プログラミング言語 8
字句解析器 (lexer) と構文解析器 (parser)

田浦

はじめに

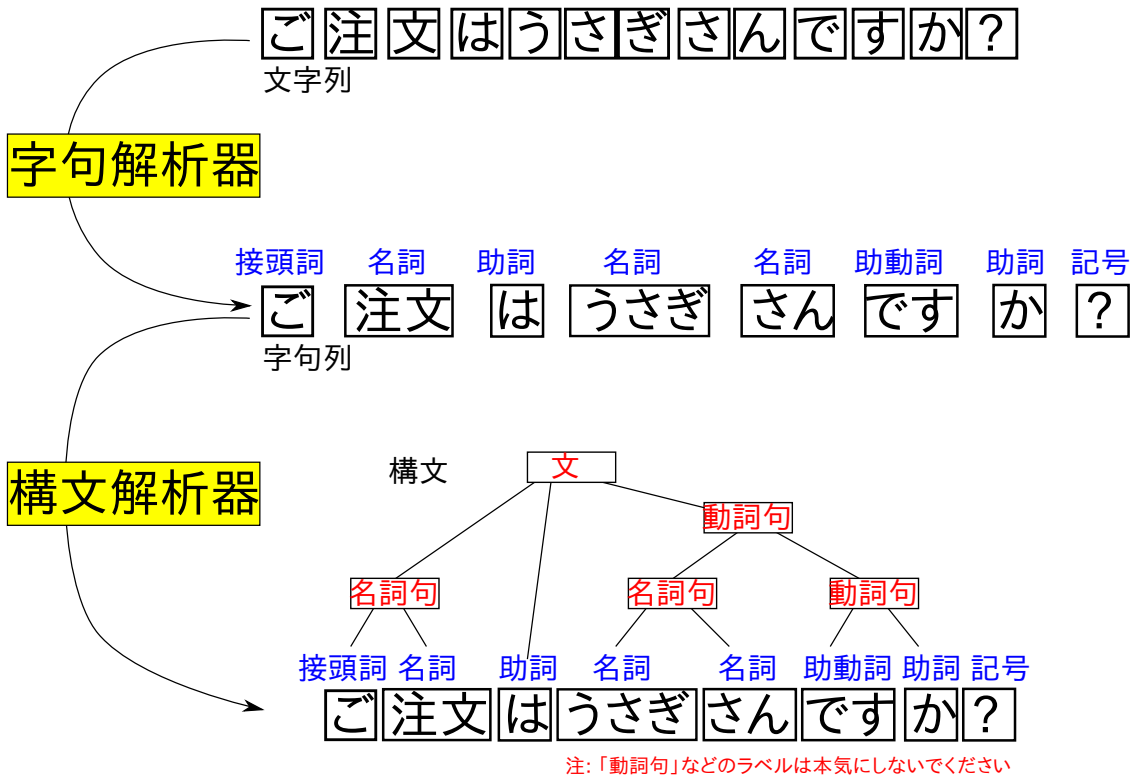
- あらゆるプログラミング言語処理系は，最初にプログラムを読み込み，文法のチェックを行う
  - 字句解析器 (lexer, tokenizer)
  - 構文解析器 (parser)
- それらは,「言語処理系」でなくてもあらゆる場面で必要
  - Web Page (HTML や XML) の読み込み
  - CSV, SVG, . . . ファイル. . .
  - ソフトの config file. . .
- それらを「さっと作れる」ことは実践的にも重要なスキル
  - アドホックに文字列処理をやるだけではきっとうまく行かない
  - そのための便利なツール (生成器) がある
  - 一度使っておいて損はない!

本授業残りのロードマップ

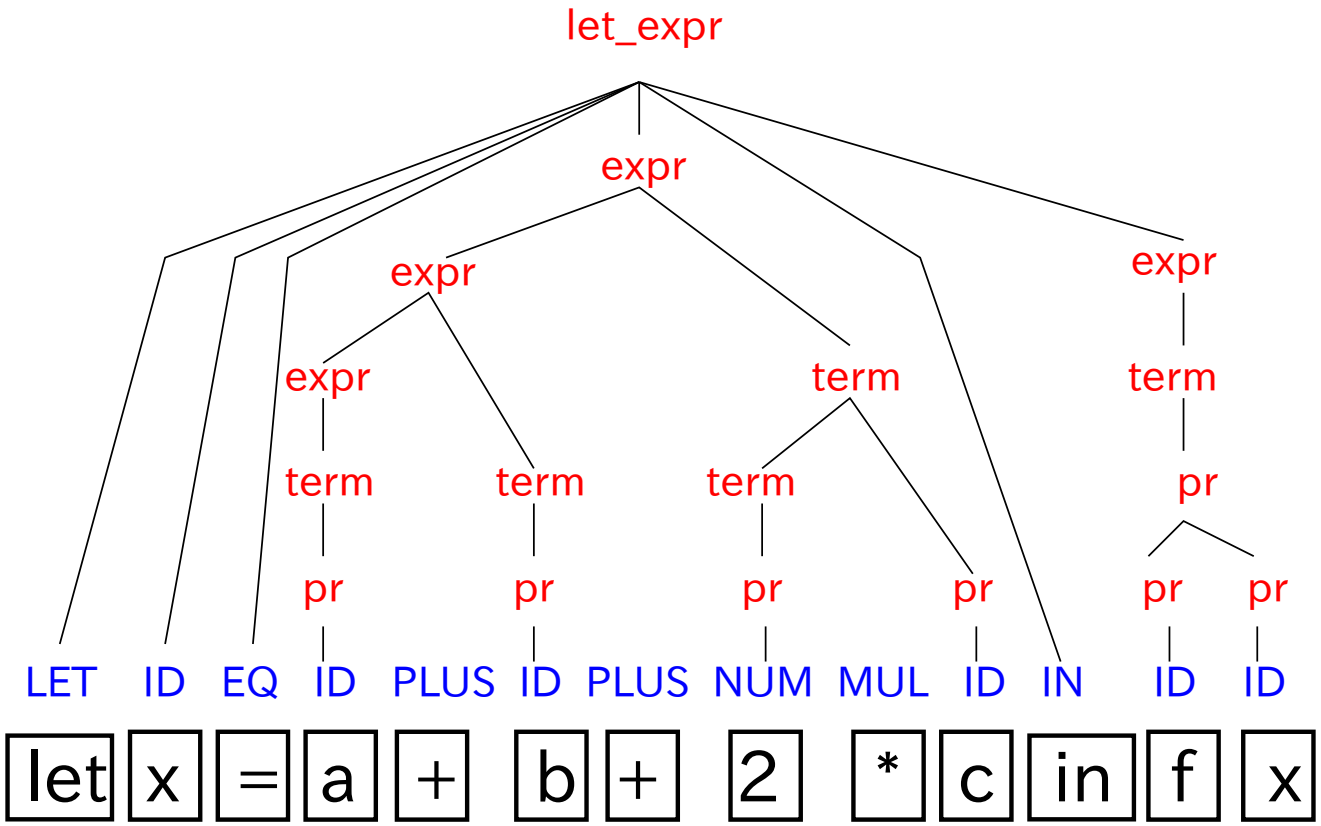
1. 字句解析器，構文解析器生成ツールを使いこなす
   1. OCaml：ocamllex と ocamlyacc
   2. Python：ply
2. それを利用して，ミニ処理系を簡単に作っちゃえ!

