

# 関数・論理型プログラミング実験 第5回

江口 慎悟  
酒寄 健  
塚田 武志  
松下祐介

# 講義のサポートページ

<http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/>

- 講義資料等が用意される
- レポートの提出先
- 利用にはアカウントが必要
- 名前/学籍番号/希望アカウント名をメールを  
tsukada@kb.is.s.u-tokyo.ac.jp  
までメールしてください。
  - 件名は「FL/LP実験アカウント申請」
  - アカウント名/パスワードを返信
  - PCからのメールを受け取れるように

# インタプリタを作る（全5回）

## 第5回 基本的なインタプリタの作成

- 字句解析・構文解析、変数の扱い方

## 第6回 関数型言語への拡張

- 関数定義・呼び出し機構の作成

## 第7回 型システムと単純型推論

- 単純型検査器

## 第8回 単一化、let多相

- 単一化の定義とアルゴリズム、let多相

## 第9回 様々な拡張

- パターンマッチング

# 今日の内容

- インタプリタの概要
  - 字句解析
  - 構文解析
- OCaml での構文解析
  - ocamllex
  - ocaml yacc
- 簡単な評価器の作成

# インタプリタの概要

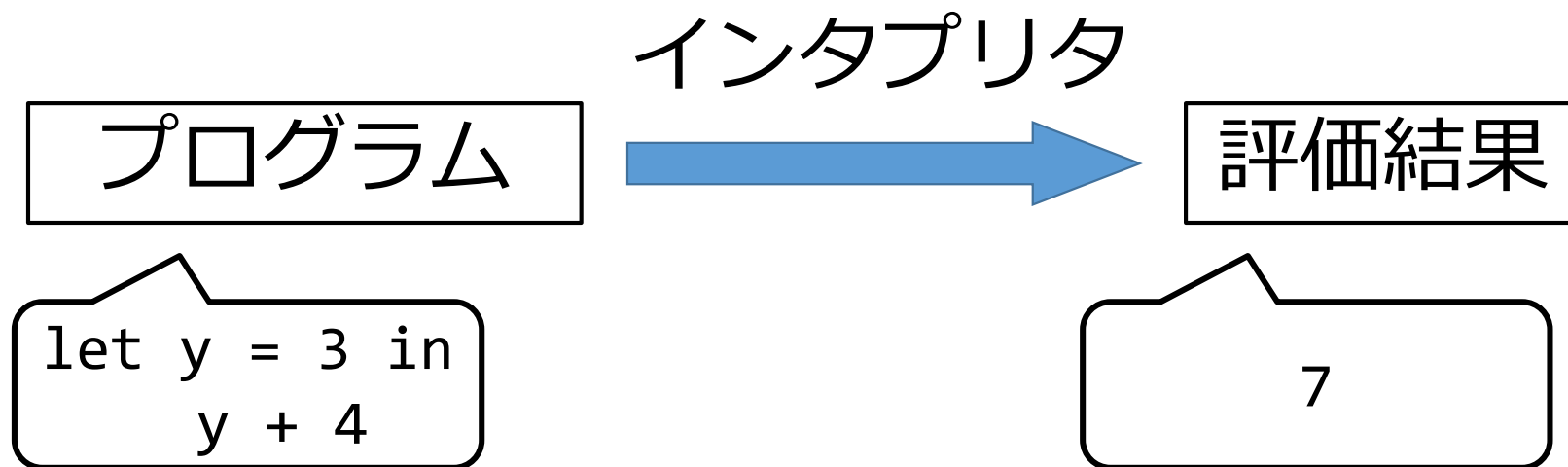
字句解析

構文解析

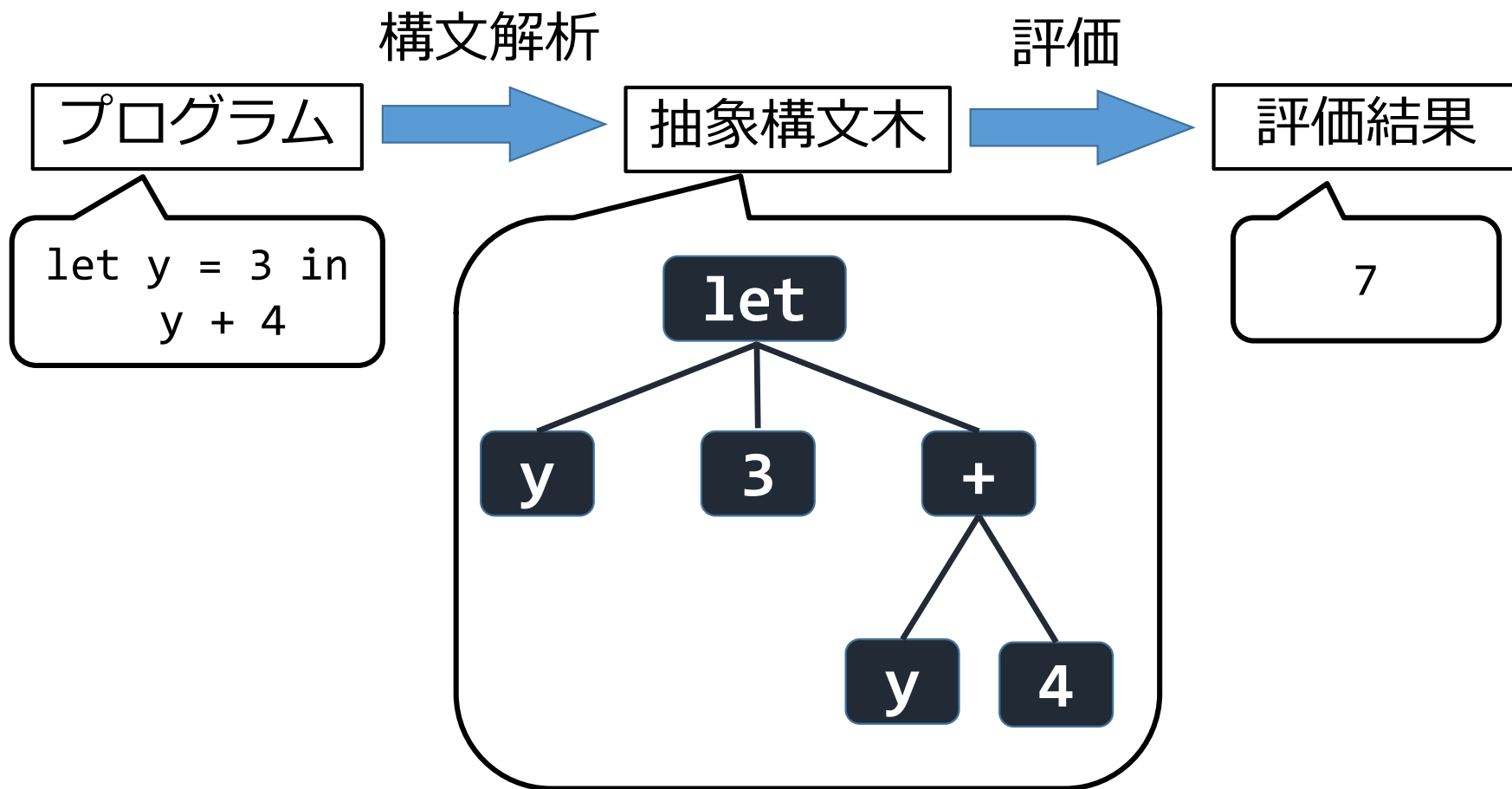
評価器

# インタプリタとは

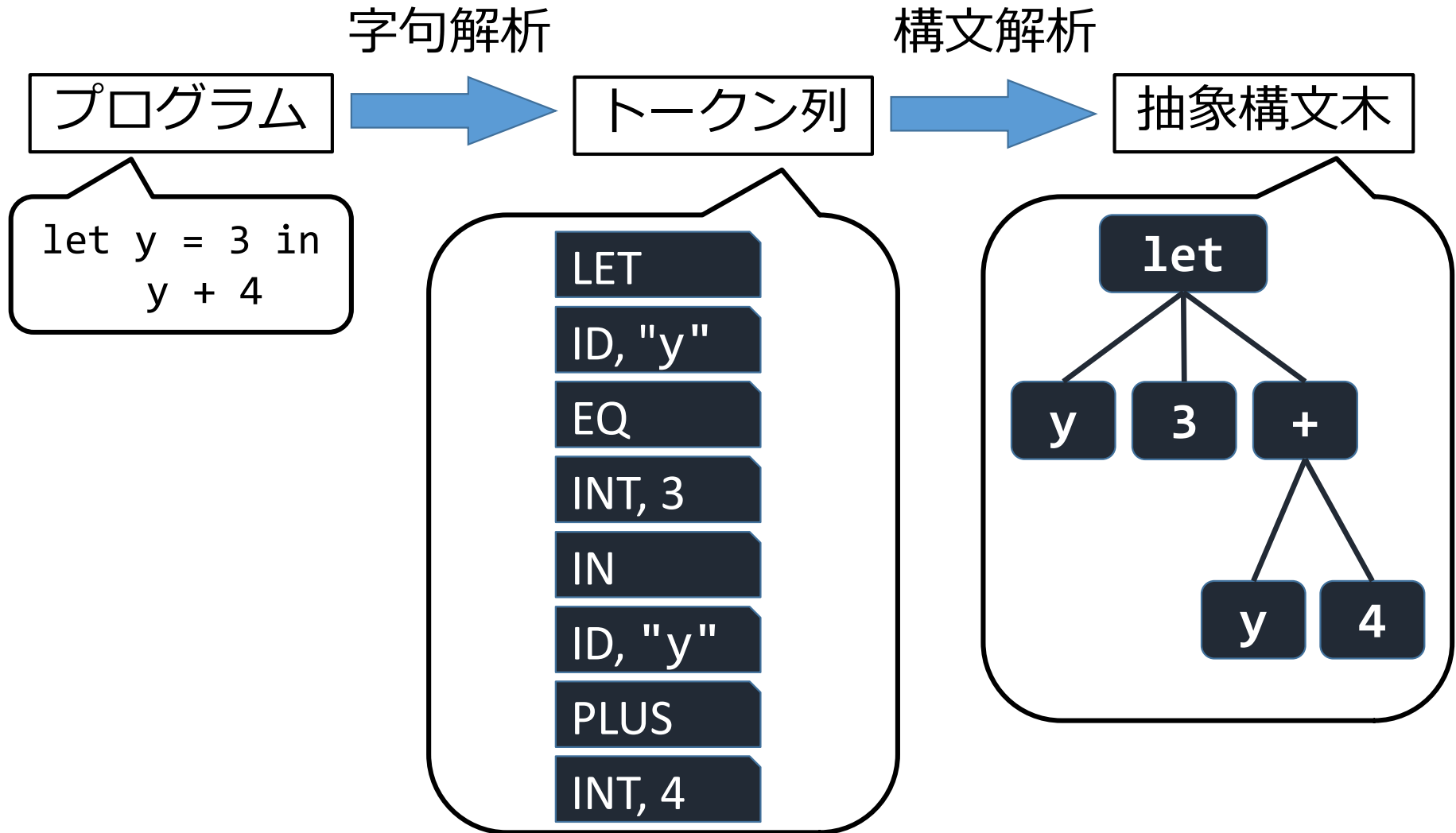
- 。プログラムを入力にとり  
その評価結果を返すプログラム



# 典型的なインタプリタの構造



# 典型的な抽象構文木の作り方 (~~作り方は~~ いろいろある.)





# インタプリタの概要

字句解析

構文解析

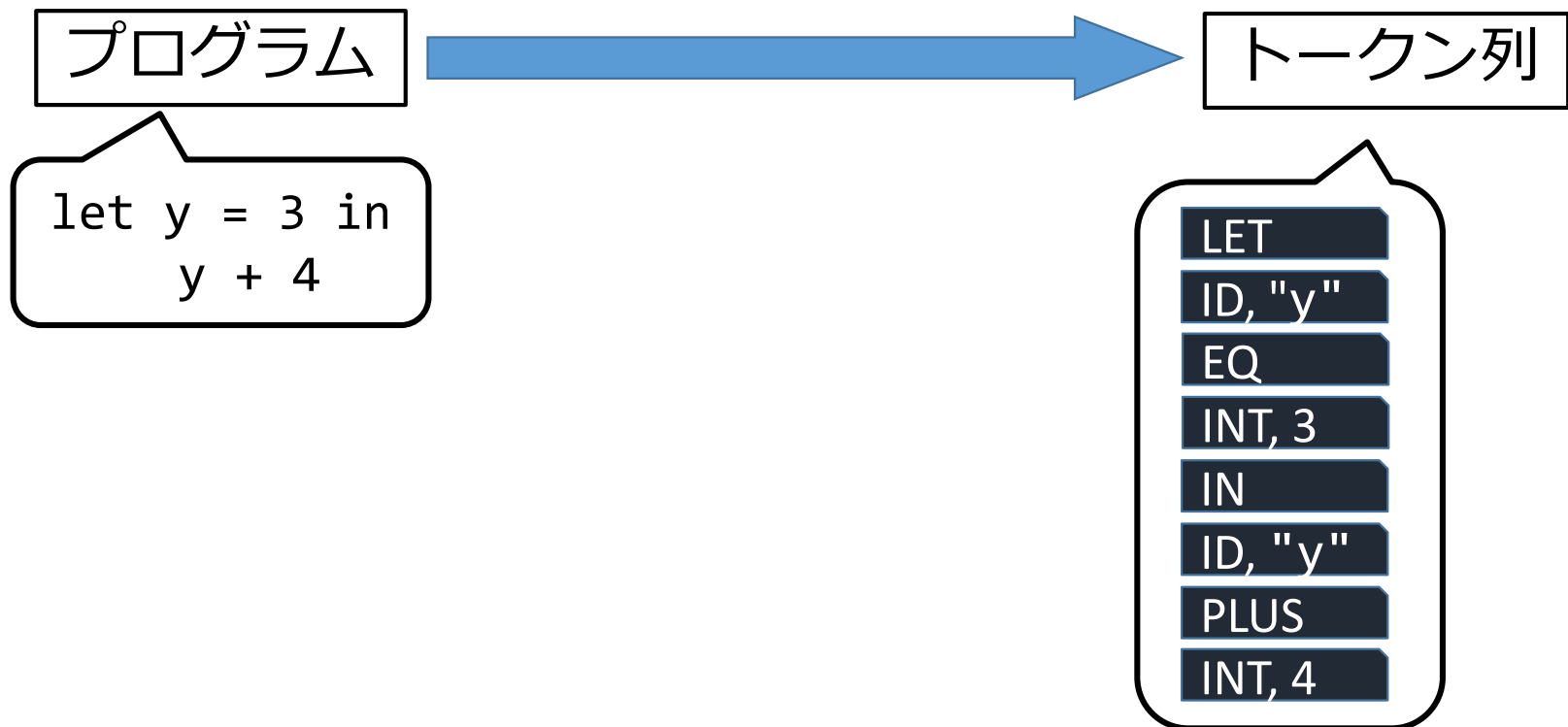
評価器

# 字句解析 (lexing)

○ 文字列をまとまり毎に切り分ける

■ **トークン**：切り分けられた一つ一つ

(e.g. キーワード、識別子、数字、...)



