『プログラム言語論』 期末試験 と解答例
2011 年 6 月 27 日

問 1. (配点 28 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．ただし、MiniC 言語の `print e;` という式は、C 言語の `printf("%d\n", e);` という式と同じ意味である。

```
int x;          /* 大域変数の宣言 */
int f (int y, int z) {
  int x;
  x = 200 + y + y;
  x = x + g(3);
  return (x);
}
int g (int u) {
  print (x + u);
  return (x + u);
}
int main () {
  x = 100;
  print (f(g(1),g(2)));
}
```

問 1-1.(配点 5 点) 上記のプログラムを、静的束縛かつ値呼び方式で実行する。ただし、複数の引数を持つ関数呼び出しでは、引数は左から右に評価される。この時、どのような値が印刷されるか (印刷されるものを全て、印刷される順番に) 答えなさい。

答 1-1. 以下の通りに印刷される。(補足: 採点にあたって改行やスペースの具合いは問わない。従って、以下の 4 行を 1 行につめこんで解答しても構わない。)

```
101
102
103
505
```

問 1-2.(配点 8 点) 前問の実行の開始から終了までの間に、スタックがどう変化するか図示しなさい。ただし、実行開始時点では、大域変数を格納したフレーム 1 つだけがスタックにつまれ、関数呼び出しごとにフレーム 1 つがつまれる。1 つのフレームには、(1) 局所変数たちの値、(2) Access Link, (3) Control Link の 3 種類のデータを書けばよい。(main に対応するフレームでは (1) はなく、大域変数を格納するフレームでは、(2), (3) はない。)

答 1-2. 図 1 の通り。(ただし AL は Access Link, CL は Control Link をあらわし、リンク先のスタックフレームへの矢印で表現した。また、式を実行してもスタックの状況がかわらない場合は、適宜省略した。)

採点のポイント: (1) 「関数呼び出し」が 1 回起きるごとにスタックフレームが 1 つ積まれ、その関数呼び出しが終了するごとにスタックフレームが 1 つはずされること、(2) CL は、必ず「そのスタックフレームより 1 つ下の (1 つ前に積まれた) スタックフレーム」を指すこと、(3) AL は、C 言語の場合、必ず「大域変数に対応するスタックフレーム」を指すこと、の 3 点である。

この図は、かなり丁寧に書いたものである. 実際の解答では、話の本筋に関係ない部分は適宜省略してもよい。

問 1-3. (配点 1 つ 5 点、合計 15 点) 上記のプログラムを、以下の方式で実行したとき、それぞれ、どのような値が印刷されるか答えなさい (スタックを図示する必要はない)。

