関数・論理型プログラミング実験
ML演習第4回

松田 一孝

TA: 武田広太郎 寺尾拓

今日から言語処理系の作成

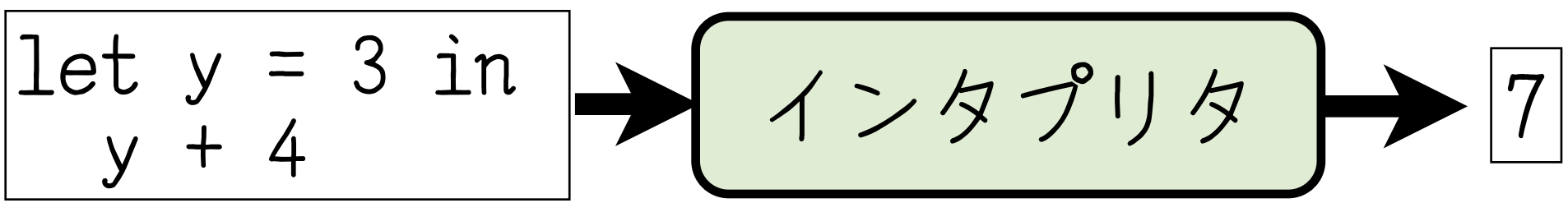
- 今後の予定
  - 5/01：簡単な評価器
    - 字句解析・構文解析．簡単な評価器
  - 5/13：関数型言語の評価器
    - （高階）関数定義・呼出機構の作成
  - 5/20：型システム
    - ML風の型推論の実装
  - 5/27：その他拡張
    - 評価規則等

今日の内容

- 簡単な言語のインタプリタの作成
  - 字句解析・構文解析
    - ocamllex
    - ocamlyacc
  - インタプリタの作成（レポート課題）
    - 第2回問5の拡張

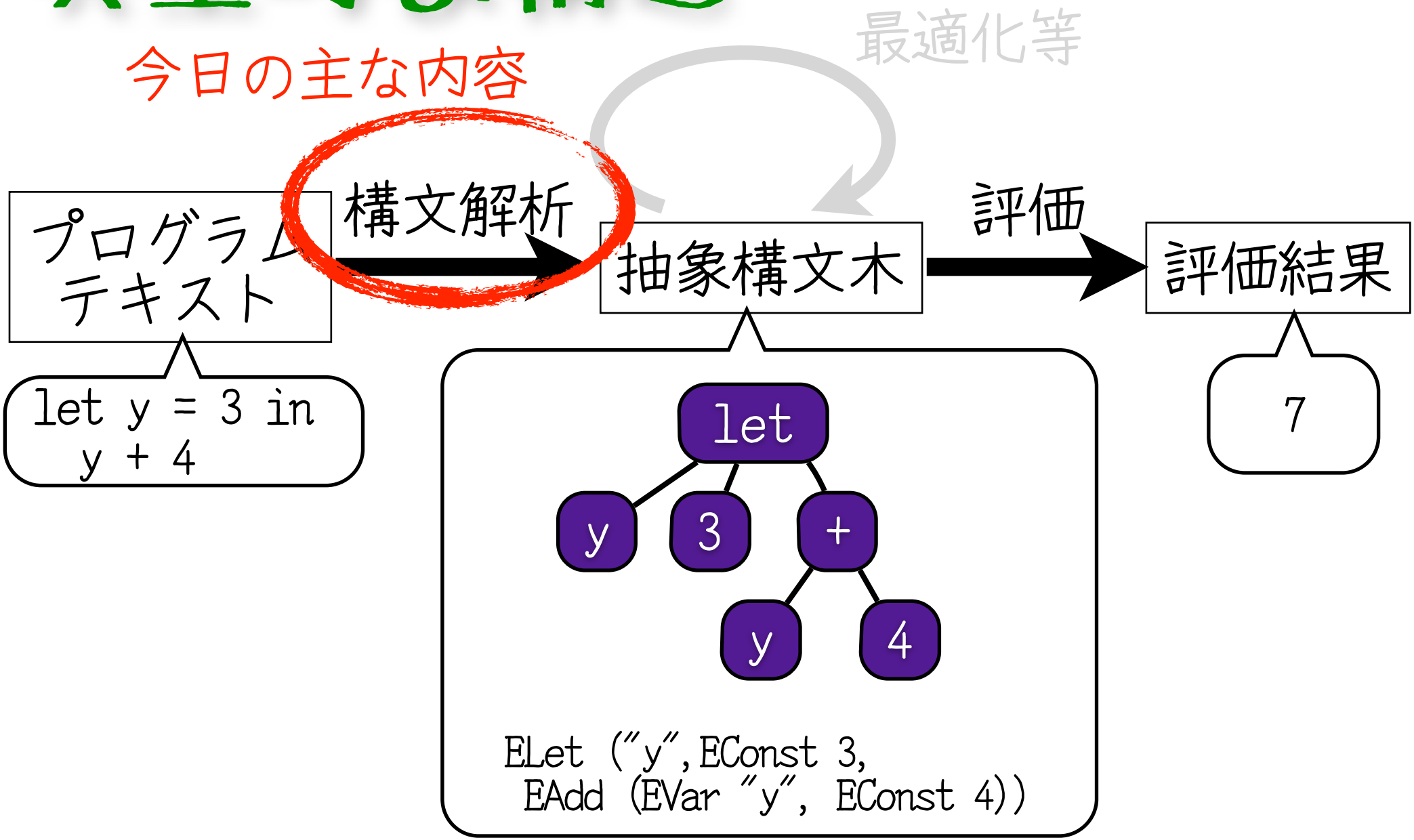
復習：インタプリタとは

- 式を入力にとり，その評価結果である値を出力するプログラム
  - cf. コンパイラ
    - ある言語のプログラムを，別の言語のプログラムに翻訳するプログラム