# インタプリタの概要

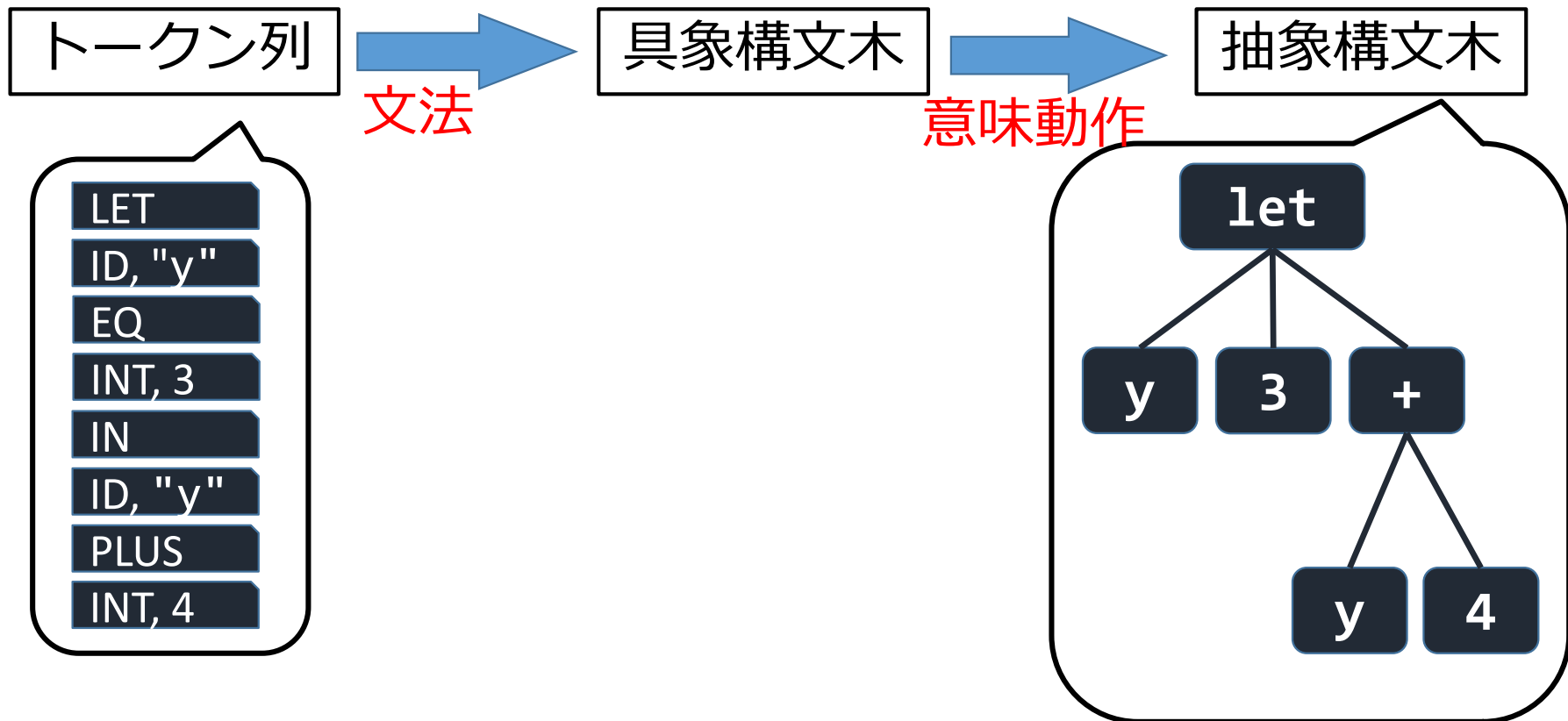
字句解析

構文解析

評価器

# 構文解析 (parsing)

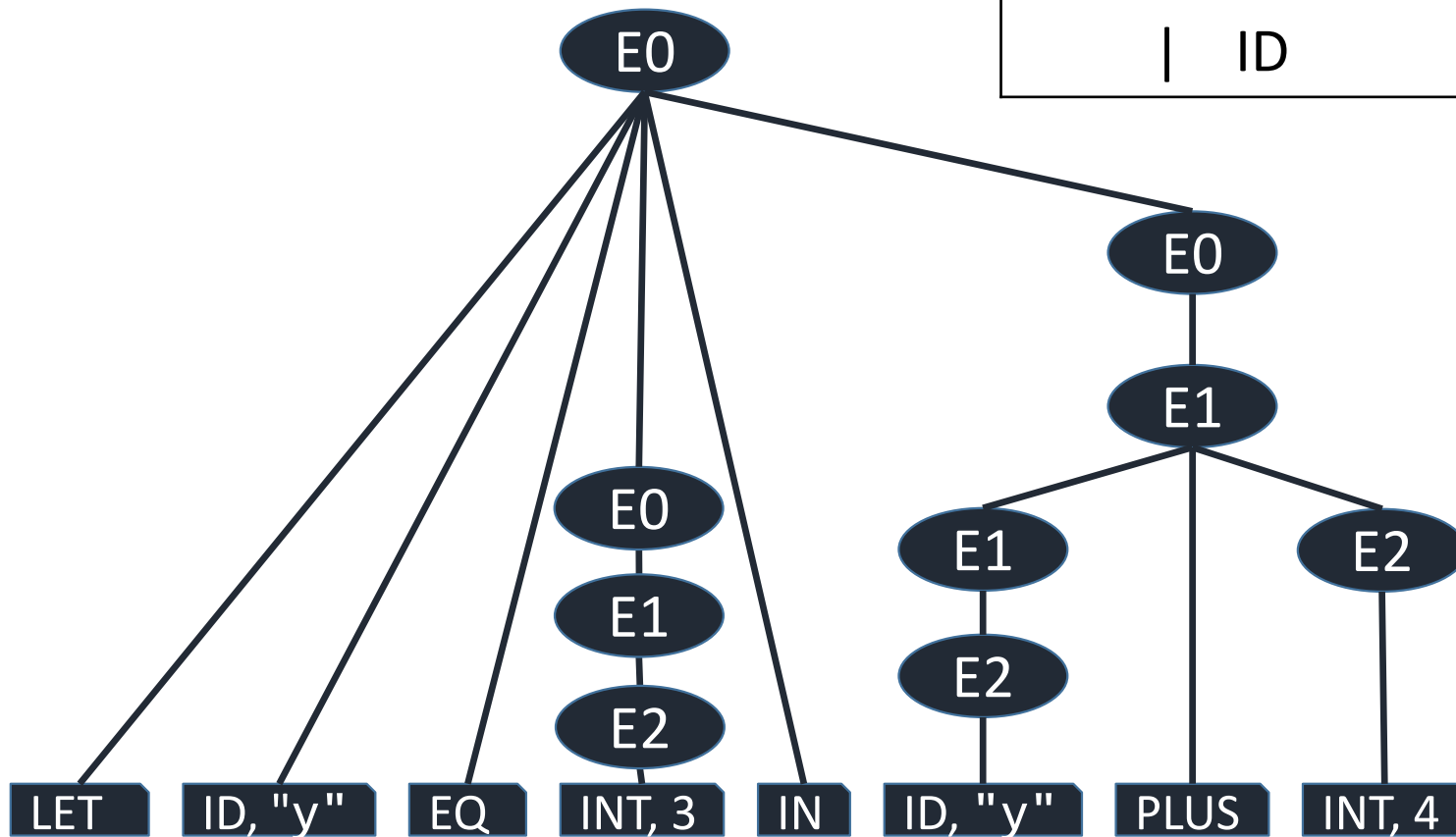
- トークン列を抽象構文木に変換する



# 具象構文木

文法自由文法の構文木

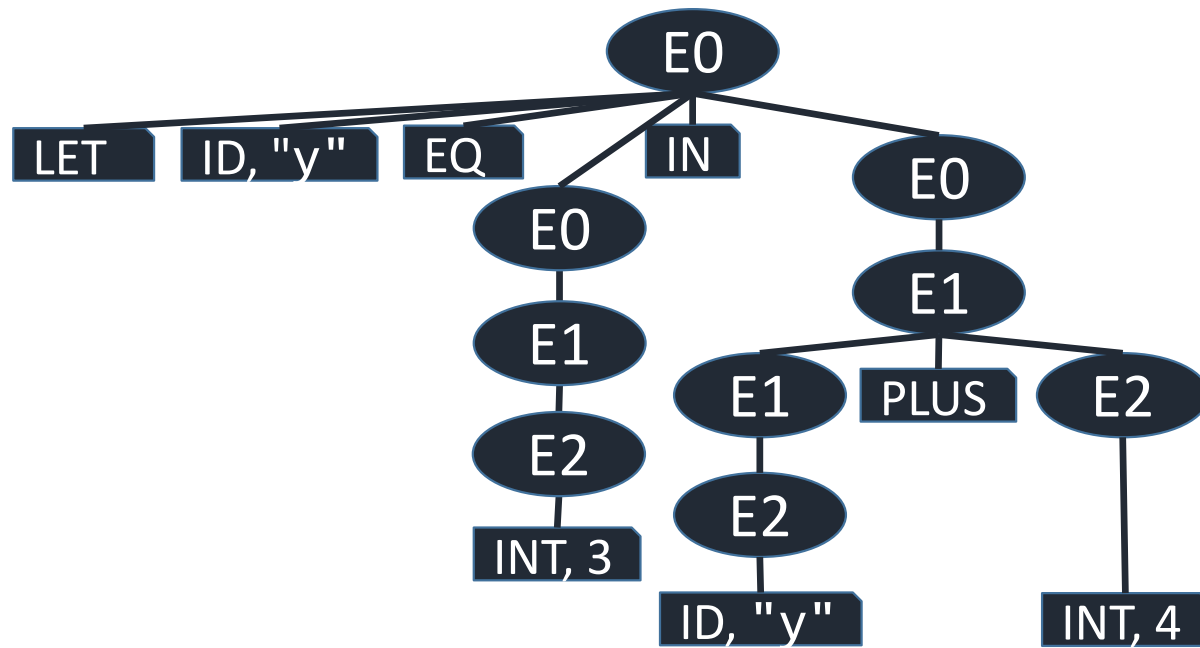
○ 文法の導出木のこと

$$\begin{aligned} E0 &\rightarrow \text{LET ID EQ } E0 \text{ IN } E0 \\ &\quad | \quad E1 \\ E1 &\rightarrow E1 \text{ PLUS } E2 \\ &\quad | \quad E2 \\ E2 &\rightarrow \text{INT} \\ &\quad | \quad \text{ID} \end{aligned}$$


# 意味規則

○ 具象構文木に応じて抽象構文木を生成

$E0 \rightarrow \text{LET } \underline{ID} \text{ EQ } \underline{E0} \text{ IN } \underline{E0} \quad \{ \text{ELet } (\$2, \$4, \$6) \}$   
|  $E1 \quad \{ \$1 \}$   
 $E1 \rightarrow E1 \text{ PLUS } E2 \quad \{ \text{EAdd } (\$1, \$3) \}$   
|  $E2 \quad \{ \$1 \}$   
 $E2 \rightarrow \text{INT} \quad \{ \text{EConstInt } (\$1) \}$   
|  $ID \quad \{ \text{EVar } (\$1) \}$

