

</>

2023年度未踏IT人材発掘・育成事業
竹迫PM担当プロジェクト

2023/11/26 八合目会議発表

Pythonにトランスパイル可能な 静的型付けプログラミング言語 の開発

芝山駿介

目次

- はじめに
- Pythonの欠点
- 解決策: 新言語の開発
- 新言語のコンセプト
- これまでの進捗
- 直近の進捗
- 今後の予定

- 質疑応答

自己紹介

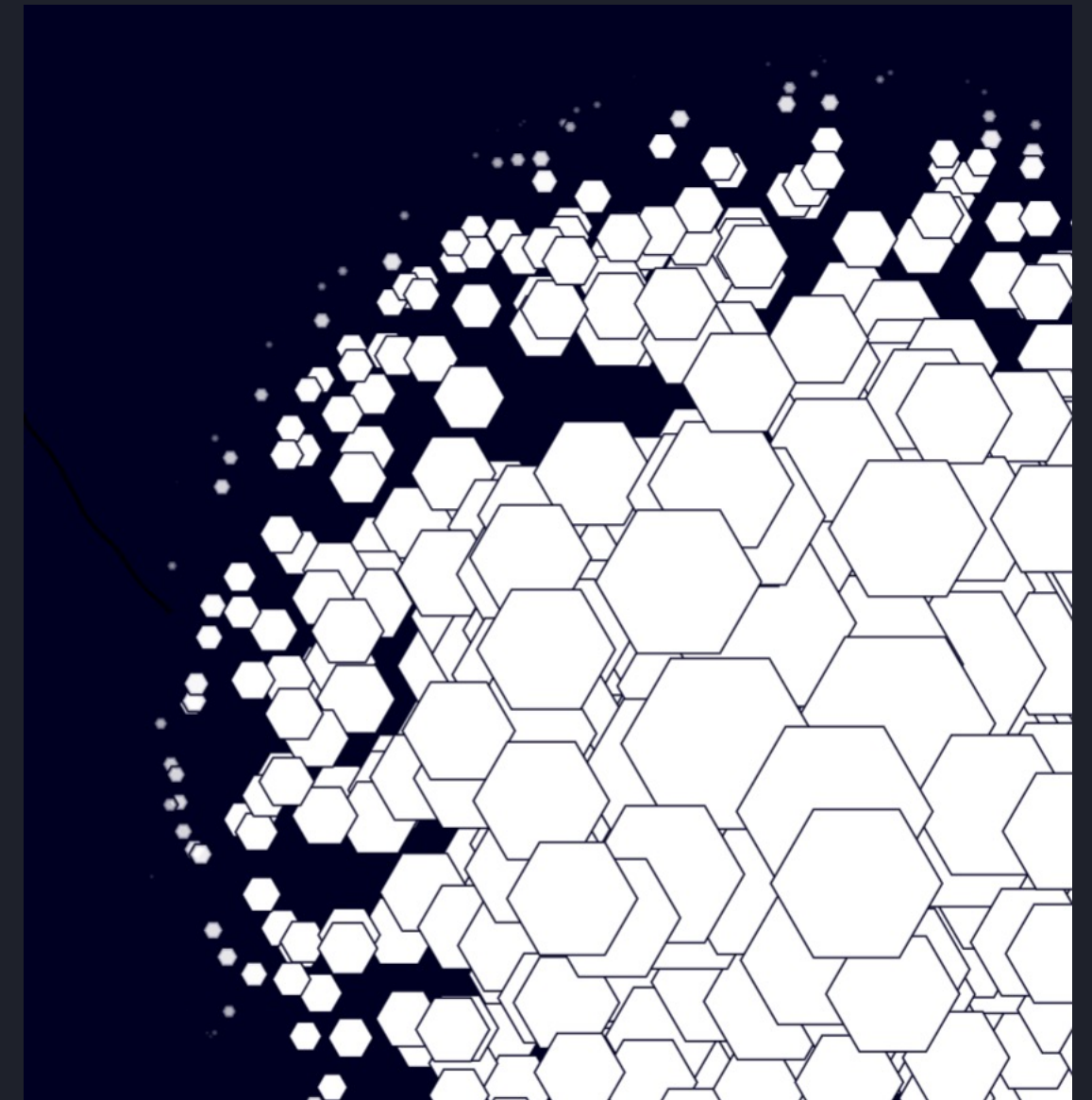
名前: 芝山駿介

所属: 早稲田大学先進理工学部物理学科4年

興味: 物理学、プログラミング言語

GitHub: [mtshiba](#)

