言語処理系論

小林　直樹

講義内容

- プログラミング言語処理系（インタプリタ、コンパイラ）の仕組みについて学ぶ
  - 字句解析
  - 構文解析
  - 意味解析（型検査、プログラム解析）
  - 最適化
  - コード生成

なぜ言語処理系を学ぶのか？

- プログラミング言語処理系は
  OSとならび最も重要な基盤ソフトウェア
- プログラム意味論や言語設計にも密接に関係
  - 処理系のことを考えないと遅すぎて使えない言語に
  - 逆にプログラム意味論（cf.「言語モデル論」）を
    知らないと挙動が謎の処理系に
- 代表的かつ大規模な記号処理プログラム
  - 字句解析、構文解析などは自然言語処理など
    他の記号処理とも多くの共通点

教科書等

教科書：

Andrew Appel,
Modern Compiler Implementation in ML,
Cambridge University Press
ISBN 0 521 60764 7
http://www.cs.princeton.edu/~appel/modern/ml

講義ホームページ：

http://www-kb.is.s.u-tokyo.ac.jp/~koba/class/compiler/
（補足資料、レポート課題等を掲載予定。）

本日の講義内容

- 言語処理系って？
  - インタプリタとコンパイラ
- 高級プログラミング言語と機械語の違い
- インタプリタとコンパイラの内部構造

言語処理系の種類

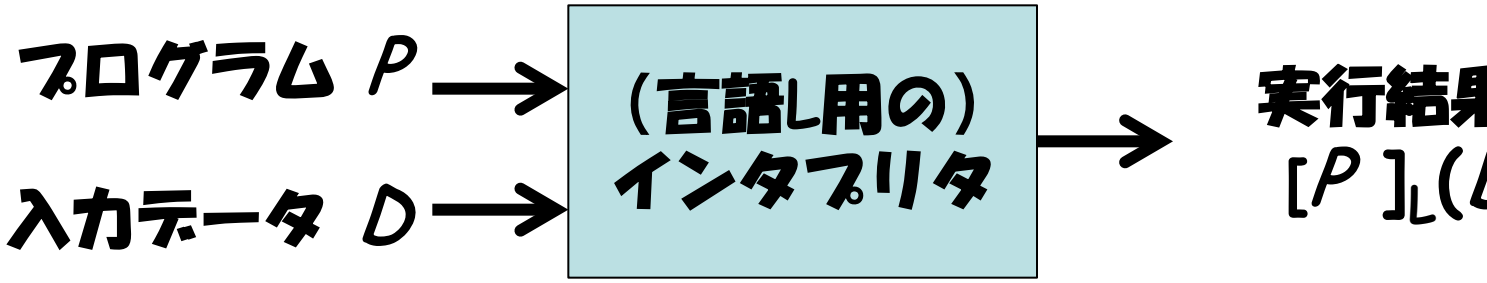
- インタプリタ
  - プログラムを入力として受け取り、解釈実行するソフトウェア

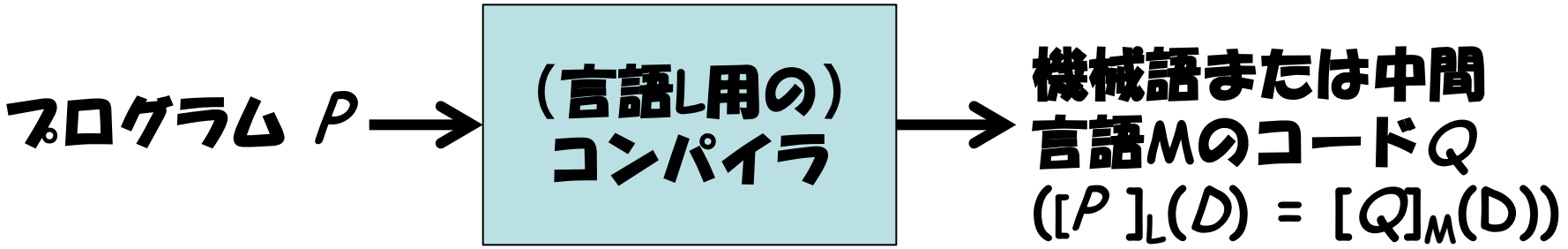
  例：MLのインタラクティブ環境
- コンパイラ
  - プログラムを入力として受け取り、機械語（または中間言語）のプログラムを出力するソフトウェア

