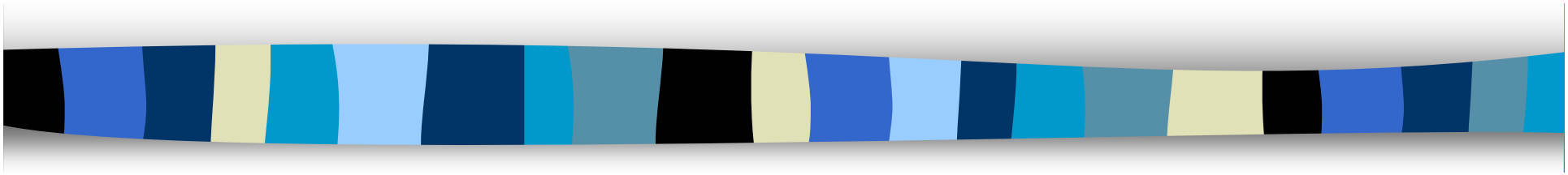
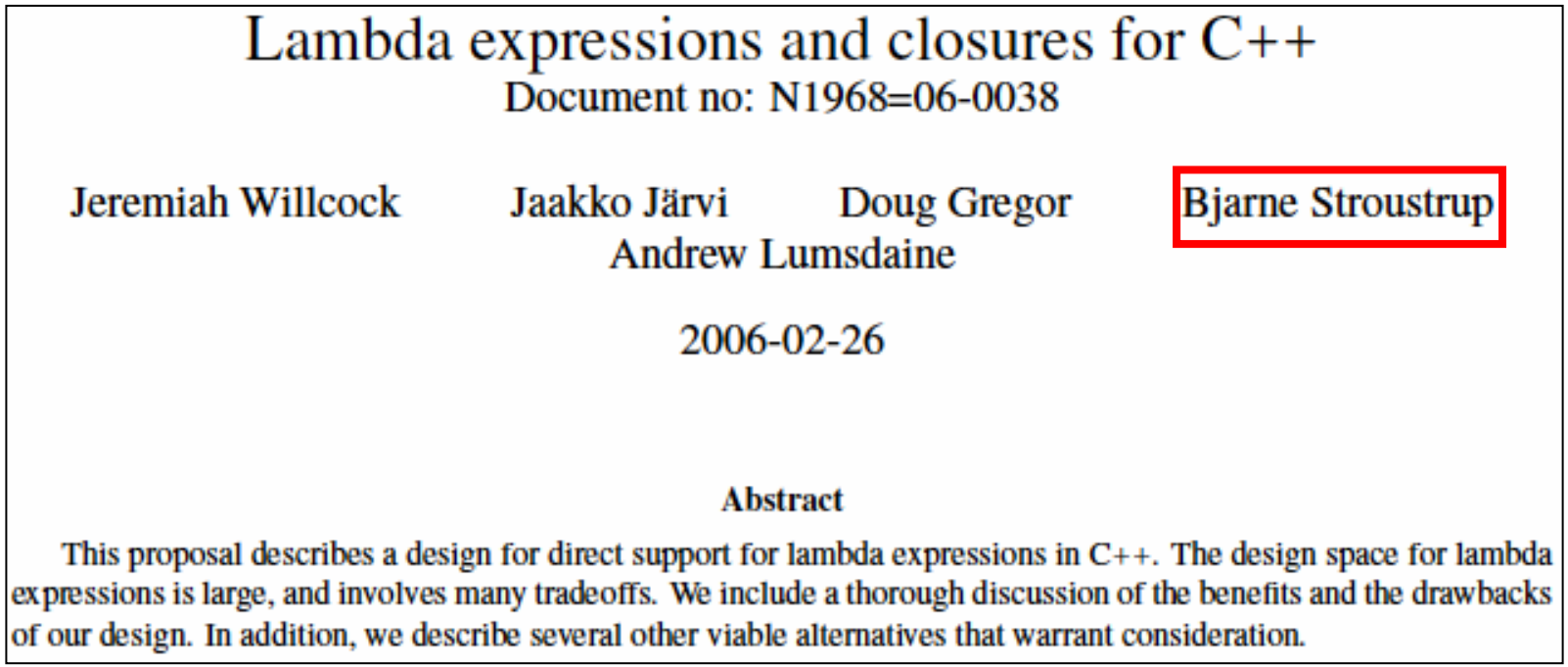
2時間で真似(まね)ぶ 関数型言語のコンパイラ



PPL**サマースクール**2006

住井 英二郎 (東北大学)

関数型言語ブーム？



Lambda expressions and closures for C++

Document no: N1968=06-0038

Jeremiah Willcock　Jaakko Järvi　Doug Gregor　Bjarne Stroustrup
Andrew Lumsdaine

2006-02-26

Abstract

This proposal describes a design for direct support for lambda expressions in C++. The design space for lambda expressions is large, and involves many tradeoffs. We include a thorough discussion of the benefits and the drawbacks of our design. In addition, we describe several other viable alternatives that warrant consideration.

関数型言語ブーム？

国内ニュース　日経ソフトウエア

ITpro ＞ 最新テクノロジー

「次のC#では，データとオブジェクトのミスマッチを解消したい」---C#の開発者が語る(1)

記事一覧へ▶

［画像のクリックで拡大表示］

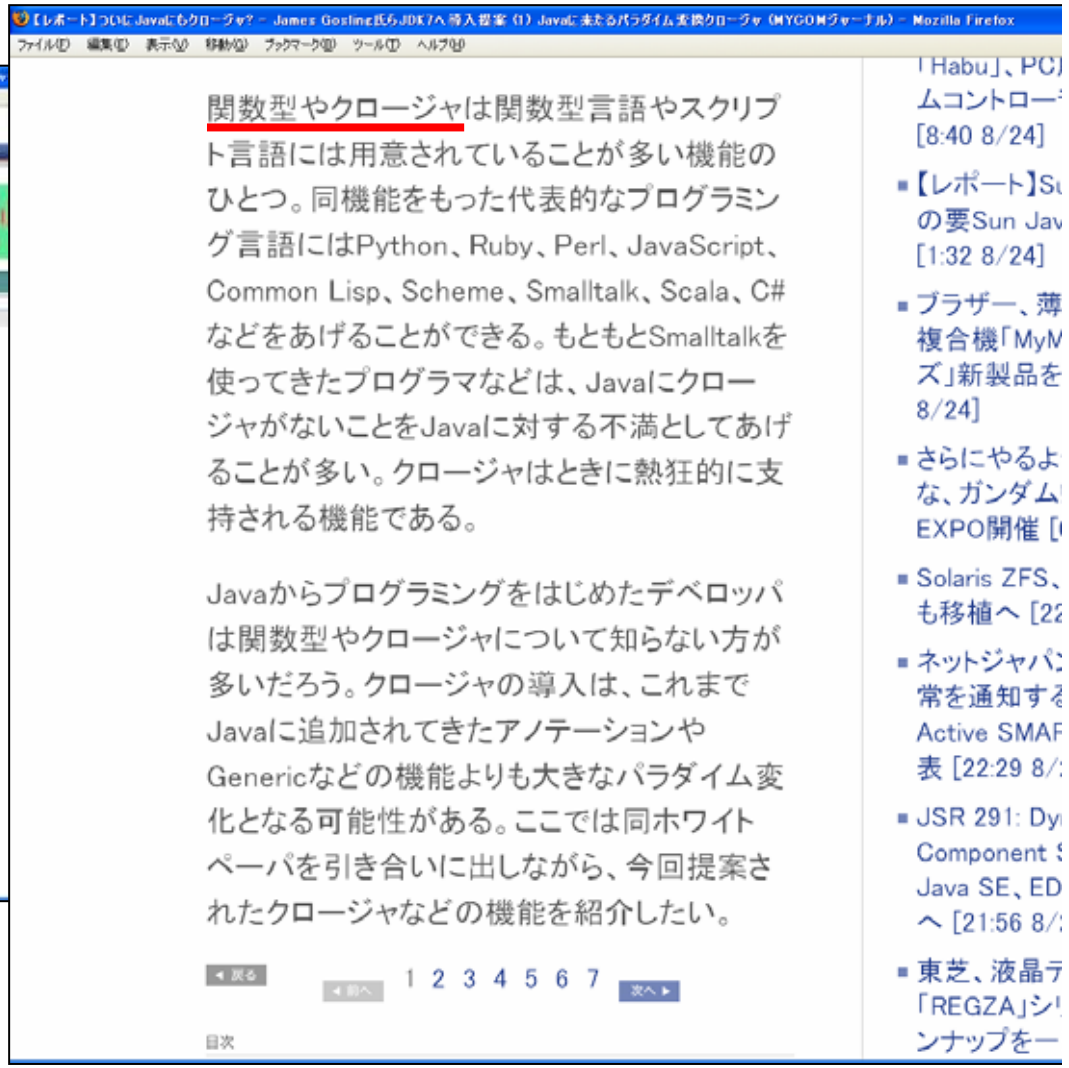
米MicrosoftのTechnical FellowであるAnders Hejlsberg（アンダース・ハイルズバーグ）氏がマイクロソフトの開発者会議で講演するために来日，本紙記者のインタビューに応じた。Hejlsberg氏はC#言語の開発者であり，.NET Frameworkの設計にも深く関わっている。新製品Visual Studio 2005や .NET Framework 2.0についてだけでなく，次世代のC#の計画についても語ってくれた。発言の要旨は以下の通り。

