コンパイラ構成

イントロダクション

情報工学系
権藤克彦

なぜコンパイラを学ぶのか？

- コンピュータの動作原理の一部として学ぶ
  - 中身が分からず，APIを使うだけではダメ
- ソフトウェアの作り方が成熟した成功事例として学ぶ
  - 字句解析，構文解析，意味解析の分離
  - 閉包という考え方（関連するものを全部集める演算）
    - 閉包 closure
    - 関数型言語のクロージャとは別概念
- コンパイラや言語処理系作りは面白いから
  - 学部生でもマイ言語処理系を作成可能
    （特に効率を気にしなければ）

参考図書（コンパイラ）

- A.V. Aho他「コンパイラ―原理・技法・ツール」，サイエンス社，第2版， ISBN 478191229X，2009
- 湯浅太一「コンパイラ」，オーム社，ISBN 4274216209，2014
- 佐々政孝「プログラミング言語処理系」，岩波書店，ISBN 4-00-010345-8，1989
- 中田育男「コンパイラの構成と最適化」，朝倉書店，第2版，ISBN 4254121776，2009
- A.W. Appel「最新コンパイラ構成技法」，翔泳社， ISBN 4798114685，2009
- Terence Parr「言語実装パターン」，オライリージャパン，ISBN 4873115329，2011
- 青木峰郎「ふつうのコンパイラをつくろう」，ソフトバンククリエイティブ，ISBN 4797337958，2009

参考図書（C言語）

- カーニハン，リッチー「プログラミング言語C 第2版 ANSI規格準拠」ISBN:4320026926，1989
- ハービソン他「Cリファレンスマニュアル第5版」 ISBN: 4434124234，2008

コンパイラとアセンブラ（1/2）

- コンパイラはC言語をアセンブリ言語に変換する.
  - これを「コンパイルする」という.
  - 実行例：`gcc -S foo.c` で，`foo.s` ができる.
- アセンブラはアセンブリ言語を機械語に変換する.
  - これを「アセンブルする」という.
  - 実行例：`gcc -c foo.s` で，`foo.o` ができる.

コンパイラとアセンブラ（2/2）

add5.c

```
long add5 (long n)
{
   return n + 5;
}
```

テキスト

% gcc –S add5.c

コンパイル

add5.s

テキスト

```
    .text
    .globl _add5
    .p2align 4, 0x90
_add5:
    pushq   %rbp
    movq    %rsp, %rbp
    addq    $5,   %rdi
    movq    %rdi, %rax
    popq    %rbp
    retq
```

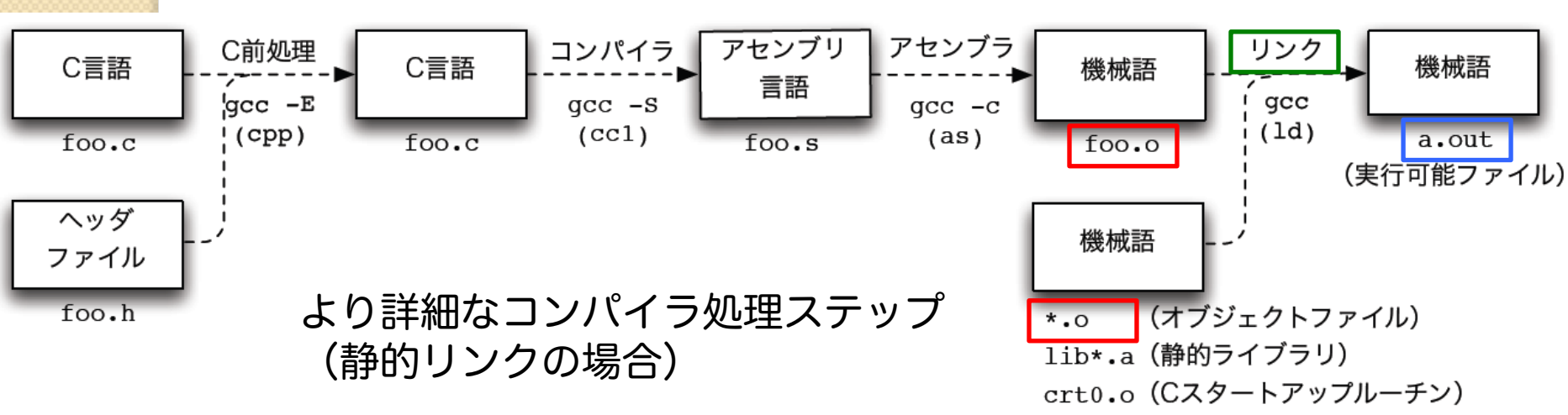
.cfi_... は無視でOK（ここでは削除）
call frame information

