プログラミング言語処理系論
Design and Implementation of Programming Language Processors

佐藤周行
（情報基盤センター/電気系専攻融合情報学コース）
schuko@satolab.itc.u-tokyo.ac.jp
http://www-sato.cc.u-tokyo.ac.jp/SATO.Hiroyuki/PLDI2018/

講義の目的(シラバス)

アルゴリズムを表現するためのプログラミング言語でかかれたプログラムを具体的なアーキテクチャに向けて最適なコードに変換することは、従来からコンピュータサイエンスの中心的な話題であったが、プログラミング言語の抽象度とアーキテクチャの複雑度の両方が飛躍的に増している現在、その重要度と問題の複雑度は飛躍的に増加している。
本講義では、プログラミング言語をどう設計するか、そこで表現する概念をどう実装するか、その時に処理効率をあげるにはどうしたらよいかを理解することを目標とする。

今年度の具体的な目標

- 現代的なプログラミング環境の理解
  - 動的な型付け
  - 高速プログラミングが可能な言語に対するリッチなAPI/ライブラリの提供
  - スクリプト言語でのAgileな開発環境
- 現代的なプログラミング環境の構築
  - スクリプト言語を実際に定義して書いてみる
  - VMを実際に定義してそこへのコンパイラを書いてみる
- （後述しますが）データ処理のための簡便な環境の構築は、（情報関係に限らず）研究を加速する

- （スクリプト）言語を設計、実装するのは実はそれほど難しくないが…
- 世間の状況（state of the art）をきちんと理解しておかないと…

- ということで、state of the artの説明をすることが目的の一つ

今日の予定

- 今回はオリエンテーションです。つまり
- この授業を取る（受講する）べきかどうかの決定をするための情報を与えるのが目的です
- 進行予定
  - 授業の進行予定
  - 背景の説明
  - プログラミングの変遷
  - プログラミング言語の歴史

講義予定

ここまで聞いて、少し手を動かすと、
自分でプログラミング言語を設計したく
なります（たぶん）

1. 授業導入とプログラミング言
   4/11, 4/18

(4/25はお休みにします。ごめんなさい

2. プログラミング言語の定義の手法(4回)
   5/2, 5/9, 5/16, 5/23

(5/30は木曜の振替日なんだそうです)

3. VMと実行時環境, 統合プログラミング環境(3回)
   6/6, 6/13, 6/20
4. プログラム解析と最適化(2回)
   6/27, 7/4
5. 進んだ話題--「正しさ」の保証(1回)
   7/11

具体的にはどういう授業か

- プログラミング言語の処理系についての話です
- 処理系については、最近の話題は「最適化」に集中しています(2014年はこう書いた)。 しかし…
- 次第に「プログラミング環境」提供の視点が優勢になってきました
  - 「高速化」は、依然、コンピュータアーキテクチャ系の研究やアプリケーションソフトの研究で行われている
- 処理系については、Webプログラミングに適した動的型環境をはじめとしたスクリプト言語を軸とした環境提供が当たり前のことになっています
  - Go, Swift, PHP, Python, …

つまり

- 現代的な意味でのプログラミング言語の需要は「スピード」だけではない
  - プログラムの生産性の重要性が明らかに。
    - High Performance Computing から High Productivity Computingへ（誰うま）
    - オブジェクト管理、型チェック、メモリ管理等、地味な仕事の自動化
    - 高速化させるところの局所化（ライブラリ提供等）
  - 標準規格の重要性が明らかに。
    - プログラミング言語を設計、実装したら、普及までさせないと努力が報われない
    - 専門チームでメンテするか、公的機関を使うか

そんなわけで

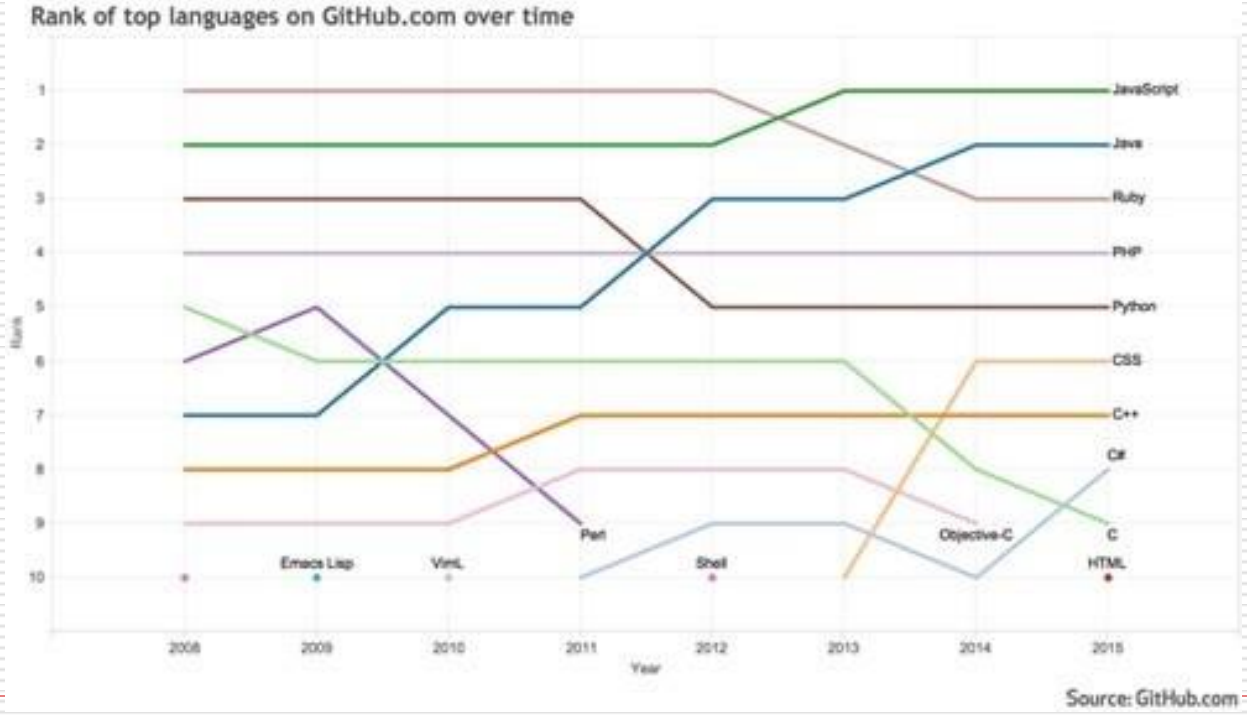
- 現在のプログラミング言語のトレンド
  - スクリプト言語の地位向上
    - Python, Perl, Ruby, …
    - 今回の講義はこちらをにらんで行います。
  - Domain specific 言語の台頭
    - PHP, JavaScript, …
  - 標準的な言語に対するExtension
    - Fortran 2008（Array Extension), X10 (Java Extension), …

