簡単に ✔ ✔ ちょっと速い インタプリタをつくる方法

Koichi Sasada
STORES, Inc.

Kernel/VM探検隊@東京 No18

この発表について

- 簡単に
→ C で AST Traversal Interpreter を作る程度の労力で
- ちょっと速い
→ GCC/-O0 程度の速度で動く
- インタプリタを作る方法
→独自言語向けインタプリタ（+コンパイラ）を作る方法

をご紹介する発表です

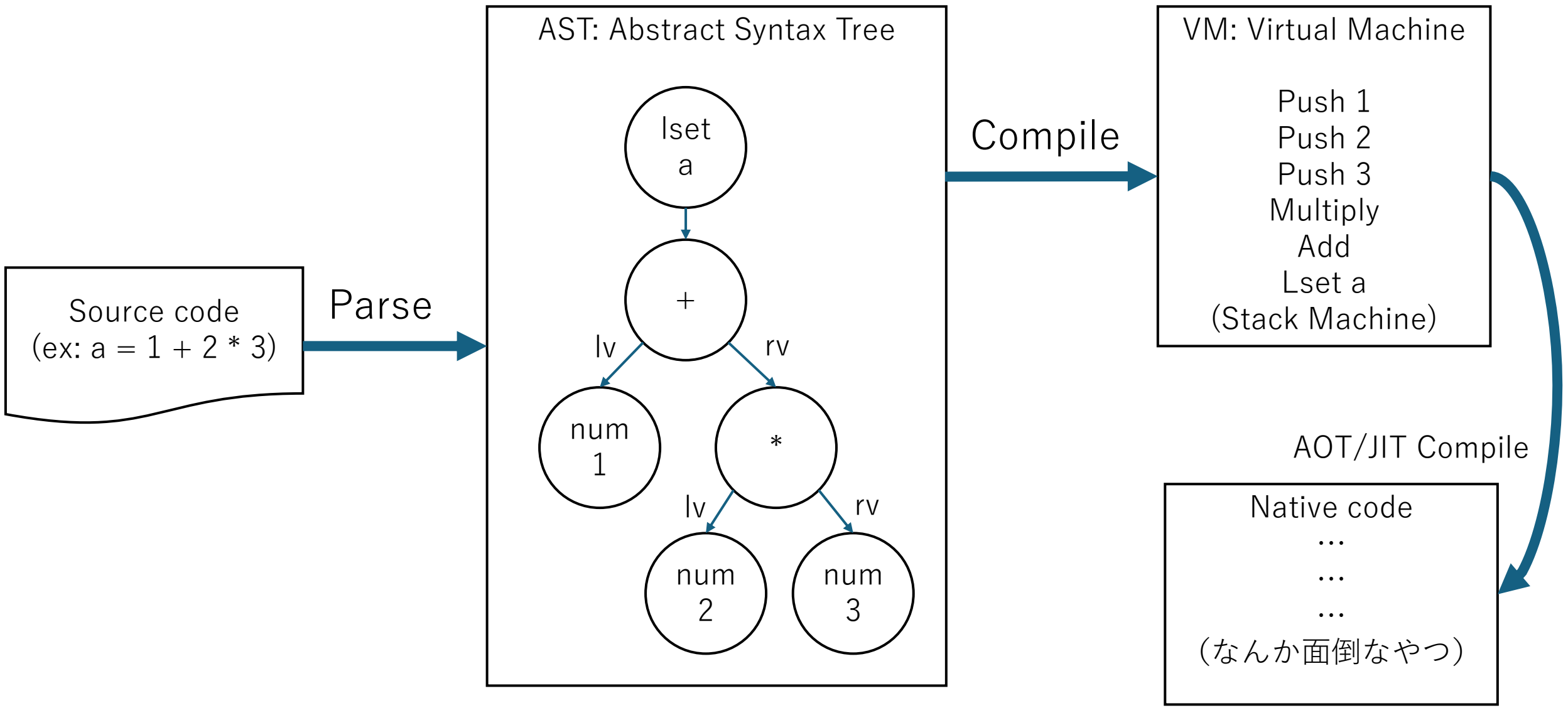
自己紹介
Koichi Sasada（笹田耕一）

- Ruby interpreter developer 2004~, now employed by STORES, Inc. (2023~)
  - YARV (Ruby 1.9~)
  - Generational/Incremental GC (Ruby 2.1~)
  - Ractor (Ruby 3.0~)
  - debug.gem (Ruby 3.1~)
  - M:N Thread scheduler (Ruby 3.3~)
  - …
- Ruby Association Director (2012~)
- 農工大並木研（OSの研究室）出身、その後大学教員など
  - 卒修論は Thread library for multi-threaded architecture
  - 大学在職中、VMM の研究とかも

