2006－CE－83（19）
2006／2／18

比較的大きなプログラミング課題のための自動採点システム

田上　恒大†　　阿部　公輝†

† 電気通信大学情報工学科
〒 182–8585 調布市調布ヶ丘 1–5–1
E-mail: †{taga,abe}@cacao.cs.uec.ac.jp

あらまし　プログラミング演習において、小規模な課題では目視で誤りを発見することが可能であるが、複雑な課題では困難な場合が多い。テスト入力を提出プログラムへ与え結果を照合すればよいが、これを手動で行うには講師の負担が大きく自動化が望まれる。従来の自動採点システムは比較的小規模な課題を対象とする例が多い。実行結果の照合では、一字一句合わなくてもキーワードが指定された順番に出力されたときに、正答とする必要がある。データ構造を直接検査することにより、より詳細な検査が行える。メモリ確保/解放関数が適切に使用されているかを検査する必要がある。さらに、提出プログラムの誤りにより、処理や記憶のリソースを消費し尽くす可能性等への対策が必要である。本研究では比較的大規模な課題を対象とし、これらの問題を考慮した自動採点システムを提案する。

キーワード　プログラミング演習、比較的大規模な課題、学習支援システム、教育ソフトウェア開発

A Testing System for Comparatively Large Programming Exercises

Kôta TAGAMI† and Kôki ABE†

† Department of Computer Science, The University of Electro-Communications
1–5–1 Chofugaoka, Chofu-shi, Tokyo, 182–8585 Japan
E-mail: †{taga,abe}@cacao.cs.uec.ac.jp

Abstract　In programming exercises it is easy to discover errors by reviewing source codes writen by students for small programs, but herd for medium to large ones. To verify such programs we need to give inputs to them and compare the outputs with expected results. However, the verification process takes time, its automatization being desired for helping instructors evaluate student's programs. The matching of outputs with expected results must be made so that the outputs be accepted when some keywords appear in an specified order instead of strict matching. Detailed testing can be made by examining the trace of data structures. Testing if memory management functions such as malloc/free are properly used is also required. Further, some means of preventing the memory and processing resource from being exhausted by erroneous programs needs to be provided. In this paper, we describe a system which overcomes above problems in testing and evaluating comparatively large programs.

Key words　Programming exercise, comparatively large program, learning aid system, educational software development

1. はじめに

一般的にプログラミング演習の流れは以下の通りとなる。

（1）　講師は学習者に課題を提示する。

（2）　学習者は課題で指示されたプログラムを作成、デバッグする。

（3）　学習者はプログラムと実行結果を添えて提出する。

（4）　講師は提出物を読み、学習者が正しくプログラミングしたことを確認する。

学習者が誤ったプログラムを提出したときには (2)〜(4) を繰り返せるとよい。迅速に採点結果を返せれば学習の効率が上がる。しかし多人数の学習者に対してこれを行うのは講師の負担が大きすぎて現実的でない。

上の (4) では、講師が目視で提出されたプログラムと実行結果を読み、課題の意図どおりにプログラミングされているかを確かめることになる。目視でプログラムを読む場合、初級者レベルの課題であれば誤りの発見は比較的容易であるが、中級者以上の課題では発見は不可能に近い。

初級者向きの課題での実行結果の照合では、一字一句合わなくてもキーワードが指定された順番に出力されたときに、正答

とする必要がある。中級者以上の課題において、確保されたメモリの外側を使用してしまうのはC言語ではよくある誤りの一つである。また、解放されたメモリを使用してしまう誤りもある。これらの誤りに適切に対処する必要がある。さらに初級者と比べて中級者向けの課題では、より多くのテストを行わなければ正誤の判定はできないが、学習者の書いた main 関数を用いて何度も実行しながらテストを行うのは現実的ではない。提出プログラムの誤りにより、処理や記憶のリソースを消費し尽くす可能性等への対策も必要である。