プログラミング言語の人気の推移

- GitHub発表 2017 Oct.
  1. Javascript 2.3M
  2. Python 1 M
  3. Java 986K
  4. Ruby 870K
  5. PHP 565K
  6. C++ 413K
  7. CSS 335K
  8. C# 326K
  9. GO 285K
  10. C 239K

プログラミング言語の人気の推移

- GitHub発表 2015 Aug. – 順位はほとんど変動していない



やさしめの参考書ならいくつかある

- 2週間でできるスクリプト言語の作り方

- **単行本（ソフトカバー）:** 384ページ
- **出版社:** 技術評論社 (2012/2/10)
- **ISBN-10:** 4774149748
- **ISBN-13:** 978-4774149745
- **発売日**: 2012/2/10

こちらも学部生向けかな

- 古いし、間違いも多々あるけど

この講義の目的は

- スクリプト言語くらいは「ほしくなったら」負担を感じることなく「設計」くらいはできるようになること
- 手を動かしてプロトタイプくらいは自分で実装「できる気になる」こと
  - 上質のスクリプト言語は、優秀な実験器具と同じで生産性を著しく高めてくれる（多分）
  - 情報科学の分野に限らない。データ処理を計算機で行うところには必ずついてまわる

クラシックな話題…

- 言語処理系の王道は昔も今も実は「最適化」
  - High Performance Computing
  - コンパイラによる最適化
  - 細かいパラメタのチューニングによる個別の最適化

最適化にどの程度ページを割いているか

- 1986年版　150/750
- 2007年版 380/960

しかし

- 生産性が重視されるようになった
  - High Productivity Computing
  - プログラミングスタイル、開発環境、…
- 生産性には「セキュリティに対する耐性」が含まれるようになった(この話は、7月にやります)
  - CVE
  - セキュアコーディング

超高級言語/IDE/Script言語

- 超高級言語によるデータ処理のサポート
  - Matlab, Mathematica
- スクリプト言語
  - Agilityが最大の魅力
  - Perl, PHP, …
- IDE (Integrated Development Environment)
  - 特殊な開発環境に適応
    - 例：GPU向けのOpenCL
    - 生産性をあげるのが主目的

スクリプト言語

- データ処理量が爆発するにつれて、データ処理のための簡易言語を設計することがペイするようになりました
  - スクリプト言語の設計
  - PERL, PHP, …
- スクリプト言語は往々にして「いい加減なデザイン」に基づいています。
- スクリプト言語は、そのとっつきやすさから、ユーザが多くつくようになります。
- ユーザが多くなると、実装を再デザインして「まともな」プログラミング言語にすることがペイするようになってきます。

スクリプト言語(II)

- 「スクリプト言語」に求めるのは生産性
  - プラットフォーム独立性
  - 柔軟な型システムの提供
  - オブジェクトの簡便な管理
  - 管理された並列性の実現
  - …
- 速度は、高速ライブラリの提供で実現
  - 最低限の速度は、VMのISAで対応
  - C等とのインタフェース提供

（次回に話しますが）

- スクリプト言語の定義がどのようになされているか
  - Reference implementationとspecificationに関する誤解はほとんどのケースで絶望的なレベルに達しています
  - その他もろもろのカオス
- 標準的に定義されている「から」はやるというわけでもありませんが

そこで

- プログラミング言語の設計の「作法」を覚えておくのは、悪くない
- 「よいデザイン」「よい実装」についてのstate of the artを知り、直観を養うことで、たとえばスクリプト言語のスタートアップを効率的にできる

- ＝＞この講義の目的

具体的にはどういう授業か(I)

- クラシックなコンパイラとはどういうものか？

- 典型的な教科書
  Compilers Principles, Techniques, and Tools
  A. Aho et.al.
  ISBN 0321547985

