# 研究・教育用途の関数型プログラミング言語処理系を構築するためのフレームワークの実現に向けて

高野 保真[1,a)]　千代 英一郎[1,b)]

**概要**：本研究は，研究や教育用途の関数型プログラミング言語処理系を構築するためのフレームワークを提案する．提案するフレームワークは，構文解析・最適化などのプログラミング言語処理系に必要な要素を部品という形で用意し，それらを組み合せて，関数型プログラミング言語処理系を構築するためのものである．コンパイラ・インフラストラクチャのように，中間表現を中心として処理を部品化することで，手軽に言語処理系を構築できるものを目指す．フレームワークにより構築された言語処理系は，ウェブブラウザ上で実行可能であるように実現するため，ウェブ技術を使い内部のオブジェクトの視覚化が可能である．なお，現時点では，まだフレームワークの設計段階である．

**キーワード**：関数型プログラミング言語，プログラム可視化

## 1. はじめに

関数プログラミング [3] は，処理を関数という単位に分けて定義し，関数を組み合わせることで，より高い抽象度でプログラムを記述するプログラミングスタイルである．宣言的な記述により，計算機が何をするかという手続きを記述するのではなく，プログラムで解く問題の性質を記述するという特徴がある．関数プログラミング自体は古くからある手法であるが，高い抽象度で問題を扱えるという特徴から，高品質なプログラムを作成するための重要な概念として再び注目が集まっている．

記述性の利点の一方で，手続き的に書かれたプログラムと比較すると，プログラムのふるまいを把握することが難しい．これは高い抽象度で記述されたプログラムと，実際にコンピュータで処理される手順との対応を取る必要があるためである．多くの場合は，そこまでふるまいの把握を重視する必要がないことが関数プログラミングでの高度な抽象化の利点であるが，研究分野や教育分野においては，実際の処理手順が分かっていることが望ましい．

研究分野においては，新たな構文や最適化手法などを提案する際に，プログラムに対してどのような影響を与えるかを理解する必要がある．まず，最終的な結果として，プログラムがどのように処理されるかを把握しなくてはならない．それに加えて，プログラムのコンパイル過程にどのような変化をもたらすかについても興味があることが多い．これまでは，そのようなふるまいを観察するために，プロトタイプという形で言語処理系を実装したり，既存の処理系を改造するといったことを，個々の研究者がしてきてきたが，これは手間のかかる作業であった．

[1] 成蹊大学
[a)] yasunao-takano@st.seikei.ac.jp
[b)] chishiro@st.seikei.ac.jp

教育分野においては，構文や概念などを理解する際に，プログラムと処理手順を対比させて処理を追うことがプログラムの理解に役立つ．特に，初学者にとっては，処理系内のオブジェクトを視覚的に表示することや，1 ステップごとに実行結果を観察することがプログラムの理解に有効である．

本研究は，プログラムのふるまいの観察を目的とした言語処理系を構築するためのフレームワークを提案する．提案するフレームワークでは，構文解析・最適化・実行方式などの言語処理系に必要な要素を「部品」という形で個別に用意し，それらを組み合せて，関数型プログラミング言語処理系を構築することができる．構築された言語処理系は，ウェブブラウザ上で実行可能であるように実現し，既存のウェブ技術を用いて処理系内部のオブジェクトの視覚化が可能であるという特徴を持つ．ふるまいの観察が目的であるので，実行効率や処理速度を焦点とするような言語処理系は対象としない．

なお，現在は、実現に向けた設計を進めている段階である．

## 2. 提案フレームワーク

### 2.1 概要

本研究では，研究や教育用途の言語処理系を構築するためのフレームワークを実現する．前節で述べたように，フレームワークの目的は，関数プログラムのふるまいを観察できる言語処理系の手軽な構築であるため，以下のような特徴を持つ．

- 言語処理系に必要な要素を部品として組み合せることができる．
- 構築された言語処理系は，ウェブブラウザ上で実行可能であり，ウェブ技術を使った，実行過程の視覚化が可能である．