  例：cc, javac

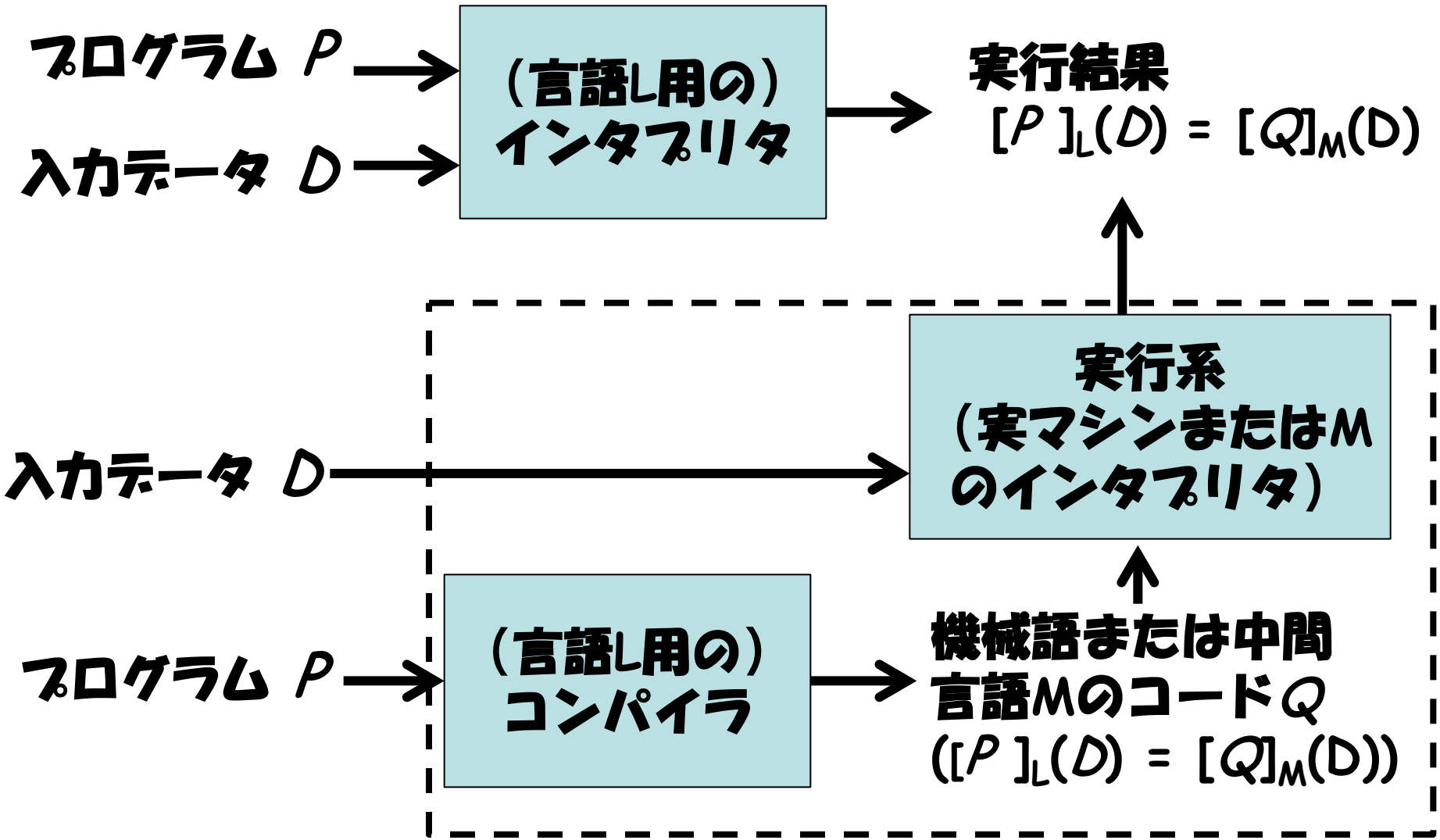
インタプリタ ≅ （典型的には中間コードへの）コンパイラ+実行系

インタプリタとコンパイラ

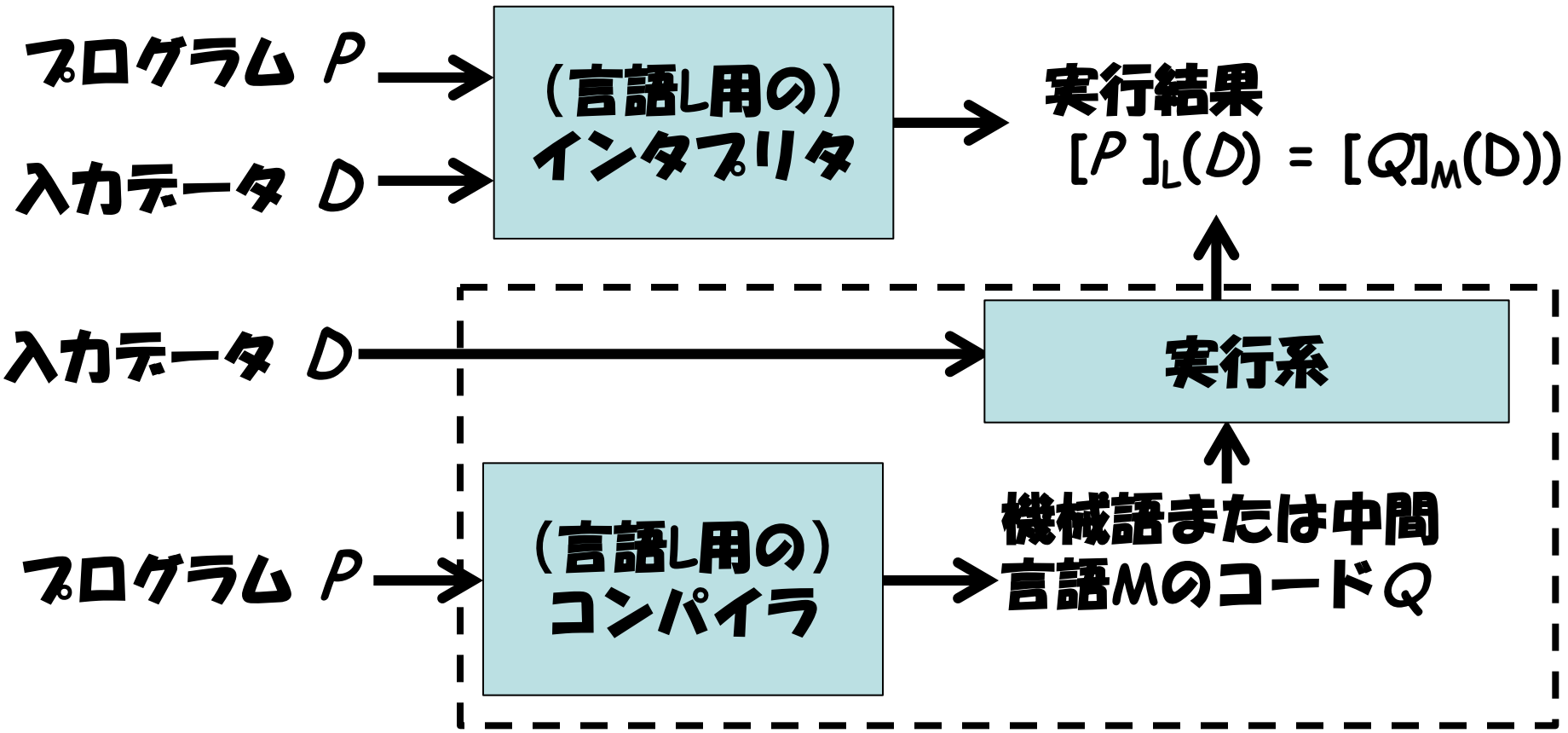




インタプリタ=コンパイラ+実行系



インタプリタ=コンパイラ+実行系



効率上の違いあり：

- インタプリタ：入力(P,D)に対する一回の実行が速くなるように設計
- コンパイラ：複数の入力データD=D1,D2,...に対するQ(D)の実行が速くなるようなQを出力

