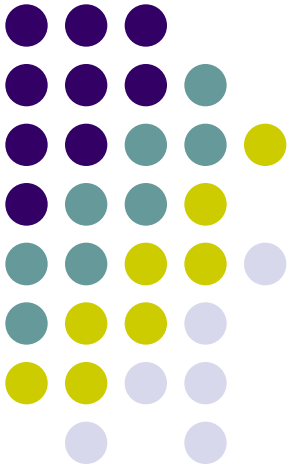
ソフトウェアサイエンス概論 II
プログラム言語の科学

亀山 幸義

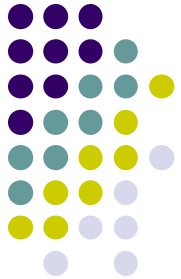
http://logic.cs.tsukuba.ac.jp/~kam



今日の話題

- べき乗関数
- プログラム特化
- メタプログラミング

べき乗の関数(1)



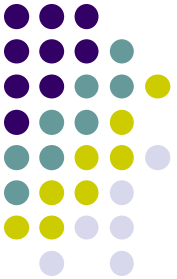
```
let rec  power  x  n =
   if  n = 0  then  1
   else  x  *  (power  x  (n - 1))
```

power 3 1 → 3

power 3 2 → 9

power 3 4 → 81

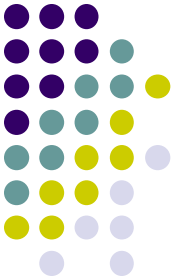
べき乗の関数(2)



```
let rec  power  x  n =
  if  n=0  then  1
  else  x  *  (power  x  (n - 1))
```

問題。n=12に対して、power関数を何度も繰り返し使いたい。どうすれば効率的に計算できるか？

べき乗の関数(3)



```
let power12  x =
  x*x*x*x*x*x*x*x*x*x*x*x
```

アルゴリズム自体は同じ

```
let power12'  x =
  let  x2=x*x in
  let x4=x2*x2 in    x4*x4*x4
```