Twitter(X): [@s_sbym](#)



近年注目される言語**Python** の大きな**3**つの欠点

動的型付け

歪な仕様

環境構築難度



Pythonの欠点

動的型付け

動的型付けは実行時にコードの正当性を検証するため、バグの発見が遅れる
=コード品質に影響する可能性がある [1] [2]

また、静的型付け言語に比べて実行効率も悪い傾向にある

よって、実行に時間がかかる・大規模なソフトウェアにおいては静的型付け言語の方が一般に適していると言える

学術計算・機械学習技術はこれに当てはまるが、**Python**が第一言語となってしまう

[1] http://www.washi.cs.waseda.ac.jp/wp-content/uploads/2016/11/HASE_2017_paper_25.pdf

[2] <https://danluu.com/empirical-pl/>

Pythonの欠点

動的型付け

このように話すと、小規模なソフトウェア開発・プロトタイピングならまだ動的型付けの方が有利という意見が挙がりがちだが、批判を恐れずに言えば勘違いである

このような人は恐らくPythonなどの動的性 — JupyterやREPLが使える — を動的型付け固有と誤認している

漸進的型付けの理論研究・実装も進歩しており、型付けできないプログラムの集合は年々縮小している

Language Serverの登場も静的型付けの優位性を引き上げた
→ Language Serverは実行前に多くの情報が得られる言語に有利な仕組み

Pythonの欠点

歪な仕様

Pythonは初心者にもわかりやすい平易な文法と謳われているが、それは極めて表面的な部分だけである

→ **Pythonの設計は場当たりのである**

オブジェクトの参照や変数のスコープに非直感的な部分[1]があり、バグの温床となっている

また、Pythonは後付けのオブジェクト指向言語であり近年注目されている関数型プログラミングにはあまり向いていない(詳しくは後述)

[1] <https://github.com/satwikkansal/wtfpython>

```
Python 3.11.0 (main, Jul 12 2023, 00:27:36) [Clang 14
Type "help", "copyright", "credits" or "license" for
>>> a = "wtf"
>>> b = "wtf"
>>> a is b
True
>>> a = "wtf!"
>>> b = "wtf!"
>>> a is b
False
>>> a, b = "wtf!", "wtf!"
>>> a is b
True
```

```
Python 3.11.0 (main, Jul 12
Type "help", "copyright", "c
>>> class C: pass
...
>>> C() == C()
False
>>> c = C(); c == c
True
```

Pythonの欠点

歪な仕様

Pythonの型設計は不健全である
(e.g., Liskovの置換原則に違反している)

また最近のPythonは型指定(もどき)ができるようになったが、後付けゆえの見苦しさがある

```
Python 3.11.0 (main, Jul 12 2023, 00:27:36)  
Type "help", "copyright", "credits" or "lic  
>>> from collections.abc import Hashable  
>>> isinstance(list, object)  
True  
>>> isinstance(object, Hashable)  
True  
>>> isinstance(list, Hashable)  
False
```

```
from typing import TypeVar, Generic  
  
T = TypeVar("T")  
  
class LoggedVar(Generic[T]):  
    ...
```


Pythonの欠点

開発環境の構築が難しい

Pythonは公式の提供する開発ツールが比較的貧弱であるため、サードパーティの開発ツールが乱立している

仮想環境: `venv`, `pyenv`, `pyenv-virtualenv`, `pipenv`, `Anaconda`

フォーマッタ: `black`, `autopep8`, `yapf`, `autoflake`

パッケージマネージャ: `pip`, `poetry`, `pipenv`, `Anaconda`

Lint: `flake8`, `pylint`, `Prospector`, `ruff`

→ 有名所だけでも**320**パターン以上の中から選定する必要がある
他人のコードを動かす場合...