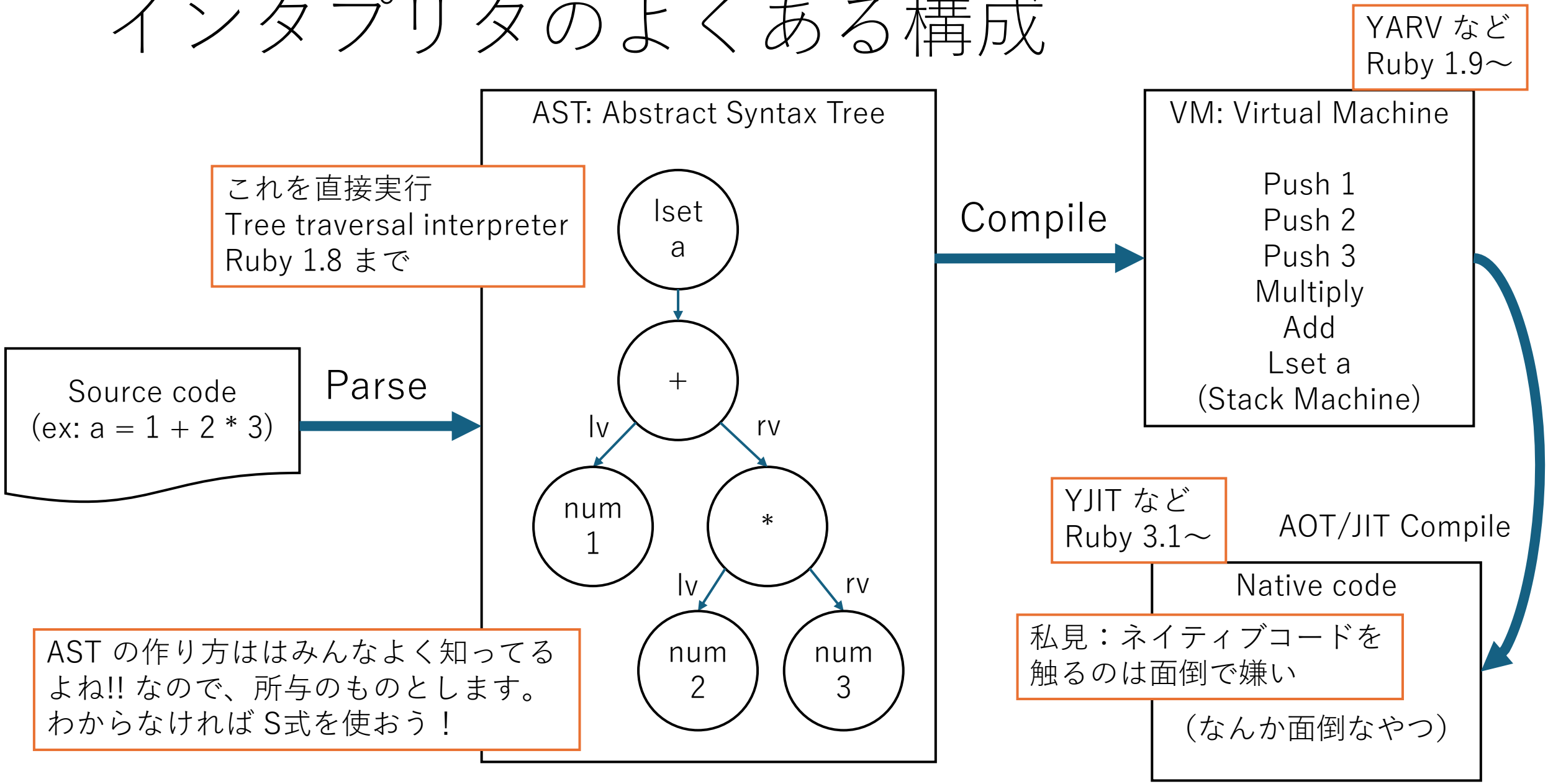
簡単に速いインタプリタをつくる

- そもそもインタプリタをいじるのは簡単では？
  - バイナリ出てこないし…
  - gdb とか問題なく使えるし…
- 速度を出そうとすると難しくなる
  - ネイティブコードに変換したりする