字句解析と構文解析

- 字句解析器 ≈
  - 「文字」の列 → 「字句」(≈ 単語) の列
  - 字句にならない文字の列が来たらエラー
- 構文解析器 ≈
  - 「字句」の列 → 文
  - 文にならない字句の列が来たらエラー



プログラミング言語の例



let x = a + b + 2 * c in f x

字句と構文をどう定義するか?

- 通常,
  - 字句: 正規表現
  - 構文: 文脈自由文法

  という枠組みを使って定義する
- 宣言的な記述から，プログラム (字句解析器，構文解析器) を生成するツールがある
- 「習うより慣れろ」実例から入る

ocamllex：字句解析器生成ツール

- ocamllex の入力 = .mll ファイル (OCaml 風だが同じではない)
- 例 (calc_lex.mll)

```
1  { (* 前置き:  任意の OCaml コード. 無くても可 *)
2  type token =
3    NUM of (int)
4  | PLUS
5  | EOF
6  }
7
8  (* 本題 *)
9  rule lex = parse
10 | [' ' '\t' '\n'] { lex lexbuf }          (* 空白を読み飛ばす *)
11 | "+"              { PLUS }
12 | ['0'-'9']+ as s { NUM(int_of_string s) }
13 | eof              { EOF }
14
15 { (* 任意の OCaml コード. 無くても可 *)
16
17 }
```

.mll ファイルの形式

```
{
  任意のOCaml コード
  通常は ,「 字句 」のデータ型を定義
}

(* 規則 *)
let id = 正規表現
  ...

rule lex = parse
| 正規表現 { 式 }
| 正規表現  { 式 }
   ...
| 正規表現  { 式 }

{
  任意のOCaml コード
}
```

- |「正規表現 { 式 }」 の意味:

  入力の先頭辞 (prefix) が「正規表現」にマッチしたら,「式」を評価して返す (それがひとつの字句)

- 「正規表現 as 変数名」 で，規則中で，マッチした文字列を変数で参照できる

```
1 | ['0'-'9']+ as s { NUM(int_of_string s) }
```

- 後に使う正規表現に名前を付けられる

```
1 let digit = ['0'-'9']
```

```
1 | digit+ as s { NUM(int_of_string s) }
```

ocamllexの正規表現の例

正規表現	意味 (マッチする文字列)
`_`	任意の1文字 (アンダースコア)
`'a'`	a
`[ 'a' 'b' 'c' ]`	a, b, c どれか
`['0'-'9']`	0, 1, ..., 9 どれか
`"abc"`	abc
`"abc"|"def"`	abc または def
`"abc"*`	abc が0回以上繰り返された文字列
`"abc"+`	abc が1回以上繰り返された文字列
`("abc"|"def")+`	(abc または def) が1回以上
`eof`	入力の終わり

- 一覧は http://caml.inria.fr/pub/docs/manual-ocaml-400/manual026.html (英語) または http://ocaml.jp/archive/ocaml-manual-3.06-ja/manual026.html (日本語) を見ましょう

参考: 正規表現のフォーマル (本質的) な定義

- 列を構成する文字 (アルファベット) の集合を A とする
- 以下が A 上の正規表現

正規表現		意味 (マッチする文字列)
ϵ		空文字列
a	$(a \in A)$	a
RS	(R, S は正規表現)	R にマッチする文字列と S にマッチする文字列の連接
$R \mid S$	(R, S は正規表現)	R または S にマッチする文字列
$R*$	(R は正規表現)	R にマッチする文字列の 0 回以上の繰り返し

- 必要に応じて括弧を使う (例: `(abc|def)+`)
- 前述のあらゆる例は上記の組み合わせ (またはその省略記法)

ocamllexが生成するファイルと関数

- ocamllexは字句解析の定義ファイル(.mll)から，OCamlのファイル(.ml)を生成する

```
1 $ ocamllex calc_lex.mll
2 6 states, 267 transitions, table size 1104 bytes
3 $ ls
4 calc_lex.ml  calc_lex.mll
```

- .mlファイル内に関数`lex`が定義される(.mll内の`rule lex = parse ...`に対応)

```
1 $ ocaml -init calc_lex.ml
2         OCaml version 4.01.0
3
4 # lex ;;
5 - : Lexing.lexbuf -> token = <fun>
```

- `Lexing.lexbuf`は，文字を読み出すためのバッファ(≈Cの`FILE*`)．mutableなrecord
- `lex` *buf*は，*buf*の先頭から文字列を消費し，字句を返す

Lexing.lexbuf の作り方いろいろ

- 文字列から

```
1 Lexing.from_string "12+34* 56"
```

- 標準入力から

```
1 Lexing.from_channel stdin
```

- ファイルから

```
1 Lexing.from_channel (open_in "exp.txt")
```

字句解析器使用例

```
$ ocamllex calc_lex.mll
6 states, 267 transitions, table size 1104 bytes
$ ocaml -init calc_lex.ml
        OCaml version 4.01.0

# let b = Lexing.from_string "12 + 34+56";;
val b : Lexing.lexbuf =
  { ... (省略) ... }
# lex b;;
- : token = NUM 12
# lex b;;
- : token = PLUS
# lex b ;;
- : token = NUM 34
# lex b ;;
- : token = PLUS
# lex b;;
- : token = NUM 56
# lex b;;
- : token = EOF
```

```
rule lex = parse
| [' ' '\t' '\n'] { lex lexbuf }
| "+"             { PLUS }
| ['0'-'9']+ as s { NUM(int_of_string s) }
| eof             { EOF }
```

ocamlyacc：構文解析器生成ツール

```
/* 宣言 + 任意のOCaml コード*/
%{
  (* 任意のOCaml コード *)
%}
/* 字句の定義 */
%token <int> NUM
%token PLUS EOF

/* 先頭記号とその型（必須） */
%start program
%type <int> program

%%  /* 文法定義と評価規則 */
expr :
| NUM              { $1 }
| expr PLUS NUM    { $1 + $3 }

program :
| expr EOF         { $1 }

%%
  (* 任意のOCaml コード *)
```