</>

では、解決策はあるか？

</>

— Pythonの欠点を克服した新言語を作る

新言語のコンセプト

PythonのAPIを直接呼び出せる(トランスパイルされる)

先述のような欠点があるにもかかわらず
Pythonが人気なのは、圧倒的な量の
コード資産があるという点が大きい

これらのコード資産を流用できる言語があれば
そちらに移行できる

このような既存言語の資産を流用できる言語は
ScalaやTypeScriptなどがある




新言語のコンセプト

高度な静的型システムを持つ

静的型付けは実行効率のみのためにあらず
型システムはコードの堅牢性を高めてくれる

新言語は依存型と呼ばれる高度な型を持つ
これを用いると例えば配列の境界チェックなどを
コンパイル時にも検査できる

```
tests > should_err >  dependent.er
1  dic = {"a": 1, "b": 2}
2  print!(dic["c"]) # ERR
3
4  arr = [1, 2, 3]
5  print!(arr[5]) # ERR
6  |
```

新言語のコンセプト

型推論機能を持つ

型推論とは変数や関数の型を指定せずともコンパイラが自動で推論してくれる機能

動的型付けのようなシンプルな記述でありながら検査はしっかりと静的に行われている

```
test.er
1  add (x: Int, y: Int): Int = x + y
2  print! add x:= 1, y:= 2
3  print! add x:= 1, y:= "a" # ERR
4  print! add x:= "a", y:= 1 # ERR
5
```

新言語のコンセプト

関数型 + オブジェクト指向

オブジェクト指向(**Object-oriented**)はPythonやJavaなど多くの言語が採用するパラダイム

しかし近年は、より数学的で形式的な取り扱いが容易な関数型(**Functional**)プログラミングが注目されている

両者は組み合わせることも可能であり、そのようなアプローチを取る言語もある(e.g., Scala)

提案する言語もこのアプローチを採用する



The screenshot shows the Scala Programming Language website. At the top, the Scala logo is on the left and a hamburger menu icon is on the right. Below the logo, the title "The Scala Programming Language" is displayed. A paragraph describes Scala as a language combining object-oriented and functional programming. Below this, three buttons are shown: "SCALA 3.2.2", "SCALA 2.13.10", and "ALL RELEASES". The main content area features a code editor titled "Encode and decode custom data types to JSON". The code defines a case class Pet, creates an instance, and demonstrates JSON serialization and deserialization. A "Run in playground" button is in the top right of the code editor. The code is as follows:

```
case class Pet(
  name: String,
  kind: String
) derives Codec // enable coding Pet to and from text

< val coco = Pet(name = "Coco", kind = "Cat") >

val message = writeJson(coco)
// ^^^^^^^ contains the text: {"name":"Coco","kind":"Cat"}

readJson[Pet](message) // convert message back to a Pet!
```

At the bottom of the code editor, it says "The pluggable derivation system gives custom types new capabilities." and "2/4".

Source: <https://scala-lang.org>

新言語のコンセプト

開発ツールの統合

仮想環境マネージャ、フォーマッタ、
パッケージマネージャ、**linter**などを
すべてコマンドひとつに統合

このようなアプローチはGo言語などが採用
環境構築が容易になり、
高い再現性も保証される

```
~ via ▲ v3.23.2 via 🐧 v3.11.0  
at 21:54:34 > go  
Go is a tool for managing Go source code.
```

Usage:

```
go <command> [arguments]
```

The commands are:

bug	start a bug report
build	compile packages and dependencies
clean	remove object files and cached files
doc	show documentation for package or symbol
env	print Go environment information
fix	update packages to use new APIs
fmt	gofmt (reformat) package sources
generate	generate Go files by processing source
get	add dependencies to current module and install them
install	compile and install packages and dependencies
list	list packages or modules
mod	module maintenance
work	workspace maintenance
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	report likely mistakes in packages

新言語のコンセプト

+ ネイティブコードバックエンド

新言語の既定のバックエンドは**Python**インタプリタだが、せっかく静的解析したのだからバイナリ生成もできると嬉しい

プログラムの高速化も期待できるし、シングルバイナリにまとめられるのでプログラムの配布が容易になる

CPythonを使わず計算できるところはネイティブに実行し、どうしても必要な部分はインタプリタと通信する形式

……という言語「Erg」を開発しています

これまでの進捗

✓言語機能の実装

未踏期間前から基本的機能は実装済み

未踏期間中はツール開発に必要な機能の実装にフォーカス

実装した主要な機能は以下の通り:

