プログラミング言語処理系論 (2)
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）

今日の予定

- プログラミング言語処理系の論点
  - 何を決めれば「プログラミング言語」になるのか？
  - プログラミング言語では「制御」と「データ」はどう表現されているのか
  - 実行のための環境とはなにか

- Web Page

http://www-sato.cc.u-tokyo.ac.jp/SATO.Hiroyuki/PLDI2022/

プログラミング言語の定義

- Imagine「自分でプログラミング言語を設計する」
  - 何を参考にしたらいいのか？
  - どこまでカバーしたら漏れがないのか？
  - 現代的な定義とは何か？
  - 定義は誰のためにあるのか？

大切なこと

- プログラミング言語の教科書の記述
  ≠
  マニュアル
  ≠
  プログラミング言語の定義

- Industrialな「プログラミング言語の定義」 ISO/IEC
  - Fortran, C, C++, …
- 提供ベンダーで「仕様」を保守
  - Java, C#, …
- Community-based メンテを行っているスクリプト言語等にみられる「定義」
  - Python, PHP, …

プログラムは何を表現しているか

- プログラム＝アルゴリズム＋データ構造
  （クラシックな見方）
  アルゴリズムをどう表現するかが一つの見方
- **アルゴリズムとデータ構造 (岩波講座 ソフトウェア科学)**

The Art of Computer Programming

- Knuthの教科書を読ませる大学は（日本では）まずありません。勉強するには不向きかも

具体的な言語の規格を見てみる

規格の章立てを観察してみる

- (ISO/IEC 1539-1 (Fortran 2018))
  - JISが2022年に制定されます
- 630ページ (phew!)

- ISO (International Organization for Standardization)
- IEC (International Electrotechnical Commission)

Fortran Conceptsとは

- コンセプト
  - High Level Syntax
  - Program Unit Concept
    - Program units and scoping units
    - Program
    - Procedure
    - Module
    - Submodule
  - Execution Concept
    - Statement Classification
    - Statement order
    - The END statement
    - Program execution
    - Execution sequence
    - Image execution states
    - Termination of execution

- Data Concept
  - Type
  - Data value
  - Data entity
  - Definition of objects and pointers
  - Reference
  - Array
  - Coarray
  - Established coarrays
  - Pointer
  - Allocatable variables
  - Storage
- Fundamental Concepts
  - Names and designators
  - Statement keyword
  - Other keywords
  - Association
  - Intrinsic
  - Operator
  - Companion processors

High Level Syntax

R501 program is program-unit
[program-unit] ...
R502 program-unit is main-program
or external-subprogram
or module
or submodule
or block-data
R1401 main-program is [program-stmt]
[specication-part]
[execution-part]
[internal-subprogram-part]
end-program-stmt

目次は続く

6 Lexical tokens and source form
7 Types
8 Attribute declarations and specifications
9 Use of data objects
10 Expressions and assignment
11 Execution control
12 Input/output statements
13 Input/output editing
14 Program units
15 Procedures
16 Intrinsic procedures and modules
17 Exceptions and IEEE arithmetic
18 Interoperability with C
19 Scope, association, and definition

プログラミング言語の定義とは？

- 言語の持つコンセプトを列挙し、
  - それぞれのコンセプトを実現する
    - Syntax, 実行モデル, Semantics
  - 言語の代表的なコンセプトはExecutionとData
    - 少しブレークダウンすると…
      - Type（は何のためにあるのか？）
      - Object（は何のためにあるのか？）
      - Parallelism（は何によって実現されるのか？）
    - Procedures, module
    - 組み込み/predefinedの関数群
- ≠プログラミング言語を利用する

Cではどうか

- Foreword
- Introduction
- 1. Scope
- 2. Normative references
- 3. Terms, definitions and symbols
- 4. Conformance
- 5. Environment
- 6. Language
- 7. Library

