プログラム言語論-第 10 週-

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義, 2009 年 6 月 22 日

1 はじめに

2 制御構造

3 授業のまとめ

前回の Short Quiz(2 限)

同じようなプログラムを再利用するためには，エディタで、プログラム断片をコピーして使えばいい？

答 (の 1 つ)．必ずしもそうではない．1 つのプログラム断片を (たとえば)3 回コピーして改変して使う場合，「プログラムを書く」手間は確かに 1 回プラスアルファ程度で済むが，コンパイルする時間，コンパイル後のコード量などは 3 倍になる．さらに重大な問題は，このプログラムの維持 (ディバッグ，改良，再利用など) にかかる手間が 3 倍になることである (コピー前のプログラムにバグがあったとき，そのバグは 3 箇所にコピーされる．) もし，3 つの断片を何らかの方法で抽象化して 1 つにまとめて管理することができれば，プログラムの維持にかかる手間が減ると予想される．

前回の Short Quiz (3 限)

「事実やルール (推論規則) を記述するだけで，プログラムとして動く (一般に,「仕様」を書くだけでそれを満たすプログラムができる, executable specification) という言語が，現実に有用な場面は何か？

答えの例:

- Prototyping (プロトタイピング，プロトタイプ作り)：
  いきなり本格的な「もの」を作るのではなく，その前に，目的物を少し簡単化したもの (プロトタイプ) を作ることが多い．早い段階でのテスト (そもそも仕様や設計に矛盾はないか等) に用いられる．
  “Proof of Concept” (概念の正しさの証明)
  ソフトウェアのプロトタイピングでは，素早く作ることが大事であり，プロトタイプの実行速度等はあまり気にしない．
- その他： 仕様が頻繁に変更になるシステムの設計など．

概要

今回：

- プログラムの制御構造
- 授業のまとめ

プログラムの制御構造

制御構造 Control Structure

- プログラム中のどの部分を実行するか (プログラムの制御がどこにあるか) を決める構造
- 逐次的な手続き型 (命令型) 言語の場合、「上から下へ」という制御の流れを変更する命令として、go to (jump), return, 例外などがある。
- 関数型言語にも、制御構造がある。

GO TO-1

FORTRAN 言語

```
   V = 1
   I = 1
10 IF (I.GT.10) GO TO 20
   V = V * I
   I = I + 1
   GO TO 10
20 ...
```

C 言語のコード:

```
   v = 1;
   for (i=1; i <= 10; i++)
      v = v * i;
```


GO TO-2

E. W. Dijkstra, “GO TO Statement Considered Harmful.”, 1969.
GO TO 文 (プログラム中の任意の場所へのジャンプ命令) を乱用すると、プログラムを理解することが困難になる。
Dijkstra の主張は広く受け入れられ、その後に設計された多くのプログラム言語では、無制限の GO TO 文は使われない方向になった。

より良い制御構造

- ブロックなど、プログラムの一定のまとまりの外から中に飛び込む制御構造は「悪い」。
  - 「まとまり」ごとにプログラムの意味を考えることができない。
- ブロックの中から抜けるための制御構造は「良い」。
  - C 言語の continue や break, また関数における return

強力な制御構造: 例外-1

「例外機構」＝大域脱出

- 現代の多くのプログラム言語が持つ。
  - Java 言語の try-catch-finally (と throw)
  - ML 言語の try(handle)-raise
  - Lisp 言語の catch-throw
  - C 言語の setjmp-longjmp (ライブラリをリンクする必要あり)
- 入れ子 (ネスト) になったループや関数呼出しの中から一気に外へ飛び出す。

強力な制御構造: 例外-2

```
exception Zero of int
let mult lst =
  let rec visit lst =
    match lst with
      | [] -> 1
      | 0 :: lst2 -> raise (Zero 0)
      | x :: lst2 -> x * (visit lst2)
  in
    try visit lst with Zero x -> 0
```

mult [1;2;0;3;4] を計算すると、
try 1*(2*(visit (raise (Zero 0)))) with Zero x->0 となり、ただちに 0 を返す。

強力な制御構造: 例外-3

例外機構=大域脱出
ジャンプする際にスタックを巻きもどす操作が必要。

```
                          -----------
                          lst=[0]
              -----------  -----------
              lst=[2;0]    lst=[2;0]
-----------   -----------  -----------
lst=[1;2;0]   lst=[1;2;0]  lst=[1;2;0]
-----------   -----------  -----------  -----------
```


強力な制御構造: 例外-4

- 例外機構は、プログラムを書くのに必要不可欠か？NO. 表現力の点では、例外機構がなくても同等のことは書ける。
- 例外機構は、何のためにあるのか？プログラムをわかりやすく、簡潔に書くため。また、場合によっては、実行効率をあげるため。

強力な制御構造: 例外-5

リストの要素の平方根の和を求める。

```
let sqrtsum lst =
  let rec visit lst =
    match lst with
      | [] -> 0.0
      | x :: lst2 ->
         if (x<0.0) then -1.0
         else
           let res = (visit lst2) in
   if res < 0.0 then -1.0
           else (sqrt x) +. res
  in visit lst
```


強力な制御構造: 例外-6

リストの要素の平方根の和を求める。

```
exception Negative of float
let sqrtsum lst =
  let rec visit lst =
    match lst with
      | [] -> 0.0
      | x :: lst2 ->
         if (x<0.0) then raise (Negative x)
         else (sqrt x) +. (visit lst2)
  in
    try visit lst with Negative x -> -1.0
```


