『プログラム言語論』 期末試験 解答例
試験実施: 2012 年 7 月 2 日

問 1. (配点 30 点) MiniC 言語で書かれた次のプログラムについて以下の問に答えよ.

```
int x; int y; int w;
int f (int z) {
  print x+y+z;  /* C 言語の printf(%d\n", x+y+z); と同じ */
  return z;
}
int g (int z) {
  int x;
  x = 100;
  if (z == 0) {
    int y;   /* ここで局所変数を導入 */
    y = 200;
    return f(z);
  } else {
    y = 300;
    return f(z);
  }
}
int main () {
  x = 10; y = 20;
  w = g(f(0));
  return g(1);
}
```

上記のプログラムを以下のそれぞれの方式で実行すると、どのような値が印刷されるかを示しなさい.

1-a. 静的束縛かつ値呼び

1-b. 静的束縛かつ名前呼び

1-c. 動的束縛かつ値呼び

1-d. 動的束縛かつ名前呼び

1-e. また、上記のプログラムを静的束縛かつ値呼び方式で実行した時、`print x+y+z;` が実行される瞬間のスタックの内容を図示しなさい。

1-a. 解答例 実際に MiniC 言語処理系で走らせると以下の解答を得る。

```
miniC interpreter Version 6.5 (2012/05/17):
30
30
311
```

(補足) なぜ、この出力が得られるかを知るため，静的束縛かつ値呼び方式で実行した時のスタックの変化を図 1(後述) に示す.

1-e. 解答例

図 1 のスタックの図のうち print が実行される瞬間のものを図示すればよい.

1-b. 解答例 1-b については、2 通りの異なる答えがあり得る。授業では、MiniC 言語の return 文の振舞いを厳密に定義しなかったため、その部分で差異が出るためであり、もちろん、どちらも正解である。

まず、多くの人の解答にあった出力を見よう。

```
30
30
30
311
```

一方、MiniC 処理系の出力は、少しだけ異なる。

```
miniC interpreter Version 6.5 (2012/05/17):
30
30
30
30
311
```

若干の解説 (1 つ目の答えの場合): 静的束縛かつ名前呼びの場合、g(f(0)) という関数呼びだしでは f(0) は評価されず、g の仮引数 z が f(0) に束縛される。そのあと if (z==0) という比較において、z の値が必要になるので、f(0) を計算しに行き、その際に (静的束縛なので) x=10,y=20,z=0 という環境が使われ「30」が出力される。f(0) の返り値は 0 である。次に、(z==0) が true となって、int y; y=200; を実行し、f(z) を計算しにいく。(このとき、z=f(0) だが、まだこの z は計算されない。) f の本体の計算において x+y+z の計算があるので、z の計算をしに行って、もう 1 度 30 が印刷され、さらに x+y+z の値が計算され、さらに 30 が印刷される。ここまでで g(f(0)) の計算は終わりで、あとは g(1) の計算の際に 311 が印刷される。よって、この場合は 30 が 3 回印刷されたあと 311 が印刷される。

若干の解説 (2 つ目の答えの場合): MiniC 言語では、return z; という文を実行するときも、z の値を計算しにいってしまう。上記の計算過程で、1 度だけ return z; の z が f(0) のときがあり、その際にもう 1 度 30 が印刷される。よって、この場合は 30 が 4 回印刷されたあと 311 が印刷される。

return z; を実行するとき、(z がまだ値ではなく f(0) のような式であるとき)、z の値を計算するかどうかは、授業では特に決めておらず、どちらもあり得るところであるので、両方とも正解にする。(ただ、1 つ目の方が自然な意味論と思われるので、来年度の授業で使う MiniC 言語の処理系は、そちらに修正しようと考えている。今年度については、今さらこっそり処理系の実装を変更してしまうのは良くないので、そのままとしている。)

1-c. 解答例

1-c は以下のものが正解である。

```
miniC interpreter Version 6.5 (2012/05/17):
30
300
401
```

これは演習で何度もやったパターンであり、解説の必要はないだろう。

1-d. 解説: 出題意図が曖昧だったため、採点せず、全員に得点を与える

「動的束縛かつ名前呼び」という計算方式の意味論は、授業では正確に定めておらず、かつ、「自然なもの」は1つには定まらないので、この問題に「絶対の正解」はない。この点をお詫びして、この問題を採点対象からはずし、全員にこの部分の得点 (6 点) を与えることにする。

