

# プログラミング言語処理系 イントロダクション

末永 幸平

# この資料について

- 京都大学工学部専門科目「プログラミング言語処理系」の講義資料
- 講義 Web ページ:
  - <https://kuis-isle3sw.github.io/loPLMaterials/>
- 講義をする人: 末永幸平
  - <https://researchmap.jp/ksuenaga/>
  - <https://twitter.com/ksuenaga>

# 講義全般についての情報

- Prerequisite

- OCaml をある程度かけること
  - 専門科目「プログラミング言語」中の OCaml に関する部分
- 正則言語，有限状態オートマトン，文脈自由言語に関する知識
  - 専門科目「言語・オートマトン」

- 本講義を前提とする科目

- 「計算機科学実験及演習 3（ソフトウェア）」は  
本講義前半の内容をマスターしていることが前提

この講義でやること =  
「プログラミング言語の作り方」

# そもそもプログラムとは

- 計算機に行わせるべき処理を記述した文字列
  - `int main(void) { printf("%d", 3); return 0; }`
  - `let x = 3 in x + 2`
  - ...

世界は  
プログラミング言語で  
溢れている

[https://en.wikipedia.org/wiki/List\\_of\\_programming\\_languages](https://en.wikipedia.org/wiki/List_of_programming_languages)

# なぜこんなにもたくさんの プログラミング言語が!?

- よく言われる理由：言語ごとに得意なことが違うしライブラリも違う
  - 低レイヤが得意な C
  - 知っていると五十嵐さんと仲良くなれる Java
  - 機械学習が得意な Python
  - Rails がある Ruby
  - 言語処理系を書くのにお手軽な OCaml
  - 知っていると賢そうな Haskell
  - 参考書が分厚い C++

# なぜこんなにもたくさんの プログラミング言語が!?

- 本当の理由：言語を作るのは人間にとって楽しい！
  - だってかっこいいから！
  - どのくらい楽しいか
    - **プログラミング言語が作れば，その言語でどんなプログラムが書けるかなんて全然気にならないくらい**



この講義の目標:

「一人でも多くの学生を  
プログラミング言語沼に  
引きずり込む」

# 本科目をやる少し真面目な理由

- プログラミング言語は情報科学の重要な部分を占めている
  - 大体の情報科学系の学科では必ず教えられている
- 形式言語理論（オートマトン，正則表現，文脈自由言語）が実世界でどう使われているかが分かる
- ソフトウェアで解決すべき問題があるときに，その問題専用の言語が作れると，解決策の幅が広がりうる
  - DSL (domain-specific language)：ある問題に特化して設計された言語
- まあまあ大規模のソフトウェア作成として，言語処理系は良い課題

ようこそ言語沼へ

# 今日の内容

- ~~布教活動~~イントロダクション
- **シンタックスとセマンティクス**
- プログラミング言語の実装方式
- 典型的なインタプリタの構成
- 典型的なコンパイラの構成
- 本科目全体の構成

# 何をしたら

## 「プログラミング言語を定義した」と言えるのか

- 理論的な答え：**シンタックスとセマンティックスの定義**
  - シンタックス (syntax, 統語論)
    - プログラミング言語の文法 (どのような文字列が文法的に正しいかを定める規則)
  - セマンティックス (semantics, 意味論)
    - 文法的に正しいプログラムが, どのような意味を持つかを定める規則
    - プログラムの意味 = どのような計算をするか

# Syntax

- プログラミング言語の文法（どのような文字列が文法的に正しいかを定める規則）
  - ```
int main(void) {  
    int x = 3;  
    return 0;  
}
```

は文法的に正しい C 言語のプログラム
  - ```
public class Main {  
    static public void main(String[] argv) {  
        int x = 3;  
        return;  
    }  
}
```

はだいたい同じように振る舞う Java プログラムだが、C プログラムとしては文法的に正しくない

# Syntax の定め方

- 大体**文脈自由文法**を使って文法的に正しいプログラムの集合を定義
  - 例：足し算と掛け算を持つ正しい算術式の集合は、以下の書き換え規則を持つ文法で定められる

$S \rightarrow E$   
 $E \rightarrow E + E$   
 $E \rightarrow E * E$   
 $E \rightarrow \text{自然数}$   
 $E \rightarrow (E)$

ただし、  
開始記号 =  $S$ ,  
終端記号の集合 =  $\{ (, ), *, + \} \cup \{ 0, 1, 2 \dots \}$   
非終端記号の集合 =  $\{ S, E \}$

# Semantics

- 文法的に正しいプログラムの意味を定める
  - プログラムの意味 = どのような計算をするか
  - 例: C プログラム

```
int main(void) {  
    int x = 3;  
    return 0;  
}
```

はローカル変数に 3 を代入したあとに 0 を返すプログラム



# Semantics の定め方

- 普通は**自然言語**で定める
- より厳密に定めたいときには**数学的に意味論を定める**
  - 「プログラム意味論」という理論計算機科学の一分野を形成
  - ある種のトポロジーやら圏論やらが出てきたりしてすごいぞ
  - 詳しく知りたい人は教科書に挙げている参考書を読もう

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- **プログラミング言語の実装方式**
- 典型的なインタプリタの構成
- 典型的なコンパイラの構成
- 本科目全体の構成

# Syntax と Semantics を定めれば終わり？

- 数学的には終わりだが，動くプログラミング言語を作るには  
**プログラミング言語を実装**する必要
  - プログラミング言語の実装＝  
実際のプログラムを受け取って，そのプログラムが意図する計算を  
計算機で実行するためのソフトウェア

このソフトウェアを  
**プログラミング言語処理系**と呼ぶ

# プログラミング言語の実装方式の代表例

- **インタプリタ (interpreter)**

- プログラムを受け取ると、そのプログラムが表す計算を実行するソフトウェア

- **コンパイラ (compiler)**

- プログラムを受け取ると、そのプログラムと等価な  
（多くの場合別の言語の）プログラムを生成するソフトウェア
- 変換先がマシン語ならば、直接計算機で実行できる