- Annex A Language syntax summary
- Annex B Library summary
- Annex C Sequence points
- Annex D Universal character names for identifiers
- Annex E Implementation limits
- Annex F IEC 60559 floating-point arithmetic
- Annex G IEC 60559-compatible complex arithmetic
- Annex H Language independent arithmetic
- Annex I Common warnings
- Annex J Portability issues
- Annex K Bounds-checking interfaces
- Annex L Analyzability
- Annex M Change History
- Bibliography

Fortran規格で何が決められているか

- まずはプログラムの構成
- Program units are the fundamental components of a Fortran program. A program unit is a main program, an external subprogram (subroutine or function), a module, a submodule. (or a block data program unit.)

Hello World

- program hello
  print *,'Hello World!'
  end program hello

- こんなものを想定している
  program main
  do i=1,100
  …
  end do
  subroutine ..
  …
  end subroutine
  end program main

module abc
  subroutine …
  …
  end
  …
end module abc

Execution Control

- 次に制御構造
  (5.3.4 par 1) Execution of a program consists of the asynchronous execution of a fixed number (which may be one) of its images.
- ここで、image, asynchronous executionという概念が出てくる。
  (5.3.4 par 2) A team is an ordered set of images that is either the initial team, consisting of all images, or a subset of a parent team formed by execution of a FORM TEAM statement.

- 並列性を制御するための記述 （Fortranは、賛否両論あったが、並列性を言語に内包したproduction levelの仕様を定めた）
  - 「実行」とは何かを定義することが必要になった
  - もちろん、自明ならば、ここは下の赤線部分だけでよい。

- 話を続ける
  (5.3.4 par 3) During execution, each image has a current team, which is only changed by execution of CHANGE TEAM and END TEAM statements. Image indices, and thus coindexing of variable names with an image-selector, are relative to the current team unless a different team is specified. Initially, the current team is the initial team.

ここだけの補足

- Fortranの並列サポート
  - (昔の)Cray Fortran – SPMD with coarray（並列実行モデルを指定）
  - 今のFortran
    - ImageでのSPMDの考えを残しつつ
    - Teamで、「連携する」imageを指定 -- topology指定
    - Asynchronousに実行される。メモリ同期をサポート
  - ということで、MPIを使うのとあまり変わらなくなっている

定義の仕方の典型例

R1119 *do-construct* **is** *do-stmt block end-do*

R1120 *do-stmt* **is** *nonlabel-do-stmt*

or *label-do-stmt*

R1121 *label-do-stmt* **is** [*do-construct-name* :] DO *label* [*loop-control*]

R1122 *nonlabel-do-stmt* **is** [*do-construct-name* :] DO [*loop-control*]

R1123 *loop-control* **is** [,] *do-variable* = *scalar-int-expr*, *scalar-int-expr* [, *scalar-int-expr*]

or [,] WHILE (*scalar-logical-expr*)

or [,] CONCURRENT *concurrent-header concurrent-locality*

R1124 *do-variable* **is** *scalar-int-variable-name*

C1120 (R1124) The *do-variable* shall be a variable of type integer.

R1125 *concurrent-header* **is** ([*integer-type-spec* ::] *concurrent-control-list* [, *scalar-mask-expr*])
R1126 *concurrent-control* **is** *index-name* = *concurrent-limit* :
concurrent-limit [: *concurrent-step*]
R1127 *concurrent-limit* **is** *scalar-int-expr*
R1128 *concurrent-step* **is** *scalar-int-expr*
R1129 *concurrent-locality* **is** [*locality-spec*]...
R1130 *locality-spec* **is** LOCAL (*variable-name-list*)
or LOCAL_INIT (*variable-name-list*)
or SHARED (*variable-name-list*)
or DEFAULT (NONE)

- この「Syntaxを定める文法」については、次々回になってから詳述します

- BNF（CFG）で規則(Rxxx)を書いて、そこで記述しきれない部分を制約(Cyyy)で書く。
  - R部分からparserの自動生成が可能
  - C部分は、parserへアクションとして組み込むことで実現