何人かの人は、「120, 300, 300, 401」が印刷されると解答したし、これは1つの自然な解答であるが、現在の MiniC 言語処理系 (version 6.5) は、「30, 30, 300, 30, 401」を順に印刷する。後者はまったく意外な結果かもしれないが、これは、f(0) という式をフリーズし (計算をせずに、式のまま、g の実引数として与え)、その後に f(0) という式を計算するとき、フリーズした地点の環境に戻って計算する、というものである。この問題の場合、f(0) という式が関数 g の引数となった地点では、x=10,y=20 なので、最初に印刷されるのは 10+20+0=30 となる。この「フリーズした時点の環境に戻って計算する」ということをしなければ、前者のように「120」が最初に印刷される。

実は、MiniC 言語処理系は、この両方の意味論を実装しているのであるが、前者の意味論 (おそらく、皆さんの大多数が賛成する意味論) では、別の計算で困るので、後者の意味論を採用した、という経緯がある。「別の計算で困る」というのは、本問のプログラムにもあるが、仮引数も実引数も z だったとき、z=z という環境を生成するが、そのもとで z を評価すると、「環境をフリーズした時点までまきもどす」ということをしなければ、いつまでたっても z のままで計算が停止しない、ということである。

というわけで、結局、「動的束縛かつ名前呼び」に関しては,「これぞ」という適切な意味論がないし、そもそも、今年の授業では、このパターンは教えないことにしていた (演習問題からは、はずした) ので，これを問う問題を出したこと自体が不適切であった。

問 2. (配点 30 点) 次の 2 つの MiniML プログラムについて考える.

```
let rec f n =
 if n < 2 then
   1
 else
  f(n-2)+f(n-1)
in f 5 ;;
```

```
let rec g n =
  if n < 2 then
   (1,1)
  else
   let p = g(n-1) in
     ((snd p),(fst p)+(snd p))
in snd (g 5) ;;
```

ただし、授業で使った MiniML 言語は `<` や `-` という記号を持たなかったが，ここでは，C 言語や OCaml 言語と同様，大小比較および引き算の記号として使えるものとする．また，MiniML の `(a,b)` は，対 (ペア) のデータ構造を表す．また，`fst` と `snd` は，それぞれ，対の第 1 要素，第 2 要素を取りだす関数である．

これらのプログラムを MiniML で静的束縛かつ値呼び方式で実行する時，以下の問に答えなさい．

2-a. 上記の左のプログラムを実行し，スタックが最も深くなった瞬間 (最も多くのスタックフレームがつまれた瞬間) のスタックの中身を図示しなさい.

2-a. 解答例 スタックが最も深くなるのは，f(1) が呼ばれたときと，f(0) が呼ばれたときであり，図 2 のとおりである．(本問は，どちらか 1 つだけ答えても満点とする．)

2-b. 上記の右のプログラムを実行すると、その途中で `(g 3)` という関数呼び出しする。

その関数呼び出しの瞬間と，プログラムが終了する瞬間の、スタックとヒープの中身を図示しなさい。なお、変数の値が決まっていないときは,「未定義 (undefined)」と書きなさい。

2-b. 解答例 図 3 に，右側のプログラムの実行に伴う，スタックとヒープの概略を示す．(ただし，アクセスリンクは常に (global) を指し、コントロールリンクは常に 1 つ下のスタックフレームを指すときまっているので、この絵では省略した．また，ヒープの絵は特に書きかたが決まっているわけではないので，ペアのデータ構造を 1 単位として書いた．)

解答としては，この図で (g 3) を呼びだした瞬間と，プログラム終了の瞬間の部分を記せばよい.

2-c. 「対」のデータはヒープに作成されるが，もし，これがスタックに作成されたとしたら，どのような問題が発生するか説明せよ。(上記のプログラムを例に取ればよい)。

2-c. 解答例 もし，ペア (対) のデータ構造が，現在のスタックフレーム (スタックのトップにあるスタックフレーム) において作られるとすると，関数呼び出しが終了したあとに消去されてしまうので，関数が返す値としては使えない.

