プログラム言語論: 最終レポート課題 についての解説

亀山幸義 (plm@logic.cs.tsukuba.ac.jp)

1　P. プログラミング課題 (プログラム、説明、例題 5 個以上)

プログラミング課題のすべてに共通するポイントとして、1. プログラムに対する説明をつけること、2. プログラムの実行例を 5 個以上つけること、の 2 点がある。また、文献 (ウェブページを含む) に記載されたものを引用したり、生成 AI などのコードを参考にした場合は、そのことを明記する必要がある。

これらを行っていない場合は減点した．特に，「2 (実行例を 5 個以上つける)」は授業で繰返し説明したことなので，それをふくまない答案は 30% 程度の大幅減点とした．

1.1　(課題 P1) 巨大正整数の演算

サンプルとして，「最上位の 0 を除く」，「2 つの巨大整数の加算」，「2 つの巨大整数の乗算」のコードを示した．(final1-bigint.ml)

まず，「最上位に 0 が (いくつか) 表れるかもれない巨大整数」にたいして，それらの 0 を除く操作を normalize 関数としてっ実装した．これは，最上位から順番に見ていく必要があるため，リストを逆順にしてから再帰をまわすのがよい．プログラムそのものは比較的単純な再帰である．

加算は、リストに対するプログラミングにおける非常に基本的な問題である。加算は、下のけたから順番に (けた上がりを考慮して) 足していけばよい.

ただし、私が chatGPT から得たコードに酷似したコードを出してきている人が結構たくさんいたのは、情報科学類の授業としては残念であり、この程度のプログラム (この授業のすべてのレポートを自力でといてきた人にはかなり簡単なプログラミング課題) は、自力で解答してほしかった。

乗算は、加算よりは少し大変であるが，人間が手書きで乗算をするときと同様に，「1 けたの整数」と「N けたの整数」の乗算をする関数をまず実装し，それを使って，「M けたの整数」と「N けたの整数」の乗算を 1 けたずつずらしながら加算していけばよい.

採点のポイント: 加算と「上位の 0 を除く」ということが最低ラインであり、それができていて、プログラムの説明があり、5 個以上の実行例を与えている (実行結果も表示している) ならば、70% の得点を与えた。

加算だけでなく乗算までできていたら (説明、実行例もあれば)100% である。

加算、乗算、除算までできていたら (説明、実行例もあれば) ボーナス得点を与えることにして 120% とした。さらに、べき乗や階乗なども実装したら、若干加点した。

なお，除算を実装するには (通常のやりかたなら)，減算の実装が必要であり，これら 2 つは，加算や乗算と違って，「最上位のけたから順番に計算する」必要がある．

実装した関数のテストについて，サンプルコードでは，乱数 (Random モジュールの関数) を発生させて，多くの巨大整数を発生させて，OCaml の演算結果と照合するようなコードを書いてテストをおこなった．

皆さんのレポートを見ていると，「5 個の例題でのテスト」を示してあっても，非常に短い例 (巨大整数といいつつ，3 けた以下のもの) ばかり 5 個を選んでいる人がおおくて，すこし残念であった．今後の人生でソフトウェアに携わる職業につく人たちには，是非とも，(1) ソフトウェアは書けばよいのでなく十分にテストをする必要があること，(2) そのためには，多くの場合，テストケースを生成するプログラムや，大量のテスト

を走らせるプログラムなどを書かないといけないこと，を理解して実践するようにしてほしい．

なお，「テストケースの作成に生成 AI を使った」とかいてあるレポートが複数あった．生成 AI の使用はまったく問題ないし，今回の問題についてはランダムに生成したテストケースでも十分であるので，生成 AI で作っても問題ない．ただ，一般的にいうと，現在の生成 AI は「予測」をするだけであり，論理式を解いて答えをもとめているわではないので，「本当に良いテストケースを生成できているか」という部分ではあまり信用しない方がよいだろう．

1.2 (課題 P2) 非常に長いリストのソート

非常に長いリストをソートするため、末尾再帰でプログラミングをするという問題である。ただし、「非常に長いリスト」を扱うのでない部分は末尾再帰でなくてよい。

サンプルコードは final2-tailsort.ml に示す．これは，以前の演習で示した insertion sort を末尾再帰になおしたものである．ただし，「比較関数」を引数として受けとる一般的ソートにすると実行速度が遅い気がしたので，そこは特殊化して「整数を小さい順に並べる」よう固定した．