Data

- Type　—type is a named categorization of data that, together with its type parameters, determines the set of values, syntax for denoting these values, and the set of operations that interpret and manipulate the values.
  - Intrinsic Type（integer, real, complex, character, logical）← 他の言語ではprimitive typeとも
  - Derived Type
- +CLASS constructor (polymorphic type用)
- +constructors (enum, array)

- Data Entity
  - Data object
    - Fortranではscalarとarray、さらにそれらのsubobjectを扱う
  - Variable
  - Constant
  - Expression
  - Function reference
- Definition of objects and pointers
- References
- ここから先はFortranで重要なデータ構造の解説
  - Array
  - Coarray
  - Established coarrays
  - Pointer
  - Allocatable variables
  - Storage

Fortranの特徴づけ（とてもラフに）

- 実行制御として、image, teamの導入とそれによる並列実行の表現
- データ構造として、scalarとarrayを実体として、それらから派生する型、ポインタを持つ
- coarrayは、並列実行とともに導入され、並列性を表現することができる
- Fotranはone for allを昔から（今も）標榜する

By contrast, C

- Cは並列性を表現しない
- Cはオブジェクトを表現しない

→他のCをベースにした言語拡張で実現してくださいというポリシー

- 課題1　Javaの言語仕様を入手し、同様の解析を行なってみよ。仕様はここ↓

https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf

Javaは、Oracleが仕様を保守する、典型的なオブジェクト指向言語である。オブジェクトを核として、パッケージ等、プログラム単位がどのように定義されているか記述せよ。

課題1(1′, 1″)についての注意

- 解析すべきプログラミング言語としての特徴のポイントはFortranとほぼ同じです
  - 並列性を含む実行制御における特徴。Javaならthreadとメモリモデル
  - データ実体の表現。特にオブジェクト指向ならばクラスを中心にした解析
  - Syntaxの定義には、本質的にBNFが使われていることを観察してください

プログラミング言語処理系の論点

- 言語仕様のベースになる考え方
  - 何をどう表現するか？（アルゴリズム＋データ構造）
  - 実行モデルは？　　ここまでを話してきました
- 処理系をどう作るか？
  - Community-based　メンテをしているところは、自らが管理すればよい
  - 複数ベンダーが競合している場合はどうなのか？
- そこで規格
  - 規格 means「（規格中の）要件を満たしているならば、規格準拠とみとめなくてはならない」
  - 複数のベンダーが言語処理系を提供している場合、「規格準拠」かどうかで争いがおこるのは必然

スクリプト言語では…

- 作成の経緯がintelligent toolの扱いであることがとても多い(かった)
  - プログラミング言語ではそもそもなかった…
  - Perl for shell + awk + …
  - PHP for http handling
- Community basedな保守が多くみられる
- 結論
  - スクリプト言語では、これまでのような議論がsquareな形でなされないのが普通だった
  - Rubyのように言語規格が定められるほうが例外
    - 規格があるから偉いというわけではない

例外としてのJavascript

- ECMAScriptとして保守
  - 最新はEcmascript 2021
  - https://262.ecma-international.org/12.0/
  - たくさんのベンダーが存在して実装
    - Mozilla, IE, Chrome, Safari, Konqueror, iCab, .NET, Flash, Acrobat, …
    - 政治の場になっている(らしい)

ISO/IEC, or JIS

- ECMAScriptにはISO規格があって

JIS規格もあって

- JIS X3060:2000
  - 2000年に制定されてそのまま
- プログラミング言語にJISは必要か？
  - 皆が皆、英語がわかるか
  - 国内市場で共通のtermを用いなくなることは「先進国」からの転落にならないか

- Ecmascriptでは
  - Conceptはあきらか
  - Syntaxもきれいに定められる
  - Semanticsこそが重要
    - オペレーションごとに意味を記述
- 彼らは、概念をきれいにまとめ切れているか？