さらに強力な制御構造 (参考)

「継続」 (Continuation) ＝今から先の残りの計算すべて

- Scheme 言語の call/cc, SML/NJ の callcc

「残りの計算」を保存しておいて、後で使うことにより、「現在の地点に戻ってくる」効果がある。
応用: バックトラック、ループ、コルーティン (coroutine) 等の制御、ディバッグ、スナップショットなど。
※ Ruby や JavaScript の yield は、継続を扱う制御構造とも考えられる。

データ構造と制御構造

データ構造と制御構造は、プログラム言語の 2 大構成要素。

- これらは、プログラムを構造化することにより、
- プログラムを書きやすく、理解しやすく、維持しやすくする。

Short Quiz (2 限)

なぜ、GO TO 文 を乱用するとプログラムが読みにくくなるのだろうか？

この授業

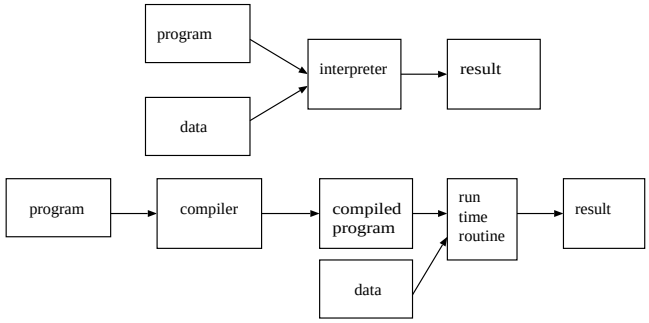
これまでの話題

- インタープリタとコンパイラ
- プログラム言語の構文
- プログラム言語の意味論: ブロック構造、変数のスコープと束縛、評価順序、メモリ管理
- 手続き型言語と関数型言語
- データ構造: ヒープ、ゴミ集め
- 型システム、抽象化データ型、モジュール
- オブジェクト指向言語
- 論理型言語
- 制御構造

インタープリタとコンパイラ

1 つの見方:

- インタープリタ (interpreter) は解釈系 (実行系)
- コンパイラ (compiler) は翻訳系 (変換系)



コンパイル時と実行時

- Compile Time (コンパイル時) vs Run Time (実行時)
- Compile Time の情報
  - プログラムを実行する前にわかる事，静的な情報
- Run Time
  - プログラムを実行している時にわかる事，動的な情報

静的束縛 vs 動的束縛、静的な型システム (型検査、型推論) vs 動的な型システム (型検査)、静的解析 vs 実行時解析 etc.

John Mitchell 先生の (過去の) 期末試験から

以下の URL から “Handouts” → “Sample Final Exam” を参照:
`http://theory.stanford.edu/ jcm/books/cpl-teaching.html`

プログラム言語はなぜたくさんあるか？

将来的に、表現力が非常に高く、高速実行可能なプログラム言語がたった 1 つあれば、良いか？

- 言語の表現力: A 言語が B 言語より表現力が高いとは、B のプログラムと同等なプログラム全てを A 言語で書けるとき。

答: おそらく NO。

- (答その 1) 巨大すぎ、複雑すぎるプログラム言語は使えない。
  - 言語の処理系を書く人やプログラムを保守 (検証、再利用 etc.) する人にとっては、言語が大き過ぎると大変。
  - Ada 言語の失敗。
- (答その 2) 解くべき問題に応じた、適切な抽象度の言語を使うべき。
  - 数式処理のアルゴリズムを書く人は、ゴミ集めアルゴリズムの詳細は知らなくてよい。
  - 非常に高速のネットワーク・スイッチの内部コードを書く人は、(どう効率的に実装されるかわからない) オブジェクトを使っていられない。

プログラム言語: 汎用 vs 特化

- この授業で取り上げてきた言語はすべて汎用 (general purpose) プログラム言語
- 領域特化言語 (Domain Specific Language) :
  特定の領域 (数値計算、数式処理、推論、グラフィクス etc.) に特化した言語。

汎用言語の役割と DSL の役割は相補的 (どちらか一方だけでは、すべてのニーズに対応できない)。
お勧めの読みもの: “Purpose-Built Languages”, ACM Queue, Mike Shapiro (インターネット上で無料閲覧可能)。
`http://queue.acm.org/detail.cfm?id=1508217`
大事なこと: 「原典にあたれ」二次的著作物があふれている世の中：誰が何を言ったかを正確に (自力で) 見極められることがとても大事。

Short Quiz (3限)

自分が最も好きなプログラム言語 (1-3 個) を、好きな理由とともにあげなさい。

期末試験

- 6 月 29 日 (月) 3 限 (12:15-13:30)
- 3 限開始後すぐに用紙を配布して開始するので、開始時刻に間にあうようにしてください。
- 資料等の持ち込みは不可です。ただし、留学生が辞書を持ち込む場合は、監督員に申し出てください。(電子辞書の場合は、通信機能がないものにしてください。) また、日本語、英語のいずれで解答しても構いません。
- 問題は、技術的な (プログラムのコードを見て解答する) 形式のもの (少し) と、文章で質問して解答する形式のもの (大部分) とから構成されます。
- 知識ではなく、理解度を問います。たとえば、C 言語や ML 言語の詳細な構文を覚えていないと解けない、というものは出しません。
- これまでの出席については、2 週以内の欠席はノーカウントで、それを越える場合は、5 点ずつ減点です。
- 答案返却期間を、期末試験終了後 1-2 週間に設定します。