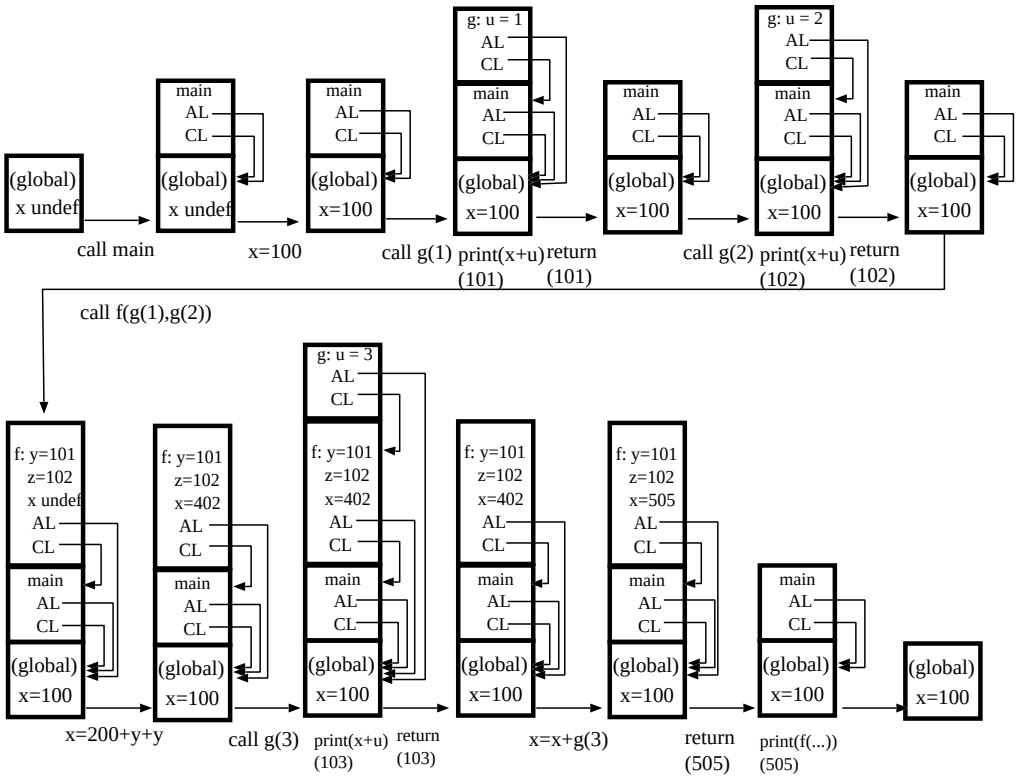

図 1: (1-2) 実行に伴うスタックの変化

a. 静的束縛かつ名前呼び方式で実行した時

b. 静的束縛かつ必要呼び方式で実行した時

c. 動的束縛かつ値呼び方式で実行した時 (複数の引数を持つ関数呼び出しでは引数は左から右に評価)

答 1-3. 以下の通り (スペースの節約のため、改行コードを印刷するかわりにスペース 1 つを印刷することにした。)

```
(a)   101 101 103 505
(b)   101 103 505
(c)   101 102 405 807
```

補足. 問 1-2 と問 1-3-a の違いは、値呼び方式ならば、関数呼び出しの前に g(1),g(2) が計算されるが、名前呼び方式ならば、関数呼び出しの後に、g(1) や g(2) の値が使われるごとに、計算される点である。この場合、f の中で g(1) は 2 回使われる (y が 2 回使用される) ので、101 が 2 回印刷され、f の中で g(2) は使われない (z が使用されない) ので、102 は印刷されない。

補足. 問 1-3-a と問 1-3-b の違いは、名前呼び方式と必要呼び方式の違い。両者は非常によく似ているが、必要呼び方式では、1 度「y」(つまり、g(1)) を計算すると、その値を覚えておいて、次に f の中で y が使われるときは、g(1) の計算はしない。そのため、y が 2 回使われているのに、101 は 1 回しか印刷されない。

補足. 問 1-2 と問 1-3-c の違いは、静的束縛と動的束縛の違い。ここで問題になるのは、変数 x のスコープであり、関数 g の中で使われている x が大域変数の x か、f の中で宣言された x なのか、ということである。

問 2. (配点 20 点) 次の図は、MiniC と OCaml で同じ意味を持つプログラムである。ただし、OCaml プログラム (右側) における `(x-1,x*y)` や `(5,1)` などは、対 (ペア) のデータ構造を表し、`print_int` は整数値を印刷する関数である。