- 入力 = .mly ファイル
- 形式: %% で 3 分割
  - 宣言 + 任意の OCaml コード
  - 文法定義と評価規則
  - 任意の OCaml コード
- 宣言
  - %token：全字句名と各字句に付随するデータの型
  - %start：先頭記号 (入力全体に対応する記号) 名
  - %type：各記号が認識されたときに対応して返す型 (先頭記号については必須)
- 注: .mll と.mly 両方で字句の定義をしている (マシなやり方は後述)

ocamlyacc と menhir

- ocamlyacc とほぼ互換で，新しいツールとして menhir がある
- この説明の範囲ではどちらでも同じ
- 演習では ocaml をインストールすれば自動的にインストールされる ocamlyacc を使う

ocamlyacc の文法定義

- 文脈自由文法に沿った定義
- 例:

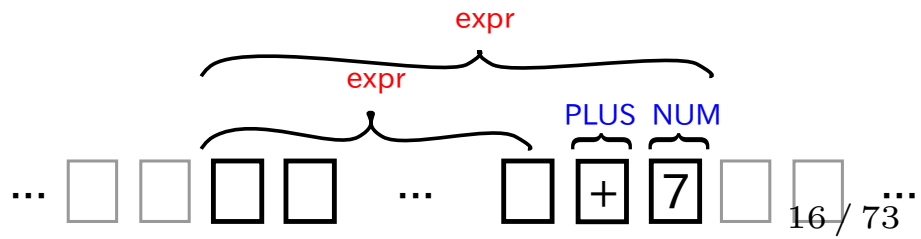
```
1 expr :
2 | ...
3 | expr PLUS NUM  { ... }
```

の読み方:

- expr (にマッチする字句列),
- PLUS (1 字句)
- NUM (にマッチする字句列),

をつなげたものは，expr である (にマッチする).

- | で，複数の可能性があることを示す.
- { ...} は,「評価規則」(後述)
- 注:
  - 右辺で自分自身を参照しても良い (再帰)
  - 複数の記号がお互いを参照していても良い (相互再帰)



文脈自由文法のフォーマルな定義

- 終端記号 (字句) の集合: T
- 非終端記号の集合: NT
- 先頭記号: $S \in NT$
- 規則の集合. 一つの規則は,

$$a = b_1 \ \cdots \ b_n$$

の形 ($n \geq 0$, $a \in NT$, $b_i \in NT \cup T$).

  - この規則の意味:
    - ★ b_1 にマッチする字句列,
    - ★ ...
    - ★ b_n にマッチする字句列,

    をつなげた字句列は，a にマッチする
- おそらく言わずもがなだが厳密さのため:
  - 上記で b_i が字句の場合，b_i はその 1 字句 (からなる字句列) に (のみ) マッチする

評価規則

- 任意の OCaml の式．ただし，`$1`，`$2`，...などで，右辺の対応する位置にある記号に対する値を参照できる
- 意味: 入力中のある部分字句列が，規則 $a = b_1 \ \ldots \ b_n$ により a にマッチしたら，対応する「評価規則」を計算し，その字句列に対応する値として保存する
- 例:

```
1 expr :
2 |  ...
3 | expr PLUS NUM  { $1 + $3 }
```

読み方: ある部分字句列が `expr PLUS NUM` にマッチしたら，その部分字句列に対応する値は，

- 右辺 1 番目の `expr`(にマッチした字句列) に対応する値
- 右辺 3 番目の `NUM`(にマッチした字句) に対応する値

の和である

ocamlyacc が生成するファイル

- ocamlyacc は.mly から，2 つの OCaml のファイル (.ml と.mli) を生成する
  - .mli って? ⇒ 後述

```
1 $ ocamlyacc calc_parse.mly
2 $ ls
3 calc_parse.ml calc_parse.mli  calc_parse.mly
```

- .ml ファイル内に，先頭記号名で，関数が定義される
  - つまりここでは，.mly 内の`%start program` に対応し，`program` という関数が定義される

ocamlyacc が生成する構文解析器 (関数)

```
1  $ ocaml -init calc_parse.ml
2           OCaml version 4.01.0
3
4  # program ;;
5  - : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int = <fun>
```

- Lexing.lexbuf -> token は字句解析器の型
- int は，.mly 内 (%type <int> parse) で指定した型
- 構文解析器は，
  - 字句解析器と文字バッファを受け取り，
  - 字句解析器によって，文字バッファから次々と token を取り出し，
  - token 列全体が先頭記号とマッチするか計算し，
  - マッチしたら評価規則によって (token 列全体に対応して) 計算された値を返す

字句解析と構文解析を合体させる

- 以上で字句解析器 (lex) と構文解析器 (program) ができた
- 以下のようにして組み合わせて動くことを期待したくなる

```
1 # program lex (Lexing.from_string "12+ 34 - 56")
```

- 残念ながらそうは行かない．理由:
  1. 両者は別々のファイルに書かれている．互いを参照するための「お作法」が必要
  2. もっと面倒な理由: .mll から生成された `token` と，.mly から生成された `token` を，そのままでは「同じもの」と思ってくれない

字句解析内の token ≠ 構文解析内の token

- .mll

```
1 $ ocaml -init calc_lex.ml
2 # lex;;
3 - : Lexing.lexbuf -> token = <fun>
```

  この token は，`calc_lex.ml` 中の `token`

- .mly

```
1 $ ocaml -init calc_parse.ml
2 # program ;;
3 - : (Lexing.lexbuf -> token) -> Lexing.lexbuf -> int = <fun>
```

  こちらは `calc_parse.ml` 中の `token`

- 同じ名前でも別のもの．定義が一致していても別のもの
- 一見理不尽だが，一般に OCaml では，他のファイル中の定義を参照するには，お作法が必要なのでこうなる

token 定義の不一致の解決法

- 方針 1: .mll に対して，
  「お前は token を定義するな．.mly にあるやつを使ってね」と指示する
- 方針 2: .mly に対して，
  「お前は token を定義するな．.mll にあるやつを使ってね」と指示する
- どちらでもできるが，以下では一旦方針 1 を説明

token 定義の不一致の解決法

- calc_lex.mll を以下のように変更:

```
1  {
2    (* ここにあった token 定義を除去 *)
3    (* 以下のおまじないで，PLUS などは
4       calc_parse.ml 内のものを参照できる (する)
5       ようになる *)
6  open Calc_parse
7  }
8  rule lex = parse
9  | [' ' '\t' '\n']  { lex lexbuf }
10 | "+"              { PLUS }
11 | ['0'-'9']+ as s  { NUM(int_of_string s) }
12 | eof              { EOF }
```

