正しいプログラムを簡単に書くには？

プログラムの型とそのデバッグ手法

2016 年 10 月 20 日 市民講座

国立情報学研究所　アーキテクチャ科学研究系
対馬かなえ

本講座の内容

- 「プログラム」とはどんなものか
- プログラムの「正しさ」とはなにか
- 「型」とそのデバッグ手法とはどんなものか
- 型以外に正しさを高める手法の紹介

プログラムとはどんなものか？

プログラムはどんなものか？

- プログラムとはなにか？
- プログラムはどう実行されるか？

プログラムとはなにか？

・コンピュータへの指令・指示書

指令・指示書

・「1 から 10 まで足してください」

どう計算する？

- 「1 から 10 まで足してください」
  - 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
  - 10 + 9 + 8 + … + 1
  - 1 + 10 + 2 + 9 + …
  - (1 + 10) × 10 / 2　(等差数列の和)

→ いろんな解き方がある。実はこの指令は曖昧。

プログラム＝「明確な手順を指定した指令」

「1から10まで足す」プログラム例：

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

```
for (i = 1; i <= 10; i++){
    sum = sum + i
}
```

プログラムはどう実行されるか？

コンピュータはプログラムをどう実行する？

- 部分に区切って構造へ、構造を変化させて簡単な命令列へ変換後に実行する。

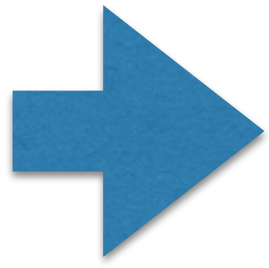
人が書くプログラムは難しいため、より簡単なプログラム (命令列) に変換する

コンピュータはプログラムをどう実行する？(1)

