ゼミ: Design Concepts in Programming Languages (Kickoff)

久野　靖*

2012.9.6

□ Franklyn Turbak, David Gifford, Mark A. Sheldon, Design Concepts in Programming Languages, MIT Press, 2008.

0 Preface

□ 6.821 Programming Languages (MIT の院における*入門レベル*の科目 (1 セメスター)

- プログラミングができ、数学ができることは前提
- 演習問題は実際に授業/試験で使用しているものが大半

□ 目標:　簡潔かつ厳密なフレームワークを学びそれに基づいてプログラミング言語の設計と実装を理解

- 実際の言語をあれこれ見るツーアはやらない
- 代わりに構文的には類似した簡潔な教育用言語を題材にする
- 今日は safety/security 重視 → それに関連する概念を多く取り上げる
- それらの概念の現実のプログラミング言語への適用についても機会があれば取り上げる

□ 構文は S 式を用いる ← 解析が容易で操作しやすい

- すべてを明示的にという我々の方針に合致

□ 「プログラミング言語は柔軟で表現力の高いメディアである。コンピュータ科学の土台となるこのコンピューテーションのキャンバスについて情熱を持って語りたい」

1 Introduction

□ 「整理と簡単化は対象理解の第 1 歩 --- 本当の的は未知。」(トーマス・マン、「魔の山」)

1.1 Programming Languages

□ プログラミングは楽しい --- が、それは簡潔で明快なとき。ぐちゃぐちゃなコードで読みづらいと確信が持てず、楽しくない

□ プログラミング言語の設計はメタなプログラミング活動であり、普通のプログラムと同じくらい楽しい

- 言語の設計においては簡潔さと明快さが一層重要
- 今日では何百もの言語が使われている
- よいアプリケーション設計は新しい言語や既存言語の拡張とコミだったりする ← 柔軟かつ拡張可能なアプリケーションは何らかのプログラミング機能を提供する必要

□ 言語設計の要素は、通常のプログラミングにも存在

- 例:　コレクションデータ構造のインタフェース設計
- コレクションに対するイテレーションはどのようにカプセル化する?
- これはプログラミング言語にどういうループ構造を入れるかということに類似した問題

□ 本書の目標:　プログラミング言語における偉大なアイデア群を、簡潔なフレームワークの元で (複雑さをそぎ落として) 学ぶこと

- プログラミング言語の意味を規定する手法を複数学ぶ
- 規定が少し変わっただけでプログラムのふるまいが大きく変わることもある
- プログラミング言語設計空間の多くの次元について見て行く
- 各次元にどのような選択肢があるかについても
- それらの次元の選択肢どうしがどのように干渉し合うかについても
- プログラミング言語に関するうまいやり方/拡張方法 ←とり消し可能な状態遷移、ループ脱出、パターンマッチなど
- 我々のアプローチ:　言語のふるまいがより複雑になっている今日、そのふるまいの記述方法はより簡潔であるべきという信念

*筑波大学ビジネスサイエンス系

1.2 Syntax, Semantics, and Pragmatics

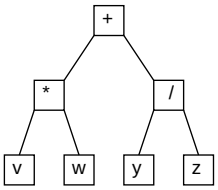
□ 伝統的にプログラミング言語は3つの側面から考えられて来た

- Syntax（構文）--- 言語の形式
- Semantics（意味）--- 言語の意味
- Pragmatics（実践）--- 言語の実装

1.2.1 Syntax

□ 構文は言語のフレーズの具体的な記法を定める。

- 例: v と w の積と、y を z で割ったものとの、和が欲しいとする。
- 伝統的な数式: vw + y/z
- Lisp のかっこつき前置記法: (+ (* v w) (/ y z))
- 逆ポーランド電卓: [v][↑][w][↑][×][y][↑][z][↑][÷][+]
- スプレッドシートのフォーム
- ツリーの図