クラシックなコンパイラの教科書では

- こっちのほうが年寄りには有名

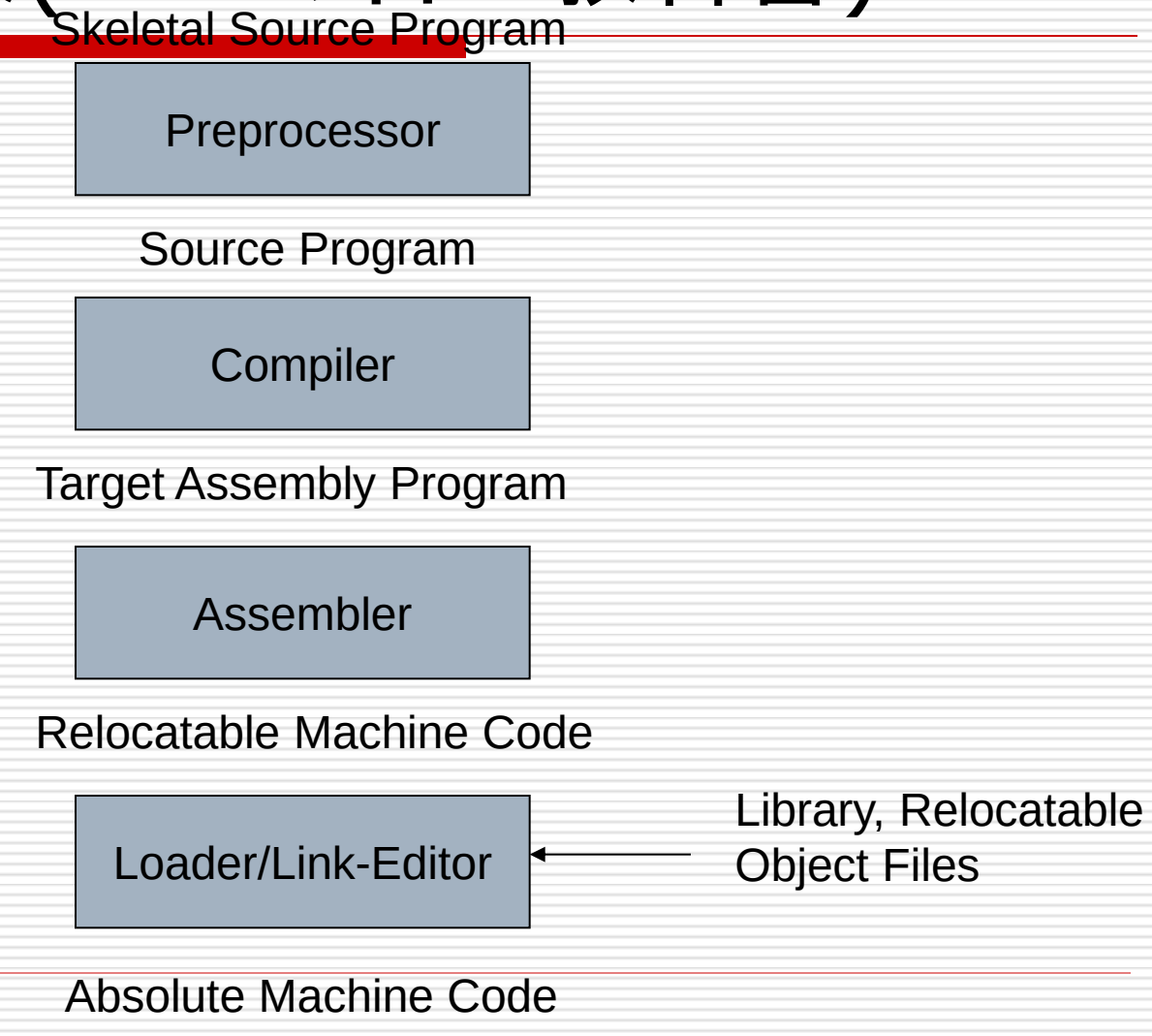
日本語の教科書

- 労作です

中田育夫
コンパイラの構成と最適化

出版社: 朝倉書店
(1999/09)
ISBN-10:
4254121393
ISBN-13: 978-
4254121391
発売日: 1999/09

言語処理系(Ahoの古い教科書)



Modern Compiler Construction

- 言語処理系のクラシックな研究は「最適化」に集中しています。

- Syntax Analysis/Semantic Analysisについては自動化が進み、ここをまじめに論じる人はいません
  - 知らなくて良いわけではない
  - 情報系の学部なら3年くらいの教材のはず
  - 一昔前までは大学院の入試によくでた