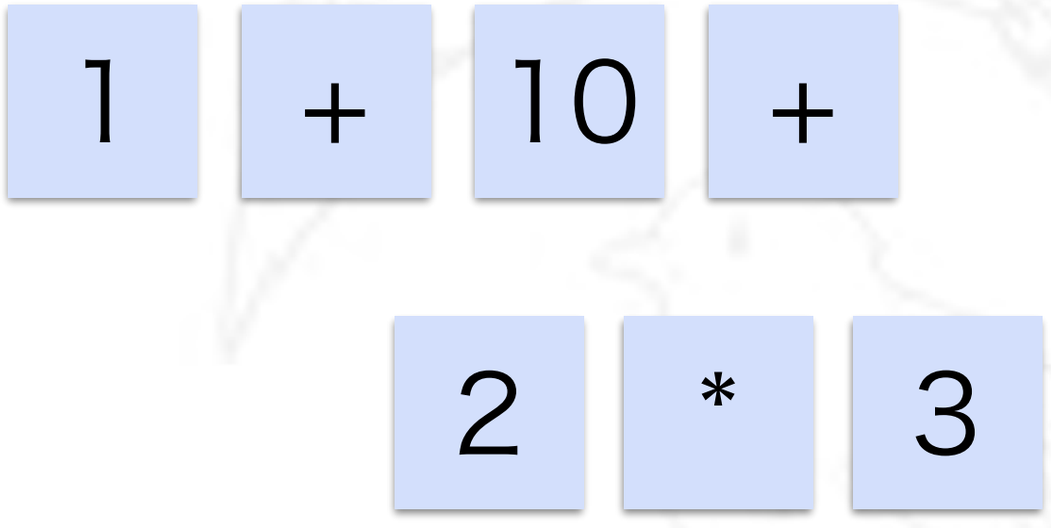
字句解析

プログラム：

```
1 + 10 + 2 * 3
```



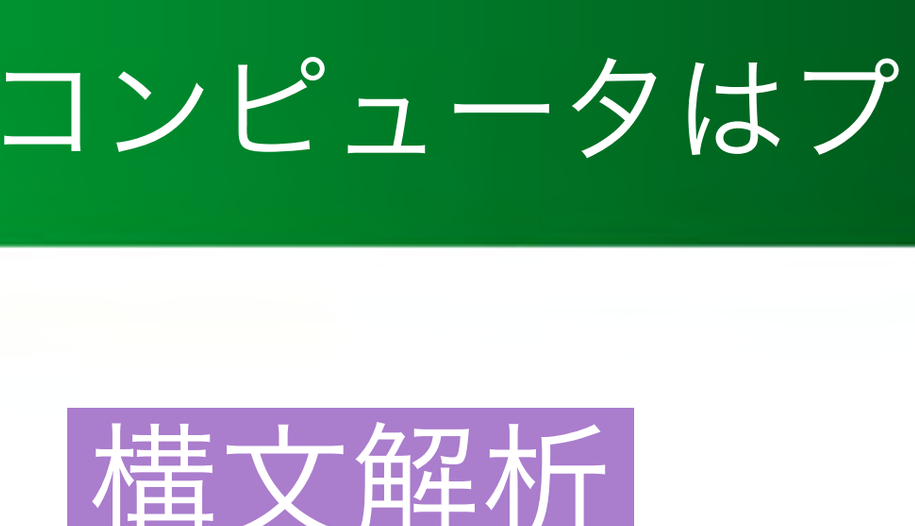
トークン列：



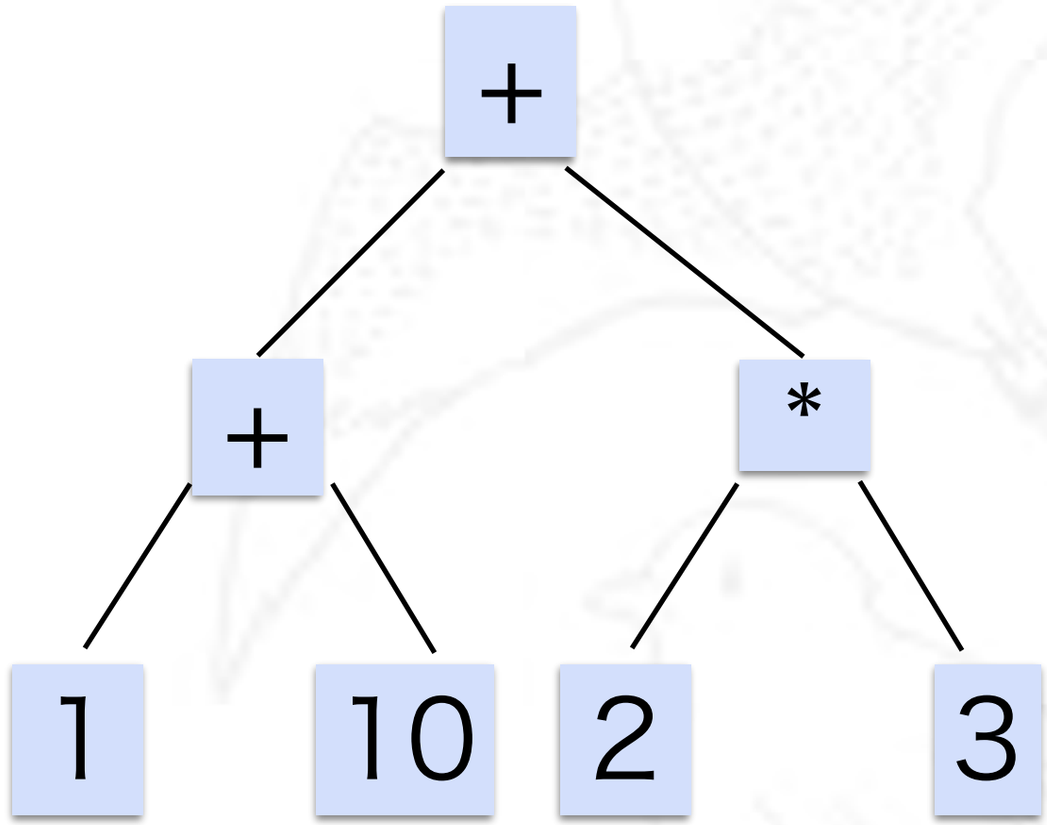
コンピュータはプログラムをどう実行する？(2)

構文解析

トークン列：



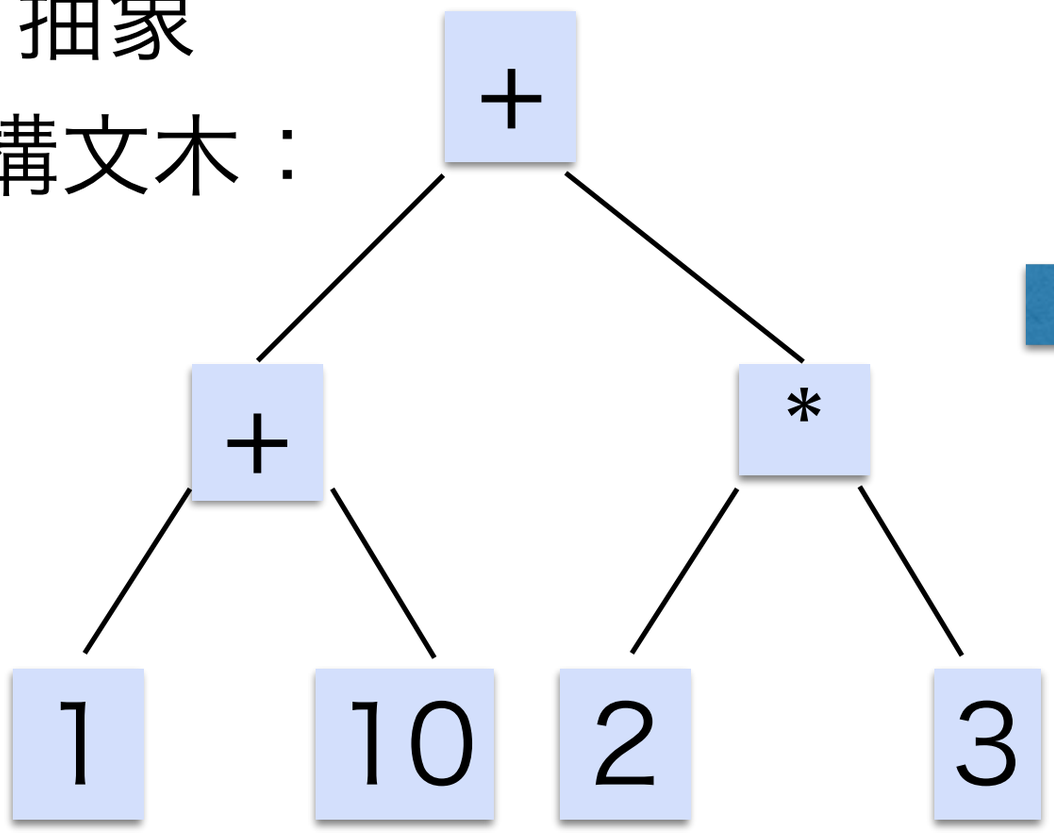
抽象構文木：



コンピュータはプログラムをどう実行する？(3)

