ML 演習 第 4 回

新井淳也、中村宇佑、前田俊行

2011/05/10

言語処理系の作成

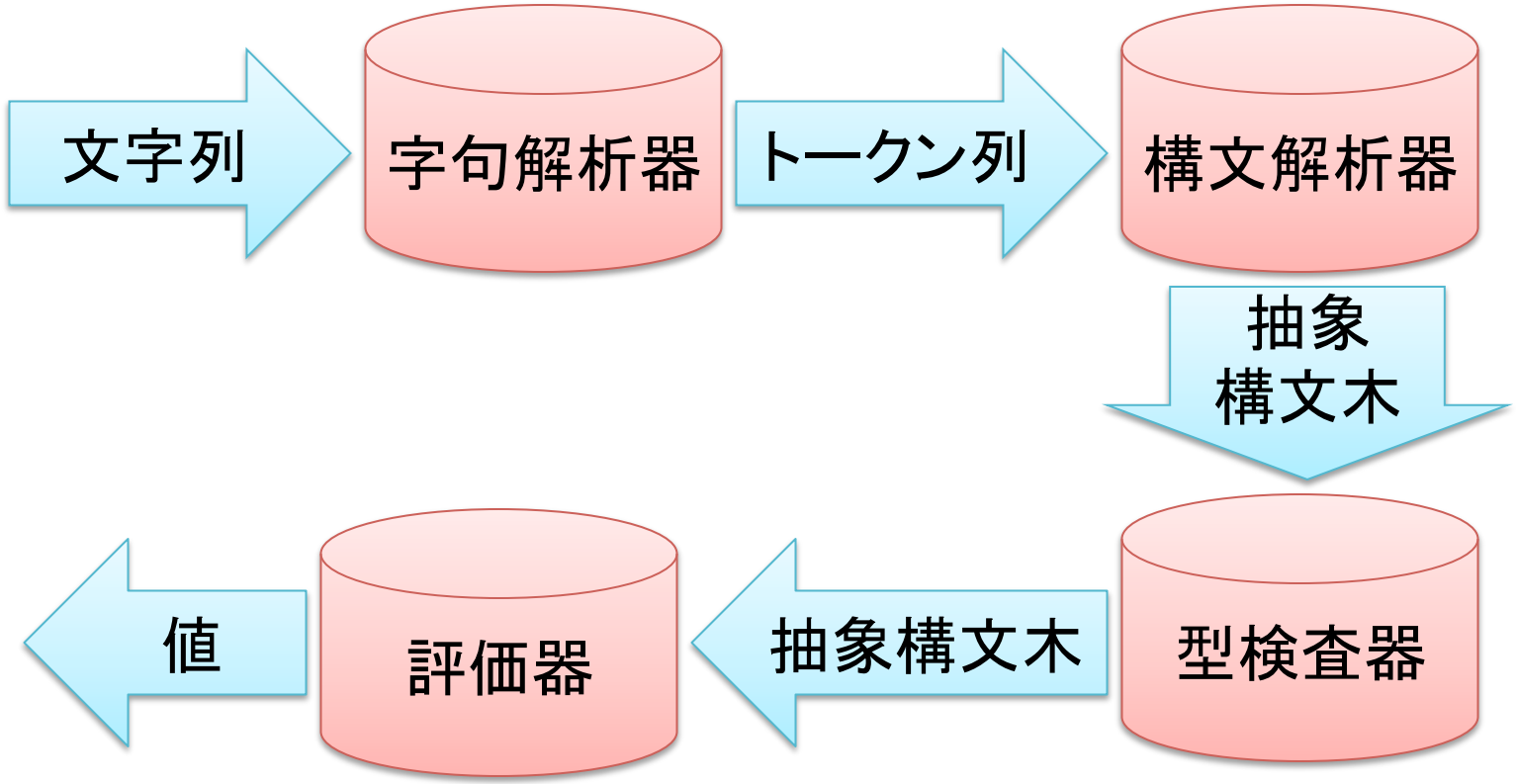
- 今後 4 回の予定
  - 第 4 回: 基本的なインタプリタの作成
    - 字句解析・構文解析・簡単な言語の処理系を作成する

    第 5 回: 関数型言語への拡張
    - 関数を作成し呼び出せるようにする
  - 第 6 回: 言語処理系と型システム
    - ML 風の型推論を実装する
  - 第 7 回: インタプリタの様々な拡張
    - 式の評価順序について考える

今回の内容

- ocamlyacc, ocamllex を用いて構文・字句解析を実装する
- 簡単な言語を解析・評価するインタプリタを作成する (課題)

インタプリタの構造



字句解析

- 文字列をトークン (単語) の列に変換する
  - 入力例: `let i=1-2 in i * i`

字句解析

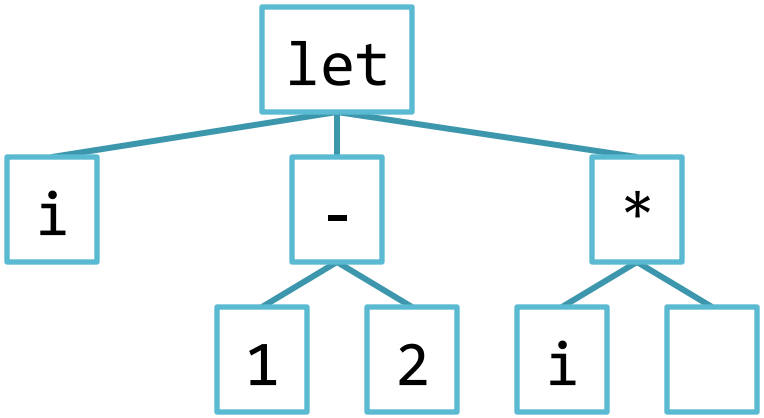
  - 出力例: `let` | `i` | `=` | `1` | `-` | `2` | `in` | `i` | `*` | `i`