インタプリタのよくある構成



インタプリタのよくある構成



Tree Traversal Interpreter in C by recursive eval function

```
int eval(CTX *c, NODE *n){ // c は実行コンテキスト
  switch(n->type) {
    case NODE_NUM:
      return n->value;
    case NODE_ADD:
      return eval(n->lv) + eval(n->rv);
    case NODE_MUL:
      return eval(n->lv) + eval(n->rv);
    case NODE_LSET:
      return lset(c, n->name, eval(n->rhs));
```

Tree Traversal Interpreter in C by recursive eval function

```
int eval(CTX *c, NODE *n){ // c は実行コンテキスト
  switch(n->type) {
    case NODE_NUM:
      return n->value;
    case NODE_ADD:
      return eval(n->lv)
    case NODE_MUL:
      return eval(n->lv)
    case NODE_LSET:
      return lset(c, n->name, eval(n->rhs));
```

👍 いい点
簡単！！
一瞬でインタプリタ書ける！！
（評価器部分だけ）

👎 わるい点
遅そう！！

速くするために

- いい感じの VM を設計して使う
  - YARV みたいなケース
  - さらに JIT compiler とか作っちゃう

  → いくらかかるの…？
  　前段（ASTとか）の修正が後段（VM, JITとか）に波及して大変
- いい感じの既存の VM を使う
  - いわゆる JVM 言語とか
  - LLVM とか今はどこにでもありそう？

  → 使い方覚えるのが大変
  　特定のツールに bind されると厳しそう

さっきの C のインタプリタ書くだけで速くならんかな…？？？

eval1 (again)
Tree Traversal Interpreter w/ switch/case

```
int eval1(CTX *c, NODE *n){ // c は実行コンテキスト
  switch(n->type) {
    case NODE_NUM:
      return n->value;
    case NODE_ADD:
      return eval(n->lv) + eval(n->rv);
    case NODE_MUL:
      return eval(n->lv) + eval(n->rv);
    case NODE_LSET:
      return lset(c, n->name, eval(n->rhs));
```

太字の部分が各ノードの挙動。
ボディ部というようにします。

eval2
Tree Traversal Interpreter w/ func ptr

```
int eval2(CTX *c, NODE *n){
  return (*n->eval)(c, n);
}
```

各ノードは eval 関数へのポインタをもつ。
ちょっと見やすくなった？
速度はむしろ遅くなりそう

```
int eval_num(CTX *c, NODE *n) {
  // さっきの case の中身（ボディ部）を
  // 関数にくくりだしただけ
  return n->value;
}
```

…

eval3
Tree Traversal Interpreter w/ dispatch func

```
int eval_3(CTX *c, NODE *n){
  return (*n->dispatcher)(c, n);
}

int dispatch_num(CTX *c, NODE *n) {
  return eval_num(c, n, n->value);
}

int eval_num(CTX *c, NODE *n, int value) {
  return value;
}
```

…

eval 関数を直接じゃなくて
dispatcher 経由に。
なんかさらに無駄なことしてる？

Num ノードの中身（value）を
取り出して eval（ボディ部）に渡す

eval1〜3 のまとめ

- eval1
  - 一番素直なノード評価器
- eval2
  - eval1 の case 部分を各ノードごとに関数化（eval_*）
- eval3
  - eval2 の eval_* を、一度 dispatch_* を経由して呼び出し
  - dispatch_* は、eval_* にノードの中身を渡す
- 1 ~ 3 は、書き方が違うだけで、同じことをやっているだけ
- **実際、どれもふつうに動く（試してないけど多分 1 が一番速い）**

速くするのはどうしたの！？

二村射影 (Futamura Projection)