各種変換・最適化・コード生成

抽象構文木：

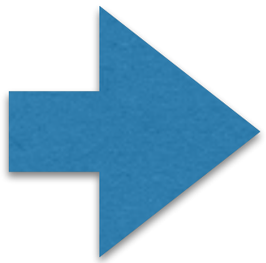


機械語

コンピュータはプログラムをどう実行する？(4)

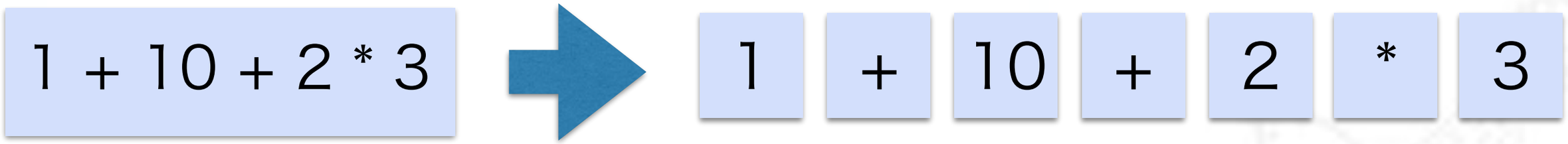
プログラムの実行

機械語

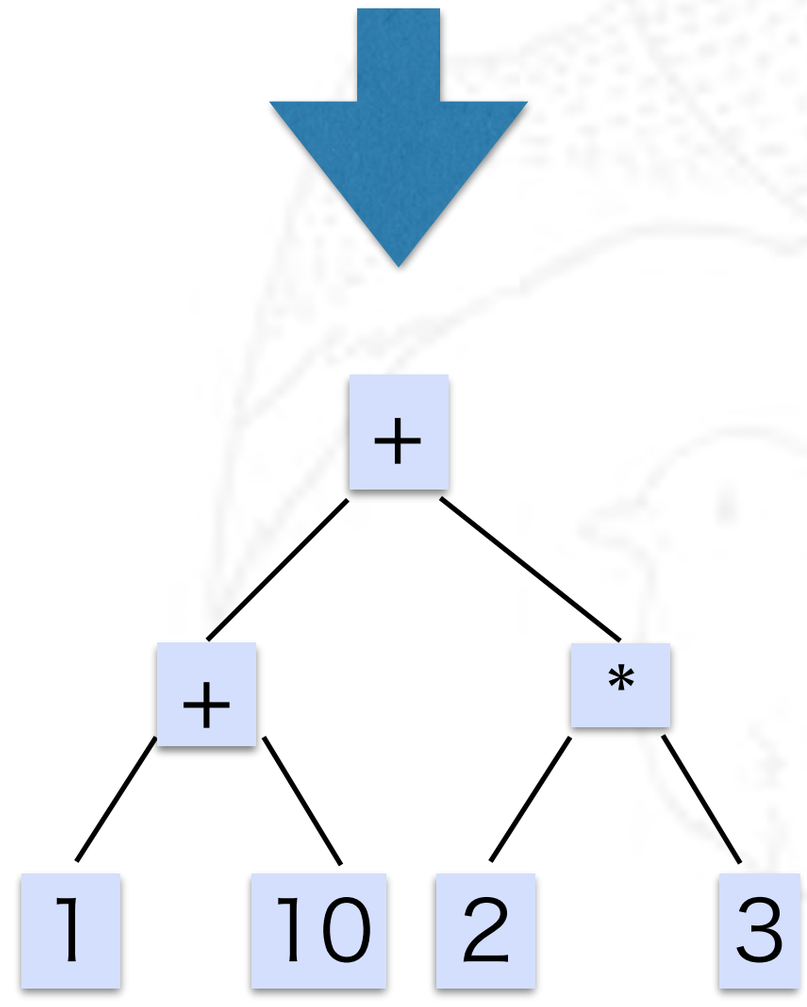


17

まとめ：コンピュータはプログラムをどう実行する？



- 部分に区切って構造へ、
構造を変化させて簡単な命令列へ、その後実行。



機械語

コラム 1: 構文解析

- 字句解析・構文解析は一般的にツールを使って書く (lex, yacc)
- 構文解析器は古くからあるが、現在も研究されているトピック
- いくつのトークンを先読みするかなど難しい点が多い
- 興味があれば：「Compilers: Principles, Techniques, and Tools」Alfred V. Aho et.al., 「LR構文解析の原理」大堀淳

