第1章 プログラミング言語論の概要

1.1 プログラミング言語の意味とは

プログラミング言語の仕様を定める時には、その言語の"構文" — つまり、どのような記号の列が正当なプログラムとして受け入れられるのか — とともに、その言語の"意味"を定める必要がある。

プログラミング言語にはさまざまな構成要素がある。例えば、ほとんどのプログラミング言語には条件分岐・繰り返しなどの制御構造（C 言語では **`if`** 〜 **`else`** 文, **`while`** 文, **`for`** 文, **`do`** 〜 **`while`** 文など）がある。 C++ や Java には、例外処理のための **`try`** 〜 **`catch`** 文もあるし、 Prolog にはユニフィケーション（単一化）・バックトラッキング（後戻り）などの仕組みがある。関数型言語 Haskell には遅延評価、オブジェクト指向言語にはクラス・インタフェースなど、やはり特有の仕組みが存在する。

これらの構成要素の"意味"については、現在のところ、日本語などの自然言語によって、「"条件式"が成り立つ間、"文"の実行を繰り返す」のように記述するのが普通である。しかし、自然言語による記述では、曖昧な点が多く、例えば次のような疑問が生じた時に厳密に議論することができない。

- コンパイラーの正当性 — コンパイラーは、本当に仕様書に定められた通りの動作をする目的プログラムを生成しているか？
- プログラムの同値性 — プログラムのある部分を、（おそらく効率の良い）別の形に書き直した時に、書き直しの前後でプログラムの意味は本当に同値か？

そのため、プログラムの"意味"を形式的に記述するために、さまざまな方法がこれまでに考えられて来た。

1.2 さまざまな意味論

プログラミング言語の意味論は大きく分けて、操作的意味論・公理的意味論・表示的意味論の3つに分類される。（ただし、この分類はある程度便宜的なもので、それぞれの境界がはっきりしているわけではない。）

操作的意味論 (operational semantics) は、仮想的な抽象機械を定義し、プログラムの意味をその抽象機械の内部状態の変化として記述しようという方法である。

公理的意味論 (axiomatic semantics) は、プログラムの意味を公理と推論規則により与えようとする方法である。ホーア論理 (Hoare logic) などがその代表的なものである。

表示的意味論 (denotational semantics) は、集合や関数のような数学的概念を用いて、プログラムの意味を記述しようとする方法である。

この講義では、主に表示的意味論に関連する事柄を紹介する。

プログラミング言語の"構文"の記述については偉大な先人の努力により、 BNF のような記法や `yacc` などの LALR パーサージェネレーターのようなツールが考え出され、仕様の記述や処理系の作成に欠かせないものとして、完全に定着している。

しかし、現在のところ、プログラミング言語の"意味"の記述については、 "構文"に比べて標準的な記法やツールが定着するには至っていない。

それでも、プログラミング言語の意味論について、これまで蓄積された知見は、上に挙げたようなさまざまなプログラミング言語の構成要素についての理解を深めるために必要不可欠なものとなっている。本講義ではこのような"意味論"の簡単なところを"つまみ食い"して行く予定である。

1.3 ラムダ計算

ラムダ計算（λ-calculus）は表示的意味論を展開するのに利用される計算体系である。 ラムダ計算は"関数"に関するもっとも単純な記法・体系であり、また、もっとも単純なプログラミング言語と考えることもできる。

本講義では、このラムダ計算を用いて、より複雑なプログラミング言語の構成要素の"意味"を記述していくことにする。

本来は表示的意味論では、ラムダ計算に加えて、D_∞ や $P\omega$ と呼ばれる領域（domain）に関する理論を展開する。 しかし、本講義では領域に関する理論については触れない。興味のある人は各自で参考文献を調べて欲しい。

ラムダ計算はシンプルでエレガントな体系ではあるが、これですべてを記述しようとすると、記述量がたいへん多くなり手に負えなくなる。そこで、ラムダ計算に基づくプログラミング言語である関数型言語 Haskell を紹介する。Haskell は、基本的にはラムダ計算にいくつかの構文上の糖衣（syntax sugar ― 本質的ではないが便利な記法）を追加したものである。そのため、その気になれば、Haskell のプログラムは簡単にラムダ計算の式に書き直すことができる。

1.4 接続

プログラミング言語のさまざまな構成要素の意味を記述する上でさまざまな概念を導入するが、なかでも接続（継続ともいう、 continuation）の概念は制御構造・例外処理・後戻りなど、さまざまな事柄を説明するために、重要な役割を果たす。また、接続はコンパイラーの設計の際にも用いられる。接続の概念を理解することは、プログラミング言語の意味論を理解する上で不可欠である。

