# 『プログラム言語論』 期末試験 解答例
# 2009年6月29日

**問1 (構文: 20点)**

あるプログラム言語 $L$ において，式の構文は，以下の BNF で定義される．ただし，$e$ が「式」に対応し，$f$ は補助的にのみ用いられる．

$$e ::= f \mid e@f$$
$$f ::= 0 \mid 1 \mid f\&f$$

**問 1-1 (5点)** この言語で，0&1 は式であるかどうか判定しなさい．式の時はその構文木 (parse tree) を書き，式でない場合はその理由を述べなさい．

**答.** 0&1 は式である。その理由: 0 と 1 は $f$ クラスに属す。$f\&f$ の形のものは $f$ クラスに属すので、0&1 は $f$ クラスに属す。$f$ の形は $e$ クラスに属するので、0&1 は $e$ クラスに属す。

「構文木」の書き方はいろいろあり得るが、代表的なものとして図??の左側のものをあげておく。

**問 1-2 (5点)** 0@1&1@0&0 に対して、前問と同じことをしなさい。

**答.** 0@1&1@0&0 も式である。その理由: 前問と同様にして、1&1 と 0&0 が $f$ クラスに属すことがわかる。また、0 は $f$ クラスに属し、よって $e$ クラスにも属すので、0@1&1 は $e$ クラスに属す。よって、0@1&1@1&1 は $e$ クラスに属す。

構文木は図??の右側にあげる。

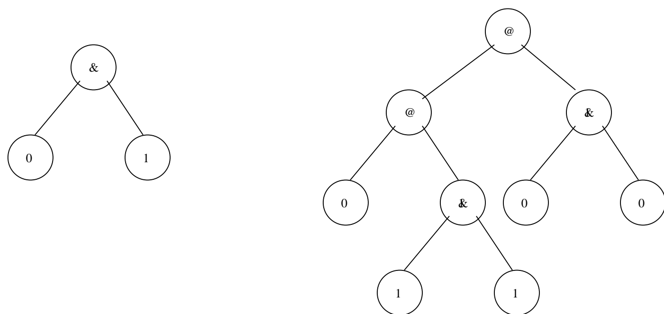

図 1: 問 1-1 と問 1-2 の構文木

**補足.** 上記のように理由を書いていなくても、構文木が書けていればよい。また、構文木の書き方はいろいろな流儀があるので、構造がきちんと書けていれば良いことにした。ただし、本問の場合、構文木は1通りしかないので、他の構造を持つ構文木を解答した場合は、間違いである。

**問 1-3 (5点)** 1つの表現が，複数の構文木に対応するとき，その構文定義を曖昧という．上記の構文が曖昧であるかどうかをその理由とともに書きなさい．

**答.** 曖昧である。なぜなら、0&1&1 という表現は、図??の2通りの構文木に対応するので。

**補足.** ここでは、曖昧であることの理由を具体的に示せなければ減点した。

ちなみに、この文法で曖昧さをなくすためには、たとえば、以下の定義にすればよい。

$$e ::= f \mid e@f$$
$$f ::= 0 \mid 1 \mid f\&0 \mid f\&1$$

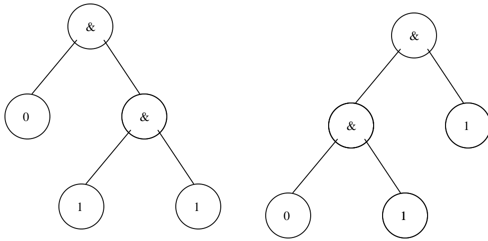


図 2: 問 1-3 の構文木

この場合、@ と & は、いずれも、左結合的 (left associative) になる。

**問 1-4 (5 点)** 上記の構文に、「かっこ」を追加して、(0@1)&(1&0) のような表現が「式」となるようにしたい。上記の BNF をどのように拡張すればよいか?

**答.** 「かっこをつけた式」に、さらに & 等をつけたいのであるから (& をつけるには $f$ クラスまで戻らないといけないので)、「$e$ クラスのものにかっこをつけると、$f$ クラスにもどる」ようにすればよい。すなわち：

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

**補足.** 上記の正答以外の可能性として、

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

というものもありそうだが、これでは (0@1) という表現が、式にならない。同様に以下のものでも、うまくいかない。

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

その他にもいろいろなバリエーションがあるが、採点基準としては、(0@1)&(1&0) が式になるかどうか、で判定した。(どんなに下手な文法でも、これが式になるなら、5 点を与えた。)