サンプルコードにおける mysort1 からよばれる myinsert は末尾呼び出しの位置にはいないが，これは問題ないものである (mysort1 そのものは末尾再帰であり，myinsert そのものも末尾再帰である．)

このプログラムで使っている List.rev_append というのは，List.rev と似たような関数であるが末尾再帰であるものである．

こちらのプログラムについてもテストを行うため，「mysort の出力のリストが，たしかにソートされているか」をチェックする関数 isSorted を実装した．(一方，mysort の入力と出力が，並べかたを無視すれば同一のリストであるということはチェックしていない．)

末尾再帰になっている (stack overflow しない) ことをチェックするためには，非常に長いリストを生成して，それを mysort にかける必要がある．そこで，長さを 100， 1000， 10000 とのばしていってテストをした．が，今回の実装は非常に遅い Insertion Sort を採用したため，stack overflow するかどうかの限界点 (次の段落を参照のこと) に到達する前に，計算時間がかかり過ぎてしまった．これは，問題設定がいまいちだったせいであり，皆さんが心配する必要はない．(私自身は，merge sort を末尾再帰で実装して，stack overflow しないことを確認したが，そのコードは末尾再帰のあつかいのために結構よみづらいものになったので，ここでは示さない．興味がある人は個別に連絡してほしい．)

「stack overflow テストの限界点」について：OCaml で，末尾再帰でない関数をどのくらい「深く」最適に呼びだすと stack overflow を置こすかは，処理系によって異なる (stack として確保している領域のサイズによる)．したがって，自分で確認しないといけない．この値を近似的に知るためには「末尾再帰でない関数」を 1 つ定義して，それを何回再帰呼び出しすると stack overflow するか調べることになる．サンプルコードは，最後の方にそのテストのコードを含んでいる．亀山が使っている OCaml 処理系では，この値は，10^8 と 10^9 の間であった．従って，長さ 10^9 のリストを mysort にかけてみて，stack overflow しなければ，「末尾再帰できている」と言えるはずである．

方針で実装しており，おおむね，できていた．が，この問題については，「非常に長いリスト」を入力してみないと，テストになっていないことが明らかであるので，「5 個のテスト例題」の中に 1 つ以上の「非常に長いリスト」がないものは 80% の得点とした．(ソートが正しく動いているかどうかのテストはなくてもよい．)

上記のように，insertion sort などの遅いソートでは，「限界点」を試すことはできない (とおもわれる) ので，10^9 の長さのリストを試していなくてもよく，1000 以上の長さのリストを 1 つ試していればよいものと

した．当然ながら，長さ 1000 のリストを手入力できるわけがないので，ランダムな要素をもつリストを生成する関数 (テストのための関数) も実装する必要がある．実際，ここまでちゃんとやった人も何人かいてそのようなレポートは見応えがあった．

1.3 (課題 P3) 単一化

単一化は，実装方針がさまざまにわかれるのでとくにサンプルコードは示さない．

採点のポイント: こちらも例題 5 個をしっかりテストしておくのが必要である．レポートの中には，ものすごく単純な例題ばかり 5 個を選んでいる (まともなテストをしたくないと言わんばかりの $x = Int \to Int$ のようなテスト) ものがあり，これでは，実装がうまくいっているかどうかわからないため，(コードがそれなりにかけていたら) 80% の程度の得点とした．説明がしっかりしててテストについてしっかりやっているものは 100% に近い得点である．

2 R. 調査課題

2.1 (課題 R1) 継承とサブタイピングの違い

継承は，実装 (コード) を引き継ぐ (再利用) することであり，サブタイピングは，インタフェース (外部から見た仕様) を引き継ぐことというのが，一番単純な定義である．

ただし，問題文に明記されているように，「どのように異なるか、具体的なプログラム例を示した上で、説明しなさい。」ということなので，この 2 つの概念的な違いがわかるプログラム例を示す必要がある．(継承の例と，サブタイピングの例をそれぞれ示すのではなく，「継承とサブタイピングが異なる例」を示す必要がある．)

このようなプログラム例は，Java 言語ではちょっと難しい．(継承をする場合，サブタイプになる必要があるので，これらを明確に区別する例題が書きにくい．もっとも，Java 言語でも，「抽象クラスを，具体クラスで実装する」場合は，サブタイピング関係のみ成立する (抽象クラスにはコードの実態がないので) と言えなくもない．(これが良い例というわけではないが．)