プログラムの実行方式

- インタプリタ方式
  - インタプリタを用いてプログラムを実行
    （例：SMLのインタラクティブ環境）
- コンパイラ方式
  - コンパイラによって機械語に変換した後に実行
    （例：C言語、MLのネイティブコードコンパイラ）
- コンパイラ・インタプリタ方式
  - コンパイラによって中間言語のコードに変換した後に中間コードのインタプリタによって実行
    (javac + java)

プログラムの実行方式の利害得失

- インタプリタ方式
  - ×起動のたびにプログラムを解釈しなおすので、何度も同じプログラムを実行するには不向き
  - ○ コンパイラを別に起動せずに即実行できるので、開発段階で便利
  - ○ プログラムが計算機の機種に非依存
- コンパイラ方式
  - ○ あらかじめ機械語に変換してあるので高速な実行が可能
  - × プログラムを変更するたびにコンパイルし直さなければならないので、プログラム開発段階では不便
  - ×計算機の機種ごとに違う機械語コードを用意する必要あり
- コンパイラ・インタプリタ方式
  - 両者の利点・欠点を併せ持つ

Java言語での処理方式

- Javaソースプログラムをjavacでコンパイルして Java 仮想機械言語(JVML)で書かれたバイトコード（クラスファイル）に変換
- バイトコードをバイトコードインタプリタで解釈実行
- バイトコードのうちの一部は実行中に実際の機械語にコンパイル(JIT: Just-In-Time compilation)

本日の講義内容

- 言語処理系って？
  - インタプリタとコンパイラ
- 高級プログラミング言語と機械語の違い
- インタプリタとコンパイラの内部構造

高級プログラミング
の違い

変数名でデータにアクセス

メモリアドレスやレジスタ名でデータにアクセス

```
int x, y;
while(x!=y)
{
    if(x>y)
        x=x-y;
    else
        y=y-x;
 };
```

```
L11: movl   -4(%ebp), %eax
     cmpl   -8(%ebp), %eax
     jne    L13
     jmp    L12
L13: movl   -4(%ebp), %eax
     cmpl   -8(%ebp), %eax
     jle    L14
     movl   -8(%ebp), %edx
     leal   -4(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
L14: movl   -4(%ebp), %edx
     leal   -8(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
L12: ...
```

高級プログラミング言語と機械語の違い

型によって
データの種類を
区別

```
int x, y;
while(x!=y)
{
    if(x>y)
        x=x-y;
    else
        y=y-x;
};
```

```
L11: movl   -4(%ebp), %eax
     cmpl   -8(%ebp), %eax
     jne    L13
     jmp    L
L13: movl
     cmpl
     jle    L14
     movl   -8(%ebp), %edx
     leal   -4(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
L14: movl   -4(%ebp), %edx
     leal   -8(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
L12: ...
```

型は無し（ただのビット列）

高級プログラミング言語と機械語の違い

```
int x, y;
while(x!=y)
{
    if(x>y)
        x=x-y;
    else
        y=y-x;
};
```

while文、if文、再帰などの制御構造

（条件付き）ジャンプ命令のみ

```
L11: movl   -4(%ebp), %eax
     cmpl   -8(%ebp), %eax
     jne    L13
     jmp    L12
L13: movl   -4(%ebp), %eax
     cmpl   -8(%ebp), %eax
     jle    L14
     movl   -8(%ebp), %edx
     leal   -4(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
     movl   -4(%ebp), %edx
     leal   -8(%ebp), %eax
     subl   %edx, (%eax)
     jmp    L11
L12: ...
```

高級言語と機械語の違い（まとめ）

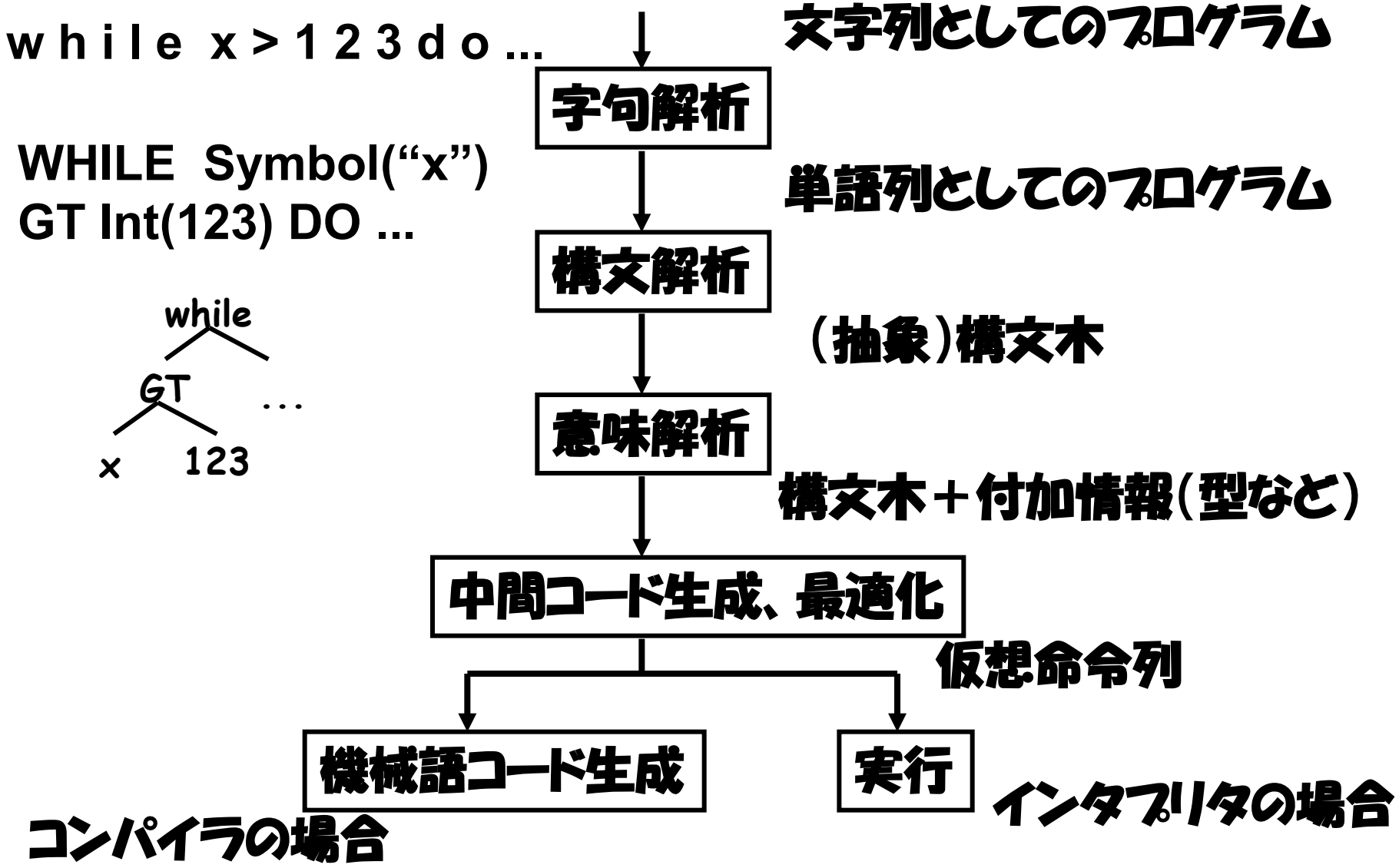
	高級言語	機械語
データの参照	変数名	メモリアドレス、レジスタ名
データの型	あり（型に応じた演算子の選択、型エラーの検出）	なし
制御構造	分岐、繰り返し(while,for)、再帰関数、高階関数	（条件つき）ジャンプ