**問 2 (変数の束縛: 20 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 g (int y) {
  if (y == 1+1+1)
     return x;
  else {
     x = x + y;
     return g(y+1);
  }
}
```

```
int f () {
  int x;
  x = 10;
  return (g(0));
}
int main () {
  x = 0;
  print f();
}
```

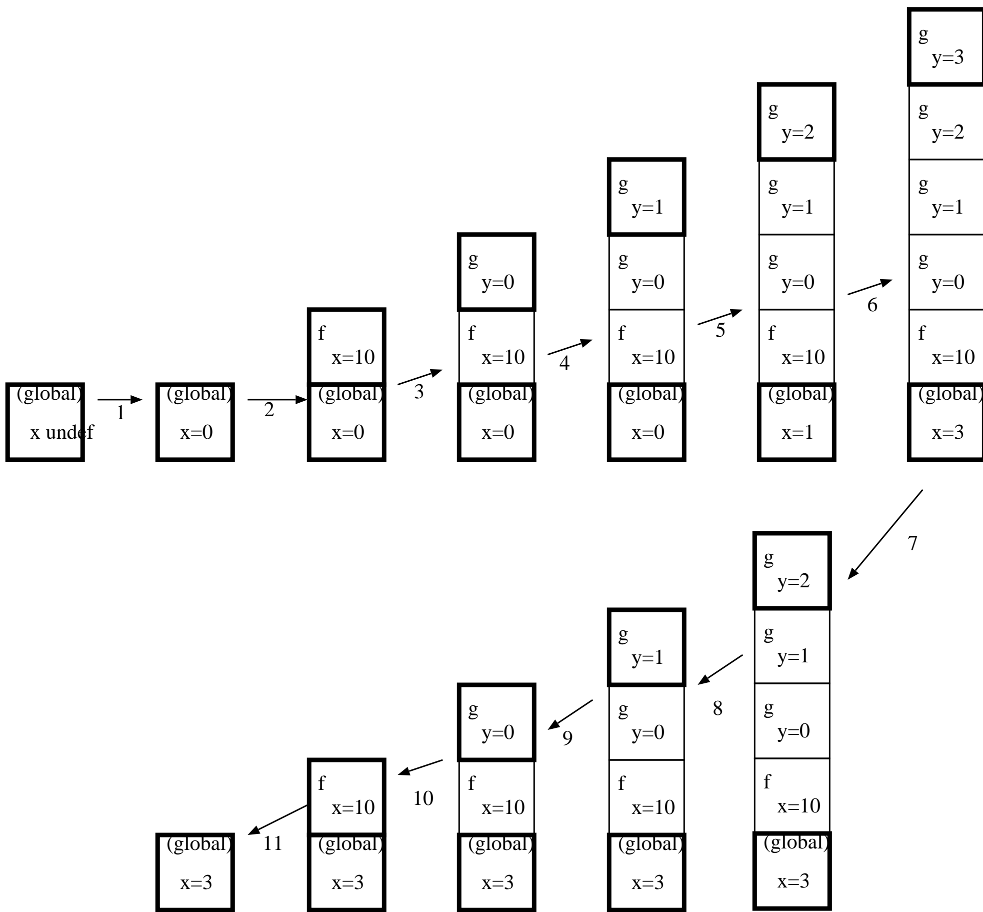


図 3: 問 2-1 のスタックの変化

**問 2-1 (15 点)** このプログラムの実行開始から終了まで，スタックの内容がどう変化するかを図示せよ。ただし、標準的なモード (静的束縛，値呼び) での実行とする。

**答.** 図?? に、スタックの変化を図示する。

ただし、ここでは、以下の流儀で表示した。

- 1 つのスタックフレームを四角で表示。わかりやすさのために、どの関数呼び出しに対応するスタックフレームかも書いた。
- スタックのトップが上の方向。
- 「スタックトップから Control Link をたどって行けるスタックフレーム」を太い枠で囲った。

**補足 1.** 言うまでもなく、スタックをこのように表示しなければいけない、という理由はないので、これと違う形で書いていても、スタックの変化として理解できるものは、なるべく好意的に解釈して採点した。授業中の演習で使った処理系では、show コマンドで表示されるのは、Control Link でたどって行けるスタックフレームだけだったので、図?? における太枠で囲ったフレームだけが表示される。本問では、そのあたりは特に指定しなかったので、それでも正解である。

また、大域変数 (関数の外で定義されている変数、ここでは $x$ のこと) は、MiniC 処理系ではスタックに積むが、通常の処理系では、スタックには積まれないので、図?? のスタックの一番下にあるフレームはスタック上にはないことが多い。したがって、そのような絵を描いても正解とした。

**補足 2.** 授業では一瞬説明しただけであるが、「tail call (末尾呼び出し) に関する最適化」を行っている処理系の場合、「関数 g の再帰呼び出し」において、新たにスタックフレームを積むことはなく、既にあるスタックフレームを再利用する。(あるいは、既にあるスタックフレームを捨ててから、新たに 1 つ積む。)

MiniC 言語では、このような最適化はおこなっていないが、ML や Scheme などの関数型言語の処理系では、この最適化を行っている。その場合のスタックの変化を図??に示した。(こちらの流儀で図示しても正解である。)

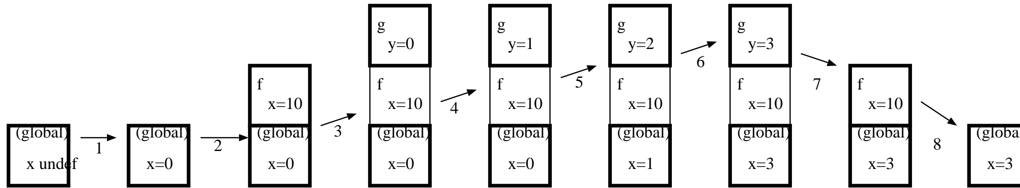