他の言語では，たとえば，OCaml のオブジェクトシステムでは，授業資料に，これらの 2 つが区別できる例を記載しているので参照してほしい．(ただし， 授業のなかでは，この例を説明する時間はなかったので，自力で「解読」して説明する必要がある．その例が示しているのは，OCaml では，「継承」しなくても「サブタイプ」になることがあり，また，「継承」していても，「サブタイプにならない」という事も生じるということである．後者は不思議かもしれないが，self が存在する場合に生じる現象である．

採点のポイント: 文献等の説明をよく理解せまずに，そのままうつして提出してしまった人もいるようで，以下の説明は不適切なので減点要素とした．(これは，生成 AI でもそう答えるようである．)

よくない例: 「長方形クラス」を継承して「正方形クラス」を作成した例を示した．長方形クラスでは，縦横の長さを独立に設定・変更できるが，正方形クラスでは，縦横の長さを独立に設定・変更できないので，リスコフのいう「代入可能性」が成立せず，「サブタイプ関係」が不成立である．(つまり，継承しているのにサブタイピングでないという例である．)

この例がなぜ「よくない」か？

- 上記のレポートにおける「長方形クラス」等は，あるプログラム言語のクラスとして具体的な実装が与

えられいてた.

- しかし，その実装においては,「長方形クラスは，縦横の長さを独立に変更できる」という性質はかかれていない．(縦を変更するメソッドと，横を変更するメソッドは別のものとして与えられているが，片方を使うと他方の値も変わってしまうのか，変わらないのかなどの「性質」は書かれていない．)
- 従って,「レポートに書かれている正方形クラス」は,「長方形クラス」に代入可能である.

もし,「長方形クラス等が満たすべき性質」なども記述できていれば，もちろん，話はかわってくる．ただ,(リスコフ先生の頭にあるモジュール概念は別として) 実際に使えるプログラミング言語の「クラス」や「モジュール」はほとんどの場合,「性質」を記載することができない．せいぜいコメントで書けるだけである．したがって，たとえば,「スタッククラス」の実装として，pop(push(stack,x))=stack が成立「しない」ようなのも許してしまっている．(それが成立しないと「スタック」とは呼べないはずであるが，そのような性質を保証する手段が多くのプログラミング言語にはないので，しょうがない．)

というわけで，どうやら，生成 AI が作成した解答は,「昔のリスコフ先生の時代の (理想的な) モジュールに対するサブタイプや代入可能性に関する文献」と「現代のオブジェクト指向プログラミング言語におけるクラスに関する話」をごっちゃにまぜてしまったようであり，理解不能な内容になっているようである.

参考までに，現在われわれが使えるプログラミング言語の中には，プログラムの性質を記述して，その性質が成立するかどうかをチェックしてくれるものがある．たとえば，Agda は「依存型」という型システムをもっていて，これは，プログラムの性質を記述できる．また，Coq (最近名前が Rocq に変更された) は，Agda よりさらに強力な型システムをもっていて，プログラミング言語というよりは論理的な推論や検証のための言語という側面が強い (でも，プログラムもいっぱいあ書ける) 言語である．これらは，普通にインストールして使える言語であるので，興味がある人はあたってみてほしい.

2.2 (課題 R2) row 多相とサブタイプ多相

これは，row 多相の特徴をのべたあとで，subtyping 多相と比較すればよく，どちらも文献 (インターネット上の記事をふくむ) で十分説明されているので，ここでは説明は省略する.

提出されたレポートは，総じて非常によく書けているものがおおかった.

2.3 (課題 R3) 動的型付け言語と静的型付け言語 (明示的，暗黙的)，型付けのない言語

これら 4 つについて，代表的なプログラミング言語を 1 つ以上あげて，それらのメリット，デメリットを比較すればよい.

これも文献で十分説明されているので，ここではきちんした解説は省略するが，一点だけ，この授業では静的な型付けについて様々な話をしてきたので単純に「静的な型があると信頼性が高い」というようなレベルではなく，もっと踏みこんで書いてほしかった．たとえば,「静的な型があると，コンパイラがそれを利用して，より高速な実行コードを生成できる可能性がある」とか,「型の情報をプログラムのドキュメントとして利用できる」とか,「ユーザ定義の型をつかうと，US ドルと日本円の区別をできるなど，プログラムの意味に応じた (高度な) 安全性を確保できる」などといったことである.

また，ここでも文献 (インターネットや生成 AI による正しいことも間違ったこともまぜこぜにしてしまう解説) をそのままうつしてしまったようなレポートが散見された．これは，あまり細かく (精密に) 減点することはしていないが，間違いが目立つものは減点した．また，当然ながら,「4 つの種類で 1 つ以上ずつのプロ

グラミング言語の例」を書いていないものは減点した.