プログラム言語論

亀山幸義

CS 専攻

筑波大学 情報科学類 講義

ここまで

この授業:

- 自分を知る事は他人を知る事，他人を知る事は自分を知ること．
- 外国旅行「日本の良いところはどこですか？」，就職面接「あなたの強みは何ですか？」
- 他のプログラム言語を知る事は，自分の得意なプログラム言語を知ること．

C 言語の特徴は？

- 手続き型言語
- ...

他と比較したときの C 言語の特徴は？

- 関数の扱い
- メモリ管理・ヒープ
- 制御構造・型システム
- まずは、他の言語を知ろう!

関数型言語

- ラムダ計算 (＝「関数」概念を追究した体系) に基づくプログラム言語たちのこと
- 例. Scheme, Lisp (Common Lisp etc.), ML (SML, OCaml), Haskell
- 関数型言語の機能 (の一部，有力なもの) は Ruby 他の言語が受け継ぐ
- 例. 関数クロージャ, Java の Generics

ラムダ計算 (λ-calculus)

- 関数の入力と出力を明記する記法
- 「$f(x) = x^2 + 5x$ となる関数 f」 を,「$\lambda x.\ x^2 + 5x$」と表す．(無名関数，匿名関数)
- 「上記の f に引数として 10 を与えた結果 (値)」を「$f\ 10$」あるいは「$(\lambda x.\ x^2 + 5x)\ 10$」と書く．
- つまり，$f\ 10 = (\lambda x.\ x^2 + 5x)\ 10 = (10^2 + 5)10$ が成立．
- 高階関数 (higher-order function): 関数を引数としてもらったり，返す値にしたりする (高いレベルの) 関数，(数学では「汎関数」と言うこともある．)
- この授業では，ラムダ計算の本格的な説明はしない．(⇒ 2 学期の「計算モデル論」)

関数型言語

C 言語などの手続き型 (あるいは命令型) 言語と比較したときの関数型言語の特徴:

- ラムダ計算に基づく．つまり,「関数」概念に基づく．
- 単一代入が基本． 参照透明性 (referential transparency)
- 意味論が明快・簡潔で検証しやすい
- 簡単な割に実は強力; 高階関数，データ型
- 得意な分野: 種々のアルゴリズムの記述，プログラム言語処理系，記号処理システム (不定長データの複雑な処理)
- 不得意な分野: 固定長データの数値計算，高性能計算

関数型言語

- Lisp: 古くからある関数型言語，人工知能システムや数式処理システムなどの記述言語．
- Scheme: Lisp の意味論を洗練したもの．
- ML (Meta-Language): 関数型言語の一族の名前，SML, OCaml などがある．最も成功した関数型言語．
- ほかには，ErLang (企業が実際に利用), Haskell (研究者が作った言語)，F#(MicroSoft の ML-like な言語) など．

miniML は OCaml のサブセットとして設計．Scheme や Haskell などとは構文は異なるが，それらのサブセットと思うことも可能．

miniML の構文と意味

資料を参照のこと．

miniML のプログラム例

```
(let x = 1 in x + x) + 5
let f = (fun x -> x + 1) in f (f 3)
(fun f -> f (f 3)) (fun x -> x + 1)
(fun f -> fun x -> f (f x)) (fun y -> y + 1) 3
let x=5 in let f=(fun y->x+y) in let x=10 in (f 20)
```

来週の演習より前にこの言語を使いたいときは、coins マシンで ocaml とやって OCaml を起動してください。print と show 以外そのまま OCaml で動くはずです。(MiniML 処理系は、去年バージョンのままなので、来週の演習用ではありません。)

miniML の特徴

- 「式」しかない (「文」がない).
- ブロック構造, 静的束縛、値呼び計算。
- 単一代入: 変数への値の代入 (assignment) がない.
- 高階関数: 関数もデータ (値) となる。
- データ構造として対 (ペア) がある。
- (miniML のみ) 1 引数関数しかない。
- (miniML のみ) 型を書かない。

プログラム例

対 (ペア) を作る操作を多数回繰返すプログラム:

```
let limit=10000000 in
  let rec f x =
     if x =limit then "ok"
     else let _ = (x,x+1) in f (x+1)
  in f 0
```

ただし _ というのは、「無名の変数」のこと (後で使わない変数)。
これだけ繰返しても、エラーは起きず、計算が正常に終了する。

- ペア $(x, x+1)$ のデータは、スタック上に取られるのではない。
- ペア $(x, x+1)$ のデータを格納するためのメモリ領域は、このペアが不要になったら自動的に回収され、他のペアのために再利用される。