- 「おまじない」の意味は後に説明

合体して動かす実際の手順

- それをやってもなお，残念ながら以下ではどれも動かない

```
1 $ ocaml ocaml_lex.ml ocaml_parse.ml  # NG
2 $ ocaml -init ocaml_lex.ml -init ocaml_parse.ml # NG
```

- 理由: ocaml コマンドは複数の.ml ファイルを受け付けない
- ocaml コマンドは，.ml ファイルを直接実行するコマンドだと思わないほうが心の平穏を保てる
- 事前に ocamlc というコマンドで，「コンパイル」したもの (.cmo) を渡すのが基本

合体して動かす実際の手順

```
$ ocamllex calc_lex.mll
$ ocamlyacc calc_parse.mly
  # ocamlc でコンパイル．以下の 3 ファイルの順序重要!
  # parse が先，lex が後
$ ocamlc -c calc_parse.mli calc_parse.ml calc_lex.ml
  # ocaml に.cmo を渡す
$ ocaml calc_parse.cmo calc_lex.cmo
        OCaml version 4.01.0

# Calc_parse.program;;
- : (Lexing.lexbuf -> Calc_parse.token) -> Lexing.lexbuf -> int = <fun>
# Calc_lex.lex;;
- : Lexing.lexbuf -> Calc_parse.token = <fun>
```

- これでめでたく両者が「整合」

```
# Calc_parse.program Calc_lex.lex (Lexing.from_string "12+34 + 56");;
- : int = 102
```

要点

- OCaml 世界では，直接 .ml を実行するのは例外と思うが吉
- .ml を .cmo (バイトコード) にコンパイルし，ocaml に与えるのが基本
- そして，複数ファイルからなるプログラムの場合，それが「必須」になる
- .cmo を作るには，`ocamlc -c` で「コンパイル」すればよいが，引数 (ないしコマンド実行) の順番が重要
- ルール: 「依存するファイルを後に書く」
  - `calc_lex.ml` が `calc_parse.ml` 中の token を参照 → `calc_parse.ml calc_lex.ml` の順

楽な方法：ocamlbuild

- OCaml 専用のビルドツール
- 何してるかわからない長大なコマンド列が不愉快 (だが一応便利)

```
$ ocamlbuild calc_lex.byte
/usr/bin/ocamllex -q calc_lex.mll
/usr/bin/ocamldep -modules calc_lex.ml > calc_lex.ml.depends
/usr/bin/ocamlyacc calc_parse.mly
/usr/bin/ocamldep -modules calc_parse.mli > calc_parse.mli.depends
/usr/bin/ocamlc -c -o calc_parse.cmi calc_parse.mli
/usr/bin/ocamlc -c -o calc_lex.cmo calc_lex.ml
/usr/bin/ocamldep -modules calc_parse.ml > calc_parse.ml.depends
/usr/bin/ocamlc -c -o calc_parse.cmo calc_parse.ml
/usr/bin/ocamlc calc_parse.cmo calc_lex.cmo -o calc_lex.byte
$ ls
_build/ calc_lex.byte   calc_lex.mll  calc_parse.mly
  # 生成物は全て，_build フォルダ内にある
  # -I _build という，またおまじない
$ ocaml -I _build _build/*.cmo
        OCaml version 4.01.0
#
```

ocamlmktop

- *.cmo を作った後，毎回

```
$ ocaml -I _build _build/*.cmo
        OCaml version 4.01.0
#
```

のように，それらを指定して ocaml を起動する代わりに，

```
$ ocamlmktop -o calc.top _build/*.cmo
```

として，それらの*.cmo を「焼入れ」した，対話的処理系を指定した名前 (上記では calc.top) で生成することができる

```
$ ./calc.top -I _build
        OCaml version 4.01.0
#
```

OCamlで複数ファイルからなるプログラムを作る際の最低限の知識のまとめ

- 他のファイル (例: `abc.ml`) で定義される名前 (関数/変数名，型名，型のコンストラクタ名，etc.) を参照する場合，
  - 方法 1: 参照するたびに名前を「`Abc.` 名前」のように参照する
  - 方法 2: 先頭に，`open Abc` と書く
- 前述したとおり,「依存関係」の順に ocamlc でコンパイルする
- ocaml や，ocamlmktop で生成した処理系は，`*.cmi` や`*.cmo` を探す場所を，`-I` で指定する

文法定義でよく問題となる事項 (1)
左結合と右結合

- 先の文法定義

```
1 expr :
2 | NUM
3 | expr PLUS NUM  { $1 + $3 }
```

は，以下ではいけないのだろうか?

```
expr :
| NUM
| NUM PLUS expr  { $1 + $3 }
```

```
expr :
| NUM
| expr PLUS expr  { $1 + $3 }
```

- 元々の規則は，足し算 (+) が,「左結合 (left associative)」であることを反映した規則
- 左結合:

$$a + b + c = ((a + b) + c)$$

文法定義でよく問題となる事項 (2)
優先度の処理

- * (掛け算) を扱えるようにしたとする
- 以下では何かまずいか?

```
1 expr :
2 | NUM
3 | expr PLUS NUM  { $1 + $3 }
4 | expr MUL NUM  { $1 * $3 }
```

- この定義では,

$$3 + 4 * 5 = (3 + 4) * 5 = 35$$

- 「掛け算の方が足し算より強い」という規則を文法に反映させたい

$$3 + 4 * 5 = 3 + (4 * 5) = 23$$

- 適宜記号を追加して文法を変更する
  - 「数」が*で結合されて「項」(term) になり
  - 「項」が+で結合されて「式」(expr) になる

注:

- ocamlyacc には，これらの問題を宣言的に解決する記法も用意されている
  - `%left` (左結合)
  - `%right` (右結合)
  - それらを書く順番で優先度の高さを明示
- もしそれらを使いたければ，マニュアル参照
- でも，これらの記号の本当の意味がわからなければ，素直に自分で記号を増やせば良い

演習の構成

- 3 種類のサンプルファイル
- どれも「整数を足し算する式」だけを受け付ける
  1. 字句解析器だけ (1 ファイル)
  2. 字句解析器，構文解析，両者を使う OCaml プログラム (3 ファイル)．構文木を作らない
  3. 構文木定義，字句解析器，構文解析，両者を使う OCaml プログラム (4 ファイル)．構文木を作る．
- それらを拡張する
  - 浮動小数点数
  - 引き算，掛け算，割り算 (演算子の優先度処理)
  - 括弧
  - let 式 (変数)