(以下は補足) そもそも，関数の中で，ペア (対) のデータ構造が何個作られるかは，事前に予測はできないので (関数の中で，ループによってペアを多数生成することも可能)，スタックフレームの中にペアのデータ構造を作るためには，(既に作成されたスタックに積まれた) スタックフレームのサイズを，動的に (どんどん) 大きくしなければならない，という問題が生じる.

2-d. 上記の 2 つの関数 f と g は同じ計算をするプログラムだが、実行効率が異なる．その理由を簡潔に説明せよ.

2-d. 解答例 右のプログラムの方が効率が良いと考えられる．なぜなら，左のプログラムは，f(5) を計算するために，f(3) や f(2) を何度も呼びだすのに対して，右のプログラムは，引数ごとに 1 回の関数呼び出しだけであるから.

補足 0. ただし，右のプログラムでは，ヒープ領域上にペアのデータ構造を展開する分で余計な手間がかかっているため，小さい n に対して「f(n) が g(n) より計算時間がかかる」とは言いきれない可能性がある．n が大きくなれば,「f(n) が g(n) よりかなり遅い」のは確かである.

補足 1. 左のプログラムでは，一般に f(n) を計算するためには，f(1) の関数呼び出しは，n 番目の fibonacci 数の回数起きる．(これは、n の指数関数程度で抑えられる数である。)

補足 2. メモリの使用量でいえば，左のプログラムの方が良い．なぜなら，左右で，スタックの使用量は同じ (か右の方が多く使う) のに対し，ヒープは右のプログラムのみ使っているからである．ただし，右のプログラムで使用されたヒープ領域は，いつかは，ごみ集めにより回収されるので，それほど害はないと言える．結局，計算時間とメモリ使用量のトレードオフ (trade off, 一方を良くすると他方が悪くなる) という関係にある.

2-e. 上記の右側のプログラムを，更に次の図の左側のように書きかえた．(参考までに，OCaml での通常の書き方をした同等のプログラムを右側に載せている．)

```
let rec h n = fun p ->
 if n < 2 then
   p
 else
  (h (n-1)) ((snd p),(fst p)+(snd p))
in snd ((h 5) (1,1)) ;;
```

```
let rec h n p =
 if n < 2 then
   p
 else
   h (n-1) (snd p,(fst p)+(snd p))
in snd (h 5 (1,1)) ;;
```

このような形のプログラムは末尾再帰と呼ばれる。この形式の利点について，簡潔に (2-3 行程度で) 説明せよ.

2-e. 解答の前に説明 若干こみいったプログラムだが，h は (本質的には)2 引数の関数であり，第 1 引数は n (n 番目の fibonacci 数を求める)，第 2 引数は「それまで計算がおわった部分的な解」である．たとえば，(h n) (1,1) は n=3,4,5 に対して，それぞれ，(1,2), (2,3), (3,5) であり，ペアをつかった fibonacci 数の計算と本質的に同じ計算であることがわかる．本質的には同じ計算だが，プログラムの書き方によって,「プログラム言語処理系にとっては違う計算」になる，というのが本問のポイントである.

2-e. 説明 末尾再帰とは，関数の再帰呼び出しがすべて「計算の末尾」であること，つまり，再帰呼び出しを行って返ってきた値を用いて更に計算する必要がないことである．

2-e. 解答例 処理系が末尾再帰を処理する際に，関数呼び出しに対応するスタックフレームを新たにスタックに積むのではなく，現在のスタックフレームに上書きする (現在のスタックフレームを壊す) ことにより，実現できる．これにより，スタックの使用量を抑えることができ，再帰呼び出しをループと同様に処理することが可能となる．

2-f. 上記のプログラムにおける関数 f と g の型を述べよ．ただし，データ a, b が A という型を持つとき，対 (a,b) は，$A * A$ という型を持つ．(加点問題: 関数 h の型がわかるなら、それも述べると加点する。)

2-f. 解答例 関数 f は，整数 n を受け取り，整数を返す関数なので，`int -> int` という型を持つ．

関数 g は，整数 n を受け取り，整数 2 つのペアを返す関数なので，`int -> (int * int)` という型を持つ．(型の表記において、括弧を省略してもよい。)

関数 h は，整数 n を受け取ると，`fun p -> ...` という関数を返す．その返される関数は，`(1,1)` という引数を受けとっていることからわかるように，引数は `int * int` 型である．また，それが返す値は (ある場合には) `p` であり，`p` は仮引数であるので `int * int` 型であった．よって，まとめると，関数 h は，`int -> ((int*int) -> (int*int))` という型を持つ．(型の表記において、括弧を省略してもよい。)