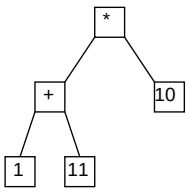
- これらの記法は構文は違うが同じ句構造を表現

□ プログラミング言語の構文: 「どのような文字の並びが正しく、正しい文字の並びがどのような句構造を表現しているか」を定める

1.2.2 Semantics

□ 意味は句構造にその句構造がどのような意味を持つかを対応づける

- どのような句にも「本来の」意味などない
- 例:



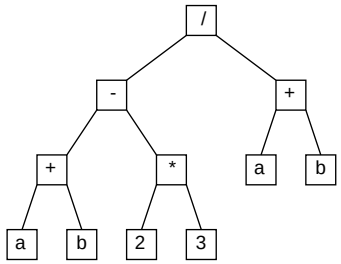
- 1、10 などが十進数値で+や*が和や積なら (1+11)・10 = 120
- *が指数なら 12 の 10 乗
- 数値が 2 進なら (1+3)・2 = 8
- 奇数が真、偶数が偽、+と*は∨と∧なら (T ∨ T) ∧ F = F
- 何の計算も表していなくて、ただそのような形だけなのかも
- 何かのものをエンコードしているのかも

□ つまり、1 つのプログラムの句が多くの意味に対応させ得る

1.2.3 Pragmatics

□ 意味は句が「何を」意味するかを定めるが、実装は「どうやって」その意味の計算をするかを定める

- とくに関心事: 資源 (時間、空間、共有デバイスなど) の有効活用
- 例: 次の式木の評価



- a や b が何らかの数値を表すとすると、(+ a b) が 2 回現れるので、素朴な方法だと同じ加算を 2 回やることに
- 別の方法として、加算は 1 回で、次に必要な時は前の値を参照
- 意味は変わらないが、資源の利用は変わっている (加算は減るが記憶場所が必要に)
- どちらがよいかは時間や領域の貴重さによる
- (* 2 3) が常に 6 になるので、何回もこれを評価するなら最初から 6 にしておくのがいいかも

1.3 Goals

□ 本書の目標はプログラミング言語デザインのイディオムの包括的な集合についてその意味を探究すること

- さらにそれらをどう組み合わせて自立したプログラミング言語とできるか
- さらに意味と実装の関連

□ 構文については標準的なコンパイラの教科書が十分カバー

- →本書では最低限の本書で使うコンベンションだけ

□ 意味についてはその記述ツールを複数紹介し、それに基づいてさまざまなプログラミング言語の機能やそのバリエーションを理解

□ 実装については高レベルの部分だけ

- いくつかの基本的な実装手法やプログラム改良手法程度まで

□ プログラミング言語の機能を議論するに当たり、ミニ言語 (群) を対象

- これらはそれぞれ既存の言語の特定側面を反映
- とても小さい言語なので実用には向かないかも
- しかし基本的な考え方は同じ。余分な複雑さがないので検討が楽
- これらのミニ言語の機能を組み合わせることで現実の言語に近くできる

□ 意味と実装の問題はプログラミング言語の特性とその上での特定のプログラムについて考えるときに重要

- さらに 2 つの基本的な実装戦略: 解釈実行と変換実行
- 解釈実行: ソース言語の表現 S を S のインタプリタが直接実行
- 変換実行: S をトランスレータでターゲット言語 T に変換し、T のインタプリタで実行
- インタプリタ、トランスレータ (コンパイラ) は実装言語で書かれている
- ソース言語、ターゲット言語には上述のようにミニ言語を使用
- 実装言語については付録 A のメタ言語で記述
- でも、実際に動く実装も作ってみることをおすすめする

□ メタプログラミング (プログラムを操作するプログラムを書くこと) こそ最も魅惑的なプログラミングである!

