『プログラム言語論』
Short Quiz (2012.05.28)

Short Quiz 1 (2 限に出題).

「OCaml などの関数型言語は単一代入 (single assignment) だ」と言われてもぴんと来ない人も多いだろう．なぜなら, `x=10; x=x+2; x=x*3; ...` といった C 言語の文の列は, `let x=10 in let x=x+2 in let x=x*3 in ...` という OCaml 言語の式、あるいは、`(let ((x 10)) (let ((x (+ x 2))) (let ((x (* x 3))) ...)))` という Lisp/Scheme の式で、あらわせているように思えるからである．

しかし，これは間違いであり,「OCaml や Lisp 言語の let 式では，C 言語の代入文のかわりにはならない (C 言語の代入文の使い方で、let 式ではシミュレートできないものがある)」のが真実である．その根拠 (具体的なプログラミング例あるいは言葉による説明など) を答えなさい.

解答例 C 言語の代入文 (assignment statement) は「状態を変更する」が、OCaml 言語の let 文は「新しいブロックを導入して、そのブロックでの局所変数の値を設定する」。

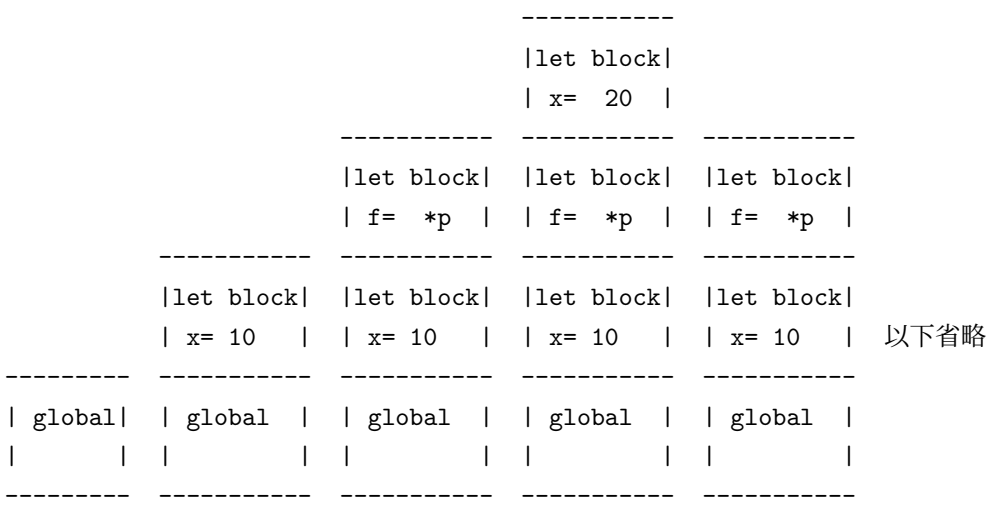
前者では可能で、後者ではできない例: 「x をプログラムの全体で (つまり、プログラムの冒頭で) 宣言されている大域変数としたとき、その値を x=x+1; と変更する関数」は C 言語では書けるが OCaml の関数では書けない。(ブロックを越えて、値を設定することはできない。)

Short Quiz 2. Short Quiz 2-1 (3 限に出題。。。したものを改変) 以下の OCaml プログラムを実行したときの, スタックの様子を図示しなさい。

```
let x = 10 in
  let f = fun y -> x + y in
    let x = 20 in
       f 30
```

解答例 実行 の進行にともない、スタックとヒープがどう変わるかを書く。

stack:



heap:
p->(fun y->x+y, x=10)

要するに、スタックの絵としては、関数クロージャがないときとほとんど同じである。(なので、short quiz である。) ただし、関数を評価すると関数クロージャができる点、また、関数クロージャ (`fun y->x+y, x=10`) は、スタック上ではなく、ヒープの中に取られる、という点だけが違う。

なお、関数クロージャの具体的表現は、授業では簡単に説明しただけだが、要するに「関数の定義 `fun y->x+y`」と「その関数クロージャを生成したときの変数の環境 `x=10`」を組みあわせたものである。(具体的に環境をどうやって作ったか、などは、関数型言語の実装の話なので、この授業ではカバーしていない。)

Short Quiz 2' (3 限に出題).