--次の .NET Framework，C# 3.0の目標はどこにありますか。それから，どのような機能を実装する予定ですか。

すでに少しお話ししましたが，C# 3.0で試みていることに一つには，データとオブジェクトのインピーダンス・ミスマッチを解消するということですね。技術的な話をすると，私たちには関数型言語（編集部注：Haskellが代表格）に関する研究の蓄積があります。その成果をC# 3.0に取り込もうとしています。オブジェクト指向言語と関数型言語を融合する媒体を作ろうとしているのです。これはとても興味深いことです。関数型言語にはとても魅力的な側面がありますが，数学的な厳しいルールがあるという指摘もあります。私たちは両者の良いところを合わせていこうと考えています。

関数型言語ブーム？


MYCOMジャーナル
【レポート】
ついにJavaにもクロージャ?
– James Gosling氏らJDK7
へ導入提案
(1) Javaに来たるパラダイム変換クロージャ
後藤大地
2006/8/23
Java言語の主要アーキテクトであるGilad Bracha氏、Neal Gafter氏、James Gosling氏、Peter von der Ahé氏らは18日(米国時間)、

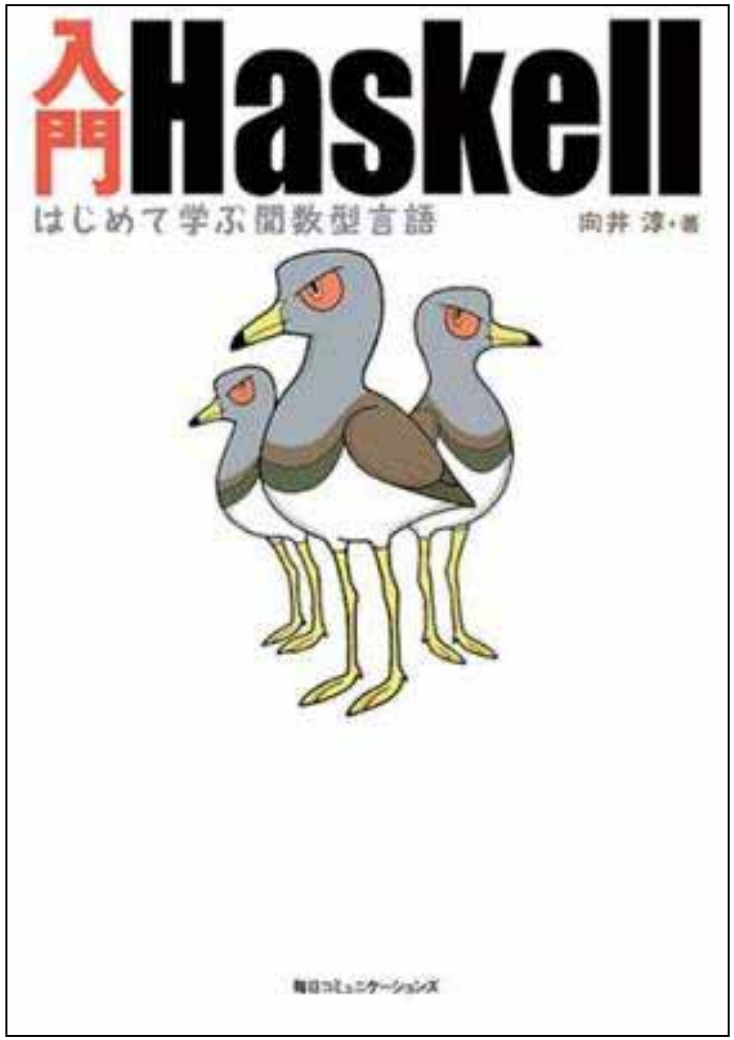

関数型やクロージャは関数型言語やスクリプト言語には用意されていることが多い機能のひとつ。同機能をもった代表的なプログラミング言語にはPython、Ruby、Perl、JavaScript、Common Lisp、Scheme、Smalltalk、Scala、C#などをあげることができる。もともとSmalltalkを使ってきたプログラマなどは、Javaにクロージャがないことを Javaに対する不満としてあげることが多い。クロージャはときに熱狂的に支持される機能である。
Javaからプログラミングをはじめたデベロッパは関数型やクロージャについて知らない方が多いだろう。クロージャの導入は、これまでJavaに追加されてきたアノテーションやGenericなどの機能よりも大きなパラダイム変化となる可能性がある。ここでは同ホワイトペーパを引き合いに出しながら、今回提案されたクロージャなどの機能を紹介したい。
1 2 3 4 5 6 7
目次

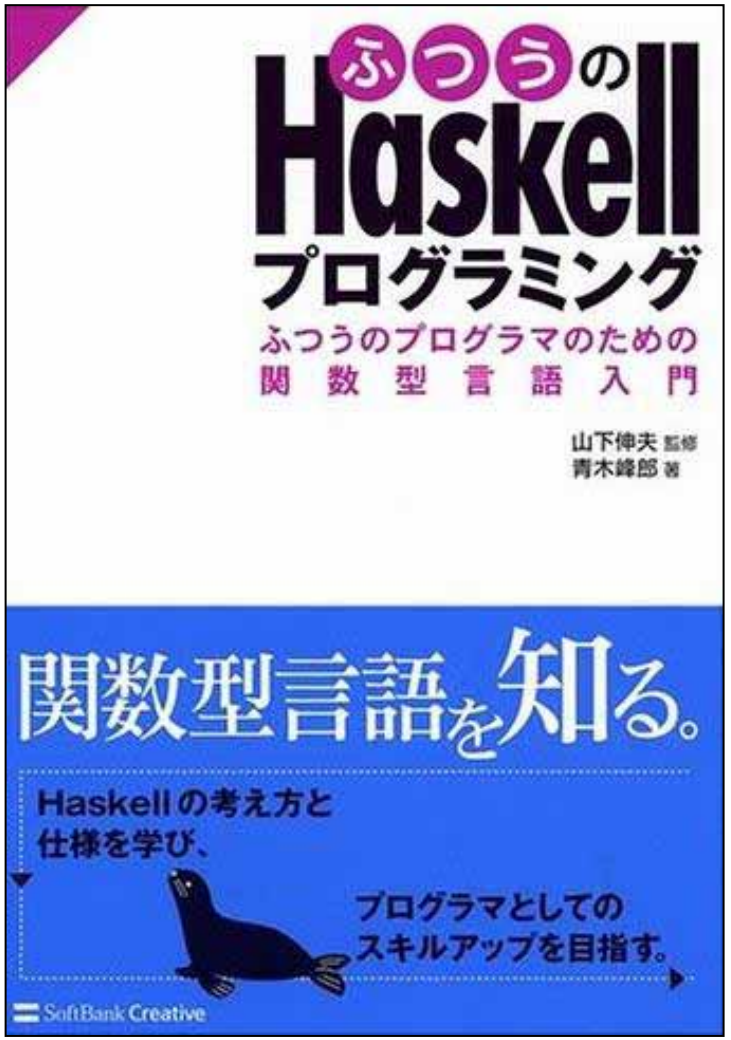
関数型言語ブーム？


プログラミング初心者にオススメの特別付録つき!
日経ソフトウエア
NIKKEI SOFTWARE
2006.06
通巻100号記念
プロのソフト技術者になるための
プログラミング実力アップ大作戦!
未経験者でもプログラミングができる方法教えます!
本物のWindowsプログラミングを学ぶ
注目の新技術OIをマスターして初心者脱出
これから流行る？関数型プログラミング
プロなら知っておきたいソフトウエア・ファクトリ
特別レポート
今年は聖地・秋葉原から!
The Student Day 2006
Yahoo! Widgets
かんたんプログラミング
さらに新連載が2本スタート!
通巻100号記念特別付録
よくわかる! まるごと
C言語学習ブック
C言語の入門から応用まで完全収録100ページ!


難易度 4
COVER STORY 4
Haskellによる
関数型プログラミング入門
酒井 政裕
Haskell

関数型言語ブーム？


入門Haskell
はじめて学ぶ関数型言語
向井 淳・著
毎日コミュニケーションズ


ふつうのHaskellプログラミング
ふつうのプログラマのための
関数型言語入門
山下伸夫 監修
青木峰郎 著
関数型言語を知る。
Haskellの考え方と
仕様を学び、
プログラマとしての
スキルアップを目指す。
SoftBank Creative

関数型言語ブーム？


BLOOD SWEAT and CODE!
Perl
Ruby
LL RING
"Blood Sweat & Code"
LIGHTWEIGHT LANGUAGE 2006
At
SHINKIBA
1st Ring
Aug 26,
2006
JavaScript
ActionScript
Squeak
FORTH
awk
zsh
bf
Gauche
Haskell
Python
PHP


LL RING
Home
Event
Blog
現在の場所: ホーム → Event
Event
ナビゲーション
ホーム
Event
LL Gong二次募集：審査結果発表
セッション内容
チケットの購入方法
じゃんけん2.0
Blog
Lightweight Language Ring 開催のお知らせ
◆開催概要
名称：Lightweight Language Ring（通称: LL Ring）
主催：LL Ring 実行委員会
協賛：
(株)アスキー(http://www.ascii.co.jp)
日本UNIXユーザ会(http://www.jus.or.jp/)
Kahuaプロジェクト(http://www.kahua.org/)
Tokyo Perl Mongers(http://tokyo.pm.org/)
日本PHPユーザ会(http://www.php.gr.jp/)
日本Pythonユーザ会(http://www.python.jp/)
日本Rubyの会(http://jp.rubyist.net/)
日本 GNU AWK ユーザー会(http://gauc.no-ip.org/)
(有)シングラム(http://www.syngram.co.jp/)
ほか
後援：
Franz Inc.(http://jp.franz.com/index.html)
日経SYSTEMS(http://itpro.nikkeibp.co.jp/SYS/)
開催日：2006年8月26日(土)
時間 ：10:30 (開演) ～21:00 (終了予定)
場所 ：新木場1stRING
(http://west-c.com/1string/)
参加費：3,500円
定員 ：300名
内容：軽量プログラミング言語に関する総合カンファレンスです。
各出版社による関連書籍の展示・販売も行う予定です。
参加申込: 2006年8月28日 (月) チケット発売開始
問い合わせ先：LL Ring 実行委員会