典型的な構造

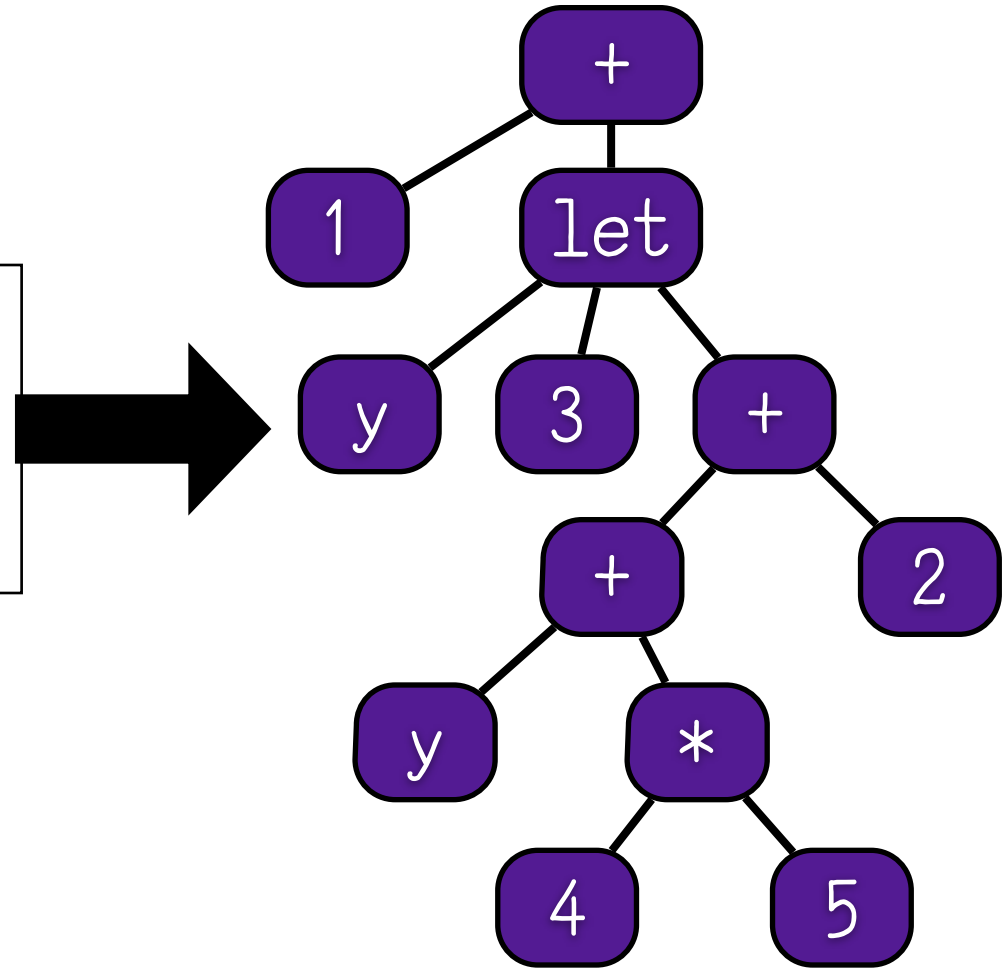
今日の主な内容



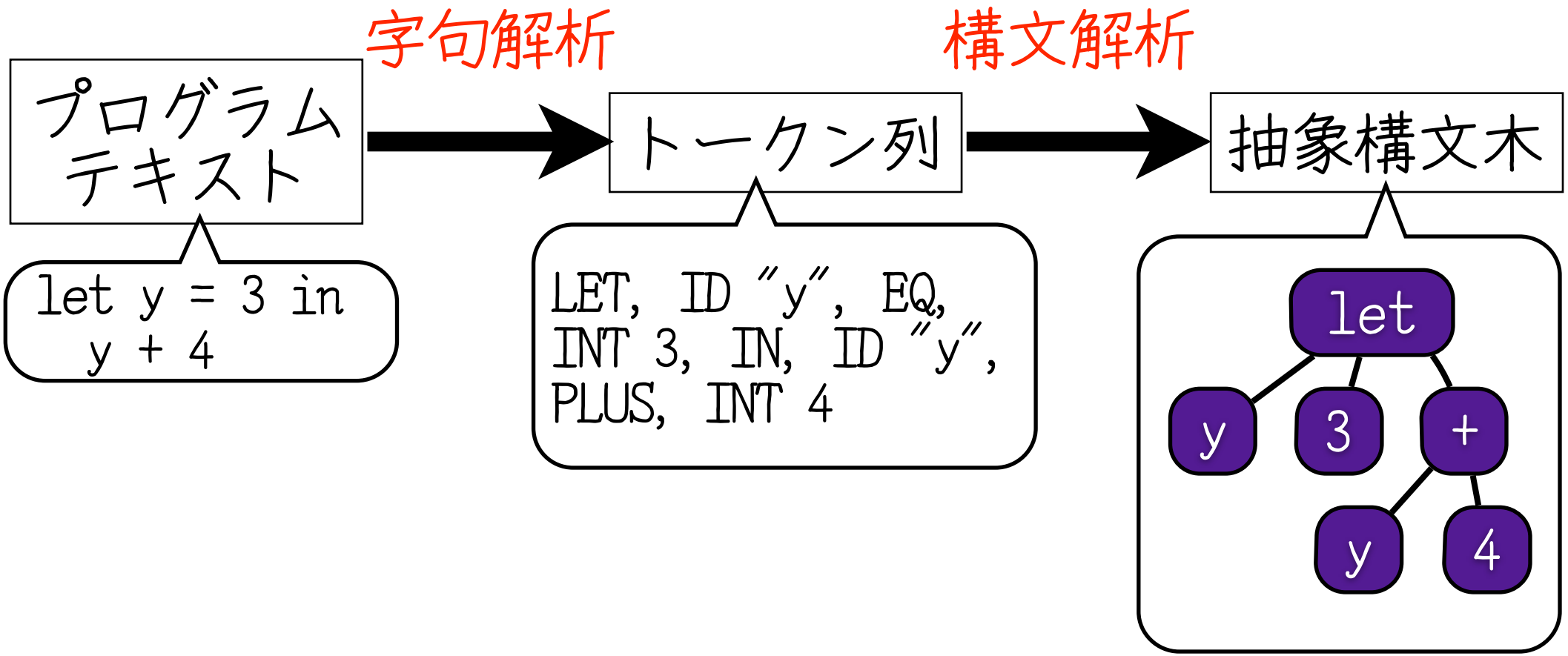
確認：抽象構文木（AST）

プログラムの構造を表現した木

```
1 +
(let y = 3 in
  (y + 4 * 5) + 2)
```



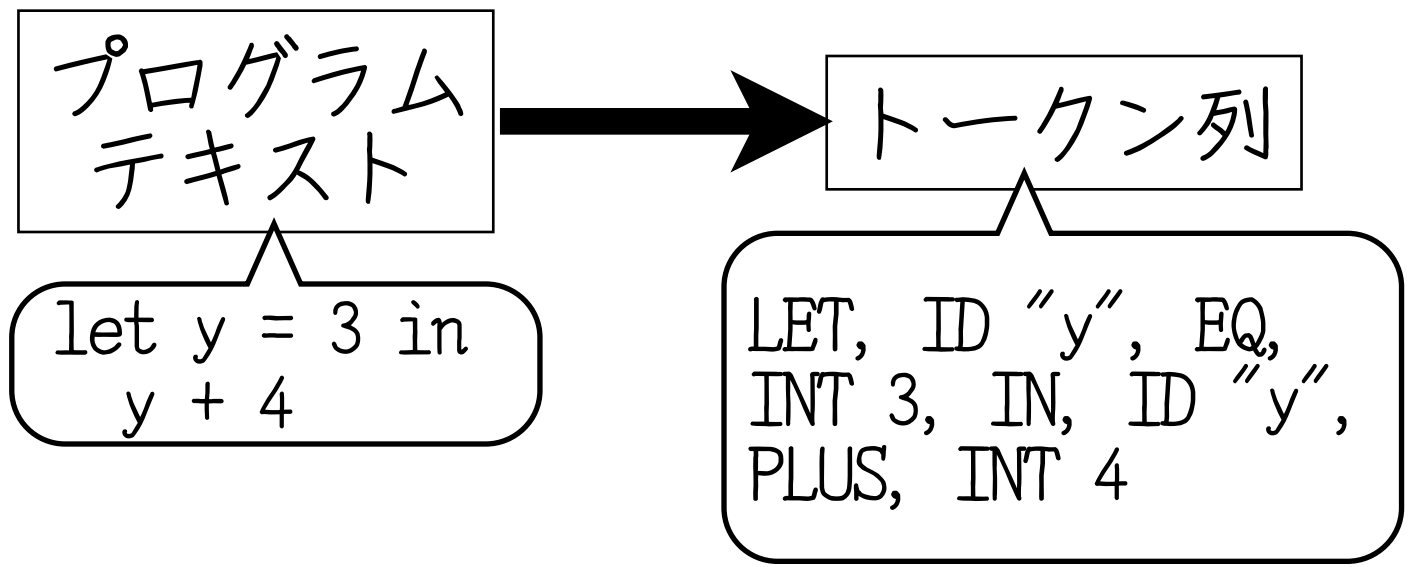
典型的なASTの作りかた



注：字句解析を用いない場合もある
(Scannerless Parsing)