- 課題1’ Ecmascriptの言語仕様を入手し、同様の解析を行なってみよ。

Ecmascriptは、Web Browserを実行環境と想定するスクリプト言語である。言語のコンセプトが何かを中心に、仕様書の構造を解析せよ

Pythonはどうか

- Pythonは、

https://docs.python.org/3/reference/grammar.html

違う、こういうのを見に来たんじゃない

Pythonの言語仕様をめぐる混乱

Python2系と3系

複数の実装（Cpython, PyPy）

このような戦いを経て、言語は成熟していく

惨状ならばPHPだって負けてはいない

- http://php.net/manual/en/ を見ると

- PHP Manualの名の下で以下の章立て
  - Getting Started
  - Installation and Configuration
  - Language Reference
  - Security
  - Features
  - Function Reference
  - PHP at the Core:A Hacker’s Guide
  - FAQ
  - Appendices

Python

- さっき悪口言ったけど、Pythonは実はましです
- https://docs.python.org/3/reference/index.html

- 書き方は、「規格」の標準的なものではない
  - おしゃべり
  - 「準拠性」を定めるのではなく、マニュアルを拡張した性格を持つ
  - Manual ≠ Specification

Rubyについて

- 課題1″ Rubyの言語仕様を入手し、同様の解析を行なってみよ。JISが定められているが、入手しやすい仕様はここ↓

http://www.ipa.go.jp/files/000011432.pdf

- JIS X3017:2013
- Rubyの最新は3.1.1(2022.2版) JISとの関係は不明(recall Javascript)
- Rubyもコミュニティベースで保守されている典型的なオブジェクト指向言語である。ここでは、FortranやJavaと比較して、言語の定義がどの程度成熟しているか併せて観察せよ

ちょっと脱線

- 規格を作る動きは以下を契機に出てくる
  - 実装が複数出てきた場合
    - 自然発生的に…
      - Ruby, Perl5 → Perl6
    - 大元の作成者の責任が問われるようになった場合
      - PDF　少し古いが、標準化、規格化に関する姿勢がうかがえる文書がある (http://www.itmedia.co.jp/im/articles/0810/28/news130.html)
    - 大元の作成者が主導権を離したくない場合
      - Java

プログラミング言語の規格(I)

- 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(II)

- 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）

- ここまで準備ができたなら、FortranやC++の規格を読めるはず（かな）

何をどう表現するか？

- アルゴリズム
  - 制御構造
  - サブプログラムの構造
    - 引数
    - フレーム
    - …
- データ構造

- 全部合わせて「オブジェクト」に
  - これはこれで少し乱暴な議論ではある

アルゴリズムの表現

- 高いレベルの表現 – Algol系言語
  - 条件分岐と繰り返し
  - If, for, while,
  - 関数呼び出し、再帰
- 低いレベルの表現 -- Fortran
  - フローチャート
  - Compare, goto

アルゴリズムの表現(＋並列性)

- 従来の実行制御モデルの他に
- (追加)並列性の表現は現代的な言語での課題の一つ
- 本来は実行モデルの要素としてアルゴリズムの表現から切り離されていたが…
- 制御部の並列実行の表現
  - 並列DO構文 (Do Concurrent)
  - 同期
  - Coroutine (yield)
- 「データを並列に処理する」概念のimplicitな表現
  - データの配置とアクセス制御 (PGAS等)
  - データ並列

最近落ち着いてきたのは…

- プログラムはスレッド(Fortranではimage)内のロジックを表現する
- スレッドは並列に実行される
- スレッド間の通信が何らかの形でなされる
  - Explicit communication (MPI style)
  - Implicit communication (data r/w)
- スレッド間の同期、criticalityの表現が導入される