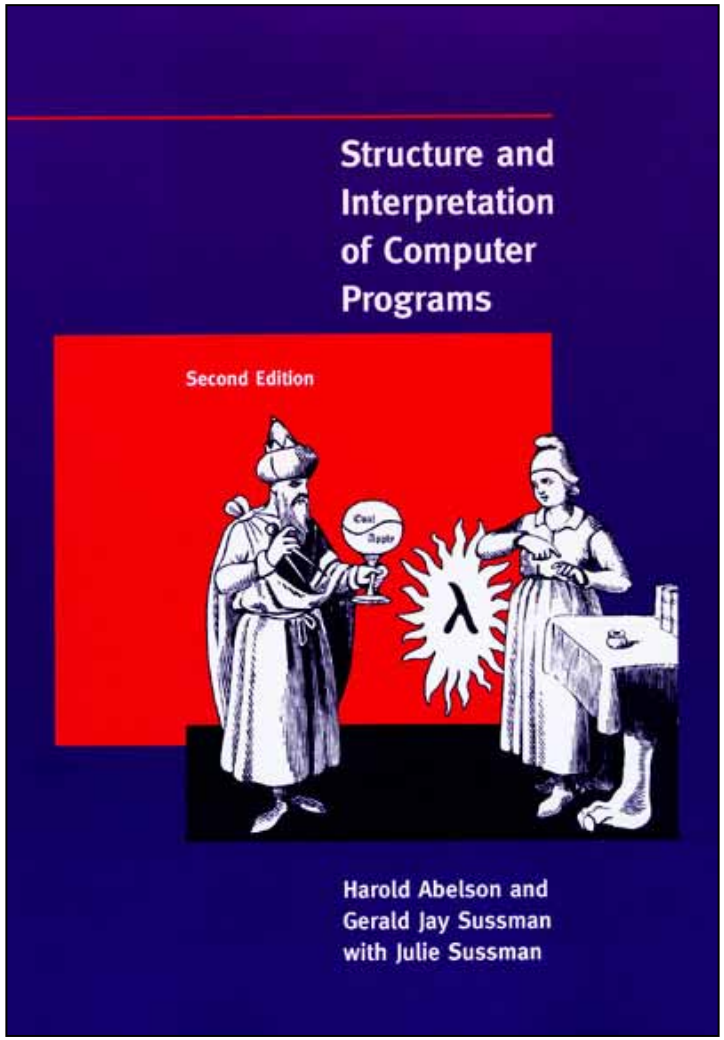
関数型言語ブーム？

	言語	参戦予定者	所属	備考
1	Scripting for Java Platform	櫻庭 祐一	横河電機	初参戦
2	FORTH	別府 政通	構造計画研究所	初参戦
3	awk	斉藤 博文	日本 GNU AWK ユーザー会	
4	ActionScript	上条 晃宏	アドビシステムズ社	初参戦
5	Scheme (Gauche)	川合 史朗	Scheme Arts	ハワイから生中継予定!
6	Python	柴田 淳	ウェブコア/日本Pythonユーザ会	
7	Squeak	梅澤 真史、南谷 千城	Squeak-ja	
8	Haskell	酒井 政裕	慶應義塾大学大学院	
9	OCaml	松本 宗太郎	筑波大学大学院	初参戦
10	PHP	廣川 類	日本PHPユーザ会	
11	Ruby	まつもとゆきひろ	ネットワーク応用通信研究所	
12	JavaScript	天野 仁史	ガイアックス /Shibuya.js	初参戦
13	Perl	竹迫 良範	サイボウズ・ラボ /Shibuya.pm	
14	Lisp	中村 正三郎	アンテナハウス本社グループマネージャ	

関数型言語ブーム？

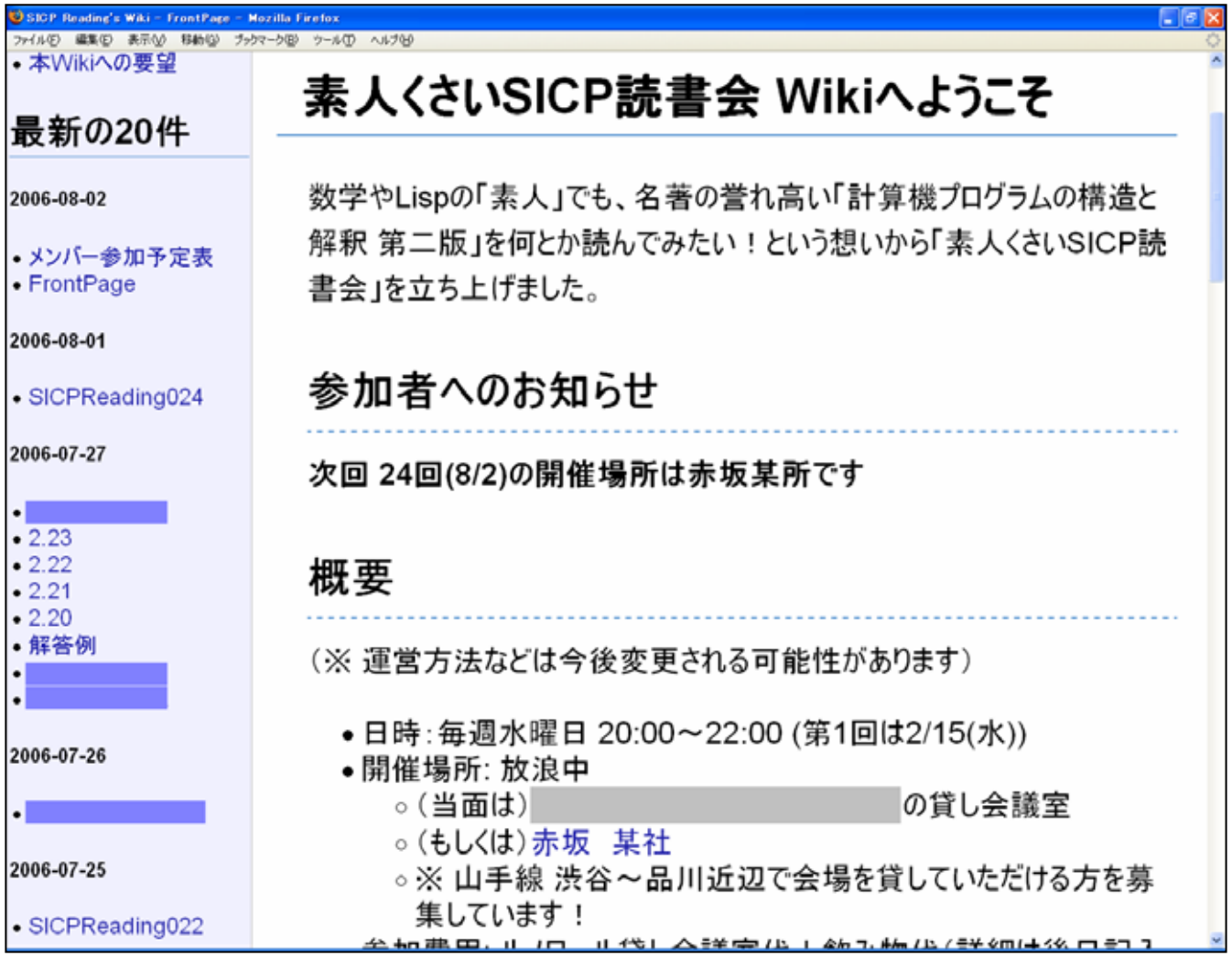

セッション内容 - Lightweight Language Ring - Mozilla Firefox
ファイル(F) 編集(E) 表示(V) 移動(G) ブックマーク(B) ツール(T) ヘルプ(H)
ナビゲーション
ホーム
Event
LL Gong 二次募集：審査結果発表
セッション内容
チケットの購入方法
じゃんけん2.0
セッション内容と参戦者の一覧
LLで関数プログラミング
No 参戦者 所属 備考
1 東北大学
2 青木峰郎 日本Rubyの会
3 久井亨 クロネッカーズデルタ
4 中村 正三郎 アンテナハウス
5 山下伸夫 タイムインターメディア
6 今泉貴史 千葉大学 司会
その場でどう書く
No フレームワーク（言語） 挑戦者 所属
1 Django(Python) 露木 誠 Django と日本の仲間たち http://www.djangoproject.jp/
2 Kahua(Gauche) 伊東 勝利 備前 達矢 タイムインターメディア /Kahuaプロジェクト
3 Ethna(PHP) 藤本 真樹 鶴岡直也 グリー サイボウズラボ
4 Ruby on Rails(Ruby) かずひこ secondlife 鈴木美保 ネットワーク応用通信研究所 はてな ツインスパーク

関数型言語ブーム？


Structure and
Interpretation
of Computer
Programs
Second Edition
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman


計算機プログラムの
構造と解釈
第二版
ピアソン

関数型言語ブーム？


SICP Reading's Wiki - FrontPage - Mozilla Firefox
本Wikiへの要望
最新の20件
2006-08-02
メンバー参加予定表
FrontPage
2006-08-01
SICPReading024
2006-07-27
2.23
2.22
2.21
2.20
解答例
2006-07-26
2006-07-25
SICPReading022
素人くさいSICP読書会 Wikiへようこそ
数学やLispの「素人」でも、名著の誉れ高い「計算機プログラムの構造と解釈 第二版」を何とか読んでみたい！という想いから「素人くさいSICP読書会」を立ち上げました。
参加者へのお知らせ
次回 24回(8/2)の開催場所は赤坂某所です
概要
（※ 運営方法などは今後変更される可能性があります）
日時：毎週水曜日 20:00～22:00 (第1回は2/15(水))
開催場所: 放浪中
（当面は）の貸し会議室
（もしくは）赤坂　某社
※ 山手線 渋谷～品川近辺で会場を貸していただける方を募集しています！