これらのギャップを埋めるのがコンパイラの役割

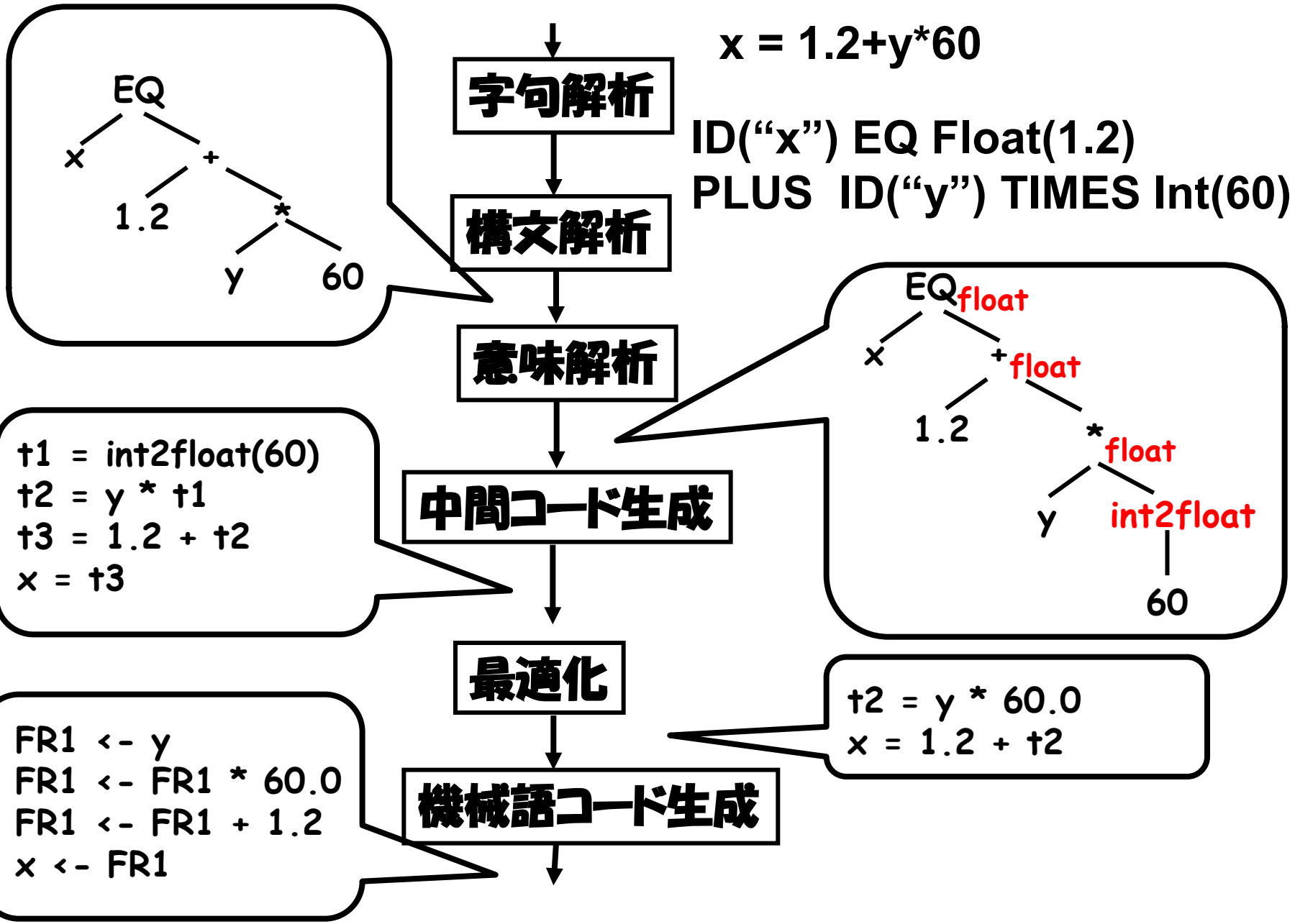
本日の講義内容

- 言語処理系って？
  - インタプリタとコンパイラ
- 高級プログラミング言語と機械語の違い
- インタプリタとコンパイラの内部構造

インタプリタ、コンパイラの内部



コンパイラの構造


x = 1.2+y*60
字句解析
ID(“x”) EQ Float(1.2)
PLUS ID(“y”) TIMES Int(60)
構文解析
EQ
x
+
1.2
*
y
60
意味解析
EQ float
x
+ float
1.2
* float
y
int2float
60
中間コード生成
t1 = int2float(60)
t2 = y * t1
t3 = 1.2 + t2
x = t3
最適化
t2 = y * 60.0
x = 1.2 + t2
機械語コード生成
FR1 <- y
FR1 <- FR1 * 60.0
FR1 <- FR1 + 1.2
x <- FR1

次回までにやっておくこと

- 教科書の購入
- ノートPCにocamlをインストール
  （プログラミング実験で指示があるはず）
- 前学期の「形式言語論」の内容を復習
  （特に、正規言語、オートマトン、正規表現、
  文脈自由文法）
- レポート課題（後述）