電卓からプログラミング言語へ

- プログラミング言語 ≈
  - 電卓 (基本的な数の演算)
  - ＋ 変数 (計算結果に名前を付けて利用)
  - ＋ 関数
  - ＋ 数以外のデータ型
- そのためのステップ
  - ステップ 0: 構文木の導入 (構文解析と評価の分離)
  - ステップ 1: 変数 (let) の導入
  - ステップ 2: 関数の導入

構文木

- 簡単な電卓であれば，構文解析器が，文字列から直接その評価値を求められる
- 以下の評価規則もそれに相当する

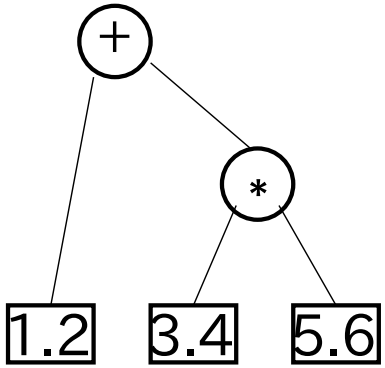
```
1 expr :
2 | NUM            { $1 }
3 | expr PLUS NUM { $1 +. $3 }
```

```
1 # parse_string "1.2 + 3.4 * 5.6"
2 - : float = 20.24
```

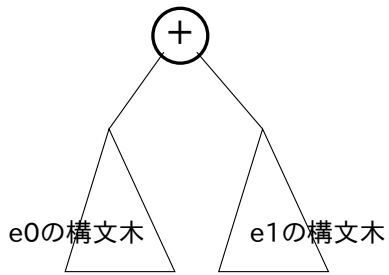
- より複雑な場合，構文解析器は，文字列を，適切なデータ構造 (構文木) に変換することに専念
- 別途，構文木から値を求める関数 (評価器) を作る
  - 構文解析器：文字列 $\rightarrow$ 構文木
  - 評価器：構文木 $\rightarrow$ 評価値

構文木

- 文字列を解析した結果を素直にデータ構造にしたもの
- 例: `1.2 + 3.4 * 5.6` の構文木:



- 一般に，$e_0 + e_1$ の構文木は:

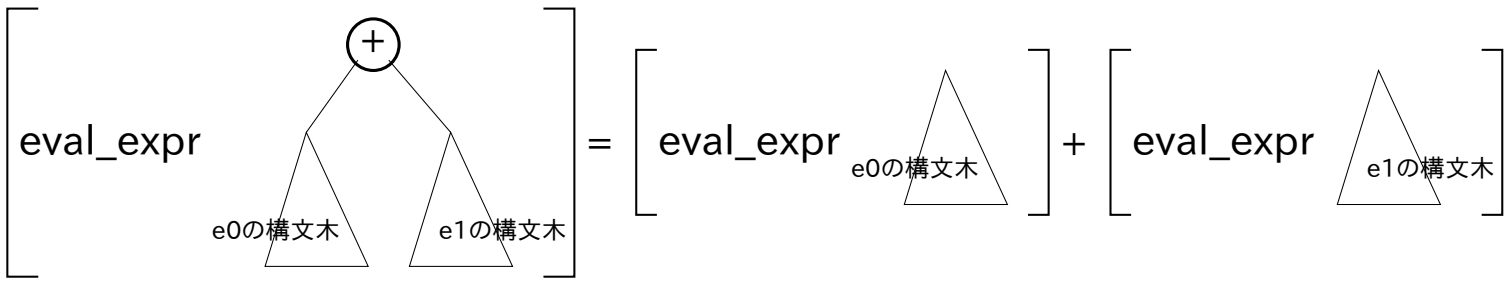


- OCaml では，以下のようなバリアント型で自然に表現できる

```
1 type expr =
2     ...
3   | Plus of expr * expr
```

評価器

- 多くのケースは，その部分式を再帰的に評価して，それを元に (普段意識していないが知っているはずの言語の仕様に従い)，全体の値を計算するだけ.



- 記号に書けば:

$$\text{eval\_expr}\ (e_0 + e_1) = (\text{eval\_expr}\ e_0) + (\text{eval\_expr}\ e_1)$$

- OCaml では:

```
1 let rec eval_expr e =
2   match e with
3       ...
4     | Plus (e0, e1) -> (eval_expr e0) + (eval_expr e1)
```

変数

- `x + 1` の値を評価するには，(当たり前だが)`x` の値を知る必要がある
- もはや構文木だけでその値が決まるわけではない

NG: 評価器 : 構文木 → 値

- `let x = 1.1 + 2.2 in x + 4.4` を評価するには，
  - `1.1 + 2.2` を評価し，
  - その結果: `x = 3.3` であることをどこかに覚えておき，
  - その元で，`x + 4.4` を評価する
- 変数の値を覚えておく「どこか」のことを「環境」という

評価器 : 構文木 → 環境 → 値

環境

- 環境：「変数とその値」の組を覚えておくデータ構造．抽象的な表記:

$$\{x \mapsto 3.3, y \mapsto 4.4\}$$

- 環境を用いた，`let x = 1.1 + 2.2 in x + 4.4` の評価:

$$\begin{aligned}
& \texttt{eval_expr (let x = 1.1 + 2.2 in x + 4.4)}\ \{\} \\
= \ & \texttt{eval_expr (x + 4.4)}\ \{\texttt{x} \mapsto 3.3\} \\
= \ & \texttt{eval_expr x}\ \{\texttt{x} \mapsto 3.3\} + \texttt{eval_expr 4.4}\ \{\texttt{x} \mapsto 3.3\} \\
= \ & 3.3 + 4.4 \\
= \ & 7.7
\end{aligned}$$

- 環境をデータ構造として実現するには色々な仕方があるが，連想リストを使うのが簡単

```
1 [ (x, 3.3); (y, 4.4) ]
```

関数

- 関数呼び出し式の評価は，環境をうまく使えばすぐにできる
- `f` がどこかで `f x = x * x` と定義されているとする
- そのもとで，`(f 3)` を評価することは，$\{\texttt{x} \mapsto 3\}$ という環境下で，`x * x` を評価するのと同じ
- 一般に関数適用の評価は，以下でだいたい正解

$$
\begin{aligned}
& \texttt{eval\_expr}\ (e_0\ e_1)\ e \\
= \quad & \texttt{let f = eval\_expr}\ e_0\ \texttt{in} \\
& \texttt{let v = eval\_expr}\ e_1\ \texttt{in} \\
& \texttt{eval\_expr}\ (f \text{ の定義式})\ \{f \text{ の引数} \mapsto v\}
\end{aligned}
$$

- 評価結果としての関数は，引数名と定義式の組として表せば良い

関数 = 引数名, 定義式 (の構文木) の組

だいたいとは?