- スライスの実装
- `bin/oct/hex`リテラル
- ネストしたパターンマッチ
- `refinement class`
- `unsound`モジュール
- 可変長(キーワード)引数の実装

これまでの進捗

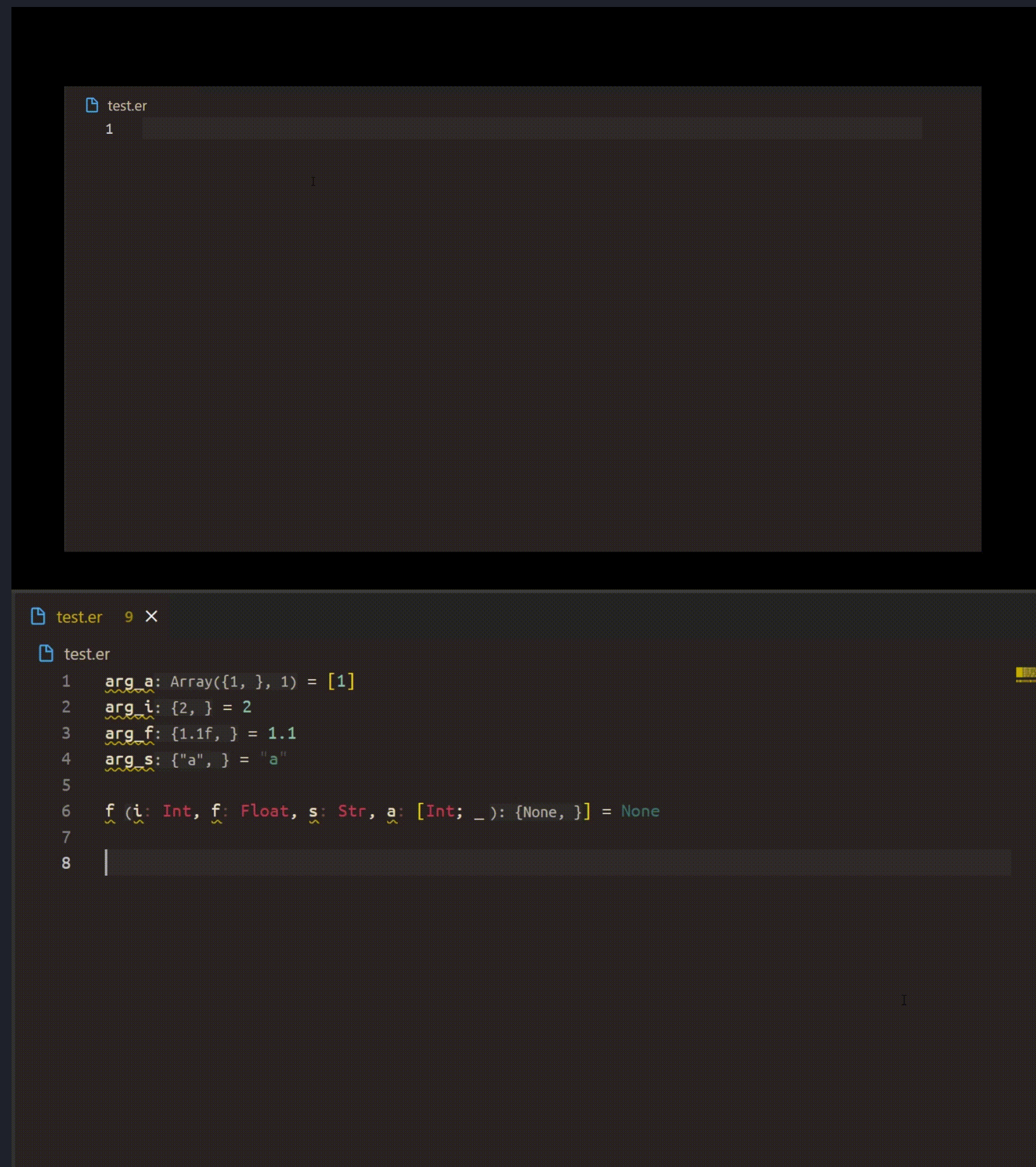
✓Language Serverの開発

RustのLanguage Serverである

rust-analyzerを参考に実装

エラーのハイライト、補完、**rename**
など基本的機能は未踏前から実装済み

未踏期間中は発展的な機能や
並列化、補完精度の向上などを行なった



これまでの進捗

✓ パッケージマネージャの実装

Erg自身を用いて実装

現在実装されている機能の例：

- **init:** パッケージを初期化(設定ファイルの生成等)
- **build:** 依存関係を解決・パッケージをビルド
- **install:** アプリケーションパッケージを擬似実行ファイルにしてbinディレクトリに配置
- **publish:** 後述するレジストリにパッケージを登録する

