『プログラム言語論』 期末試験 の解答例
2010.07.01, 亀山

問 1. プログラム言語 L における式 (expression) の構文は，以下の BNF で定義される．ただし，e が式に対応し，f と g は補助的なものである。

$$e ::= f \mid e@f$$
$$f ::= g \mid g\#f$$
$$g ::= 0 \mid 1$$

問 1-1. (5 点) 言語 L における式 0@1#0 の構文木 (parse tree) を書きなさい。

答 1-1. 図 1 の構文木となる。

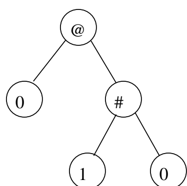

図 1: 問 1-1 の構文木

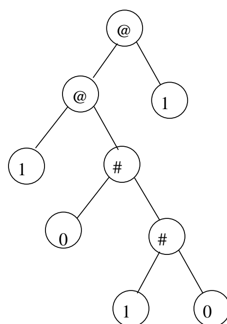

図 2: 問 1-2 の構文木

補足. 0@1#0 が e クラスに属することを、詳細に示すと以下のようになる。

1. 1 は g クラスに属する。

2. 0 は g クラスに属する。

3. g は f クラスに含まれるので、2 を使って、0 は f クラスに属する。

4. $g\#f$ の形は f クラスに含まれるので、1,3 を使って、1#0 は f クラスに属する。

5. f は e クラスに含まれるので、3 を使って、0 は e クラスに属する。

6. $e@f$ は e クラスに含まれるので、5,4 を使って、0@1#0 は e クラスに属する。

そこで、これを木の形で表現すると、図 1 となる。

なお、上記以外の構文木には対応しない．たとえば，式 0@1#0 を「式 0@1 と 0 を # でつなげたもの」と考えようと思っても無理である．なぜなら,「0@1」は e クラスのみに属する (f や g には属さない) が，# を含む構文定義では,「$g\#f$」となっていて，# の左には g クラスのものが来ないといけないからである.

問 1-2. (5 点) 言語 L における式 1@0#1#0@1 の構文木 (parse tree) を書きなさい。

答 1-2. 式 1@0#1#0@1 の構文木は図 2 で与えられる。

補足. 前問と同様に考えればよい。式 1@0#1#0@1 についても，構文木は一意的である．なお、構文の定義を見ると、# は @ より強く結合し、# は右結合的、@ は左結合的であることがわかる。

問 1-3. (5 点) この式の定義に、括弧でくくった式 (e) を追加したい。たとえば、(0@1)#1 が式になるようにしたい。このためには、上記文法をどう変更すればよいか。簡単な理由を付けて答えよ。

答 1-3. $(0@1)\#1$ が式になるようにするためには、$\#$ が含まれるのが、$g\#f$ の形だけであるため、(e) の形のものを g クラスに加える必要がある。そこで、解答例としては、以下のようになる。