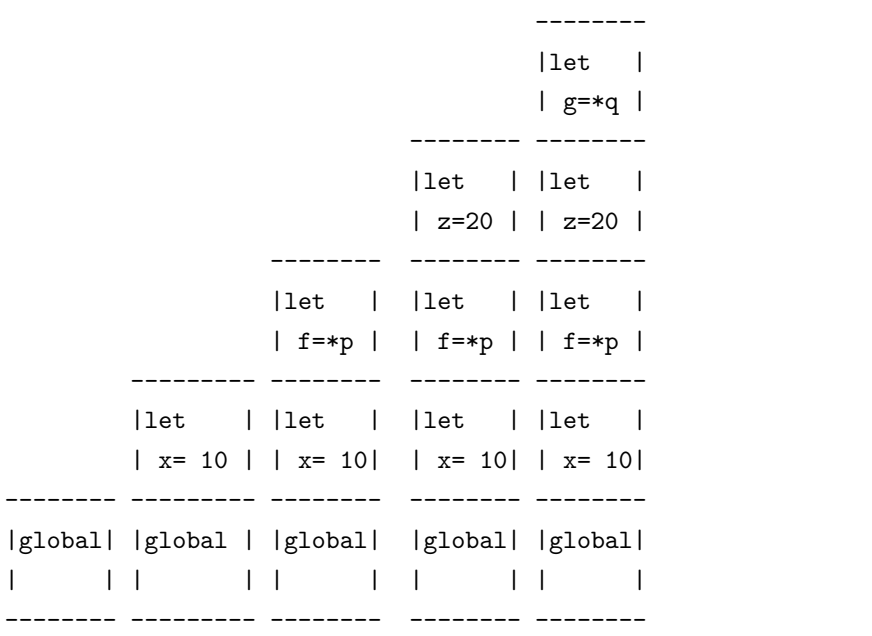
以下の OCaml プログラムを実行したときの、スタックの様子を図示しなさい。

```
(let x = 10 in
 let f = fun y -> x + y in
 let z = 20 in
 let g = fun y -> (f x) + (f z) + y in
   g
```

)
30

前半のかっこ内の式が返すのは g という関数 (のクロージャ) であり、それを 30 に適用していることに注意せよ。
　これも書くのが大変だが、以下のようなぐあいになる。(適宜省略している。)

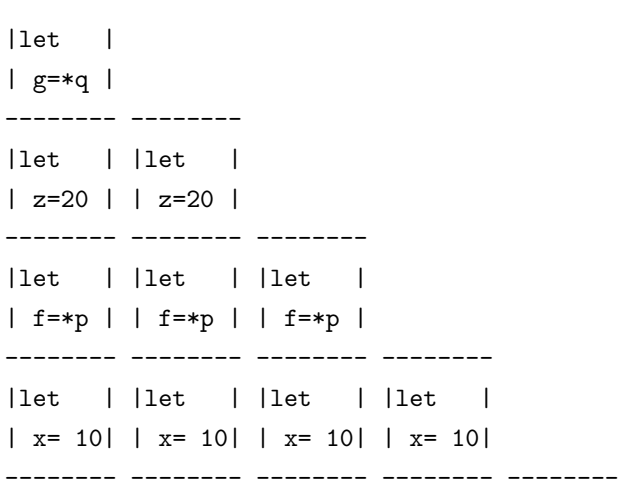
stack:



heap:
p->(fun y->x+y, x=10)
q->(fun y->...,x=10,z=20)

さらに続いて、以下のようになる。

stack:



```
|global| |global| |global| |global| |global|
|      | |      | |      | |      | |      |
-------- -------- -------- -------- --------


heap:
p->(fun y->x+y, x=10)
q->(fun y->...,x=10,f=*p,z=20)
```

