プログラム言語論

亀山幸義

筑波大学 情報科学類

授業のまとめ

Java と Ruby

Java

- C/C++言語に似た構文
- ブロック構造言語; 変数は静的束縛
- 静的な型システム、型安全性
- 信頼性、安全性を重視

Ruby

- スクリプト言語
- 動的言語 (静的型付けをしない)
- クラス定義は宣言ではなく実行文
- 開発効率を重視

共通すること: オブジェクト指向であること (動的ルックアップ、情報隠蔽、継承)

静的 vs 動的

1. 宣言された変数はすべて，式の中にあらわれるかどうか.
2. 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.
3. 実行中に使用されるスタックの量.
4. 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.
5. 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.

C, ML では、1,5 は静的に決められる。2,3,4 は決められない (実行時の情報)。

静的 vs 動的

コンパイラとインタープリタの差 (再訪問)

- 多くのコンパイラは、プログラム全体 (あるいはプログラムの 1 つのモジュール全体) を読みこみ、可能な限り多くの静的情報を取得して、それを生かした効率良いコードを生成する。
- 多くのインタープリタは、そのようなことをしない。

静的情報を使って効率がよくなる例:

- 局所変数が、スタックフレーム中のどこにあるかを解析すれば、実行時には、「局所変数がどこにあるかを探す」必要がなくなる。
- 「a+b」という式で、a,b が整数型とわかっていれば、整数型かどうかのチェックはいらず、いきなり機械語の「加算」命令を使える。

現在の Optimizing Compiler は、上記以外にも様々な効率化をはかっている。

静的型付け vs 動的型付け

Java, ML, C は静的型付け: コンパイル時にプログラムの型の整合性を検査 (あるいは推論) する。実行時には (ほとんど) 型の整合性を検査しないですむ。

Ruby, Perl, Python, Lisp, Scheme などは、動的型付け: コンパイル時には型の整合性はチェックしない。実行時に型が整合しない演算を行えば、エラーを発生させる。

これらの帰結:

前者は、より高い信頼性を得やすい。

後者は、より柔軟なプログラミングが可能。

この授業

これまでの話題:

抽象構文と具体構文、意味論、インタープリタとコンパイラ、抽象機械

式の評価、ブロック構造、変数のスコープと束縛

評価順序と制御構造; 値呼び、名前呼び、必要呼び、再帰

スタック機械と PostScript

1 階言語の意味論、関数クロージャ

高階関数、制御構造、副作用、末尾再帰

データ構造と型システム、型検査と型推論

命令型言語、ストア

抽象データ型、モジュール

オブジェクト指向: 4 つの基本概念

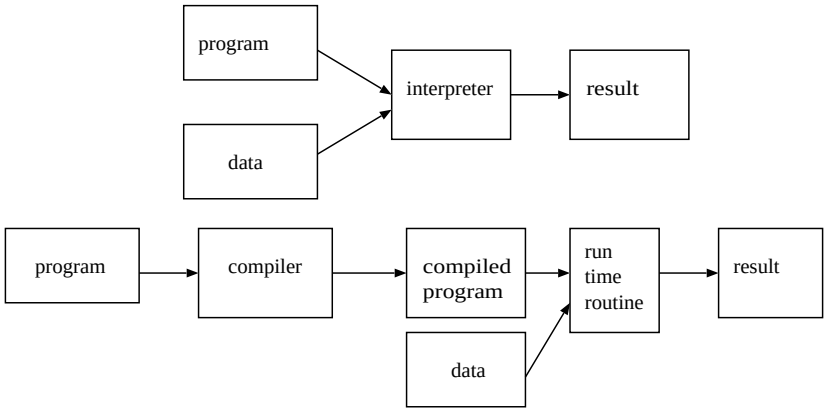
静的言語と動的言語、ドメイン特化言語

インタープリタとコンパイラの違い (1)

1 つの見方: 入出力が違う

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)



インタープリタとコンパイラの違い (2)

通常の見方: 実行性能 (速度) が違う

インタープリタ (interpreter) は低速

コンパイラ (compiler) で生成されたコードは高速

コンパイラで生成されたコードがインタープリタより必ず高速ということはないが、多くの場合そうである。なぜか？

コンパイル時と実行時

Compile Time (コンパイル時) vs Run Time (実行時)

- コンパイル時: プログラムを実行する前，静的な情報
- 実行時: プログラムを実行する時，動的な情報
- コンパイラは、コンパイル時の情報を活用して、質の良いコードを生成。
- インタープリタは、(通常は) そのようなことはしない。

静的 vs 動的

- 静的束縛 vs 動的束縛 (および、静的ルックアップ vs 動的ルックアップ)
- 静的な型システム vs 動的な型システム
- 静的解析 vs 実行時解析

コンパイラの最適化 (optimization) の例

変数 x は、環境 (変数の値の列) の 3 番目の変数であることを知って、「変数 x の値を探して取り出す」という操作を「環境の 3 番目の値を取り出す」命令に置きかえる。