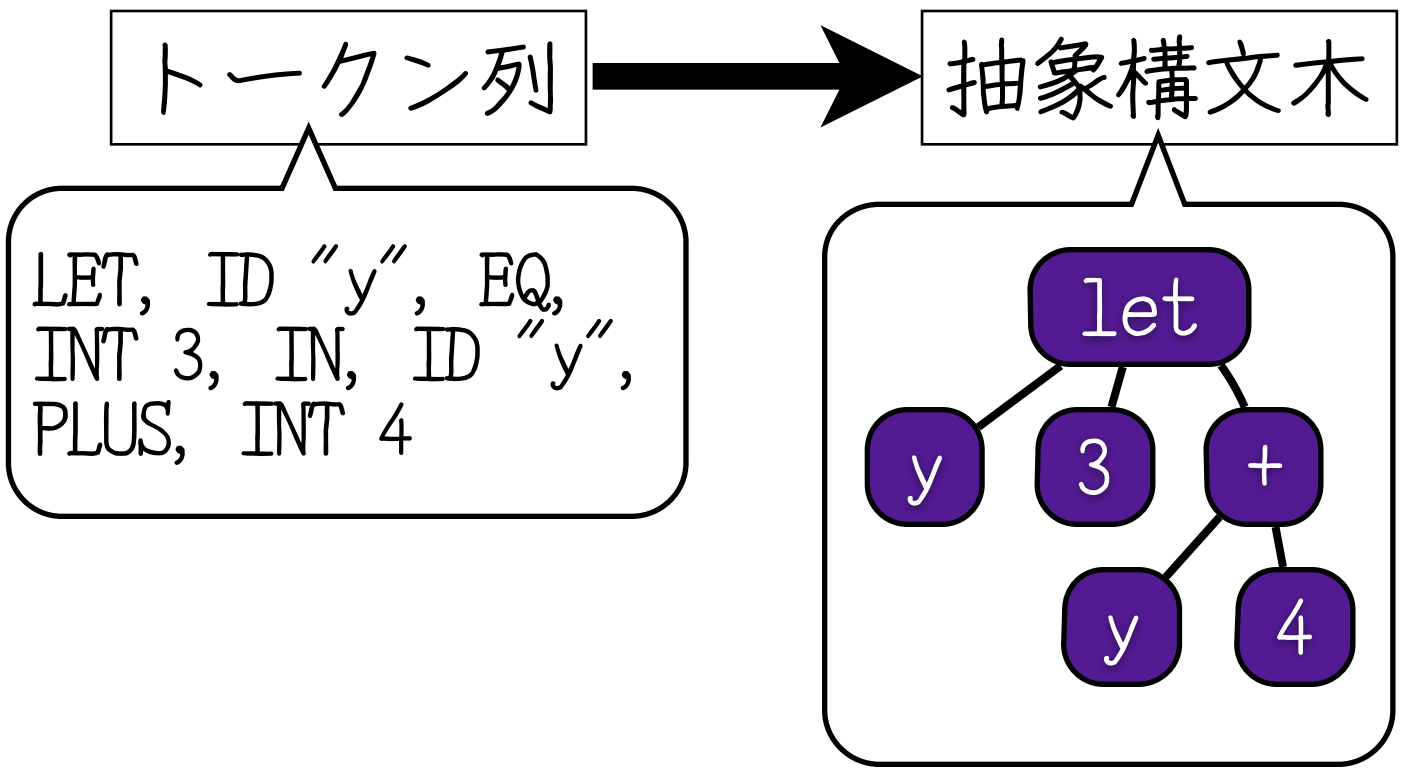
字句解析（lexing）

- 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
    - キーワード，識別子，数字，…



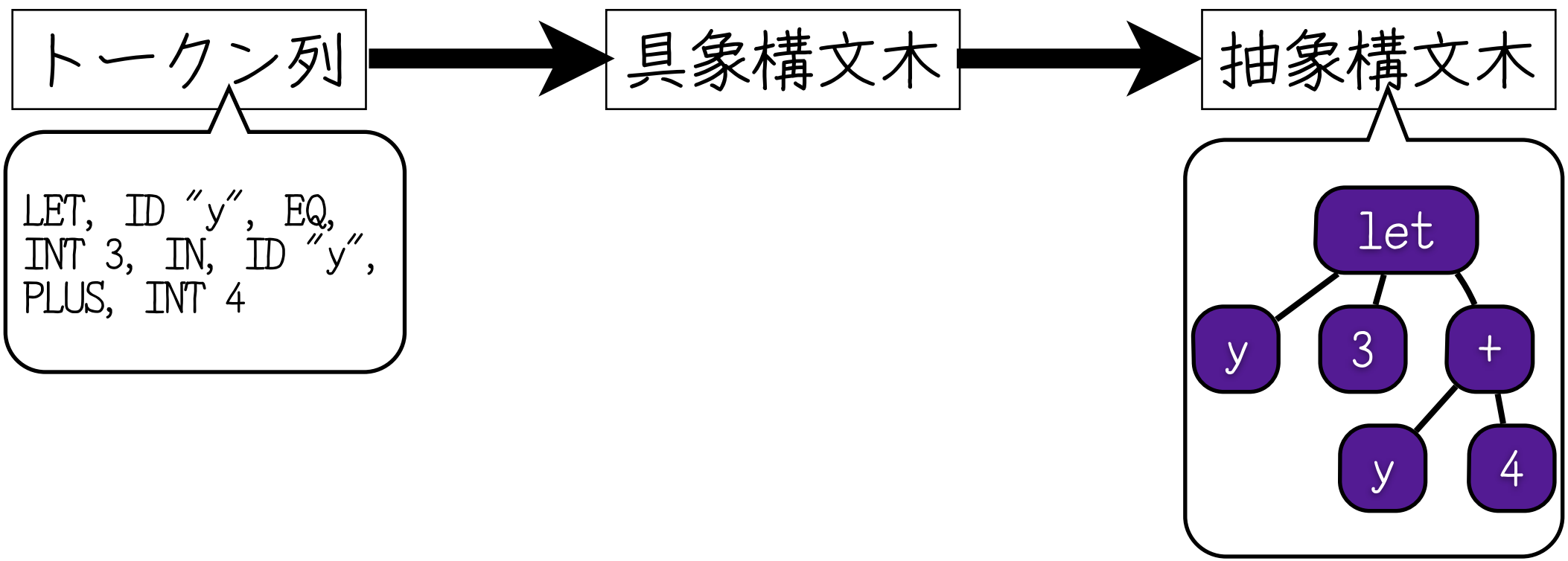
構文解析（parsing）

- （ここでは）トークン列を抽象構文木に変換する



構文解析の構造

文法と意味動作



文法と意味動作

- （ここでは）トークン列から，
  （暗黙に）具象構文木を経由し，
  抽象構文木をどう作るかの記述