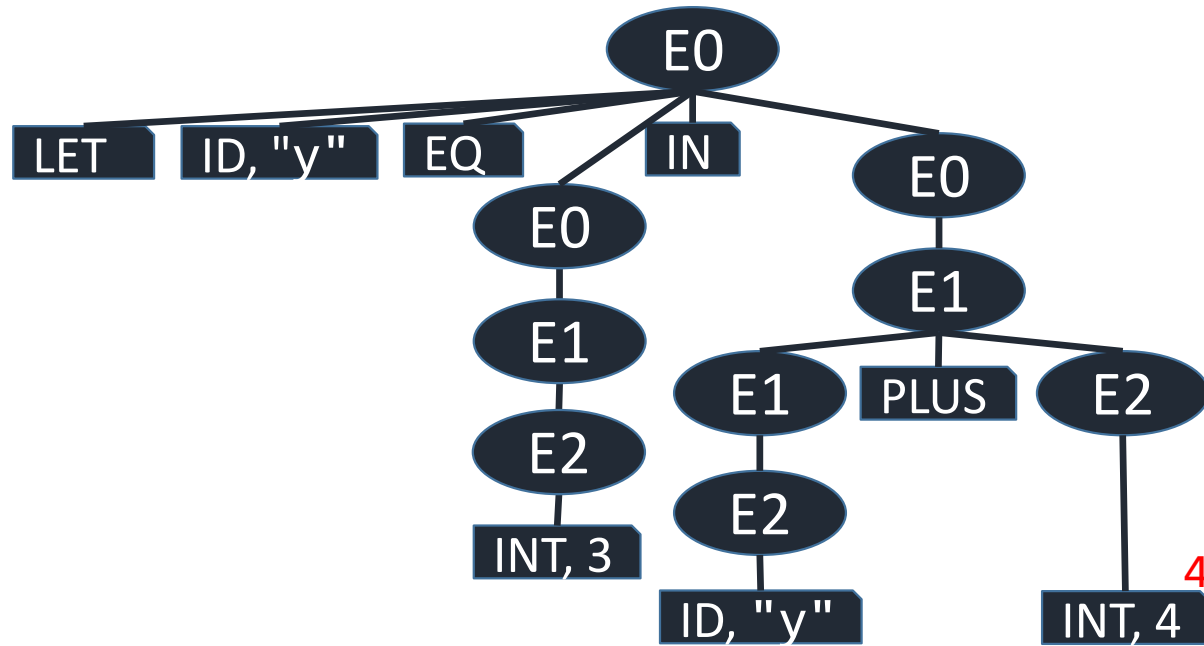


この規則から、

# 意味規則

。具象構文木に応じて抽象構文木を生成

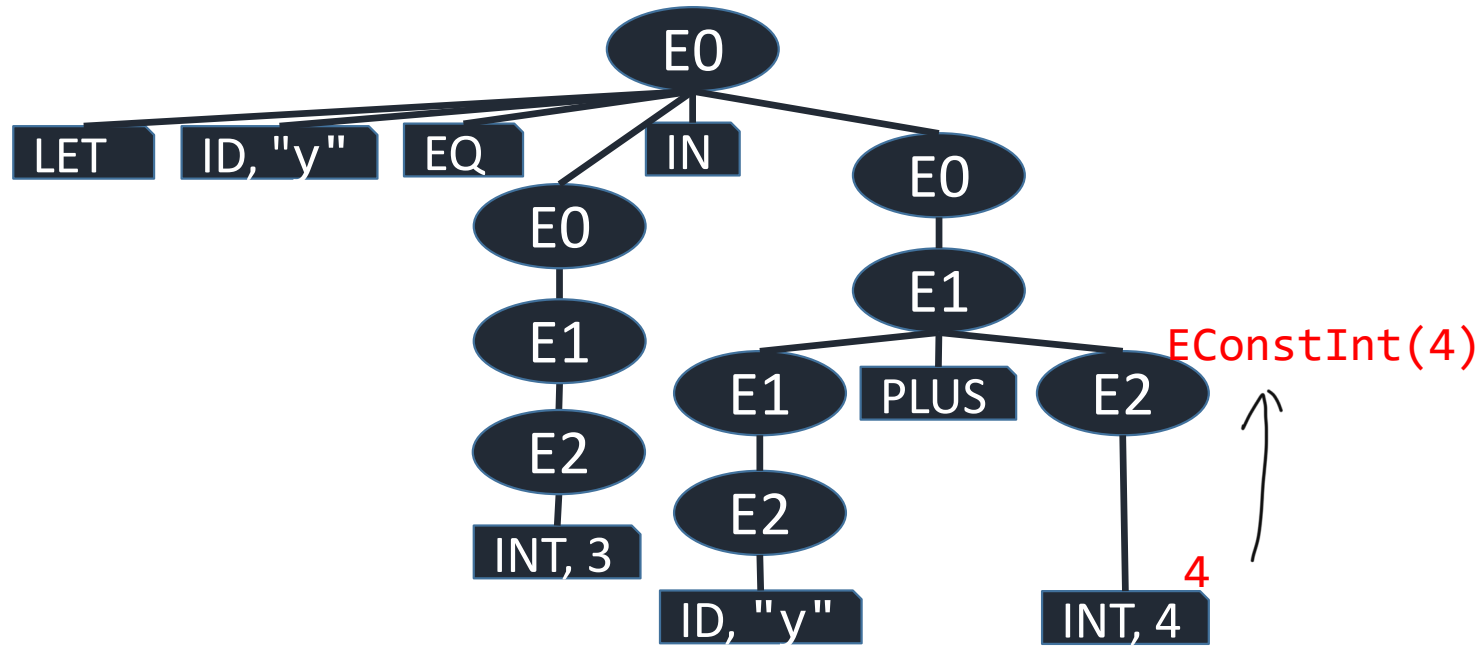
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 意味規則

