6　sum 関数を定義した際に出力されるコード

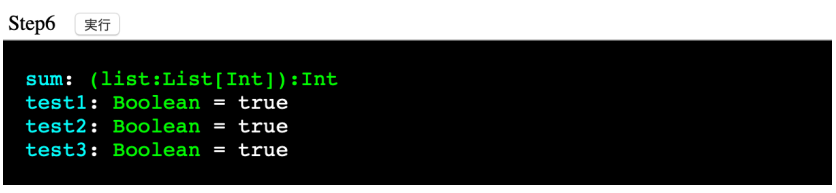

図 3.7　sum 関数を定義した際の Step6 での表示例

3.4　未実装の機能

現在、上にあげた機能の多くはサポートできているが、以下の 4 つは未実装である。

- ユーザが定義した型の情報の取得 (Step 1)
  ユーザによって定義されたデータ型の使用を可能とするため、型定義を Scala ファイルから読み込む。これがあると、Step 6 でのテストの実行が可能となる。
- 構文チェック (Step 2)
  ブラウザ上で動作する既存のエディタに、拡張機能として実装する。たとえば、ace.js[*1] は Javascript などの言語に対する構文チェック機能を備えており、それを Scala 用に書き換えることでサポートできると考えている。
- テンプレートの正誤チェック (Step 4)
  ユーザのテンプレートと環境側で自動生成されたものに対して、全ての場合分けにおけるパターンと使用する式を比較し、正誤を判断する。基本は完全に一致することを要求する。ただし、3.5 節で述べる `zip` 関数など、一方のみが空である場合分けなどを含む関数に関しては、判断基準を緩める必要があるのではないかと考えている。
- テストの実行 (Step 6)
  構文チェックと同様に、既存のエディタの拡張を行うことによって、本環境の中で Scala プログラムを実行できるようにする。具体的には、JavaScript で実装した Scala のインタプリタを用いる。具体的には、Scala のサブセット（たとえば、東京工業大学の講義で扱う構文）に対するインタプリタを JavaScript で実装する。

3.5　sum 関数以外の対応可能な関数

上記で用いた `sum` 関数は、単純な構造的再帰の中でも簡単な例である。その他にも以下のような関数に対応できることを確かめた。

- 複数の引数を持つ関数
- 複数の再帰部分を持つデータ構造を扱う関数
- 同じデータ構造に対して同時再帰が起きる関数

■複数の引数を持つ関数　たとえば、整数リストの各要素に整数 n を足す関数 `addN` 関数（図 3.8）が挙げられる。この関数では、リストと整数を受け取るが、関数を定義する上で重要になるのはリストの方のみである。よって、Step 1 のデータ構造の分析はリストに対してのみ行う。Step 2 で、リストの型をした引数に対して場合分けを行うと確認

[*1] https://ace.c9.io

する。Step3 では、Step1 で分析したデータ構造以外の構造を持つ入力例はユーザが自分で打つ。Step4 での入力は、sum 関数の時とほとんど同じであるが、使う式の表示が少し異なることに注意する。

図 3.8　addN 関数を定義する際の Step 2, 3, 4 の入力例

■複数の再帰部分を持つデータ構造を扱う関数　たとえば、二分木の各要素の合計を求める関数 sumTree 関数（図 3.9）が挙げられる。二分木では、左の木と右の木の 2 箇所が再帰部分となっているため、Step 1 ではこれらを共に再帰部分として指定する。これに対応して、Step 4 では使う式として、2 つの再帰呼び出しを書く。

図 3.9　sumTree 関数を定義する際の Step 1, 4 の入力例

■同じデータ構造に対して同時再帰が起きる関数　たとえば、2 つの整数リストの各要素のペアを要素として持つリストを求める関数 `zip` 関数（図 3.10）が挙げられる。この関数は 2 つの引数に対して同時に再帰するため、まず Step 2 で 2 つの引数に対して場合分けを行うことを指定する。すると、Step 1 でのデータ例の全ての組み合わせを入力として持つテストが得られる。ユーザは必要に応じてこれらを削除できる。また、Step 4 でのパターンマッチの場合分けは 4 種類になる。ここで、使う式は、空でないリスト同士のときのみパターン変数と再帰呼び出しの結果となる。しかし、一方のみが空でないリストの場合において、ユーザが `x` といった式を書いた場合に、それの正誤をどう判断するかは検討すべき課題である。