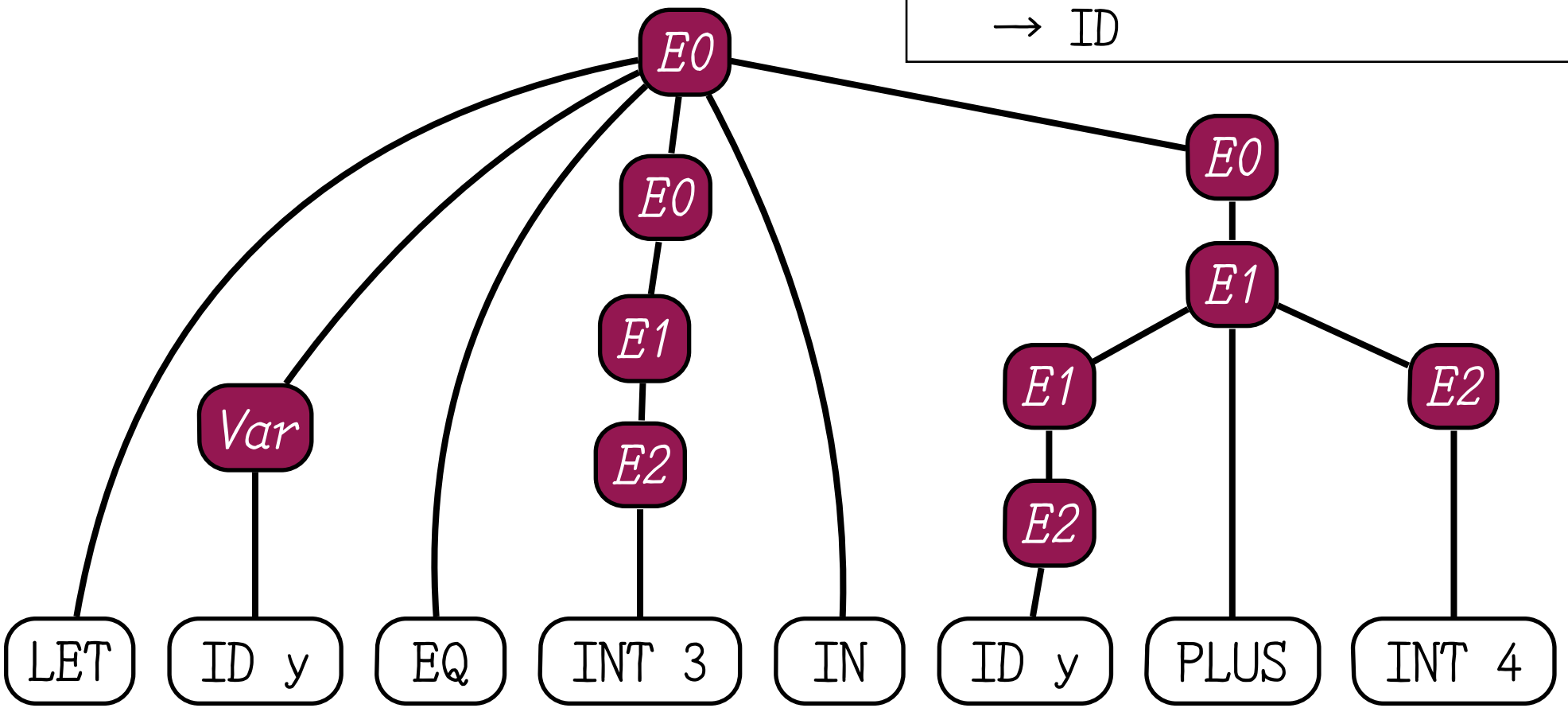
```
E0 → LET Var EQ E0 IN E0   {ELet ($2, $4, $6)}
   → E1                    {$1}
E1 → E1 PLUS E2            {EAdd ($1, $3)}
   → E2                    {$1}
E2 → INT                   {EConst (VInt $1)}
   → ID                    {EVar $1}
```