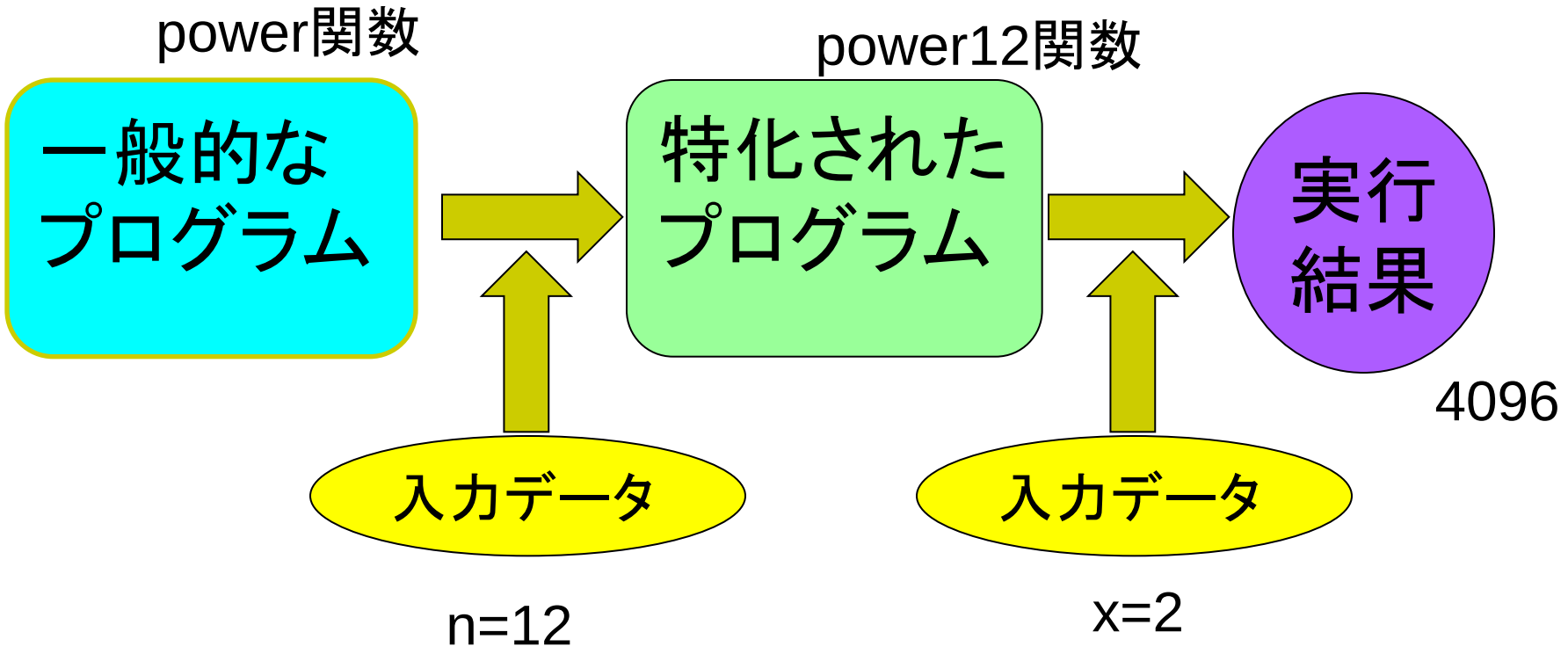
アルゴリズム自体の改善

べき乗の関数(4)

```
let rec  power  x  n =
  if  n=0  then  1
  else  x ＊ (power  x  (n - 1))
let  power12  x = x*x*x*x*x*x*x*x*x*x*x*x
```

問題2. power関数からpower12関数を自動的に得るにはどうしたらよいか？

プログラム特化（部分計算）



一般的なプログラムと、一部の入力データから、
そのデータに特化したプログラムを得る。

べき乗関数の特化（1）

```
let rec  power  x  n =
  if  n=0  then  1
  else  x  *  (power  x  (n - 1))
```

プログラム特化における仮定

nの値が分かっている

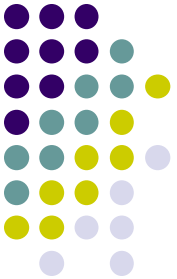
xの値は分からない

べき乗関数の特化(2)

```
let rec  power  x  n =
  if  n = 0  then  1
  else  x ＊ (power  x  (n - 1))
```

nのみに依存した計算(xに依存しない計算)は、「今」やってしまえばよい。

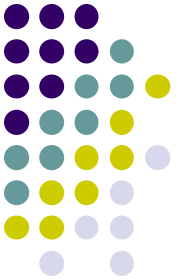
べき乗関数の特化(2)



```
let rec  power  x  n =
  if  n = 0  then  1
  else  x  *  (power  x  (n - 1))
```

nのみに依存した計算(xに依存しない計算)は、「今」やってしまえばよい。

べき乗関数の特化(2)



```
let rec  power  x  n =
  if  n = 0  then  1
  else  x * (power  x  (n - 1))
```

nのみに依存した計算(xに依存しない計算)は、「今」やってしまえばよい。

べき乗関数の特化(2)

```
let rec  power  x  n =
  if  n = 0  then  1
  else  x * (power  x  (n - 1))
```

nのみに依存した計算(xに依存しない計算)は、「今」やってしまえばよい。

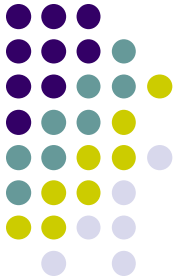
べき乗関数の特化(3)

```
let rec  power  x  n =
  if  n = 0  then  1
  else  x * (power  x  (n - 1))
```

赤色:　今できる計算(静的)

黒色:　後に残る計算(動的)