図 3.10　zip 関数を定義する際の Step 2, 3, 4 の入力例

第 4 章

展望 1 ：さまざまなデザインレシピへの対応

今の環境では単純な構造的再帰のデザインレシピが部分的にサポートできている。しかし、これとは違うデザインレシピを要求するプログラミングのパターンもさまざまある。本章では、今後対応したいレシピを 3 つ取り上げ、それぞれについて期待される動作と実装の方針を説明する。

4.1 ネストしたデータ構造

ネストしたデータ構造というのは、整数のリストを要素とするリストなど、複数のデータ構造が入れ子になったものを指す。具体的な例として、整数リストのリストが与えられた時に、その全ての合計を求める関数 sumNest を定義する。整数リストのリストを引数とする sumNest の関数定義の中に、整数リストを引数とする補助関数 sumNestAux を定義する必要がある。このように、ネストしたデータ構造は、入れ子のデータ定義の数だけ補助関数が必要となる。各補助関数の本体には、その引数のデータ構造に従ったテンプレートが挿入される。さらに、メインの関数の本体には、補助関数の呼び出しが挿入される。よって、期待される sumNest 関数のテンプレートは以下のようになる。

```
def sumNest(lList: List[List[Int]]): Int = {
  def sumNestAux(iList: List[Int] ): Int ={
    iList match {
      case Nil => ...
      case x :: xs => ... x ... sumNestAux(xs)...
    }
  }
  lList match {
    case Nil => ...
```

```
    case l :: ls => ...sumNestAux(l)...sumNest(ls)...
  }
}
```

これをサポートするためには、Step 1 でのデータ構造の分析の際の入力フォームとしてネストしたデータ構造であるとユーザーが判断した場合、入れ子となっているデータ構造についても分析できるように実装していく。

なお、ネストしたデータ構造を受け取る関数は必ずしも補助関数を必要としてるとは限らない。例えば、ネストしたリストの長さ (内側のリストの数) を求める関数の場合、内側のリストを処理する必要がない。これは、入力をネストしたリストではなく、任意の型の要素を持つリストとして扱うべきだったことを意味する。つまりここでは構造的再帰のデザインレシピを選択するべきだったのだ。ユーザーがこのような間違いに気づくのは Step 5 であると考えられる。この際に、補助関数の削除や、場合によってはパターン変数の名前の付け替えが必要となるが、前のステップに戻る必要はないと予想される。

4.2　一般的な再帰

第 3 章で紹介した構造的再帰のテンプレートでは、コンストラクタの数と場合分けの数が一致しており、データの再帰的な部分と再帰呼び出しの挿入位置も一致していた。しかし、これは分割統治法などを用いたプログラムの場合では成り立たない。

一般的な再帰を必要とするプログラムは自明に解決されるケースと、そうでないケースに分けられる。後者では、部分問題を解くために再帰が必要となる。たとえば、`quickSort` 関数の場合、自明なケースは空のリストと、要素が 1 つのリストの 2 つのケースがある。非自明なケースでは、基準値より小さいもの、および大きいものを集めたリストの並べ替えという二つの部分問題が生成される。よって、期待されるテンプレートは以下のようになる。

```
def quickSort(list: List[Int]): List[Int] = {
  list match {
    case Nil =>  ...   // 自明なケース
    case x :: Nil =>  ...   // 自明なケース
    case _ =>  ...quickSort(_)...quickSort(_)...  // 非自明なケース
  }
}
```

ユーザーは、Step 1 で非再帰的/再帰的なパターンではなく、非自明/自明なパターンを与える必要がある。また、Step 4 の段階で部分問題の数を与える必要がある。

4.3　アキュムレータを使用する関数

再帰関数を定義する際には、アキュムレータを使う場合も多い。典型的な例として、リストを逆順に並べ換える reverse を考えよう。アキュムレータを使用する際は、この余計な引数を受け取る補助関数を定義する。メインの関数から補助関数を呼び出すときには、アキュムレータの初期値として Nil を渡す。よって、期待されるテンプレートは以下のようになる。

```
def reverse(list: List[Int]): List[Int] = {
  def reverseAcc(list: List[Int], acc: List[Int]): List[Int] ={
    list match {
      case Nil => ...
      case x :: xs => ...x...reverseAcc(xs, _)...
    }
  }
  reverseAcc(list, Nil)
}
```