- 字句解析器生成ツールの例
  - lex, flex, JLex, ocamllex

この一つ一つを
トークンと呼ぶ

構文解析

- トークン列を抽象構文木に変換する
  - 入力例: 

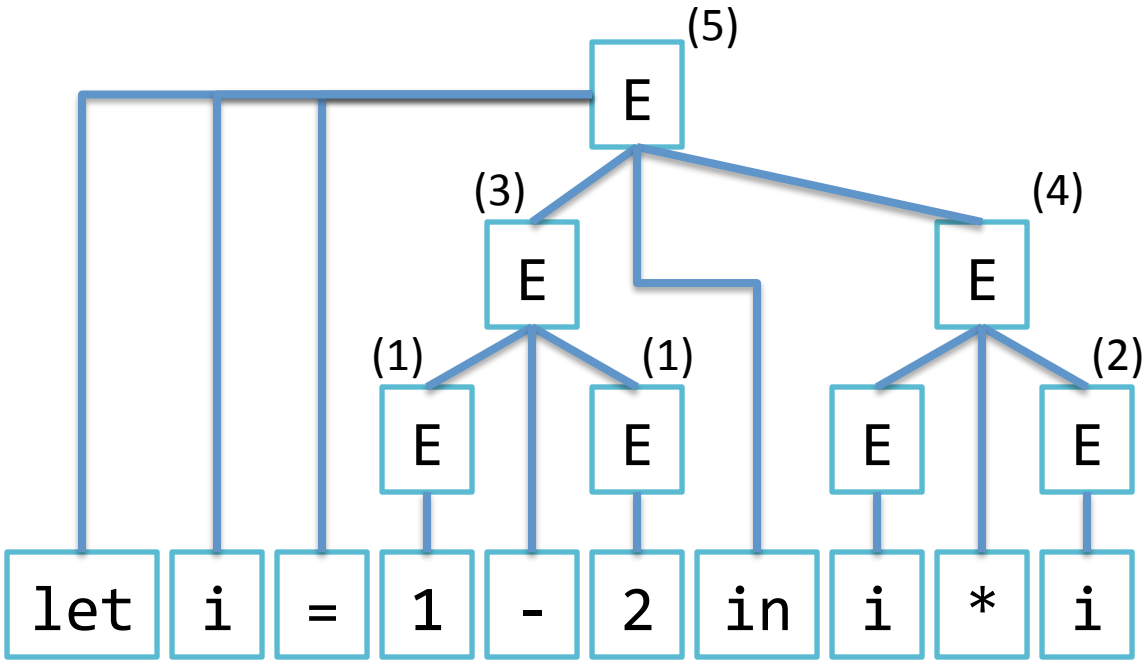
  - 出力例: 

- 構文解析器生成ツールの例
  - yacc, bison, Happy, ocamlyacc

構文定義と構文解析の関係

- 左のような構文定義から
  右の抽象構文木を導出する

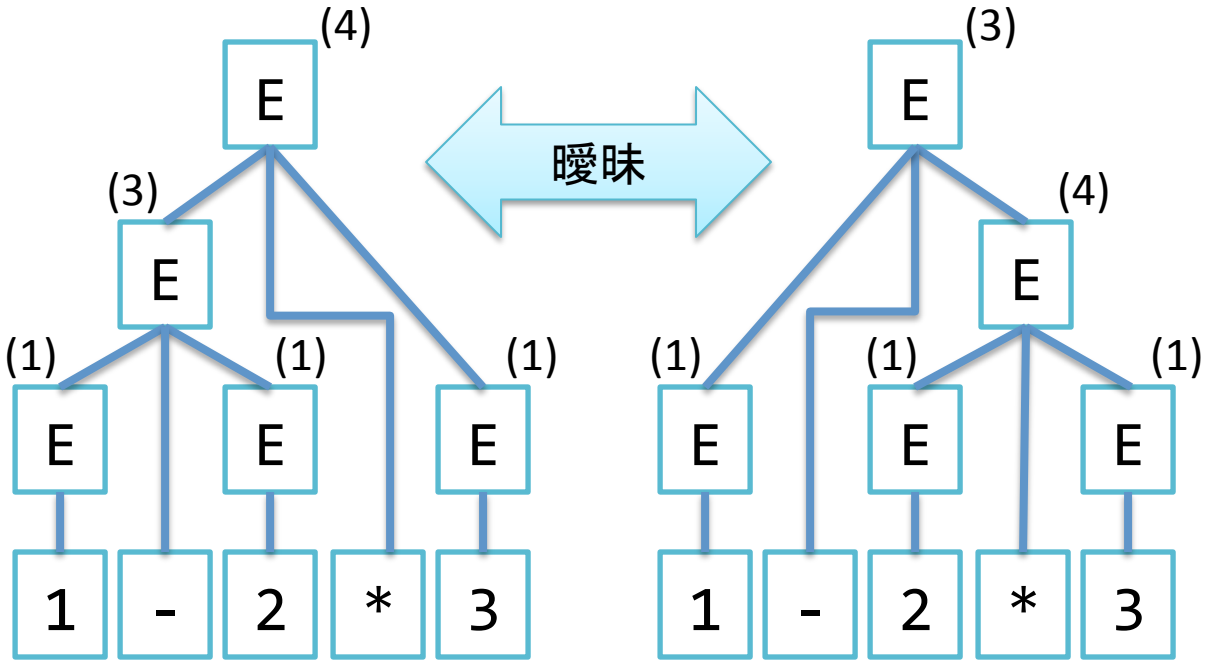
(1) $E \rightarrow V$ (値)
(2) $E \rightarrow I$ (文字列)
(3) $E \rightarrow E$ - E
(4) $E \rightarrow E * E$
(5) $E \rightarrow$ `let` I = E `in` E