一つ目の項目について，用途に適した言語処理系を構築できるように，本フレームワークはコンパイラ・インフラストラクチャのように，言語の構文に非依存の一般化された内部言語（SL と呼ぶ）を持ち，その SL を対象とした最適化を行い，実行できるようにする．SL については，2.3 節で詳しく述べるが，非常に簡素な関数型プログラミング言語である．簡素な言語に一般化が可能であるのは，本フレームワークが処理速度を重視した言語処理系を対象から排除しているためである．

また，二つ目の項目について，本フレームワークを用いて構築される言語処理系は，JavaScript で記述されたものとなる．そのため，ウェブブラウザ上のテキストエリアにプログラムを記述し，その実行過程を 1 ステップごとに観察するといったプログラミング環境を含めた言語処理系を念頭に設計している．

以降，本フレームワークにより構築された言語処理系に与える言語を「入力言語」と呼ぶこととする．

### 2.2 構成

全体の構成を図 1 に示す．

まず，言語処理系の構築に際して，以下の部品を用意する．

- 構文解析・チェック
- 最適化の定義
- 評価戦略の定義
- 可視化の方法
- デバッグ

図中の四角で示したものがそれぞれの部品で，いくつかある実装の中から，灰色の背景のものを選択し，言語処理系を構築した例を示している．それぞれの部品については，以下のとおりである．

まず，入力言語の構文を解析するための部品を用意する必要がある．この部品への入力は入力言語によるテキスト表現のプログラムで，出力はそのプログラムを SL で表現した構文木である．構文解析とチェック（型チェックなど）においては，入力言語に依存する部分が多く，一般化は難しいと考えており，1 つのまとまった部品とした．特に，構文解析であれば，既存の構文解析器生成ツールが多く提案･利用されているため，それらのツールを用いることを前提として，SL への出力用機能などをフレームワークで提供する．

最適化の部品については，SL を入出力とするような，関数プログラムの最適化手法を定義することを想定する．ここで扱う最適化は，あくまで SL

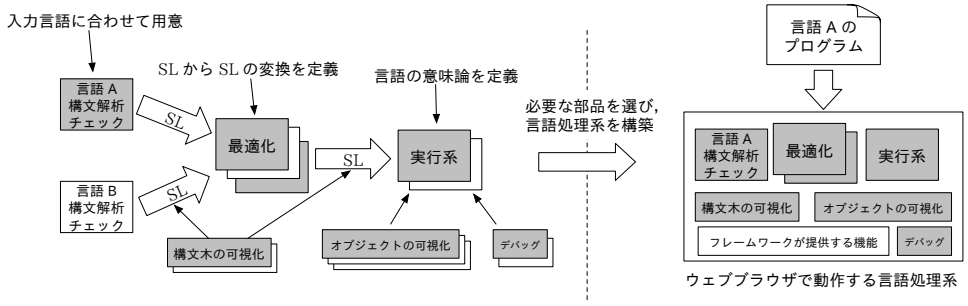


**図 1** 全体構成

上の変換で行なえる範囲を対象とし，コード生成時の最適化などコンパイラ最適化で一般的な手法を対象とするものではない．最適化手法ごとに部品を用意するため，図中のように複数の最適化部品を選んで言語処理系を構築することも想定している．

実行系の定義については，入力言語の評価戦略を SL の意味論として与える形で定義する．プログラム（式）とヒープ，スタック，環境などの計算の状態から，マッチするパターンを選択するような宣言的な定義により，部品を記述する．多くの場合はフレームワークであらかじめ用意しておく先行評価用もしくは遅延評価用の定義をそのまま使うことを想定している．

可視化の部品については，それぞれ部品間でのデータを表示できるような記述を JavaScript により与えることとする．それを容易にするため，言語処理系の内部のデータはすべて，JSON 形式 [1] で保持することとする．JSON 形式であれば，ブラウザ上で可視化しやすく，既存のウェブ技術やライブラリで扱い易い．たとえば，構文解析と最適化の間では，SL の構文木を JSON 形式で表現したデータをやりとりし，実行系内のでも保持するメモリ内のオブジェクトも JSON 形式で保持する．