データ構造の表現

- 高いレベルの表現
  - グラフ、リストなどのデータ構造
  - 構造体など、ひとまとまりのデータを表現する
  - 再帰的なデータ構造の定義を許す

  - 抽象データ型
  - データを操作する関数群を「データ」に含める
  - それ以外の操作（含アクセス）を禁止する

- 低いレベルの表現
  - マシン内部の構造の表現
  - 整数、実数、ポインタ

プログラミングにおける「概念」

- アルゴリズム＋データ構造を表現できれば、プログラミング言語としては一応合格ですが…
- どう考えながら「プログラミング」するか、というスタイルが問題になるようになって来ました
  - object
  - function
  - aspect
  - …
- こうして、プログラミング言語は「何を表現できるか」を競うようになりました
  - ツールとしてのプログラミング言語からの進化

論点（2）実行モデル

- プログラミング言語はプログラムを書くための言語であるが…
- プログラムは実行されてはじめて意味を持つ

- Semanticsとは、プログラムの持つ（計算上の）意味のことをいう

Semanticsの記述

- 代表的なものは2つ
  - 計算機械をひとつ仮定した上で、プログラムがその機械上でどのように振舞うかを記述する
    - Operational Semantics
  - ↓のような分野はあるが、言語処理系に関係しては論じないことにする
  - ある数学的な計算モデルを作った上で、プログラムがその計算モデル上のどのような対象として解釈されるかを記述する
    - Denotational Semantics

計算機械

- 計算はどのように進むか？
  - 変数と、そこに割り当てられている「値」のペアの集合を考える
  - それぞれの集合を「状態」ということにする
  - 「環境」ということもある。環境は名前と値のペアにもっぱら使われる
  - プログラムの各命令は、状態を入力として状態を出力する関数であると思うことにする

実行モデル

- 今後、プログラムの実行というときには「計算機械」を意識的、無意識的に考える
- Semanticsを記述するために、この「計算機械」の上でどういう振る舞いをするか、ということを記述することが良くある。

- 実行モデルとは、この「計算機械」のことをいうことにする。
- 傾向として、なんらかのアナロジー（社会と人間等）でモデルを与えるものがある（特に分散計算に関係するもの）

最近の傾向(Cont'd)

- 「Model by Analogy」は、数学的にいろいろな性質を証明しやすくするための枠組ではない
  - エージェントどうしがインタラクトすることでいったい何が明らかになるのか？
  - 人間界と同じように何かが進行することはわかるが、ではそれは、数学的に何かが証明できるか？
- よって、少し違和感があるが、そんなことを気にしない人が増えていることも事実である。そのかわり
  - プログラム解析の様々な方法論が発展してきた
    - (データ | コントロール)フロー解析、スライシング、型理論、types and effect systems
  - 必要な性質は、それら方法論を使って証明しよう
- この動きは、「プログラミング言語理論」と「プログラム解析理論」が分離し始めたことを表している

最初の問題

- 「値」とは何か？
  - 基本型だけではない
  - Scalar＋Array（Fortran）
  - Tuple, list, (Python)
  - Object
- 計算機械の上での実装（表現）の検討の必要性（処理系の検討には必須）
  - Referenceのメカニズムの検討
- オブジェクト管理、メモリ管理 (in VM) へつながる

ちょっと見てみる

- Perl
  - stringが基本型のひとつとして入る
  - scalar ($xの形) + array (@xの形)
  - Objectは…
    - 厳密に言えばオブジェクト指向のようなプログラミングスタイルをpackageを導入することで提供
- Python
  - 基本型＋tuple+リスト
  - Objectは最初からデザインの中に入っている