正しいプログラムとはなにか？

正しいプログラムとはなにか？

1. コンピュータが解釈できる

2. 人の思ったように動作する

「コンピュータが解釈できない」とは？

文字列：

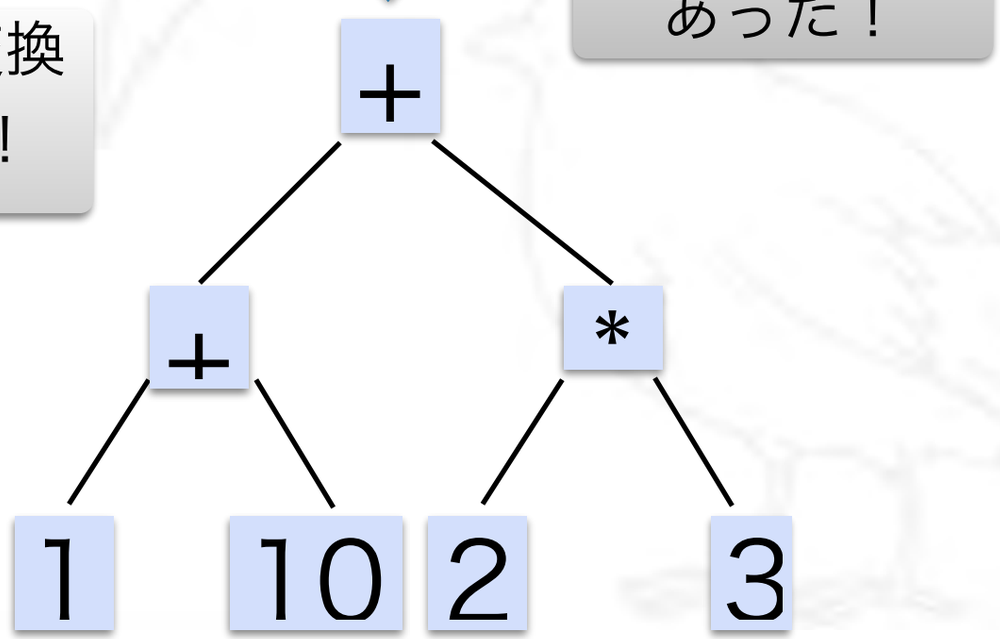
1 + 10 + 2 * 3

入力できない
文字があった！

トークン列：

1 + 10 + 2 * 3

変換できない
トークン列が
あった！



規則上、変換
できない！

機械語

「コンピュータが解釈できない」とは？

- プログラムに使えない文字

例：🐫 理解できない！

- 変換できないトークン列

例：１ ＋ * ３ 足し算？掛け算？それ以外のなにか？

- 規則上変換できない

例：ｘ ＋ ｘ 定義されていない ｘ .. なに？

正しいプログラムとはなにか？

1. コンピュータが解釈できる

→ 実行はできる

2. 人の思ったように動作する

「人の思ったように動作しない」とは？

```
irb(main):001:0> "2" + "3"
=> "23"
```

(言語：ruby)

❌ ２＋３を計算したいけれど、結果が５にならない ..

正しいプログラムとはなにか？

1. コンピュータが解釈できる

→ 実行はできる

2. 人の思ったように動作する

しかし、常に 意図通り 動くように書くのは難しい ..

簡単なプログラムでさえ…

(言語：OCaml)

```
# 1 + 4611686018427387903;;
- : int = -4611686018427387904
```

1 と大きい数字を足すだけのプログラム。
結果は負の数になってしまう ..

プログラムの正しさをどう高めるか？

プログラムの正しさをどう高めるか？

- 「人間の思ったように動作しない」ことを減らしたい .. いくつか方法がある

1. テスト (事前に実行する例) を増やす
2. データの種類に注目して、書けるプログラムに「制限」を設ける
3. 性質を検証する

データの種類 = 型

データの種類 =「型」

- ２ や ５ : 数
- “hello” : 文字列
- + : 数 * 数 → 数 (数を二つ受け取って、数を返す)

型を使って「制限」する

- +：数 * 数 → 数 (数を二つ受け取って、数を返す)