注：上はocamlyaccの構文ではない

具象構文木（CST）

- 文法の導出木のこと

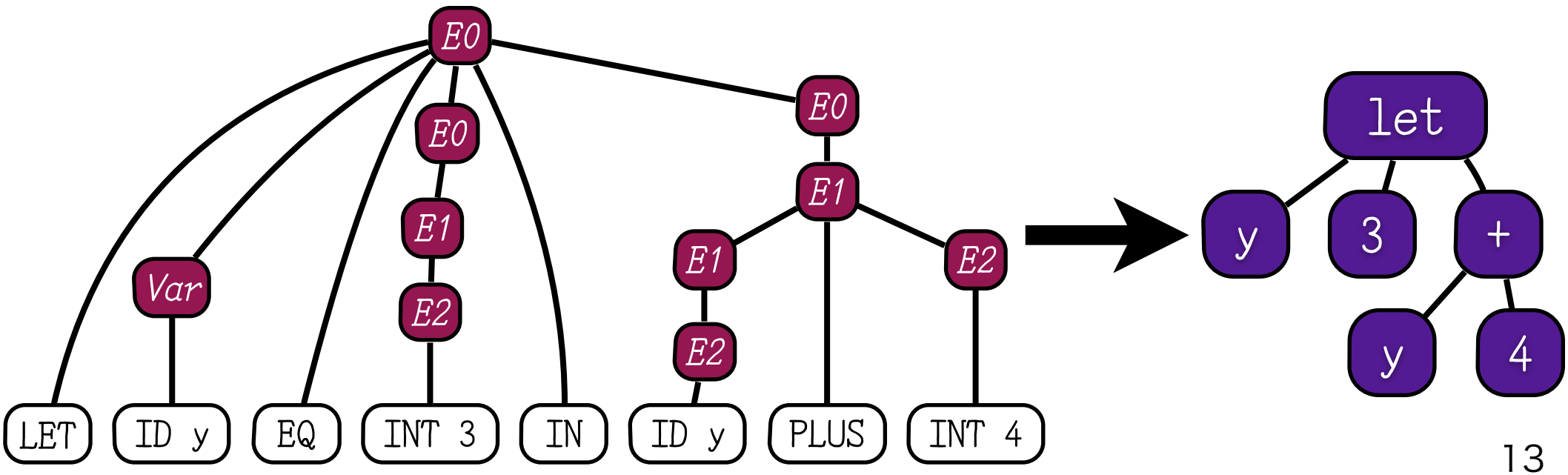
```
E0 → LET Var EQ E0 IN E0
   → E1
E1 → E1 PLUS E2
   → E2
E2 → INT
   → ID
```



意味規則

- 具象構文木に応じて抽象構文木を生成

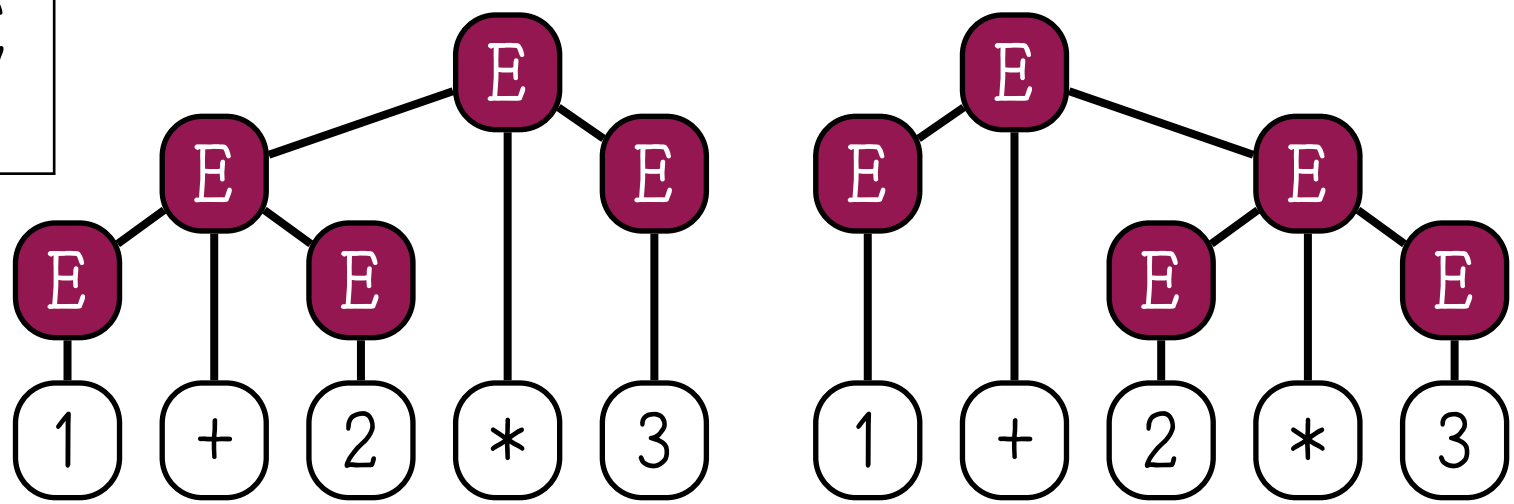
```
E0 → LET Var EQ E0 IN E0   {ELet ($2, $4, $6)}
   → E1                    {$1}
E1 → E1 PLUS E2            {EAdd ($1, $3)}
   → E2                    {$1}
E2 → INT                   {EConst (VInt $1)}
   → ID                    {EVar $1}
```