これまでの進捗

✓ ネイティブコードバックエンドの実装

ネイティブコードバックエンドは
Rustコードをターゲットとする方式で実装
バイナリ生成の仔細な実装を省略

現在は基本的な計算機能とプリミティブ
オブジェクトを実装した段階

現在は**Rust crate**を**import**する機能を実装中

$$Erg\text{スクリプト} \xrightarrow{Erg\text{コンパイラ}} Rust\text{コード} \xrightarrow{Rust\text{コンパイラ}} \text{バイナリ}$$

```
smallvec = rsimport "smallvec"  
  
v = smallvec.SmallVec!.from([1, 2, 3])  
print! v.len()  
assert v.len() == 3
```

直近(11月)の進捗

✓ crate-inspectorの開発

Rustコードバックエンドでは

Rust crateをimportできるように

したら嬉しい

→ Rust crateからErgの型定義ファイルを
生成する機構を実装中

そのためにRust crateの公開API情報を
取得するライブラリを開発

```
use crate_inspector::CrateBuilder;

let builder = CrateBuilder::default()
    .toolchain("nightly")
    .manifest_path("Cargo.toml");
let krate = builder.build().unwrap();

for item in krate.items() {
    println!("item: {:?}", item.name);
}
for strc in krate.structs() {
    println!("struct: {}", strc.name());
    println!("#impls: {}", strc.impls().count());
}
for enm in krate.enums() {
    println!("enum: {}", enm.name());
    println!("variants: {:?}", enm.variants().collect::<Vec<_>>());
    println!("#methods: {}", enm.impls().fold(0, |acc, i| acc + i.functions().count()));
}
for sub in krate.sub_modules() {
    println!("submodule: {}", sub.name());
}
if let Some(foo) = krate.get_item("foo") {
    println!("id: {:?}", foo.id);
}
```

(公開したOSS)