また、オブジェクト指向言語の動的束縛などの諸概念についても触れる予定である。

1.5 さまざまなプログラミング言語

本講義では Haskell のほかに Scheme, Prolog, JavaScript など、様々なパラダイム (paradigm) のプログラミング言語を紹介する。また、プログラミングのさまざまなイディオムを紹介する。

下の図に示すように、第 2 次世界大戦後にコンピューターが生まれて 10 年後位に最初の高級言語が生まれ、その後、ワークステーション・インターネット・機械学習といった新しいコンピューターの応用分野が広まるにつれ、新しい言語が生まれてきた。

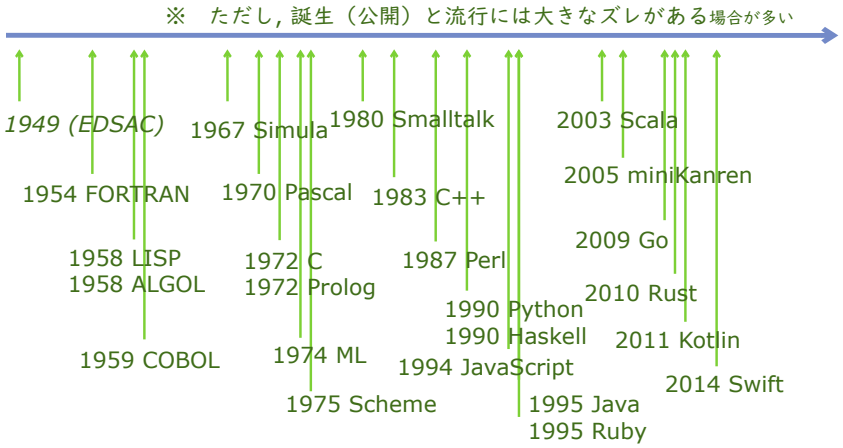

図: プログラミング言語の歴史

プログラミング言語の分類はいろいろな流儀があるが、主なものにパラダイムによる分類、型付けによる分類、実行方法による分類などがある。

- パラダイムによる分類
  命令型、オブジェクト指向、関数型、論理型
- 型付けによる分類
  静的型検査、動的型検査、他
- 実行方式による分類
  コンパイラー方式、インタプリター方式

言語によって得意分野が異なり、複数の言語を使い分ける必要がある。 また、複数の言語を知っていれば、ある言語 (A) では難しい処理を、別の言語 (B) では簡単に記述できる、という状況に出会うことがある。このときに、さまざまな都合により A 言語でプログラムを作成しなければならないとしても、 B 言語を知っていることによりプログラムの設計が容易になることがある。

例えば、オブジェクト指向言語と関数型言語は、どちらも命令型言語を拡張したものだが、方向性が異なる。オブジェクト指向言語は、GUI など用途に特化したデータ型を扱うような分野（例えばゲーム・シミュレーション）が得意である。一方、関数型言語は文字列・リストなど汎用のデータ型を扱うような分野（例えば言語処理系）が得意である（下図）。