。具象構文木に応じて抽象構文木を生成

$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }

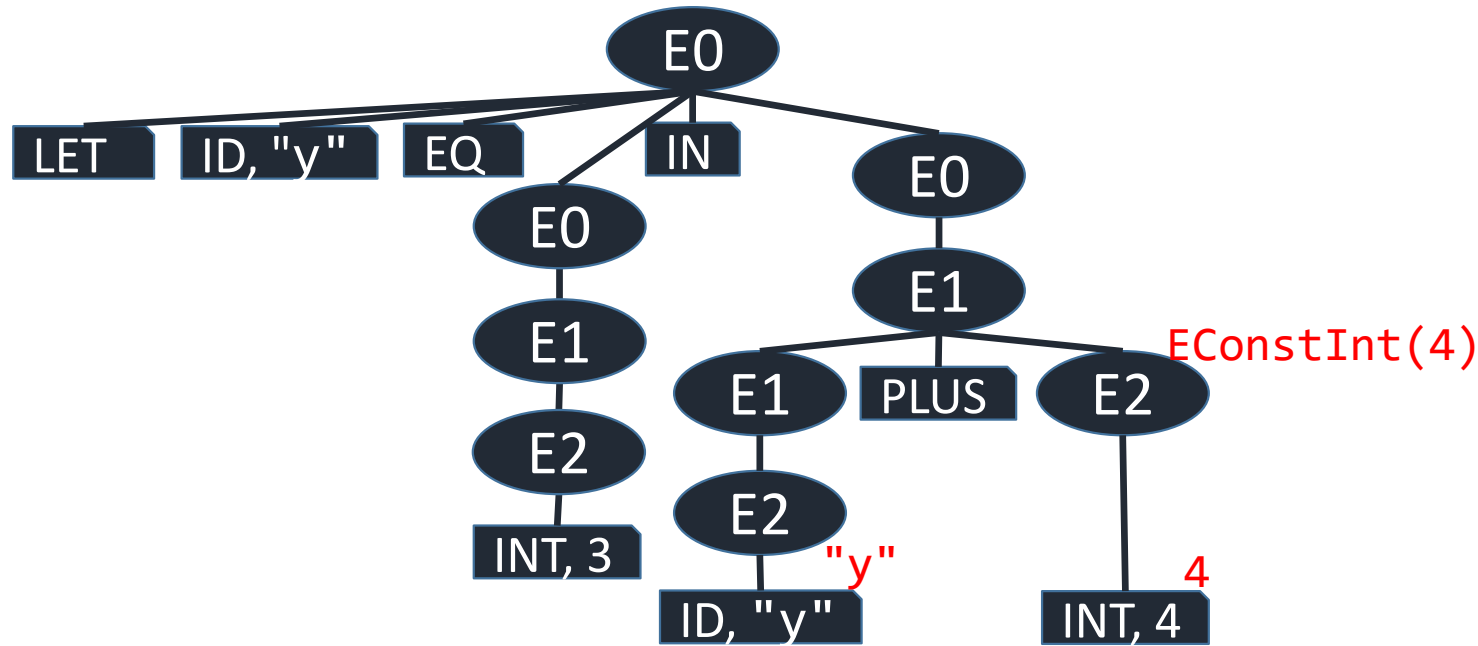




# 意味規則

。具象構文木に応じて抽象構文木を生成

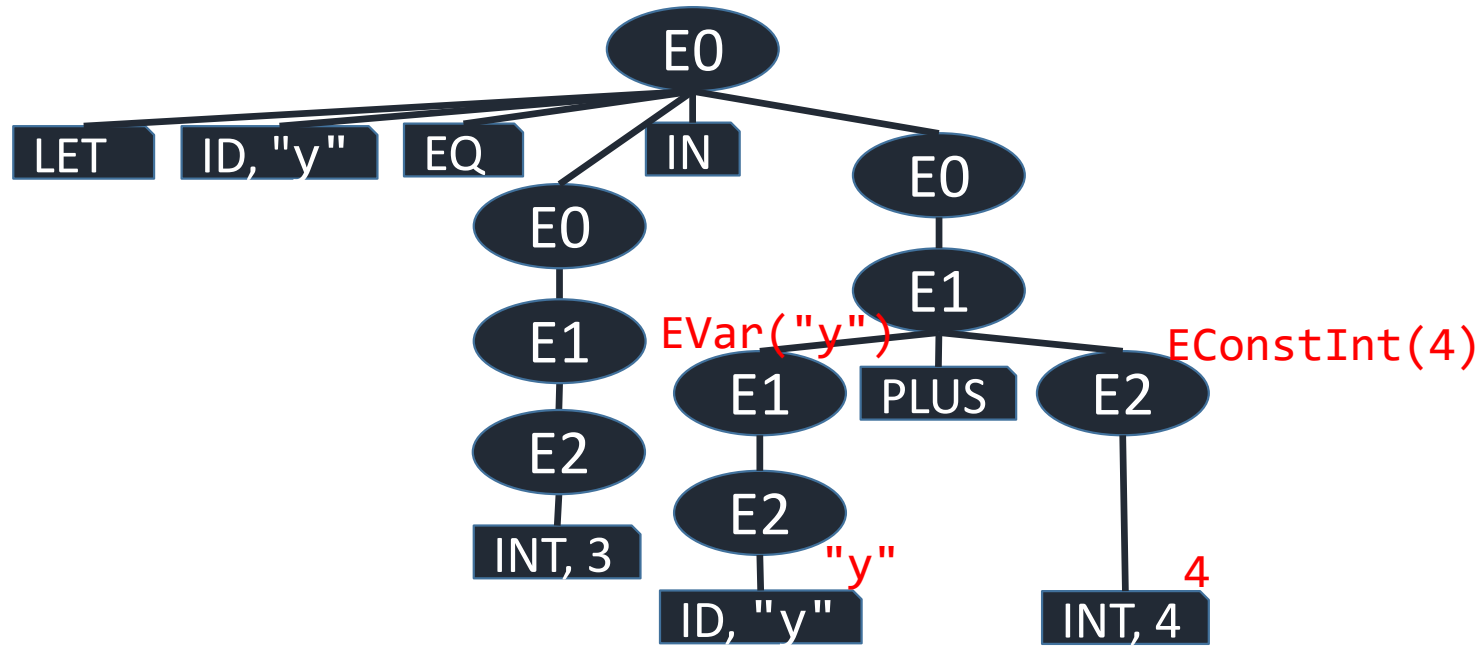
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 意味規則

。具象構文木に応じて抽象構文木を生成

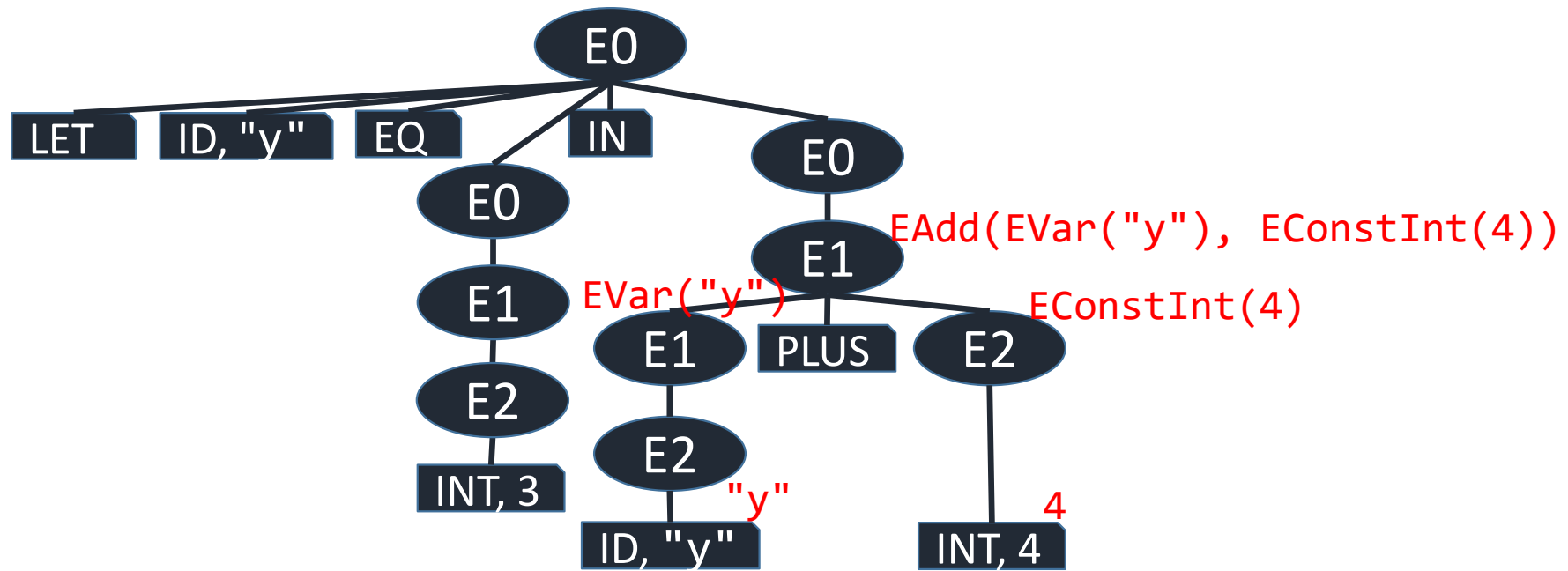
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 意味規則

。具象構文木に応じて抽象構文木を生成

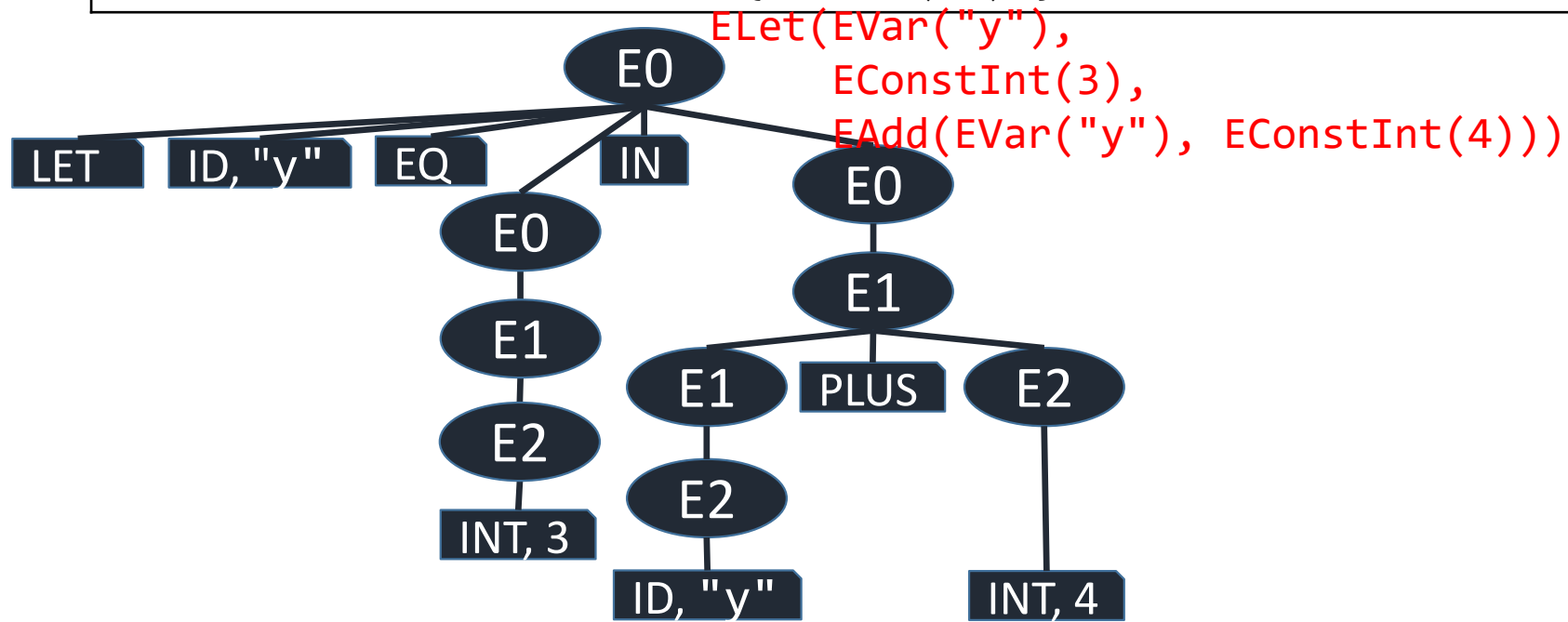
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 意味規則