$$e ::= f \mid e@f$$
$$f ::= g \mid g\#f$$
$$g ::= 0 \mid 1 \mid (e)$$

問 2. 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;                          /* 大域変数の宣言 */
int g (int u) {
  x = x + 1;
  print x;                      /* x を計算して、その結果を印刷する。 */
  return (x);
}
int f (int y, int z) {
  int x;
  x = 10 + y;
  return (g(0));
}
int main () {
  x = 5;
  print f(g(0), g(0));          /* f(...) を計算して、その結果を印刷する。 */
}
```

問 2-1. (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した時、どのような値が印刷されるか答えなさい。ただし、関数に複数の引数がある時は、左から右に評価される。

答 2-1. 順に、6, 7, 8, 8 が印刷される。

補足. 静的束縛であるので、関数 g の本体にある変数 x は (x が 4 回出現するが、その全ての出現は) この x を含むスコープを持つものであり、それは、グローバル変数 (大域変数) の x である。すなわち、関数 f から関数 g が呼ばれたときであっても、g における x は常にグローバル変数を意味する。この事がわかれば、あとは、普通に頭の中で実行してみればよい。

問 2-2. (5 点) 前問の実行の間に、スタックがどう変化するか図示しなさい。ただし、関数呼び出しごとに、スタックフレームを 1 つスタックに積むものとする。

答 2-2. 解答例を図 3 に掲げた。(ここでは、非常に丁寧に、1 つの文を実行するごとにスタックの変化を書いたが、要点さえ書かれていれば答案としては問題ない.)

なお，太い四角で囲まれているスタックフレームは,「現在見えている (Access Link をたどって到達できる) スタックフレーム」であり，細い四角で囲まれているものは,「現在見えていない (Access Link をたどって到達できない) スタックフレーム」である.

補足. 図 3 のポイントは，2 列目の右端での `x=x+1` という代入文における x が，どのスタックフレーム中のものか，ということであり，図に示したように，f のスタックフレームではなくグローバル変数の x であることがポイントである.

問 2-3. (前後半で 5 点ずつ、合計 10 点) 上記のプログラムを、動的束縛かつ値呼び方式、また、静的束縛かつ名前呼び方式で実行した時、それぞれ、どのような値が印刷されるか答えなさい。

答 2-3:前半. 「動的かつ値呼び」の場合、6, 7, 17, 17 が印刷される。

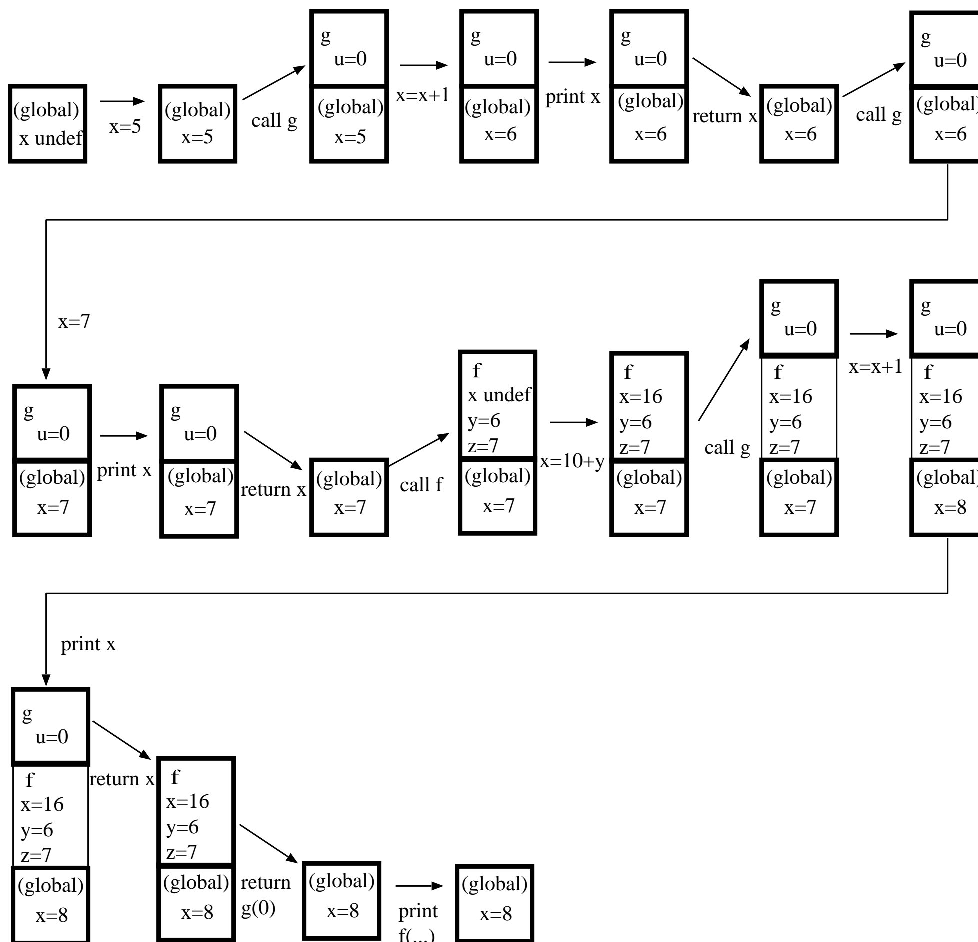

図 3: 答 2-2. 実行に伴うスタックの変化

補足. 動的束縛では、関数 `g` の本体にある変数 `x` が指すものが、`g` が呼ばれる毎に違う可能性がある。このプログラムでは、`g` が、関数 `main` から呼ばれるときと、関数 `f` から呼ばれるときの 2 通りがあり、前者では `x` はグローバル変数だが、後者では `f` でローカルに宣言されている `x` を指す。(図 3 において，すべてのスタックフレームを「太い四角」で囲った状態を思い浮かべるとよい．)

答 2-3:後半. 静的束縛かつ名前呼び方式の場合は、`6`，`7`，`7` が印刷される。