構文定義の曖昧さについて: その1

- 左のような構文定義だと
  抽象構文木が一意に決まらない

…
(3) $E \rightarrow E - E$
(4) $E \rightarrow E * E$
…

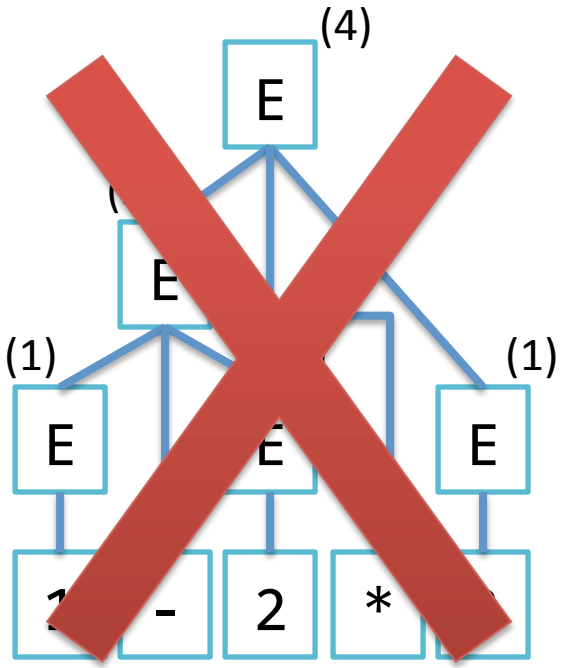


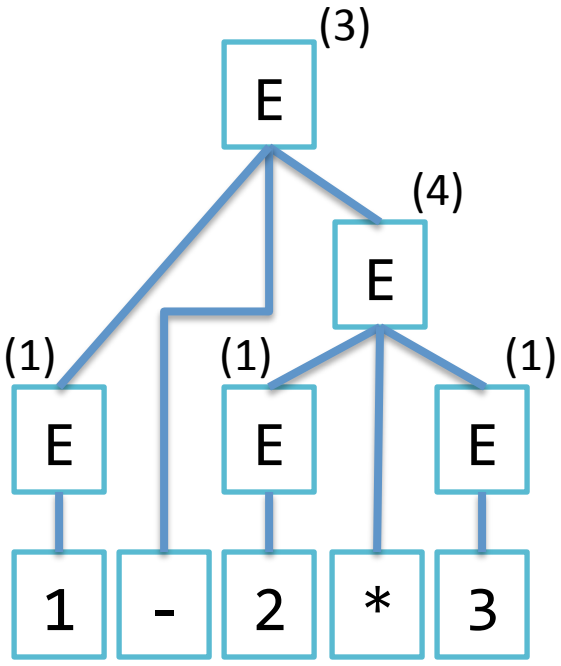
優先度を指定して曖昧さを解消する

- トークンや構文規則の間に優先度をつける

…
(3) $E \rightarrow E - E$
(4) $E \rightarrow E * E$
…

'*' を優先するように
指定すればよい

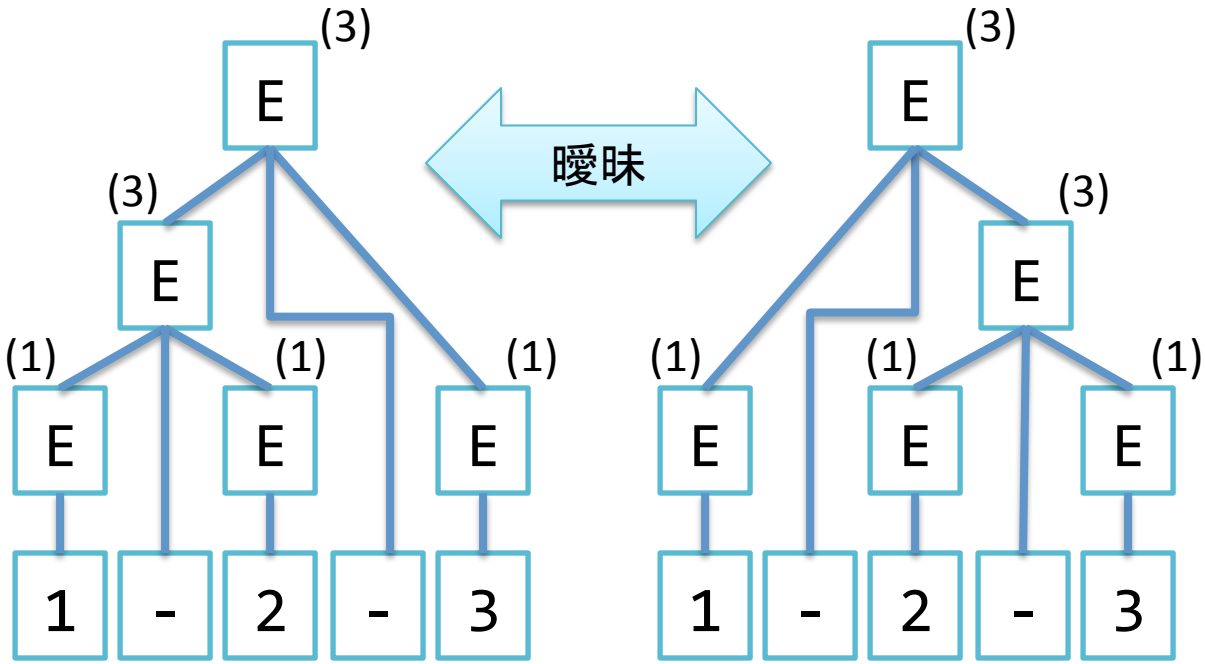




構文定義の曖昧さについて: その2

- 左のような構文定義だと
  抽象構文木が一意に決まらない

…
(3) $E \rightarrow E - E$
…

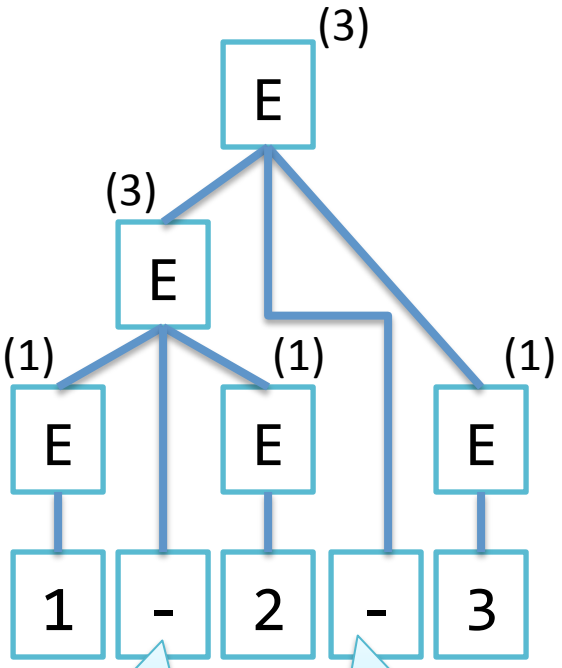


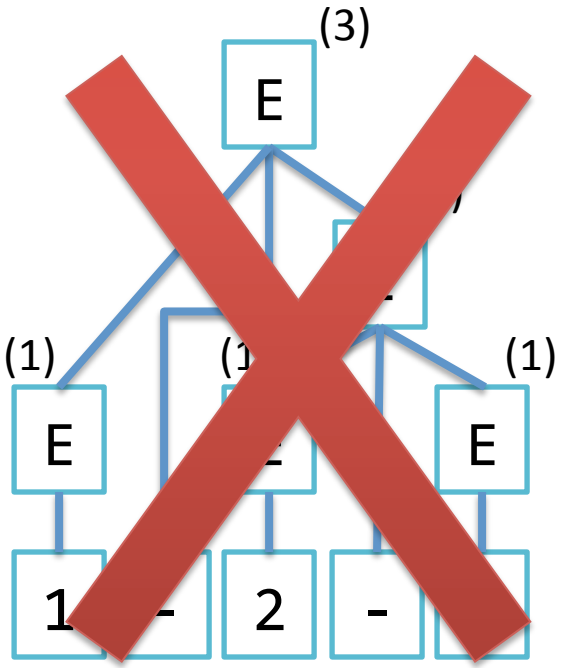
結合則を指定して曖昧さを解消する

- トークンや構文規則の結合則を指定する
  - 左結合・右結合・無結合

…
(3) $E \rightarrow E - E$
…

'-' を左結合と
指定すればよい





無結合は
構文エラーとする

左結合は
こちらを優先

右結合は
こちらを優先

Parsing with “ocamlyacc”

OCAMLYACC を用いた構文解析

ocamlyacc とは?

- OCaml 用の構文解析器生成ツール (LALR(1))
  - 入力: 構文定義ファイル (.mly)
  - 出力: 構文解析器モジュール (.mli, .ml)
- 出力される構文解析器は
  - トークン列を受け取って
  - 抽象構文木を返す
    - 構文定義ファイルの書き方によっては
      構文木以外のものを返すこともできる

構文定義ファイルの構造

%{
ヘッダー
%}
トークンや演算子の定義
%%
構文定義
%%
トレーラー

- 構文は文脈自由文法 (CFG) で定義
- ヘッダーとトレーラーに OCaml のコードを書いておくと出力される構文解析モジュールファイルの最初と最後にコピーされる
- ヘッダーとトレーラー以外ではコメントは /* … */ を使う

構文定義ファイルの例 (1): トークンの定義

- 構文

  %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」… /* 値をとるトークン */

- 例

exampleParser.mly

```
%token <int> INT                /* 整数リテラル */
%token PLUS MINUS TIMES DIV     /* 四則演算子 */
%token LPAREN RPAREN            /* 括弧 */
%token EOL                      /* 行の終わり */
```

- 上の定義からは以下のようなトークンの型が生成される

```
type token = INT of int
      | PLUS | MINUS | TIMES | DIV
      | LPAREN | RPAREN | EOL
```