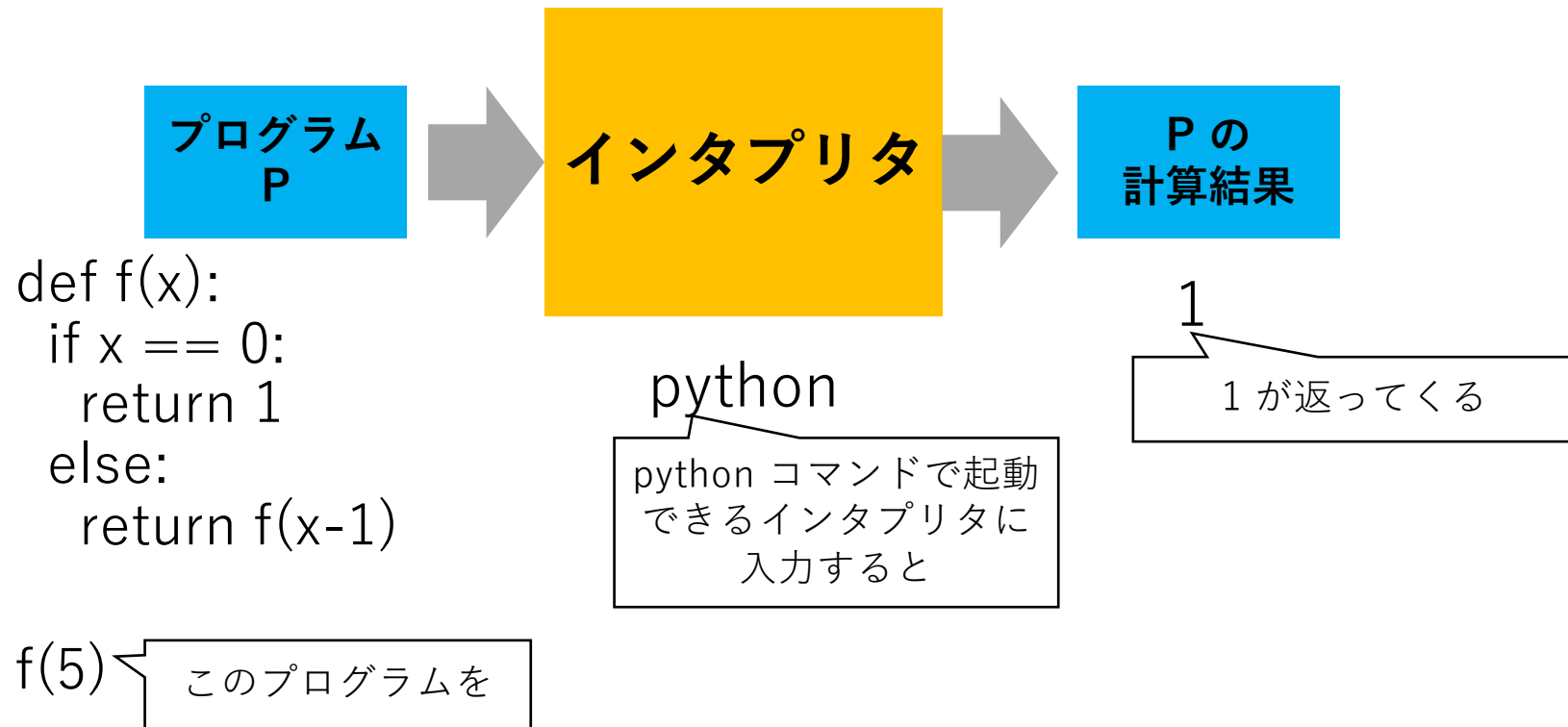
# インタプリタ

- 入力：プログラム P  
出力：P の計算結果



# インタプリタ

- 入力：プログラム P  
出力：P の計算結果



# コンパイラ

- 入力：言語  $L_1$  で書かれたプログラム  $P$   
出力：言語  $L_2$  で書かれた  $P$  と等価なプログラム

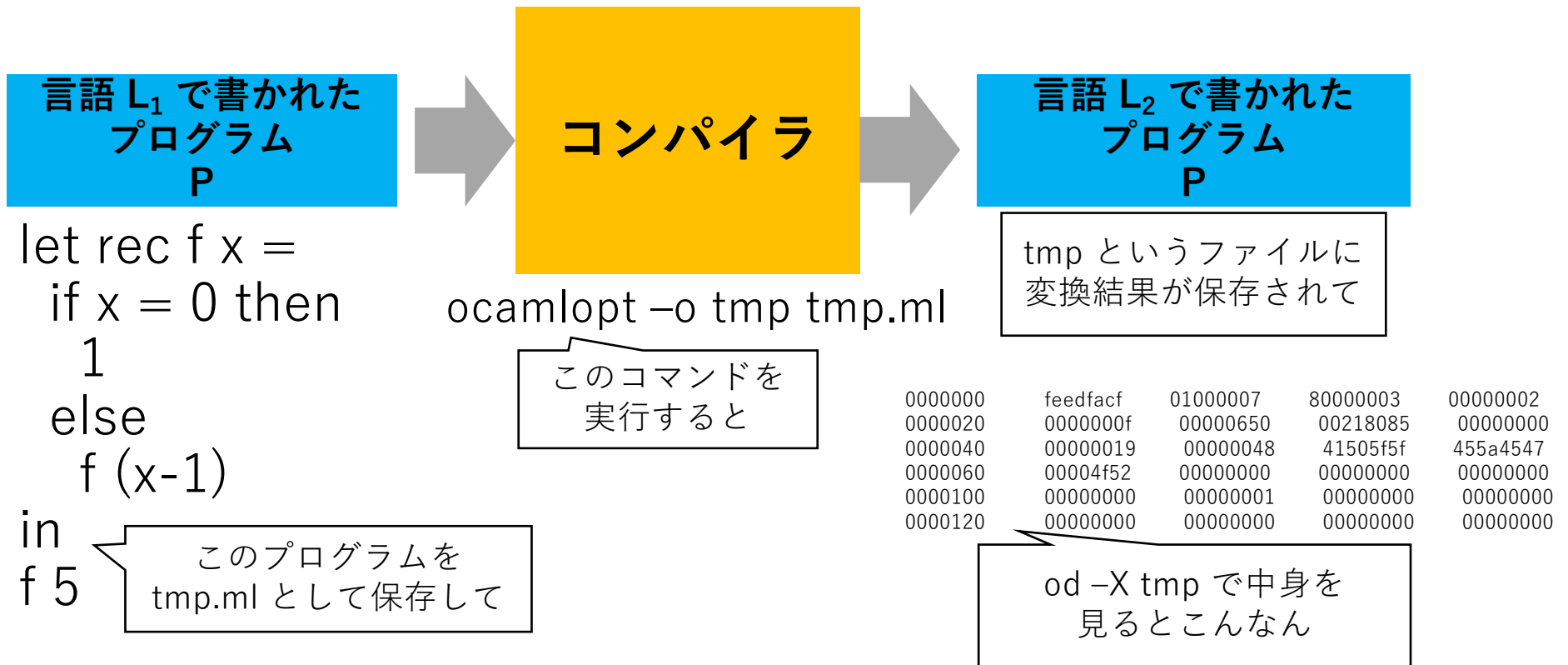
違う言語で書かれた  
同じ意味のプログラムに変換



言語  $L_2$  がマシン語なら  
そのまま実行できる

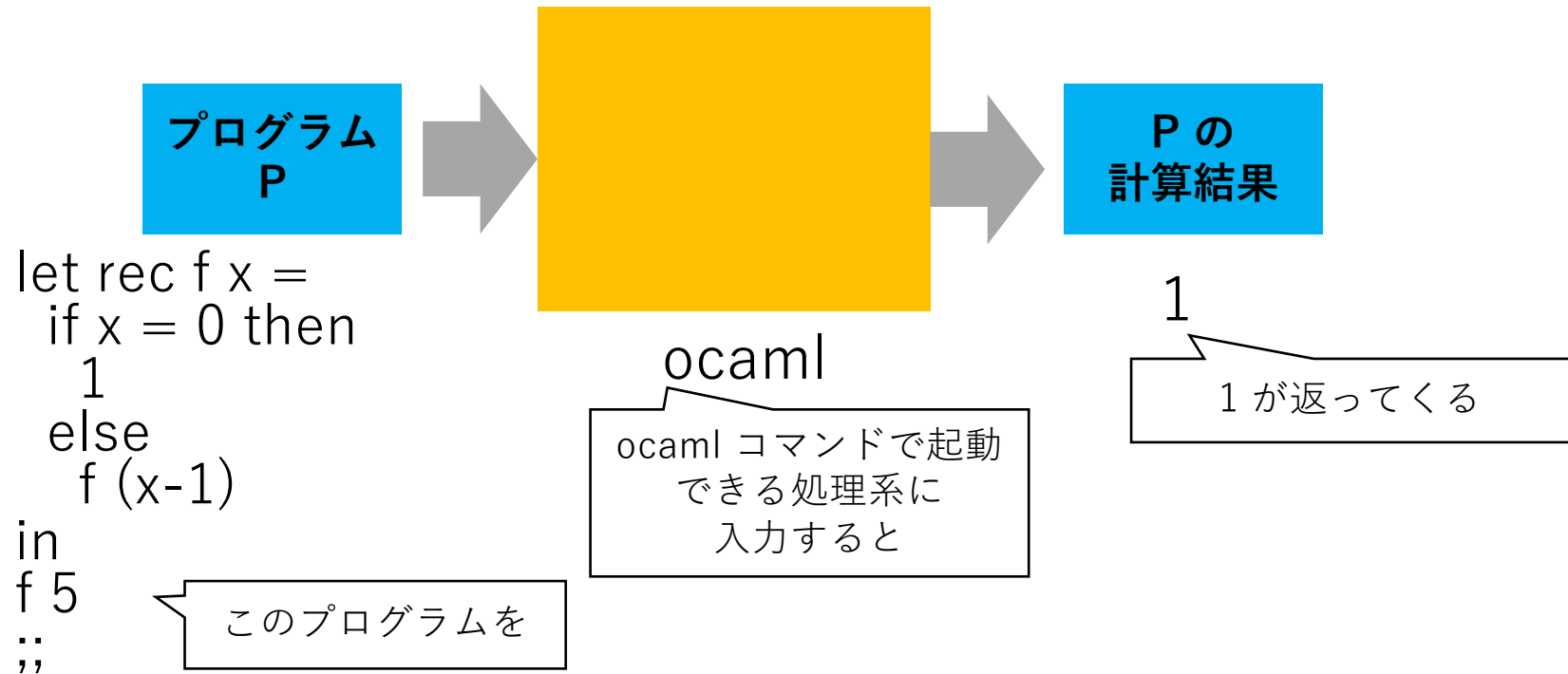
# コンパイラ

- 入力：言語  $L_1$  で書かれたプログラム  $P$   
出力：言語  $L_2$  で書かれた  $P$  と等価なプログラム