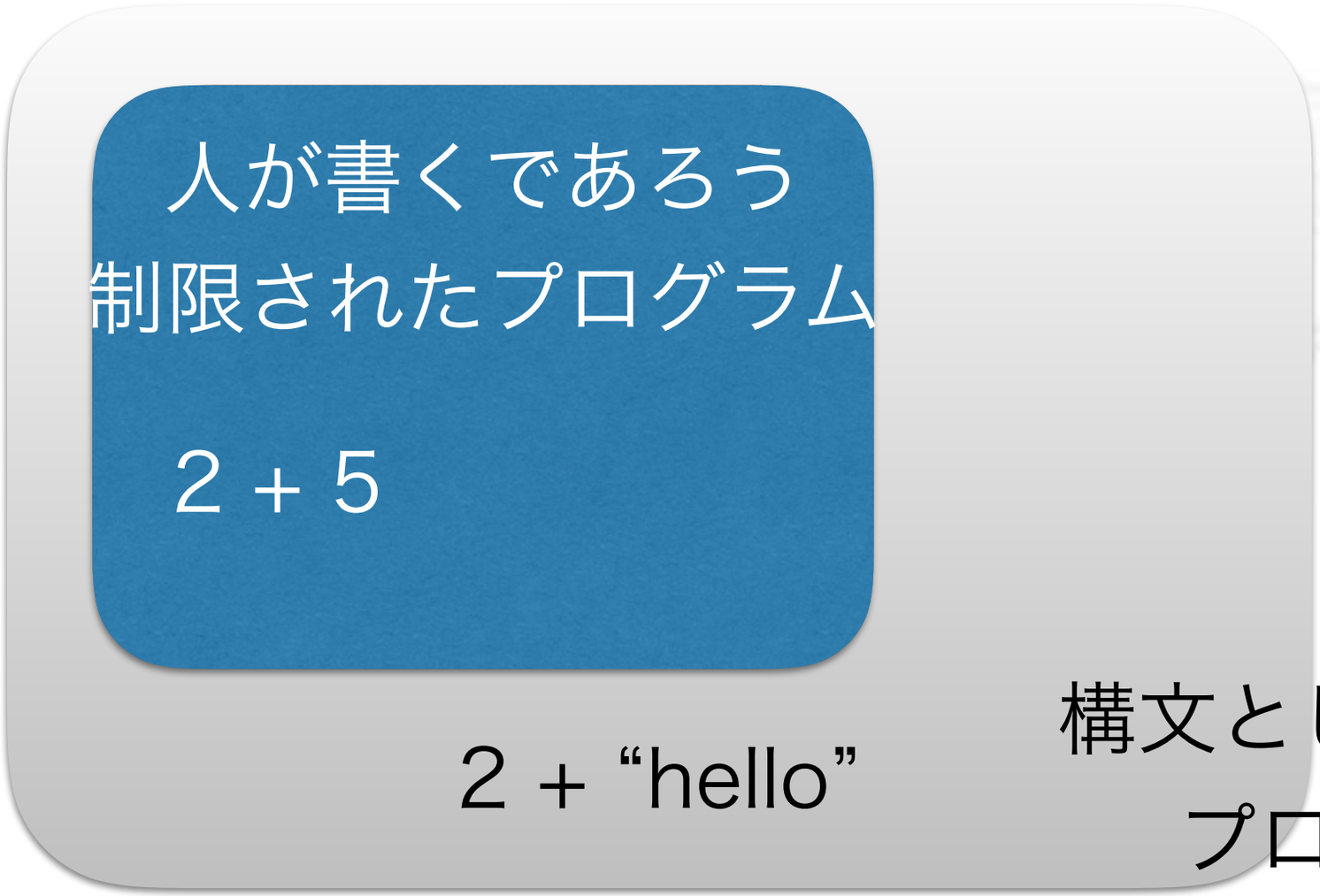
→ 2 + “hello” は制限にひっかかる

= 型エラー

- 動作を予期しないプログラムを除くことができる

型を使って「制限」する

イメージ：



いつ「型」を検査するか？

1. 静的型付け

   - プログラムの実行前にチェックする

2. 動的型付け

   - プログラムの実行時にチェックする

静的型付け (実行前検査)

```
# if true then 5 else 2 + "hello"
  ;;
Error: This expression has type
string but an expression was expected
of type int
```

(言語：OCaml)

⭕ (言語によるが) 正しく型がつけば、実行時に型に関してのエラーが出ない

❌ 実際には実行されない部分で型エラーを出して止まるかもしれない

動的型付け（実行時検査）

```
irb(main):004:0> if true then 5
else 2 + "hello" end
=> 5
```

(言語：ruby)

- 実行しながら検査して、型エラーを出す

⭕ 実行されない部分がどうかは気にしない

❌ テスト例では良くても他の例ではエラーになるかも

コラム 2: より制限の強い型

- 型にもいろいろある
  - これまで使った単純なデータの種類
  - 値の性質まで入った型 (性質の例：2 で割り切れる数, こちらの方が大きい …)
    参考：LiquidHaskell

「型」の難しさ

⭕制限に引っかからない場合、「意図通り動く」可能性が高い

❌「制限に引っかかる」と言われた場合、どうにか修正しないといけない …

型エラーのデバッグ手法

型エラーの本質的な難しさ

- 型エラーのプログラム：2 + “4”
- 正しいプログラムはなに？
  - 2 + 4 → 6
  - “2” + “4” → “24”
  - 2 == “4” → false

→ プログラマの意図次第で変わる、
どれが正しいかは自動的には決められない

型エラーのデバッグ手法

- エラーに関係している部分を見つける
- 対話的に間違っている部分を見つける
- 「最もエラーらしいもの」を見つける

型エラースライス

・ エラーに関係している部分を見つける

例：型エラーのプログラム 2 + “hello”

・ もし + の型が数字 * 数字 → 数字ならば、
2 は型エラーに関係がない

→ … + “hello” が型エラーに関係して
いる部分「型エラースライス」

対話的型エラーデバッガ

- 対話的に間違っている部分を見つける

例：型エラーのプログラム 2 + “hello”

- 「+ の型は 数字 * 数字 → 数字 ですか？」のようにプログラマの意図を聞きながら、質問を繰り返す

- プログラマの意図と違うところを見つける

型エラー原因箇所推定

- 「最もエラーの原因である可能性が高いところ」を見つける

例：型エラーのプログラム 2 + “hello”

→ + と 2 は型が合っている

→ 他と合わない “hello” が間違いの可能性が高い

自分の研究の話 (宣伝)