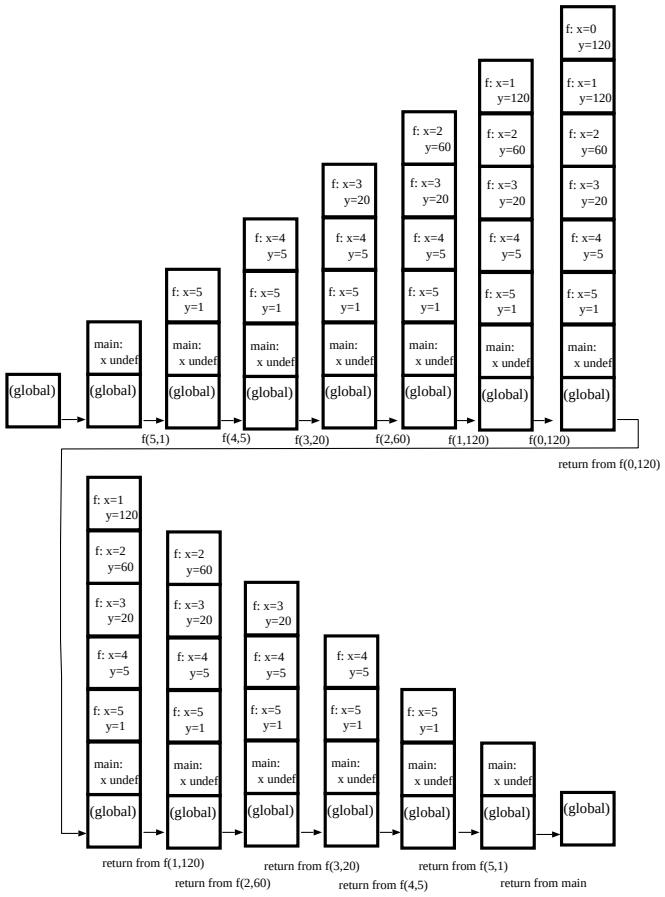

図 2: (2-1) 実行に伴うスタックの変化

```
int f (int x, int y) {
  if (x == 0)
    return (y);
  else
    return (f(x-1,x*y));
}
int main () {
  int x;
  print (f(5,1));
}
```

```
let rec f (x,y) =
  if x = 0 then
    y
  else
    f (x-1,x*y)
in
  print_int (f (5,1))
;;
```

問 2-1.(配点 5 点) 上記の MiniC プログラム (左側) を実行して、印刷される値を求めなさい。ただし、実行の途中経過 (スタックの状態の変化) も書くこと。

答 2-1. 図 2 の通りの計算で、120 が印刷される。

問 2-2.(配点 5 点) OCaml 言語の (通常使われる) 処理系では、末尾呼び出しに関する最適化が行われる。この事を念頭に置いて、上記の OCaml プログラムを実行した時のスタックの状態の変化を書きなさい。

答 2-2. 末尾呼び出しによる最適化が行われたとき、図 3 の通りの計算が行われる。(参考までに、スタックだけでなくヒープ領域がどのように変化するかの一例も付記したが、答案ではヒープの変化を図示する必要はない。)

補足: すべての関数呼び出しが末尾呼び出しの場合には、関数呼び出しの処理において、「スタックフレームを

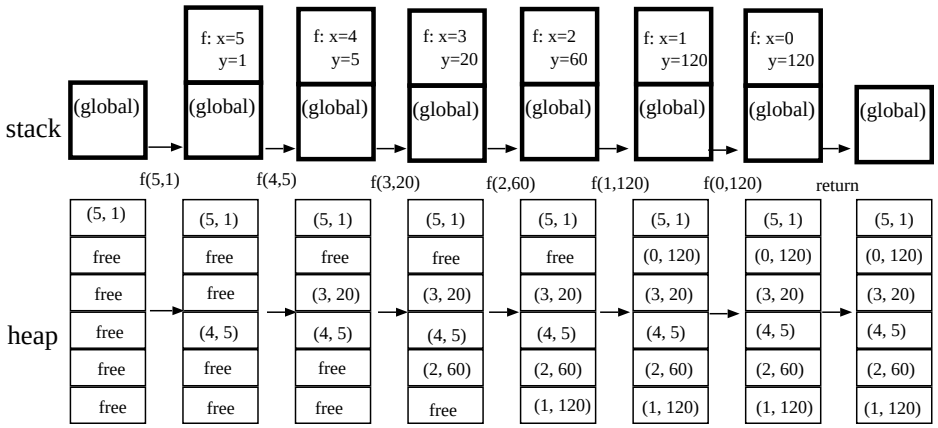

図 3: (2-2) 実行に伴うスタックの変化 (ヒープの変化も付記)

新たに積む」かわりに、「現在の (スタックの top にある) スタックフレームの場所に、新しいスタックフレームを書き込む」という処理とすることができる。(現在のこのスタックフレームは上書きされてしまうので、なくなる。) この処理をした場合、関数呼び出しが終わったとき、「スタックフレームを 1 つ外す」という操作はしないようにする。(最初にスタックフレームを積んでいないので、終了時点でもスタックフレームは外さない。)