デバッグのための機能も部品として追加できる．具体的には，トレースの方法などを他の部品と関連して付加できることとする．

なお，これらの部品の分け方についても，研究・教育用途を考えたときに十分な粒度であるか今後検討する必要がある．現状では，実行系のメモリモデルまでは部品として与えられるように考えてはいないが，ガーベージコレクションなどメモリ管理機構については言語処理系の重要な機構であるため，部品とする必要性を検討する．

$$
\begin{aligned}
x &\in \mathit{Identifier} \\
l &\in \mathit{Literal} \\
e \in \mathit{Expr} ::= {} & x \\
\mid {} & l \\
\mid {} & e\, x_1 \ldots x_n \\
\mid {} & \backslash(x_1, \ldots, x_n) \to e \\
\mid {} & \mathsf{if}\ e_1 \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
\mid {} & \mathsf{let}\ x_1 = e_1, \ldots, x_n = e_n \\
& \mathsf{in}\ e \\
\mid {} & (e_1, \ldots, e_n) \\
\mid {} & \mathsf{case}\ e_1\ \mathsf{of} \\
& \quad x@(x_1, \ldots x_n) \to e_2
\end{aligned}
$$

**図 2** 内部言語 SL

### 2.3 内部言語 SL

内部言語 SL の定義を，図 2 に示す．この文法は，純関数型プログラミング言語 Haskell の代表的な処理系である Glasgow Haskell Compiler（GHC）[4] の内部言語である Core 言語 [5] を簡素化したものである．Core 言語と異なる点で，大きな要素は，関数適用の引数が変数に限定されているところと，代数的データ型がない点である．

SL のリテラルとして整数，浮動小数点数，文字，

文字列，論理値がある．式を構成する要素は，変数，リテラル，関数適用式，ラムダ式，if 式，let 式，組，組を読み出す case 式である．

フレームワークの内部言語として，関数型言語を一般化した言語を設計するという観点から，プログラミング言語研究の形式化に用いられる言語を先行研究として参考にしている．それらの言語の多くは，SL のように基本的な構成要素のみで定義されている．こういった言語を対象に意味論，型理論，最適化などの形式的な手法が議論されているので，本フレームワークの内部言語として採用した．SL は非常に簡素であるので，SL への言語要素を部品として拡張する方法についても検討していかなくてはならない．たとえば，代数的データ型やパターンマッチの効率化を対象とした最適化手法を備えたような言語処理系を本フレームワークで実現しようと思うと，SL を拡張する必要がある．

### 2.4 実行の可視化

2.2 節で述べたように，本フレームワークを用いて構築される言語処理系はブラウザ上で動作し，中間データ・オブジェクトの内容を JSON 形式で保持することとする．そのことにより，既存の JavaScript 向け描画環境を利用でき，また，実行に際して新たなソフトウェアを導入する必要がないという利点が挙げられる．

JSON 形式のデータを表示するプログラムを作成すれば，自由にオブジェクトを可視化できるわけではあるが，フレームワークとしても SL の構文木の可視化部品と，ヒープ・スタックの可視化部品をあらかじめ用意し，カスタマイズ可能な形で提供する．その際には，手続き型プログラミング言語の学習環境における処理系内のオブジェクトの可視化 [2] を参考にする．

また，それと合わせて，ブラウザ組み込みの開発ツール（インスペクターや JavaScript デバッガなど）と，Redux フレームワーク [*1] など既存のウェブアプリケーションで用いられているデバッグ手法を利用できるように実現する．

[*1] https://reduxframework.com/

構築される言語処理系の動作イメージを図 3 に示す．プログラムをブラウザ上で記述し（図中 Code の部分），1 ステップ毎に実行を制御でき（図中 Control の部分），コード中には現在実行中のプログラムの箇所が明示されるようにする（Code 中の矢印の箇所）．記述したプログラムの SL の構文木を最適化の適用前後で分けても表示することも可能とするが，図中のイメージでは，1 つの構文木のみ表示した例を示している（図中 AST の部分）．また，その実行時点に応じて，ヒープとスタックの状態が可視化される（図中 Heap and Stack の部分）．