これをサポートするためには、まずユーザーがアキュムレータを使用したい意図を伝える機能が必要である。さらにはそのアキュムレータの初期値を入力できるフォームを実装する必要がある。

第 5 章

展望 2：フィードバック機能の導入

プログラムが未完成の状態におけるフィードバック、すなわちそのプログラムが正しい方向で書かれているかのチェックはユーザにとって有益なものになる。例えば、Step 1 や Step 3 でのパターンの十分性の判断や Step 2 での構文チェック、Step 4 でのテンプレートの正誤チェックなどである。この章では、Step 5 でのコーディングの際のフィードバックについて述べる。Step 4 まで、システムに沿って入力することで構造的にミスのないテンプレートが得られる。その先はユーザーが使う式に従って関数定義を完成させていく。ここで本研究ではそのアプローチとしてプログラム合成を利用したフィードバックの生成を検討している。

以下、プログラム合成について簡単に説明したあと、参考にする予定の研究について述べ、最後に具体的な導入の仕方について述べる。

5.1 プログラム合成とは

プログラム合成とは、ユーザーが与えた仕様をもとにプログラムを自動的に生成する技術である。生成するアプローチとして、以下のようなものがある。

- Barliman [7]
  Programming by Example の考え方に従って、ユーザーから入出力例を受け取ると、それを満たすような関数定義を合成する。
- λ_2 [8]
  Bariliman と同様にユーザーから入出力例を受け取り、`map` や `filter` などの高階関数を使って、それに渡す関数を合成する。
- Rosette [9]
  論理式の形で表現された関数の仕様を受け取って、SAT ソルバーや MAT ソルバーを使って、仕様を満たすプログラムを生成する。
- Synquid [10]
  篩型 (refinement type) を使って表現された仕様を受け取り、その型を持つ関数を

合成する。

- Sketch [11]
  入出力例と穴の開いたプログラムを受け取り、入出力例を満たすように穴を埋めていく。

5.2　プログラム合成によるフィードバックの生成

5.2.1　先行研究

Feldman ら [12] は、プログラム合成によるフィードバック機能を備えた Racket プログラミング環境を実装している。彼らの環境において、ユーザは関数の入出力例を仕様として与え、コーディングを行う。その途中で、“Am I on the right track?”（以下、AIORT）というボタンを押すと、自分が正しい方向に進んでいるかどうかについて、フィードバックを得ることができる。フィードバックの生成には、Racket のプログラム合成器 Barliman [7] を用いる。AIORT ボタンが押されると、Barliman はユーザが与えた入出力例をすべて満たすように、未完成のプログラムの「穴」を埋めようと試みる。これが成功すれば、“Yes”、失敗すれば “No”、一定時間内に解が見つからなければ “Maybe” というフィードバックを返す。

Feldman ら提案手法の評価を行うために数人を対象にしてユーザー実験を行っている。その結果、多くのケースにおいて AIORT のフィードバックは正しかった。一方で、システムから “No” が返ってきたときになぜ “No” なのかが分からないというユーザーの不満や、複雑なプログラムになるとタイムアウトになってしまうという問題があった。

5.2.2　本研究への導入

本研究ではデザインレシピの Step 5 (コーディング) を補助するために上記の AIORT フィードバックを導入したい。Feldman らは Racket を対象にしているが、本研究では Scala を対象としている。そのため、AIORT システムの導入には以下の 2 つのアプローチがある。

1. Scala の合成器 [13] を使い AIORT システムを作る
   Scala を対象とした既存の合成器は、合成されたプログラムをユーザに見せ、それに対するユーザのフィードバックをもとにプログラムを期待された形に書き換えていくという、インタラクティブな使い方を想定している。本研究では合成結果をユーザに提示しないため、このシステムを有効活用できるかは定かでない。
2. Scala のコードを Racket に変換して Feldman らの AIORT システムを使う
   Racket と異なり Scala はオブジェクト指向的にもプログラムを書くことができる言語である。しかし、本研究では関数型の考え方に従って書かれたプログラムのみを対象としているため、変換はそれほど難しくないと考えられる。