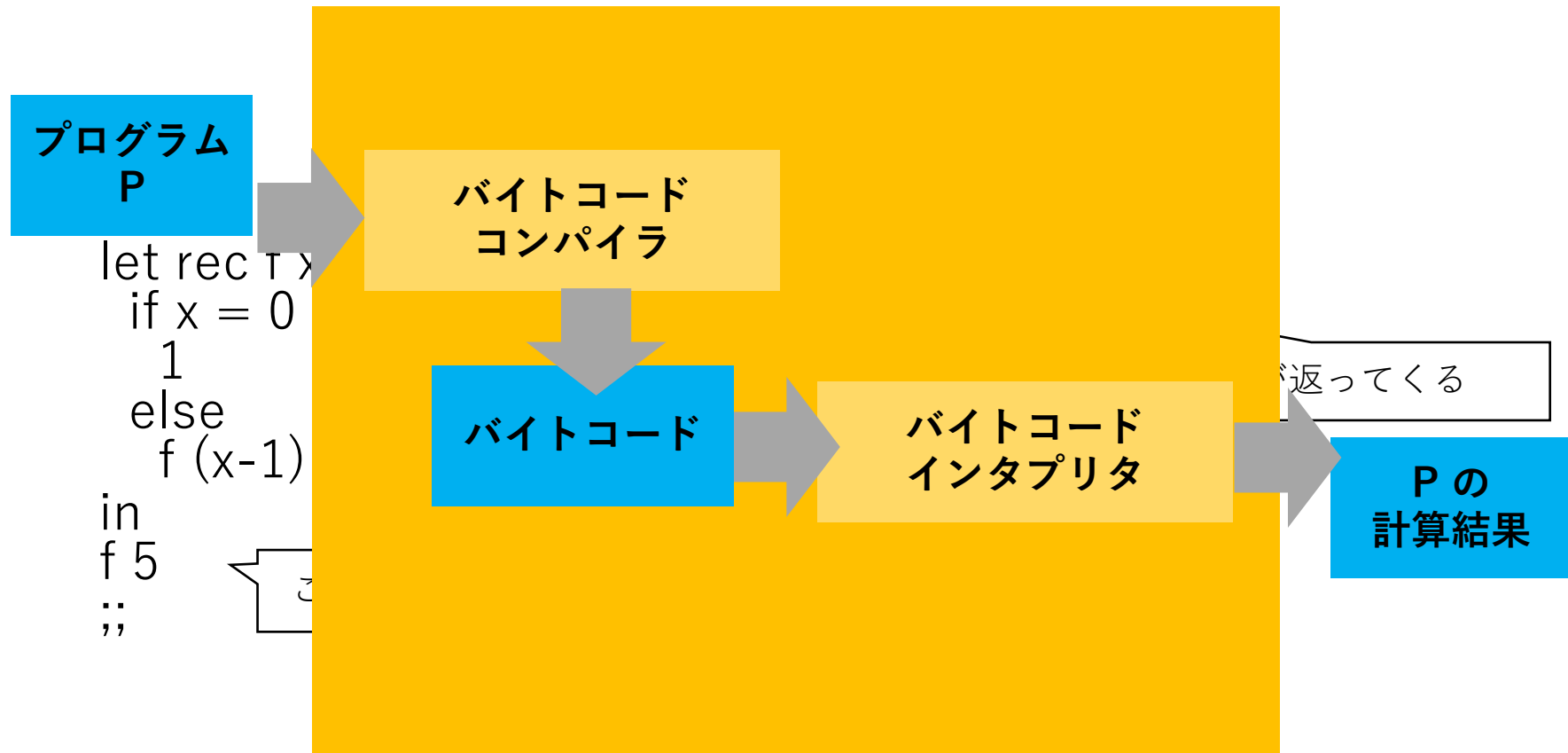




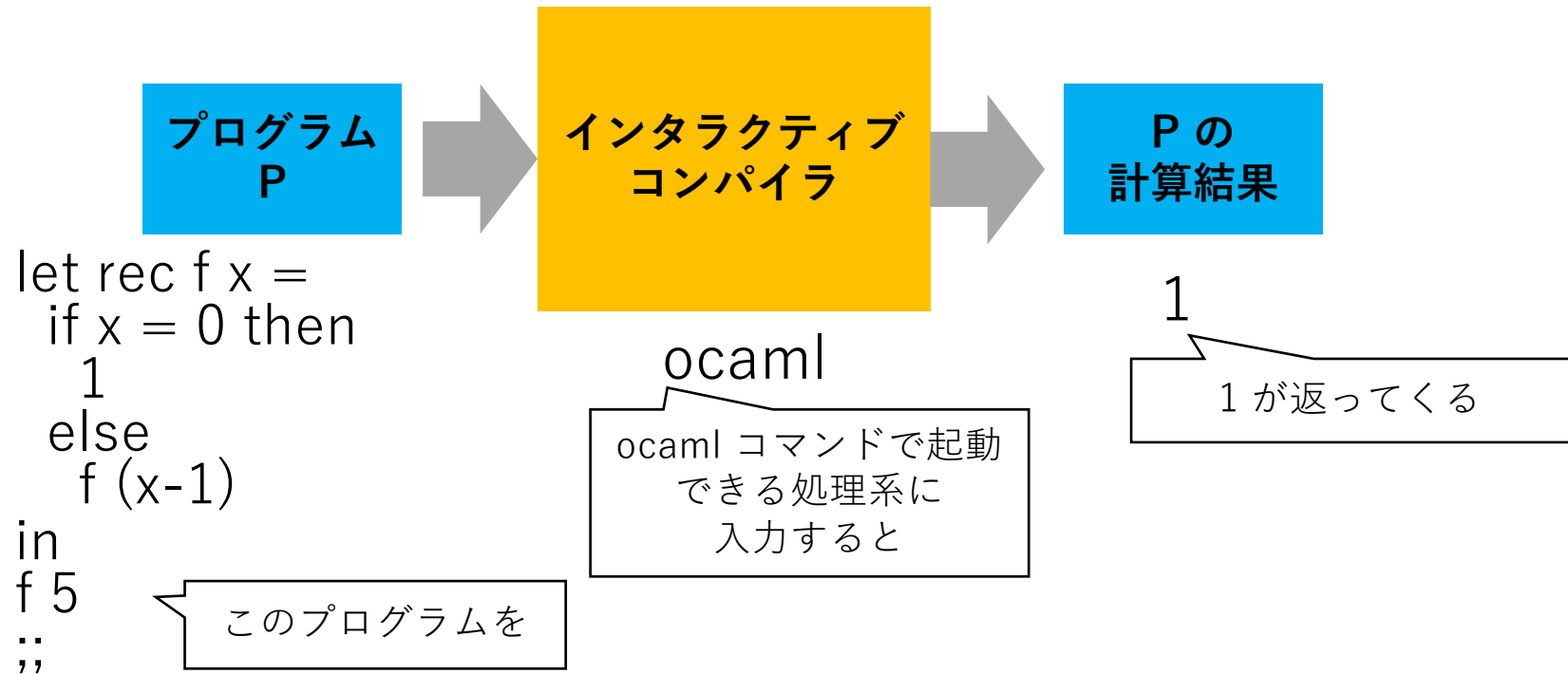
# 実装方式がこの2つのみというわけではない



# 実装方式がこの2つのみというわけではない



# 実装方式がこの2つのみというわけではない



# 注意すべきこと

- プログラミング言語の実装方式は必ずインタプリタとコンパイラの2つに分類できる**わけではない**
  - 内部的にコンパイラを呼び出すインタプリタもある
- ある特定の言語の実装方式がインタプリタかコンパイラかに決まっている**わけではない**
  - なので、プログラミング言語を「インタプリタ言語」か「コンパイラ言語」かに分類できる**わけではない**

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- **典型的なインタプリタの構成**
- 典型的なコンパイラの構成
- 本科目全体の構成