問 3. (配点 25 点) 次のプログラムは，Java 言語で 3 つのクラス Parent, Child, Test を定義したものである (実際には，3 つは，それぞれ別のファイルに記述されている．)

```
// Parent.java
class Parent {
  public String method1 () {
    return "Parent";
  }
  Parent () {
  }
}
// Test.java
class Test {
  public static void main(String args[]) {
    Parent p = new Child();
    System.out.println(p.method1());
    method2(p);
  }
  public static void method2(Parent q) {
    System.out.println("No.1: " + q.method1());
  }
  public static void method2(Child q) {
    System.out.println("No.2: " + q.method1());
  }
}
```

```
// Child.java
class Child extends Parent {
  public String method1 () {
    return "Child";
  }
  Child () {
  }
}
```

Java では，extends というキーワードで継承を表す．よって、Child は Parent の子クラスである。System.out.println

は文字列を標準出力へ出力する．`p.method1()` は，変数 p に格納されているオブジェクトに対して、method1 というメソッドを呼び出す。

3-a. 上記プログラム (Test の main) を実行すると，何が出力されるかを、簡単な説明をつけて書きなさい.

3-a. 解答例 実際に走らせてみると，以下の出力を得る.

```
Child
No.1: Child
```

説明: Parent 型の変数 p に，Child クラスのオブジェクトを代入し，p の method1 を起動すると，Child クラスのメソッドが起動される (動的ルックアップ) ため，Child と表示される.

一方， p を引数として Test クラスの持つ method2 を起動すると，p の型によりメソッドが選定される (overloading) ため，No. 1 と表示される.

3-b. この例をもとに，動的ルックアップとは何か 2-3 行で説明しなさい.

3-b. 解答例 オブジェクト指向言語におけるメソッド名とメソッド実装の対応付けは，静的ではなく動的に行われること．すなわち，p.method1() の形でメソッド呼び出しをするとき，p の (静的に決まる) 型 (クラス) によって method1 として呼びされるメソッドが決まるのではなく，実行時に，p に格納されているオブジェクトの型 (クラス) によって決まる

3-c. この例をもとに，サブタイピングとは何か 2-3 行で説明しなさい.

3-c. 解答例 オブジェクト指向言語における，サブタイピングは，親クラスの変数 p に，子クラスのオブジェクト (new Child() で生成されたオブジェクト) を代入できる，また，親クラスのオブジェクトを仮引数とするメソッド M に，子クラスのオブジェクトを実引数とすることができる，といった機能のことである. (一般に，親子間だけではなく，先祖と子孫の間にも適用される．)

サブタイピングは，片方向的であり，通常，子クラスの変数に，親クラスのオブジェクトは代入できない (代入を許す言語はあるが，型がおかしくなってしまう危険がある).

問 4. (配点 15 点) 以下の事項から 3 つを選び、それぞれ 3-5 行程度で説明しなさい。

4-a. プログラムの静的情報と動的情報について。

4-b. 高階関数と関数クロージャについて。

4-c. 静的型付けと動的型付けについて。

4-d. メモリ管理におけるゴミ集めについて。

4-e. 抽象データ型とオブジェクト指向について。

解答例

4-a. プログラムの静的情報とは，プログラムを実行する前に (コンパイルする言語であれば，コンパイル時に) 利用できる情報であり，動的情報とは，プログラムの実行中に利用できる情報である．後者は前者を包含する．コンパイラは静的情報を最大限に活用して，高速に実行可能なコードを生成する．(補足:例として，静的束縛/動的束縛，静的型システム/動的型システムなどに言及してもよい．)

4-b. 高階関数は，関数を引数として受けとったり関数を返す関数のことであり，関数型言語では標準的に利用可能である．高階関数の処理のためには，実行時に関数をデータと見なして処理をする必要があり，静的束縛の言語では，関数本体と環境をセットにした「関数クロージャ」というデータ構造が用いられる．

4-c. 静的型付けとは，型の整合性の検査 (場合によっては型推論を含む) をプログラムの実行前に行うことであり，動的型付けは，それをプログラムの実行時に行うことである．静的型付けの言語は，コンパイルする言語であり，コンパイル時に型付けのエラーを発見することから，プログラムの信頼性が高い．一方，動的型付けの言語は，型エラーの早期発見の点では劣るが，型の制約によらない自由なプログラムの記述が可能であるため，用途によっては，書きやすく生産性の高いプログラミングが可能である．