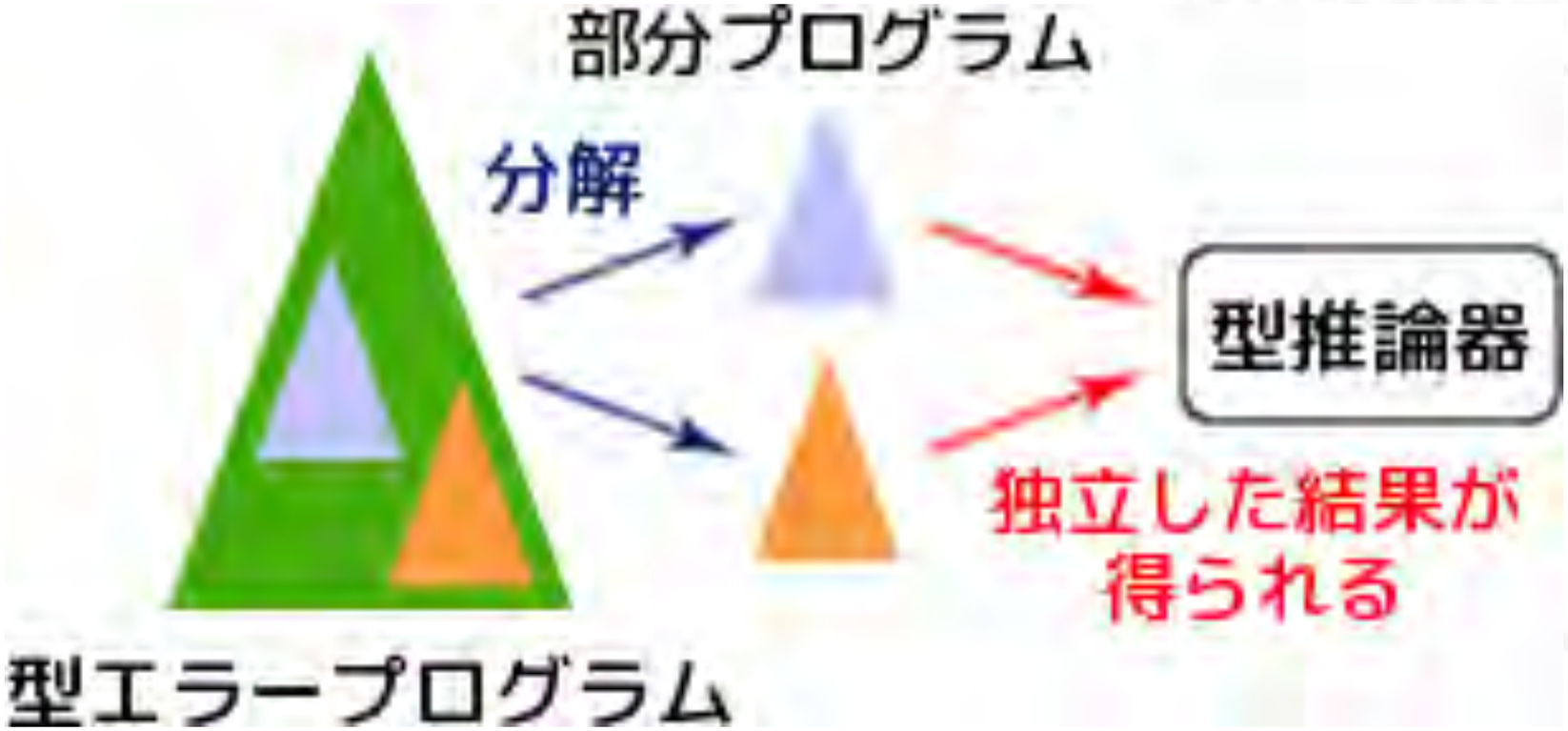
- 既にある「コンパイラ」の型を求める機能を再利用して、デバッグ手法を実装する

自分の研究の話 (宣伝)