## 3. 想定するユースケース

### 3.1 研究面での用途

研究用途では，ある最適化手法を実現しようとするとき，中間言語から中間言語への変換で記述できるものであれば，他で用意された部品と組み合わせれば，その手法をとりあえず動くようなプロトタイプを作成できる．この際，中間言語の構文木を可視化すれば，動くプログラムに対して，最適化によりどのように構文木が替わるかが分かる．また，実行時のオブジェクトのふるまいも分かれば，その手法を理解するのに有効である．ただし，実行時間の削減を対象とするような最適化の場合には，本フレームワークでは正しく見積ることができないため，ステップ数などのおおまかな情報しか得られないので注意が必要である．

また，別の例としては，型チェックなどのコンパイラのフロントエンドだけが対象となる新たな手法を実験的に導入する場合が挙げられる．そのような場合でも，本フレームワークを活用できるが，SL への構文の追加などが必要な場合が考えられるため，2.3 節の最後に述べた SL を拡張自体を部品として提供する方法について今後検討していかなければならない．

### 3.2 教育面での用途

プログラミング学習環境を構築する際に重要となるのは，プログラムの構文と，プログラムの状態の可視化である．まず，プログラムの構文につい

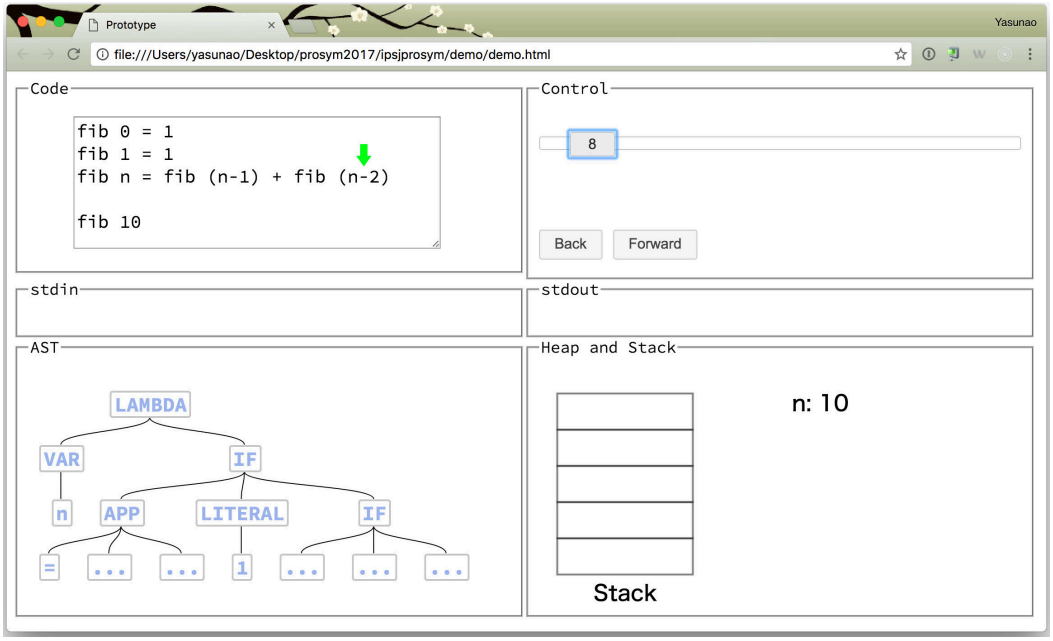


**図 3** プログラミング環境の動作イメージ

ては，本フレームワークを用いると，入力言語を中間言語にコンパイルする部分に絞って実装すれば，動作する言語処理系が作成できる．また，中間言語の構文木，メモリ上のオブジェクトなどの情報を JSON 形式で保持するため，コンパイラや実行系の細かい仕様を知ることなく，標準的な処理でプログラムの実行状態の可視化を行なえる．

また，プログラミング学習の発展的な課題として，プログラミング言語処理系を作成する情報実験等で本フレームワークを利用できる．そのような実験では，言語処理系中の一部の部品を実装し，用意した部品と組み合わせて言語処理系を構築できることができるようなものを目指す．