問 2-3.(配点 5 点) 上記の OCaml プログラムの実行終了時点でのヒープ領域の状態 (どのような値が書き混まれているか) を書きなさい。ただし、実行開始時はヒープに値はなく、実行中にごみ集めは行われなかったものとする。

答 2-3. 実行終了時点では、以下の 6 つの対が、ヒープ中のどこかに書き込まれている。

(5,1), (4,5), (3,20), (2,60), (1,120), (0,120)

(補足) これらの 6 つの対が、ヒープのどこに書き込まれているか、その順番はどうか、などは、処理系に依存するし、また、同じ処理系を使って実行していても処理するごとに異なる場所に書きこまれるかもしれない。そのあたりは問題ではなく、ともかく、「対を生成するごとに、ヒープに書きこまれる」ということがわかっているかどうかを尋ねる問題である。ヒープがどうなっているかの、一例は、前問の答 (図 3) の一番右側に記載した。

問 2-4.(配点 5 点) C 言語におけるヒープ領域の管理方法と、OCaml 言語や Java 言語におけるヒープ領域の管理方法の違いを簡潔に (2-3 行で) 述べよ。

答 2-4. (ここでは答えの材料を示すため、2-3 行を越えて記述するが、答案では 2-3 行でよい。)

C 言語では、ヒープ領域の管理を処理系が行わず、プログラマが行なう。すなわち、プログラム中に、明示的に malloc 関数 (ヒープ領域の割り当て) や free 関数 (ヒープ領域の解放) を記述する必要がある。一方、Lisp,OCaml,Java などの言語では、ヒープ領域の管理は処理系の仕事であり、オブジェクトや関数クロージャなどのデータ構造を生成すると、自動的にヒープ領域の中のあいている部分を使ってデータが書きこまれ、それらのデータが使用されなくなると、ごみ集めにより自動的にヒープから消去される、という仕組みが備わっている。

前者は、ヒープ管理の重荷がプログラマに帰着されるという欠点があるが、システムプログラミングなどにおいて、プログラマ自身が精密にメモリを管理したい場合には適している。後者は、プログラマは、ヒープ管理に頭を悩ますことがなくなるため、他の仕事 (アルゴリズムの設計等) に集中できるという利点がある。

問 3. (配点 20 点) OCaml 言語で関数 f と g と h を以下のように定義する。

```
let f x = x + 1 ;;      (* たとえば f 10 の計算結果は 11 *)
let g x = (x, x);;      (* たとえば g 1.1 の計算結果は (1.1, 1.1) *)
let h x y = x + y;;     (* たとえば h 10 20 の計算結果は 30 *)
```

問 3-1.(配点 1 つ 6 点、合計 12 点) 関数 f と関数 h の型を答えなさい。

答 3-1. 関数 f について: f の定義が成立するためには (エラーなく f を定義できるためには) f の型が整合しなければいけない。x+1 という部分があるので、 x は整数型の変数である。また、f が返す値は x+1 なので、これも整数型である。よって、関数 f は、`int -> int` という型を持つ。

関数 h も同様に、整数型の x と整数型の y をもらい、整数型の x+y を返す。ここで、授業中に注意したように、関数 h は「2 個の引数をもらって、1 個の値を返す関数」ではなく、Curry 化された関数定義であるので、「1 個の引数をもらって、「1 個の引数をもらって 1 個の値を返す関数」を返す関数」である。よって、h の型は `int -> (int -> int)` である。ここで、括弧は省略してもよい。つまり、`int -> int -> int` でも正解である。(なお、h の型について、このように面倒に考えなくても、int 型の値を 2 つ続けてもらって、int 型の値を返すので、`int -> int -> int` と考えてよい。)

(補足： なぜか g の型を答えていた人がいた。また、f の型として `int` と答えていた人がいて、それは授業をやった立場としては非常に残念であった。)

問 3-2. (配点 8 点) 関数 map は以下の例のように実行される。(map の定義は省略する。) このとき、map が多相型を持つことを説明した上で、一般に (関数 map に限らず) 多相型を持つことの利点を簡潔に (2-3 行で) 述べなさい。

```
map f [100; 200; 300];;   ==> [101; 201; 301]
map g [1.4; 1.7; 2.0];;   ==> [(1.4,1.4); (1.7,1.7); (2.0,2.0)]
```