- 部分評価、Partial Evaluation (PE)
  - 例: f(s, n) = s ? n : -n のとき、
    f(1, n) という呼び出しは f1(n) = n に特殊化できる
  - 一般化: 処理pを既知のデータdについて特殊化すると、
    PE(p, d) → pd（dについて最適化された処理）ができる
- 二村射影（第1二村写像）（つまり、第2、第3まであるよ）
  - 処理をインタプリタ I、データをIで動かすプログラム P とすると、
    PE(I, P) → IP（P に最適化された専用のインタプリタ処理）ができる
    → ネイティブコンパイラやん？
- くわしいことは ChatGPT とかに聞いてください

C のインライン展開みたいだね

二村射影を利用するインタプリタ

- PyPy
  - Rpython (Restricted Python) で記述したインタプリタを特殊化
- Truffle/Graal
  - Truffle DSL（Java） で記述した AST Traversal Interpreter を特殊化
  - Interpreter → JVM (profile) → Native code
  - むっちゃ速いと評判（起動速度は見ないこととする）

でも、これらを使うのは大変そうだなぁ…

覚えること多そうだし…

→ **C の環境だけでできないかな？　どこにでもあるし**

ASTro
AST based Reusable Optimizer

初公開だよ！

ASTro framework

- **ノードの挙動だけ**を読ませて、インタプリタを自動生成
  - ノード型定義、**評価器**、**部分評価器**、出力器、…
- 二村射影を用いた、特殊化したコード生成までやってくれる
  - といっても、吐くのは C のコード片
  - JIT とかで使うなら、実行時に C のコンパイルが必要
    - Kyoto Common Lisp, UT Lisp, Ruby/MJIT などと同じ
    - ASTro は、これを利用しやすくするためのあたらしい工夫を入れているけど、時間も足りないのでまた今度
  - AOT compiler, Specialized interpreter なら簡単に生成可能

node.def
ユーザーに書いてもらうノードの挙動

引数とボディ部をノードごとに記述
つまり、eval3 の eval_* 相当を書いてもらう

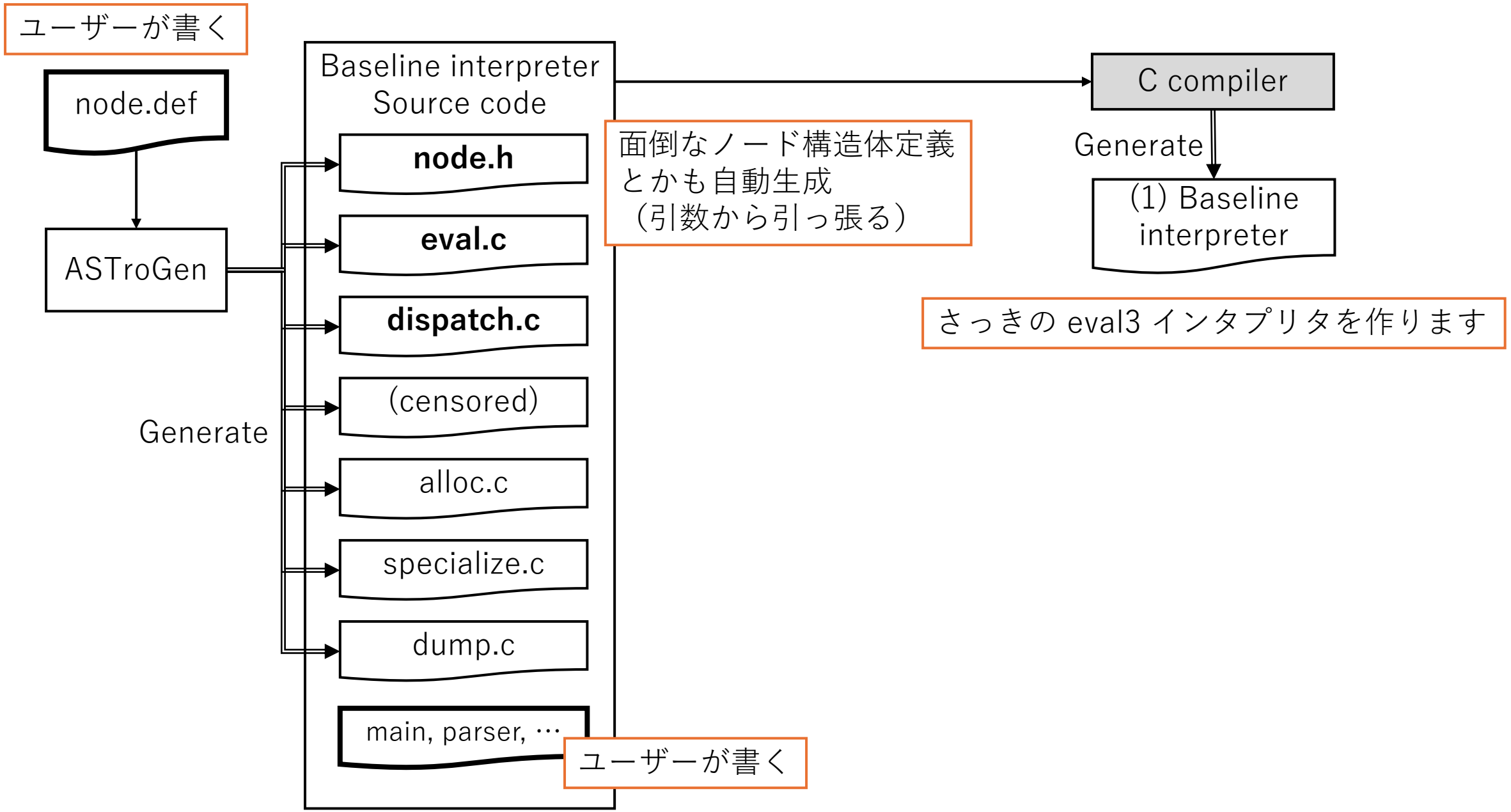
```
NODE_DEF
num(CTX *c, NODE *n, int value) {
  return value;
}

NODE_DEF
add(CTX *c, NODE *n, NODE *lv, NODE *rv) {
  return EVAL_ARG(c, lv) + EVAL_ARG(c, rv);
}
...
```