最適化にフォーカスをおいた教科書

- 中田本
- クラシックなものはこれ

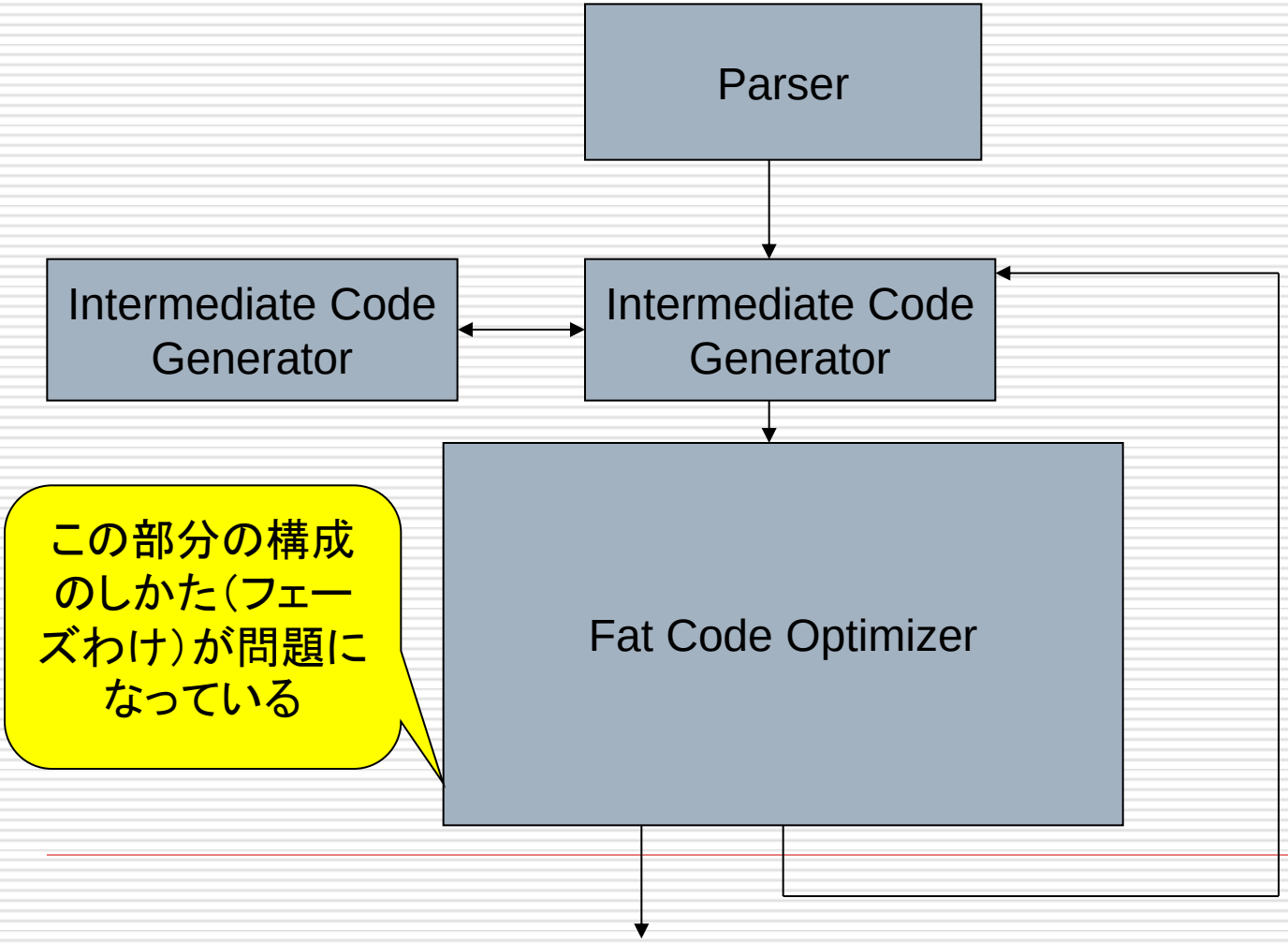
Muchnick, Steven S.

1997 Academic Press

Advanced Compiler Design and Implementation

ISBN 1-55860-320-4

Modern Compiler Construction



実行環境に関する理解

- 最近はコード生成だけでなく、実行環境(VM)の重要性も認識されている
  - クラシックな教科書は、適当なマシンを選んでコード生成のターゲットにしていた
  - 今は、JVMをはじめとして、標準的なWORKING PLATFORMを考えることができる
  - Jython (Python/JVM) その他異種言語のポート等いろいろ

プログラムの実行環境としてのVM

- 古いScript言語は、ソースをパースした「source tree」をそのまま実行していました (source tree traversal)
  - Perlはその代表例
- 実行環境がメモリ管理 （GC）、並列性管理を求められるようになると、VMを定義して、それをターゲットとする「コンパイラ」を作ることがはやりました
  - (Script言語ではないが)Java, すべての関数型言語
  - Python, Perl6その他

VMを論じ始めると…

- ISA
  - Stack Machine/Register Machine
  - Iteratorなどの「特殊」命令のサポート
- 並列性サポート
  - Frame設計
  - Thread/coroutine 等並列プリミティブの設計と実装
- メモリ管理
  - オブジェクト管理
  - GC

Parsing (I)

- ParsingをSyntax AnalysisとSemantic Analysisにわけて、それぞれに効率的な手法、自動化の手法を追求したのは1970年代の話です。
- LL(1)やLALRといった言語のクラスはParsingを楽にできる観点からCFGの部分クラスとして研究されました。(Chomskyもびっくり)

Parsing(II)

- プログラミング言語を定義するにはそれほど複雑な文法は必要ないというのが皆の共通の理解でした。
- C++がその常識に挑戦しました（文法のセンスを無視して機能を詰め込みすぎた結果のような気がします）
- 現在、C++は規格自体がメルトダウンしはじめているような感じがします
- C++の文法は(LALR＋アクション)ではかけないことがわかっています。Parsingは難しい
- Parser専門の会社があったほどです (http://www.edg.com/)

Parsing (III)

- 今はいろいろあってだな （PEG等）、しかし

- ParsingはCompilerでの中心的な話題からは引退しました。
- この講義ではLL(1), LL(*)とかLALRについての具体的な説明をすることはしません
  - 繰り返しになりますが、情報系の学科の出身なら、すでに学んだはず（覚えているかどうかは…）

- でも、Parserをかけないと、自分で言語処理系を書けませんから、たとえば「打倒PHP」とか、「打倒Ruby」と考える人はParserの書き方は学習しましょう

Parsing (IV)

Parseなんか、ツールを使えば楽にできます

「楽にできるようなツール」もいろいろでてきました

Pythonなんかは、文法を制限してParsingに凝らないようにしました

しかし、そんなこと言わずに

XMLのパースくらいはできるようになりましょう

- Regular Expressionとその受理オートマトンを含む (DTDの処理に必要)

- XMLの定義が読めるようになりましょう
  - BNF記法
  - 定義はここです
    http://www.w3.org/TR/2006/REC-xml11-20060816/

閑話休題

- 少し、個別の話題に足をつっこみすぎました。
- プログラミング言語とそれをとりまく環境の現状についての話に戻ります

プログラミングの変遷

- プログラミングの目的の変遷
- プログラミングの価値の変遷
- 目的と価値が決まれば、プログラムを何を使って表現するかの手段(プログラミング言語)の方向が決まる