。具象構文木に応じて抽象構文木を生成

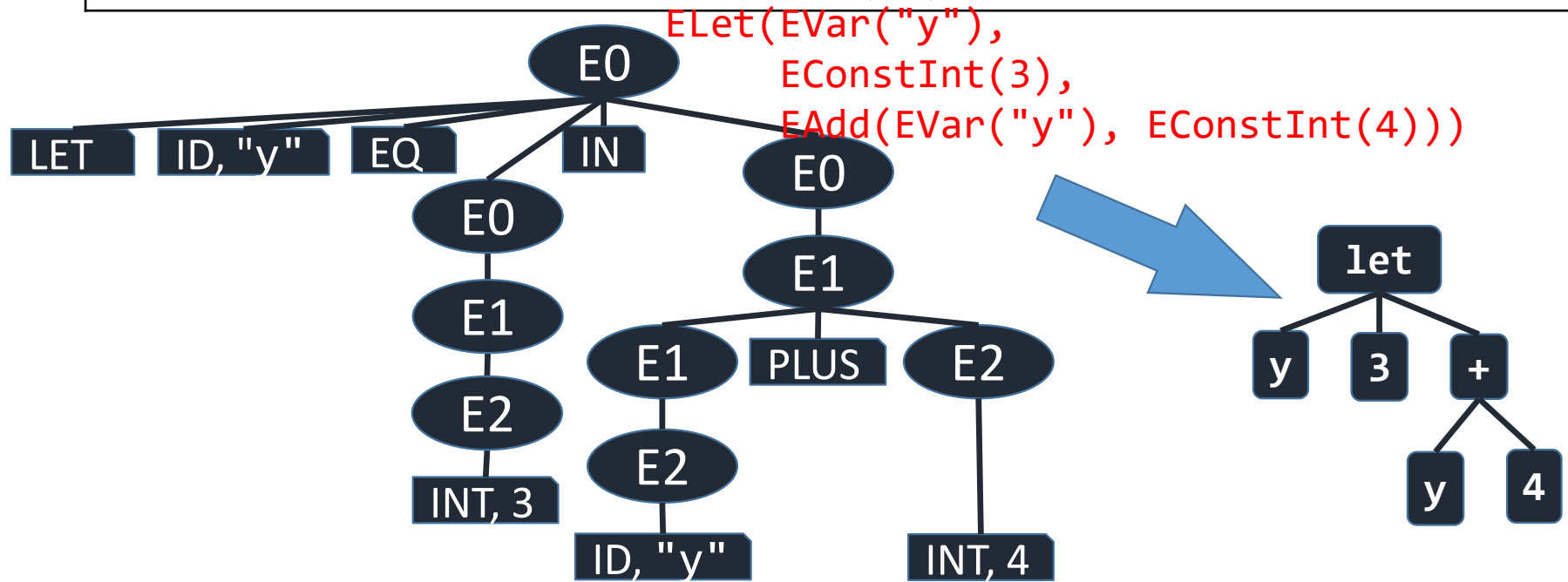
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 意味規則

。具象構文木に応じて抽象構文木を生成

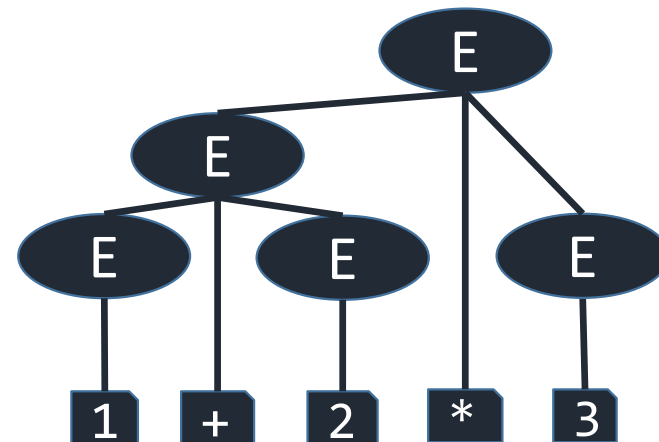
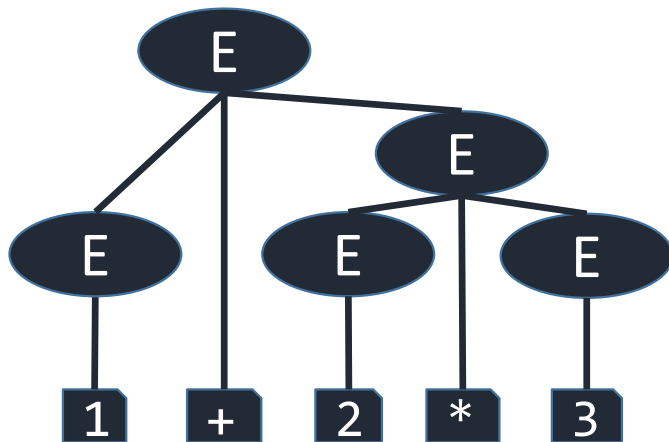
$E0$	$\rightarrow$	LET ID EQ $E0$ IN $E0$	{ ELet (\$2, \$4, \$6) }
		$E1$	{ \$1 }
$E1$	$\rightarrow$	$E1$ PLUS $E2$	{ EAdd (\$1, \$3) }
		$E2$	{ \$1 }
$E2$	$\rightarrow$	INT	{ EConstInt (\$1) }
		ID	{ EVar (\$1) }



# 曖昧な文法

- 1つのトークン列に対して複数の具象構文木を持つ
  - 構文解析の結果も複数になる

$E \rightarrow$	$E + E$
	$E * E$
	INT
	ID

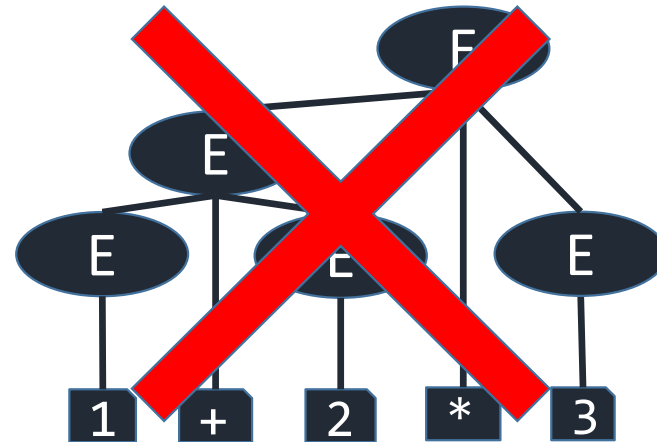
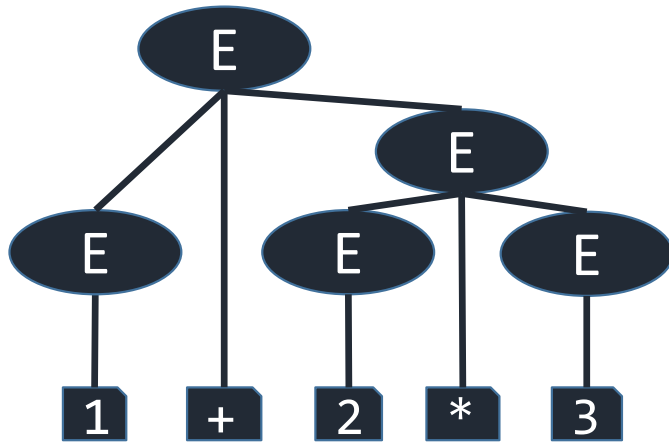


# 解決法 1 : 結合性と優先順位

$E \rightarrow$	$E + E$
	$E * E$
	INT
	ID

○ 例 :

- + と \* は左結合
- + は \* より結合が弱い (= 優先度が低い)

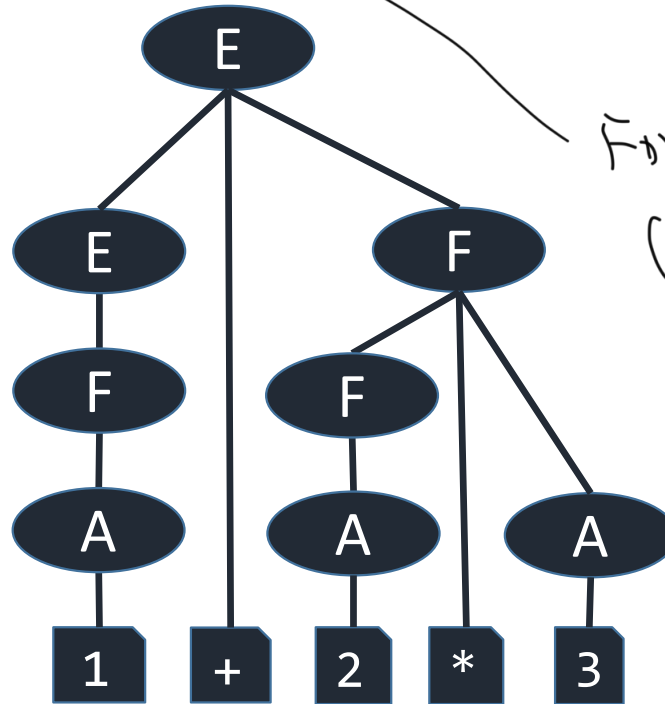


# 解決法 2 : 階層に分ける

$E \rightarrow E + F$   
|  $F$   
 $F \rightarrow F * A$   
|  $A$   
 $A \rightarrow \text{INT}$   
|  $\text{ID}$

具象構文木は唯一

$E$ は7-しご  
 $F$ はかけざん



$F$ が2の中にある。

( $F$ の方が優先順位が高い)