$(a + b)$ という式は、a,b ともに整数のとき、それらの加算で置き換え、それらが文字列のとき、文字列の連接に置き換える。

$(a + a)$ という式は、a が副作用を含まないとき、$(a * 2)$ に置き換える。

John Mitchell 先生の (過去の) 期末試験から引用

`http://theory.stanford.edu/~jcm/books/cpl-teaching.html`
以下の性質は，コンパイル時に決定できる情報 (C)，実行時にならないと決定できない情報 (R)，どちらでも決定できない情報のどれか (N)？

- プログラム中の全ての変数が，宣言された時に初期値を与えられているか？ C.
- プログラムの実行が終了するか？ N.
- (C 言語) 配列の要素への参照は，宣言された配列の範囲 (下限以上，上限未満) におさまっているか？ R.
- (C++言語) プログラムは型が整合しているか？ C.
- すべての宣言された変数は，式の中に現れるか？ C. なお,「実際に使われるか？」という質問なら，答は「R」
- システムコールの返り値は，そのシステムコールの呼び出し元でチェックされているか？ C.
- 2 つの変数名がメモリ上の同じ番地を指しているか？ R.

いろいろなプログラム言語

John Mitchell, Concepts in Programming Languages, 2003 から抜粋。

言語	式	関数	ヒープ	例外機構	module	object	thread
Lisp	x	x	x				
C	x	x	x				
Algol60	x	x					
Modula-3	x	x	x	x	x	x	
ML	x	x	x	x	x		
Simula	x	x	x			x	x
Smalltalk	x	x	x	x		x	x
C++	x	x	x	x	x	x	
Java	x	x	x	x	x	x	x

いろいろなプログラム言語

言語	変数束縛	呼び出し	closure	object	型付け	副作用
Lisp	静的	CBV	有り		動的	有り
Scheme	静的	CBV	有り		動的	有り
C	静的	CBV			静的	有り
C++	静的	CBV		有り	静的	有り
Java	静的	CBV		有り	静的	有り
OCaml	静的	CBV	有り	有り	静的	有り
Haskell	静的	必要呼び	有り		静的	無し
Ruby	静的	CBV	有り	有り	動的	有り
Prolog	-	-			動的	有り

プログラム言語はなぜたくさんあるか？

表現力が非常に高く、高速処理が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) 巨大過ぎたり、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 (right level of abstraction) 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。

プログラム言語の理解

プログラム言語を理解する 3 つの方法:

> その言語の「概念」を説明している文献を読む。
> その言語で書かれた、質の良いコードを理解しようとする。
> その言語のインタープリタを、その言語自身で書いてみる。

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ほとんどすべて汎用 (general purpose) プログラム言語
>
> ドメイン特化言語 (Domain Specific Language, DSL)
> 特定の問題領域 (数値計算、数式処理、データベース利用，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)。

ドメイン特化言語

例

- データベースを操る: SQL
- 数式を処理する: Mathematica
- パーサ (構文解析器) を作る: yacc
- 電卓: 整数、+, −, ···
- 亀の子グラフィクス
- ロボットの関節を動かす

お勧め: "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`

ドメイン特化言語の実現

2 段階の実現方法:

- 1 つ 1 つの DSL に対して、素晴しいコンパイラを作るわけには (なかなか) 行かない。
- 汎用言語で、DSL 処理系 (インタープリタ) を記述する。
- ユーザの書いた (DSL 語による) プログラムは、その処理系が実行する。

DSL の 2 つのタイプ:

- 外部 DSL (external DSL): ホスト言語 (汎用言語) と別の構文
- 内部 DSL (internal, embedded DSL): ホスト言語の中に埋めこまれる

ドメイン特化言語の実現

大きな問題:

- DSL 処理系をどうやって (手軽に) 書くか。
- DSL 処理系の実行性能をどうやって (手軽に) あげるか。

DSL 処理系の実行性能 (Abstraction Overhead の削減)

- メタプログラミング・アプローチ

プログラムを作るプログラム

MetaOCaml プログラミング:

```
let rec gen_power n x =
  if n=0 then .<1>.
  else   .< .~x * .~(gen_power (n-1) x)>.
let power n = .<fun x -> .~(gen_power n .<x>.)>.

power 5
=> .<fun x -> x * (x * (x * (x * (x * 1))))>.
```

引数 n に特化した効率良いプログラムが作られる
⇒ DSL 処理系を、あるプログラムについて特化すると、そのプログラム専用の高速の処理系になる。

プログラム生成とメタプログラミング言語

Program Generation (Code Genreation): 特定の入力パラメータ (あるいは、特定のハードウエア環境) に特化した 高速なプログラムを作ってから、それを実行する。

Program Generation をサポートする機能を持つ言語

- Lisp/Scheme マクロ (擬似引用と eval), Ruby
- (C++ template)
- Template Haskell
- MetaOCaml
- Scala LMS(軽量モジュラーステージング)

期末試験

6 月 25 日 (木) 3 限プラスアルファ(12:15 から 90 分程度)

遅刻厳禁! (12：45 を越えると入室を認めない)

資料等の持ち込みは不可です。(留学生が辞書を持ち込む場合は OK)

暗記物ではなく、理解度を問います。

> ML 等の、言語の表層構文を覚えていないと解けない、というものは基本的に出しません。
> いろいろなプログラム言語の個別の知識ではなく、プログラム言語によらない基本概念として説明したものを、その意味内容とともに説明できるように理解してください。