Erg本体もOSSで公開されているが、
開発過程で書いたコードのうち
汎用的に利用できる部分は別のOSSとして公開

molc: Language Serverのテスト用ダミークライアント

ruast: Rustコード生成用のRust AST

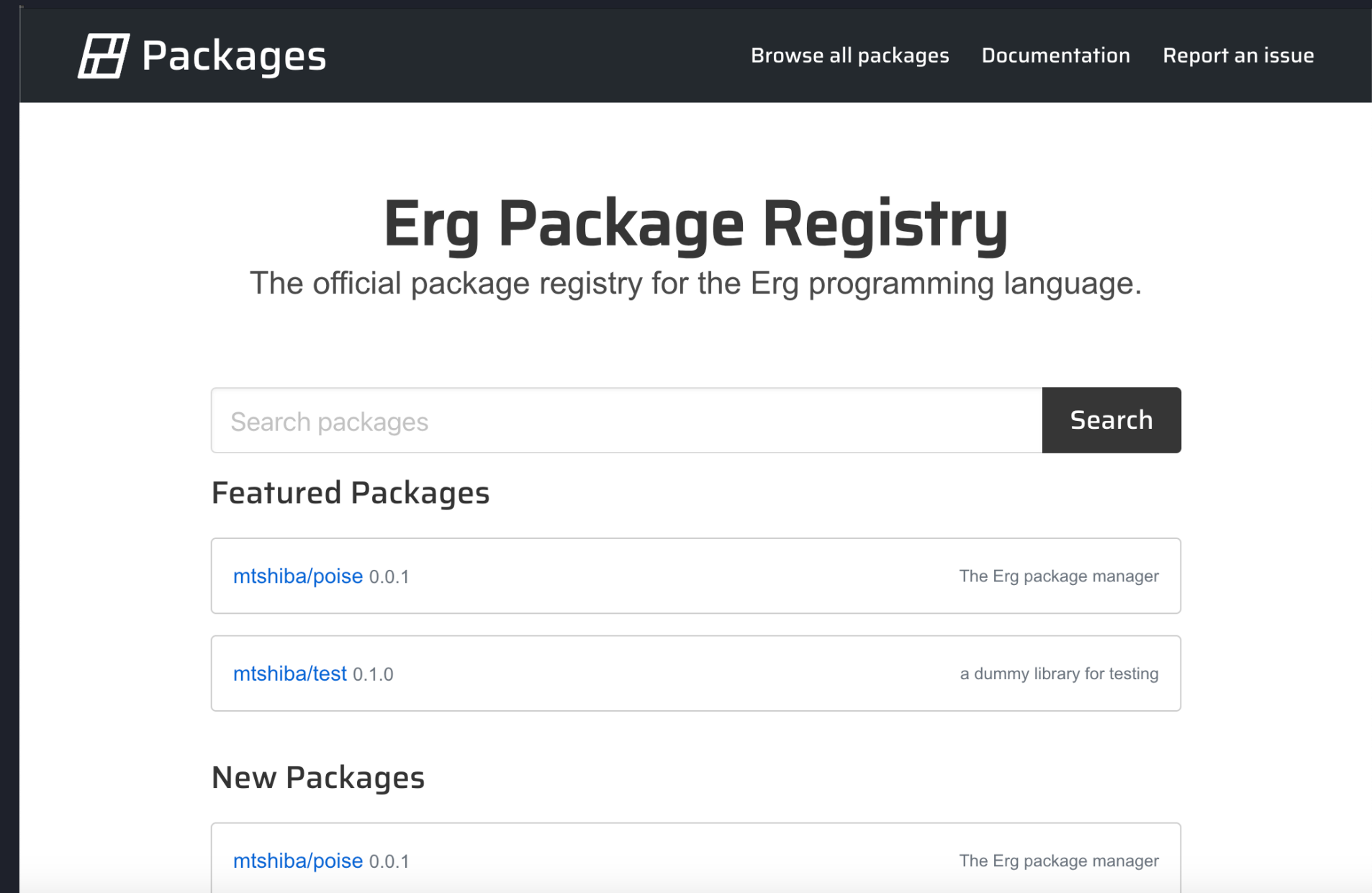
crate-inspector: Rust crateの公開API情報を取得する

直近の進捗

✓ レジストリサイトの作成

追加されたパッケージの詳細を閲覧
できる サイト を作成

レジストリリポジトリへの **push** をトリ
ガーにしてコンテンツが更新される



直近の進捗

✓ コンパイラをPythonライブラリとして公開

ErgコンパイラはRustで書かれている
ので本来Erg ASTやコンパイラを実行時に
触ることはできない

しかしモジュールを動的にロードしたい
ケースが出てきたので、コンパイラを
([pyo3](#)を使って)Pythonライブラリ化した

```
erg_compiler: PyModule("erg_compiler") = pyimport "erg_compiler"
erg_parser: PyModule("(...)") = pyimport "erg_compiler/erg_parser"
erg_ast: PyModule("(...)") = pyimport "erg_compiler/erg_parser/ast"

mod: ast.Module = erg_parser.parse code:= ".i = 1"
ast: ast.AST = erg_ast.AST.new name:= "test", mod
test: Module(_: Str) = erg_compiler.exec_ast ast
i: Obj = test.__dict__.get(key:= "i")
assert i in test:= Int
assert test:= i == 1
```

Demo dayまでの予定

パッケージマネージャのバイナリ化

ネイティブコードバックエンドの目標は
今の所**CPython**で実行しているパッケージマネージャを
バイナリへコンパイルできるかというところで設定

パッケージマネージャはレジストリと通信など
しているので、その辺りをなんとかする必要がある

Demo dayでやりたいこと

ライブコーディング

コンピュータにErgがインストールされていない状態から

- ツール一式のインストール
- 何らかのパッケージの作成
- コーディング
- 実行
- パッケージ登録

までスムーズにやれるように持っていく

未踏期間後の展望

布教

- PythonやRustコミュニティを中心にカンファレンスに参加するなどしてアピールしていく
- ドキュメントの拡充
 - 言語仕様
 - チュートリアル
 - コントリビュータ向け内部資料

</>

提案者
芝山駿介

発表日
2023 / 11 / 26