- 既にある「コンパイラ」の型を求める機能を再利用して、デバッグ手法を実装する

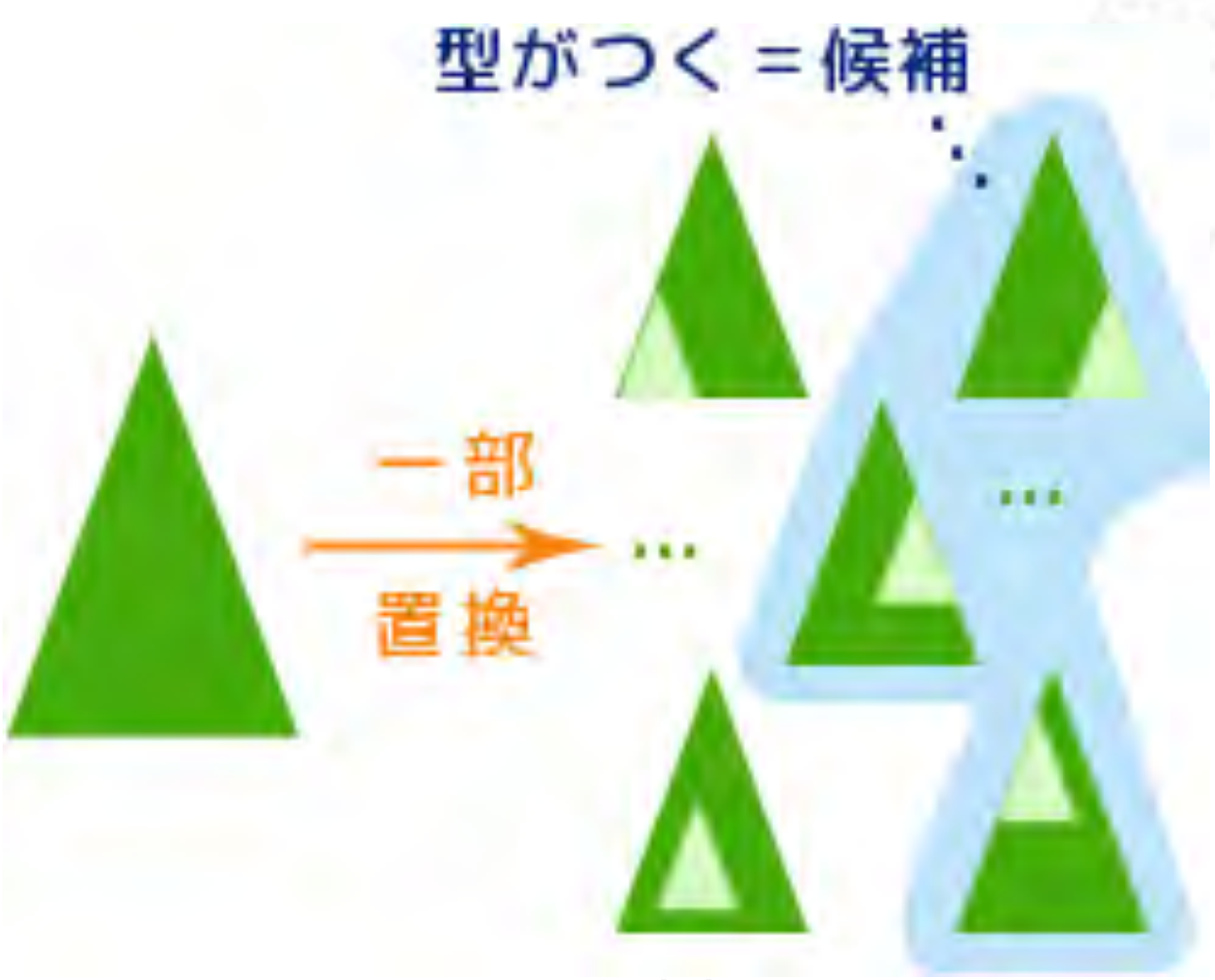
自分の研究の話 (宣伝)

- 既にある「コンパイラ」の型を求める機能を再利用して、デバッグ手法を実装する



自分の研究の話 (宣伝)

- 既にある「コンパイラ」の型を求める機能を再利用して、デバッグ手法を実装する



型以外の手法

プログラムの正しさをどう高めるか？

- 「人間の思ったように動作しない」ことを減らしたい .. いくつか方法がある

1. テスト (事前に実行する例) を増やす
2. データの種類に注目して、書けるプログラムに「制限」を設ける
3. 性質を検証する

1. テスト

- 多くの例を実行してみて意図通りでない場合には修正する

⭕ 簡単で取り入れやすい

❌ うまくいかない例を見つけられる？

1. テスト例自動生成

- テスト例の自動生成 & 自動検査も現在も研究されているトピック
  - 満たすべき性質を記述、自動で検査
- 自動生成手法は様々
  - ランダムに作る (QuickCheck)
  - 小さいものから全通り作る (smallcheck) など

3. 性質の検証

- 記述した性質が満たされるか「証明」する
- 定理証明系言語 (Coq, Agda)、自動定理証明など
- 古くからあるが、ここ 10 年ほどホットトピック
- 「コンパイラ」の証明 (CompCert) も！