なお，この問題は，試験の範囲を逸脱していたので，100 点満点の枠外とした．(解けたらボーナス点を与え，解けなくても他が全部でてきたら 100 点満点となるよう配点を調整しなおした．理由は後述．)

補足. 実行の経過を詳細に追ってみよう。

`f(g(0)，g(0))` という関数呼び出しの瞬間には、引数の `g(0)` は評価されず、`y=g(0)` と `z=g(0)` という変数束縛がスタックに格納される。その後、`f` の本体で `x=10+y;` が実行されるとき，`y` に束縛された `g(0)` が呼ばれる。その呼び出しで、グローバル変数 `x` の値が `6` になり、この値が印刷される。この結果，`f` の 3 行目の `x=10+y;` を実行後に、ローカルな `x` の値は `16` になる。

次に，`f` の 4 行目の `return(g(0));` で、`g(0)` を呼び出すので、グローバル変数 `x` の値が `7` になり、この値が印刷され、その値が返る。(さらに、`f(g(0),g(0))` 全体の返り値となる。)

最後に、`main` 関数で、返り値 `7` が印刷される。

補足の補足「名前呼び方式」については、miniML 処理系でのみ説明しており、miniC 言語の演習では名前呼び方式はやっていなかった。miniML は「代入文」がないので，厳密にいえば、本問のように「代入文がある言語

で，名前呼び方式がどうなるか」は授業・演習の範囲を越えていた．そこで，本問を除いて 100 点満点とし、本問を含めると 105 点満点になるよう配点を調整した。(できた人にはボーナス点が与えられ、できなくても、他が完璧なら満点を取れる、という意味である。)

問 2-4. (5 点) C, Java, ML など、現在使われているプログラム言語の多くが、静的束縛を採用している。これらがコンパイラにより処理される言語であることを念頭に置いて、その理由を説明しなさい。

解答例 2-4. 静的束縛では、参照されている変数に対応する変数宣言がどれであるかがコンパイル時に (静的に) 決定できる．このため、「変数の値の参照 (dereference)」の処理において,「対応する変数がどこに格納されているかを探す」という操作が不要となり，高速化がはかれる．一方，動的束縛では、これらの対応関係がコンパイル時に決まらないため、変数参照の高速化ができない。この理由により、コンパイラによる高速な実行コードの生成を前提としたプログラム言語の多くが、静的束縛を採用している。

問 2-5.(5 点) 必要呼び方式 (call by need) について、値呼びや名前呼びと比較しながら、その特徴を説明しなさい。

答 2-5. 必要呼び方式とは、関数呼び出しにおける引数 (実引数) を、実際にその値が必要になったときに初めて評価する方式であり、さらに、同じ引数を 2 回以上評価しないよう、1 度評価した引数の値を記憶しておく方式である。値呼び方式では、引数の値が実際には不要な場合でも評価をしてしまう。また、名前呼び方式では、そのようなことはないが、同じ引数を 2 回以上使うときはその回数だけ、同じ計算を行ってしまう。いずれのケースでも、必要呼び方式の方が、引数の評価回数は少なくて済むという利点がある。

補足. この解答例では、「必要呼び方式が最高の呼び出し方式」と言っているように聞こえてしまうが、この方式の実装自体がやや複雑になるため、実際のプログラムの処理が値呼びや名前呼びより、いつでも高速になるわけではない。また、必要呼び方式の場合、引数がどのタイミングで評価されるのか直感的にわかりにくいので、ディバッグがやや難しいという問題もあり、これらは研究課題である。

問 3. 次の MiniML プログラムについて，以下の問に答えなさい．