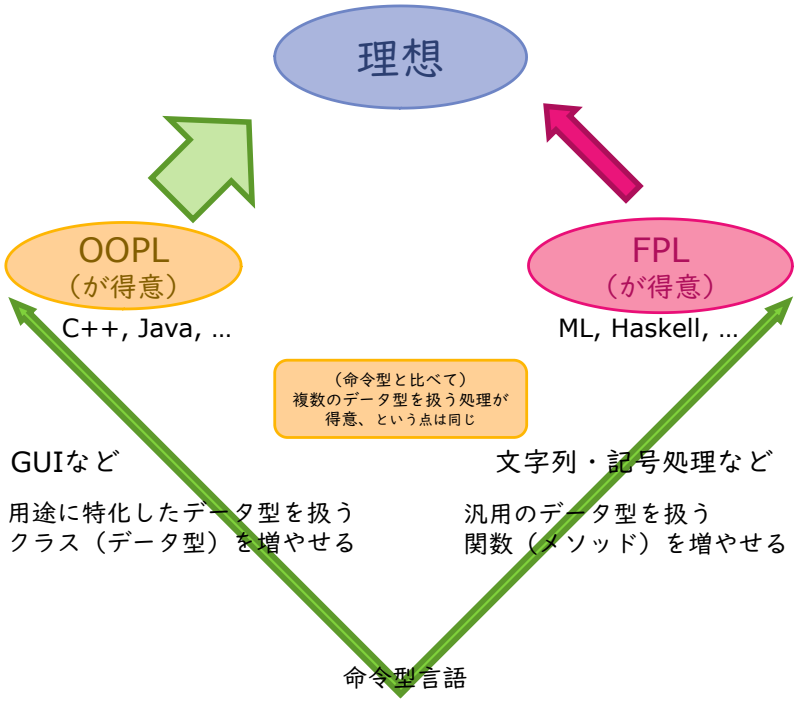

図: オブジェクト指向 vs. 関数型

一般的にスクリプト言語と呼ばれるプログラミング言語は動的型付けを採用していることが多い。動的型付けは実行前にチェックを行わないので、柔軟なプログラミングができて生産性が高い。一方でめったに使わない箇所を使ったときに、初めて不具合が見つかるということもありうる。そのため信頼性が必要とされる分野では使いづらい（下図）。

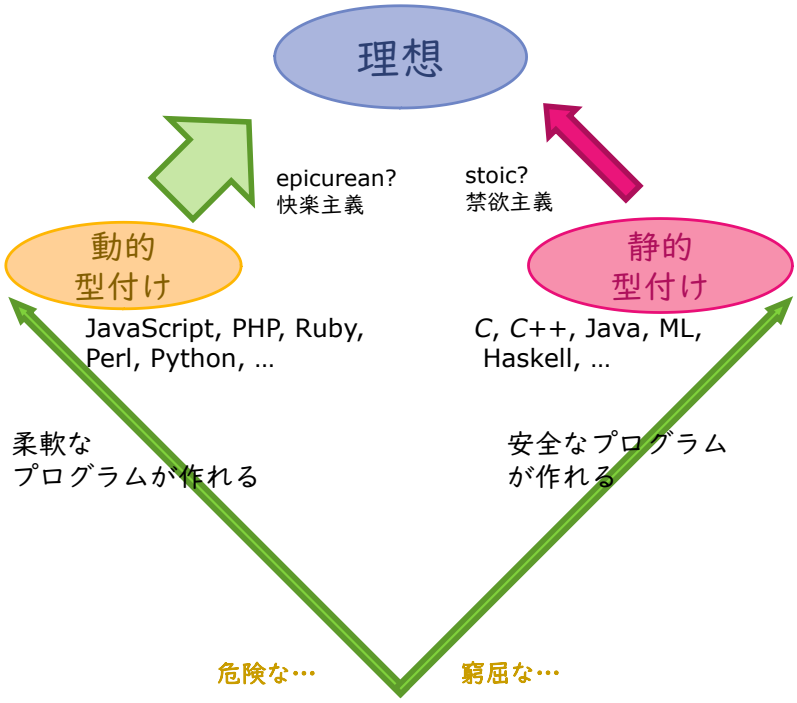

図: 動的型付け vs. 静的型付け

命令型やオブジェクト指向言語は"副作用"を多用する傾向がある。関数型言語は副作用を撤廃しているか、ひじょうにまれに使用する。副作用があると、プログラムの実行順序に実行結果が依存するので、プログラムの並列実行をしたときに予期しない結果となるときがある。副作用がないと並列実行しても結果が予期しやすいが、プログラムが書きにくいと言われている（下図）。

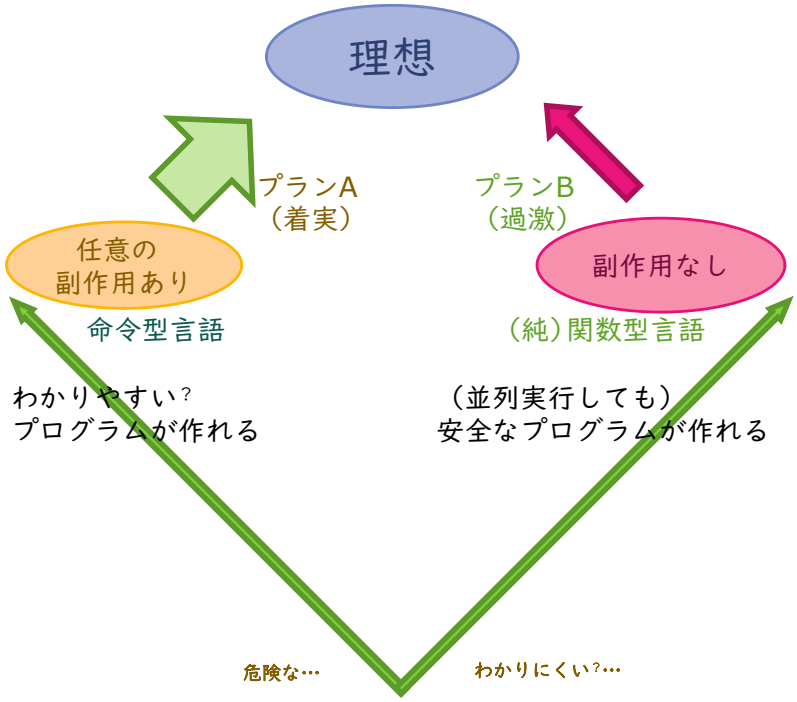

図: 副作用あり vs. 副作用なし

命令型言語やオブジェクト指向言語は共有の領域を書き換えていくことで計算が進むが、関数型言語は、新しいメモ用紙をどんどん使って、それを受け渡していくイメージである。