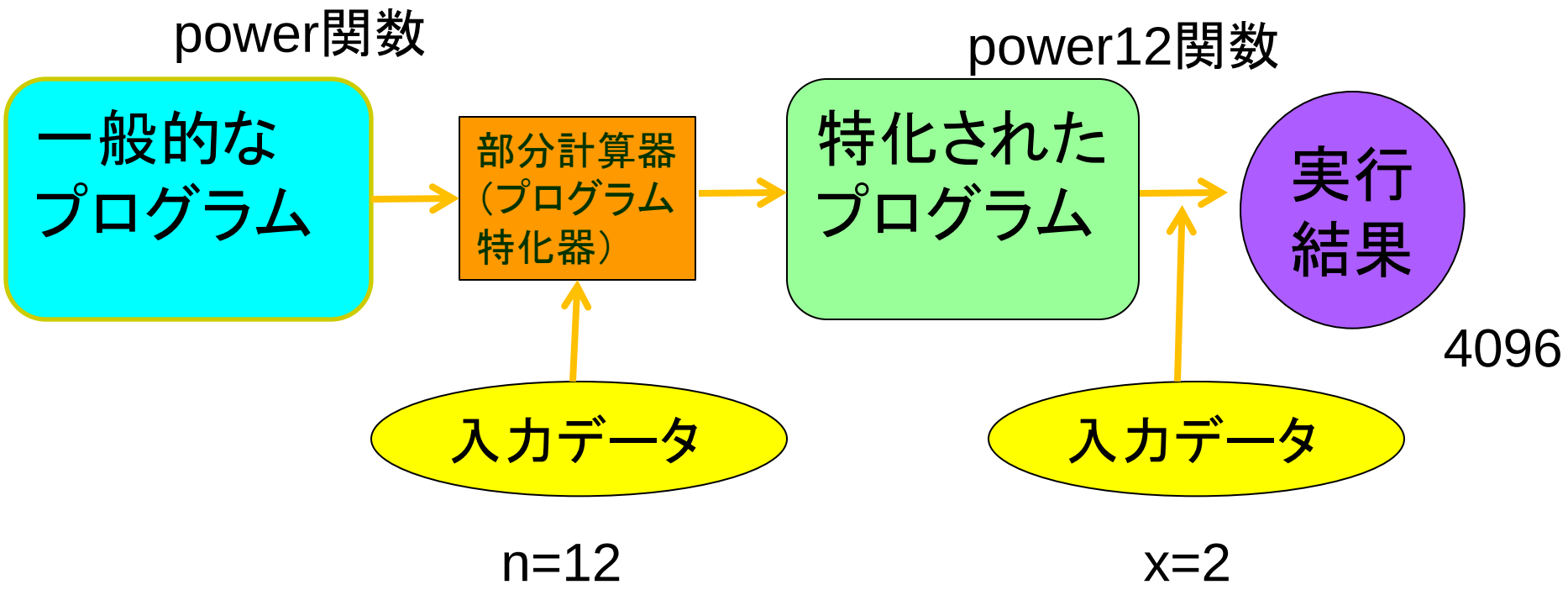
べき乗関数の特化（4）



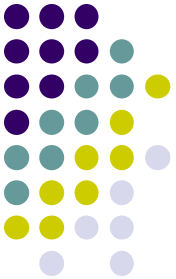
```
let rec  power  x  n =
  if  n = 0  then  1
  else  x * (power  x  (n - 1))
```

```
power  x  12
-> if 12=0 then 1 else x*(power x (12-1))
-> x*(power x 11)
-> x*(x*(power x 10))
->... ->   x*(x*(x*(x*(x*(x*(x*(x*(x*(x*(x*(x*1)...)
```

プログラム特化（部分計算）



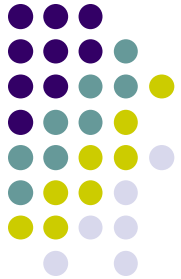
プログラム特化の例
SKKソフトウェア [Sato 1987]



```
(defun skk-insert (&optional arg parg)
  ...
  (and (memq ch skk-set-henkan-point-key)
    (or skk-okurigana
        (not (skk-get-prefix skk-current-rule-tree))
        (not (skk-select-branch skk-current-rule-
  ...))
...
```