Macでは，.text の部分が以下となっている（効果は同じ）

```
.section    __TEXT,__text,regular,pure_instructions
```

広義のコンパイル

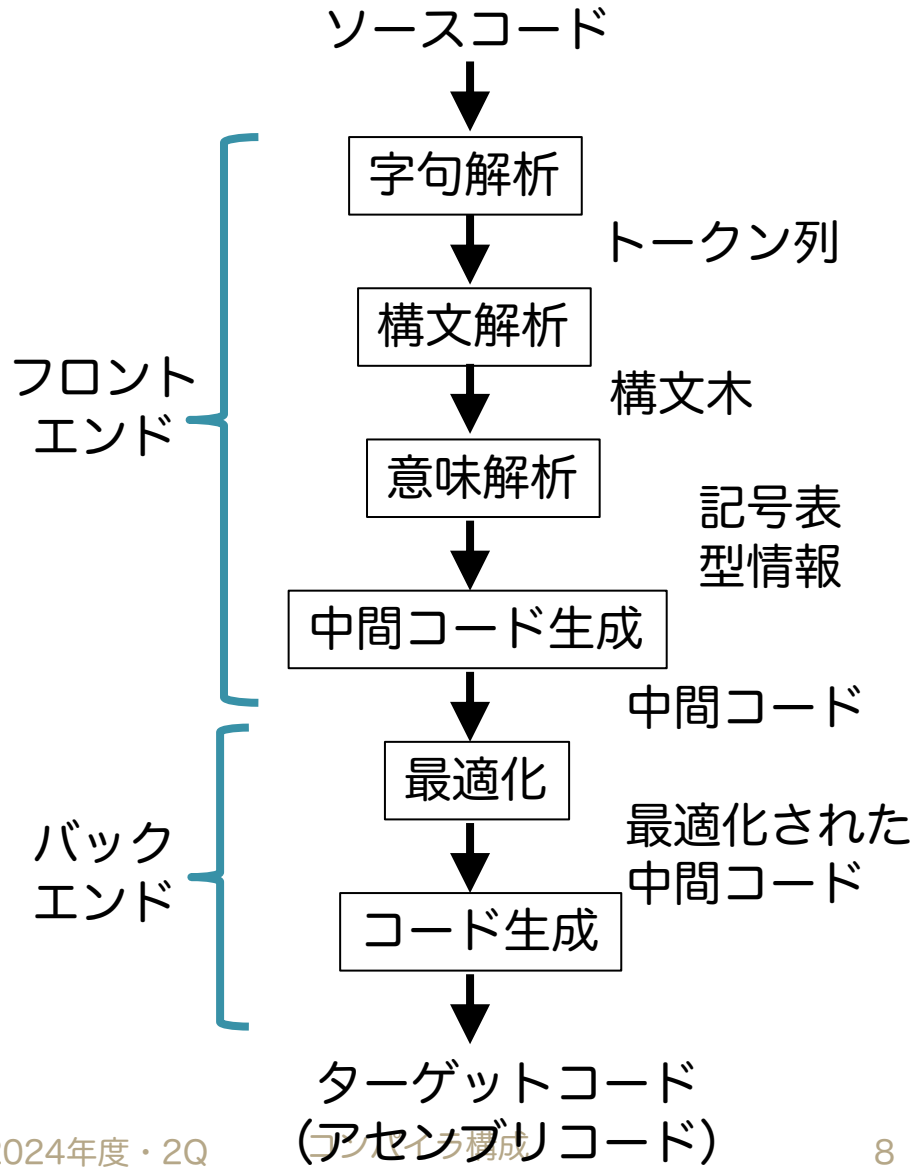
- （狭義の）コンパイル，アセンブル，リンクを含む



より詳細なコンパイラ処理ステップ
（静的リンクの場合）

狭義のコンパイラの実行フェーズ

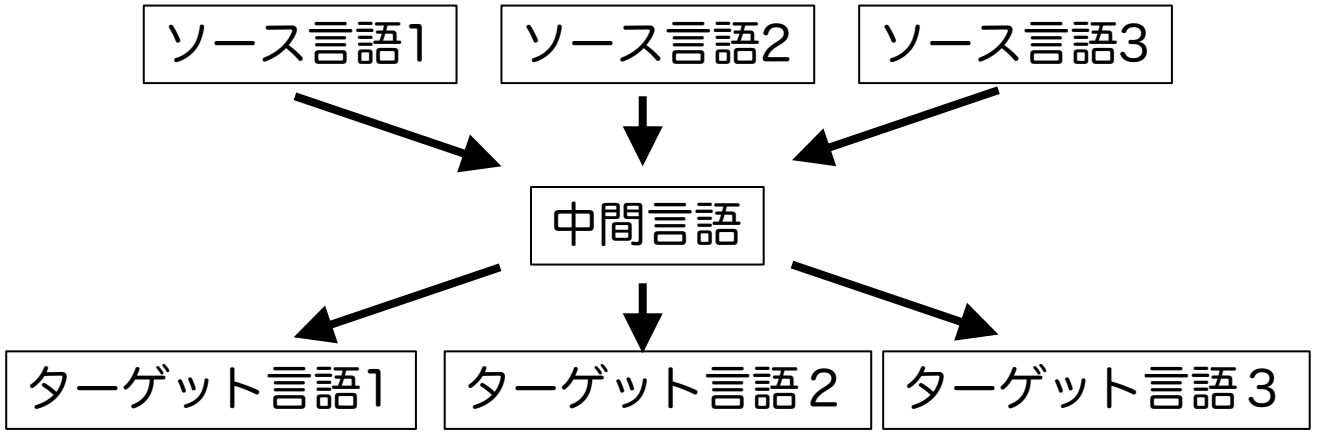
- フロントエンド
  - ソースコードに依存
  - 中間コード生成まで
- バックエンド
  - ターゲットコードに依存



厳密な境界があるわけでは無い

中間コード intermediate code

- 中間コード
  - ≒仮想機械の命令コード
  - 機械語コードに近いもの（例：LLVM IR，GNU RTL）や，構文木+α （例：GNU Generic）