図 4: 問 2-1 のスタックの変化 (末尾再帰の最適化をする処理系)

**補足 3.** 本問は、演習でかなり時間を費してやった課題そのものであるので、ほとんどの人が解答できると思ったのだが、残念ながら完全な正解者は少なかった。

多かった間違い: 関数 g は局所的に x という変数を宣言していないので、関数 g が呼ばれるたびにスタックにつまれるスタックフレームには、変数 x のためのメモリ領域はとられない、という点を見落としたものである。(従って、上記の図にあるように、g のためのスタックフレームは、局所変数としては、y のみを含む)。これは、プログラムの 1 行目で、大域変数として宣言されている x が、g の中で宣言されていると勘違いしたものであろう。変数のスコープ (有効範囲) は、C 言語などのブロック構造言語の最も基本的に大事な要素の 1 つなので、是非復習してほしい。(私のプログラムの書きかたが、いまひとつ、わかりにくかったせいかもしれない。)

他の間違い: ほかにも、静的束縛がわかっておらず、関数 g の中で値が更新される変数 x が、どの x のことか、対応がわかっていない答案が結構多かった。

完璧な答えが少なかったので、少し採点基準を甘くして、関数呼び出しによってスタックフレームがつまれる、ということがわかっているかどうか、等により部分点をなるべく与えるようにした。

**問 2-2 (5 点)** 現在の多くのプログラム言語は，静的束縛を採用している．その理由を簡潔に述べよ．

**答.** いくつかの理由が考えられるので、代表的な理由 2 つを以下にあげる。どちらか 1 つでも書いてあればよい。

(理由 1) 静的束縛では、変数の「宣言」と「使用 (参照)」の関係 (この変数は、どこで宣言されたか、という情報) が、プログラムを書いた時に決まる (実行時に決まるのではない) ため、プログラムが人間にとってわかりやすくなる。

(理由 2) 現代のプログラム言語の処理系はコンパイラが主流であるが、変数の宣言と使用の関係が、コンパイラによって静的に決められるため、効率的なコードを生成することができる。

**補足.** 当初は、「理由 2」のみを正答とするつもりであったが、「理由 1」を書いてある答案を見て「なるほど」と思い、そちらを書いていても正答とすることにした。

**問 3 (関数呼出し: 20 点)**

次の MiniML プログラムについて，以下の問に答えなさい．

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

**問 3-1 (5 点)** 上記のプログラムが返す値 (return value) を書きなさい．

**答.** 単純に計算すれば、$f(5) = 8$ となることがわかる。(簡単な問題なので、答えさえ書けばよい。)

**問 3-2 (5 点)** このプログラムの実行中に，スタックに積まれるスタックフレームの個数 (関数呼出しの深さ) の最大値を答えなさい.

**答.** この問題のポイントは、関数呼び出しの「深さ」とは何かということである。これを考えるために、$f(5)$ の実行途中で $f(2)$ を計算している時点以降のスタックの状態を観察する。

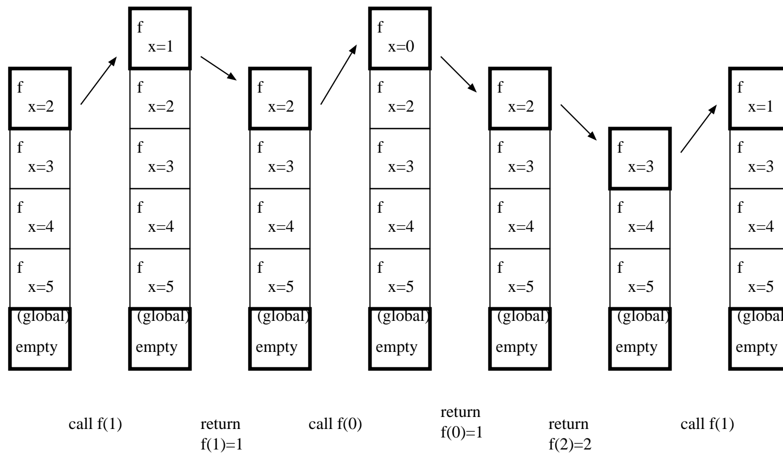