本研究では比較的大規模な課題を対象とし、以上の問題を考慮してプログラムの提出、コンパイル、動作確認を行う自動採点システムを提案する。

2. 関連研究

自動テスト機能を備えたプログラム提出システムとして [1] がある。Web を介して提出されたプログラムをコンパイルし、あらかじめ用意したテスト入力を与えテストが行われる。コンパイルエラーが出た場合、コンパイラのエラーメッセージが表示される。また、典型的なエラーメッセージには日本語訳が付加される。さらに、エラーメッセージに応じてデバッグするときに有用なヒントが表示される。このシステムは、初心者向きの課題を想定している。

ACM プログラミングコンテストではメールを利用した採点システムが使われている [2]。解答のプログラムと出題者が用意した入力を与えて得られた実行結果をメールで送信する。採点システムではコンパイルは行わない。実行結果は厳密なマッチングが行われ、スペースが一つ多く入っていただけでも不正解となる。

また、自動コンパイルを行うプログラム提出システムとして [3] が提案されているが、構文チェックのみを行う。

3. 設　　計

3.1 システム概要

プログラムの提出にはメールを用いる。提出するファイルがテキストファイル一つでよい場合には本文にファイルの内容を記述し指定されたメールアドレスに投稿することにより提出する。複数のファイルを提出しなければならない場合は、添付ファイルとして提出する。システムがメールを受け取るとファイルが展開され、コンパイルされる。コンパイルに失敗した場合、学習者にエラーメッセージを返却する。課題に応じたテストを行う。テスト方法は課題の規模に応じて変える。学習者にテスト結果を返却する。

3.2 初級者向け課題のテスト方式

初級者向け課題のテスト方法として以下の方法を用いる。テスト入力とテスト出力の組をテストケースとして拡張 BNF 記法で図1のように定義する。テストケースは先頭から順に行単位で解釈される。行頭が'<' だった場合、行頭を除いた文字列がテストプログラムに入力される。行頭が'>' だった場合、行頭を除いた文字列がテストプログラムから出力されて来るまで待つ。それ以外の文字列が出力された場合無視する。一定時間出力されてこなければ不正解だと見なす。テストケースのすべての行が処理され、テストプログラムが正常に終了したら、正解とする。これによって、学習者によって生じるプログラムの揺らぎを吸収できる。

```
<input>       => (<line> '\n')*
<line>        => <input data> |
                 <output data>
<input data>  => '<' STRING
<output data> => '>' STRING
```

図1　テストケースの拡張 BNF 記法

```
main(){
    int x, y;
    scanf("%d %d", &x, &y);
    printf("%d\n", x + y);
}
```

図2　課題1において想定される解答その1

```
main(){
    int x, y;
    scanf("%d %d", &x, &y);
    printf("answer: %d\n", x + y);
}
```

図3　課題1において想定される解答その2

```
main() {
    int x, y;
    printf("x=");
    scanf("%d", &x);
    printf("y=");
    scanf("%d", &y);
    printf("x+y=%d\n", x + y);
}
```

図4　課題1において想定される解答その3

テストケースは課題の規模に応じていくつか用意し、それにすべて正解した場合、正しいプログラムが提出されたと判断する。

次のような課題を出題したときについて考える。

課題1　数値を二つ受け取り、加え合わせた値を表示せよ。

このとき学習者によってプログラムや実行結果は異なり、図2, 図3, 図4 の解答などが想定されるが、図5 のテストケースで問題なく正誤を判定できる。

3.3 中級者向け課題のテスト方式

中級者向けの課題では主要な関数のプロトタイプ宣言だけを実装したサンプルを学習者に示し、学習者に関数内を実装させる課題が多くある。例としてはツリーの操作、ヒープの操作、

```
<3\n
<5\n
>8
```

図 5　課題 1 のテストケース

リストの操作など基本的なデータ構造に関連する課題である。

中級者向けの課題でも上記の方法は十分適用可能である。しかし、いくつか問題がある。一つめは上記の方法では適切なメモリ管理が行われているかどうかを判断するのは難しいことである。プログラミング演習で中級者の学ぶべきこととして、メモリの動的確保/解放は大きなウェイトを占めているのでこれは重要である。二つめはデータ構造を直接調査することができれば、正誤判定の大きな助けになる。