このチュートリアルの目標

- 非常に単純な関数型言語「MinCaml」のコンパイラを、できるだけ詳細に解説する
- 「関数型言語は難しい」
  「関数型言語は仕組みがよくわからない」
  といった神話(myth)を打破する

⇓

関数型言語ブーム？を
「ブーム」で終わらせない
(ことを目指す)

関数型言語とは何か？

- 関数型プログラミング：
  破壊的代入などの副作用を、できるだけ使用しないプログラミング・スタイル
- 関数型言語：
  関数型プログラミングを推奨・支援するプログラミング言語

関数型言語の分類

- 純粋 vs. 純粋でない
  - 副作用がない（隔離されている）か、
    副作用がある（混在している）か
- 正格評価 vs. 遅延評価
  - 式の値がすぐに計算されるか、
    必要となるまで計算されないか
- 静的型 vs. 動的型
  - 型チェックがコンパイル時に行われるか、
    実行中に行われるか

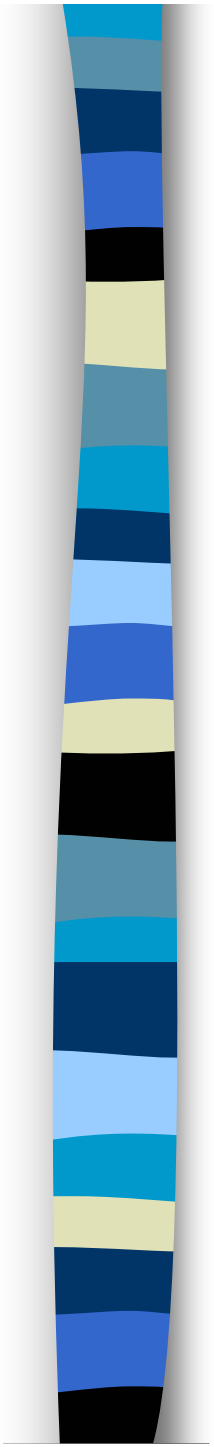
代表的な関数型言語

- Haskell (www.haskell.org)
  - **純粋、遅延評価、静的型**
- ML (www.standardml.org, caml.inria.fr)
  - Standard ML (SML), Objective Caml (OCaml)
  - **純粋でない、正格評価、静的型**
- Scheme (www.schemers.org)
  - **純粋でない、正格評価、動的型**

コンパイラとは何か？

- **ある言語のプログラムを、**
  よりlow levelな言語へ翻訳するシステム
  - GCC（C**など → アセンブリ**）
    - GCJ（Java → **アセンブリ**）
  - javac（Java → JVM）
  - ocamlc（OCaml → OCaml VM）
  - ocamlopt（OCaml → **アセンブリ**）
  - MLj（SML → JVM）

  etc.

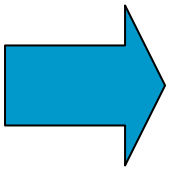


例 (MinCaml)

```
let rec gcd m n =
  if m = 0 then
    n
  else if m <= n then
    gcd m (n - m)
  else
    gcd n (m - n)
```

例 (MinCaml → SPARC)

```
let rec gcd m n =
  if m = 0 then
    n
  else if m <= n then
    gcd m (n - m)
  else
    gcd n (m - n)
```



```
gcd.7:
        cmp     %i2, 0
        bne     be_else.17
        nop
        mov     %i3, %i2
        retl
        nop
be_else.17:
        cmp     %i2, %i3
        bg      ble_else.18
        nop
        sub     %i3, %i2, %i3
        b       gcd.7
        nop
ble_else.18:
        sub     %i2, %i3, %i2
        mov     %i3, %o4
        mov     %i2, %i3
        mov     %o4, %i2
        b       gcd.7
        nop
```

この大きなギャップを どうやって埋めるのか？

- **適切な中間言語を設定すれば、ほとんど自明な記号変換の連続**

```
(* MinCamlのmain.mlの中核部分 *)
Emit.f outchan
(RegAlloc.f
 (Simm13.f
  (Virtual.f
   (Closure.f
    (iter !limit
     (Alpha.f
      (KNormal.f
       (Typing.f
        (Parser.exp
          Lexer.token l)))))))))
```

```
> wc --lines *.ml*
  46 alpha.ml
   2 alpha.mli
  18 assoc.ml
   1 assoc.mli
  38 beta.ml
   1 beta.mli
 106 closure.ml
  33 closure.mli
  50 constFold.ml
 (中略)
2020 total
```

我々の対象言語MinCaml

- 純粋でない、正格評価、静的型
  (ただし多相型はない)
- 文法はOCamlのサブセット
- 機能はminimal
  (MLともいえないぐらい)

MinCamlの抽象構文 (1/2)

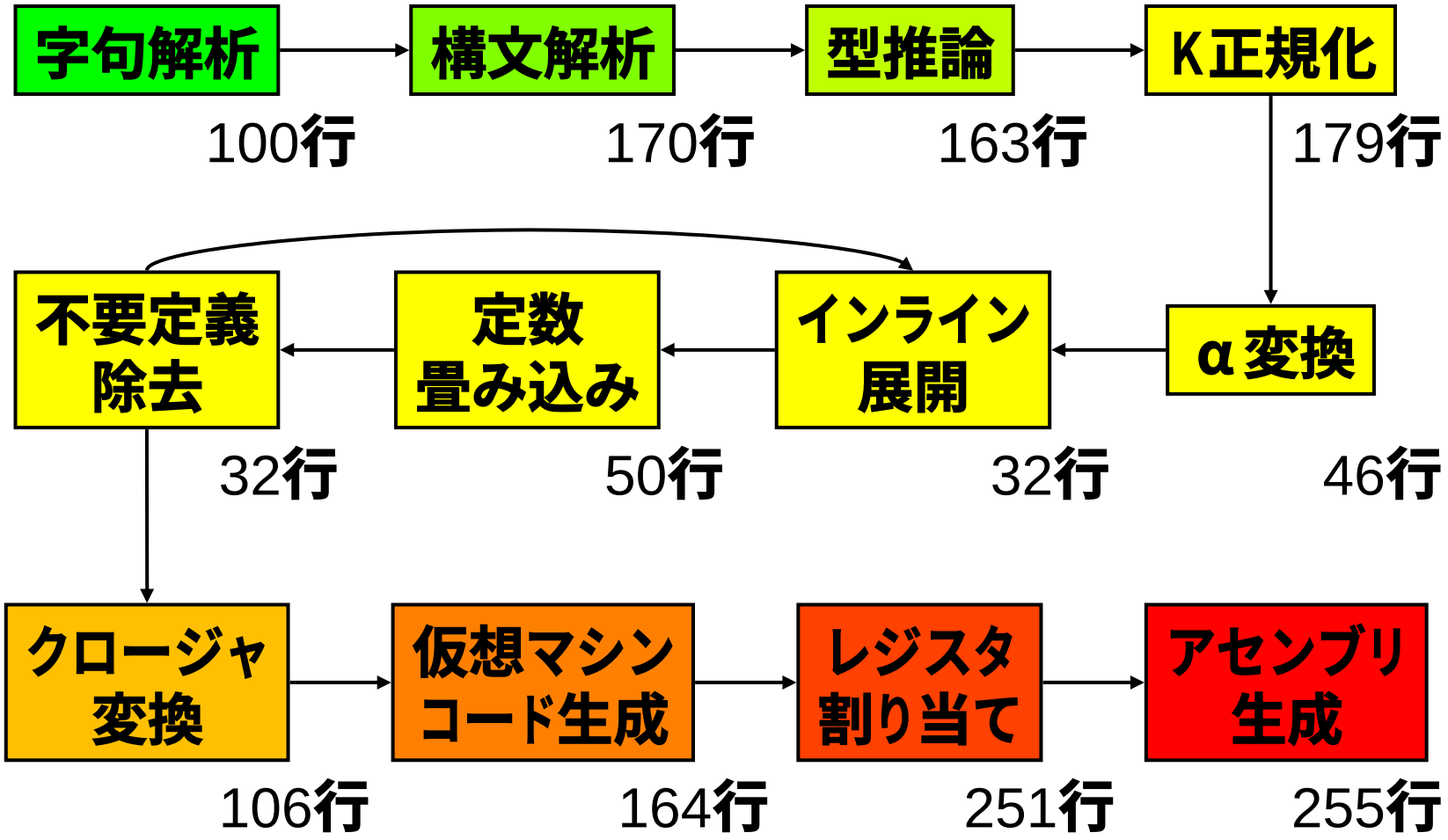
M, N ::=	
c	定数
$op(M_1,...,M_n)$	算術演算
if M then N_1 else N_2	条件分岐
let $x = M$ in N	変数定義
x	変数読み出し
let rec $x\ y_1 \ldots y_n = M$ in N	関数定義
$M\ N_1 \ldots N_n$	関数呼び出し
...	