今、もっとも需要があるプログラミング

- 今、もっとも需要のあるのはJavaでしょうか（Android向けのアプリをつくらなきゃ）
- でも、それと同等に需要のあるのはWeb関係
  - Webブラウザの動作記述
  - Webサーバの動作記述
    - テキストを柔軟に処理できるライブラリを自由に呼び出せる
    - 通常のプログラミングと同じ制御構造が使える
    - …
  - JAVASCRIPT, PHP, …

スクリプト言語

- Ruby, Perl, PHP, JavaScript, Python, …
- 言語仕様は結構シンプル
  - 自分で言語を設計できる
- 自分で実行環境を持つ
- ソース言語→VMのマシンコード→VM上での実行　といったものが多い
- 結構速いが、「遅くない」といった程度。最適化への関心はそれほど高くない（やることがなくなれば、当然ターゲットとして浮上する）
- 記述性やデータ変換の柔軟性にフォーカス

プログラミングの目的の変遷

- システムプログラミングの需要
  - コンピュータのためのプログラミング
  - メモリシステムのモデル化などの成果
  - ネットワークプログラミングはここに入るだろう
- 分散環境とクラウド上でのプログラミングの隆盛
  - ネットワークプログラミングの上のレイヤ
  - Embedded Systemとともに有力な場所
  - ターゲットの激変
  - Web環境でのプログラミング
    - サービスの概念　→　SOA
    - コードの移動
    - HTML,XML（数値や文字列だけではなく、より大きい文書やサービスを扱うプログラム）
- 実は、王道は昔から数値計算なのですが…

プログラミングの目的の変遷

- オブジェクトの発明
  - Alan Kay 1970ころから（Smalltalk 80）
  - 処理のパッケージ化とAPIの定義による抽象化
- より複雑なプログラミングへの対応
  - 複雑なオブジェクトへ
  - 複雑な継承と複雑なパターンへ
- 今では、ほとんどのプログラミング言語が「オブジェクト」の概念を言語内にもっています(Cは別)

オブジェクトへの反省

- プログラムの設計はしやすくなった
- 仕様の拡張に対応しづらい
- 新しい概念はないか…

プログラミングの価値の変遷

- 高速性から＋正しさ、＋生産性へ

- 高速性は常に「善」だった
  - 高速実行のためのコンピュータ
  - 高速実行のためのアルゴリズム
  - 高速実行のためのコンパイラ

  - 互いに影響しあう

高速実行のためのコンパイラ

- プログラムの可読性、移植性を高めるには、「プログラムは素直に書く」ことが基本

（それ以前に速いアルゴリズムが必須）

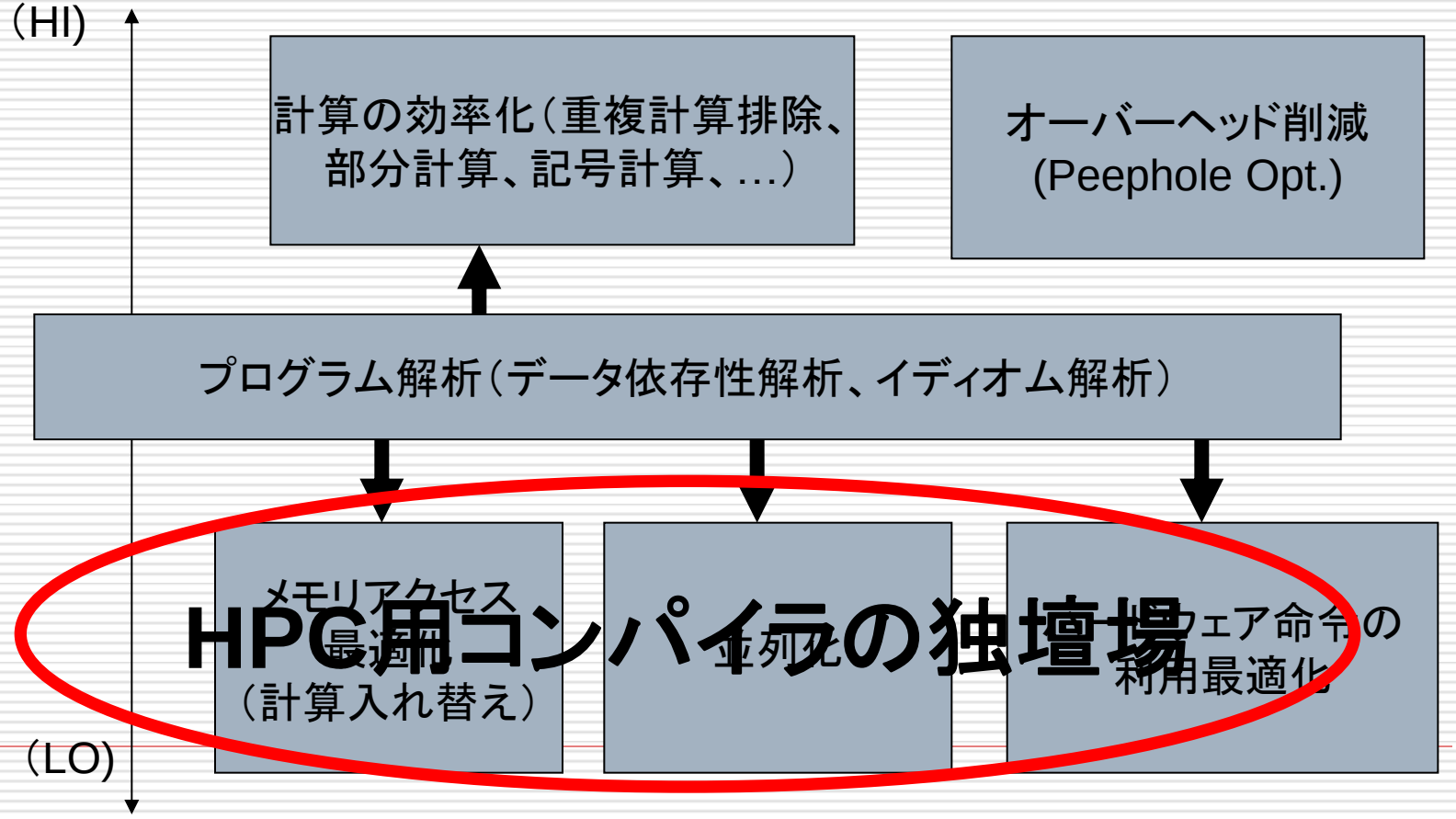
  - 「最適化コンパイラ」は、高速化のためにさまざまな最適化を行なう
    - 並列化
    - メモリ最適化
    - イディオム認識