- 関数がトップレベルに限らない，任意の場所で定義される言語ではもう少し複雑
- 関数が,「自身が定義された時の環境」を覚えておく必要がある
- 例:

```
1 let make_adder x =
2   let f y = x + y in f
3 ;;
4 let a11 = make_adder 1.1 in
5   a11 2.2
```

  - a11 2.2 を評価するのに，x + y を環境 $\{y \mapsto 2.2\}$ で評価するのではダメ (x の値が欠けている)
  - この例では x = 1.1 で，それは，a11 が生まれた (定義された) 時 (2 行目) の値
  - 関数は,「生まれた時の環境を覚えている」

関数 = 引数名, 定義式 (の構文木), 定義時の環境の組

付録: 正規表現と文脈自由文法についてもう少し

- どちらも「アルファベット列」の集合を定義する枠組み
  - ほんとに 2 つ必要?
  - 全てを正規表現で，または全てを文脈自由文法では書けないの?

正規表現は文脈自由文法に包含される

正規表現	対応する文脈自由文法
ϵ	$A =$
a	$A = a$
RS	$A = RS$
$R \mid S$	$A = R$, $A = S$
$R*$	$A =$, $A = A\ R$

- 上の要領で，適宜新しい記号を導入していけば，任意の正規表現と等価な文脈自由文法を作ることが可能

文脈自由文法は本当に正規表現より強力か?

正規表現	対応する文脈自由文法
ϵ	$A =$
a	$A = a$
RS	$A = R\ S$
$R \mid S$	$A = R$, $A = S$
$R*$	$A =$, $A = A\ R$

- 正規表現は文脈自由文法の規則の右辺に「ある」制限を課したもの．どんな制限か?
  - 規則に再帰的な要素が一切なければ，正規表現で書ける
  - 再帰的な定義が,
    - ★ $A =$, $A = AR$
    - ★ $A =$, $A = RA$

    (R は A に依存していない記号) という形のものだけであれば正規表現で書ける
  - 「再帰がこの形しかない」ところに違いがありそう

文脈自由文法は正規表現に包含されるか?

- アナロジー: 正規表現 = ループはあるけど一般的な再帰がない
- たとえば一見，正規表現では書けなさそうに思えるもの
  - $A =$, $A = aAb$
  - $a^n b^n$ $(n \geq 0)$ という文字列とマッチ
- 本当に書けないことを証明しようとすると難しいが...

$A =, A = aAb$ は正規表現では書けない

- そのような正規表現があったとしたら少なくともひとつ$*$を含む (さもなければマッチする文字列の長さはある一定値以下)
- そして，それにマッチする十分長い文字列は，その$*$ の部分を 1 回以上は繰り返しているはず
- その部分を任意回繰り返したものもまたマッチする
- 十分大きな k に対して，$a^k b^k$ のどこがその「部分」になりうるか?
  - その部分が a (または b) だけを含む $\rightarrow$ そこだけを繰り返せば，a (または b) の数が多くなってしまう

    $aa\cdots\boxed{aaa}\cdots aabb\cdots bb$

    $\rightarrow \quad aa\cdots\boxed{aaa}\boxed{aaa}\cdots aabb\cdots bb$
  - その部分が ab 両方を含む $\rightarrow$ そこだけを繰り返せば，$a\cdots b\cdots a$ という文字列ができてしまう

    $aa\cdots\boxed{aabbb}\cdots bb$

    $\rightarrow \quad aa\cdots\boxed{aabbb}\boxed{aabbb}\cdots bb$

OCamlで複数ファイルからなるプログラムを，動かすときの理屈を一から理解するための付録

OCamlでの複数ファイルプログラム開発 (≈分割コンパイル)

ここでの動機:

- ocamllex, ocamlyacc はそれぞれ，字句定義，構文定義から，それを受け付ける OCaml のプログラム (.ml ファイル) を生成する
- 実際のアプリは，それらと，本体の OCaml プログラムを組み合わせて作る
- → 嫌でも複数ファイルからなるプログラムになる
- 知っておかないといけない規則は複雑かつ意外性に富んでおり，イライラの元なので易しく解説

OCamlプログラムの3つの実行方法

まず有権者に訴えたいのは，OCamlプログラムには以下の3つの実行方法があるということ

1. 直接実行，対話的実行 (ocaml)
2. バイトコードへコンパイル (ocamlc); 実行
3. ネイティブコードへコンパイル (ocamlopt); 実行

単一ファイルなら簡単

- 例: 以下のプログラム (hi.ml)

```
1 Printf.printf "hello\n"
```

- 直接

```
1 $ ocaml hi.ml
2 hello
```

- 対話的

```
1 $ ocaml
2 # #use "hi.ml";;
3 #
```

```
1 $ ocaml -init hi.ml
2 #
```

- バイトコード

```
1 $ ocamlc hi.ml
2 $ ls
3 a.out* hi.cmi hi.cmo hi.ml
4 $ ./a.out
5 hello
```

- ネイティブコード

```
1 $ ocamlopt hi.ml
2 $ ls
3 a.out* hi.cmi hi.cmx hi.ml hi.o
4 $ ./a.out
5 hello
```

色々な生成物

名前	説明	ocamlc	ocamlopt
.cmi	hi.ml の「インタフェース」(≈ hi.ml 中で定義されている名前と型)	○	○
.cmo	hi.ml をバイトコード化した本体 (≈ .o ファイル)	○	
.cmx	hi.ml をネイティブコード化したもの (≈ .o ファイル)		○
.o	正真正銘のオブジェクトファイル		○
a.out	実行可能ファイル	○	○

ディレクトリがかなり，とっ散らかります

-c とか-o とか

ocamlc, ocamlopt とも，以下は普通の C コンパイラと同じ

- “-o ファイル名” で出力ファイル名が指定できる
- “-c” で，実行可能ファイルまで出さずに，オブジェクトファイルまでで終了
  - ocamlc : {.cmi, .cmo}
  - ocamlopt : {.cmi, .cmx, .o}

複数ファイルへの第一歩

- 例: 以下の 2 つのファイル． 同一ディレクトリにあるとする
  - msg.ml

```
1 let greeting = "hello"
```

  - hi.ml

```
1 Printf.printf "%s\n" Msg.greeting
```

- hi.ml が msg.ml 内の greeting の定義を参照している (前者が後者に依存している)
- 規則 1:

  他のファイルで定義された名前 *(msg.ml* の `greeting`*)* は，*Msg.greeting* などとして参照する

- 「モジュール名. 名前」で参照
- 「モジュール名」= ファイル名の basename (.ml を除いた部分) を capitalize したもの

open 節

- 「open モジュール名」という句をファイル内に書いておけば，名前だけで直接参照も可能

```
1 open Msg;;
2 Printf.printf "%s\n" greeting
```

- 簡潔で良いが，
  - 他人が読む場合どのモジュールの機能を呼んでいるのかわかりにくい
  - 全容がわからないモジュールを，安易に open すると，名前衝突の危険性がある

バイトコードの場合の手順

- 方法 1: 一度に全てコンパイル

```
1 $ ls
2 hi.ml  msg.ml
3 $ ocamlc -o greet msg.ml hi.ml
4 $ ls
5 greet* hi.cmi hi.cmo hi.ml msg.cmi msg.cmo msg.ml
```

- 方法 2: 一個ずつ (分割) コンパイル

```
1 $ ls
2 hi.ml  msg.ml
3 $ ocamlc -c msg.ml                  # -> msg.{cmi,cmo}
4 $ ocamlc -c hi.ml                   # -> hi.{cmi,cmo}
5 $ ocamlc -o greet msg.cmo hi.cmo    # -> greet
6 $ ./greet
7 hello
```

- 前者は，後者を一コマンドでやっているに過ぎない

失敗する手順とその理由

- 失敗 1: 複数一度にコンパイルする場合で，依存関係を持つものを先に書くという失敗

```
$ ocamlc hi.ml msg.ml
File "hi.ml", line 1, characters 21-33:
Error: Unbound module Msg
```

- 失敗 2: 分割コンパイルする場合で，他に依存しているものを最初にコンパイルするという失敗

```
$ ls
hi.ml  msg.ml
$ ocamlc -c hi.ml
File "hi.ml", line 1, characters 14-26:
Error:  Unbound module Msg
```

- 規則 2:

あるファイル *(例: hi.ml)* をコンパイルする際，それが参照するモジュールの *.cmi (例: msg.cmi)* が存在していなくてはならない

まとめると

hi.ml が msg.ml を参照している (に依存している) 時,

- バイトコード

OK	NG
`$ ocamlc msg.ml hi.ml`	`$ ocamlc hi.ml msg.ml`
`$ ocamlc -c msg.ml` `$ ocamlc -c hi.ml`	`$ ocamlc -c hi.ml` `$ ocamlc -c msg.ml`

- ネイティブコード (理屈は全く同じ)

OK	NG
`$ ocamlopt msg.ml hi.ml`	`$ ocamlopt hi.ml msg.ml`
`$ ocamlopt -c msg.ml` `$ ocamlopt -c hi.ml`	`$ ocamlopt -c hi.ml` `$ ocamlopt -c msg.ml`

対話的処理系 (ocaml) の場合

- ステップ 1: ocamlc で，全部.cmo にコンパイルする (依存関係に気をつけて)
- ステップ 2:

```
$ ocaml msg.cmo hi.cmo
#
```

- ここでも依存関係のないものから並べる．以下は NG

```
$ ocaml hi.cmo msg.cmo
File "_none_", line 1:
Error: Reference to undefined global ‘Msg’
```

- ocamlc を使わずに，.ml だけで話を済ませることができないか，と願ってもそんなものはない

ocamlmktop

- ocamlmktop というコマンドで，.cmo ファイルを組み込んだ対話的処理系を作れる

```
1 $ ocamlmktop -o greet msg.cmo hi.cmo
2 $ ./greet
3 hello
4         OCaml version 4.01.0
5 #
```

- 依存関係の順に，.cmo ファイルを毎回並べるなんて下衆の極み，という人向け

話はまだ終わりぢゃないヨ：インタフェース

- OCaml には，2 種類のソースファイル (.ml と.mli) がある
  - .ml：実装 (≈ C の.c)
  - .mli：インタフェース (≈ C の.h)
- .mli には，対応する.ml で定義されている名前のうち，「外に見せたいもの (だけ)」の型 (だけ) を書く
- 例:
  - `msg.ml`

```
1 let real_mind = "you a** h*le"
2 let greeting = "glad to see you"
```

  - `msg.mli` (`real_mind` は見せない)

```
1 val greeting : string
```

.mli は何のためのもの?

- モジュールのドキュメント
- モジュールの実装の詳細を隠蔽
- → あるモジュールのユーザが，明示的に「外部に見せる」ことにした機能だけに依存していることを保証する
- → 後から実装を変えやすく，変えても他へ影響を与えずに動く可能性を高くする
- ソフトウェアの作り方として推奨される
- だがここではそれとは関係なく，ocamlyacc が.mli を生成するので，それに対処するために仕方なく説明している

.mli のある・なしでコンパイラの挙動が違う

- 原則は，.mli から.cmi が生まれ，.ml から.cmo が生まれる

```
1 $ ocamlc a.ml
```

は，

1. a.mli が存在しない場合
   - ★ a.cmi, a.cmo の両方を生む
   - ★ a.cmi は a.ml で定義される名前を「全部見せます」なものになる
2. a.mli が存在する場合
   - ★ a.cmi がなければエラー (事前に a.mli から作っとけ!)
   - ★ あれば，.ml と.cmi の整合性をチェックしながらコンパイル

規則 3:

あるファイル *(例: msg.ml)* をコンパイルする際，対応するインタフェースファイル *(例: msg.mli)* が存在するならば，それをコンパイルしたもの *(例: msg.cmi)* が存在しなくてはならない

.mli を含めた正しいコンパイル手順

- 例: 6 個のファイル a.{mli,ml}, b.{mli,ml}, c.ml, d.ml
- まず依存関係を調べる．図のようだったとする
- 方法 1: 一度にコンパイル

```
$ ocamlc -o program b.mli a.mli b.ml a.ml d.ml c.ml
```

- 方法 2: 一個ずつコンパイル

```
$ ocamlc -c b.mli
$ ocamlc -c a.mli
$ ocamlc -c b.ml
$ ocamlc -c a.ml
$ ocamlc -c d.ml
$ ocamlc -c c.ml
$ ocamlc -o program b.cmo a.cmo d.cmo c.cmo
```