MinCamlの抽象構文 (2/2)

M, N ::=

...	
$(M_1,\ldots,M_n)$	**組を作る**
let $(x_1,\ldots,x_n) = M$ in N	**組から読み出し**
Array.create M N	**配列を作る**
$M.(N)$	**配列から読み出し**
$M_1.(M_2) \leftarrow M_3$	**配列へ書き込み**

コンパイラの構成


字句解析
100行
構文解析
170行
型推論
163行
K正規化
179行
α変換
46行
インライン展開
32行
定数畳み込み
50行
不要定義除去
32行
クロージャ変換
106行
仮想マシンコード生成
164行
レジスタ割り当て
251行
アセンブリ生成
255行

字句解析・構文解析

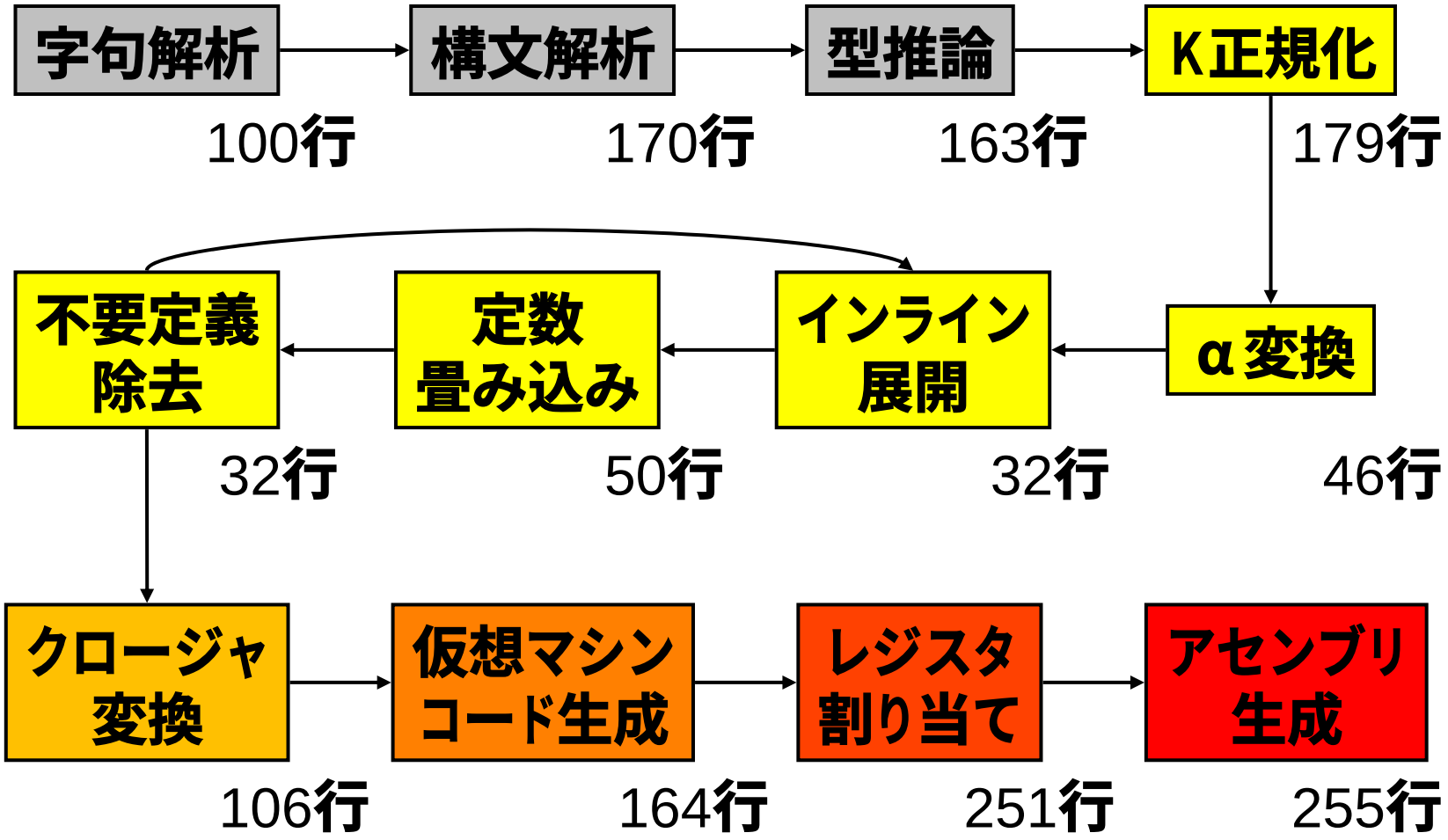
- OCamlLex/OCamlYaccを使っているだけ
  - よって、詳しいアルゴリズムは略

    Cf. packrat parsing
    (http://pdos.csail.mit.edu/~baford/packrat/)
  - 唯一のtrickyな部分：マイナス記号
    - 「x-y」を「x(-y)」とparseされないように、
      関数の引数となりうる式simple_expと、
      それ以外の式expをわける

型推論

- 標準的な単一化を破壊的代入で実装
  (型変数をOCamlの参照セルで表現)
  - 型システムのチュートリアルではないので、これも略
  - 唯一のpeculiarな部分：外部変数
    - 定義されていない変数がプログラムの中に現れたら、外部関数または外部配列とみなす
    - 多相型がないので、具体化されなかった型変数はintとしてしまう

コンパイラの構成


字句解析
100行
構文解析
170行
型推論
163行
K正規化
179行
α変換
46行
インライン展開
32行
定数畳み込み
50行
不要定義除去
32行
クロージャ変換
106行
仮想マシンコード生成
164行
レジスタ割り当て
251行
アセンブリ生成
255行

復習

コンパイルとは:

- 高レベル言語から低レベル言語への翻訳
  - ここではMinCaml ⇒ SPARCアセンブリ
- うまい中間言語を設定すれば、
  ただの記号変換の連続

K正規形

- ML Kit (http://www.it-c.dk/research/mlkit/)
  というコンパイラの中間言語
- **中間式もすべて明示的に変数へ束縛する**
  ⇒ MinCamlをSPARCアセンブリに少し近づける
  例: `(x-y)-(y-z)`

```
      → let tmp1 = x - y in
        let tmp2 = y - z in
        tmp1 - tmp2
```

- **一般的な構文定義と変換方法は図**4, 5

- 簡単な最適化：最初から変数になっている部分式は、そのままにしておく

例：`12345-x` は

```
let tmp1 = 12345 in
let tmp2 = x in
tmp1 - tmp2
```

ではなく、単に

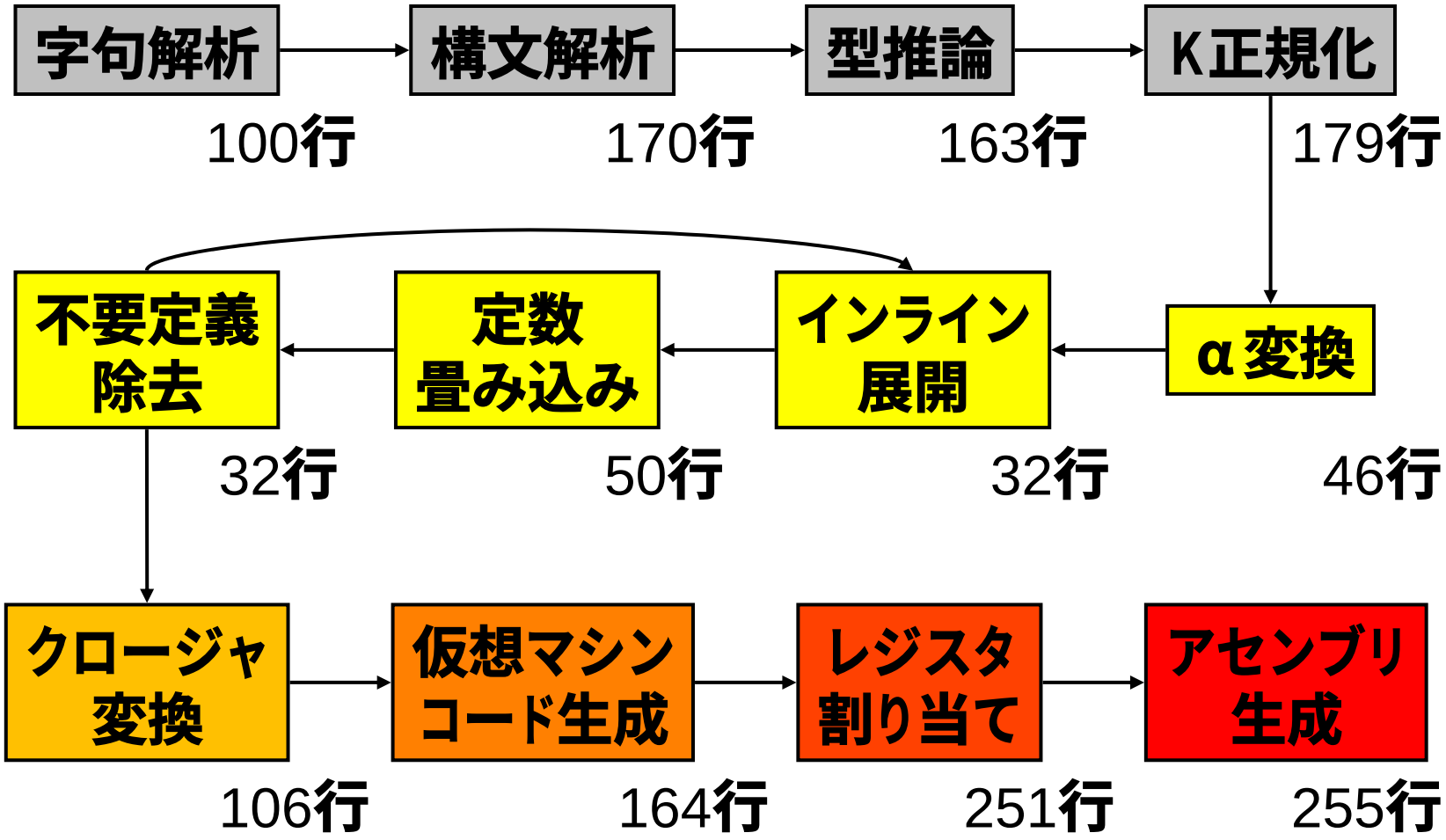
```
let tmp1 = 12345 in tmp1 - x
```

とする

補助関数を利用すれば、簡単に実装できる

```
let insert_let e k =
  match e with Var(x) -> k x
  | _ -> let x = gensym () in Let(x, e, k x)
```

コンパイラの構成


字句解析
構文解析
型推論
K正規化
100行
170行
163行
179行
不要定義除去
定数畳み込み
インライン展開
α変換
32行
50行
32行
46行
クロージャ変換
仮想マシンコード生成
レジスタ割り当て
アセンブリ生成
106行
164行
251行
255行

α変換

- 束縛変数の名前を"fresh"なものにつけかえる

  ⇒ 意味的に異なる変数には異なる名前を与え、以降の処理を容易にする

  例: `let x = 123 - 456 in let x = x - 789 in -x`

  ↓

  `let` x_1 `= 123 - 456 in let` x_2 `=` x_1 `- 789 in -`x_2

- 一般的な変換方法は図6

インライン化

- **関数の呼び出しを、その関数の本体で置き換える**

  let rec f x y z = M in ... (f a b c) ... →
  let rec f x y z = M in ... ([a,b,c/x,y,z]M') ...