ノード numとは、value を要素としてもつノードであり、
その挙動は value を返す。

引数で渡されたノードを
評価する専用マクロ

ノード add とは、lv, rv を要素としてもつノードであり、
その挙動は lv, rv を評価し、それを足して返す。


ユーザーが書く
node.def
ASTroGen
Generate
Baseline interpreter
Source code
node.h
eval.c
dispatch.c
(censored)
alloc.c
specialize.c
dump.c
main, parser, …
ユーザーが書く
面倒なノード構造体定義
とかも自動生成
（引数から引っ張る）
C compiler
Generate
(1) Baseline
interpreter
さっきの eval3 インタプリタを作ります

Partial evaluation without touching bodies
ここからが高速化のための新規手法

例: a = 1 + 2 * 3 → lset(“a”, add(**num(1)**, mul(**num(2)**, **num(3)**)))

手法: 具体ノードを受け取り、特殊化された dispatcher **だけ**を生成

num(1) の特殊化

```
dispatch_num_1(CTX *c, NODE *n) {
   return eval_num(c, n, 1); // 1 に特殊化
}
```

eval_num() の実装は `return value` だけなので、
つまりこの関数は `return 1` だけにインライン展開される

`// 2, 3` も同様に生成

Partial evaluation without touching bodies
ここからが高速化のための新規手法

例: a = 1 + 2 * 3 → lset(“a”, add(1, **mul(2, 3)**))

手法: 具体ノードを受け取り、特殊化された dispatcher **だけ**を生成

mul(2, 3) の特殊化

```
dispatch_mul_2_3(CTX *c, NODE *n) {
  return eval_mul(c, n, n->lv,
                  dispatch_num_2,
                  n->rv,
                  dispatch_num_3);
}
```

```
#define EVAL_ARG(c, n) ¥
  (*n##_disp(c, n))

eval_mul(CTX *c, NODE *n,
  NODE *lv, DISP *lv_disp,
  NODE *rv, DISP *rv_disp) {
  return EVAL_ARG(c, lv) *
         EVAL_ARG(c, rv);
}
// 自動生成だからできる工夫
```