静的に（ユーザ個人の設定ファイルで）
値が決まる変数がたくさん含まれる
→特化により、コード量で10-40%の削減

プログラム特化の例
SKKソフトウェア [Sato 1987]



一般的なプログラム＝SKKのプログラム

静的なデータ＝ユーザの個人設定ファイルにおける各変数の設定（「送り仮名あり」など）

動的なデータ＝文章（かな漢字変換の対象）

＝＞　プログラム特化により、プログラムの効率化を達成（プログラムのサイズの大幅な縮小、実行時間の短縮）

プログラム特化の例
yacc (以下はocamlyaccの例)

新しいプログラム言語を作成する際の便利なツール

その言語の構文の定義を記述すると、構文解析プログラムを自動的に作成してくれる。

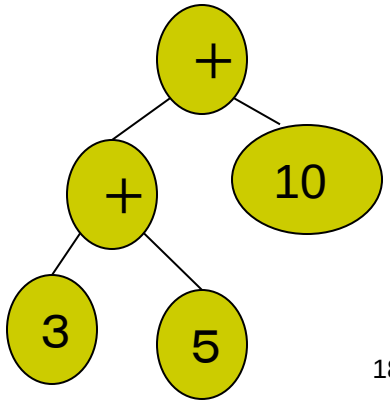
対象言語の構文の定義

<exp> ::= <num> | <exp> “+” <exp>

対象言語のプログラム

3 + 5 + 10

構文解析: 文字列(あるいはトークン列)をもらって構文木を返す。

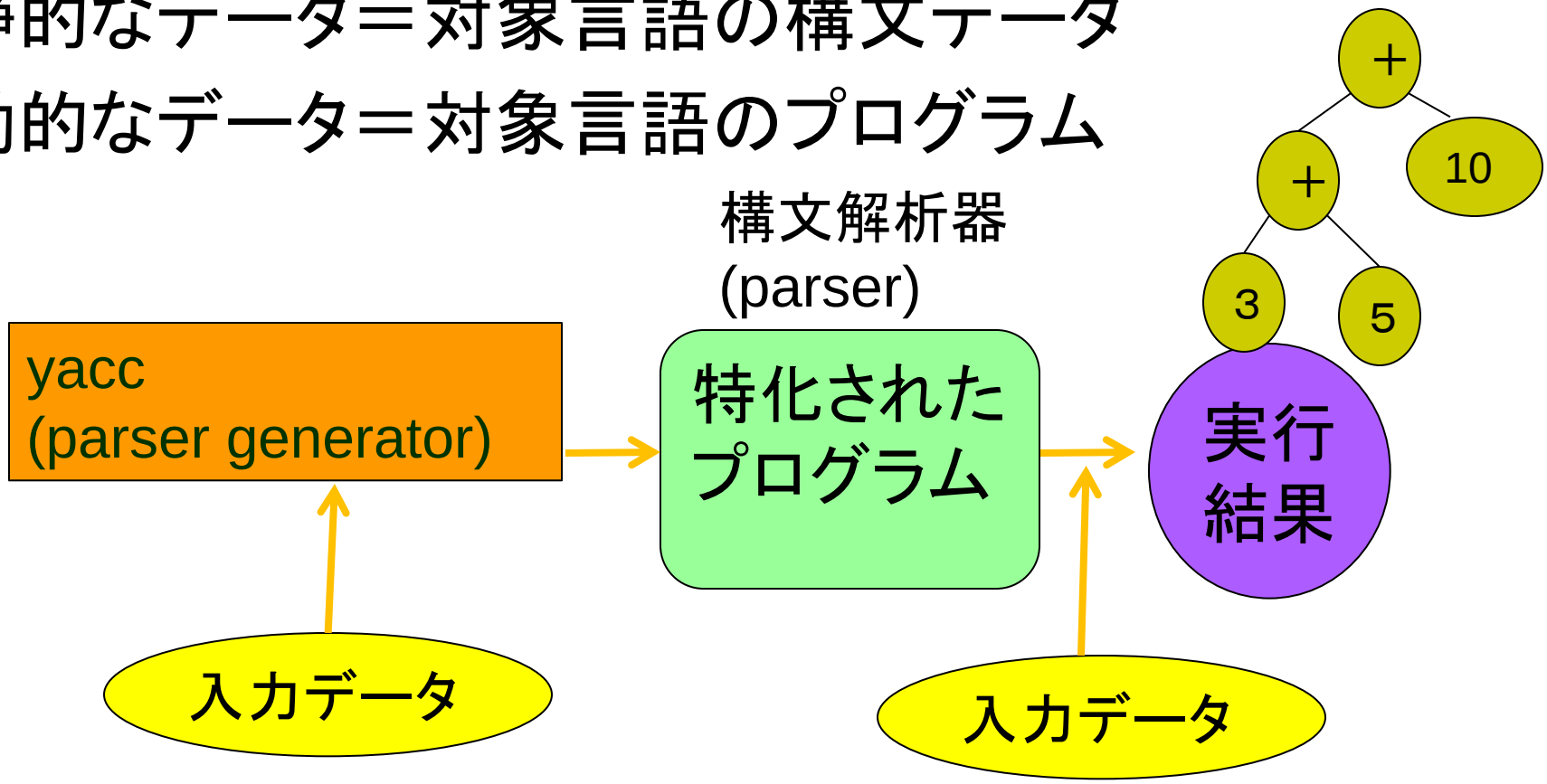


プログラム特化の例
yacc

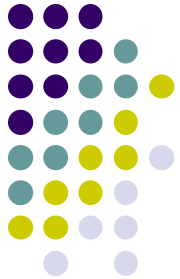
一般的なプログラム＝yacc

静的なデータ＝対象言語の構文データ

動的なデータ＝対象言語のプログラム



プログラム特化の例
インタープリタの高速化1