  – **インライン化する項は、直前にα変換する**
  – **Mの中に現れるfの呼び出しも置き換えてよい**
  – **ただし、むやみにやると...**
    - **コードサイズが爆発する**
    - **コンパイルが停止しない**

      ⇒ 何らかのheuristicsを用いる

定数畳み込み

- 演算のオペランドが「明らかに」定数だったら、コンパイラが計算してしまう

  let x = 7 in let y = 3 in x – y

  → let x = 7 in let y = 3 in 4

- インライン化の後に行うと吉

不要な束縛の除去

- **副作用がなく、変数も使用されないようなlet (およびlet rec) を省略する**

  let x = M in N → N

  ただし

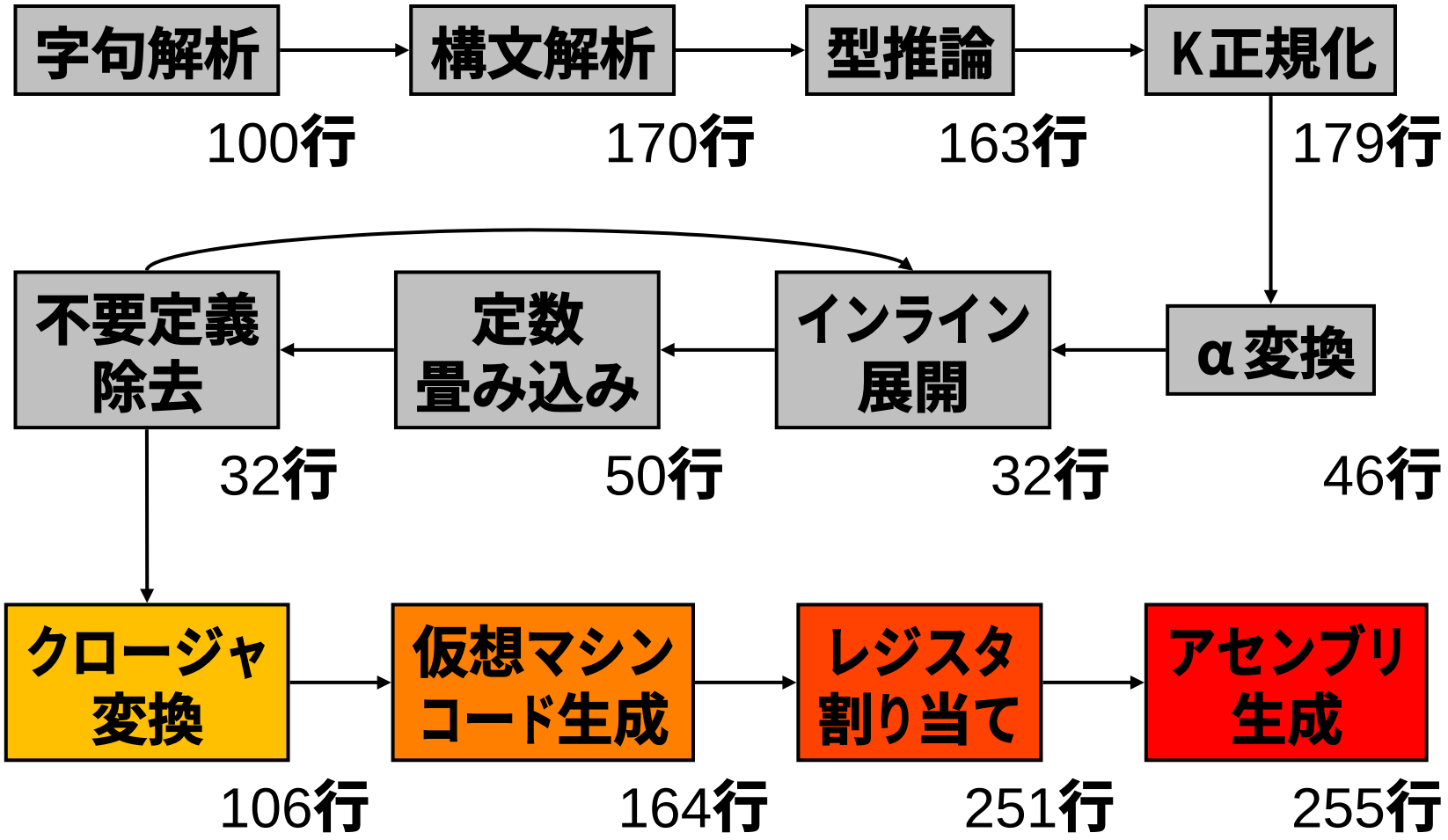
  – x**はN**に現れない

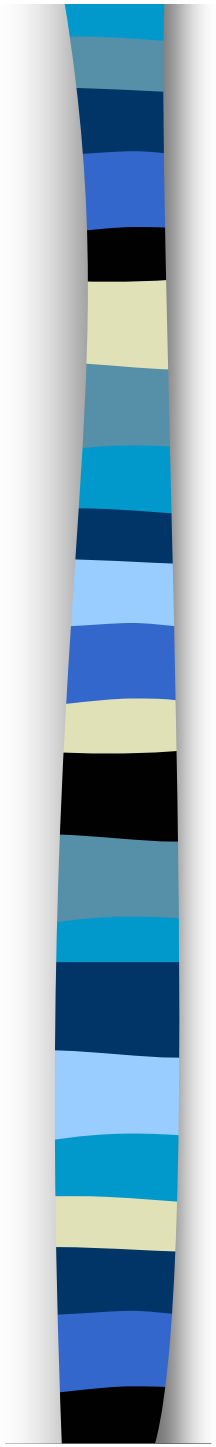
  – M**は副作用を持たない**

    - **「配列への書き込みと、関数適用を含まない」で近似するのが簡単**

- **インライン化・定数畳み込みの後に行うと吉**

コンパイラの構成


字句解析
構文解析
型推論
K正規化
100行
170行
163行
179行
不要定義除去
定数畳み込み
インライン展開
α変換
32行
50行
32行
46行
クロージャ変換
仮想マシンコード生成
レジスタ割り当て
アセンブリ生成
106行
164行
251行
255行



Closure変換

ネストした関数定義を平らにする

⇒ K正規形をSPARCアセンブリへ
さらに近づける

このチュートリアルの
最大の山場（かもしれない）

例1: 容易に平らになる場合

(OCamlの文法で)

```
let quad x =
    let dbl y = y + y in
    dbl (dbl x)
in quad 3
```

↓

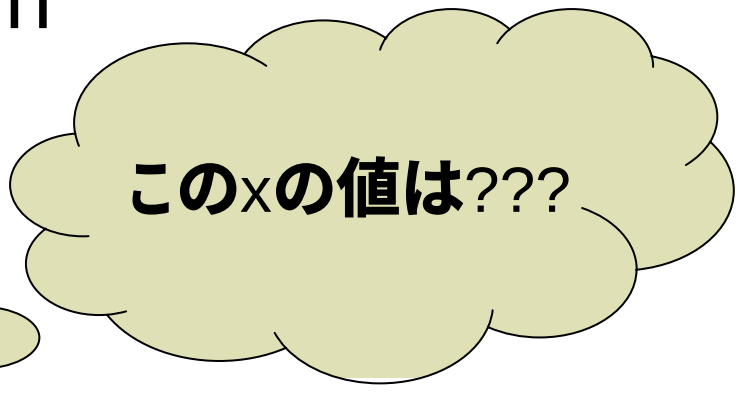
```
let dbl y = y + y ;;
let quad x = dbl (dbl x) ;;
quad 3
```

例2:
容易には平らにならない場合

```
let make_adder x =
    let adder y = x + y in
    adder
in (make_adder 3) 7
```

↓ ???

```
let adder y = x + y ;;
let make_adder x = adder ;;
(make_adder 3) 7
```



何が悪かったのか?

adderの本体が
xのスコープを逸脱してしまった
⇒ 関数の外側で定義されている
変数(自由変数)が問題

解決策:

a. **自由変数を持つ関数を許さない**
   - C, C++**などのほとんどの処理系**

b. **動的にコードを生成する**!
   - GCC

c. **関数のアドレスと一緒に、**
   自由変数の値も保存しておく
   - Scheme, ML, Java (inner class)**などの**
     ほとんどの処理系