- 中間コードは開発を容易にする
  - コンパイラは数多くの言語，CPU，OSに要対応
  - 共通の中間コードがあれば，コード削減が可能



プログラムの実行方式

現在では境界線が曖昧

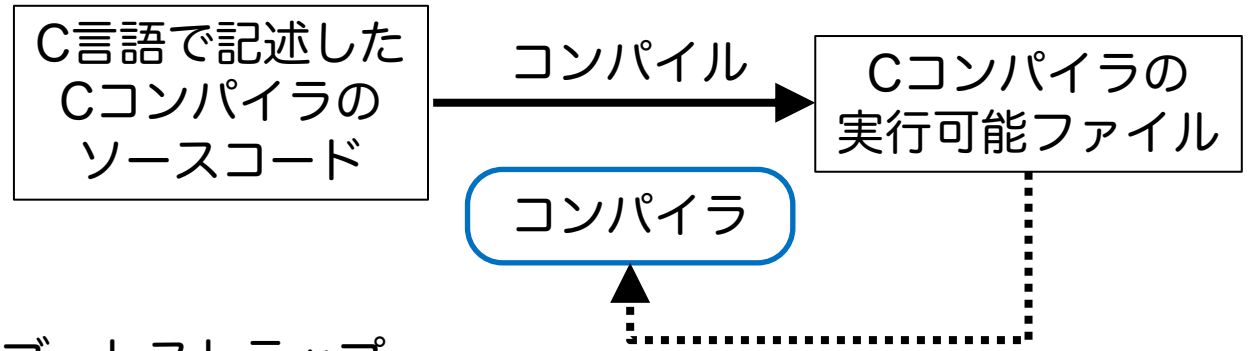
- コンパイラ方式
  - 事前にプログラムを機械語に変換しておき実行
  - 比喩：本の翻訳
- インタプリタ方式
  - 機械語に変換せず，（機械語で書かれた）インタプリタがプログラムの構文木や中間コードを実行
  - 比喩：同時通訳
- コンパイラ・インタプリタ方式
  - 中間コードにコンパイルし，それをインタプリタで実行
    - 例：Javaはバイトコードにコンパイルし，
    - Java仮想機械（＝バイトコードのインタプリタ）で実行
- 実行時コンパイル（JIT）方式
  - 実行時に中間コードを機械語に変換して実行