新しいプログラム言語を作成する際の便利なツール

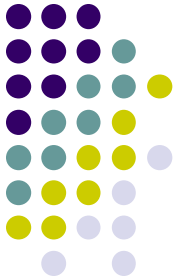
その言語のインタープリタの定義を記述すると、高速のインタープリタを自動的に作成してくれる。

対象言語のインタープリタの定義

```
let rec  eval  exp env = match  exp  with
   Var(x) ->  lookup x  env
|  Plus(e1, e2) ->  (eval e1) + (eval e2)
   ...
```

素朴なインタープリタは遅い！

プログラム特化の例
インタープリタの高速化2

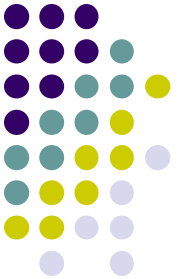


```
let rec  eval  exp env = match  exp  with
     Var(x) ->  lookup x  env
   |  Plus(e1, e2) ->  (eval e1) + (eval e2)
      ...
```

素朴なインタープリタは遅い！

理由。計算の途中に構文木を何度も走査（スキャン）する

プログラム特化の例
インタープリタの高速化3



```
let rec  eval  exp env = match  exp  with
     Var(x) ->  lookup x  env
   |  Plus(e1, e2) ->  (eval e1) + (eval e2)
      ...
```