レポート課題

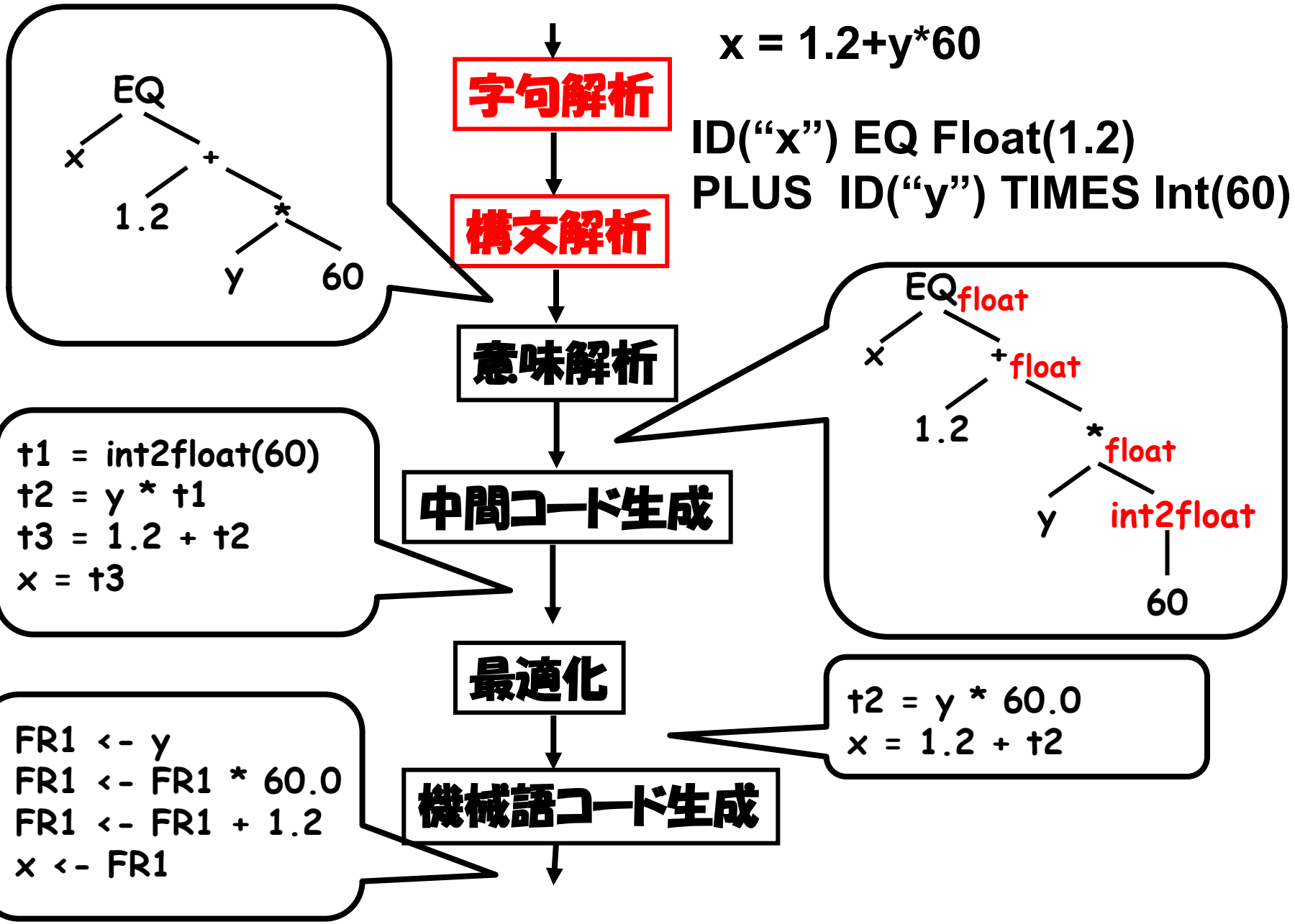
1. Cで簡単なプログラム（ただし非自明な
   再帰またはwhile文を含めること）を記述
2. 1のプログラムをアセンブリコードにコンパイル
   (gcc -S <filename>)
3. 2のアセンブリコードの各部分が1のソースプログラムのどこに対応しているか考えてまとめる

提出方法：電子メールで koba@is.s.u-tokyo.ac.jpまで
（件名（subject)は、「言語処理系論第一回レポート」
本文に氏名、学生証番号を記述すること）
提出期限：次回講義開始時

アウトライン

- 字句解析・構文解析とその役割
- 字句解析

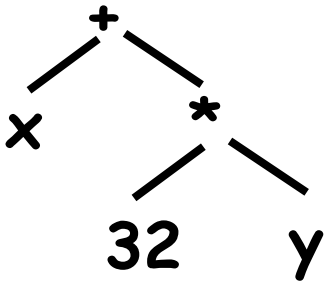
コンパイラの構造


x = 1.2+y*60
字句解析
ID(“x”) EQ Float(1.2)
PLUS ID(“y”) TIMES Int(60)
構文解析
EQ
x
+
1.2
*
y
60
意味解析
EQ float
x
+ float
1.2
* float
y
int2float
60
中間コード生成
t1 = int2float(60)
t2 = y * t1
t3 = 1.2 + t2
x = t3
最適化
t2 = y * 60.0
x = 1.2 + t2
機械語コード生成
FR1 <- y
FR1 <- FR1 * 60.0
FR1 <- FR1 + 1.2
x <- FR1