# インタプリタ

- 入力：プログラム P  
出力：P の計算結果

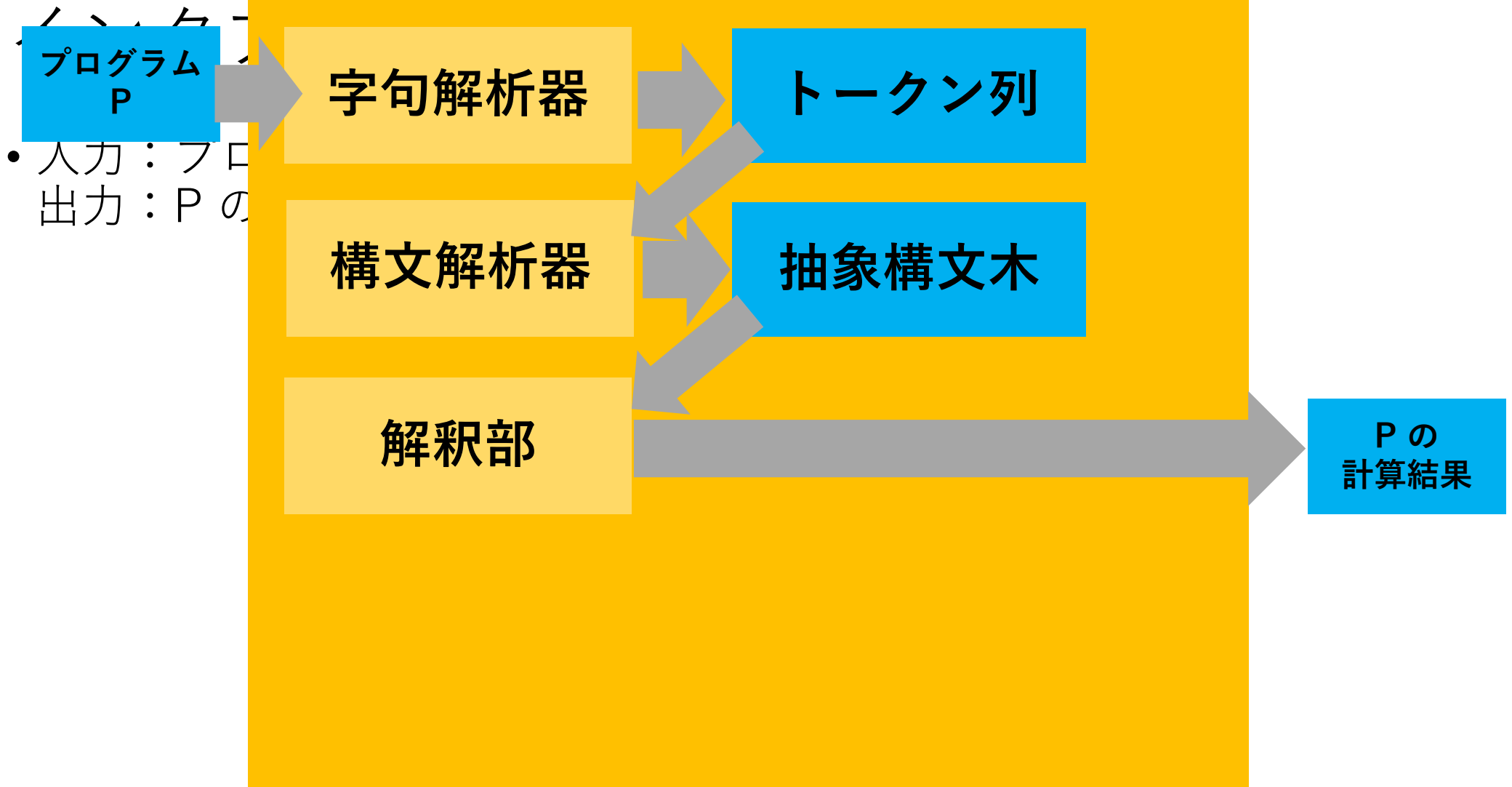


プログラム  
P

- 入力：フロ  
出力：P の

インタプリタ

P の  
計算結果



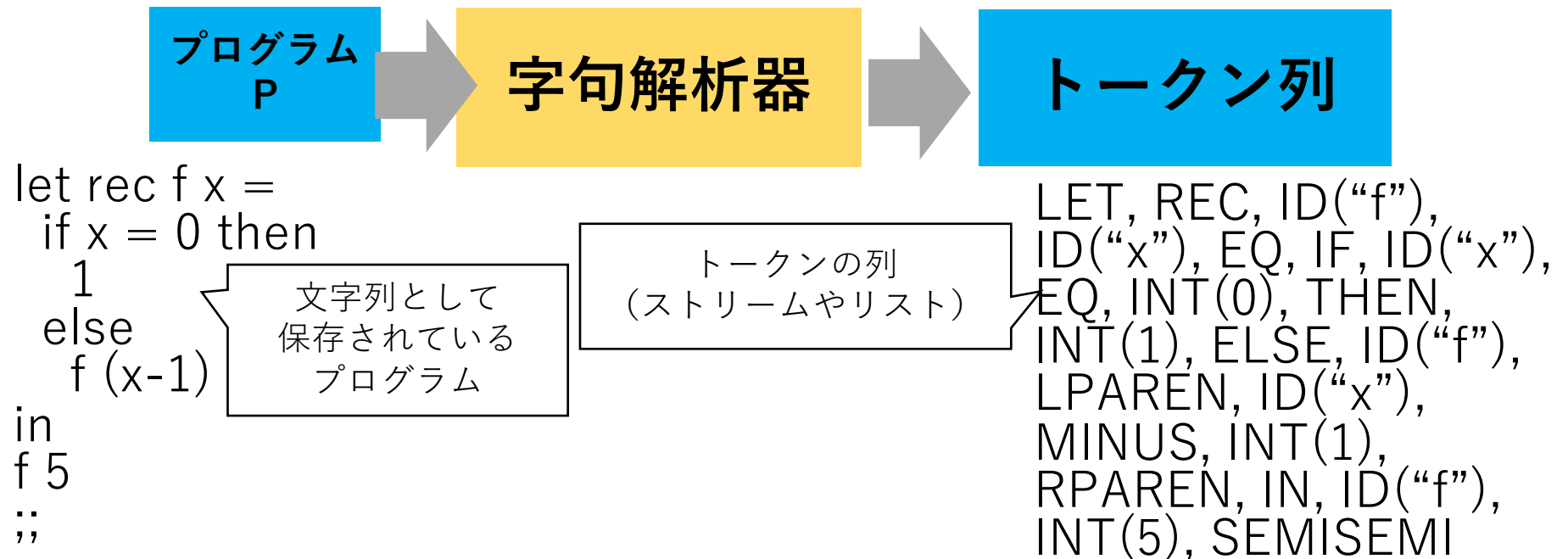


# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- **典型的なインタプリタの構成**
  - **字句解析器**
  - 構文解析器
  - 解釈部
  - 形式検証
- 典型的なコンパイラの構成
- 本科目全体の構成

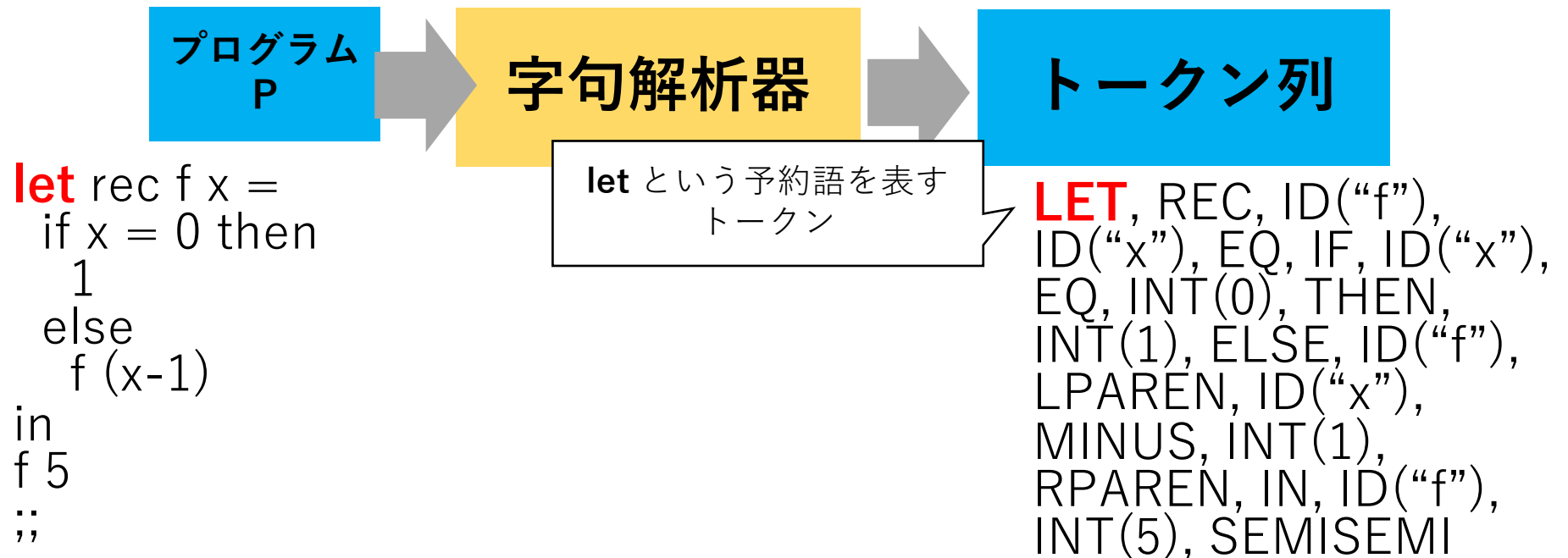
# 字句解析器 (lexer)

- 入力: プログラム P を表す文字列  
出力: 入力文字列を単語 (**トークン**) に区切って得られる列



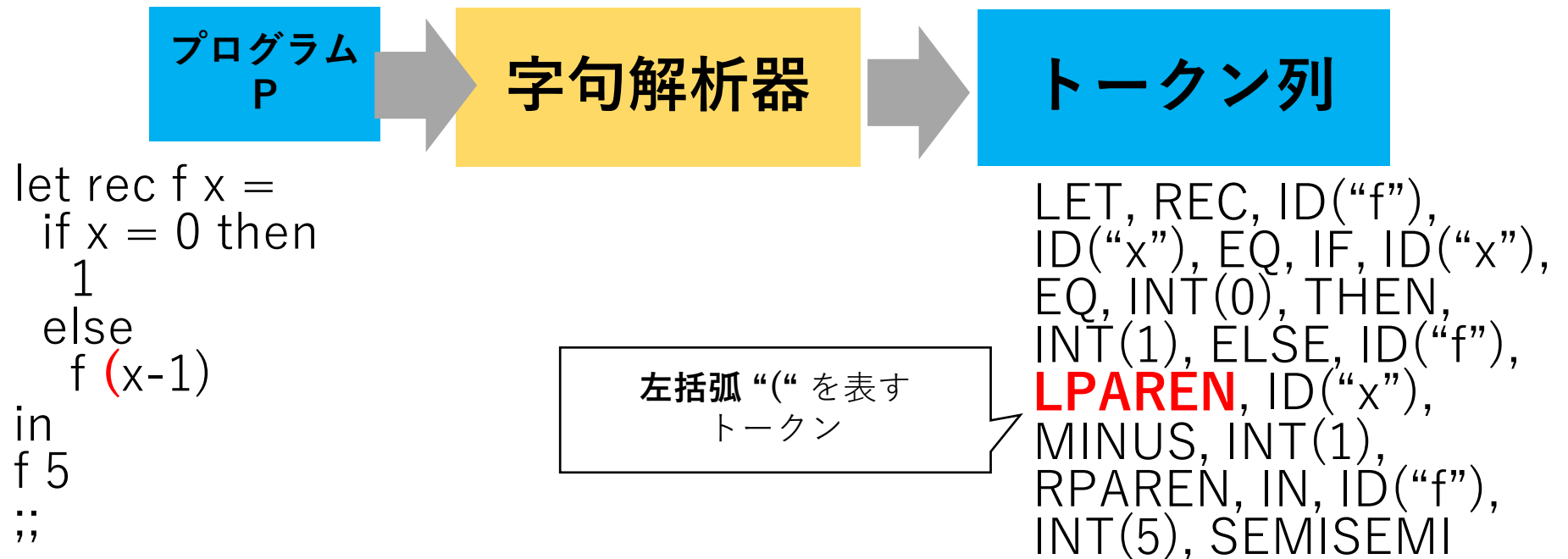
# 字句解析器 (lexer)