なお、本研究のプログラミング環境を用いると、Feldman らのユーザー実験で出た 2 つの課題を部分的に解決できる可能性がある。まず、システムから返された “No”の理由が分からないという問題について、Feldman らの例を用いて考えてみよう。

```
def replace(list: List[Int]): List[Int] = {
  list match {
    case x :: xs => 0 :: replace(xs)
  }
}
```

`replace` 関数の目的は受け取った整数リストの中の要素を全て 0 に置き換えることである。上の定義は、ベースケースが抜けているという点に対して間違っている。一方でリストが空でないケースは正しく書かれており、実際このプログラムを書いたユーザーは自分の関数定義が正しいと考えている。

本研究の環境を使う場合は、データ構造の分析やテンプレートを経由する。それによって、ベースケースの書き忘れが起きにくい。そのうえ、正しいテンプレートをもとに、Step 5 でコーディングをしているので、システムから “No” が返される時は、具体的な計算の部分が間違っている時である。そのため、システムが返すフィードバックと、ユーザーの認識のギャップは小さくなる。

次に、Feldman らの 2 つ目の問題であるタイムアウトの発生について考える。デザインレシピの各ステップを経由することは、これの解決策にもなり得る。例えば、ベースケースに対する出力は、プログラム合成の効率を大きく左右することが知られている[*1]。また、再帰呼び出しの場所が分かっているということもプログラム合成に役立つことが容易に想像できる。

[*1] Will Byrd らとの議論（2019 年 9 月 13 日）による。

第 6 章

関連研究

6.1　プログラミングのステップ細分化

Glaser ら [14] はデザインレシピに似た概念として、Programming by Numbers を提案した。これは、以下の 7 つのステップに沿ってプログラミングをしていくというものである。

1. 関数の名前をつける
2. 入出力の型の決定
3. 場合分けを明示する
4. 単純なケースに対する計算を書く
5. 複雑なケースで使える要素を列挙する
6. 複雑なケースに対する計算を書く
7. 見直しをする

デザインレシピと比較すると、1, 2 が Step 2 に、3, 5 が Step 4 に、4, 6 が Step 5 に、7 が Step 6 に対応している。これを実際の授業で用いたところ、生徒の反応としては、小さなタスクに分けられたことで白紙のまま手が止まってしまうということがなくなった、という声があった。

Minjie ら [15] は、ゴールとプランという概念に基づいたブロックプログラミング環境を提案した。ゴールというのは問題を解決するために達成すべき目標のことである。平均を求める問題の場合、ゴールは「合計を求める」「要素の数を求める」「前者を後者で割る」の 3 つのタスクからなる。プランというのは 1 つ 1 つの目標を実現するための具体的な操作のことであり、予めプログラム環境側で提供されているものである。例えば、「合計を求める」という目標に対しては `sum` 関数のブロックが対応する。Minjie らはこのゴールとプランの概念に基づいた手順として以下の 5 つのステップを提示している。

1. ゴールの分析をする
2. プランを選択し、それらの関係性を考察する

3. プランを展開する
4. プランを結合する
5. 見直しをする

このプログラムのスタイルは、大まかな計画から徐々に詳細化していく、という点においてデザインレシピの考えと類似している。一方で、Minjie らの研究は手続き型プログラミングの習得を目指している。データ構造の中心軸とするデザインレシピに比べると、一連の問題解決の流れを中心軸としている。

6.2　デザインパターン

デザインパターンとは、オブジェクト指向言語において、プログラムの再利用性を高めるための概念である [16]。これは、与えられた問題に適したプログラムの設計方法を示すものであり、具体的にはクラスやオブジェクト間の関係を整理したものである。デザインレシピの提唱者による教科書 How to Design Classes [17] では、デザインパターンをオブジェクト指向言語用のデザインレシピとして用いている。例えば、デザインレシピのデータ分析は、データクラスを理解する、という作業に対応している。また、デザインレシピにおける Step 2 以降は基底クラスのメソッド定義に適応できる。

Paterson ら [18] は初心者向けの Java プログラミング環境 BlueJ [*1]に対して、デザインパターンをサポートする拡張をした。この拡張を使うと、ユーザーはメニューボタンから使用したいデザインパターンを選択し、クラスの実装の型紙を取得することができる。これはデザインレシピのテンプレートの自動生成と類似している。なお、Paterson らの拡張は、システムの実装を知らない一般のユーザーでも、新たなデザインパターンを自分で追加できるという特徴がある。この点は本研究のプログラミング環境でも参考にしたいと考えている。独自のデザインレシピをユーザーが追加できるようにすることで、汎用性を向上させることができるからである。