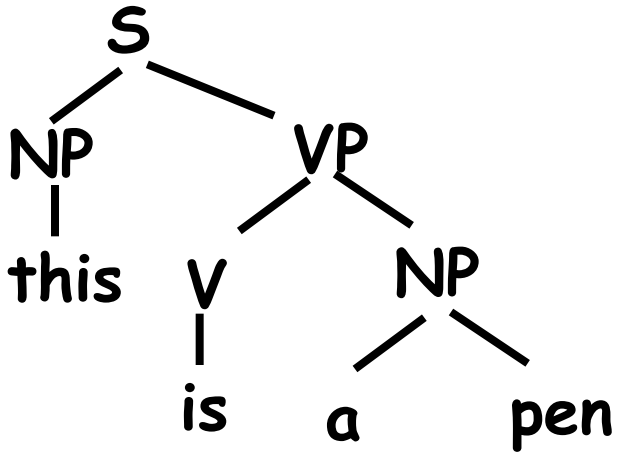
字句解析・構文解析の役割

- 文字列をその論理的構造を表す木（**抽象構文木**、*AST*）に変換

x+32*y ⟶

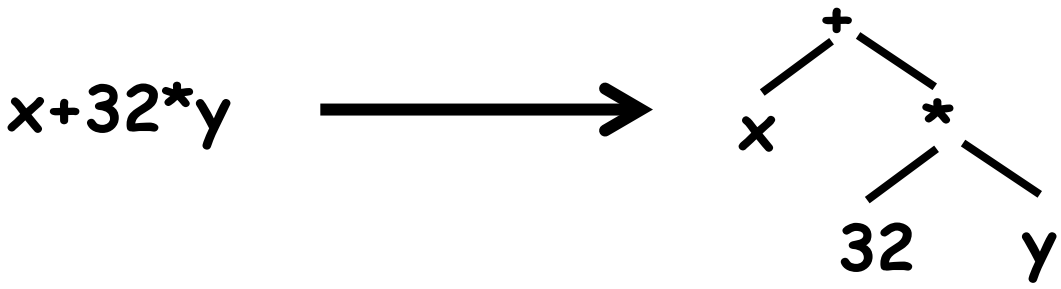


This is a pen. ⟶



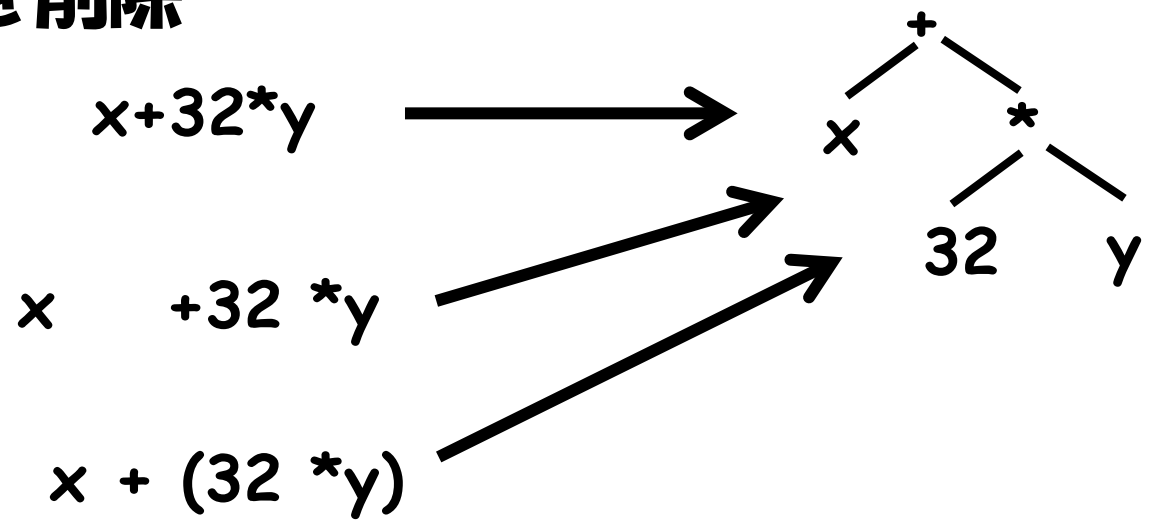
字句解析・構文解析の役割

- 文字列をその論理的構造を表す木（**抽象構文木**、*AST*）に変換
  - 文（プログラム）の構造を明確にし、後の処理を簡単化



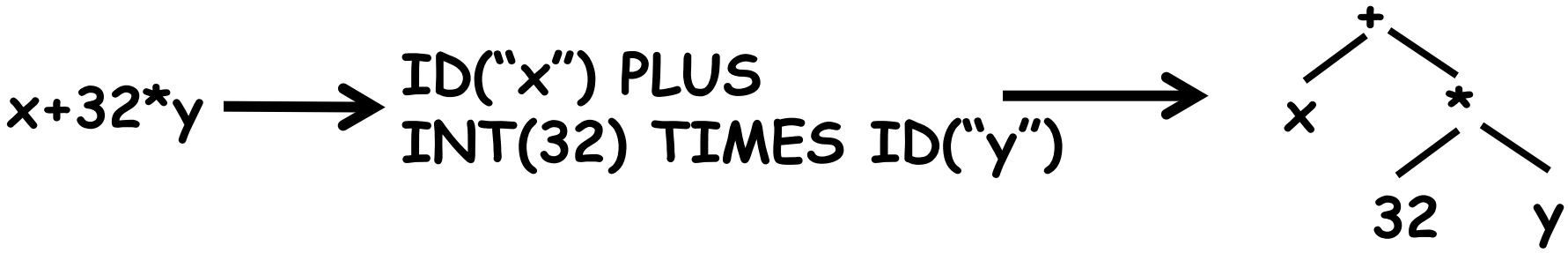
字句解析・構文解析の役割

- 文字列をその論理的構造を表す木（**抽象構文木**、*AST*）に変換
  - 文（プログラム）の構造を明確にし、後の処理を簡単化
  - スペースや括弧など文の意味に関係ない情報を削除



字句解析と構文解析

- 文字列からASTへの変換を２ステップに
  - 字句解析
    - 文字列からトークン（単語）の列に変換
  - 構文解析
    - トークン列を抽象構文木（AST）に変換



字句解析と構文解析

- 文字列からASTへの変換を２ステップに
  - 字句解析
    - 文字列からトークン（単語）の列に変換
  - 構文解析
    - トークン列を抽象構文木（AST）に変換
- 分割する意義
  - それぞれに特化した処理・理論を適用
    - 字句解析：正規言語
    - 構文解析：文脈自由言語