- 入力: プログラム P を表す文字列  
出力: 入力文字列を単語 (**トークン**) に区切って得られる列



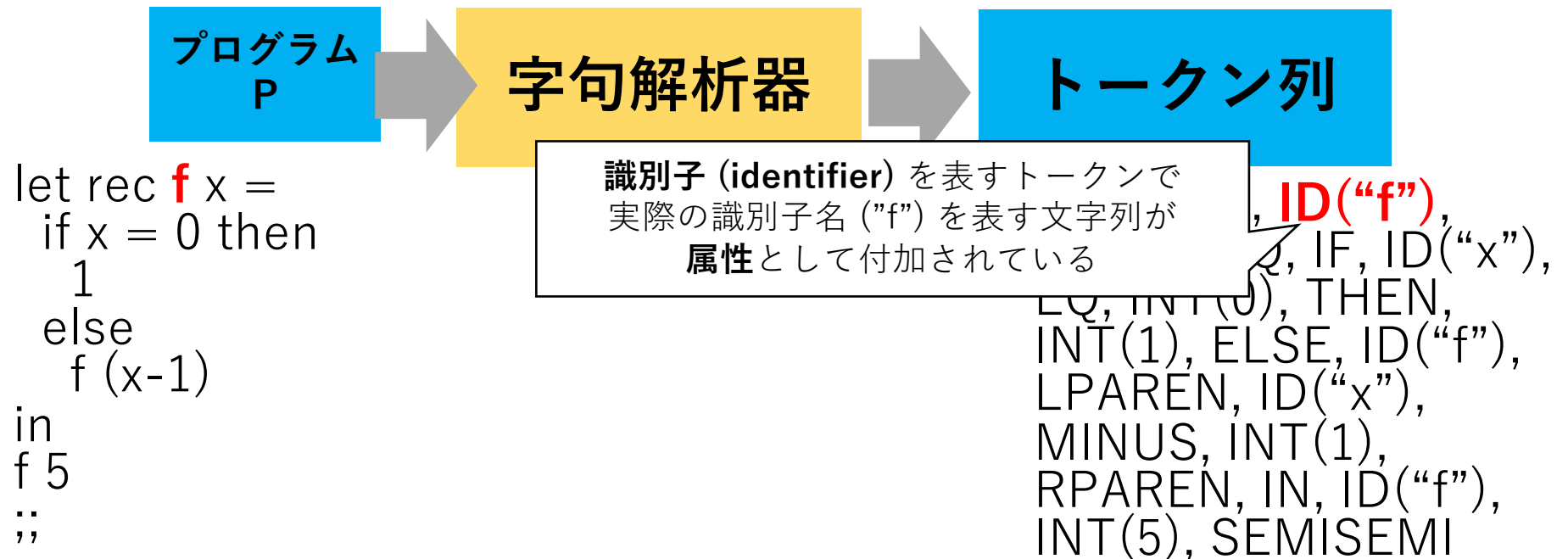
# 字句解析器 (lexer)

- 入力: プログラム P を表す文字列  
出力: 入力文字列を単語 (**トークン**) に区切って得られる列



# 字句解析器 (lexer)

- 入力: プログラム P を表す文字列  
出力: 入力文字列を単語 (**トークン**) に区切って得られる列



# 字句解析器の作り方

- 入力文字列を 1 文字ずつ読みながら、特定のトークンにマッチする文字列が見つかった場合は、そのトークンを出力列の末尾に追加
  - 例: "let rec f x = ..." の場合, "let " まで読んだところで LET を追加
  - 例: "hoge + fuga" の場合 "hoge " まで読んだところで ID("hoge") を, "+" を読んだところで PLUS を, "fuga" を読んだところで ID("fuga") を追加
- **Lexer generator** を使うともう少し容易に作れる
  - トークンと、そのトークンになるべき文字列集合を表す正則表現の対応関係を指定して、字句解析器を自動生成してくれるツール
    - lex, flex, ocamllex, ...
  - 例:  
"let" → LET    "rec" → REC    "=" → EQ  
[a-z][a-zA-Z0-9]\* → ID(この文字列)  
という内容が書かれたファイルから、上記の lexer が自動生成される

# 字句解析器の作り方

- 入力文字列を 1 文字ずつ読みながら、特定のトークンにマッチする文字列が見つかった場合は、そのトークンを出力列の末尾に追加
  - 例: "let rec f x = ..." の場合, "let " まで読んだところで LET を追加
  - 例: "hoge + fuga" の場合 "hoge " まで読んだところで ID("hoge") を, "+" を読んだところで PLUS を, "fuga" を読んだところで ID("fuga") を追加
- **Lexer generator** を使うともう少し容易に作れる
  - トークンと、講義3回目くらい?で出てくる列集合を表す正則表現の対応関係を指定して、ツール
    - lex, flex, ocamllex, ...
  - 例: 識別子になりうる名前を表す正則表現 "let" "=" → EQ  
[a-z][a-zA-Z0-9]\* → ID(この文字列)  
という内容が書かれたファイルから、上記の lexer が自動生成される

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- **典型的なインタプリタの構成**
  - 字句解析器
  - **構文解析器**
  - 解釈部
  - 形式検証
- 典型的なコンパイラの構成
- 本科目全体の構成



# 構文解析器(parser)

トークン列の文法構造を表現する  
木構造データ

- 入力: トークン列  
出力: 抽象構文木 (abstract syntax tree; AST)

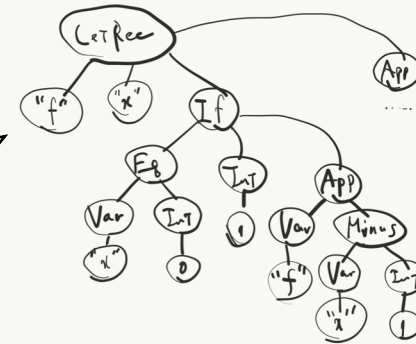
トークン列

構文解析器

抽象構文木

LET, REC, ID("f"),  
ID("x"), EQ, IF, ID("x"),  
EQ, INT(0), THEN,  
INT(1), ELSE, ID("f"),  
LPAREN, ID("x"),  
MINUS, INT(1),  
RPAREN, IN, ID("f"),  
INT(5), SEMISEMI

文法構造に沿った  
木構造

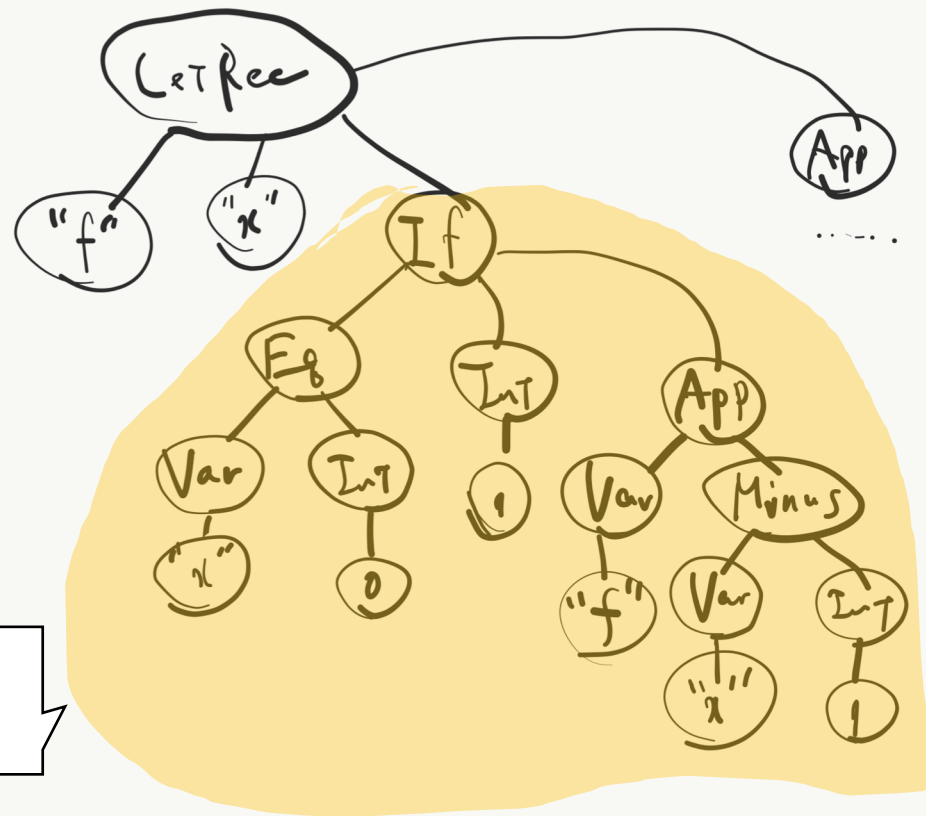


# 構文解析器

- 入力: トークン
- 出力: 抽象構文

## トークン

LET, REC, ID("f")  
ID("x"), EQ, IF, ID  
EQ, INT(0), THEN  
INT(1), ELSE, ID(  
LPAREN, ID("x"),  
MINUS, INT(1),  
RPAREN, IN, ID("  
INT(5), SEMISEM

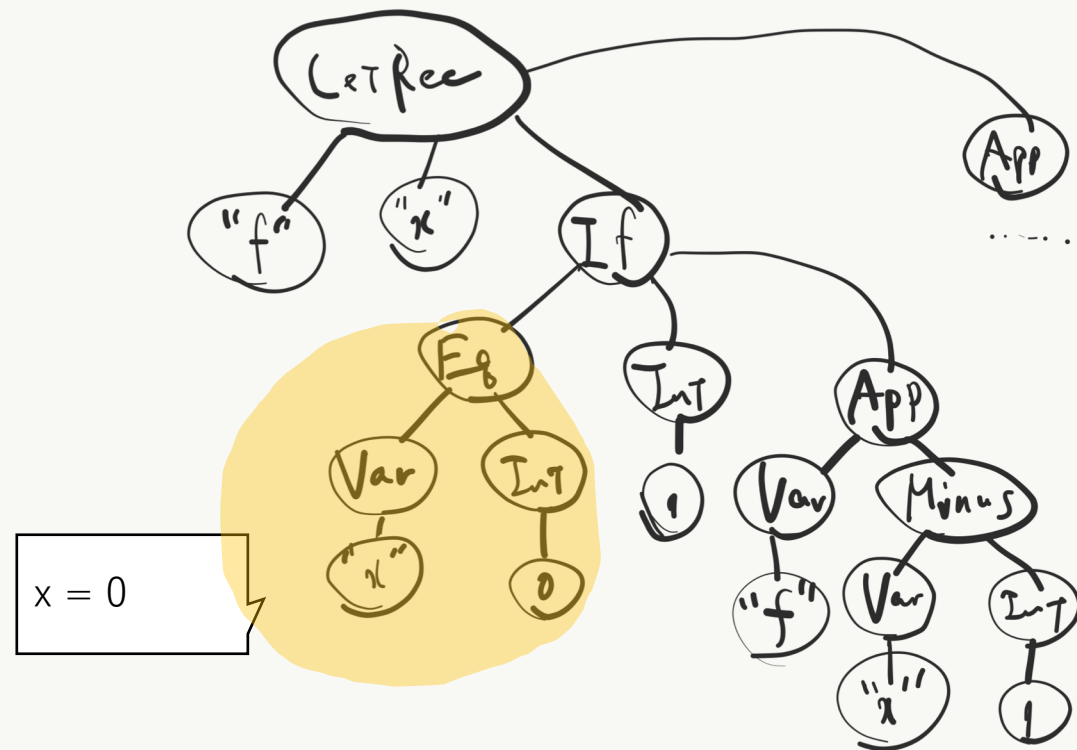


# 構文解析器

- 入力: トークン
- 出力: 抽象構文

## トークン

LET, REC, ID("f")  
ID("x"), EQ, IF, ID  
EQ, INT(0), THEN  
INT(1), ELSE, ID(  
LPAREN, ID("x"),  
MINUS, INT(1),  
RPAREN, IN, ID("  
INT(5), SEMISEM