一つめの問題について議論する。確保されたメモリの外側を使用してしまうのは C 言語ではよくある誤りの一つである。また、解放されたメモリを使用してしまう誤りもある。これらのバグは、誤ったコードが実行された直後は何ともなく、書き換えられた場所を再び使用しようとしたときにデータ構造の不整合によってプログラムが不正な動作をすることである。つまり原因になったコードが実行された直後ではなく、しばらくしてからプログラムが不正な動作をするためデバッグ困難なバグを生む原因になりうる。

適切なメモリ管理が行われているかどうかを検査するために、テスト対象のプログラムの malloc, free といったメモリ確保/解放関数をテスト用の関数に置き換え、メモリ管理関数への引数を監視する。具体的には以下の通りである。確保、解放の回数を記録する。確保されたメモリの両端に検出用の領域を配置し、書き換わっていた場合に境界を越えたアクセスがあったと見なす。確保したメモリは一定の値で初期化する。解放するメモリは一定の値で初期化してから解放することにより解放後のメモリ使用を防ぐ。解放するときに検出用領域を検査する。プログラムの終了時に解放されていない領域がないかを検査する。これらは一般的なデバッグ用ライブラリで行われていることである。

二つめの問題について議論する。初級者と比べて中級者向けの課題ではより多くのテストを行わなければ正誤の判定はできない。そのため、学習者の書いた `main` 関数を用いて何度も実行しながらテストを行うよりも、講師の書いたテスト用 `main` 関数を用いて学習者の作成した関数を検査するようにするほうが効率がよい。検査プログラムは学習者の作成した関数を呼び出すごとにデータ構造が正しいかを検査すれば、データ構造に不整合が起こった場所が解り都合がよい。

3.4　リソース保護

不特定多数の学習者の書いたプログラムをコンパイルし実行するため、実行時に提出されたプログラムの誤りによりリソースを消費し尽くしてしまうことへの対策が必要である。また、悪意あるプログラムが実行された場合への対策が必要である。これらの問題に対処するため学習者のプログラムの実行は SandBox と呼ばれる他のプロセスや重要なファイルやネットワークから隔離された空間で実行する機構を用いることにする。

SandBox の実装にはいくつかあるが、大きく分けると 2 つの実装方法がある。カーネルレベルでの実装とユーザーレベルでの実装である。カーネルレベルでの実装ではカーネルのモジュールとして SandBox が実装される。ユーザーレベルでの実装よりも高速であるという利点を持つ。ユーザーレベルでの実装ではデバッガと同様の仕組みを利用し実行ファイルとして実装される。カーネルへのモジュールの追加が不要でユーザー権限でも利用できるという利点がある。

カーネルレベルの SandBox は研究が盛んで、セキュリティ関連の応用範囲も広く最近注目されている。Linux でも SELinux [4]、LSM [5] として組み込まれているディストリビューションもある。しかし、ユーザーレベルの SandBox はその構造から速度がどうしても遅くなるため実用に向かず、そとため研究もあまりされておらず実装も少ない。

ここでは他の OS への移植性やインストールのしやすさを考え、ユーザーレベルの SandBox の s4g [6] を用いる。

一つのシステムを構成するような大きな課題に対してのテストは、誤動作したときのリソース消費が激しいことが予想されるので、必ず SandBox に閉じこめて実行する必要がある。また、このような課題にはネットワークを扱う場合が多いので、ネットワークへの細かいアクセス制限ができる SandBox が望まれる。具体的には、接続先ポートごと、接続ホストごとのアクセス制限が可能な SandBox である。

4. 実　　装

本章では、先に定義したテストケースの解釈を行うテスター、データ構造の検査を行うテスター、メモリの確保/解放関数をフックするライブラリの実装について述べる。

4.1　テストケースの解釈を行うテスター

Perl と仮想端末への入出力をエミュレーションするライブラリ `Expect.pm` を用いて実装した。テストケースを解釈し、テスト対象プログラムへ入力を与え、出力を監視する。テストケースのすべての行を処理したら終了コード 0 を返し正常終了する。学習者の書いたテスト対象プログラムは通常の実行ファイルにコンパイルする。