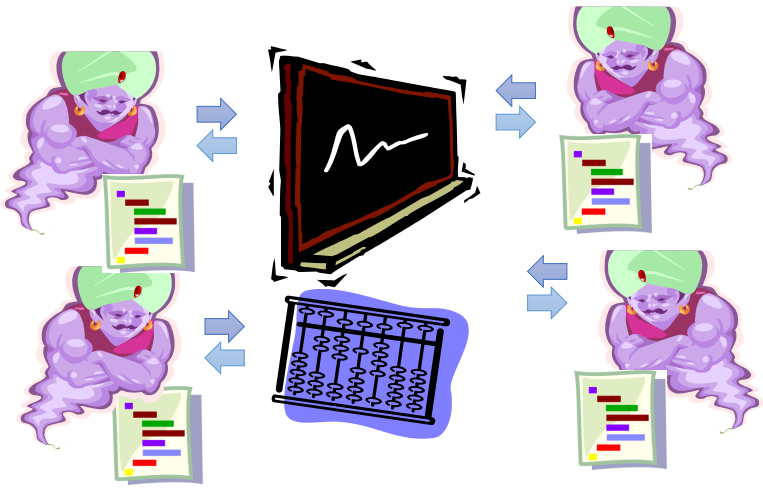

図: 命令型言語の実行のイメージ図

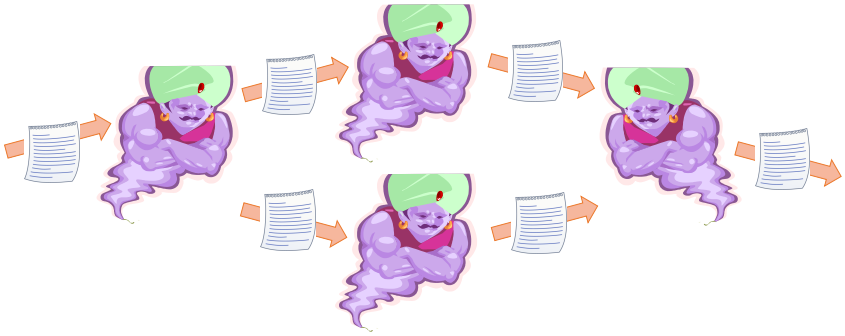

図: 関数型言語の実行のイメージ図

さらに、今後も当面は"万能の"言語が生まれるとは考えにくく、新しい言語が生まれ続けるだろう。あるいは、既存のプログラミング言語にも新しい概念が導入されていくだろう。しかし、新しいプログラミング言語といっても、まったく新しい概念ばかりからなるということはありえず、既存の言語から概念を借りてくることがほとんどである。そのため、複数の言語を知っていれば、新しい言語に対応するときにも有利である。

1.6 Haskell

Haskell は、代表的な関数型プログラミング言語であり、本質的に単純な構造を持っている。その代わりに型推論などを持つ高度な型システムを有している。副作用を撤廃し、代数的データ型 という OOPL のクラスとは異なる方向の拡張性を持つデータ型を提供している。多くの面で、メジャーな言語とは違う方向から理想に近づく言語ということができる。

この講義では、メジャーな言語と異なる特徴を多く持ち、かつ他の言語に大きな影響を与えている言語として Haskell を取り上げる。

Haskell はパズルや言語処理系など記号処理を得意とする。また、遅延評価は他の言語では難しいプログラムの組み立て方を可能にする。 Haskell を学習することは、既知の言語でのプログラミング能力の向上にも役立つと考えられ、また、新しい言語に Haskell の特徴が取り入れられることも多い。

1.7 さらに詳しく知りたい人のために...

この講義を構成するのに参考にした教科書を 2 つ挙げる。残念ながら、いずれも現在は入手困難である。

> **（中島 1982）** 中島 玲二 「数理情報学入門 — スコット・プログラム理論」 朝倉書店, 1982, ISBN4-254-11413-3
> **（武市 1994）** 武市 正人 「プログラミング言語 — 岩波講座ソフトウェア科学4」 岩波書店, 1994, ISBN4-00-010344-X