次の問題

- 関数(メソッド)呼び出しはどうか？
  - 仮引数と実引数の名前の対応をどうとるか？
  - 変数のバインディングをどう表現するか？
    - Call by value, Call by Reference, Call by Name, …
  - ローカルな変数とグローバルな変数の違い
- 関数呼び出しごとのフレームはどうか？
  - 特に再帰に関して典型的に出てくる
  - 関数コールごとに異なる実行状態(フレーム)を作り出すことをどう表現するか？
  - フレーム管理は並列性の表現と結びつくと途端に難化
  - 「フレーム」は、Cではスタックのようなマシンに密着したものとして、Pythonでは変数と値の結合のリスト(DIC)として与えられる

関数コール、フレーム管理のSemanticsの例

- Javaの規格
  - 15.12.4 Runtime Evaluation of Method Invocation
  - 15.12.4.1 Compute Target Reference (If Necessary)
  - 15.12.4.2 Evaluate Arguments
  - 15.12.4.3 Check Accessibility of Type and Method
  - 15.12.4.4 Locate Method to Invoke
  - 15.12.4.5 Create Frame, Synchronize, Transfer Control
  - 15.12.4.6 Example: Target Reference and Static Methods
  - 15.12.4.7 Example: Evaluation Order
  - 15.12.4.8 Example: Overriding
  - 15.12.4.9 Example: Method Invocation using super
- Fortranの規格
  - 実引数、仮引数および引数結合(association)

さて、実行モデルの記述

- Virtual Machineを定義する
  - Operational Semanticsの忠実な表現
    - ISAを定義する(大別してstack machineとregister machine)
    - プログラムの意味とは、VM上にコンパイルされたコードのVM上の意味と定義する
    - ハードウェアシステムとは薄皮一枚で隔てられている
  - 「状態」の記述
    - メモリの状態
    - スレッドの状態
  - 「状態遷移」の記述
    - 主だった命令がマシンの状態をどう変化させるかを記述
- VMで行うのは、計算機械の「実装」だけではない
  - オブジェクトの生成・GCに係るメモリ管理
  - 関数呼び出しに係るフレーム管理、並列性に係るスレッド管理
  - 並列性の管理をするところもありまして…

Java VMの定義

- 典型例としてJava VMがある
  - 3 The Structure of the Java Virtual Machine
    - 3.1 The class File Format
    - 3.2 Data Types
    - 3.3 Primitive Types and Values
      - 3.3.1 Integral Types and Values
      - 3.3.2 Floating-Point Types, Value Sets, and Values
      - 3.3.3 The returnAddress Type and Values
      - 3.3.4 The boolean Type
    - 3.4 Reference Types and Values
    - 3.5 Runtime Data Areas
      - 3.5.1 The pc Register
      - 3.5.2 Java Virtual Machine Stacks
      - 3.5.3 Heap
      - 3.5.4 Method Area
      - 3.5.5 Runtime Constant Pool
      - 3.5.6 Native Method Stacks
    - 3.6 Frames
      - 3.6.1 Local Variables
      - 3.6.2 Operand Stacks
      - 3.6.3 Dynamic Linking
      - 3.6.4 Normal Method Invocation Completion
      - 3.6.5 Abrupt Method Invocation Completion
      - 3.6.6 Additional Information
    - 3.7 Representation of Objects
    - 3.8 Floating-Point Arithmetic
      - 3.8.1 Java Virtual Machine Floating-Point Arithmetic and IEEE 754
      - 3.8.2 Floating-Point Modes
      - 3.8.3 Value Set Conversion
    - 3.9 Specially Named Initialization Methods
    - 3.10 Exceptions
    - 3.11 Instruction Set Summary
      - 3.11.1 Types and the Java Virtual Machine
      - 3.11.2 Load and Store Instructions
      - 3.11.3 Arithmetic Instructions
      - 3.11.4 Type Conversion Instructions
      - 3.11.5 Object Creation and Manipulation
      - 3.11.6 Operand Stack Management Instructions
      - 3.11.7 Control Transfer Instructions
      - 3.11.8 Method Invocation and Return Instructions
      - 3.11.9 Throwing Exceptions
      - 3.11.10 Implementing finally
      - 3.11.11 Synchronization
    - 3.12 Class Libraries
    - 3.13 Public Design, Private Implementation