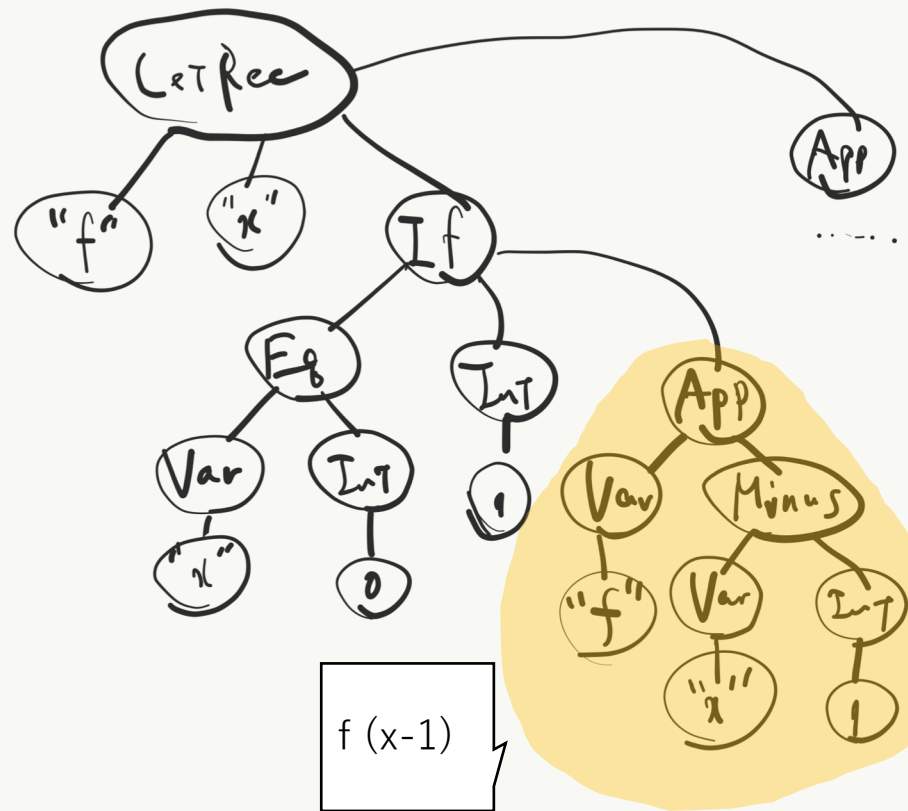


# 構文解析器

- 入力: トークン
- 出力: 抽象構文

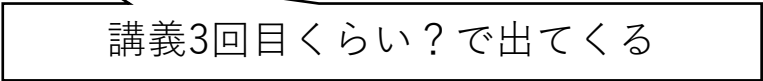
## トークン

LET, REC, ID("f")  
ID("x"), EQ, IF, ID  
EQ, INT(0), THEN  
INT(1), ELSE, ID(  
LPAREN, ID("x"),  
MINUS, INT(1),  
RPAREN, IN, ID("  
INT(5), SEMISEM



# 構文解析器の作り方

- 構文解析用のよく知られたアルゴリズムを実装する
  - CKY, LL(k)構文解析, LR(k)構文解析...
  - 講義の終盤で説明
- Parser generator を使うともう少し簡単
  - 文脈自由文法を与えると, その文法のための parser を自動生成
    - yacc, bison, menhir...



講義3回目くらい?で出てくる

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- **典型的なインタプリタの構成**
  - 字句解析器
  - 構文解析器
  - **解釈部**
  - 形式検証
- 典型的なコンパイラの構成
- 本科目全体の構成

# 解釈部

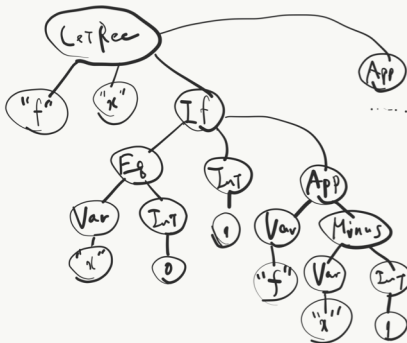
- 入力: 抽象構文木  
出力: この木が表すプログラムの計算結果

抽象構文木

構文解析器

計算結果

1



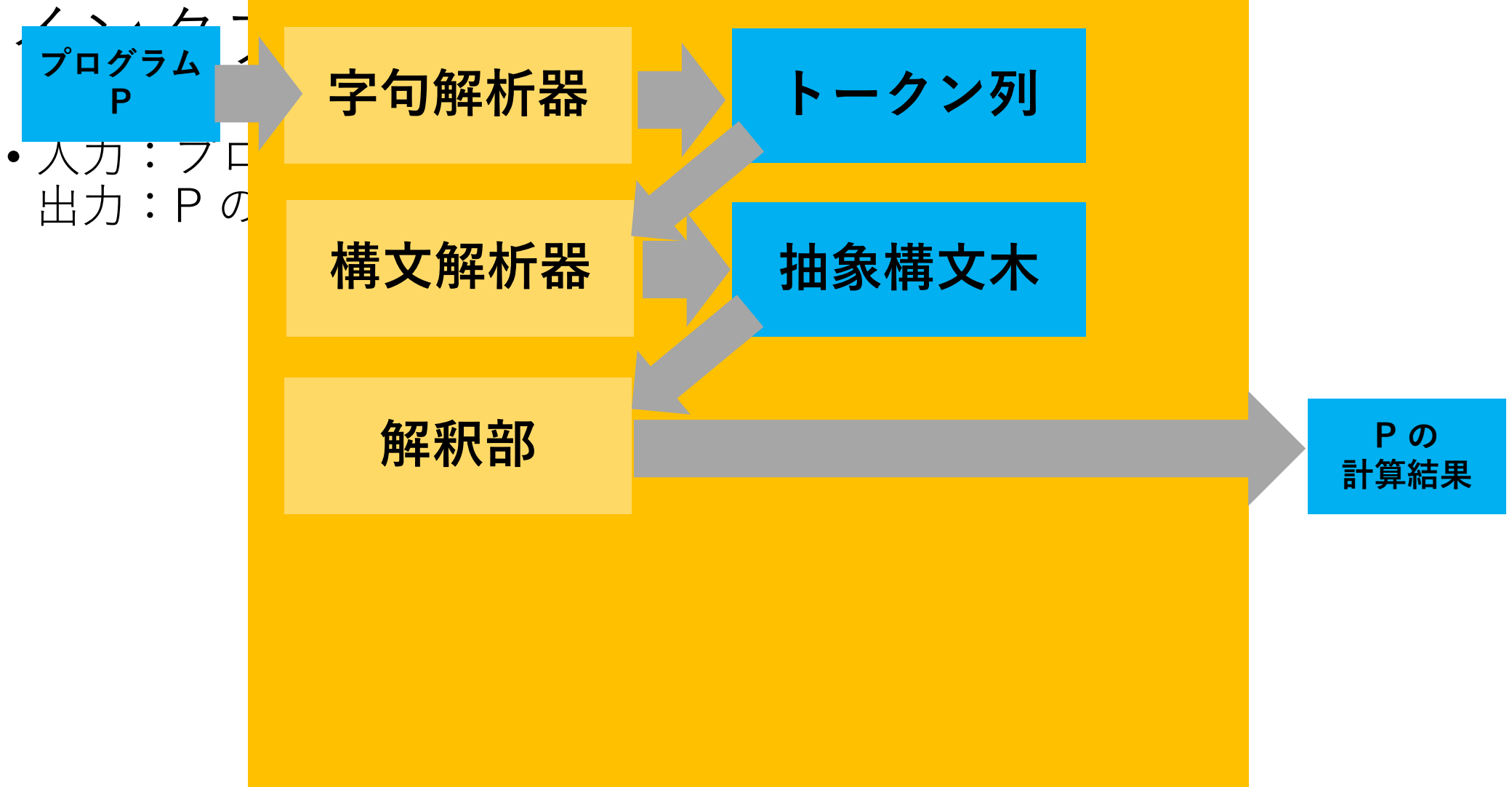
# 解釈部の作り方

- 基本的には抽象構文木を再帰的にたどりつつ、プログラミング言語の意味論に従って**評価 (evaluation)**を行うだけ
- この講義で扱う重要なトピック
  - 変数と値の対応をどう管理するか
  - (再帰) 関数を表す値をどのように表現するか