補足: 本問の典型的な誤答として「静的型付けは、変数の型を宣言するもので、動的型付けは、変数の型を宣言しない」というものがあった。この誤答では、ML (OCaml) や Haskell は、動的型付けに分類されてしまう。

4-d. ヒープ領域にあるデータは，スタックと違い，プログラムの実行が進んでもそのまま生き残るため，空き領域がいずれ不足する．そこで，プログラム中のどこからも参照されないデータ (二度と利用されることのないデータ) の領域を探して回収する必要があり，これをゴミ集めと言う．C 言語ではゴミ集めはプログラマが行う必要があるが，Lisp, Java, ML などの言語では，処理系がゴミ集めを自動的に行う仕組みが備わっており，プログラマはヒープ領域の管理の仕事から解放される．

4-e. 抽象データ型は，データ型をどう構成するか (実装) ではなく，どう使われるか (仕様) を記述するという考えであり，実装と仕様を分離することにより，多人数で開発する大規模プログラミングを可能とした．すなわち，データ型の仕様を明確にすることにより，他の部分に影響することなく実装を変更することができる．オブジェクト指向言語のオブジェクトは，外部に見せるインタフェースと，見せない実装の分離，つまり，情報のカプセル化 (情報の隠蔽) という点では，抽象データ型と共通する特徴を持つ．

以上.