just-in-time
実行直前にコンパイル

コンパイラのブートストラップ

ブーツ紐

- Cコンパイラは C言語で記述可能
  - コンパイラもプログラムの一種（例：GCCはC言語で記述）
- コンパイラのブートストラップ
  - コンパイラのソースコードをそのコンパイラ自身で コンパイルすること ← どうやって？
  - 最初は人間が手作業でコンパイル（hand compiling）
  - 一度出来れば，コンパイラプログラムをコンパイル可能



cf. OSのブートストラップ
OSは誰がメモリにロードするのか

チョムスキー階層 Chomsky Hierarchy

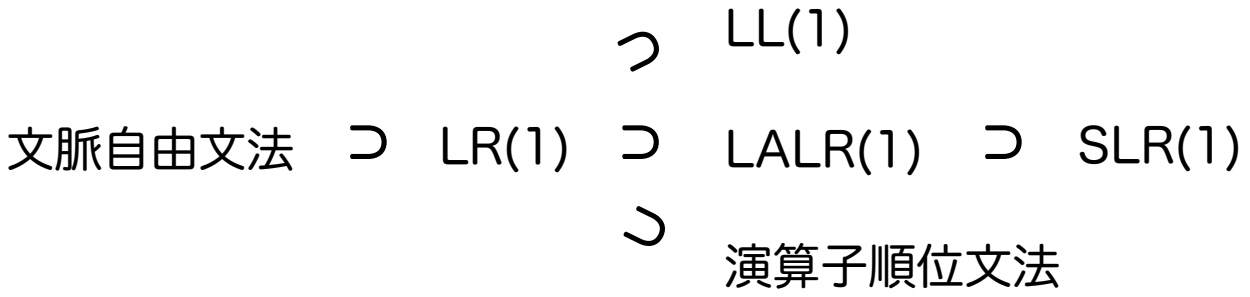
- 形式文法の包含関係
  - 形式文法＝書き替え規則（生成規則）で定義する文法
- 上の方が生成する言語がより広い
- 文脈自由文法と正規文法が重要
  - 文脈自由文法 (CFG)：構文解析に使う
  - 正規文法：字句解析に使う

文法	認識するオートマトン	生成規則
句構造文法	チューリングマシン	$\alpha \rightarrow \beta$ （制限無し）
文脈依存文法	線形拘束オートマトン	$\alpha X \beta \rightarrow \alpha \gamma \beta$
文脈自由文法	プッシュダウンオートマトン	$X \rightarrow \alpha$
正規文法	有限オートマトン	X→a，X→aY

・XとYは非終端記号，αとβは0個以上の記号列，γは1個以上の記号列
・文脈依存文法では |左辺|≦|右辺|という制限があることも

文脈自由文法の部分クラス

- 文脈自由文法の構文解析時間は$O(n^3)$と大きい
  - nは入力文の長さ
- 効率のため「1パス，O(n)，1文字先読みで解析可能」な部分クラスを使うことが多い
  - LL(1)文法
  - LR(1)文法，LALR(1)文法，SLR(1)文法
  - 演算子順位文法 (operator precedence grammar) 本講義では扱わない
- 文法クラスの言語の包含関係