本講座で話したこと

- 「プログラム」とはどんなものか
- プログラムの「正しさ」とはなにか
- 「型」とそのデバッグ手法とはどんなものか
- 型以外に正しさを高める手法の紹介

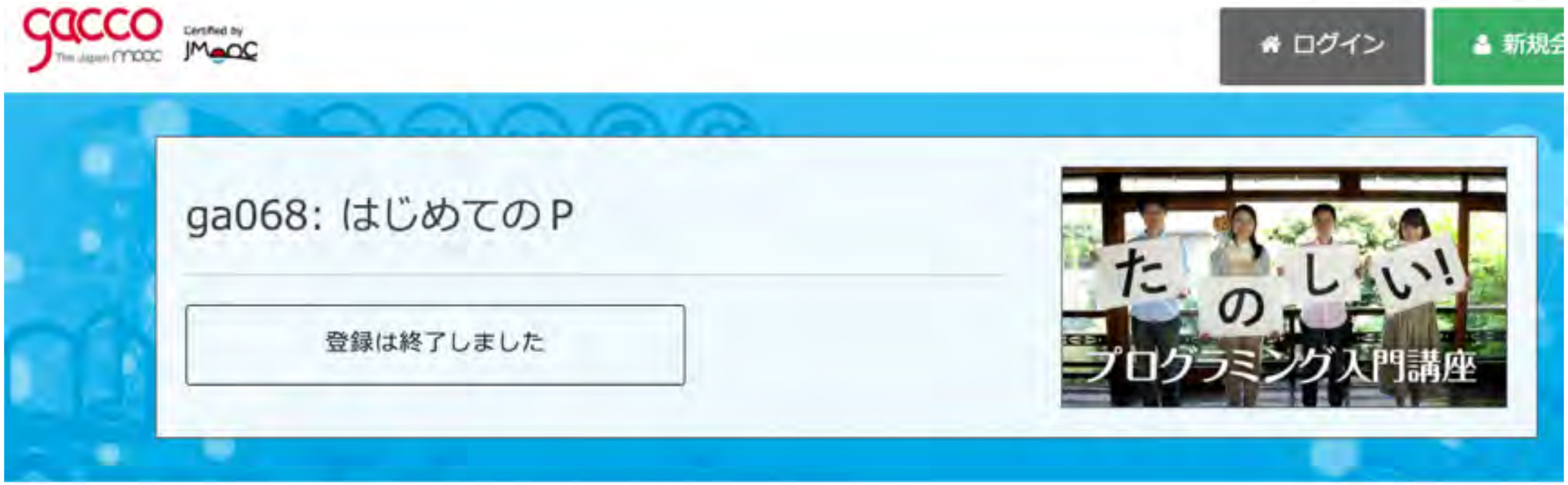
プログラムを書くには？

プログラミングの学び方

- 作りたいものをつくる (ゲーム, Web アプリ, iPhone アプリ)
- プログラミングコースに参加してみる
- 競技プログラミング・プログラミングコンテストに挑戦してみる

gacco はじめての P (宣伝)

- プログラミング初心者向け、全 4 回



講座概要

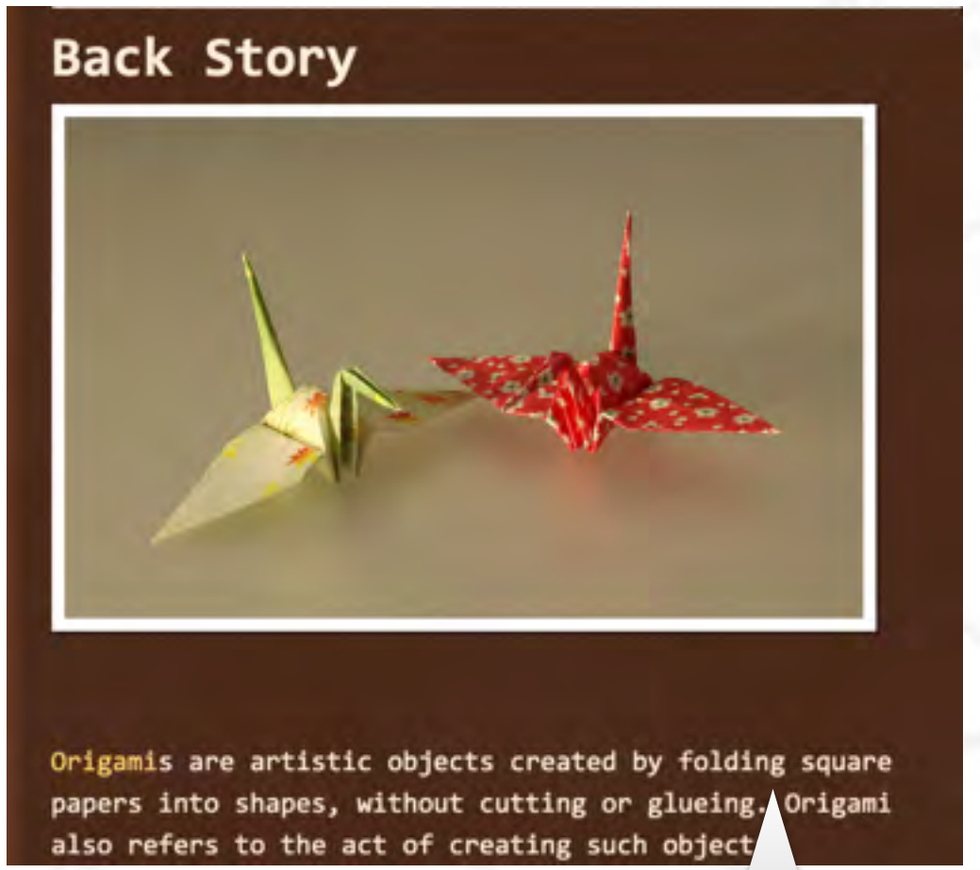
講座内容

現代社会のありとあらゆる場所にはコンピュータがあり、それらは全てプログラムで動いています。国立情報学研究所のプログラミング入門講座「はじめてのP」は、できるだ

ICFP programming contest (宣伝)

ICFP という学会が主催

ICFPc 2011



ICFPc 2016

参考文献

構文解析・コンパイラ

- 「Compilers: Principles, Techniques, and Tools」
  Alfred V. Aho et.al.
- 「LR構文解析の原理」大堀淳
- Mincaml コンパイラ 住井英二郎

型

- 「型システム入門 –プログラミング言語と型の理論–」,
  Benjamin C. Pierce, 住井英二郎 監訳