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html より

構文定義ファイルの例 (2): 演算子の結合優先度と結合則の定義

- 構文

  (%left|%right|%nonassoc)「トークン名1」「トークン名2」…

- 例

```
%left PLUS MINUS    /* 加算・減算 */
%left TIMES DIV     /* 乗算・除算 */
%nonassoc UMINUS    /* 単項マイナス（符号反転） */
```

  - 下に書くほど優先度が高くなる
  - 行頭のキーワードで結合則を指定する
    - %left → 左結合
    - %right → 右結合
    - %nonassoc → 無結合

構文定義ファイルの例 (3): エントリーポイント(開始記号)の定義

- 定義の例

```
%start main
```

  - main という非終端記号を開始記号として定義
- 非終端記号の型の宣言の例

```
%type <int> main
```

  - 構文解析の結果として
    解析器が最終的に返す値の型を指定する
  - 普通は構文木を表すデータの型を指定する
    - が、この例では int

構文定義ファイルの例 (4): 構文定義

- 構文

```
「記号」:
      「記号11」「記号12」… { 「式1」 }
    | 「記号21」「記号22」… { 「式2」 }
    ;
```

- 例

```
main:
      expr EOL                    { $1 }
;
expr:
    | INT                         { $1 }
    | LPAREN expr RPAREN          { $2 }
    | expr PLUS expr              { $1 + $3 }
    | expr MINUS expr             { $1 - $3 }
    | expr TIMES expr             { $1 * $3 }
    | expr DIV expr               { $1 / $3 }
    | MINUS expr %prec UMINUS     { - $2 }
;
```

{ … } 内に解析結果として返す値を書く

$n は n 個目のトークンまたは非終端記号の解析結果の値を表す

構文は MINUS expr だが結合優先度・結合則は UMINUS に従うという意味

ocamlyacc の使い方

- .mly ファイルを ocamlyacc に渡すと構文解析器モジュールが .mli, .ml に生成される

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

ocamlyacc の出力する警告: shift/reduce conflict と reduce/reduce conflict について

- Shift/reduce conflict
  - 普通はそれ程深刻な問題ではない
    - 優先度や結合則の指定が適切かを確認すべき
- Reduce/reduce conflict
  - 構文定義に問題がある可能性が高い
    - 構文定義に致命的な曖昧さが無いかどうか確認すべき

- ocamlyacc に -v オプションを付けると LALR(1) の状態遷移表や遷移の競合に関する情報が .output ファイルに出力されるので参考にするとよい、かもしれない

Lexical Analysis with “ocamllex”

OCAMLLEX を用いた字句解析

ocamllex とは?

- OCaml 用の字句解析器生成ツール
  - 入力: 字句定義ファイル (.mll)
  - 出力: 字句解析器モジュール (.ml)
- 出力される字句解析器は
  - 文字列を受け取って
  - トークン列を返す
    - 正確には、文字列の入力バッファを受け取って、一つのトークンを読み出して返す

字句定義ファイルの構造

{
ヘッダー
}
正規表現の宣言
字句定義
{
トレーラー
}

- 字句は正規表現で定義
- ヘッダーとトレーラーに
  OCaml のコードを書いておくと
  出力される字句解析
  モジュールファイルの
  最初と最後にコピーされる
- コメントは (* … *)

正規表現の宣言の書き方

- 構文

  let「名前」=「正規表現」

- 字句定義で使う正規表現を
  変数として定義しておける

字句定義の書き方

- 構文:

```
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」}
| …
```

– 文字列の入力バッファ (Lexing.lexbuf 型)を受け取ってトークンを返す字句解析関数が指定したエントリポイント名で定義される
– { … } の中に、返すトークンを書く

正規表現の例

```
'T'       (* 文字 'T' にマッチ *)
_         (* アンダースコア: どんな文字にもマッチ *)
eof       (* 入力の最後にマッチ *)
"Hello"   (* 文字列 "Hello" にマッチ *)
['_' 'a'-'z']
          (* 文字 '_'、文字 'a' ～ 'z' にマッチ)
['0'-'9']+
          (* 文字 '0' ～ '9' の1回以上の繰返しにマッチ *)
[' ' '\t']*
          (* 文字 ' ' か '\t' の0回以上の繰返しにマッチ *)
```

詳細は http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html を参照

字句定義ファイルの例

exampleLexer.mll

```
{
open ExampleParser
}
let digit = ['0'-'9']
rule token = parse
   | [' ' '\t']+    { token lexbuf }
   | '\n'           { EOL }
   | digit+ as lxm  { INT(int_of_string lxm) }
   | '+'            { PLUS }
   | '-'            { MINUS }
   | '*'            { TIMES }
   | '/'            { DIV }
   | '('            { LPAREN }
   | ')'            { RPAREN }
   | eof            { raise End_of_file }
```

トークンの型 token は
ExampleParser モジュールにある

http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html より一部改変

字句定義の注意 (1)

- 正規表現は
  できるだけ長い文字数がマッチするもの
  が選ばれる
  - そのような正規表現が複数ある時は
    字句定義の中で先に出てくる方が選ばれる

字句定義の注意 (2)

- 正規表現の中で as キーワードを使うと
  マッチした文字列を変数に束縛できる
  - 直後の { … } 中でマッチした文字列を参照できる

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

  - マッチした「一つ以上の数字」を変数 lxm に束縛

字句定義の注意 (3)

- 字句解析関数自身を再帰呼び出しすることで
  その時マッチしている正規表現を飛ばして
  次のトークンを返すことができる

```
rule token = parse
    [' ' '\t']+   { token lexbuf }
```

  - 字句解析関数 token を再帰呼び出し
    - 入力バッファは lexbuf という変数で参照できる

ocamllex の使い方

- .mll ファイルを ocamllex に渡すと
  字句解析器モジュールが .ml に生成される

```
$ ocamllex exampleLexer.mll
11 states, 267 transitions, table size 1134 bytes
$ ls exampleLexer.*
exampleLexer.ml  exampleLexer.mll
```

How to Use Modules Generated by “ocamllex” and “ocamlyacc”

生成されたモジュールの使い方

実際に構文解析をするには?