# インタプリタの概要

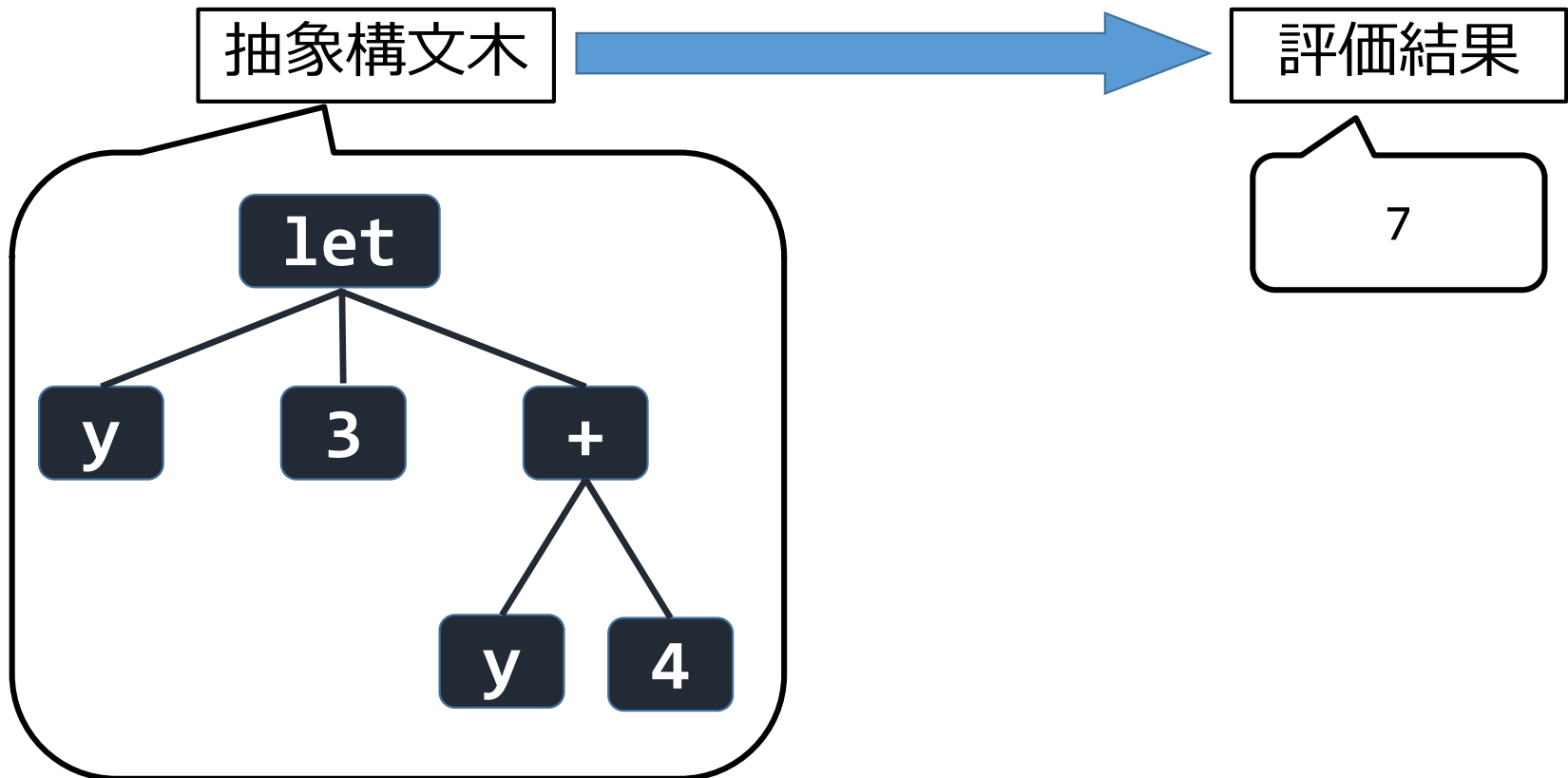
字句解析

構文解析

評価器

# 評価器

- 抽象構文木を評価し、値を得る
  - 簡単なものは、第2回 問4 で実装した



# OCaml での構文解析

ocamllex

ocamlyacc

生成されたモジュールの使い方

# 二つの補助ツール

字句解析器生成器

**ocamllex**

構文解析器生成器

**ocamlyacc**

プログラム

トークン列

抽象構文木

```
let y = 3 in  
  y + 4
```

LET

ID, "y"

EQ

INT, 3

IN

ID, "y"

PLUS

INT, 4

let

y

3

+

y

4

# OCaml での構文解析

`ocamllex`

`ocamlyacc`

生成されたモジュールの使い方

# ocamllex とは？

- OCaml 用の字句解析器生成器
  - 入力：字句定義ファイル（.mll）
  - 出力：字句解析モジュール（.ml）

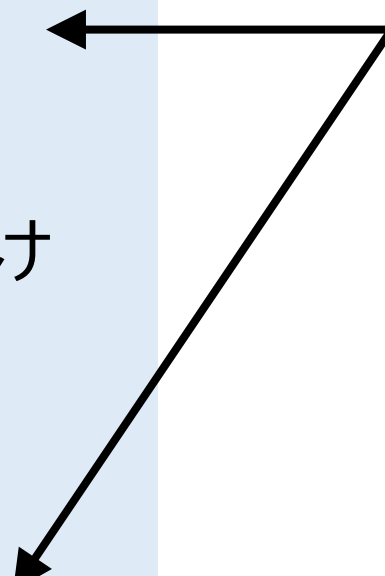
# .m11 の構造

```
{  
  (* header *)  
}
```

正規表現に名前付け  
解析規則の定義

```
{  
  (* trailer *)  
}
```

header と trailer 部に  
コードを書くと、  
生成されるモジュールの  
それぞれ先頭と末尾に  
コピーされる



※ コメントは (\* ... \*)

# .m11 の例

```
let digit = ['0'-'9']
let space = ' ' | '¥t' | '¥r' | '¥n'
let alpha = ['a'-'z' 'A'-'Z' '_']
let ident = alpha (alpha | digit)*
```

これは正規表現。  
↑

```
rule main = parse
```

```
| space+          { main lexbuf }
| "let"           { Parser.LET }
| "in"            { Parser.IN }
| "="             { Parser.EQ }
| "+"             { Parser.PLUS }
| ";;"            { Parser.SEMISEMI }
| digit+ as n     { Parser.INT (int_of_string n) }
| ident as id     { Parser.ID id }
```

↑  
n-hを  
ID

INTというタグをつけて。  
↓  
文字列nをintとして  
変換する。



# 正規表現に名前付け

- `let` 変数 = 正規表現
  - 正規表現が書けるところに、定義した変数可以利用できる
  - 詳しくはマニュアル参照

# 字句解析規則の定義

```
○ rule エントリポイント名 = parse
  | 正規表現1 { トークン1 }
  | 正規表現2 { トークン2 }
  | ...
and エントリポイント名2 = parse
  | ...
```

- 生成される字句解析モジュールは、  
エントリポイント名と同名の関数を含む

# 注意

- 正規表現のマッチング結果は、  
できるだけ長くマッチしたものが選ばれる
- そのようなものが複数ある場合には、  
もっとも上にあるものが選ばれる

キーワードを処理する規則は最初に

# 補足

- as でマッチした文字列を変数に束縛できる

```
| digit+ as n    { Parser.INT (int_of_string n) }  
| ident  as id   { Parser.ID id }
```

# 補足

- 字句解析関数を再帰呼び出しすることで、そのときマッチしている文字列を飛ばして次のトークンを返すことができる

```
rule main = parse
| space+          { main lexbuf }
| ...
```

- lexbuf で入力バッファを参照できる
- 別のエントリポイントも呼び出せる

残りを lexing して

space の場合は、残りを lexing して返して下に行く

# ocamllex の使い方

`ocamllex FILENAME.mll`

- `FILENAME.ml` が生成される

```
$ ocamllex lexer.mll
```

```
...
```

```
$ ls lexer.*
```

```
lexer.ml    lexer.mll
```

# OCaml での構文解析

ocamllex

ocamlyacc

生成されたモジュールの使い方

# ocamlyacc とは？

- OCaml 用 構文解析器生成器

- 入力：文法定義ファイル（.mly）
- 出力：構文解析器モジュール（.ml）

- 生成される構文解析器は、

- トークン列を取って
- 抽象構文木を返す

※ 他のものを返すようにもできる



# .mly ファイルの構造

```
%{  
    (* header *)  
%}  
トークンや演算子の定義
```

```
%%  
生成規則の定義
```

```
%%  
%{  
    (* trailer *)  
%}
```

header と trailer 部に  
コードを書くと、  
生成されるモジュールの  
それぞれ先頭と末尾に  
コピーされる

※ コメントは `/* ... */`  
header と trailer では `(* ... *)`

# トークンの定義

%token トークン名1 トークン名2

%token <型> トークン名1 トークン名2

○ 例: %token <int> INT  
%token <string> ID  
%token LET IN EQ  
%token PLUS  
%token SEMISEMI

LET IN EQ いう  
トークンがありますよ。

次の型が生成される

```
type token = INT of int | ID of string
            | LET | IN | EQ | PLUS | SEMISEMI
```

# 開始記号の定義 / 記号の型

## ○ 開始記号の定義の例

```
%start toplevel
```

- toplevel という非終端記号を開始記号とする

## ○ 非終端記号の意味動作の型

```
%type <Syntax.program> toplevel
```

- 構文解析の結果の型を指定
- 開始記号については必須
- 通常は構文木の型を指定

※ 型指定は、構文解析の結果の型を指定する。という位。

# 生成規則定義の例

\$n: n番目の記号の解析結果

```
toplevel:
  expr SEMISEMI { Exp ($1) }
;
```

解析結果を表す式 (意味動作)

```
expr:
| IF expr THEN expr ELSE expr { EIf ($2, $4, $6) }
| arith_expr EQ arith_expr    { EEq ($1, $3) }
| arith_expr LT arith_expr    { ELt ($1, $3) }
| arith_expr                  { $1 }
;
```

```
arith_expr:
| arith_expr PLUS factor_expr { EAdd ($1, $3) }
| factor_expr                 { $1 }
;
```

.....