Closure変換

- **関数を「アドレスと自由変数の値の組」(closure)で表現する**
  - **関数を作るとき**: closureを作って、**自由変数の値をしまっておく**

```
let make_adder x = (adder, x) ;;
```

  - **関数を呼び出すとき**: **自由変数の値を取り出して、引数に追加して渡す**

```
let (body, fv) = make_adder 3 in
  body 7 fv
```

  - **呼び出される関数**: **自由変数の値を、追加の引数として受け取る**

```
let adder y x = x + y ;;
```

ふたたび例2

```
let make_adder x =
    let adder y = x + y in
    adder
in (make_adder 3) 7
```

↓

```
let adder y x = x + y ;;
let make_adder x = (adder, x) ;;
apply_closure(make_adder 3, 7)
```

ただし

apply_closure((body, fv), arg) ≡ body arg fv

Closure変換の方法

- プリント図13,14
- let recの場合が問題
  1. =の右辺をclosure変換する
  2. 自由変数を計算する
  3. Closure変換後の関数定義を、
     グローバル変数に追加する
  4. Closureを作る(ためのコードを返す)

「賢くない」Closure変換の問題点

自由変数のない関数でも、
make_closureとapply_closureの
オーバーヘッドがかかる

例:

let f x = x + 3 in f 7

⇓

let L_f x () = x + 3 ;;
make_closure f = (L_f, ()) in
apply_closure f 7

(構文は適当です)

解決策

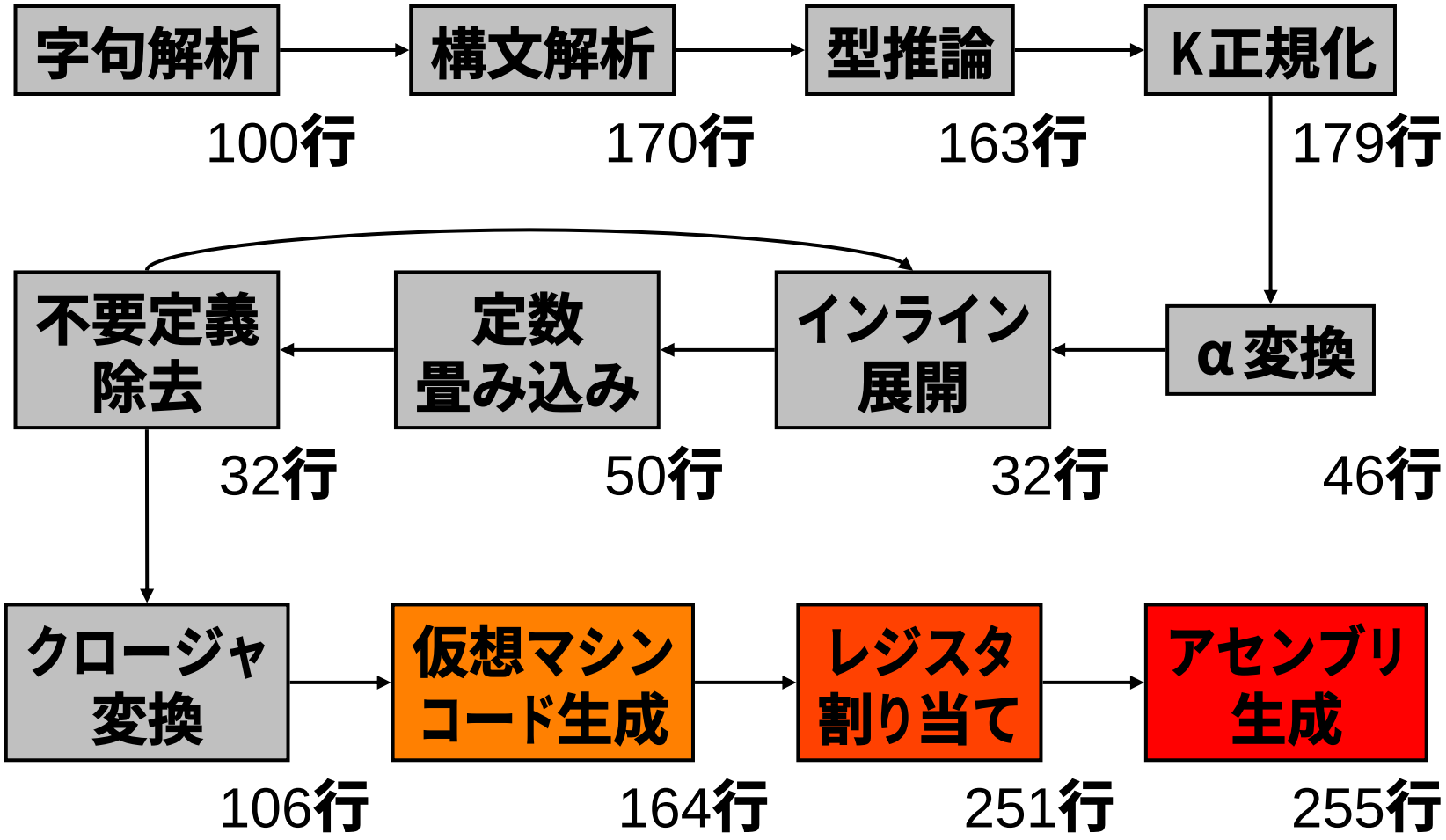
1. 「自由変数がない」とわかる関数は、closureを使わないでダイレクトに呼び出す
2. 1.により要らなくなったclosureは作らない

注：自由変数があるかないか呼び出す側でわからないときは、自由変数がなくてもclosureを作らねばならない

```
let x = 1 in
let f y = x + y in (* 自由変数あり *)
let g z = z + 2 in (* 自由変数なし *)
(if ... then f else g) 3 (* どちらだかわからない *)
```

⇒ fだけでなく、gについてもclosureを作る

コンパイラの構成


字句解析
100行
構文解析
170行
型推論
163行
K正規化
179行
α変換
46行
インライン展開
32行
定数畳み込み
50行
不要定義除去
32行
クロージャ変換
106行
仮想マシンコード生成
164行
レジスタ割り当て
251行
アセンブリ生成
255行

仮想マシンコード生成

- メモリ操作を明示化する
  - 組の生成と読み出し
  - 配列の生成と読み書き
  - クロージャの生成
  - 関数の先頭における、
    クロージャからの自由変数の読み出し
- 図17,18

レジスタ割り当て

変数への破壊的代入がないので、
(生死の計算が)
命令型言語より簡単！

- **前から順に**greedy**に割り当てるだけ**
  - **ただし、関数呼び出し等の際の**
    mov**命令が減るように先読みする**
- **(**MinCaml**の中での) 関数呼び出し規約は**
  あらかじめ決めておく

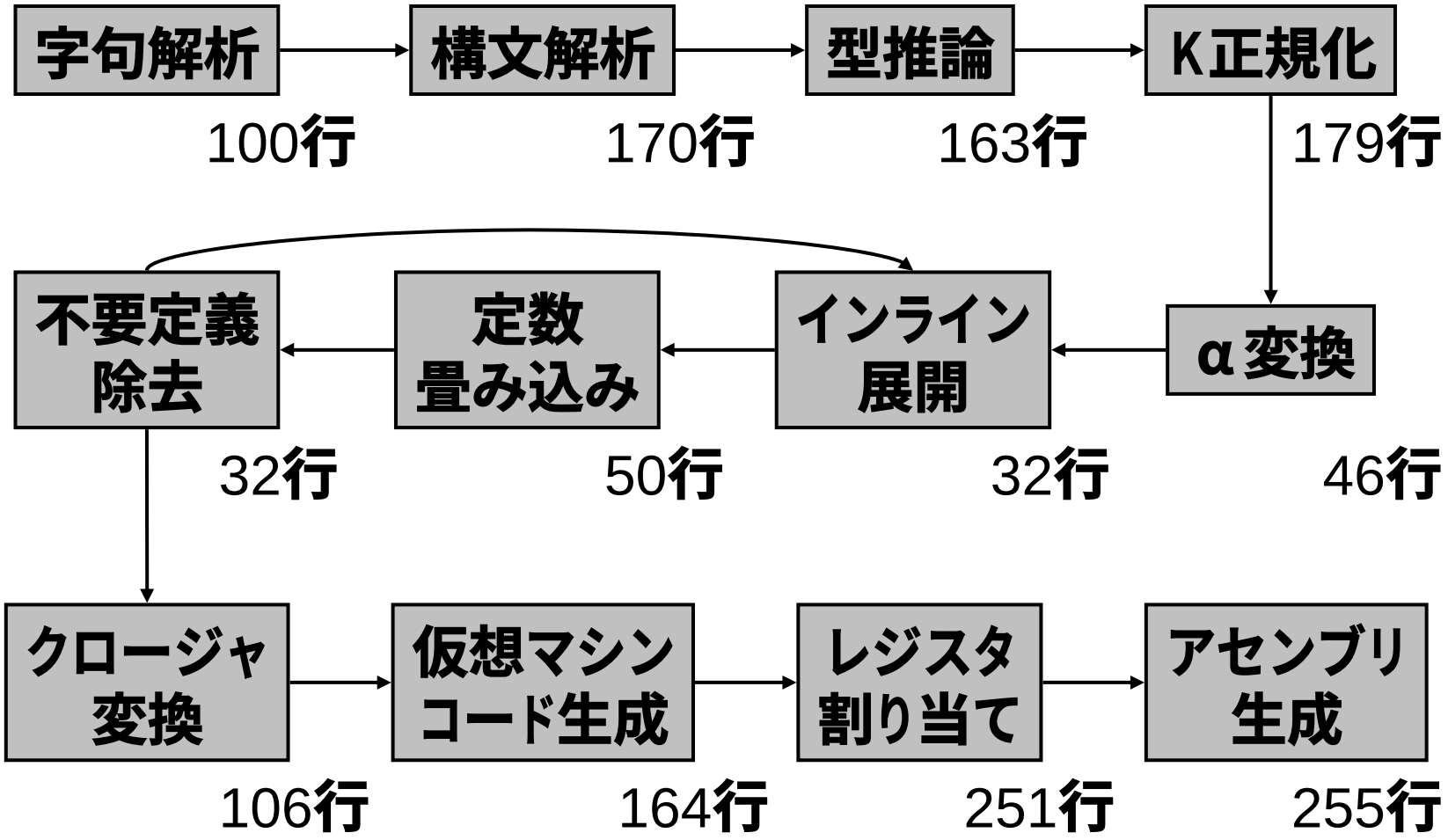
アセンブリ生成

- レジスタ割り当てされたマシンコードを pretty printするだけ

ただし、

- 関数呼び出しにおけるレジスタ並べ替え (register shuffling)は、ここで行っている
- 末尾呼び出しは、call命令ではなく ただのジャンプ命令とする