  -> わかりやすさ、効率

注意

- 言語（自然言語など）によっては
  字句解析と構文解析は明確に区別できない

  例：「ももももも。すももももも。」

  字句解析（形態素解析）の結果は複数通り。

  - もも　も　もも　。すもも　も　もも　。
  - もも　もも　も　。す　もも　も　も　も　。

  どれが正しいかは構文解析しないと不明。

- 場合によっては意味解析しないと区別不能。

  「あきた千秋公園桜まつり」

  -> 「あきた（秋田）」「千秋公園」「桜まつり」

  「あき（飽き）」「た」「千秋公園」「桜まつり」

注意

- 言語（自然言語など）によっては
  字句解析と構文解析は明確に区別できない

  例：「ももももも。すももももも。」

  字句解析（形態素解析）の結果は複数通り。

  - もも　も　もも　。すもも　も　もも　。
  - もも　もも　も　。す　もも　も　も　も　。

  どれが正しいかは構文解析しないと不明。

- 通常のプログラミング言語は、
  字句解析と構文解析を分けられるように設計

アウトライン

- 字句解析・構文解析とその役割
- 字句解析

字句解析器の構成

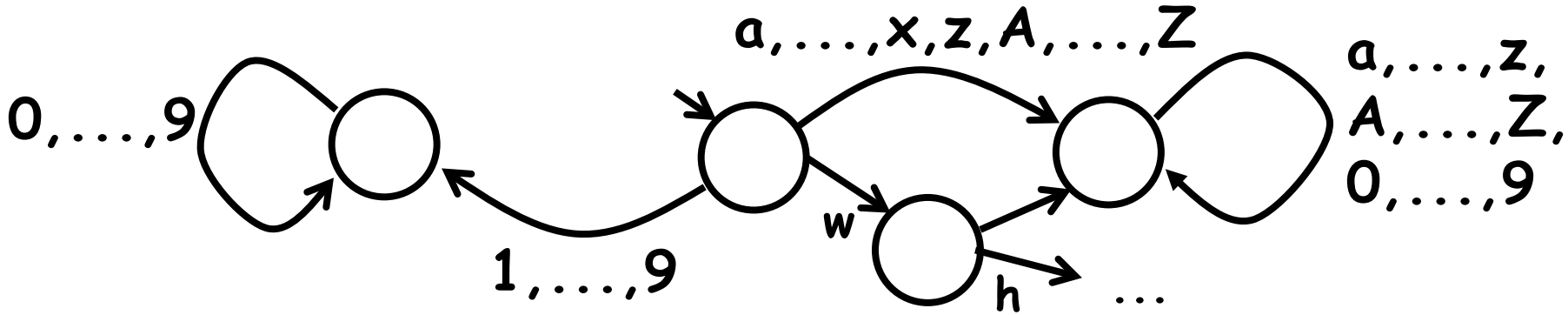
- トークンの仕様を正規表現で記述

  WHILE: “while”

  ID:　(a|…|z|A|…|Z) (a|…|z|A|…|Z|0|..|9)*

  NUM: (1|…|9)(0|…|9)*

- トークンに相当する文字列を受理するオートマトンを構築、プログラム化



正規表現（復習）

e ::= 0　　空集合

1　　空列だけからなる集合

（数字の1と紛らわしいときはεと書く）

a　　aだけからなる集合

$e_1 e_2$　　連接

$([e] = \{ w_1 w_2 \mid w_i \in [e_i]\})$

$e_1 \mid e_2$　　和集合

e^*　　Kleene 閉包

$([e^*] = \{w_1 \ldots w_n \mid w_1, \ldots, w_n \in [e]\}$

字句解析などで用いられる拡張正規表現

e ::= ...

e^+ 1個以上の繰り返し

($[e^+] = \{w_1 \ldots w_n \mid n>0,\ w_1,\ldots,w_n \in [e]\}$)

$[a_1\text{-}a_2]$ $a_1|\ldots|a_2$

(文字コードでa_1 からa_2までのいずれか)