## 4. 関連研究

コンパイラ・インフラストラクチャとしては，COINS [7] が知られている．COINS は，コンパイラの研究・開発・教育を容易にすることを目的とした共通インフラストラクチャである．2 段階の中間表現を持つことで，コンパイラの構造や最適化機構を一般化して扱うことに成功しており，各種の手続型プログラミング言語から，各種のアーキテクチャに向けたコードを生成できるコンパイラを生成できる．COINS が手続型プログラミング言語を対象としている点と，コード生成や並列化に注力している点において，本研究と異なっている．本研究では，コンパイラの処理過程の可視化を含めたフレームワークとして，コンパイラ・インフラストラクチャのようなものを目指している．

教育向けの関数型プログラミング言語の簡潔な実装としては，MinCaml コンパイラ [6] がある．MinCaml は，Objective Caml のサブセットである関数型プログラミング言語で，そのコンパイラは教育目的としてコンパクトな実装で，関数型プログラミング言語として十分な機能を有している．教育目的である点からも，本研究のフレームワークにより構築する言語処理系が目指しているところと同じであるといえる．

プログラム実行時のオブジェクトの可視化という面からは，Online Python Tutor [2] が挙げられる．元々はプログラミング言語 Python を学習するための環境として開発されたが，現在では Java，

JavaScript，C，C++ など多くの言語への対応が進められている [*2]．Online Python Tutor の実行過程はブラウザ上で記述したプログラムを，一旦サーバー側で実行し，その結果から 1 ステップ毎にプログラムの実行を再現する．本研究は，Online Python Tutor の関数型プログラミング言語版を構築するフレームワークであるという見方もできる．ただし，本フレームワークにより構築された言語処理系においては，JavaScript でプログラムを実行するため，サーバー側での実行を伴なわず，導入のしやすさを期待できる．また，処理系内のオブジェクトの可視化方法を構築時に与えることができる設計であるため，用途にあった可視化が可能となる．

## 5. まとめ

本研究は，研究や教育用途の関数型プログラミング言語処理系を構築するためのフレームワークの実現に向けて取り組んでいる．提案フレームワークは，コンパイラ・インフラストラクチャのように，手軽に関数型プログラミング言語処理系を構築するためのもので，フレームワークにより構築された言語処理系は，ウェブブラウザ上で実行可能であるように実現する．

現状は，フレームワークの設計を進めるために，フレームワークの出力結果となる言語処理系のプロトタイプを JavaScript により開発している．このプロトタイプは，プログラミング言語 Scheme のサブセットである言語を入力言語とし，SL にコンパイルし，SL の実行系と可視化の機能を備えている．この実装を基に，それぞれのフェーズを部品として与える際の記述方式などを検討し，フレームワークの設計を進める予定である．

### 参考文献


[1] ECMA: The JSON Data Interchange Format, (online), available from ⟨`http://www.ecma-international.org/publications/standards/Ecma-404.htm`⟩ (accessed 2016/11).

[2] Guo, P. J.: Online Python Tutor: Embeddable Web-based Program Visualization for CS Education, *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, pp. 579–584 (2013).

[3] Hughes, J.: Why Functional Programming Matters, *The Computer Journal*, Vol. 32, No. 2, pp. 98–107 (1989).

[4] Jones, S. P., Hall, C., Hammond, K., Partain, W. and Wadler, P.: The Glasgow Haskell Compiler: a Technical Overview, *Proc. Joint Framework for Information Technology (JFIT) Technical Conference*, ACM, pp. 249–257 (1993).

[5] Tolmach, A.: An External Representation for the GHC Core Language, (online), available from ⟨`http://www.haskell.org/ghc/docs/papers/core-6.10.ps.gz`⟩ (accessed 2016/11).

[6] 住井英二郎：MinCaml コンパイラ (ソフトウェア論文)，コンピュータソフトウェア，Vol. 25, No. 2, pp. 28–38 (2008).

[7] 中田育男，　渡辺坦，佐々政孝，森公一郎，阿部正佳：COINS コンパイラ・インフラストラクチャの開発，コンピュータソフトウェア，Vol. 25, No. 2, pp. 2–18 (2008).


[*2] http://pythontutor.com/