そして、ポイントは、スタック自体は global スタックフレームだけになってしまった後でも、ヒープ中には、p や q に対応する関数クロージャがのこっている、ということである。そこで、最後の関数適用 (実質的に関数 g を 30 に適用するところ) で、q の先にある関数クロージャである、`q->(fun y->(f x)+(f z)+y,x=10,f=*p,z=20)` を使うことができる。つまり、上記の絵の続きは以下のようになる。

```
stack:

                  --------
                  |*p    |
                  | x=10 |
          -------- --------
          |*q    | |*q    |
          | y=30 | | y=30 |
-------- -------- --------
|global| |global| |global|
|      | |      | |      | 以下省略
-------- -------- --------
heap:
p->(fun y->x+y, x=10)
q->(fun y->...,x=10,f=*p,z=20)
```

ここで、*q というスタックフレームは、ヒープの q の先にある関数クロージャの呼びだしのつもりである。つまり、そこでは `fun y -> (f x)+...` という関数が呼びだされたことになる。そのあと、その中で `(f x)` という関数呼びだしがあるが、この f は *p をさしている、ということなので、スタックの中の p のところの関数クロージャ `(fun y -> x + y, x=10)` が呼びだされる。このスタックフレームを上記の絵では、*p と書いている。

以下同様にして、計算がすすみ、最終的に 80 が計算結果となる。

Short Quiz 2-2 (3 限に出題; 発展課題). 以下の Scheme プログラムは、関数 g を呼びだすごとに異なる値を返す。

```
(let ((x 0))
  (define (g n)  ;; 引数 n は無意味
    (set! x (+ x 1))
    x
  )
 ...
 (g 0) ;; 1 を返す。(本当は print すべき)
 ...
 (g 0) ;; 2 を返す。(本当は print すべき)
 ...
)
```

関数 g を呼ぶ以外に、このプログラムにおける x の値を読みだしたり、変更したりすることは可能か。また、上記のプログラムがどのような意義をもつか、考察せよ。

解答例上記の変数 x は、最初の let ブロックに局所的な変数であるので、外から直接アクセスすることはできない。(C 言語であれば、x へのポインタを取得するという方法もあるかもしれないが、Scheme 等の言語ではそういうことは原則としてできない。) よって、上記の関数 g を呼びだす以外に、x の値にふれることはできない。これは、オブジェクト指向言語において、クラスの中で局所的に宣言された変数 (Java では「フィールド変数」とよぶ) と極めて似た仕組みであり、Scheme でオブジェクト指向言語をシミュレートするのに使える。

Short Quiz 2-3 (3 限に出題; 解答した人はメイルで TA あてに提出) 関数クロージャに格納する「環境」は、スタックでなくヒープに置く必要がある。一方、今までの方式 (関数呼び出しに対応したスタックフレームを、スタックに積むことよって実装する) では、環境に相当するデータ (局所変数 x の値など) は、すべてスタックに積まれていると考え、ヒープに積んだのは、構造を持つデータだけであった。

この差異をどうやって吸収すればよいだろうか？

解答例変数への代入がないプログラム言語、(あるいは、代入可能な変数が明示的に区別されている OCaml 言語) などでは、「スタックに格納された環境」を「ヒープ (その中の関数クロージャを格納したいとおもっていた場所)」へコピーすればよい。こうすると、1 つの環境がコピーされて 2 か所に存在することになってしまうが、変数の値を書き換えない言語であれば、2 か所に同じ変数の定義があっても、さしつかえない。

一方、Scheme のように、関数クロージャもあるし、変数への値の代入がいくらでもできる (代入可能な変数と、そうでない変数との区別がない) 場合は大変である。この場合のやりかた自体はいろいろ考えられる。本当に効率のよい Scheme 言語の処理方式が何であるかについては、私ではなく、専門書、あるいは、専門家にあたってほしい。(たとえば、Scheme の教科書の著書としても名高い、もと京都大学の湯淺先生の処理系 TUT Scheme がある。また、米国で活躍する日本人プログラマである Shiro Kawai さんによる Gauche という名前のすばらしい Scheme 処理系がある。)