4.2　データ構造の検査を行うテスター

学習者の書いたテスト対象プログラムはダイナミックリンク可能なライブラリ形式にコンパイルする。このライブラリを `leaner.so` と呼ぶことにする。これにより、テスト対象プログラムの `main` 関数のテストも、その他の関数のテストも可能になる。テスト関数もダイナミックリンク可能なライブラリ形式コンパイルする。このライブラリを `tester.so` と呼ぶことにする。`tester.so` に `leaner.so` 内のデータ構造を出力する関数 `dump` と、いくつかのテスト関数で構成される。共通関数はテスターへ配置する。テスト関数では、一つの操作を行う度に `dump` を呼び出しデータ構造を出力する。

模式図を図 6 に示す。まずテスターは `leaner.so` をロードする。続いて `tester.so` をロードする。`tester.so` 内のテス

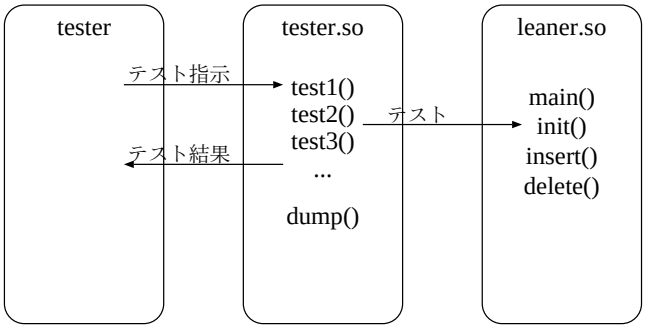

図 6　テスターの模式図

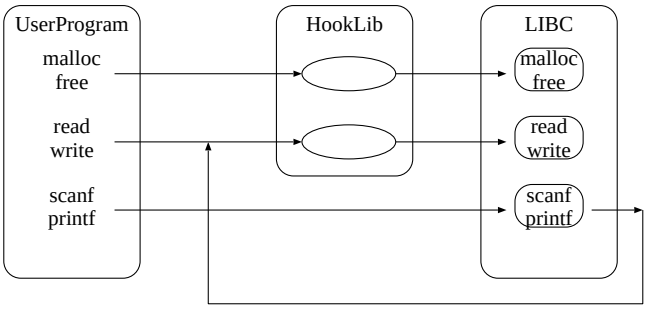

図 7　LIBC のフック

```
$ LD_PRELOAD=./libsfl.so cat /dev/null
malloc 4180byte(6), free 4096byte(1), realloc (0)
```

図 8　プリロードライブラリを用いたメモリ管理関数の監視

ト関数を一つずつ `fork` した子プロセスで実行する。一つのプロセスでは一つのテスト関数のみ実行する。これは他のテストの影響を排除するためである。

4.3　メモリの確保/解放関数のフック

境界を越えたアクセス検出のためのプリロードライブラリを作成した。図 7 のように LIBC の `malloc`, `free` といったメモリ管理関数を横取りすることにより、引数を監視し検出用領域を埋め込む。割り当てられたメモリの外側へのアクセスを検出した場合 `abort` によって強制終了させる。また、割り当てられたメモリの総量と回数を数え、プログラム終了時に表示させる。このライブラリは環境変数 `LD_PRELOAD` にライブラリのパスを指定することにより実行可能である。

4.4　SandBox

SandBoxにはs4gを用いる。s4gは `s4g command arg1 arg2 ...` のようにコマンドとその引数を与えると、SandBox 内でそのコマンドを実行できる。SandBox 内ではデフォルトでファイルへの書き込みが禁止される。また、ファイルへの読み込みはカレントディレクトリのみに制限される。ネットワークは使用不可能になる。これによって悪意があるプログラムであっても安全に実行可能になる。ディレクトリごとに読み込み、書き込み、実行を個別に制限可能である。ネットワークはすべて使用可能にするか、不可能にするかのどちらかが設定可能である。

```
$ sfl_emuterm testcase.txt ./lec1_1
3
5
8
$ sfl_emuterm testcase.txt ./lec1_3
3
5
answer: 8
$ sfl_emuterm testcase.txt ./lec1_2
x=3
y=5
x+y=8
$
```