図 5: 問 3-2 のスタックの変化

図**??**からわかるように、$f(2) = f(1) + f(0)$ という計算においては、$f(1)$ という関数呼出しに対応するスタックフレームが積まれ、その呼び出しが終了してから (スタックフレームがまきもどされてから)、次の関数呼び出し $f(0)$ が起きている。要するに、$f(1) + f(0)$ のような形の関数呼び出しにおいて、$f(1)$ と $f(0)$ の呼び出しは、「入れ子 (nest)」ではなく、「1 つ呼ばれて、その呼び出しから戻ってきてから 2 つ目が呼ばれる」という対等な関係にある。従って、関数呼び出しの「深さ」を計算する場合、これらの深さの「和」に 1 を足すのではなく、これらの深さの「大きい方」に 1 を足すべきである。

このことを念頭に起くと、$f(5)$ の計算で、最も関数呼び出しが「深い」瞬間は、図**??** における、左から 2,4 番目のときとなる。すなわち、ネストの深さの最大値は「5」が正解である。ただし、大域変数のためのスタックフレームをスタックに積む処理系では (世の中にはそういう処理系は多くないと思われるが、MiniC の処理系ではそうやっている)、スタックの一番下に、大域変数用のスタックフレームがつまれるので、「6」となる。本問では、それでも「正解」としたので、結局「5」あるいは「6」と書いてあれば、正解である。

**問 3-3 (5 点)** 上記のプログラムは，効率が良くない．効率を改善するにはどうすればよいか。もし可能ならば MiniML で改善したプログラムを書き、そうでなければ、改善のアイディアを書きなさい。

**答.** 上記の計算からわかるように、$f(0)$ や $f(1)$ の計算を何度も繰返しており、これは無駄である。

改善のための 1 つの方法は、$f(0)$ や $f(1)$ などの値を 1 度計算したら記憶しておき、2 回目以降は、その記憶した値を利用する、というものである。(ただし、「既に計算した値を記憶する」ことは、MiniML の範囲では簡単ではない。)

別の方法: この問題をよく考えると、「$f(n)$ の値を計算するためには、その前の 2 個の値 $f(n-2)$ と $f(n-1)$ を知ればよい」ということがわかるので、関数呼出しのたびに、これら 2 つの値を持ち歩くことが考えられる。

**補足.** 本問では、改善したものをプログラムとして実装することは求めていないが、参考までに、上記のうちの後者の方針で、OCaml で実装したものを掲げる。

(OCaml 言語での解)