曖昧な文法

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

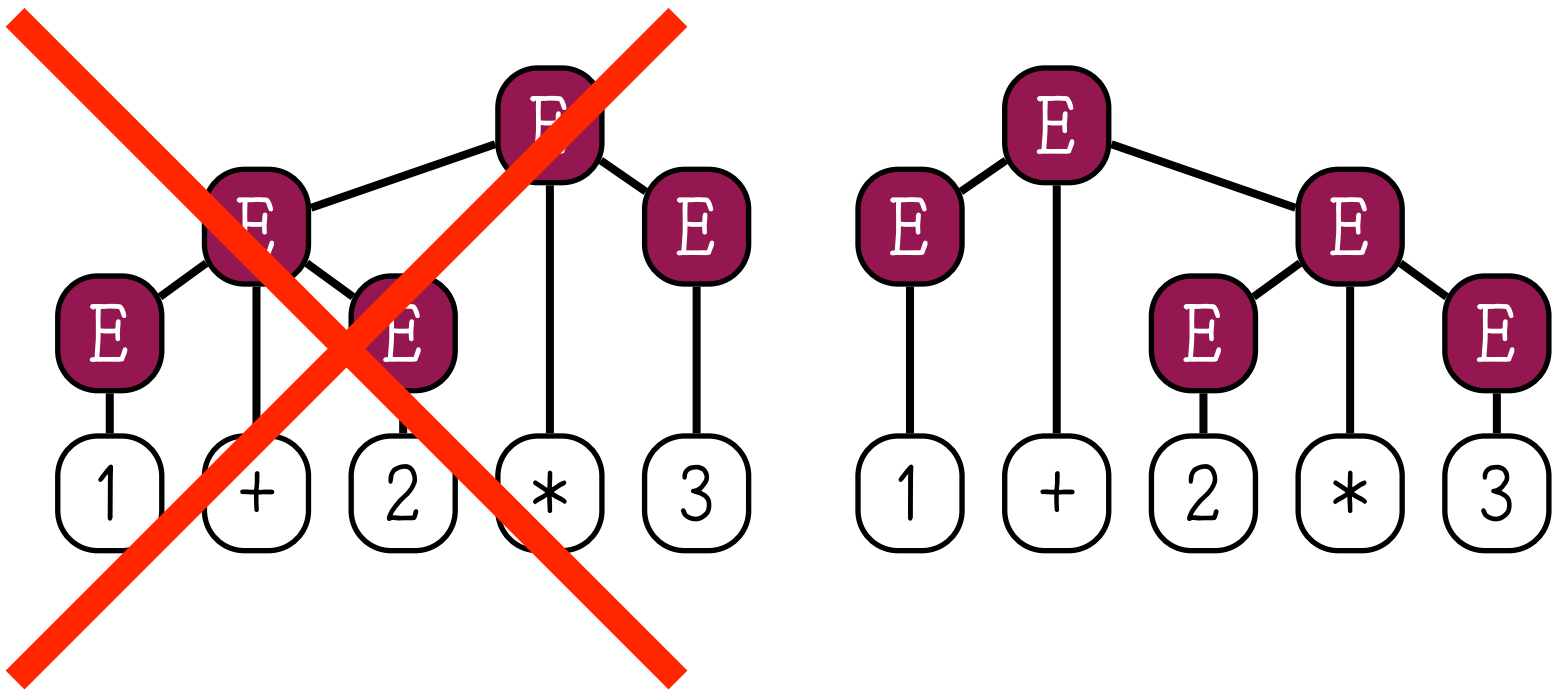
```
E → INT
E → E + E
E → E * E
E → (E)
```