- 処理系と対話したければ,

```
$ ocaml b.cmo a.cmo d.cmo c.cmo
```

または,

```
$ ocamlmktop -o program b.cmo a.cmo d.cmo c.cmo
```

ocamlbuild の動き

- 使い方

```
$ ocamlbuild name.byte
```

- *name*.ml というファイルを見つける
- それが依存するモジュール，それがまた依存するモジュール，. . . という具合に依存関係を調べる (ocamldep)
- 依存関係の順にコンパイルし，*name*.byte という，バイトコードを作る

```
$ ocamlbuild name.native
```

とすると，ネイティブコードを作る

- 対話的処理系は作れない

token 定義の不一致の解決法

- 方針 1: .mll に対して,「token は.mly にあるやつを使ってね」と指示する
  - .mll 中の token の定義を消す
  - そして,
    - ★ 具体的方法 1: 最初に，`open Calc_parse` と書いて，モジュール名なしで，`Calc_parse` 中の名前を参照できるようにする (前述)
    - ★ 具体的方法 1': token を参照する際， `Calc_parse.PLUS`, `Calc_parse.MINUS` のように，モジュール名をつけて参照する
- 方針 2: .mly に対して,「token は.mll にあるやつを使ってね」と指示する
  - 具体的方法 2: ocamlyacc の代わりに menhir を使う．menhir に「`--external-tokens` モジュール名」オプションを渡す

```
1 $ menhir --external-tokens Calc_lex calc_parse.mly
```

不一致の解決法: 方法1 (再掲)

- calc_lex.mll を以下のように変更:
- Calc_parse を open し，calc_parse.ml 内のトークン名を参照

```
1  {
2  open Calc_parse
3  }
4  rule lex = parse
5  | [' ' '\t' '\n']  { lex lexbuf }
6  | "+"              { PLUS }
7  | "-"              { MINUS }
8  | ['0'-'9']+ as s  { NUM(int_of_string s) }
9  | eof              { EOF }
```

不一致の解決法: 方法1'

- calc_lex.mll を以下のように変更
- トークン名が calc_parse.ml で定義されることを反映して，すべて，Calc_parse.PLUS などで参照

```
1  {
2  }
3  rule lex = parse
4  | [' ' '\t' '\n']  { lex lexbuf }
5  | "+"              { Calc_parse.PLUS }
6  | "-"              { Calc_parse.MINUS }
7  | ['0'-'9']+ as s  { Calc_parse.NUM(int_of_string s) }
8  | eof              { Calc_parse.EOF }
```

不一致の解決法: 方法2

- 手順:

```
1 $ menhir --external-tokens Calc_lex calc_parse.mly
```

- .mll と.mly で実質同じ token 定義を書かないといけないことには変わりないので，あまり推奨されない

合体の手順: 方法1, 1' の場合

```
1  $ ocamllex calc_lex.mll
2  $ menhir calc_parse.mly
3  # lex が parse に依存しているので, parse が先, lex が後
4  $ ocamlc -c calc_parse.mli calc_parse.ml calc_lex.ml
5  $ ocaml calc_parse.cmo calc_lex.cmo
6          OCaml version 4.01.0
7
8  # Calc_parse.program;;
9  - : (Lexing.lexbuf -> Calc_parse.token) -> Lexing.lexbuf -> int = <fun>
10 # Calc_lex.lex;;
11 - : Lexing.lexbuf -> Calc_parse.token = <fun>
```

正しい手順: 方法2の場合

```
1  $ ocamllex calc_lex.mll
2  6 states, 267 transitions, table size 1104 bytes
3    # calc_parse.ml にtoken の定義は calc_lex.ml にあるぞと指示
4  $ menhir --external-tokens Calc_lex calc_parse.mly
5    # lex が先, parse が後
6  $ ocamlc -c calc_lex.ml calc_parse.mli calc_parse.ml
7  $ ocaml calc_lex.cmo calc_parse.cmo
8          OCaml version 4.01.0
9
10 # Calc_parse.program;;
11 - : (Lexing.lexbuf -> Calc_lex.token) -> Lexing.lexbuf -> int = <fun>
12 # Calc_lex.lex;;
13 - : Lexing.lexbuf -> Calc_lex.token = <fun>
```

ocamlbuildで，方法1, 1'の場合

- calc_lex.mlyがcalc_parse.mllに依存してるので，calc_lex.byteを作れといえば，calc_parse.cmoも作られる

```
$ ocamlbuild -use-menhir calc_lex.byte
/usr/bin/ocamllex -q calc_lex.mll
/usr/bin/ocamldep -modules calc_lex.ml > calc_lex.ml.depends
menhir --raw-depend --ocamldep '/usr/bin/ocamldep -modules' calc_parse.mly
          > calc_parse.mly.depends
menhir --ocamlc /usr/bin/ocamlc --infer calc_parse.mly
/usr/bin/ocamldep -modules calc_parse.mli > calc_parse.mli.depends
/usr/bin/ocamlc -c -o calc_parse.cmi calc_parse.mli
/usr/bin/ocamlc -c -o calc_lex.cmo calc_lex.ml
/usr/bin/ocamldep -modules calc_parse.ml > calc_parse.ml.depends
/usr/bin/ocamlc -c -o calc_parse.cmo calc_parse.ml
/usr/bin/ocamlc calc_parse.cmo calc_lex.cmo -o calc_lex.byte
$ ocaml -I _build _build/*.cmo
        OCaml version 4.01.0

#
```

ocamlbuild で方法2の場合

- calc_parse.mly が calc_lex.mll に依存してるので，calc_parse.byte を作れといえば，calc_lex.cmo も作られる

```
1  $ ocamlbuild -use-menhir -yaccflags --external-tokens,Calc_lex
           calc_parse.byte
2  menhir --raw-depend --ocamldep '/usr/bin/ocamldep -modules' calc_parse.mly
            > calc_parse.mly.depends
3  menhir --ocamlc /usr/bin/ocamlc --external-tokens Calc_lex --infer
           calc_parse.mly
4  /usr/bin/ocamldep -modules calc_parse.mli > calc_parse.mli.depends
5  /usr/bin/ocamllex -q calc_lex.mll
6  /usr/bin/ocamldep -modules calc_lex.ml > calc_lex.ml.depends
7  /usr/bin/ocamlc -c -o calc_lex.cmo calc_lex.ml
8  /usr/bin/ocamlc -c -o calc_parse.cmi calc_parse.mli
9  /usr/bin/ocamldep -modules calc_parse.ml > calc_parse.ml.depends
10 /usr/bin/ocamlc -c -o calc_parse.cmo calc_parse.ml
11 /usr/bin/ocamlc calc_lex.cmo calc_parse.cmo -o calc_parse.byte
12 $ ocaml -I _build _build/*.cmo
13         OCaml version 4.01.0
14
15 #
```