コンパイラ最適化の機能と限界

- コンパイラ最適化は大きく進歩を見せ、現在ではソースコードからオブジェクトコードを類推することは困難になっているほどです。
  - どのような最適化が適用されたのか一目ではわからない
  - ソースコードをチューニングして、最適化と張り合おうとするのは愚の骨頂です
- でも、最適化は魔法ではありません
  - プログラム（静的）解析に基づくプログラム変換
  - アルゴリズムを解析してプログラム変換を行なうのは普通最適化といわない
  - 最適化　⊂　チューニング

高速化のための最適化

- 現在の最適化の流れ



高速化のための最適化

- HPC用の最適化は魅力的なテーマでした。
- 1980年代から2000年代前半にかけてデータ依存性解析に基づいたメモリアクセス最適化や並列性の抽出がさかんに研究されました。

- この講義では残念ながらこれをあまり扱いません。興味のある人は自習してください

プログラミングの価値の変遷

- 正しさへの要求の強まり
  - 分散環境では、データが外から飛んでくる（悪意を持ってデータを流し込んできたら…）
  - 分散環境では、コードが外から飛んでくる（悪意を持ってコードを流し込んできたら…）
  - セキュリティに関する要求の高まり
  - そもそも、最適化ルーチンは正しく動作しているのか？
- プログラムの複雑さのアップ
  - 人間が人手でチェックするには複雑になりすぎた
  - 個々の最適化の正しさを人間が証明することは大変
- プログラムの生産性アップへの要求
  - 大規模なプログラムを効率よく開発するための言語でのサポート、ツールでのサポートの要求

正しさへの要求 (incl. セキュリティ)

- この講義ではこのトピックを最後にちょっとだけ掘り下げます。以下のことを考慮することの重要性はますます高まっています。

- 言語処理系の特に最適化が間違ったコードを生成しないことの保証はどこに？
- 悪意をプログラミングできないようなプログラミング言語とは？
- コードが「正しい」ことを証明しやすいプログラミング言語とは？
  - JAVAでのポインタの追放
  - 強力な型システムの導入
  - コンパイル時・実行時でのコードチェック

プログラミング言語の変遷

- プログラミングの概念の変遷（さっきやった）
  - 数値計算以外を対象に
- オブジェクトの登場（さっきやった）
- 「生産性」が評価軸に
  - プログラムのエラー（いろいろなレベル）をコンパイル時にチェックしてくれないか
  - 少量のコーディングですまないか
  - 「自然に」コーディングできないか
    ＝　High Level Programming
  - 一度書いたコードを使いまわしできないか
  - 一度書いたコードを他のマシンでも使えないか

プログラミング言語の変遷

- 現在の主流

- 高いレベルの概念を直接扱えるように
  - アプリケーション指向超高級言語
- 型の登場
  - データ型と関数プロトタイプ
- 再利用への関心
  - モジュール、ライブラリ、APIの言語による支援
- 分厚いライブラリとパターンで援護されたプログラミング （プログラム何行…ということが無意味になっている）