# ocamlyacc の使い方

`ocamlyacc FILENAME.mly`

- `FILENAME.mli` と `FILENAME.ml` が生成される

```
$ ocamlyacc parser.mly
```

```
...
```

```
$ ls parser.*
```

```
parser.ml    parser.mli    parser.mly
```

# 注意

- header や trailer 部の定義は .mli に含まれない
  - 他のモジュールから参照したい定義は書かない
- %type や %token に他のモジュールの型を書くときは、モジュール名を必ず書くように

# 注意：conflict

## ○「文法が曖昧な恐れあり」

### ■ shift/reduce conflict

- shift が優先される
- 優先度などが適切かどうか確認
- 問題とならないケースもあるが、解消した方がよい

### ■ reduce/reduce conflict

- 文法に問題がある可能性
- 文法に曖昧さがないか確認
- `ocamlyacc -v` で出力される `.output` ファイルに問題解決のヒントが？
  - どこで conflict が生じたかの情報有
  - LALR(1) パーサについての知識が必要

# 意図とは違う parse をされる場合

- conflict が原因？

- .output ファイルの確認

- そのまま読む
    - OCAMLRUNPARAM=p で実行し遷移を確認

- そもそも文法が意図通りではない？

- 実は字句解析に問題がある場合も



# OCaml での構文解析

ocamllex

ocamlyacc

生成されたモジュールの使い方

# 実際に構文解析を行うには

- 構文解析モジュールにある、  
構文解析関数を呼べばよい
  - 関数の名前は、開始記号の名前
  - 入力：字句解析関数と入力バッファ
  - 出力：解析結果

# 利用例

- 構文解析結果を端末に表示

```
let _ =  
  let lexbuf = Lexing.from_channel stdin in  
  let result = Parser.toplevel Lexer.main lexbuf in  
  print_command result; print_newline ()
```

(自分で定義した) 解析結果を表示する関数

# コンパイル

```
$ ocaml yacc parser.mly
```

```
...
```

```
$ ocamllex lexer.mll
```

```
...
```

```
$ ocamlc -c syntax.ml
```

```
$ ocamlc -c parser.mli
```

```
$ ocamlc -c parser.ml
```

```
$ ocamlc -c lexer.ml
```

```
$ ocamlc -c example.ml
```

```
$ ocamlc -o example syntax.cmo parser.cmo ¥  
lexer.cmo example.cmo
```

```
$ echo "if x < 0 then 0 else x;;" | ./example
```

# 簡単な評価器の作成

# 振り返り：第2回 問4

○ 次の式を評価する評価器を書け

$$\begin{array}{l} E ::= 0 \mid 1 \mid 2 \mid \dots \\ \quad \mid E + E \mid E - E \mid E * E \mid E / E \\ \quad \mid \text{true} \mid \text{false} \\ \quad \mid E = E \mid E < E \\ \quad \mid \text{if } E \text{ then } E \text{ else } E \end{array}$$

○ 提出期限を過ぎたため、必要なら解説します

- 第2回 問4の実装は必要なので

# 拡張：変数

- あらかじめ定義された変数を式中で使えるように改良する
  - 今は新しい変数を定義する構文は考えない

- 構文：

$E ::= \dots \mid I$       (I は識別子)

- 変数定義：

$i = 1$

$v = 5$

$x = 10$

# 環境

- 変数から値へのマップ
  - 例えばリストを使って、次のように表現できる

```
exception Unbound
```

```
type env = (string * value) list
```

```
let extend x v env = (x,v) :: env
```

```
let lookup x env =
```

```
  try
```

```
    List.assoc x env
```

```
  with
```

```
  | Not_found -> raise Unbound
```



# 改良された評価器

`eval : env -> expr -> value`

- 新しく環境も引数に取る
  - 式が変数の場合に、環境を使って評価する

```
let rec eval env e =  
  match e with  
  | EConstInt i -> VInt I  
  | ...  
  | EVar x -> lookup x env  
  | ...
```

# サンプルプログラム ex0

## ○ 次の構文の式を評価する

$$\begin{array}{l} E ::= 0 \mid 1 \mid 2 \mid \dots \mid E + E \\ \quad \mid \text{true} \mid \text{false} \mid E = E \mid E < E \\ \quad \mid \text{if } E \text{ then } E \text{ else } E \mid I \end{array}$$

- I は識別子

## ○ ファイル構成 (主なもの)

- syntax.ml      構文の定義 (と表示関数)
- eval.ml        評価器
- main.ml        主要なループ、初期環境の生成など

## ○ make でビルドできる

# 例題

理解の確認をするための課題です

課題提出システム上での提出の必要はありません

例題を解きTAに見せることで出席とします

分からないことがあったら、積極的に質問しましょう

# 例題

- サポートページのプログラム ex0 をビルドし、動作を確認せよ
- プログラムを書き換え、大域環境に次の変数定義を加えよ
  - 変数 ii が 2
  - 変数 iii が 3
  - 変数 iv が 4

## TAチェック

- 加えた変数を用いたプログラムを評価せよ

# レポート課題 5

締切：2019/5/28 13:00(JST)

# 問 1

- 付録の example.ml の例を自分で試せ
  - make に任せずにビルドの各ステップを手で実行し出力を確認せよ
  - いろいろ試してみよ
    - 例：
      - lexer.mll や parser.mly を適当に変更してみる
        - arith\_expr 等を expr に潰してみる
      - 標準入力ではなく文字列を読むように書き換える
      - ocaml yacc -v の出力を読んでもみる

## 問 2

- 例題のインタプリタを改良し、  
整数の減算、乗算、除算を扱えるようにせよ

$$\begin{array}{l} E ::= 0 \mid 1 \mid 2 \mid \dots \\ \quad \mid E + E \mid E - E \mid E * E \mid E / E \\ \quad \mid \text{true} \mid \text{false} \\ \quad \mid E = E \mid E < E \\ \quad \mid \text{if } E \text{ then } E \text{ else } E \\ \quad \mid I \mid (E) \end{array}$$

- I は識別子
- 参考資料の ex2 に適当な eval.ml を与えればよい

# 問 3

↑ これは  $\lambda$  関数  
のみを扱っていた。

○ 問 2 のインタプリタを改良し、  
変数定義を扱えるようにせよ

- トップレベル定義と局所定義の両方

$C ::= E;; \mid \text{let } I = E;;$

$E ::= \dots \mid \text{let } I = E \text{ in } E$

- 参考資料の ex3 に適当な eval.ml を与えればよい



# ヒント

- `let x = e1 in e2` の評価
  - `e1` を評価
  - `x` と評価結果の対応を環境に追加
  - 新しい環境の下で `e2` を評価
- `let x = e;;` の評価
  - `e` を評価
  - `x` と評価結果の対応を環境に追加
  - 新しい環境と評価結果を返す

# ヒント

○ eval\_command の返値は三組み

- 変数名（無名の場合は "-")
- 新しい環境
- 評価結果
  - 式ならば評価結果の値、宣言ならば束縛される値

○ 第一フィールドと第三フィールドは表示にのみ使っている

# 問 4

- 問 3 のインタプリタを改良し、  
ブール値の演算を扱えるようにせよ
  - 少なくとも論理和、論理積をサポートせよ

$$E ::= \dots \mid E \ \&\& \ E \mid E \ \|\ E$$

- この課題については参考資料はない
  - 構文定義や構文解析も自分で与えること

# 注意

- 問 2 ・ 問 3 ・ 問 4 はまとめて提出してもよい
  - どの問に答えているかは明らかにすること
  - 考察はそれぞれ行うこと
- OCaml の標準ライブラリは自由に用いてよい
- ビルド方法の記述は忘れないように
- Conflict は可能な限り消すこと
- 参考資料のコードを使う必要はない

# 発展 1

○インタプリタのエラー処理を改良せよ

例：

- 例外発生時に適切なメッセージを表示する
- 例外発生時はインタプリタが終了してしまうのを、インタプリタプロンプトに戻るようにする

例外は評価器だけでなく、字句解析器と構文解析器も発生させうることに注意

# 発展 2

- OCaml では let 宣言の列を一度に入力することができる。この機能を実装せよ。

- 動作例

```
# let x = 10
    let y = x+1
    let z = x*y;;
val x = 10
val y = 11
val z = 110
```

# 発展 3

- OCaml では let and を使って  
複数の変数を同時に定義することができる。  
この機能を実装せよ。

- 動作例

```
# let x = 10;;  
val x = 10  
# let x = 50  
    and y = x * 2;;  
val x = 50  
val y = 20
```

```
# let x = 10;;  
val x = 10  
# let x = 50  
    and y = x * 2  
    in x + y;;  
- = 70
```