```
let rec fib n =
    if n = 0 then 1
    else if n = 1 then 1
    else (fib (n+(-2))) + (fib (n+(-1)))
in fib 5 ;;
```

問 3-1.(5 点) 上記のプログラムを計算して、返す値を求めなさい。ただし、実行の途中経過 (関数 `fib` の呼び出しにより、スタックがどのように変化するか) も記すこと。

答 3-1. 簡略化したスタックの変化図を、図 4 に掲げる。この実行により `8` が返ることがわかる。

上記のプログラムを高速化するため、対 (pair) のデータ構造を用いて、以下のプログラムを書いた。

```
let rec fib2 n =
    if n = 0 then (1, 1)
    else
       let x = fib2 (n+(-1)) in
         (snd x,  (fst x) + (snd x))
in fib2 5 ;;
```

問 3-2. (5 点) 上記の `fib2` のプログラムを計算して、最終的に返す値を書きなさい．(途中経過を書く必要はない。)

解 3-2. 前問と同様に実行すると、`(8,13)` が返る。

問 3-3. (5 点) 上記の 2 つのプログラムにおいて、関数 `fib` および関数 `fib2` がそれぞれ何回呼ばれるか、答えなさい。

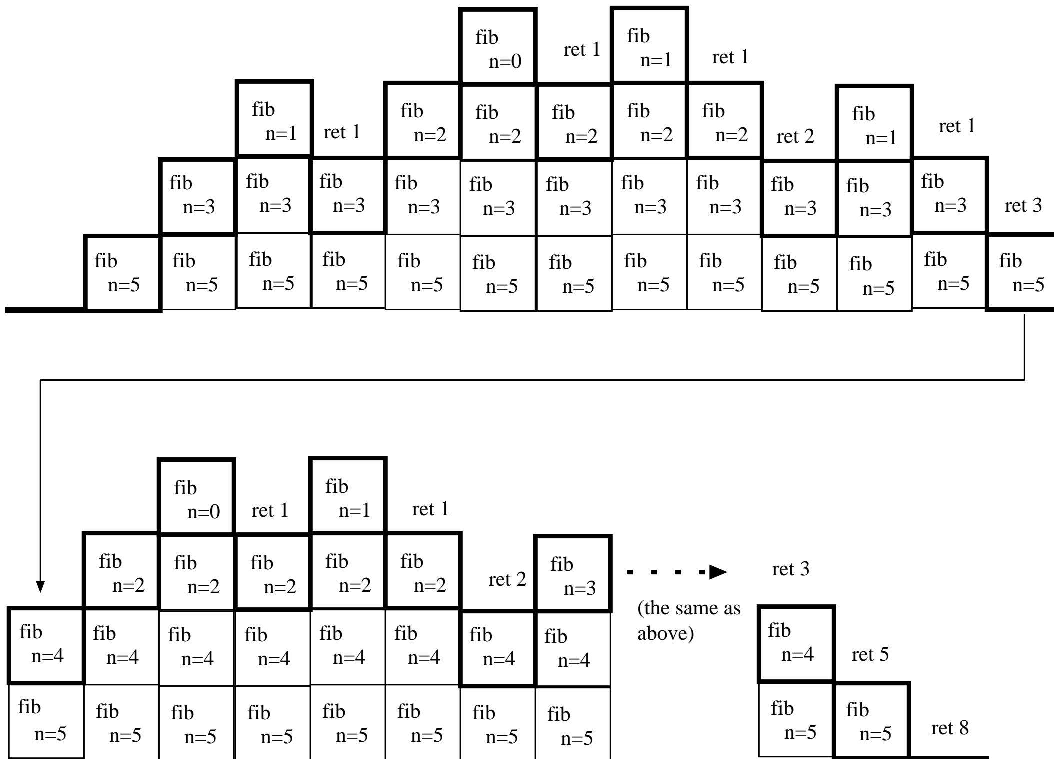

図 4: 問 3-1 のスタック

答 3-3: 前半. `fib` は 15 回よばれる。

補足. これは地道に計算してもよいが、系統的に計算することもできる。`fib(n+2)` の計算で呼ばれる `fib` の回数は、`fib(n)` のそれと、`fib(n+1)` のそれに 1 を加えたものである。

よって、`fib(0)` と `fib(1)` の計算では、`fib` が 1 回だけ呼ばれ、`fib(2)` の計算では、`fib` が 3 回呼ばれ、`fib(3)` の計算では、`fib` が 5 回呼ばれ、`fib(4)` の計算では、`fib` が 9 回呼ばれ、`fib(5)` の計算では、`fib` が 15 回呼ばれる。

答 3-3: 後半. `fib2` は 6 回よばれる。

問 3-4. (5 点) 2 つのプログラムの効率の差が生じる理由を述べなさい。

答 3-4. `fib 5` の計算では、`fib 3` が 2 回行れるなど同じ計算が何度も実行されるが、`fib2 5` の計算では、無駄な計算が行われないため。

補足. 「`fib2` が末尾呼び出しであるため効率が良い」と書いた答案がいくつかあった。去年もそうだったが、`fib2` は末尾呼び出しではないので、これは正解にはできない。(`fib2` から `fib2` を呼びだしたあと、`snd` 等の計算をしているので、再帰呼び出しが「末尾」になっていない。)

問 3-5. (5 点) 上記の 2 番目のプログラムは、実行時にヒープ領域を用いる。このようなプログラムを利用する場合に必要な「ごみ集め (garbage collection)」の概要について 3 行程度で述べなさい。

答 3-5. ごみ集めとは、ヒープ領域に、関数クロージャ、オブジェクトなどのデータ構造を展開するプログラム言語の処理系が行う処理であり、今後利用されることのないデータを見分け，そのデータが格納されているヒープ領域を開放して、その領域をその後再利用できるようにするための処理である。

問 4.