```
let rec f1_sub x i prev1 prev2 =
  if x <= i then prev2
```

```
    else f1_sub x (i+1) prev2 (prev1 + prev2)
  in  f1 5 2 1 1 ;;

  let f1 x = f1_sub x 2 1 1;;
```

「入れ子になった関数」を使うと、もう少しきれいになる。

```
let f2 x =
  let rec visit i prev1 prev2 =
    if x <= i then prev2
    else visit (i+1) prev2 (prev1 + prev2)
  in visit 2 1 1
in f2 5 ;;
```

(OCaml など、フルセットの ML 言語ではなく) MiniML では、「複数の引数を持つ関数」も「入れ子になった関数」も書けないので、上記のプログラムを実装するのは、ちょっと大変である。すなわち、上記の `f1_sub` の 4 つの引数を、「ペア」を 3 回使って、(x,(i,(prev1,prev2))) と表現すれば、1 つの引数として表現できるので、上記のプログラムを MiniML でも実装可能である。が、あまりにも汚ないプログラムになり、私のポリシーに反するので、ここには記さない。

本問の解答としては、プログラムまで書く必要はなく、「$f(n-2)$ と $f(n-1)$ の 2 つの値を記憶すればよい」という趣旨を書きさえすれば正解とした。

**補足.** 結構多くの人が、「末尾再帰に変形すれば効率化できる」と書いていた。これは、高級で良い答えのように見えるかもしれない。(このように答えた人は、おそらく、私の授業のやりかたや、本問の流れからして、スタックを節約すれば効率化できる、ということが言いたいのだろう、と深読みしてしまったためと思われる。そんな難しい推論を求めるつもりはなく、単なるプログラミング上のことを聞いただけだったのだが、思わぬ落とし穴になってしまった。)

実は、本問の場合、「末尾再帰に変形する」というのは、的外れであり、不正解となる。というのは、本問の関数 (fibonnacci 関数) を、末尾再帰の形になおすのは簡単ではないからである。fibonacci 関数では、関数 f の中で f を呼ぶ箇所が 2 箇所あるので、どちらかに着目して末尾再帰風に変更しても、もう 1 箇所の f の再帰呼び出しが残ってしまうのである。実は CPS 変換という、亀山の研究室の学生が研究しているプログラム変換を使うと、fibonacci 関数も末尾再帰に変形できることはできるのであるが、この場合、その代償として、「高階関数」をたくさん作るプログラムになってしまい、スタックを消費しなくなるかわりに、ヒープをたくさん消費してしまう。つまり、この変換をやって末尾再帰にしても、効率がよくなったとはいえない。

長々と書いてきたが、fibonacci 関数の効率化は、「過去に計算した値を覚えておく」というのが鍵であり、これを「メモ化」 (Memoization) という。

**問 3-4 (5 点)** ML など関数型プログラム言語の有利な点，不利な点を，C 言語と比較して，簡潔に (3 行以内で) 説明しなさい.

**答.** 関数型プログラム言語は、単一代入、(原則として) 副作用を持たないようにプログラムを書くこと、繰返しより再帰呼び出しを主として利用すること、高階関数が使えることなどが特徴であり、これらのうちからメリットを書けばよい。また、比較対象が C 言語であるので、データ構造が豊富で、かつ、ヒープ領域に対する「ごみ集め」の機能がある点もメリットにいれられる。(「ごみ集め」は Java 言語など多くの言語で採用されているので、関数型言語のみの特徴とは言えないが、ここでは、C 言語との比較なので、メリットにいれてもよい。)

メリットの正解例としては、「単一代入であり、副作用を持たないように記述するため、プログラムの意味が明快でわかりやすく、保守性がよいと考えられる。」「高階関数が利用できるため、より抽象度が高いプログラムを記述することができる。」「プログラマが自由にデータ構造 (データ型) を定義できること、および、ヒープに割り