答 3-2. map 関数の 1 つ目の使用例では、`int -> int` の関数 f と `int list` 型の値をもらって、`int list` 型の値を返している。一方、map 関数の 2 つ目の使用例では、`float -> (float * float)` の関数 g と `float list` 型の値をもらって、`(float * float) list` 型の値を返している。(なお、授業では、float 型や、(float * float) 型などの詳しい話はしていないので、これらの型を正確に書けなくてよい。大事なのは、これらが、int 側でない、と認識することである。)

これらの事実により、map 関数は、それが使われる状況によって、`(int -> int) -> (int list) -> (int list)` 型になったり、`(float -> (float * float)) -> (float list) -> ((float * float) list)` 型になったりする。このように、複数の型を持つことができる関数を多相型を持つ関数と呼び、式や項の型として多相型を許す言語を多相的な言語と言う。

多相的な言語では、1 つのプログラムを様々な型に適用して使うことができるため、多相型のない言語で、上記と同じことを記述した場合に比べて、プログラムのディバッグや再利用の際に有利である。また、1 つのコードを様々に使うことができるので、(コンパイルされた後の) コード量が小さい、コンパイル時間が短いなどの利点がある。

(補足: 本問で扱っている多相は、いわゆる Parametric Polymorphism である。)

問 4. (配点 32 点, 1 問あたり 8 点, 5 個以降答えた時は良い方から 4 つ採用) 以下の事項から、4 問を選んで、それぞれ 3-5 行程度で説明しなさい。

問 4-1. 大規模プログラミングに対処するためのデータの抽象化とは何か、その利点は何か。

答 4-1. データの抽象化の代表例である抽象データ型について説明する。通常の (具体的な) データ型が、そのデータ型を具体的にどう構成するかを記述するのに対して、抽象データ型では、そのデータ型をどう使うか (どのような関数で、そのデータ型を使うか) のみを記述する、という違いがある。抽象データ型は、そのデータ型の「インタフェース」のみを記述しており、「実装」には言及しないので、インタフェースを変更せずに実装方法を変更することが可能となり、プログラミングを多人数で分業することが容易となる。

(追加するとすれば) (1) 抽象データ型のアイディアのうち、「インタフェースと実装の分離」あるいは、「情報の隠蔽」の部分は、オブジェクト指向言語の「オブジェクト」に受け継がれており、今日の大規模プログラミングへの最も重要な対処法の 1 つとなっている。(2) たとえばスタックの抽象データ型は、インタフェース関数とし

て、スタックの生成関数、push 関数, pop 関数, top 関数、空スタックかどうかの判定関数などを用意し、それらの引数の個数や型は明示するが、これらの関数が具体的にどう実現されているかは記述しない。これにより、スタック型を使うプログラムに影響することなく、スタック型の実装を変更することができる。

問 4-2. インタープリタとコンパイラの違いは何か、また、通常、コンパイラを使って得たプログラムの方がインタープリタより高速に実行されるのはなぜか。

答 4-2. インタープリタは解釈系 (実行系) であり、プログラムと入力データを受け取り、それを実行した結果を返すプログラムである。コンパイラは翻訳系 (変換系) であり、プログラムを受け取り、それを、より高速に実行できる形式のプログラム (多くの場合は機械語のプログラム) に変換するプログラムである。

インタープリタが (内部で) コンパイラを呼んでいることもあるので、必ずしも「コンパイラで得たプログラムが、インタープリタより高速である」とは言えないが、多くの場合、それが成立する。なぜなら、コンパイラは、プログラム (あるいはファイル) 全体を読み込んで、型情報やプログラムの性質に関する種々の静的情報を収集・解析して最適化を行い、高速なコードを生成しているからである。一方、インタープリタは、多くの場合、プログラム全体を解析しないで実行するため、静的情報に基づく高速化をはかることができない。

問 4-3. 静的型システムと動的型システムの違い、および、それぞれの利点は何か。

答 4-3. 静的型システムは、プログラムを実行する前 (すなわちコンパイル時に) 型の整合性をチェックできる型システムである。一方、動的型システムは、型の概念は存在するが、実行前には型の整合性をチェックせず、実行時に型の整合性をチェックする型システムのことである。前者は、C, Java, OCaml など、後者は Lisp/Scheme, Ruby, JavaScript などが代表例である。