余談

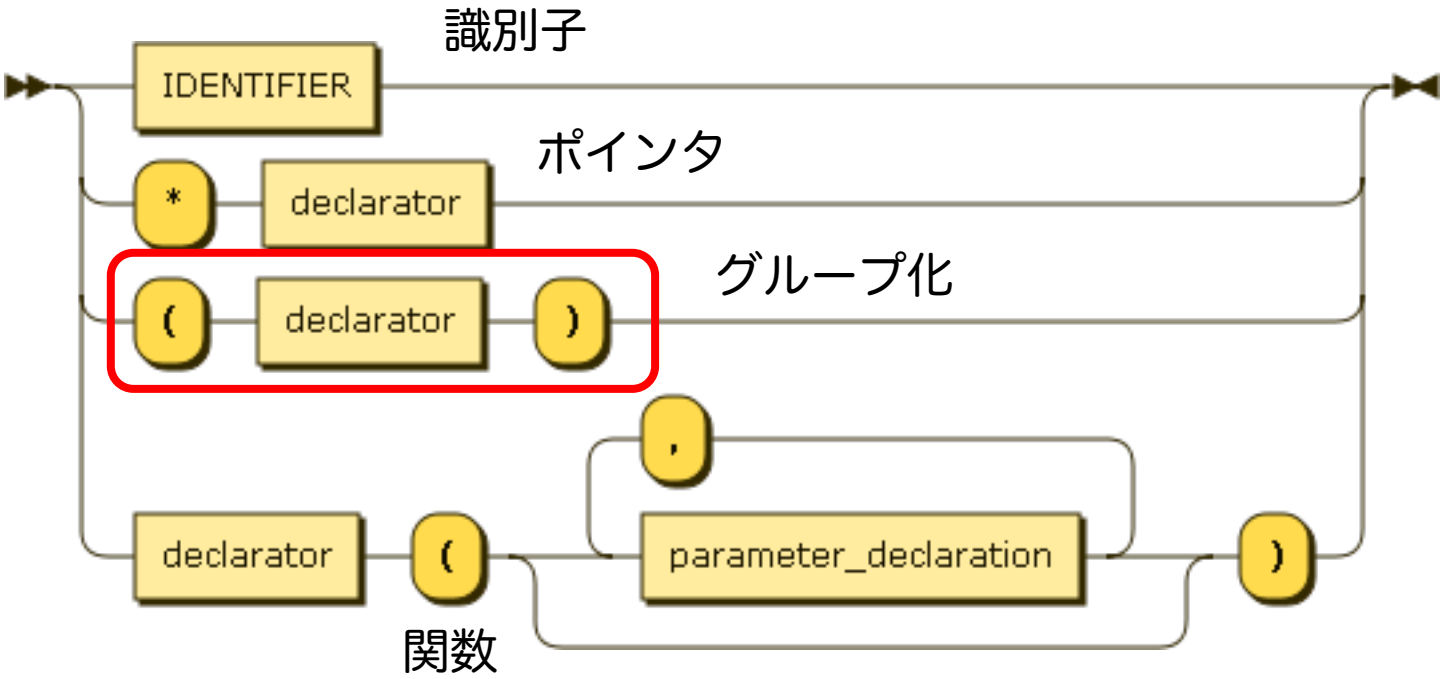
- プログラミング言語の文法を定義する際は，LALR(1)やLL(1)にするのが良い
  - でもLL(1)は左再帰が使えない→不便
- しかし，CやC++はLALR(1)でもLL(1)でも無い
  - Cではtypedef名が悪者.
    文脈の情報が無いとfooが関数か型名か判別できない.
  - C++はもっと遥かに悲惨.
    構文解析のために意味解析が必要.

```
typedef int foo;
foo (x);
```

foo (x)は関数呼び出しではなく，
型fooの変数xの宣言になる

int (x); は int x; と同じ

C言語の declarator 宣言子



配列は省略

- int (*f)(); の「(*f)()」の部分が declarator
- declarator はカッコでくくってグループ化できる
  - int x; は，int (x); と書いても（冗長だが）OK

パーサ生成系 parser generator

- 文法を与えるとパーサを自動生成するプログラム
  - コンパイラ・コンパイラとも呼ばれる
- 代表的なもの
  - カッコ内は先読み記号の数

名前	文法	入力言語	出力言語
GNU Bison	LALR(1)+α	BNF	C, C++
ANTLR4	LL(*)	EBNF	Java他多数
JavaCC	LL(k)	EBNF	Java, C++

LL(1)とLALR(1)

- LALR(1)は計算機科学の勝利と言われたが
  - GCC-4.0からLALR(1)自動生成をやめ，
    再帰下降型の手書き構文解析器に戻った
- PEG，パーサコンビネータ，LL(*)の台頭
  - コンパイラ技術は枯れたと思われたが，そうではなかった
  - PEG = parsing expression grammar