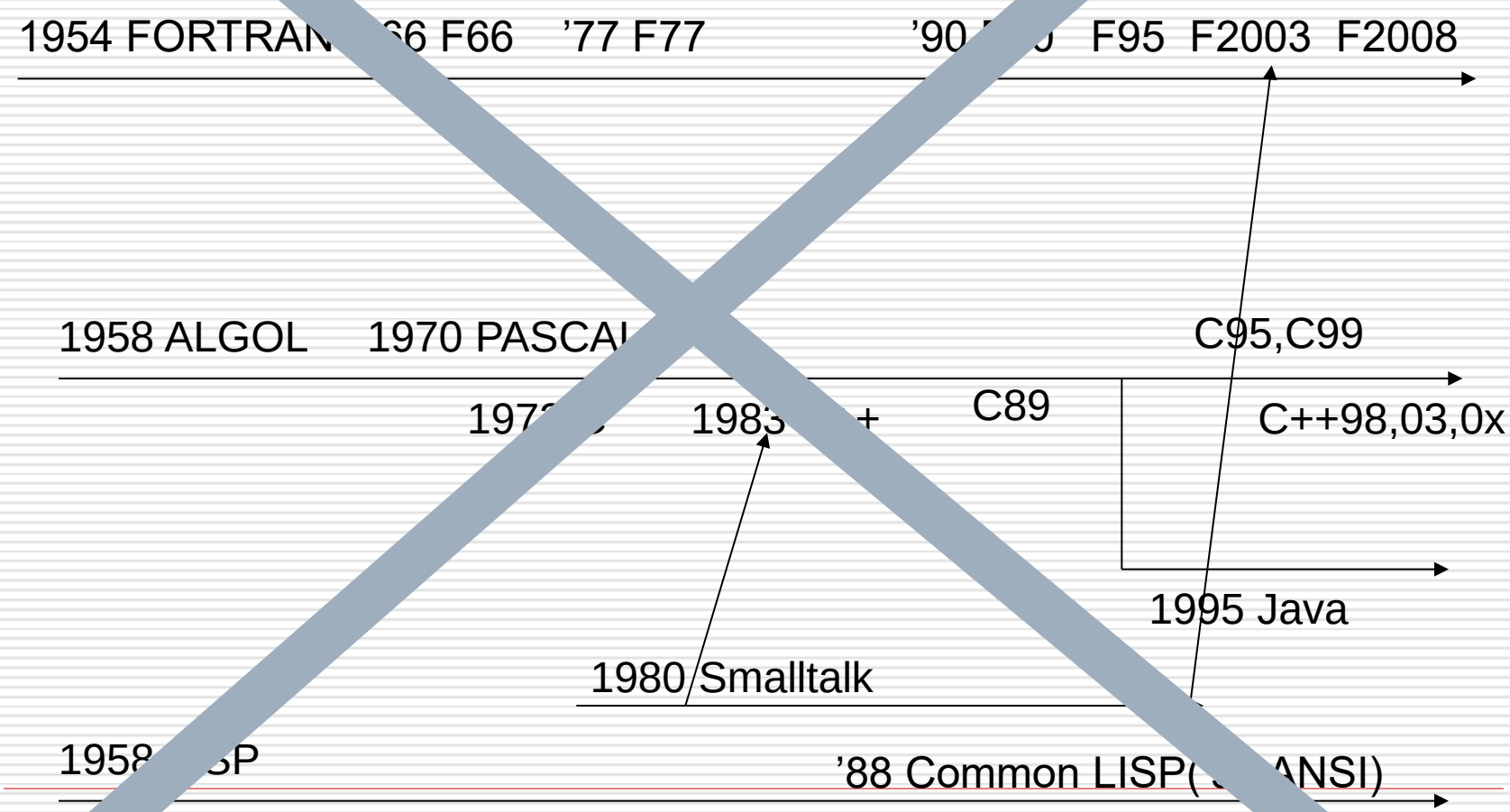
プログラミング言語の変遷

- でも、マシンが高級になるわけではない
- プログラムと物理の乖離が大きくなっている
  一方
- マシンが速くなっているから、速度よりも生産性を意識するならば、共通のVMのAPIをひとつ切っておけば楽ちんになる時代とも言える
- ここらへんのVMまで含めた言語処理系のデザインが重要になってくる
- （言語処理系）のデザイン

プログラミング言語の歴史

- Fortran, Algol系
- 規格、言語仕様の重要性の認識
- スクリプト言語
- JAVAの登場
- VMの登場

Programming Languages at a Glance
（年代記りになるほどこの手の図を出したがる）



Fortranの登場

- 高級言語の概念の提示
- BNFを使った定義
- 数式（変数を使った数学の式）を直接表現することに主眼があった
- Assembly言語（GOTOが主たる実行制御方式）＋数式

- 言語規格を制定するときに大きな議論の場を提供した

ALGOL系言語の登場

- アルゴリズムの記述に主眼
- 制御構造その他で、「ALGOL系言語」にくくられるタイプをごく初期から決定した

- 子孫がたくさんいる
  - C, C++,…

オブジェクトの発明

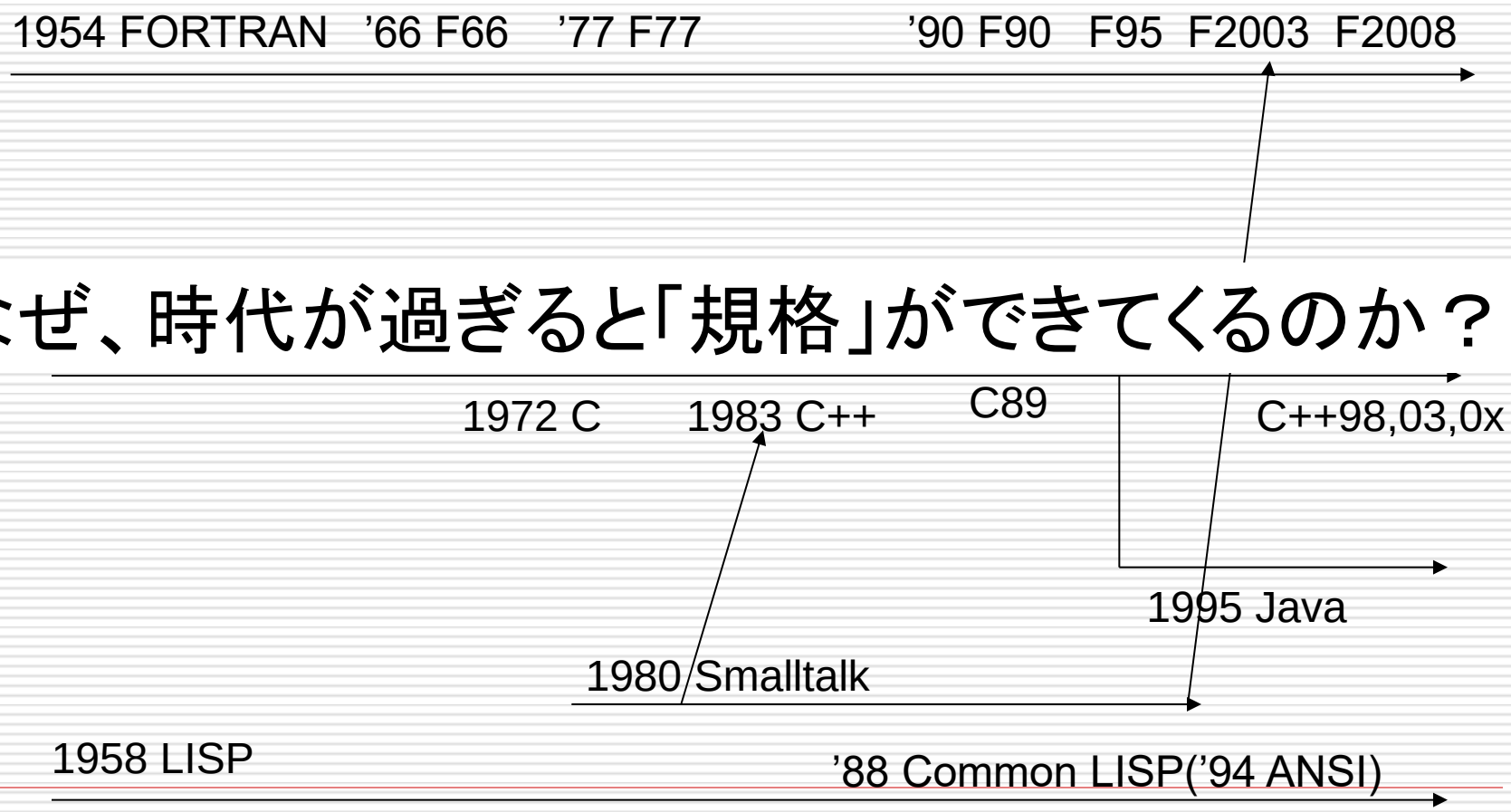
- Smalltalk 72,76,80
- 多くの言語がオブジェクトを扱えるようになった（扱える ＝ first class object）
- CのstructとC++のclassを混同してはいけない