当てられたデータに対するごみ集めの機能があるため、記号処理など、可変長データを処理するプログラムを書くのが非常に楽になる。」など。

ML 言語特有のメリットとしては、この他に「強力な型推論機構が備わっているため、プログラムの型を記述せずに、高いレベルの安全性が得られる。」「モジュール機構により、データ抽象化を行うことができ、大規模プログラミングに対応可能である。」など。

デメリットの正解例としては、「現在のコンピュータのハードウェア (特に CPU) に密着した言語ではないため、非常に高い実行性能を要求される局面では、C 言語より劣ることが多い。」「C 言語に比べるとユーザ数が多くないので、様々なライブラリやツール類の数が相対的に少ない。」など。

なお、授業ではまったく説明していないが、「ごみ集め」があるためのデメリットとして、「ヒープ上の全てのデータに、ごみ集めのための 1-2 ビット程度の印が必要になるため、高性能の数値計算などにおいて、そのビットの処理の時間がかかり、効率が落ちる。」ということがある。「ごみ集めのための必要な印」というのは、要するに、ごみを回収する際につけるマークである。この問題は、関数型言語だけでなくごみ集めをする言語の処理性能における非常に本質的な問題であり、解決方法については、今なお研究対象である。

**問 4 (実行時とコンパイル時: 20 点)**

以下の性質を，コンパイル時に決定できる情報 (C, compile time )，実行時にならないと決定できない情報 (R, run time)，どちらでも決定できない情報 (N, neither) に分類せよ．それぞれについて，ごく簡単に理由も述べよ．(理由は 1 行でよい．)

- (5 点) MiniC 言語のプログラムに対して，宣言された変数はすべて，式の中にあらわれるかどうか.

  **答.** 「C」... これは、授業で説明した通りであるが、プログラム (のテキスト、字面) を見れば、宣言された全ての変数が、実際に式の中にあらわれるかどうかはチェック可能であるので、「コンパイル時に決定できる情報」である。

- (5 点) MiniC 言語のプログラムに対して，宣言された変数はすべて，プログラム中で使われるかどうか.

  **答.** 「R」と「N」を両方正解とする。... 変数が「実際に使われるか」ということは、コンパイル時にはわからない。たとえば、関数の中で宣言されている変数は、その関数が 1 度も呼ばれなければ使われないし、`if (f(0)) {z=y;} else {z=0;}`という式の場合、`f(0)` の値によって、変数 y が使われるかどうかが変わるので、いずれにしても「C」ではあり得ない。

  だから「R」か「N」のいずれかであり、私の講義資料 10 週目のスライドでは、本問の答えは、「R」であると結論付けていた。ただ、よく考えてみると、これは、問題文の日本語があまり明確ではなく、「実行のどの時点でそれがわかるか」ということを明示しないと意味をなさない出題であった。

  たとえば、MiniC 言語で、$x = f(0); z = y;$ というプログラムがあったとき、$y$ が本当に使われるかどうかは、$f(0)$ の計算が終了するかどうかにかかっている。しかし、それは $f(0)$ の計算をするより前に一般的には決定できない。(プログラムの停止性を、有限時間内に判定するアルゴリズムは存在しないので)。従って、「実行中のある瞬間に、プログラム中のすべての変数がいずれは使われるかどうか判定せよ」という問題だとしたら、答えは「N」でなければならない。