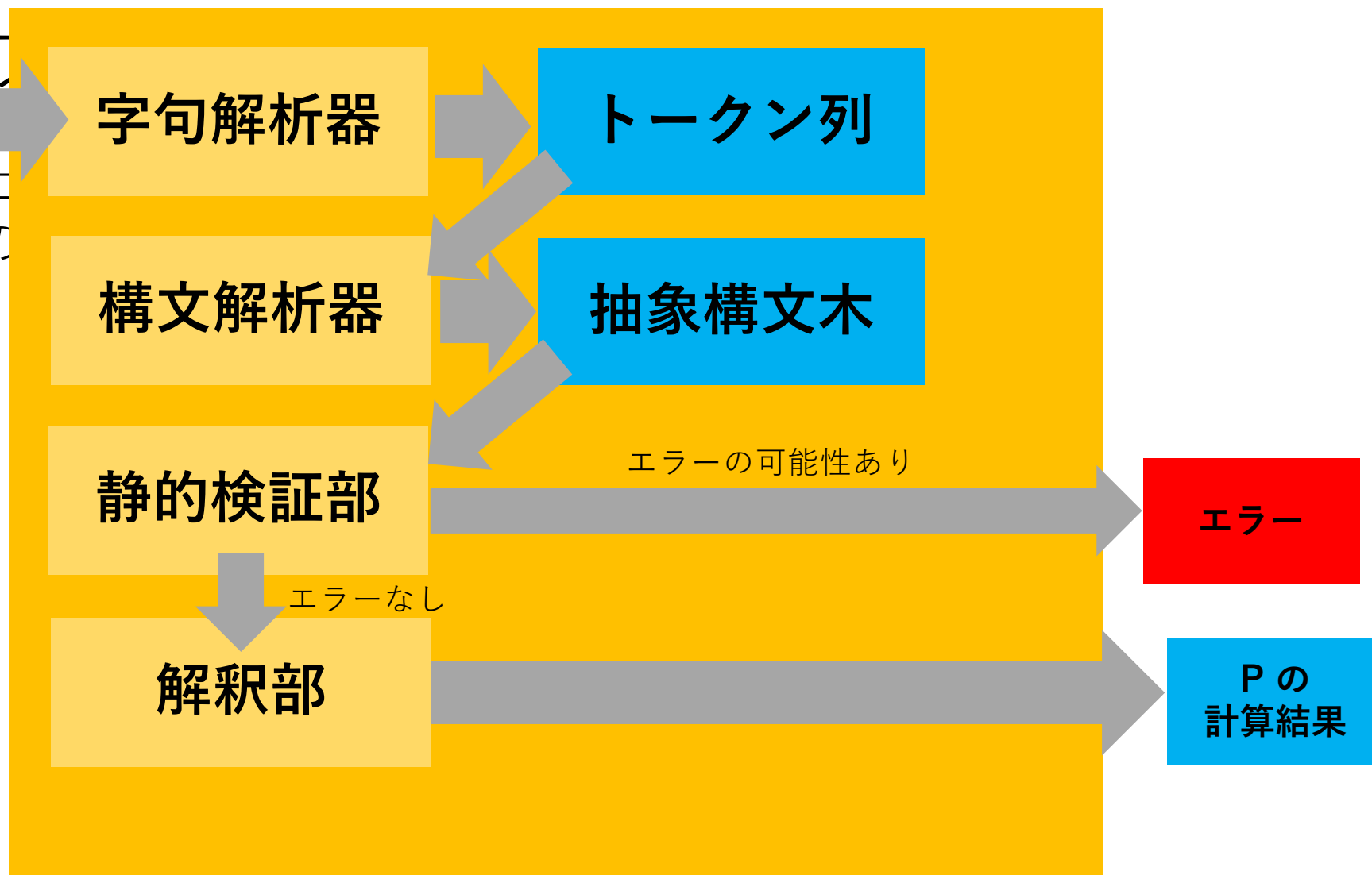


# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- **典型的なインタプリタの構成**
  - 字句解析器
  - 構文解析器
  - 解釈部
- **形式検証**
- 典型的なコンパイラの構成
- 本科目全体の構成



- プログラムの静的解析
- 入力：プログラム P
  - 出力：P の静的解析結果



# 静的検証部

- プログラムを実行すると起こりうるある種のエラーを、実行前に**形式検証 (formal verification)** によって検出
  - ある種のエラー: 実行時型エラー, Assertion failure, Null pointer dereference, 配列領域外アクセス, メモリリーク, 解放後のメモリ領域へのアクセス, 無限ループ (等の機能要件)
  - なぜ実行前に?:
    - 早目にバグが見つかるほうが安全
    - 重い計算を実行した後に結果を出力する前にエラーで落ちるとか最悪

# 静的検証部の作り方

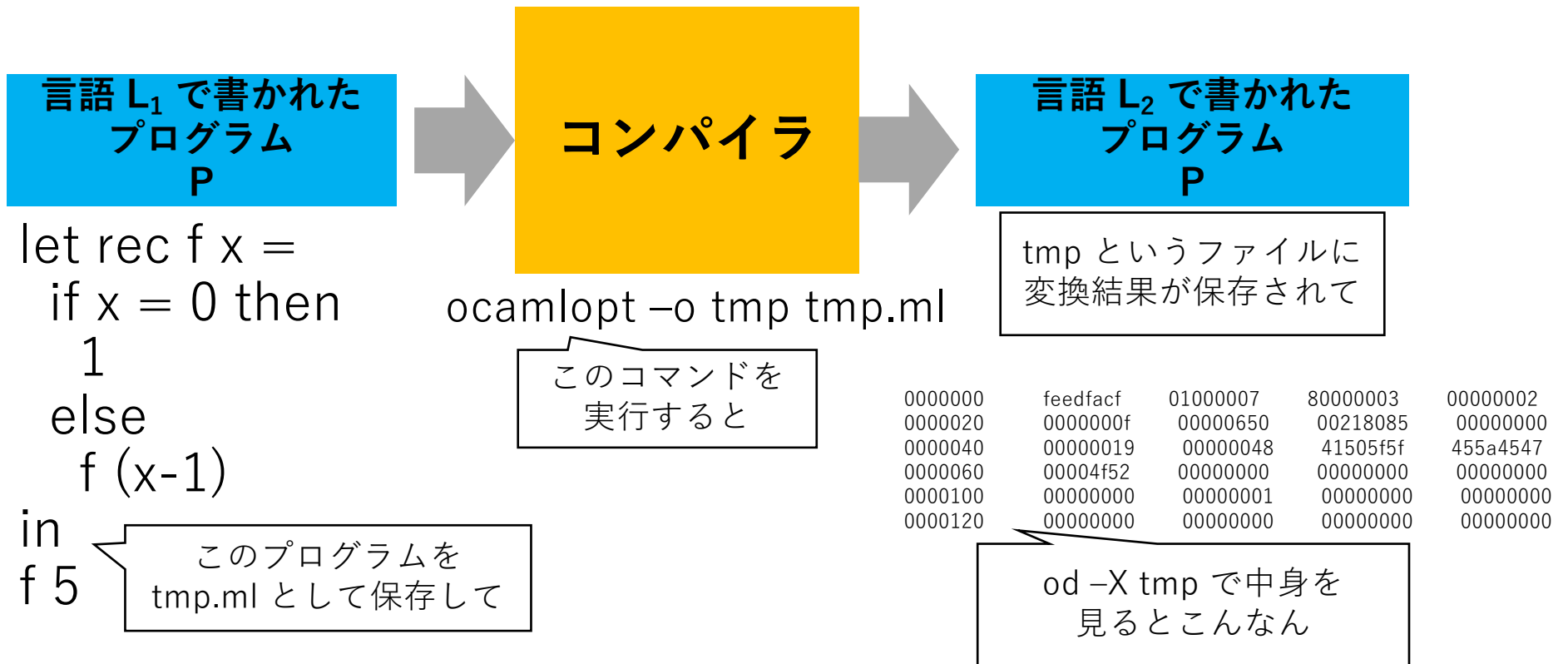
- 多くのツールや手法が存在
  - どのような言語で
  - どのような種類のエラーを
  - どのくらい効率的に
  - どのくらい網羅的に見つけたいかによる
- 例: (関数型) 言語で実行時型エラーを網羅的に見つけたい
  - 制約に基づく静的型検査・型推論 講義中盤でやる
- 例: C で Assertion failure を少し時間はかかっても網羅的に見つけたい
  - モデル検査
- 例: メモリリークをやや効率的に網羅的に見つけたい
  - [Kobayashi and Suenaga APLAS'09]
- 例: 深層学習モデルに対する adversarial example を網羅的に見つけたい
  - [Gehr et al., IEEE S&P'18], [Mirman et al., ICML'20], [Singh et al., POPL'19]

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- 典型的なインタプリタの構成
- **典型的なコンパイラの構成**
- 本科目全体の構成