- 構文解析器モジュールに定義された
  構文解析関数を呼び出せばよい
  - この関数の名前は
    構文定義ファイルで指定した開始記号の名前
    となっている
  - この関数は字句解析関数と
    入力バッファを受け取って全体の解析結果を返す

構文解析器・字句解析器の利用例

```
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
    let rec loop () =
      let result =
          ExampleParser.main ExampleLexer.token lexbuf in
      print_int result; print_newline (); flush stdout;
      loop ()
    in
    loop ()
  with End_of_file ->
    exit 0
```

標準入力を読み込む入力バッファを生成

構文解析関数に
字句解析関数と
入力バッファを渡す

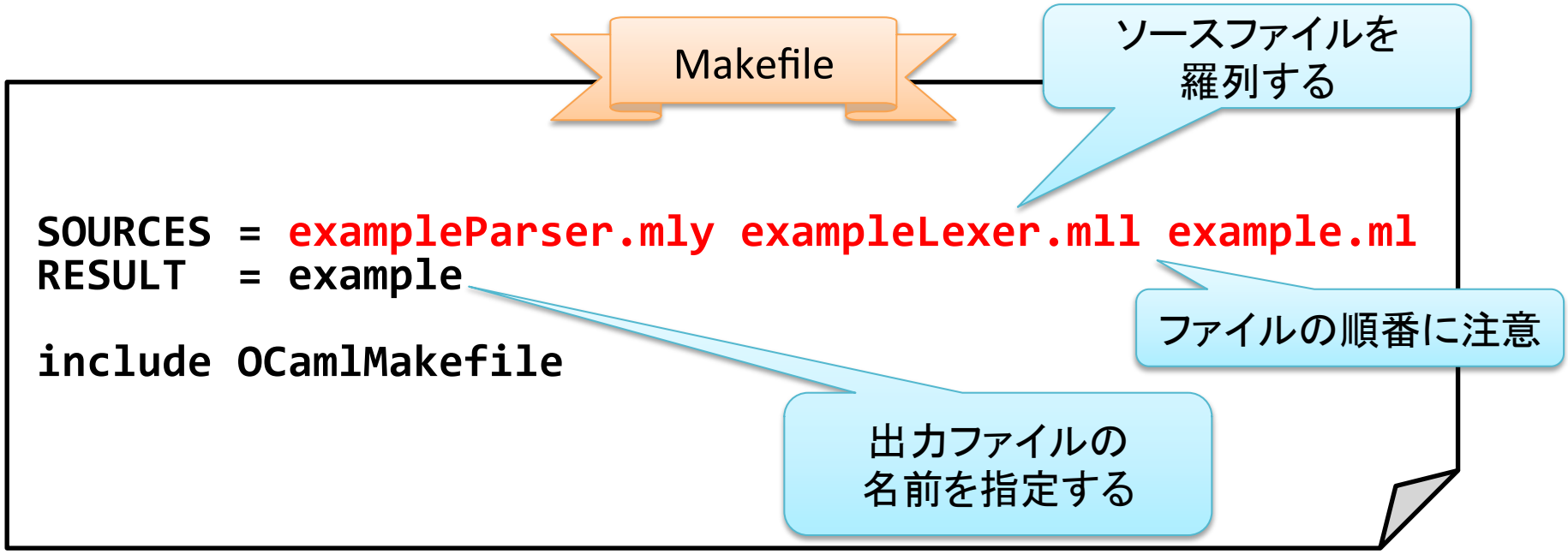
http://caml.inria.fr/pub/docs/manual-ocaml/manual026.html より一部改変

構文解析・字句解析器の利用例

```
$ ocamlyacc exampleParser.mly
$ ocamllex exampleLexer.mll
11 states, 267 transitions, table size 1134 bytes
$ ocamlc -c exampleParser.mli
$ ocamlc -c exampleParser.ml
$ ocamlc -c exampleLexer.ml
$ ocamlc -c example.ml
$ ocamlc -o example exampleParser.cmo \
                    exampleLexer.cmo example.cmo
$ ./example
1 + 2
3
```

生成された
モジュールを
コンパイル

OCamlMakefile を使う、再び



– 詳しい使い方は前回 (第3回の資料) を参照

ビルド (make) の実行例

パッケージで導入した場合はこの場所にある

このコピーは一回だけで十分
(ビルド毎にコピーする必要はない)

```
$ cp /usr/share/ocamlmakefile/OCamlMakefile ./
$ ls
Makefile        example.ml        exampleParser.mly
OCamlMakefile   exampleLexer.mll
$ make
( … 省略 … )
$ ls
Makefile        example.ml        exampleParser.cmi
OCamlMakefile   exampleLexer.cmi  exampleParser.cmo
example         exampleLexer.cmo  exampleParser.ml
example.cmi     exampleLexer.ml   exampleParser.mli
example.cmo     exampleLexer.mll  exampleParser.mly
```