  もし、「変数 $y$ が、実際に参照される直前の時点で、その変数が使われるかどうかを判定する」のでよければ、(当然ながら、いつでも YES であるので) 答えは、「R」である。

  このあたりについて、問題文が不明確であって、どちらともいえない問題になってしまっていたことをお詫びする。結論として、「R」と書いても「N」と書いても両方正解とする。(「C」でないことさえわかればよい。)

- (5 点) MiniC 言語や MiniML 言語のプログラムに対して，実行中に使用されるスタックの量.

  **答.** 「R」と「N」を両方正解とする。... コンパイル時には、関数の再帰呼出しが何回おこるかを決定することはできないので、「C」ではない。「R」であるか「N」であるかについては、前問と同様に、問題文において、「実行のどの時点で、どれだけの情報を必要とするか」ということが正確に決まらないと答えようがないので、両方正解とする。

- (5 点) MiniML 言語のプログラム (ペアを表すデータ構造を持つ) に対して，実行中に使用されるヒープの量.

  **答.** 「R」と「N」を両方正解とする。... 授業の演習課題にあったように、「ペアをどんどん (無駄に) 作りだすプログラム」がある。そこで、コンパイル時には、「ペア」のデータが最大いくつ作りだされるか決定することはできない、ということが容易にわかる。よって、「C」ではない。

  「R」であるか「N」であるかについては、前問と同様。

- (5 点) MiniC 言語や MiniML 言語のプログラムに対して，プログラムの型が整合的であるかどうか.

  **答.** これは、コンパイル時に決定できる情報であるので、「C」である。

  **補足.** 実際の MiniC や MiniML 処理系はコンパイラではないので、型検査や型推論をしていない。では、本問の答えは「R」かというと、そうではない。「その言語の処理系 (の 1 つ) が、型の整合性検査をやっていない」ということと、「プログラムが与えられたとき、型の整合性判定ができる」ということは、まったく別の問題である。前者は、処理系の性質であるのに対して、後者は、プログラム言語の性質であるので、混同しないように注意してほしい。

**問 5 (データ抽象: 20 点)**

データ抽象の手段として，オブジェクト指向とモジュールがある．これらの主な特徴と相異点を 7 行以内で要領よく説明せよ.

以下のキーワードを使ってもよいが、使う場合は必ずその意味を説明せよ。

キーワード: 情報隠蔽 (information hiding)，インタフェース (interface)，実装 (implementation)，型 (type)，継承 (inheritance)，サブタイピング (subtyping)，コードの再利用 (code reuse).

**解説.** これについては、授業でほとんどそのまま解説した話であるので、正解例については、資料 8-9 週を読み返してほしい。(特に 9 週目のスライドを参照。)

1 点だけ気をつけるべきことは、「両者の違いを説明せよ」という問題であるので、主要な違いである「動的」と「静的」という点について言及する必要がある。すなわち、オブジェクト指向言語では、4 大特徴の 1 つに「動的ルックアップ」があるが、(ML 言語などの) モジュールは、基本的には、(それに含まれる関数名から、関数の実装を取りだすときの) ルックアップ操作が静的である、という違いがある。この点には是非言及してほしかった。

採点にあたっては、なるべく部分点を拾うことにした。

以上。