もしくは

- https://docs.oracle.com/javase/specs/jvms/se18/jvms18.pdf(2022)
- (課題2)上のドキュメントを以下の観点から要約せよ [この課題は、6月にもう一度出すので今回は無視してもよい]
  - VMの仮想機械としての構造
  - Constant pool, method area等、実行に特徴的な領域の特定と、実行開始までの流れ(startup)

Python VM

基本、変化なし

- どこかの書き込み(2004)

Xavier> Is there any paper describing the instruction set of the Python
Xavier> Virtual Machine or should I learn anything from Python/ceval2.c?

I think right now Python/ceval.c is your only choice. I recently began working on something, but really have nothing more than a crude outline. If
you start looking at ceval.c and get stuck, drop me a note. I'd be happy to
try and explain things. It might motivate me to work on pyvm.tex a bit more
as well.

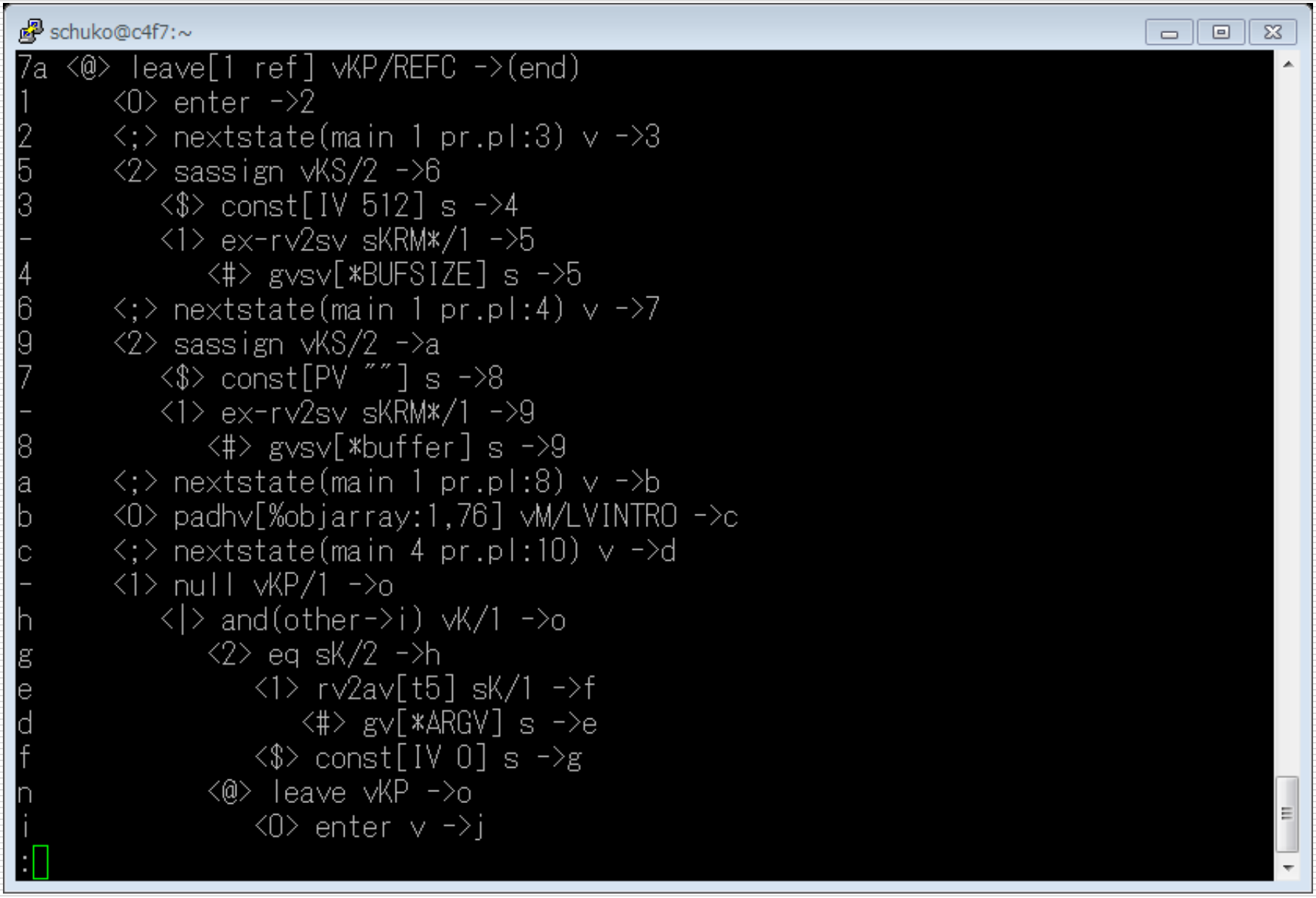
ここでしょうか（ceval.c)(泣) https://github.com/certik/python-3.2/blob/master/Python/ceval.c
もしくはdisassemblerの方を見た方がよいでしょうか
https://docs.python.jp/3/library/dis.html#python-bytecode-instructions