6.3　初心者用のプログラミング環境

DrRacket [19] とは初心者向けの Racket プログラミング環境である。DrRacket の特徴としてまず 1 つ目に、言語レベルを設定できるというのがあげられる。これは、使用できる構文を制限することで、ユーザーの知識に合ったエラーメッセージを出力できるというアイディアに基づいている。2 つ目の特徴として Stepper というツールを備えている。これは、プログラムの実行を 1 ステップずつ観察できる、というものだ。3 つ目の特徴として構文チェック機能がある。これは、文字通り構文上の間違いを指摘してくれるものだ。

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

第 7 章

まとめと今後の課題

本研究では、デザインレシピに基づいたプログラミング環境の設計と、その実装を行った。これまでに、構造的再帰のレシピをサポートした。既存の環境にはないデータ構造の分析とテンプレートの作成のステップを明示化し、構造的に正しいプログラムを作成しやすくした。その他のステップにおいても丁寧な誘導を与え、網羅性の高いテストの作成を補助した。

今後の課題としては、まず 3.4 節 で述べた未実装の機能の実装を行う。次に、第 4 章にあげた 3 種類のデザインレシピを実装する。前述の通り、デザインレシピのパターンによって、構造的再帰のレシピとは異なるユーザの入力が必要となる。さらに、第 5 章で述べたフィードバック機能を追加する。この拡張を実現するアプローチとして、Scala の合成器を利用する方法と、Racket に一度変換する方法が考えられる。

これらが完了したら、東京工業大学の「計算機科学概論」でユーザ実験を行いたいと考えている。計算機科学概論は Scala を用いた入門的な講義であり、3.3 節、3.5 節で述べた関数定義をカバーする。この授業で得たフィードバックをもとに、初学者が苦戦するポイントを見つけ、環境がユーザに与える誘導をより効果的なものにしていきたいと考えている。

参考文献

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

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

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

[4] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. Transferring skills at solving word problems from computing to algebra through bootstrap. In *Proceedings of the 46th ACM Technical symposium on computer science education*, pp. 616–621. ACM, 2015.

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

[6] Norman Ramsey. On teaching* how to design programs*: observations from a newcomer. *ACM SIGPLAN Notices*, Vol. 49, No. 9, pp. 153–166, 2014.

[7] William E Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages*, Vol. 1, No. ICFP, p. 8, 2017.

[8] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *ACM SIGPLAN Notices*, Vol. 50, pp. 229–239. ACM, 2015.

[9] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pp. 135–152. ACM, 2013.

[10] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *ACM SIGPLAN Notices*, Vol. 51, pp. 522–538. ACM, 2016.

[11] Armando Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, Vol. 15, No. 5-6, pp. 475–495, 2013.

[12] Molly Q Feldman, Yiting Wang, William E Byrd, François Guimbretière, and Erik Andersen. Towards answering "am i on the right track?" automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, pp. 13–24, 2019.

[13] Hila Peleg, Sharon Shoham, and Eran Yahav. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1114–1124. IEEE, 2018.

[14] Hugh Glaser, Pieter H Hartel, and Paul W Garratt. Programming by numbers: a programming method for novices. *The Computer Journal*, Vol. 43, No. 4, pp. 252–265, 2000.

[15] Minjie Hu, Michael Winikoff, and Stephen Cranefield. Teaching novice programming using goals and plans in a visual notation, 2012.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable software architecture. *Reading: Addison-Wesley*, 1995.

[17] Matthias Felleisen, Matthew Flatt, Robert Bluce Findler, Kathryn E. Gray, Shriram Krishnamurthi, and K. Proulx, Viera. How to design classes. available at: `https://felleisen.org/matthias/HtDC/htdc.pdf`.

[18] James H Paterson, John Haddow, and Michael Nairn. A design patterns extension for the bluej ide. In *ACM SIGCSE Bulletin*, Vol. 38, pp. 280–284. ACM, 2006.

[19] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Drscheme: A pedagogic programming environment for scheme. In *International Symposium on Programming Language Implementation and Logic Programming*, pp. 369–388. Springer, 1997.