図 9　課題 1 において想定される解答のテスト結果

```
struct tNode{
    struct tNode* next;
    int value;
};
typedef struct tNode node;
void init_list();
void insert_list(int value);
void delete_list(int value);
node* head;
```

図 10　課題 2:プロトタイプ宣言

5.　実行結果と評価

課題 1 の想定される各プログラムに対しテストケースを適用した結果を図 9 に示す。

中級者向けの課題の例として、図 12 に示すようなリスト操作を取り上げる。学習者には図 10 のようなプロトタイプ宣言を与え、`init_list`, `insert_list`, `delete_list` を実装させる。`init_list` 関数では、リストを初期化する。`insert_list` 関数では引数に与えられた値をリストに追加する。`delete_list` 関数では引数に与えられた値をリストから検索し削除する。

次に例として `insert_list` のテスト関数を、図 11 に示す。最初に `name` でテスト関数の名前を定義する。次に `init_list` でリストを初期化する。最後に `insert_list` でリストに一つずつ要素を追加していく。そのとき要素を一つ追加するごとに `dump` を用いて、データ構造を表示する。`dump` についても講師が作成する必要がある。追加される様子を図 12 に示す。

テスト関数は、課題ごとにいくつか作成する。例としてあげた課題であれば、`insert_list` のテスト関数、`delete_list` をリストの先頭から削除するように引数を与えるテスト関数、同様にリストの末尾から削除されるように引数を与えるテスト関数等である。これらのテストの出力を図 13 に示す。

テスト結果の正誤判定は講師が作成した模範解答と学習者の作成したプログラムのテスト結果を比較することによって行わ

```
tester test1() {
    name("insert test");
    init_list();
    for(int i=1; i<N; i++){
        insert_list(i);
        dump();
    }
}
```

図 11　講師が書いたテスト関数

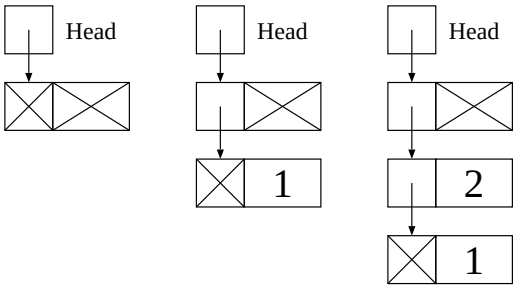

図 12　リストの挿入

```
$ ./sfl_tester -l ./leaner.so -t ./tester.so
1:insert test
1.1:1
1.2:2 1
1.3:3 2 1
1.4:4 3 2 1
1.5:5 4 3 2 1
2:delete test(head)
2.1:5 4 3 2 1
2.2:4 3 2 1
2.3:3 2 1
2.4:2 1
2.5:1
2.6:
3:delete test(tail)
3.1:5 4 3 2 1
3.2:5 4 3 2
3.3:5 4 3
3.4:5 4
3.5:5
3.6:
```

図 13　テストの実行結果

れる。

これらのテストを s4g を用いて SandBox 内に閉じこめる方法は、s4g からテスターを起動することで行える。図 14 は SandBox 内で『パスワードファイルを表示する』、『ルートディレクトリのファイルのリストを表示する』、『SMTP サーバーにアクセスする』を行おうとして拒否されている様子である。

6. 考　　察

初心者向け課題に対する方法として提案したテストケースを使用した方式では例に挙げた課題に対し正誤判定ができることを示した。しかし、配列の要素を出力する課題などで、余計な要素まで出力してしまっている場合であっても正答と判定してしまう可能性がある。また、出力されるキーワードの順番が決まっていない課題ではこの方式を適用することはできないが、これに該当するような初心者向け課題はほとんどないので、それほど問題にはならない。

メモリ確保/解放関数をフックしてガード領域を配置し確保、解放時に領域を初期化する方法は、一般的なデバッグ用ライブラリにも実装され、システム開発等でも使用されており効果を上げている方式なので、本システムでも効果が期待できると思われる。この方式の欠点は、メモリの確保/解放時に毎回確保されている領域すべての整合性を検査するとオーバーヘットが大きいことである。そのため、一般のシステム開発ではメモリ解放時には解放されるメモリの整合性のみを検査し、すべての領域の検査は一定時間ごとに検査する場合が多い。しかし今回のシステムの場合、課題にもよるが 1 回のテストで呼び出されるメモリ確保/解放関数の回数は、せいぜい数百のオーダーであるので、メモリ確保/解放時にすべての領域の整合性を検査しても処理量が問題になることはない。