問 4-1.(7 点) モジュールと (オブジェクト指向言語における) オブジェクトに共通する特徴の 1 つは、情報隠蔽 (information hiding) である。情報隠蔽の概要と、どのような意味で有用か、について、述べなさい。

答 4-1. モジュールやオブジェクトなどにおける情報隠蔽とは、「インタフェース」と「実装」を分離し、利用者 (そのモジュールやオブジェクトを利用しているプログラム) にはインタフェースのみを公開し、実装は公開しないことを指す。通常、インタフェースには、モジュールやオブジェクトに含まれる関数 (メソッド) の名前やその引数と返り値の型などの情報を含む。

情報隠蔽により、実装を変更しても、利用するプログラムには変更をする必要はなくなるという利点があり、大規模プログラムの開発における分業が容易になる。

問 4-2.(7 点) オブジェクト指向言語における継承とは何か、どのような場面で有用であるか、例をあげて説明しなさい。

答 4-2. 継承とは、既に定義したオブジェクトの定義を利用して、新しいオブジェクトを定義することである。クラスに基づいたオブジェクト指向言語では、継承は、既に定義されたクラスの定義を、新しいクラス定義に含める仕組みとして実現される。継承の仕組みにより、同じコード (プログラム) を 2 回以上書かずに済ませることが可能になり、コードのディバッグ、保守、再利用がやりやすくなる、という利点がある。

補足. 上記の解答例では、わざわざ「クラスに基づいたオブジェクト指向言語では」と断った。これは、オブジェクト指向言語には、クラスという概念をもたないものもあるからである。しかし、今回の授業で扱ったオブジェクト指向言語はクラスに基づいているものだけであるため、皆さんの答案の中で、そのようなことを断る必要はない。

問 4-3. (7 点) オブジェクト指向言語におけるメソッド (method) と、C 言語や ML 言語における関数とは、よく似た機能であるが、相異点もある。これらの間の相異点について、「動的検索 (dynamic lookup)」というキーワードを用いて説明しなさい。

答 4-3. C や ML では、関数名から関数定義を探す操作は (変数の場合と同様) 静的であり、コンパイル時にどの関数であるかが決定できるので、効率の良いコードが生成できる。一方、オブジェクト指向言語では、メソッド名から、そのメソッドの定義を探す操作は動的検索により行われる。すなわち、コンパイル時には、メソッドの本体が何であるかは決定できず、実行時に (メソッドが呼ばれるごとに) 検索する。これは効率面では不利であるが、実行時に動的にメソッドを選択することにより、柔軟なプログラミングが可能になるという利点がある。

問 5. (1 組あたり 7 点、2 組で 14 点)

以下のキーワードの組から 2 組を選んで、それぞれの組を 5 行程度 (合計 10 行程度) で説明しなさい。

(1) (プログラム言語における) 静的情報と動的情報

答. 静的情報とは、プログラムを実行せずに (実行前に) わかる情報であり、動的情報とは、プログラムを実行しないといわからない情報である。C, Java, ML などの言語では、「変数参照と変数宣言の関係」や「式の型」などが、静的情報であり、コンパイラはこれらの情報を用いて効率良いコードを生成することができる。たとえば、`x + y` という式では `x` と `y` の型が整数型か浮動小数点型などの実数の型かによって `+` に対応する機械語のコードが異なるが、その判断をコンパイラ時に行うことができる。一方、Lisp や Ruby などの言語では、「型」は動的であり、たとえば、`x + y` という式では `+`がどのような演算であるかは、実行時に `x` と `y` の値が与えられなければ判断できない。