ヒープ

ヒープ (heap): プログラム実行時に、データを格納しているメモリ領域で、スタック以外の領域。

- 構造を持つデータを格納する目的で使用される。
- そのデータを生成した関数が終了しても、そのデータが使われる可能性がある。
- ヒープに格納されるデータの例
  - ペア
  - 文字列
  - リスト
  - 関数 (正確には関数クロージャ)
  - オブジェクト指向言語でのオブジェクト
  - その他、固定長のメモリ (通常は、32bit や 64bit) にはいりきらないデータ
- 式 `let _ = (x,x+1) in ...` の計算
  - ペア $(x, x+1)$ はヒープに置かれる。
  - 局所変数 y の値は、(ヒープ領域上の) そのペアへのポインタであり、それ自身はスタックに置かれる。

ヒープ

- C 言語ではプログラマがメモリ管理を行う。malloc(), free()
- OCaml などの言語では、処理系が自動的にやってくれる。
- 論理式、数式、プログラム、XML データなど、構造のあるデータ (特に、可変長のデータ) を扱う際には、上記の 2 機能があるプログラム言語を使うことは、ほぼ必須。
- ヒープは、後で回収することも考えると、単なる「配列」状ではなく、「リンクをもつリスト」状の構造を持つ。(長く使い続けていると、ヒープ全体が「使用領域」と「未使用領域」による「まだら模様」になっていく。)

データ構造

- MiniML 言語の「対 (ペア)」データ型を使えば、様々なデータ構造を表現できる。C 言語で同様なことをするためには、構造体 (struct 型) を使えば良い。
  - MiniML 言語では、一度作ったペア (のためのメモリ) は、どうなるだろうか？ ペアを大量に作り続ける関数を作成して呼びだして、実験せよ。
  - MiniML 言語処理系において「ペアなどのデータを覚えておくためのメモリ領域」は、スタック上に取られるかどうか考えなさい。

プログラム実行時のメモリ (再掲)

- Register
- Program Counter
- Code
- Environment Pointer (スタックを指す)
- Data:
  - Stack (スタック)
  - Heap (ヒープ)

ヒープ

- ペアを作る時 (式 $(x, x+1)$ の評価)
  - 処理系は、ヒープの中のあいている場所を 1 つ確保し、このペアを格納する。その場所へのポインタを返す。
  - C 言語の malloc 関数は不要 (処理系が自動的に行う)。
- ペアを使う時: ポインタをたどる。
- ペアを使わなくなった時
  - どこからも参照されないペアを格納する場所は、いつか、回収される。
  - C 言語の free 関数は不要 (処理系が自動的に行なう)。
  - 不要になった瞬間に回収するか、ヒープが不足したときに一斉に回収するか、という選択肢がある。

ごみ集め (Garbage Collection)

- ヒープ領域において、使われなくなったデータに対応する領域を回収する (空き領域に連結する) 操作。
- 通常は、ヒープが足りなくなったときに、一斉に回収する。(その瞬間に処理系の動作が停止したように見える。cf. 昔は、Lisp 言語の大事な応用は、ロボットの制御だった。)
- 処理速度が大事; 良いごみ集めアルゴリズム採用することが必須。
- 古くから研究されているが、現在でも更なる改善のための研究がなされている。(高速性とともに、「絶対に間違いがあってはいけない」という意味で、正当性も非常に大事。)

まとめ

- 関数型言語の導入
- プログラム言語における種々の概念は、関数型言語に基づいて説明できることが多い.
- ヒープ、(クロージャ、制御構造、型システムは後で)

Quiz

- ヒープの役割は何だろう？
- プログラムを書いたときにはどれだけのメモリを使うかわからず、実行時に量が決まることがある。そのような場合、C 言語では malloc(), free() 関数を使うが、しばしば「malloc したのに、それを free し忘れる」ということがある。これを避けるにはどうしたらよいだろうか？

Quiz 1 の答

- スタックは、「ブロック構造」に対応した局所的なデータを格納するので、そのようなブロック構造に対応しないデータは、全てヒープに置く必要がある。
- ブロックの実行が終了した後に「生き残る」データすべてをヒープに置く。
- たとえば、「対」などのデータ構造を (関数呼び出しの中で) 作ったら、それを生成した関数が終わっても、生き残らなければならない。(関数の返り値に含まれる場合などがあるため。)
- 「局所的な変数の値が、対などのデータ」の場合は、「スタック (上の局所変数) から、ヒープ (上の対をあらわすセル) へのポインタ」として実装される。

Quiz 1 の答に関連して

C 言語でこんなことを書いてはいけない。(これでも動くことが多いが、仕様上は、正しく動作するとは限らない。)

```
int *foo (int x) {
  int z;
  z = x +10;
  return &z;
}

int main () {
  printf ("%d\n", *foo(5));
}
```

変数 z に対応するメモリ領域はスタック上に取られる。関数 foo の実行が終了するとその領域は、消滅する。
関数 foo が終わっても使いたいデータは、(関数の返り値にするか) ヒープ上に置く必要がある。