1 + 2 + 3についても同様

解決：結合性と優先順位

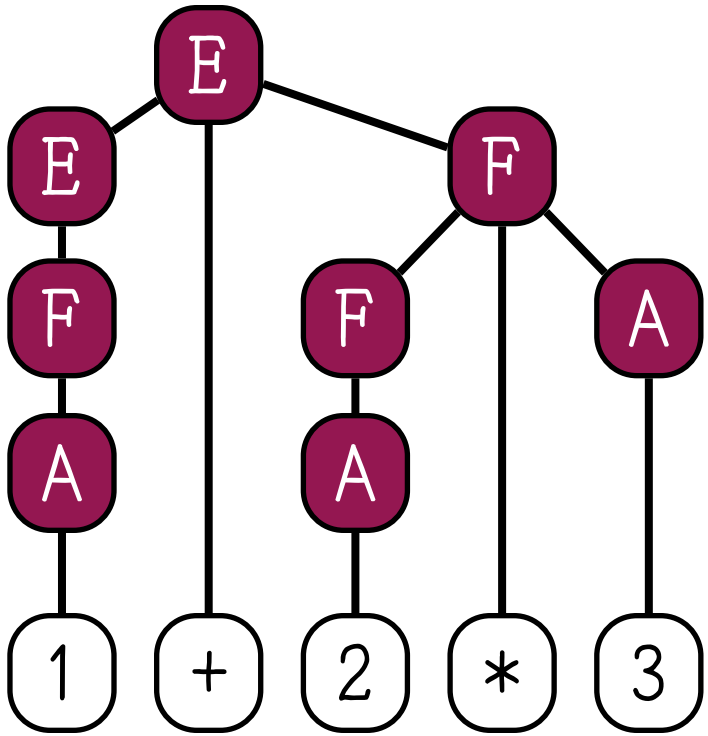
- +と*は左結合，
  +は*より結合が弱い（優先度が低い）



解決：階層に分ける

E	→	E	+	F
E	→	F		
F	→	F	*	A
F	→	A		
A	→	(E)		
A	→	INT		

具象構文木は唯一

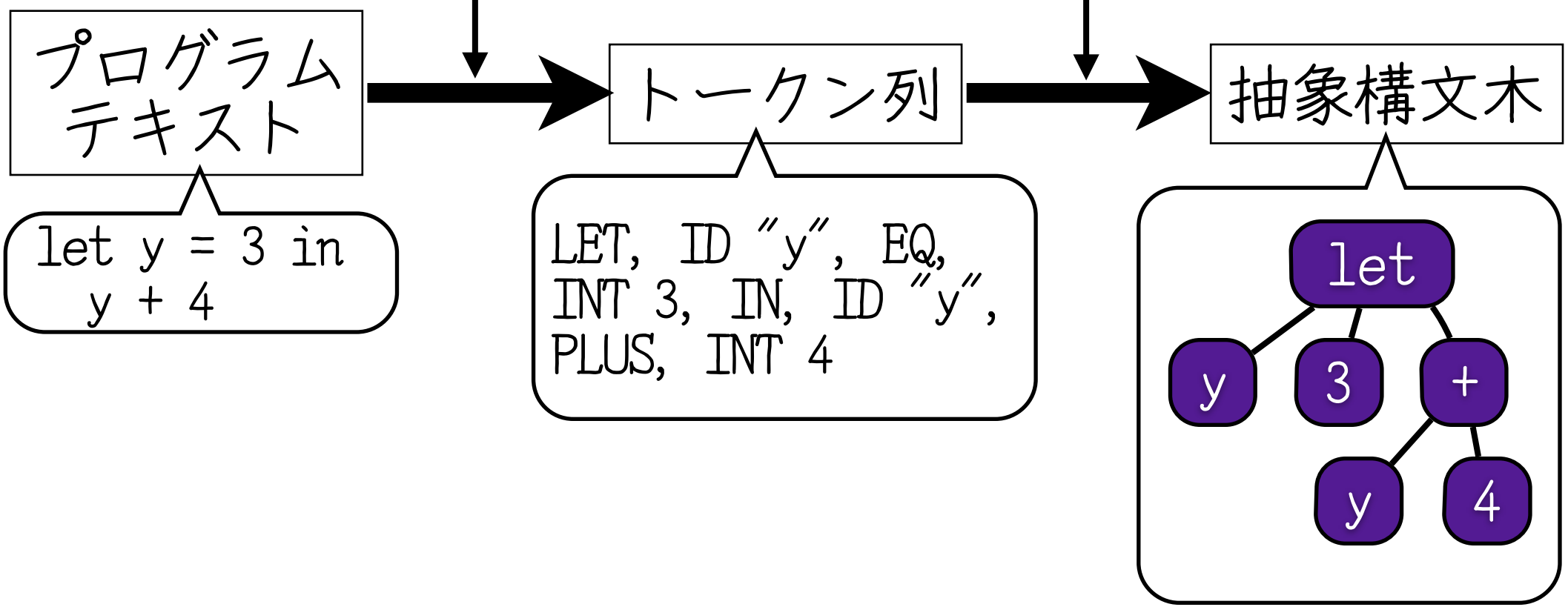


OCamlでの構文解析

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



ocamllex

ocamllexとは？

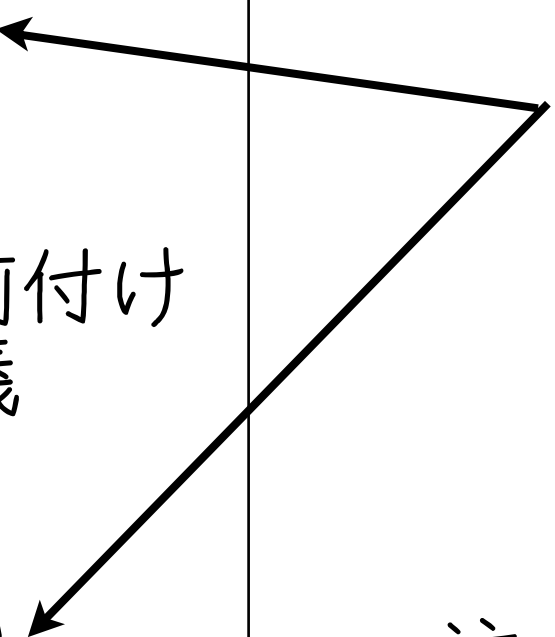
- OCaml用の字句解析器生成器
  - 入力：字句定義ファイル（.mll）
  - 出力：字句解析モジュール（.ml）

.mll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

注：コメントは(* … *)

.mllの例

```
{
(* トークンの型は後述のexampleParser.mlで定義 *)
open ExampleParser
}
let digit = ['0'-'9']
let space = ' ' | '\t' | '\r' | '\n'
let alpha = ['a'-'z' 'A'-'Z' '_' ]
let ident = alpha (alpha | digit)*

rule token = parse
| space+        { token lexbuf }
| "let"         { LET }
| "in"          { IN  }
| "="           { EQ }
| '+'           { PLUS }
| digit+ as n   { INT (int_of_string n) }
| ident  as n   { ID n }
| eof           { EOF  }
```

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，定義した変数が利用できるようになる
  - ocamllexで使用可能な正規表現はマニュアル参照

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

- 正規表現はマッチング結果は
  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
      字句解析規則は最初に

補足

- asをつかうとマッチした文字列を変数に束縛可

```
| digit+ as n { INT(int_of_string n)}
```

補足

- 字句解析関数を再帰呼出することで，その時マッチしている文字列を飛ばして次のトークンを返せる
  - rule token = parse
    …
    | space+ { token lexbuf }
    …
    * 入力バッファはlexbufで参照可
    * 別のエントリポイントも呼び出せる

ocamllexの使い方

- ocamllex FILENAME.mll
  - FILENAME.mlが生成される

```
$ ocamllex exampleLexer.mll
...
$ ls exampleLexer.*
exampleLexer.ml   exampleLexer.mll
```

ocamlyacc

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
  - トークン列をとって
    抽象構文木を返す
    * 他のものを返すようにでもできる

.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

注：コメントは/* … */
(header部とtrailer部除く)

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 EOF
```

以下の型が生成

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

演算子の優先順位・結合性

- %left トークン名$_1$ トークン名$_2$ …
  %right トークン名$_1$ トークン名$_2$ …
  %nonassoc トークン名$_1$ トークン名$_2$ …

例：

```
%left     PLUS MINUS
%right    TIMES DIV
```

- 下に書いた演算子ほど優先度高
- %left, %right, %nonassocが結合性
  - nonassocだと1 op 2 op 3と書けなくなる

開始記号の定義/記号の型

- 開始記号の定義の例
  - %start main
    - main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 <int> main
  - 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

生成規則定義の例

{}に解析結果を表す式（意味動作）を書く