データ構造をトレースし、模範プログラムと学習者の提出プログラムのテスト結果を比較する方式は、講師の書いたテスト関数次第でプログラムを関数ごとにテストができることを示した。また、プログラムのデータ構造を直接見て正誤を判定するのではなく、模範プログラムと提出プログラムのデータ構造を比較することによって正誤を判定するので、データ構造が複雑であっても、データ構造を出力する関数 dump を実装し、テスト関数では dump を用いて要所の状態を出力するプログラムを記述すればよい。そのため、テスト関数はあまり労力をかけることなく書くことができる。しかし、ハッシュ操作に関する課題など、この方式を適用しにくい課題はいくつか存在する。ハッシュ操作では、使用するハッシュやテーブルサイズによって要素の格納される場所は異なるため、模範プログラムとのトレース結果の比較によっては正誤判定はできない。もう一つの問題として、課題を出題する際、使用する関数やグローバル変数、構造体を厳格に指定する必要があり、課題が堅苦しくなりがちになる。

s4g が SandBox としてアクセス制限が適切に働いていることを示した。テストを行う際、SandBox 内で実行すれば、テスト対象プログラムが必要ないファイルへのアクセスやネットワークを使用することを防ぐことができる。

採点システムが得た情報のすべてを学習者に伝えることが良いかどうかには議論の余地がある。中級者にもなるとバグの存在箇所を発見しデバッグする能力は当然学ぶべきことであるからである。無制限に学習者に情報を与えると、採点システムは学習者にとって都合の良いバグ探し機になってしまう。われわれの意見としては、正しく実装されていない関数名まで教える

```
$ s4g cat /etc/passwd
cat: /etc/passwd: Operation not permitted
$ s4g ls /
[2789] ls stoped while calling 'lstat64' (196)':
        do not have read acces on /
$ s4g telnet localhost 25
[2784] telnet stoped while calling 'socketcall'
(102)':
        Sorry, only local and unix sockets are
allowed
```

図 14　s4g の使用

必要はないと考える。だが、バグ発見の道しるべとなる情報は与えるべきであろう。学習者ごとに採点システムの使用回数を制限するなど何らかの方策を考える必要がある。

7. おわりに

初心者向け課題のテスト方法としてテストケースを定義し、テスターを実装、評価した。中級者向けの課題へのテスト方法として、データ構造のトレースを用いたテスト方法を提案、実装、評価した。またメモリ確保/解放関数の誤使用を検出するためのライブラリを実装、評価した。テスト時にシステムに悪影響を及ぼさないように保護する SandBox として s4g を採用した。以上の結果、本システムの有効性が確かめられた。

現在、採点システムの採点の粒度は講師の書いた検査プログラムに依存している。細かい検査を行う検査プログラムを記述するのは大きな労力がかかる。s4g を改良しネットワークへのアクセス制限を細かくできるようにすることにより、ネットワークを使用するより大きいプログラムへの対処が可能になると思われる。今回はテスト部のみの制作であったのでメールの受信部を制作しシステムを完成させる。テスト出力を学習者にわかりやすいように加工する操作が必要である。実際にプログラミング演習の授業で使用し評価する。

文　献

[1] 望月 将行, 森田 直樹, 北 英彦, 高瀬 治彦, 林 照峯, "自動テスト機能を備えたプログラム提出システム," 2003 PC カンファレンス, pp343-344, 2003.

[2] Association for Computing Machinery, http://www.acm.org/

[3] 熱田 智士, 松浦 佐江子, "Java プログラミング演習向け課題レポート提出・管理機能を付加した授業支援システム," FIT2004 第 3 回情報科学技術フォーラム, pp359-362, 2004.

[4] Security-Enhanced Linux, http://www.nsa.gov/selinux/

[5] Linux Security Modules, http://lsm.immunix.org/

[6] S4G, http://s4g.gforge.inria.fr/