利点としては、前者は、実行前に型に基づくエラーを発見してディバッグできるため、プログラムの信頼性が高い点である。また、コンパイラが型情報を用いて最適化を行なうことがあり、この場合、生成されたコードが高速に実行できる。後者の利点は、厳密に型付けしにくいプログラム (比較的複雑な処理をするオブジェクト指向プログラムなど) において、プログラムを (部分的にでも) 早く完成させやすく、また、早いタイミングで試すことができる、という点があげられる。

(補足) 「静的型システム」は、「静的・型システム (static type system)」すなわち、「型システムのなかで、静的なもの」という意味であって、「静的型・システム」ではない。ここを間違えている答案は、残念ながら、問題の意味をわかっていなかったということになる。

問 4-4. 関数クロージャ(クロージャ) とは何か、関数型言語ではなぜそのようなものが必要か。

答 4-4. 関数クロージャは、関数を通常のデータの一種として持つ言語 (高階関数を持つ言語) で、かつ、静的束縛を採用している場合に、その言語の処理系が必要とするものであり、「計算結果としての関数」をあらわす。具体的には、「関数の定義と環境の対」として表現される。静的束縛のもとで、自由変数を持つ関数を処理するとき、関数を定義した時の環境 (あるいはスタック) を保存しておき、その関数を使うときには、保存しておいて環境を復元する必要がある。このため、関数クロージャの形で環境を保持しておく。

(追加するとしたら) `let x=10 in fun y -> x + y` という式を OCaml で評価すると、式の値として返ってくるのは、関数クロージャであり、「`fun y -> x + y` と「`x=10` という環境」の組」である。

(補足) Lisp 言語の生みの親である John McCarthy さんの、Lisp のマニュアルにのっているインタープリタでは、関数クロージャは使われておらず、従って、初期の Lisp は動的束縛であった。つまり、静的束縛であるラムダ計算とは異なる言語であった。

問 4-5. メソッド (オブジェクト指向言語におけるメソッド) と、関数 (C 言語や ML 言語における関数) はどう違うか。

答 4-5. メソッドは、オブジェクトごとに (Java など、クラスを持つ言語ではクラスごとに) 定められたインタフェースであり、関数と同様に、引数を持ち何らかの作用を起こして値を返す。

しかし、C 言語や ML 言語の関数と、オブジェクト指向言語のメソッドが決定的に異なるのは、メソッドでは、「その名前」と「実体」の対応付けが動的に決定されることである。すなわち、プログラム中に abc というメソッド呼び出しがあっても、その実体が何であるかは実行時に決定される。(そのメソッド呼び出しを受けるオブジェ

クトが持っている abc というメソッドが何か、ということで決まる。) 一方、C や ML では、関数名とその実体の対応付けは、静的に (コンパイル時に) 決まる。

(補足) メソッド名とその実体の対応付けが実行時に決まる仕組みを dynamic lookup と言う。

問 4-6. 静的に決められる情報 (静的データ) と静的に決められない情報 (動的データ) の違いは何か。(具体例を 1 つずつ以上あげること)

答 4-6. 静的データは、プログラムの実行前に決定できる情報であり、たとえば、(1) C 言語や Java 言語で、型が整合しているかどうか、(2) 宣言されたが、プログラム中に出現しない変数があるかどうか、(3) 呼ばれているが定義が存在しない関数はないか、などである。一方、動的データは、プログラムの実行前に決定できない情報 (ただし、実行時には決定できる情報) であり、たとえば、(4) プログラムを実行したときに使用するスタックのサイズの上限, (5) 変数は、使われる前に必ず初期化されているか (初期値が与えられているか)、(6) Ruby で、プログラムの型が整合しているかどうか、(7) C 言語で、配列へのアクセスの際に、その添字が上限と下限の間にはいっているかどうか、などである。性能の良いコンパイラは、静的情報をなるべく多く (あるいは、なるべく高速に) 集めて、それを利用して、より高速なコードを生成している。

以上.