第 4 回 課題

締切: 5/24 13:00 (日本標準時)

課題 1 (25点)

- bool 値と int 値に対する簡単な計算を表す式 E の文法を以下のように定義する

$V \rightarrow$ (bool 値)	$V \rightarrow$ (int 値)
$E \rightarrow V$	$E \rightarrow (E)$
$E \rightarrow E + E$	$E \rightarrow E - E$
$E \rightarrow E * E$	$E \rightarrow E / E$
$E \rightarrow E \,\&\&\, E$	$E \rightarrow E \,\|\|\, E$
$E \rightarrow E = E$	$E \rightarrow$ `if` E `then` E `else` E

次ページに続く

課題 1 (続き)

- 前ページの式を評価するインタプリタを実装することを考える
  - インタプリタの入力は以下のように定義する

    $C \rightarrow E$;;
- また上記の入力に対応する型cmdを以下のように定義する

```
type cmd = Cmd of expr
```

次ページに続く

課題 1 (続き)

- インタプリタを途中まで実装したものが
  以下の四つのファイルである
  - syntax.ml: 抽象構文木を表す型の定義
  - parser.mly: 構文定義
  - lexer.mll: 字句定義
  - interp.ml: 解析器を呼び出して評価するプログラム
- これらのファイルをもとにして
  インタプリタを完成させよ
  - 式の評価自体は第2回の課題6・課題7が使える (はず)
    - 型検査は省略してよい
  - 演算子の結合優先度や結合則は
    各自で考えて適切に設定すること

課題 2 (10点)

- 課題 1 のインタプリタを拡張して「コメント」を使えるようにせよ
  - ヒント: 字句解析 (lexer.mll) を修正するとよい
  - コメントのスタイルは各自で考えて適切に設定すること
    - 既存のプログラミング言語のコメントのスタイルを参考にしてよい
      - OCaml, C 等

課題 3 (25点)

- 課題 1 のインタプリタを拡張して
  変数を扱えるようにすることを考える
  - 例えば文法に以下を追加する

    $I \rightarrow$ (文字列)　　$E \rightarrow I$

    $E \rightarrow$ let I = E in E　　$C \rightarrow$ let I = E ;;
  - まず文法の変更に応じて型 **expr** 等を修正せよ
  - 次いで parser.mly と lexer.mll を修正せよ
  - 最後に関数 **eval** や interp.ml を修正せよ

次ページに続く

課題 3 (続き)

- 関数 **eval** や interp.ml の実装のヒント
  - 例えば、関数 eval が「環境」も引数にとるようにすればよい

```
eval : env -> expr -> value
```

  - 「環境」=「変数から値への写像」とすればよい
    - 仮に環境をモジュール **Env** として定義するとしたらシグネチャは以下を含むはずである

```
type t                  (* 環境を表すデータの型 *)
val empty : t           (* 空の環境 *)
val add :「変数を表す型」-> value -> t -> t
                        (* 変数と値の対応を追加する関数 *)
val get :「変数を表す型」-> t -> value
                        (* 変数に対応する値を得る関数 *)
```

    - 副作用を使わずに実装すると後で楽
- 必ずしも上記の通り実装する必要はない

次ページに続く

課題 3 (続き)

- 環境をモジュールとして実装した
  としたときの実行例
  - ここでは変数を表す型を文字列とした

```
# let oldenv = Env.add "i" (Int_value(0)) Env.empty;;
val oldenv : Syntax.value Env.t = <abstr>
# let newenv = Env.add "i" (Int_value(1)) oldenv;;
val newenv : Syntax.value Env.t = <abstr>
# Env.get "i" oldenv;;
- : Syntax.value = Int_value 0
# Env.get "i" newenv;;
- : Syntax.value = Int_value 1
```

次ページに続く

課題 3 (続き)

- 変数束縛式「let 変数 = 式1 in 式2」の評価のヒント
  1. まず式 1 を与えられた環境で評価する
  2. 変数と式 1 の評価結果の値の対応を環境に追加する
  3. 新しい環境で式 2 を評価し、その結果の値を返す
- 変数束縛入力「let 変数 = 式 ;;」の評価のヒント
  1. まず式を与えられた環境で評価する
  2. 変数と式の評価結果の値の対応を環境に追加する
  3. 新しい環境で次の入力を処理する

課題 4 (15 点)

- 次ページ以降のように構成される Parsing Expression Grammar (PEG) のための構文解析器生成器を作成せよ

  - 参考:
    - http://pdos.csail.mit.edu/~baford/packrat/
    - http://en.wikipedia.org/wiki/Parsing_expression_grammar

課題 4 (続き): PEG の定義

- PEG は以下の 4 つ組からなる
  - (N, Σ, P, S)
    - N: 非終端記号の集合
    - Σ : 終端記号の集合
      - ここでは任意の文字 (char) の集合とする
    - S : 開始記号
    - P : 構文規則の集合
- P の各構文規則は以下の形をとる
  - A ← e
    - A は非終端記号
    - e は構文解析式 (定義は次ページ)

課題 4 (続き): PEG の構文解析式の定義 (1/2)