補足. どの情報が静的でどの情報が動的かについて、絶対的な区別があるわけではない。たとえば、`(x*2 == 1)` という式は `x` が整数値である限り偽 (C 言語では `0`) になるが、このことは、静的な解析を頑張れば、(実行しなくても) わかる。それどころか、現代のコンパイラの多くのものは、`1+2*3` のように、すぐに計算できてしまう式はさっさと計算してしまい、その結果を用いて静的な解析を行ない、効率のよいコードを生成している。

ただ、そうはいっても、`f` が複雑な計算をする関数の時、`(f(0)==1)` という式がどういう値を持つかは、コンパイル時には計算できない。やはり、静的と動的の差はあるのである。

(2) 命令型言語と宣言型言語

答. 命令型言語とは、「プログラム＝命令の列」という考え方に基づくプログラム言語であり、C 言語など非常に多くのプログラム言語が含まれる。命令型言語では、実行時にコンピュータがやるべき事 (命令) が、その手順

に従って具体的に記述されたものがプログラムとなる。

一方、宣言型言語とは、「プログラム＝宣言の列」という考え方に基づくプログラム言語であり、Prolog 言語などが含まれる。宣言型言語では、(原則としては) コンピュータの処理手順がプログラムとして記述されるのではなく、「論理式 (Prolog の場合) 等が、このような性質を満たすべきという宣言」の集まりがプログラムである。プログラマは処理の手順を記述するのではなく、いわば「解くべき問題 (の仕様)」を書くだけでよいので、特に、バックトラックによる探索などを要する問題では、比較的早くプログラミングができる、という利点がある。欠点としては、処理手順を一切書かない方式では、効率良いプログラムが書けないことが多い、等があげられる。

補足. ML や Lisp などの関数型言語も、一応「宣言型言語」に分類される。関数が「こういう仕様を満たすべき」という性質 (宣言) のみを書いているという風に、一応、みなせるからである。ただし、ML や Lisp でも、「変数への値の代入」などの仕組みを使えるように拡張されており、これらを使いはじめると、もはや「宣言としての関数」とはいえなくなり、命令型言語におけるプログラミングスタイルに近くなる。これらの話題について詳しくは、2 学期の南出先生の「宣言型プログラム論」を参照のこと。

(3) インタープリタとコンパイラ

答. インタープリタは、プログラムとそれに対する入力を受けとり、プログラムを解釈しながら実行し、出力を返すプログラムである。コンパイラは、プログラムを受け取り、それを別の言語 (機械語、あるいは、仮想機械の言語など) で記述されたプログラムを返すプログラムである。

インタープリタは通常、プログラムの静的情報に基づく解析を行わないため、実行速度が速くないことが多い。一方、コンパイラは、プログラムの静的情報を利用して解析し、最適化を行ったプログラムを生成するため、コンパイラにより生成されたプログラムの実行速度は、高速であることが多い。

(4) 型検査と型推論

答. 型検査は、式とその型付けを入力とし、その型付けが成功するかどうかを返すことである。一方、型推論は、式のみ、あるいは、式と部分的な型付けを入力とし、型付けが成功するような型を推論することである。

C や Java のように、変数宣言や関数宣言において型を明示する言語では、コンパイル時に型検査が行われる。一方、ML のように、変数や関数の型をプログラム中で明示する必要がない言語では、型推論が行われる。前者のメリットは、プログラム中に型を明示した方が理解しやすくなる等であり、後者のメリットは、プログラム記述量が減りプログラマにとっての生産性があがる、等である。