```
main:
   expr EOF {$1}
;

expr:
   | LET var EQ expr IN expr { ELet($2,$4,$6) }
   | arith_expr              { $1 }
;

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

$n: n番目の記号の解析結果

ocamlyaccの使い方

- ocamlyacc FILENAME.mly
  - FILENAME.mliとFILENAME.mlが生成

```
$ ocamlyacc exampleParser.mly
...
$ ls exampleParser.*
exampleParser.ml  exampleParser.mli
exampleLexer.mll
```

注意

- header部やtrailer部に他のモジュールから参照したい型や値を定義しないように
  - 生成される.mliに含まれない
- %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

- 「文法が曖昧なおそれあり」
  - shift/reduce conflict
    - 多くは問題とならない
    - shiftが優先される
    - 優先度などが適切かどうか確認
  - 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ファイルを調べると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ocamlyaccを使わないのも手
    - LR(k)パーサでは文法が曖昧でなくてもconflictが生じる場合あり
      - 文法の曖昧さの検査は決定不能

- 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

生成されたモジュールの使い方

実際に構文解析するには

- 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，開始記号として指定した記号の名前
  - 字句解析関数と入力バッファを取り，解析結果を返す

利用例

```
let main () =
  try
    let lexbuf = Lexing.from_channel stdin in
    let result =
          ExampleParser.main ExampleLexer.token lexbuf in
    print_expr result; print_newline ()
  with
    | Parsing.Parse_error ->
      print_endline "Parse Error!"
```

コンパイル

```
$ ocamlyacc exampleParser.mly
...
$ ocamllex exampleLexer.mll
...
$ ocamlc -c syntax.ml
$ ocamlc -c exampleParser.mli
$ ocamlc -c exampleParser.ml
$ ocamlc -c exampleLexer.ml
$ ocamlc -c example.ml
$ ocamlc -o example syntax.cmo exampleParser.cmo \
                    exampleLexer.cmo example.cmo
$ echo "let y = 3 in y + 4" | ./example
ELet (y,3,EAdd (y,4))
```

第4回レポート課題
締切：5/20の13:00

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
      - exampleLexer.mllやexampleParser.mlyを適当に変更してみる
        - arith_expr等をexprに潰してみる
        - conflictを発生させてみる
      - 標準入力でなく，文字列を構文解析できるようにしてみる
      - ocamlyacc -vの出力を読んでみる

問2

- intとboolを計算する単純な式Eの構文を以下で定める（第2回問5参照）

$$
\begin{aligned}
V &::= 0 \mid 1 \mid 2 \mid \cdots \\
&\quad \mid \texttt{true} \mid \texttt{false} \\
E &::= V \mid E + E \mid E - E \mid E * E \mid E \,/\, E \\
&\quad \mid E = E \mid E < E \\
&\quad \mid \texttt{if } E \texttt{ then } E \texttt{ else } E \mid (E)
\end{aligned}
$$

- この式のインタプリタを実装せよ
  - 標準入力から読んで標準出力に出力

問2：つづき

- ただし，第2回の結果を
  できるかぎり再利用にせよ
  - 構文解析の結果が expr になるように
  - evalは可能なら再利用せよ

問3

- 問2のインタプリタを対話的に利用できるようにせよ
  - 引数にFILENAMEを渡して起動すると
    - FILENAMEを読み，評価結果を出力し終了するように
  - interpreterで起動すると，
    - ocamlのように対話的実行するようにせよ
- 対話入力の構文
  - $C \rightarrow E;;$
- 上記入力に対応する型
  - type command = CExp of expr

問4

- 問4のインタプリタを拡張し，変数を扱えるようにせよ

  （Iを識別子を表すトークンとする）

  E ::= … | I | let I = E in E | …

  C ::= … | let I = E;;

  - まずexprの定義やcmdを変更
  - .mllや.mlyを変更
  - 最後にevalを変更

問4：方針

- evalが環境も引数にとるようにする
  - eval : env -> expr -> value
- 環境：変数から値へのマップ
  - 実装は組のリストでよい
    - List.assocとかあるし

問4：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
  - 新しい環境のもとでe2を評価

- let x = e1;;の評価
  - e1を評価
  - xと評価結果の対応を環境に追加
  - 新しい環境で次の入力を処理
    - 入力待ちloopの引数に…

注意

- 問2～問4は全部まとめて提出してよい
  - ただし，どの問題の要件を満たしているか明示すること
  - もちろん考察はそれぞれ必要
- OCaml標準ライブラリ関数は自由に使用してよい
- Makefileを忘れるなかれ
  - OCamlMakefileを用いてもよい
- conflictは可能な限り消すこと

注意

- 問2～問4の構文解析器は
  付録のw4lexer.mllとw4parser.mly
  を利用したのでよい
  - Week4.tar.gzを展開して得られるWeek4
    以下のq2-4下にある
  - w4parser.mlyはsyntax.mlのexpr型を
    利用している（第2回問5の解答）
    * つまり，自分のexpr型を利用するために
      は適宜修正が必要
  - ただし，これらを利用した旨はレポー
    トに明記のこと

注意

- ocamlyaccの代わりにmenhirを利用してもよい
  - LR(1)構文解析器
    - cf. LALR(1)構文解析器
  - ocamlyaccとの高い互換性
  - 詳細は http://cristal.inria.fr/~fpottier/menhir/
- BNF Converterを使用してもOK
  - http://bnfc.digitalgrammars.com/

発展1

- LRやLL以外の構文解析手法や構文解析器生成器について調べ，まとめよ
  - 例
    - PEGとPackratパーザ
    - 構文解析コンビネータ
    - LL(*)
    - GLR
    - 言語の微分に基づくパーザ
      - dL/da = { w | aw ∈ L }
  - CYKはダメ

発展2

- 構文解析器（生成器 or ライブラリ）を作成せよ
  - 発展1で調べたものでもよい
    - 例：PEG, GLR, LL(*), …
  - 構文解析器生成器とする必要はない
    - 「インタプリタ」でよい
  - 表現力・速さ・便利さのどれに拘ってもよい
    - たとえば BNF Converterは便利である
    - 構文解析コンビネータも便利である
      - でも素朴な実装だと左再帰が…