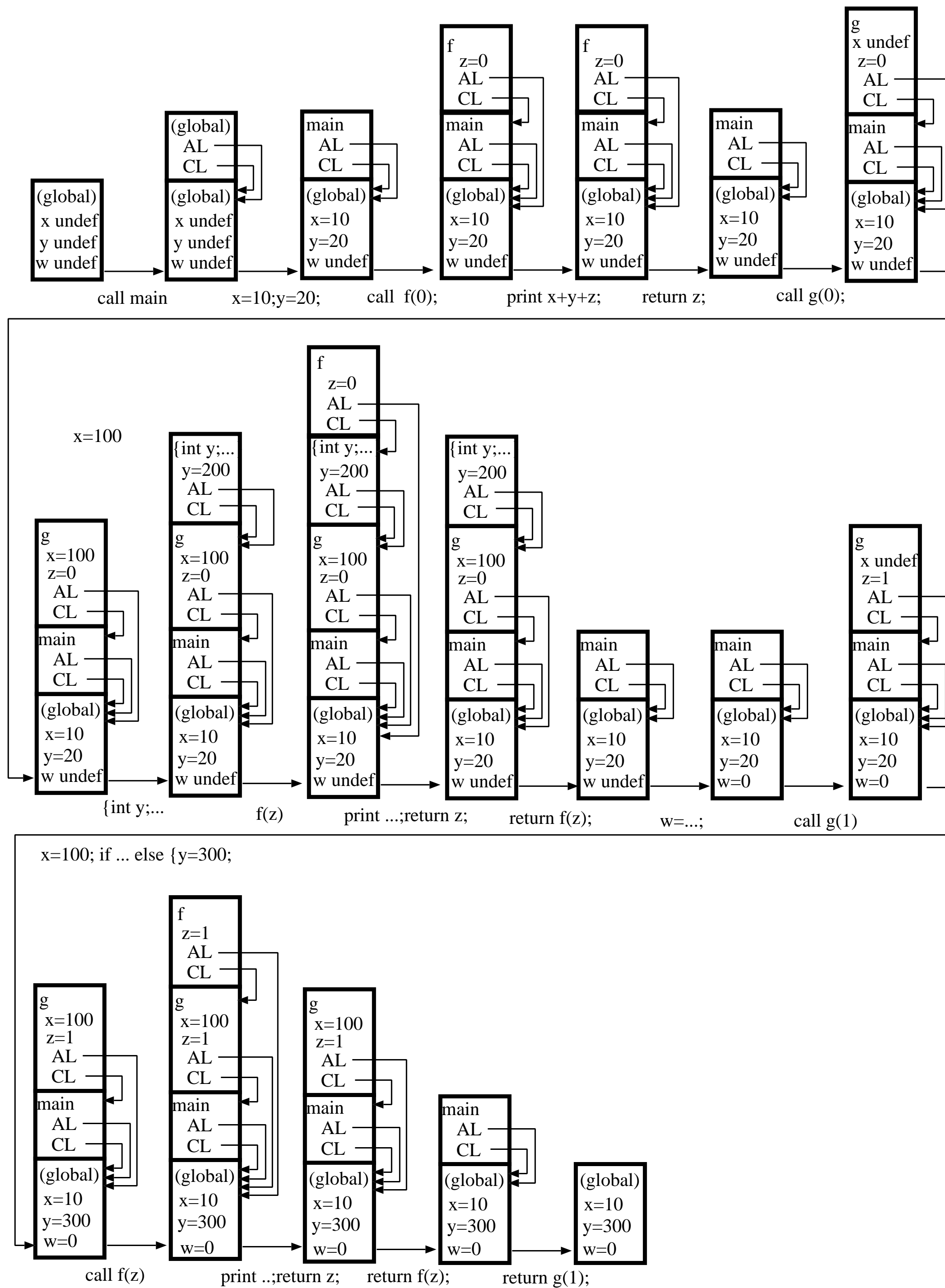

図 1: (1-e) 実行に伴うスタックの変化

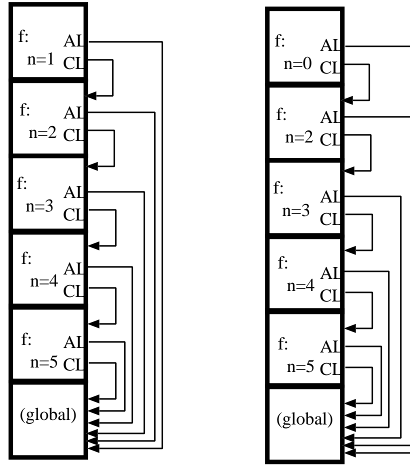

図 2: (2-a) スタックが最も深くなった瞬間

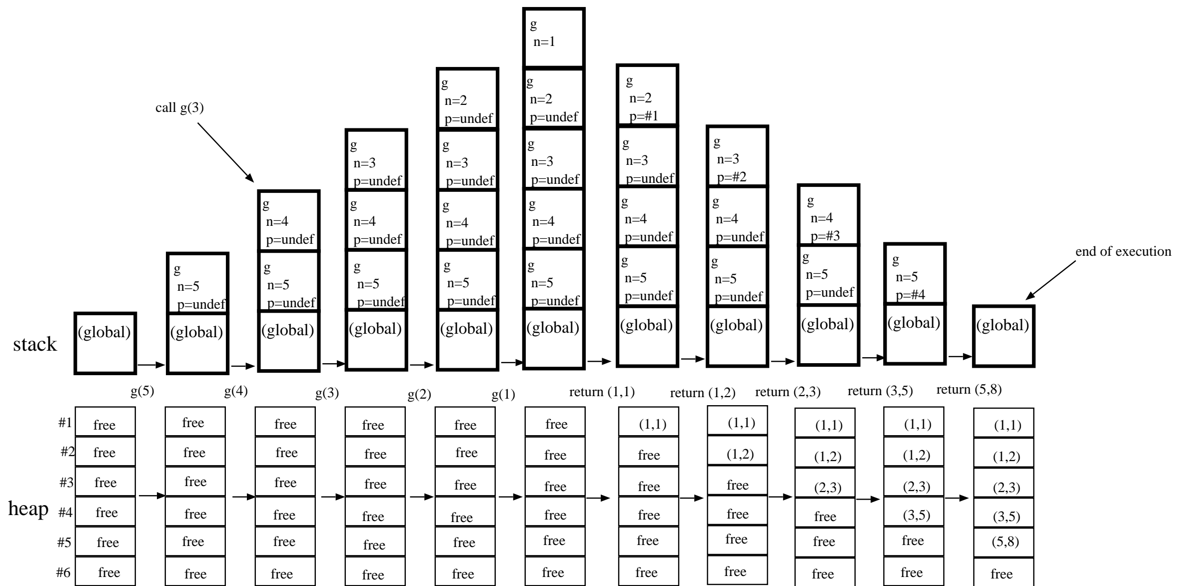

図 3: (2-b) 素朴な fibonacci プログラムの実行