# コンパイラ

- 入力：言語  $L_1$  で書かれたプログラム  $P$   
出力：言語  $L_2$  で書かれた  $P$  と等価なプログラム



コンパ

言語  $L_1$  で  
書かれた  
プログラム  
P

コンパイラ

言語  $L_2$  で  
書かれた  
プログラム  
P

18085	00000000
05f5f	455a4547
00000	00000000
00000	00000000
00000	00000000

if

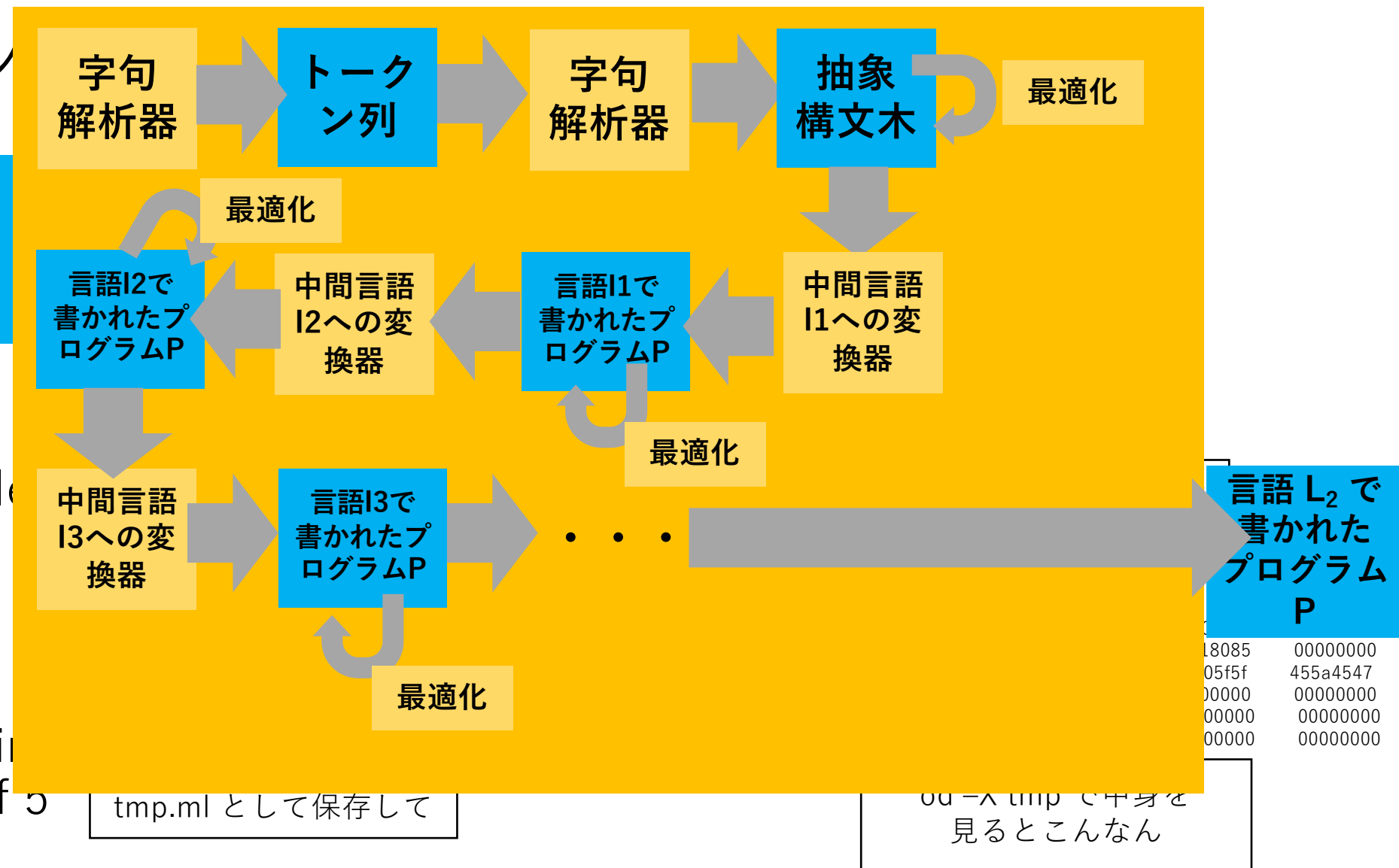
tmp.ml として保存して

od -x tmp で中身を見  
るとこんな



コン

言語  $L_1$  で  
書かれた  
プログラム  
P

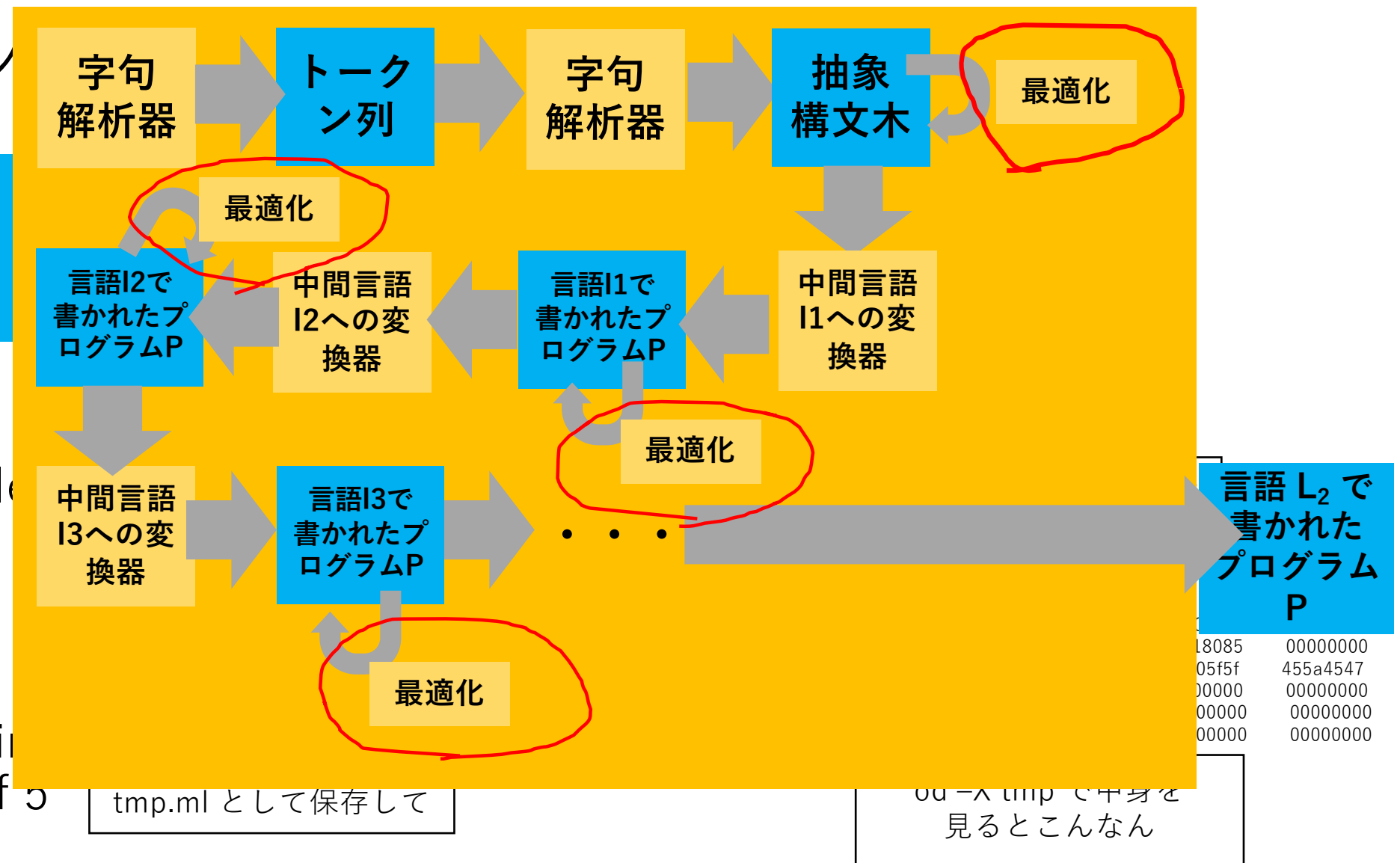


# コンパイラがやること

- **ソース言語 (source language)  $L_1$  から,**  
**いくつかの中間言語 (intermediate languages) を介して**  
**ターゲット言語 (target language)  $L_2$  に変換**
  - 中間言語を介するメリット
    - 通常高級言語である  $L_1$  と, 通常低級言語である  $L_2$  は見かけも文法も意味論もぜんぜん違うので, その差異を少しずつ埋めることで, 各変換を分かりやすくできる
    - 言語の抽象度に応じた最適化が可能
    - 異なる  $L_1$  と  $L_2$  のペアに対して, 中間言語や変換が再利用できる場合がある
      - 例: OCaml から IA-32 と MIPS のコードを出力したいときは, 途中までは同一にしておいて, 最後のアセンブリ生成のフェーズを変えれば実装可能

コン

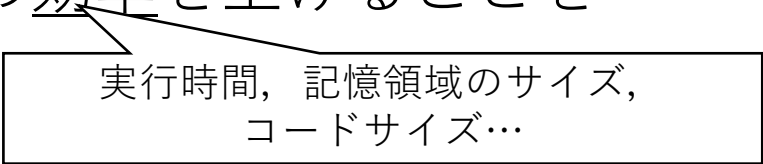
言語  $L_1$  で  
書かれた  
プログラム  
P



# 最適化

- 各中間言語で行われる, プログラムの効率を上げることを期待して行われる変換

- 定数畳み込み
- 不要な代入の除去
- 無用式の除去
- ループ融合/分解変換
- レジスタ割付
- ...



実行時間, 記憶領域のサイズ,  
コードサイズ...

# 今日の内容

- ~~布教活動~~イントロダクション
- シンタックスとセマンティクス
- プログラミング言語の実装方式
- 典型的なインタプリタの構成
- 典型的なコンパイラの構成
- **本科目全体の構成**

# 本科目の今後の予定

- OCaml の復習
- 型無し MiniML インタプリタの実装
- MiniML のための型推論アルゴリズムの実装
- MiniML コンパイラの実装
- 字句解析アルゴリズムと構文解析アルゴリズム