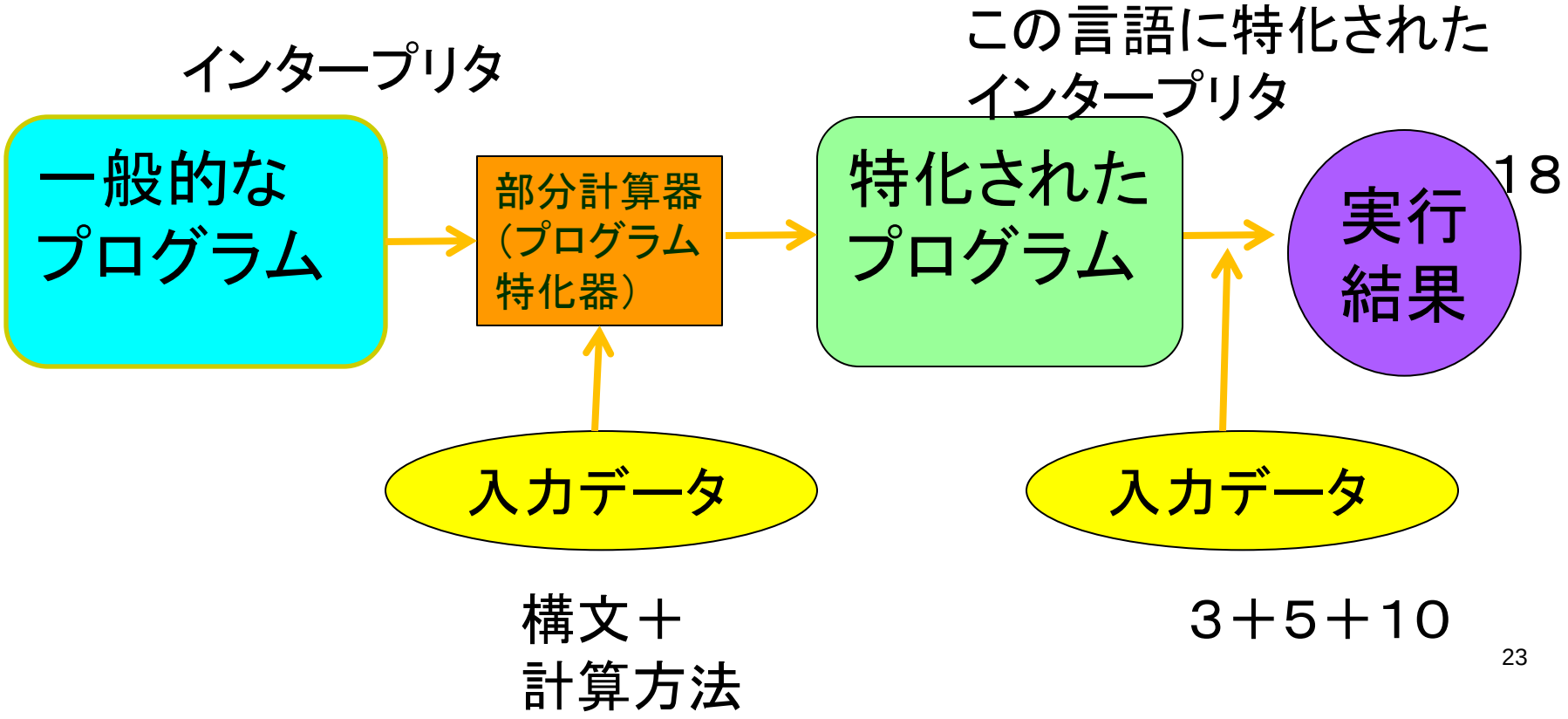
素朴なインタープリタは遅い！

解1. 対象言語をよく解析してコンパイラを作成

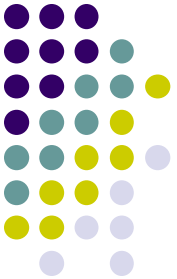
解2. インタープリタを対象言語に特化させる

プログラム特化（部分計算）

一般的なプログラム＝インタープリタ
静的なデータ＝対象言語の構文と計算方法
動的なデータ＝対象言語の個々のプログラム

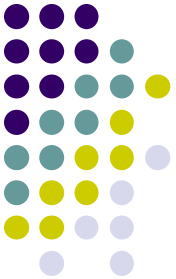


プログラム特化は メタプログラミングの一例



- メタプログラム
  - （プログラムをもらったり）、プログラムを返したりするプログラム
- メタプログラムの例
  - コンパイラ、インタプリタ、構文解析器生成プログラム
  - 実は身の回りにあふれている