正規表現からオートマトンへ

- 教科書的手法：正規表現の構文に従って合成的に構成

- Brzozowski's derivativeを用いる手法

正規表現からオートマトンへ

- **教科書的手法：正規表現の構文に従って合成的に構成**

- Brzozowski's derivativeを用いる手法

正規表現からオートマトン：教科書的手法（復習）

正規表現

↓

εー遷移付き非決定性オートマトン

↓

非決定性オートマトン

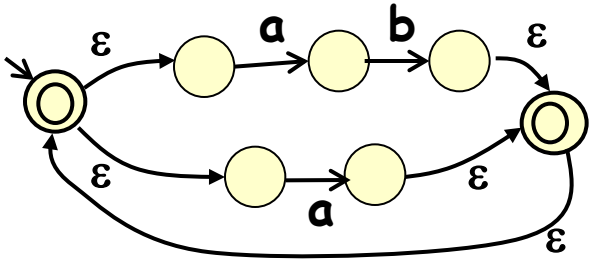
↓

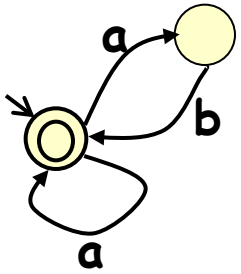
決定性オートマトン

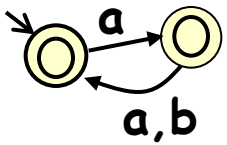
↓

状態数最小の決定性オートマトン

(ab | a)*

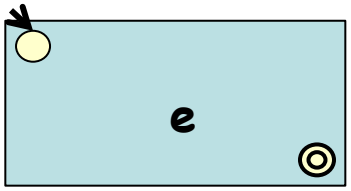




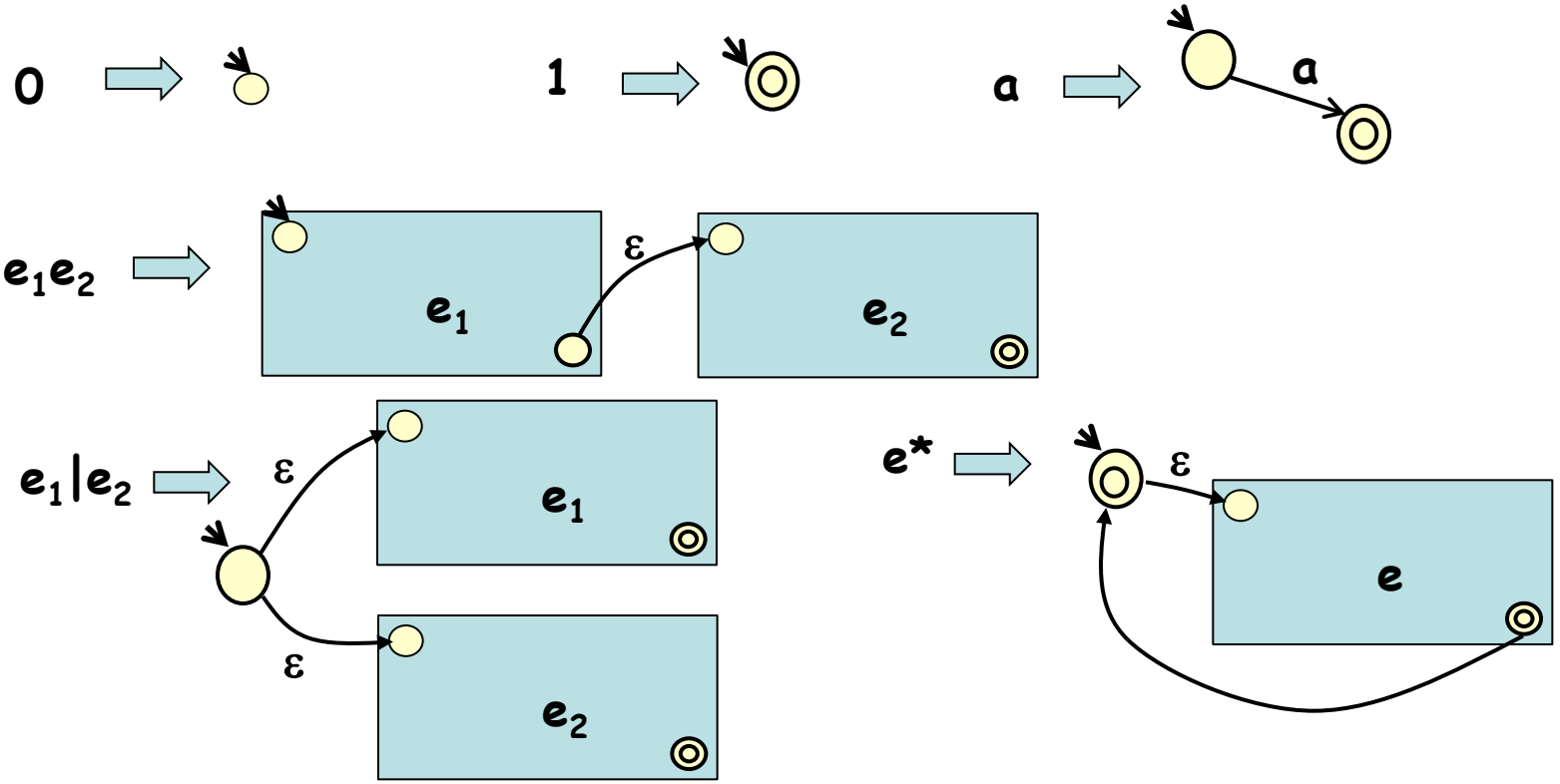


正規表現からε遷移付きオートマトン

正規表現 e に対応するオートマトンを



と表すことにすると...



教科書的変換の欠点

- 多段階の変換
- 状態爆発
  - ε遷移の除去、決定化の際の2回の
    "powerset construction"
    （「状態の集合」を状態に持つオートマトンを構成）

正規表現からオートマトンへ

- 教科書的手法：正規表現の構文に従って合成的に構成

- Brzozowski's derivativeを用いる手法

Brzozowski's derivative [1]による方法

- 各正規表現e, 入力シンボルaについて以下を計算
  - $D_a(e)$:
    eの各要素からaを取り除いてできる語からなる言語を表す正規表現
    ($L(D_a(e)) = \{w \mid aw \in L(e)\}$)
  - E(e): eが空列を含むなら1, そうでなければ0

[1] Brzozowski, "Derivatives of Regular Expressions", JACM, 1964

Brzozowski's derivative [1]による方法

- $D_a(e)$:
  eの各要素からaを取り除いてできる語からなる言語を表す正規表現
  ($L(D_a(e)) = \{w \mid aw \in L(e)\}$)
- $E(e)$: eが空列を含むなら1, そうでなければ0

$D_a(0) = 0$　　$D_a(1) = 0$　　$D_a(a) = 1$　　$D_a(b) = 0$

$D_a(e_1e_2) = D_a(e_1)e_2 + E(e_1)D_a(e_2)$

$D_a(e_1+e_2) = D_a(e_1) + D_a(e_2)$　　$D_a(e^*) = D_a(e)e^*$

$E(0)=0$　　$E(1)=1$　　$E(a)=0$　　$E(e_1e_2)=E(e_1)E(e_2)$

$E(e_1+e_2)=E(e_1)+E(e_2)$　　$E(e^*)=1$

Brzozowski's derivative の計算例

$$
\begin{aligned}
D_a((ab + a)^*) &= D_a(ab + a)\,(ab + a)^* \\
&= (D_a(ab) + D_a(a))\,(ab + a)^* \\
&= (D_a(a)b + E(a)D_a(b) + D_a(a))(ab + a)^* \\
&= (b + 1)(ab + a)^*
\end{aligned}
$$

$D_a(0) = 0 \qquad D_a(1) = 0 \qquad D_a(a) = 1 \qquad D_a(b) = 0$

$D_a(e_1e_2) = D_a(e_1)e_2 + E(e_1)D_a(e_2)$

$D_a(e_1+e_2) = D_a(e_1) + D_a(e_2) \qquad D_a(e^*) = D_a(e)e^*$

$E(0)=0 \qquad E(1)=1 \qquad E(a) =0 \qquad E(e_1e_2)=E(e_1)E(e_2)$

$E(e_1+e_2)=E(e_1)+E(e_2) \qquad E(e^*)=1$

Brzozowski's derivative [1]による方法

- 各正規表現e, 入力シンボルaについて以下を計算
  - $D_a(e)$:
    eの各要素からaを取り除いてできる語からなる言語を表す正規表現
    ($L(D_a(e)) = \{w \mid aw \in L(e)\}$)
  - $E(e)$: eが空列を含むなら1, そうでなければ0
- D_aによる閉包（与えられたeにD_aを繰り返し適用して得られる正規表現の集合）を求め、各正規表現（の同値類）を状態とするオートマトンを構成
  - 遷移関数は、$\delta(e,a) = D_a(e)$
  - $E(e)=1$を満たす状態が受理状態

例は黒板で