現在のプログラミング言語の流れ

- オブジェクトはあたりまえ
- 並列性サポート、GCもついてくる

- 総合的なプログラミング環境の提供
  - プログラミング方法論の拡張
    - テンプレート、継承、仮想関数
  - モジュールなどの提供
    - Module, namespace
  - ぶあついライブラリ群の提供
    - パフォーマンスはここで稼ぐ（PythonのMLライブラリ）

最初の表に戻る


1954 FORTRAN
’66 F66
’77 F77
’90 F90
F95
F2003
F2008
なぜ、時代が過ぎると「規格」ができてくるのか？
1972 C
1983 C++
C89
C++98,03,0x
1995 Java
1980 Smalltalk
1958 LISP
’88 Common LISP(’94 ANSI)

プログラミング言語の規格(I)

- 初期のFortran
  - IBM Proprietary
  - Fortranが「使えるソフト」としてデファクトに
  - 各メーカがこぞってサポートを開始
  - 実装ごとに言語仕様を拡張 （そのうちのいくつかはよいアイデアとして他も採用）
  - 「Fortranプログラム」がコンパイルできるシステムとできないシステムがでてきた
- 規格の重要性の認識
  - ANSI (ISO)を主戦場とするか（C#）
  - 仲間を作って管理するか(各種コンソーシアム)
  - 一社（一者）で厳しく管理するか(Ada, Java)

プログラミング言語の規格(II)

- 規格の重要性の認識
- 規格を形式的に記述する技術の向上
  - SyntaxとSemanticsの分離
  - Syntaxは形式言語で(BNF)
  - Syntaxの足りないところは、BNFに対する注釈で

  - Semanticsはプログラムの実行の意味を決める
  - Semanticsは自然言語で記述

プログラミング言語の規格(III)

- Semanticsを記述する技術の向上が、言語の規格を厳格に定義することに大きく貢献した
  - 残念ながら、現在特定の形式主義に基づいたSemanticsの定義は行なわれていない（W3で無駄な試みがいくつか…）
  - Semanticsは自然言語で厳格に定義できる（数学が自然言語で展開されていることを考えればこれは驚くに足りない）
  - 必要だったのは、「形式主義」の理解と、それを遵守する能力（現在ではSemanticsを定める人間に大きな負担がかかっている）
- Fortranの規格を読んでみようか

プログラミング言語の規格

- 国際・国内機関
  - ISO
  - JIS
- コンソーシアム
  - IETF
  - W3
- 上2つは戦場です
- その他
  - RSA他、ガリバーが保守する規格

今、紙のメディアでの出版はありえない

規格を決める場は

- 「規格」を決める場（規格策定委員会）は、戦いの場です
  - 自分の実装の特徴を取り込ませることで、他社より優位に立つことを目指すことは普通に行われる
  - 技術的な優位性を主張するのは当たり前だが、それだけではない…
  - CrayのCoarray (Fortran)など
- 規格そのものがつぶされることもある
  - ISOにおけるJavaのfast track入りを＊＊が阻止
  - ECMAScript (Javascript)のISO規格改定放棄（2008）
    - ECMAScriptはつい最近もやらかしたようで…

そんなことに嫌気がさす場合は

- 規格（JIS, ISO, IEC, …）を定めることで、複数の団体の実装にお墨付きを与えることができるが
  - 規格になれば偉いというわけでもない
  - それは技術の話ではない
  - 互換性はビジネスの話
- そこで、きちんとした開発グループを決めてそこでメンテする方法が再び見直されている（スポンサーも付けば万々歳）。
  - Python
  - Perl
  - RubyはJISになりました（2011）が、それが正しい決定だったかどうかはよくわからない

この授業のテーマ(ブレークダウン)

- 「プログラミング環境」とは何か？新しい環境の考察
- プログラミング言語はどう設計すればよいのか？
  - 言語自身(型、名前空間、...)
  - 実行環境(VM)
- 性能向上のためになにをすればよいのか？

授業のテーマ（ブレークダウン）

- われわれが注意すべき「オープンな規格」とはなにか
  - プログラミング言語の規格を書けるようになるか？
- われわれが使えるツールとしては何があるのか？