メタプログラミングの技法



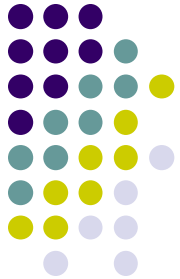
1. 文字列として生成する

（例: Rubyのリフレクション）

```
let rec gen_power n =
  if n = 0 then "1"
  else "x * (" ^ (gen_power (n-1)) ^ ")"
gen_power 3 =
  "x*(x*(x*1))"
```

非常に読みにくい。(理解しにくい。)
間違いやすい。間違っているかどうかは、プログラムを生成してみないとわからない。

メタプログラミングの技法



2. 支援ツールを使う

- C++ などの template
- Lisp系言語 quasi-quote, unquote

```
(define  (gen_power  n)
    (if   (=  n  0)  `1
          `(*  x  ,(gen_power (-  n  1))))
```

生成されるプログラムは、「かっこ」が必ず合う。(構文は常にOK)
読みやすい（通常プログラムと似ている）
変数x は一体どこから来るか？ 自由変数を持つプログラムが生成できてしまう。
変数名の衝突が起き得る。

メタプログラミングの技法

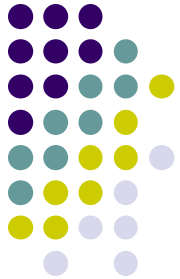
3. メタプログラム機能を持つプログラム言語を使う。

- MetaML, MetaOCamlなど

```
let rec gen_power cx n =
    if n = 0 then .< 1 >.
    else .< .~cx * .~(gen_power cx (n – 1))>.
.<fun x -> .~(gen_power .<x>. 12)>.
```

生成されるプログラムは、必ず構文的が正しい。
生成されるプログラムは、必ず型が整合的（かつ、自由変数を持たない。）
異なるメタプログラムをまぜて使っても、変数名の衝突が起きない。
これらの性質を、メタプログラムを実行する前に保証する。

メタプログラミング言語は研究途上



「途中結果を記憶しつつ計算する」関数の生成

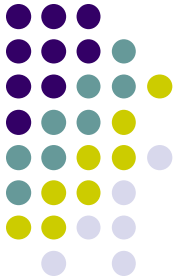
[Kameyama et al. 2009]

```
let  rec  g  n  x  y  =
     if  n = 0  then x
     else if n = 1 then y
     else  (g (n-2) x y) +  (g (n-1) x y)
```

g n 1 1　は　Fibonacci関数と一致。

単純な特化では非効率的ー＞効率的コード生成

メタプログラミング言語は研究途上



「途中結果を記憶しつつ計算する」関数の生成

[Kameyama et al. 2009]

単純に生成

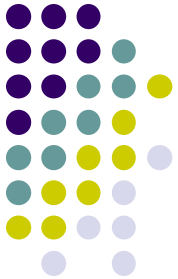
g 5 x y = (配布資料参照。同じ計算を何度も行う効率の悪いコード)

効率よいコードを生成

.<let x2=x+y in let x3=y+x2 in
let x4=x2+x3 in let x5=x3+x4 in x5>.

アルゴリズムも改善(単なる「特化」ではない)

メタプログラミング言語は研究途上



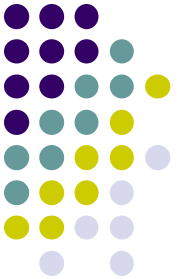
大きなギャップがある

- 理論的な安全性の保証;「副作用」のない言語
- 現実のニーズ;あらゆる種類のプログラムを生成できるようにしたい

チャレンジ

- 既存のプログラム言語を全部カバーするメタプログラム言語を設計(安全性の理論的保証、言語設計、効率的実装)

まとめ



- 今日の話題
  - プログラム特化
  - メタプログラミング
  - チャレンジ
- プログラム言語の研究
  - 保守性や拡張性と、実行効率の両立
  - 安全性や信頼性の保証
  - 理論と現実のバランス

今日の課題（氏名、学籍番号明記）

「プログラムを生成するプログラム」が、大きなメリットがあるのは、どのような場面か（どのような状況か） 例を1つ以上考えなさい。

今日の授業の要点をまとめなさい。