Java VM

- Java VMもPython VMもスタックマシンです
- JVMの当初の目的は、安全な形でコードの流通性を高めることであった（特にアプレット）
  - コードの流通性の確保が目的の一つに入っていた
- JavaのSemanticsはJava VMの上で定義することができる
- 速度を考えてJITが出現した
  - 速度はこんな技術で稼ぐんじゃないんだけど…
  - JITを実装してスピードアップを図るものとしてのRpythonの試みは後日紹介します
  - 今世紀初めの技術ではあるな

で、昔からあるか？

- わざと古い言語としてPerl5を取り上げますが
- 実は、Perl5には、「仮想機械」という概念がない
  - Perl –MO=Concise file.plとやってみる


schuko@c4f7:~
7a <@> leave[1 ref] vKP/REFC ->(end)
1 <0> enter ->2
2 <;> nextstate(main 1 pr.pl:3) v ->3
5 <2> sassign vKS/2 ->6
3 <$> const[IV 512] s ->4
- <1> ex-rv2sv sKRM*/1 ->5
4 <#> gvsv[*BUFSIZE] s ->5
6 <;> nextstate(main 1 pr.pl:4) v ->7
9 <2> sassign vKS/2 ->a
7 <$> const[PV ""] s ->8
- <1> ex-rv2sv sKRM*/1 ->9
8 <#> gvsv[*buffer] s ->9
a <;> nextstate(main 1 pr.pl:8) v ->b
b <0> padhv[%objarray:1,76] vM/LVINTRO ->c
c <;> nextstate(main 4 pr.pl:10) v ->d
- <1> null vKP/1 ->o
h <|> and(other->i) vK/1 ->o
g <2> eq sK/2 ->h
e <1> rv2av[t5] sK/1 ->f
d <#> gv[*ARGV] s ->e
f <$> const[IV 0] s ->g
n <@> leave vKP ->o
i <0> enter v ->j
:

Perl5ではコード≒パースツリー

- 実行用に少し最適化したパースツリーのトラバースを行うことで実行する
  - 伝統的な「Interpreter」方式
  - 実行速度の点でアドバンテージがない
    - でも、Perlは速いよ、と。
- この手の古い言語は今後基本的に触れません
  - 「tree traversal」方式は、素朴ながらわかりやすい方法（一回は通るべき道かもしれない）

実行環境については

- 後日、時間を取って詳しくやります

次回の予定

- 岩下英俊氏 (JIS Fortran 委員会委員長)の特別講義
  - 富士通HPCのど真ん中にいた方です
  - 富士通HPCと言えば…
  - HPCにおけるFortranの戦略や、Fortran標準化についてのお話をしていただけるでしょう