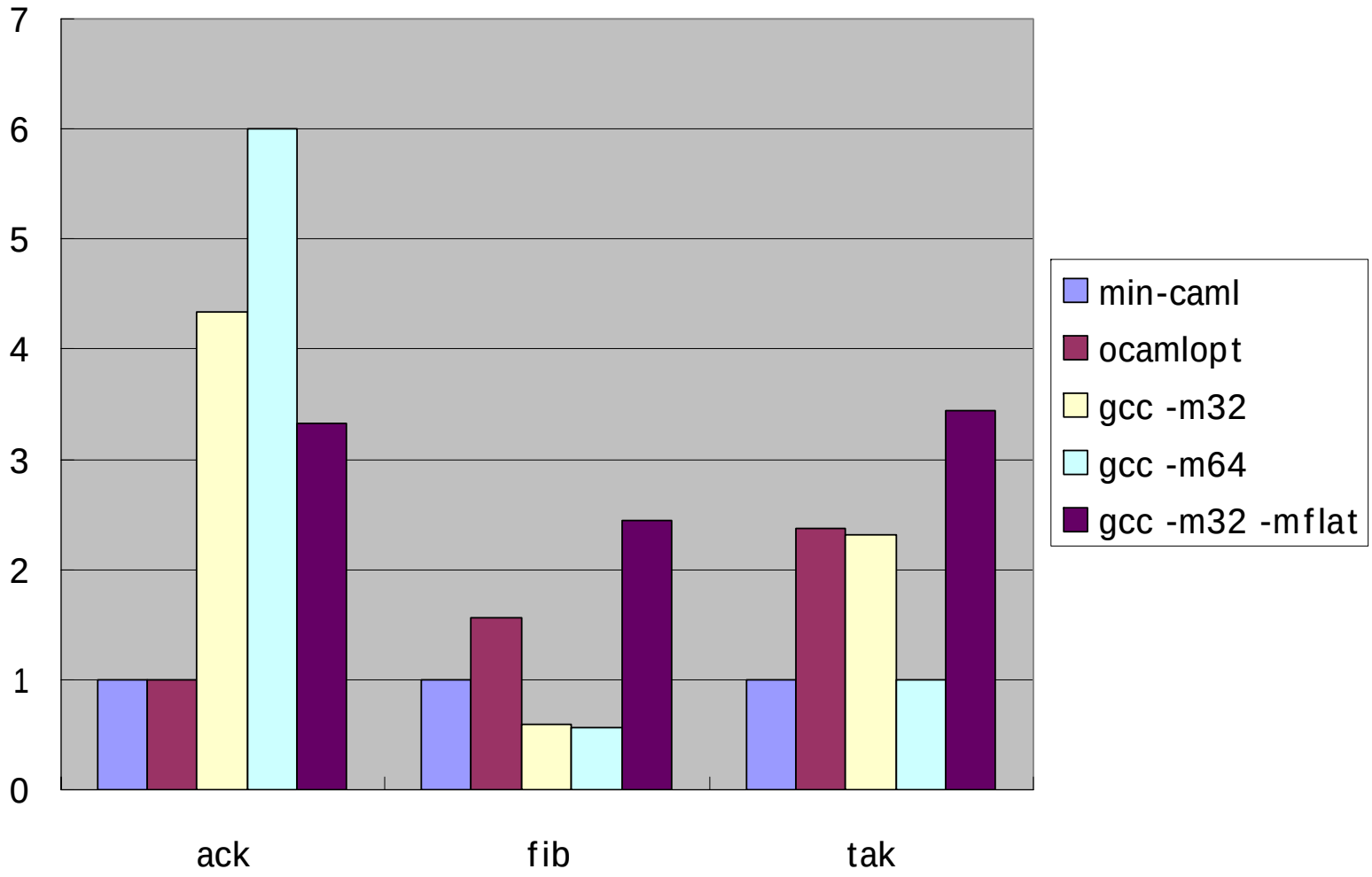
コンパイラの構成


字句解析
100行
構文解析
170行
型推論
163行
K正規化
179行
α変換
46行
インライン展開
32行
定数畳み込み
50行
不要定義除去
32行
クロージャ変換
106行
仮想マシンコード生成
164行
レジスタ割り当て
251行
アセンブリ生成
255行

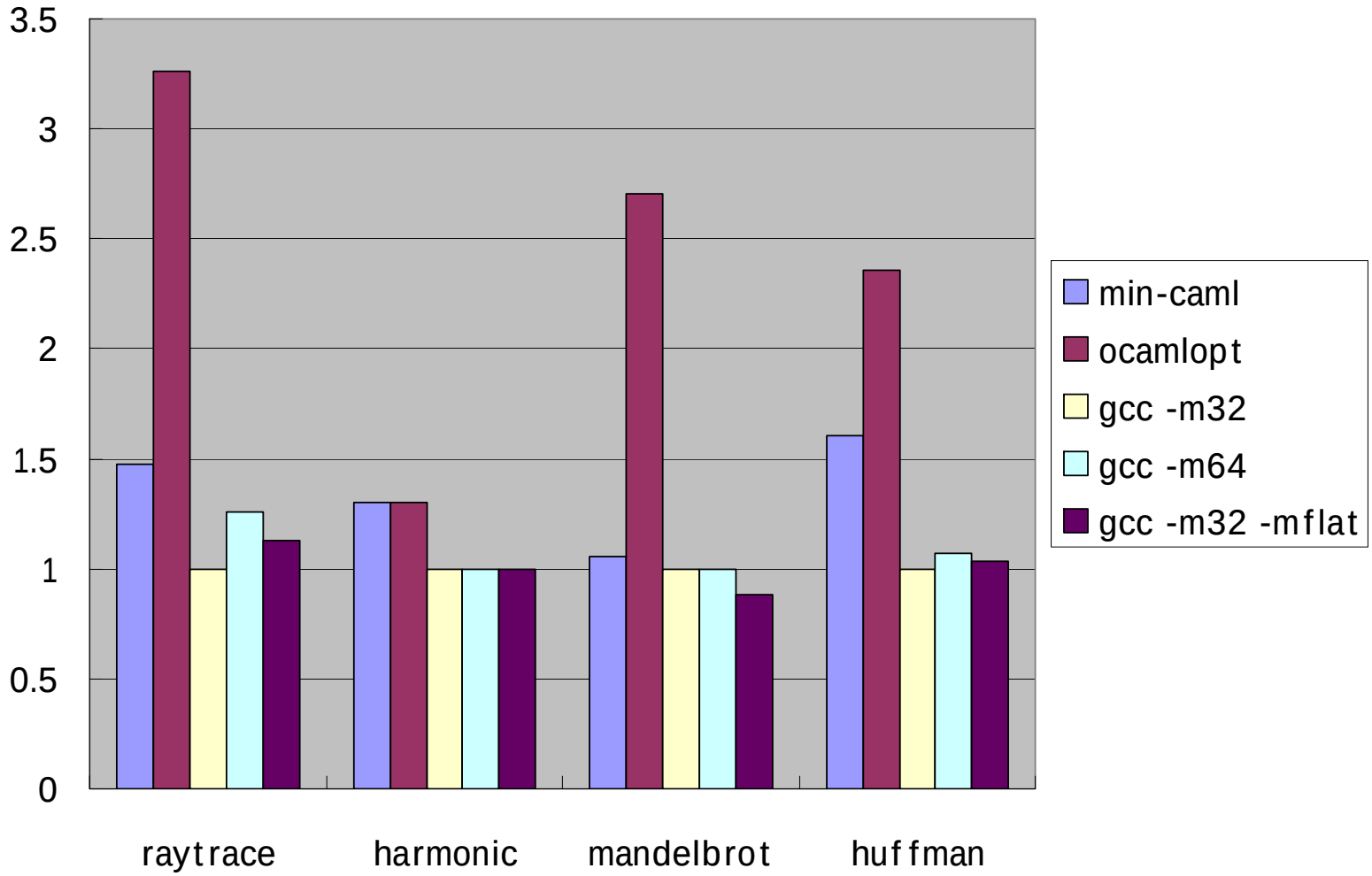
性能測定：環境

- **マシン**: Sun Fire V880
  - Ultra SPARC III 1.2GHz
  - 8 GB **メインメモリ**
  - Solaris 9
- **コンパイラ**:
  - MinCaml (32 bit, -iter 1000 -inline 100)
  - OCamlOpt 3.08.3 (32 bit, -unsafe -inline 100)
  - GCC 4.0.0 20050319 (32 bit & 64 bit, -O3)
  - GCC 3.4.3 (32 bit "flat", -O3)

関数型言語的アプリケーション



命令型言語的アプリケーション



さらに勉強したい場合は...