子ノードがいる場合、
（特殊化された）dispatcher を引数に渡す
（eval 側も引数を自動的に増やしておく）

Partial evaluation without touching bodies
ここからが高速化のための新規手法

例: a = 1 + 2 * 3 → lset(“a”, **add(1, mul(2, 3))**)

手法: 具体ノードを受け取り、特殊化された dispatcher **だけ**を生成

add(1, mul(2, 3)) の特殊化

```
dispatch_add_1_mul_2_3(CTX *c, NODE *n) {
  return eval_add(c, n, n->lv,
                        dispatch_num_1,
                        n->rv,
                        dispatch_mul_2_3);
}
```

さっき作ったやつを利用

Partial evaluation without touching bodies
ここからが高速化のための新規手法

例: a = 1 + 2 * 3 → **lset(“a”, add(1, mul(2, 3)))**

手法: 具体ノードを受け取り、特殊化された dispatcher **だけ**を生成

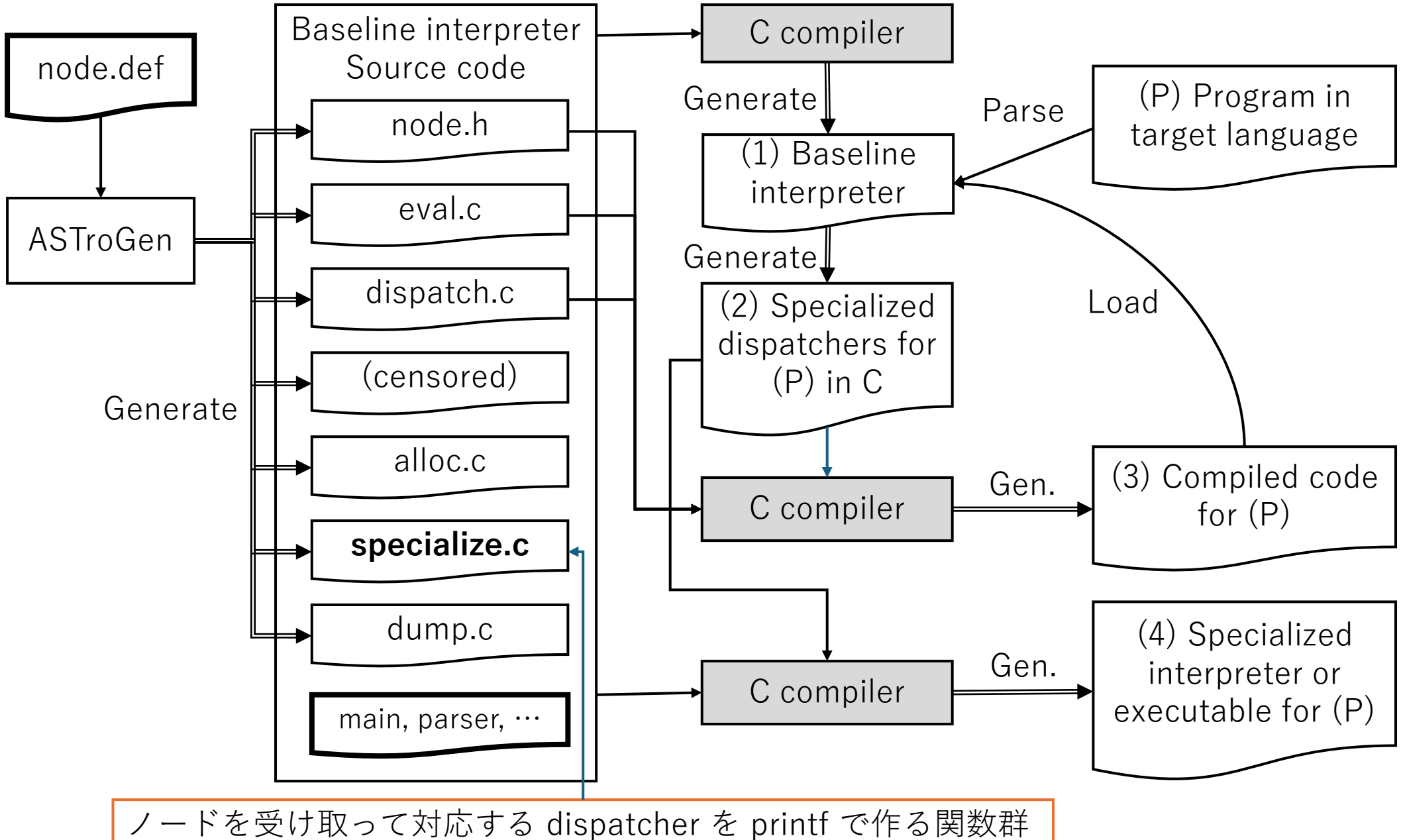
lset(“a”, add(1, mul(2, 3)) の特殊化

```
dispatch_lset_a_add_1_mul_2_3(CTX *c, NODE *n) {
  return eval_lset(c, n, “a”,
                   n->rv,
                   dispatch_add_1_mul_2_3);
}
```