- 構文解析式 e は以下の要素から構成される
  - 任意の終端記号
    - 入力文字列の先頭の文字とマッチするかチェックする
  - 任意の非終端記号
    - 対応する構文規則の定義を再帰的に適用する
  - 空文字列： ε
    - 任意の入力文字列とマッチする
  - 連結 : e_1 e_2
    - まず e1 が入力文字列とマッチするかチェックし、マッチすれば残りの入力文字列と e2 がマッチするかをチェックする
  - 順序付き選択 : e_1 / e_2
    - まず e_1 が入力文字列とマッチするかチェックし、マッチしなければ元の入力文字列と e_2 がマッチするかをチェックする

(次ページへ続く)

課題 4 (続き): PEG の構文解析式の定義 (2/2)

(前ページからの続き)

- 0 回以上繰返し : e*
- 1 回以上繰返し : e+
- 0 回か 1 回 : e?
  - それぞれ、入力文字列が、e の 0 回以上の繰返し、1 回以上の繰返し、0 回か 1 回の出現、とマッチするかをチェックする
    - 繰返しはマッチし続ける限り入力文字列を消費し続け、バックトラックはしない
- AND 条件 : &e
  - 入力文字列が e とマッチすることをチェックする
    - ただし、入力文字列を消費しない
- NOT 条件 : ! e
  - 入力文字列が e とマッチしないことをチェックする
    - ただし、入力文字列を消費しない

課題 4 (続き): PEG の構文規則の例:その 1

```
S ← E
E ← T ('+' T)*
T ← F ('*' F)*
F ← N / ( '(' E ')' )
N ← ('0' / '1' / … / '9')+
```

- ただし
  - S, E, T, F, N は非終端記号
  - '+', '*', '(', ')', '0', '1', …, '9' は終端記号
  - S は開始記号

課題 4 (続き): PEG の構文規則の例:その 2

```
S ← &(A c) a+ B !(a/b/c)
A ← a A? b
B ← b B? c
```

- ただし
  - S, A, B は非終端記号
  - a, b, c は終端記号 (シングルクォートはここでは省略)
  - S は開始記号

課題 4 (続き): PEG の構文規則の例:その 3

```
S ← 'if' C 'then' S 'else' S
  / 'if' C 'then' S
C ← …
…
```

- ただし
  - S, C, … は非終端記号
  - 'if', 'then', 'else' は終端記号
  - S は開始記号

課題 4 (続き): PEG の構文規則の例:その 4

S ← 'a'* 'a'

この式 'a' のマッチは常に失敗する
(繰返し 'a'* が 'a' を消費しつくすので)

- ただし
  - S は非終端記号 (開始記号)
  - 'a' は終端記号

課題 4 (続き): 作成方法の一例 (1/3)

- まず、各構文解析式が以下のような型の値に対応すると考える

```
type 'a parser =
  string * int -> ('a * (string * int)) option
```

  - すなわち、入力文字列を引数として受け取って、構文解析結果と残りの入力文字列を返す関数
  - ただし、
    - 入力文字列は全体の文字列 (string) とその文字列中の開始位置 (int) の組で表すことにしている
    - 構文エラーは None を返すことで表すことにしている

課題 4 (続き): 作成方法の一例 (2/3)

- 基本的には各構文解析式に対応する
  以下のような型を持つ値を定義すればよい

```
val terminal : char -> char parser
val empty : unit parser
val seq : 'a parser -> 'b parser -> ('a * 'b) parser
val ordered_choice : 'a parser -> 'a parser -> 'a parser
val many : 'a parser -> 'a list parser
val many1 : 'a parser -> 'a list parser
val opt : 'a parser -> 'a option parser
val and_p : 'a parser -> unit parser
val not_p : 'a parser -> unit parser
```

  - 非終端記号は関数に束縛される変数名に対応させればよい

課題 4 (続き): 作成方法の一例 (3/3)

- 例えば 10 進自然数の構文解析器は
  以下のように作成できる

```
let zero = fun _ -> None
let code_0 = int_of_char '0'
let digit_lst =
  let rec mk_lst n = if n < 0 then []
                     else n :: mk_lst (n - 1) in
  List.rev_map
    (fun i -> terminal (char_of_int (i + code_0))) (mk_lst 9)
let digit = List.fold_right ordered_choice digit_lst zero
let nat : int parser = fun input ->
        let r = (many1 digit) input in
        match r with
        | None -> None
        | Some (lst, input') ->
            let r = List.fold_left
              (fun r c -> r * 10 + (int_of_char c) - code_0) 0 lst in
            Some (r, input')
```

課題 5 (15 点)

- 課題 4 の構文解析器生成器を改造して
  構文解析器が入力文字列の長さ n に対して
  O(n) の時間計算量で解析できるようにせよ

課題 6 (15 点)

- 以下のような signature を持つ functor EXIST を定義せよ

```
module EXIST :
functor (T : sig type 'a t end) ->
sig
    type t
    type 'b u = { f : 'a. 'a T.t -> 'b }
    val pack : 'a T.t -> t
    val unpack : t -> 'b u -> 'b
end
```

課題 7 (15点)

- 以下の signature を持つような、
  異なる型の値を要素として持つリストを
  課題 6 の結果を用いて実現せよ

```
module ExList :
functor (T : sig type 'a t end) ->
sig
    type t
    val nil : t
    val cons : 'a T.t -> t -> t

    type 'a nil_elim = unit -> 'a
    type 'a cons_elim = { c : 'b . 'b T.t -> t -> 'a }
    val match_f : t -> 'a nil_elim -> 'a cons_elim -> 'a

    type 'a fold_elim = { f : 'b . 'b T.t -> 'a -> 'a }
    val fold_right : 'a fold_elim -> t -> 'a -> 'a
end
```