1.4 PostFix: A Simple Stack Language

□ 構文/意味/実装の例に PostFix という言語を使う

- 簡単なスタックベースの言語 ← PostScript, Forth
- 以下はインフォーマルな説明 (厳密な話は後で各種ツールとともに)

1.4.1 Syntax

□ PostFix の基本構文単位は「コマンド」。コマンドは以下の 3 種

- 整数。例: 17, 0, -3
- 特別なコマンドトークン。add, div, eq, exec, gt, lt, mul, nget, pop, rem, sel, sub, swap
- 実行シーケンス。かっこで囲んだコマンド列。例: (2 (5 mul) exec add)。これで 1 つのコマンド。

□ 「(postfix 整数 コマンド…)」がプログラム。

- すべてのかっこは必要であり省略しない (本書の他の言語でも)

1.4.2 Semantics

□ コマンドを左から右の順で実行

- 各コマンドはスタック (最初は入力の整数を保持) を操作
- スタック中の値は (1) 整数、(2) 実行列のいずれか
- 実行が終った時のスタックトップの値が出力値
- エラーは (1) 最終スタックが空、(2) 最終スタックトップが整数でない、(3) コマンド実行時にスタックの値が不適切、のいずれか

□ 各コマンドのふるまいは図 1.1

- $P \overset{args}{\rightarrow} v$ で P を実行した結果が v であることを表す
- $P \overset{args}{\rightarrow} error$ で P を実行した結果エラーになる〃
- コマンドのエラーはスタック不足かスタックの値が不適切な場合
- 以下ではコード中の { } はコメント

□ 引数とプログラムの受け取る値の数が一致しないのもエラー

- 数値演算は後置記法で表現することになる
- 比較演算子は 1(true)/0(false) を返す

□ スタック中の値を 2 回以上使うためには nget が利用できる

- index 値は 1 オリジンで、スタックが積まれると変わる

□ 実行列は 1 つの値としてスタックに積まれる

- 実行列は exec によって呼び出され、サブルーチンのように動作

□ sel は真偽値 (1/0) に応じて 2 つの値から片方を選ぶ

1.4.3 インフォーマルな記述の弱点

□ ここまでのような英語と実例による記述は典型的なもの

- まず構文が示され、それぞれの構成の意味が説明される
- 利点: 多くのプログラマがこの方法で現に学んでいる

□ しかしインフォーマルな記述が不適切な場面も多くある

- 例: プログラム変換→変換によって意味が変わらない保証?
- 例: 言語がもつ性質を示したい (PostFix コードは無限ループしない等)
- ↑自然言語記述では厳密さが不足しているので証明の土台にならない
- 例: 機能を拡張したい→機能どうしの関係を把握
- 例: 複数実装→それらが同じ言語を実装していることを保証したい

□ インフォーマルな記述は厳密さが欠けていて、また長々しくなりがち

- また、仕様不足 (すべてのケースをカバーし損なう) も起きやすい
- また、仕様過多 (仕様に含まれているべきでないことまで記述) も
- 例: PostFix はスタックで説明されるからといって、実装にスタックが無ければならないということはない
- ↑より抽象的な定義方法により、実装の自由度があるのがよい

1.5 Overfiew of the Book

□ Part I の残りは上記のインフォーマルな記述の問題を克服するツール

- 2 章: s-expression grammer ←文法は必要
- 3 章: 操作的意味論 (operational semantics)
- 4 章: 表示意味論 (denotational semantics)
- 5 章: 再帰的な仕様の意味を決める方法
- 使用するメタ言語 → 付録 A

□ Part II: 動的セマンティクス

- 6 章: FL (ミニ言語) → 名前 (7 章)、状態 (8 章)、制御 (9 章)、データ (10 章)、+プログラミングパラダイム

□ Part III: 静的セマンティクス

- 11 章: 型、well-typed
- 12 章: 型のより進んだ機能: サブタイプ、多相、種別
- 13 章: 型の明示的な記述を減らす (type reconstruction)
- 14 章〜15 章: 静的/動的セマンティクスの活用
- 16 章: effect system

□ Paret IV: 実装

- 17 章: コンパイラの例、18 章: GC

□ 形式的ツールを前面に出しているが、形式的ツールがすべてよいとは言わない

- インフォーマルな説明が分かりやすいことも多い
- 形式手法だと手におえなく面倒になることもある
- 形式手法の土台にある概念は現実のプログラミング言語を理解する上で貴重な道具だということは示したい