さっき作ったやつを利用

特殊化によって得られた dispatch 関数

- `a = 1 + 2 * 3` のノードから、下記を生成
  - **`f1: dispatch_num_1, …_2, …_3 // 1, 2, 3 を返す関数`**
  - **`f2: dispatch_mul_2_3 // 6 を返す関数`**
  - **`f3: dispatch_add_1_mul_2_3 // 7 を返す関数`**
  - **`f4: dispatch_lset_a_add_1_mul_2_3 // 7 を変数にセットする`**
- つまり、`f4` を実行すると `a = 1 + 2 * 3` が実行される
- これを C コンパイラに食わせると、**インライン展開**でうまいことやってくれて、**f4 は 7 を変数にセットするだけの関数になる！！ やった！！　速い！！**
  （関数ポインタをべた書きしたらインライン化するという驚き）
- **この手法の素晴らしい点は、body に一切手を入れないこと（面倒な C のパースとかは一切不要）**


node.def
ASTroGen
Generate
Baseline interpreter
Source code
node.h
eval.c
dispatch.c
(censored)
alloc.c
specialize.c
dump.c
main, parser, …
C compiler
Generate
(1) Baseline interpreter
Generate
(2) Specialized dispatchers for (P) in C
C compiler
C compiler
Parse
(P) Program in target language
Load
Gen.
(3) Compiled code for (P)
Gen.
(4) Specialized interpreter or executable for (P)
ノードを受け取って対応する dispatcher を printf で作る関数群

評価

- 家にある x86_64 と RISC-V マシンで評価
- naruby という Ruby 文法だけど全然 Ruby していない言語を ASTro で作って評価
- 部分評価により gcc/-O0 に匹敵する性能を確認

Table 1. Micro benchmark results on x86_64 in seconds

	loop	fib	call	prime_count
naruby/interpret	0.786	4.870	6.760	6.170
naruby/compiled	0.001	1.093	3.435	0.444
naruby/pg	0.001	1.143	2.061	0.443
gcc/-O0	0.042	0.480	1.121	0.490
gcc/-O1	0.023	0.400	1.027	0.005
gcc/-O2	0.001	0.115	0.318	0.434

Table 2. Micro benchmark results on RISC-V in seconds

	loop	fib	call	prime_count
naruby/interpret	8.096	45.580	62.268	53.177
naruby/compiled	0.003	6.834	17.657	1.289
naruby/pg	0.003	7.053	8.566	1.289
gcc/-O0	0.544	5.481	8.385	2.476
gcc/-O1	0.061	1.132	3.151	0.017
gcc/-O2	0.002	0.604	1.575	1.280

まだ言ってないこと、わからないこと

- ASTro の真の狙い、名前の由来
- 関数/メソッド呼び出しをいい感じにする仕組み
- 実行時情報をいい感じに埋め込む仕組み
- コード爆発に対処する仕組み
- JIT コンパイラにするための仕組み
  （C コンパイラで JIT コンパイラ作ったらむっちゃ遅いことに対する対処方法）
- この手法の限界
- 本当にまともな言語で work するの？　という疑問

まとめ

- 簡単に

→ C で AST Traversal Interpreter を作る程度の労力
　具体的には node.def 書くだけで

- ちょっと速い

→ 第1二村写像を用いた部分評価器により、
　GCC/-O0 程度の速度で動く

- インタプリタを作る方法

→ ASTro framework

→独自言語向けインタプリタ（+コンパイラ）を作る処理系

おわり

- ご清聴ありがとうございました
  - discord@ko1