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

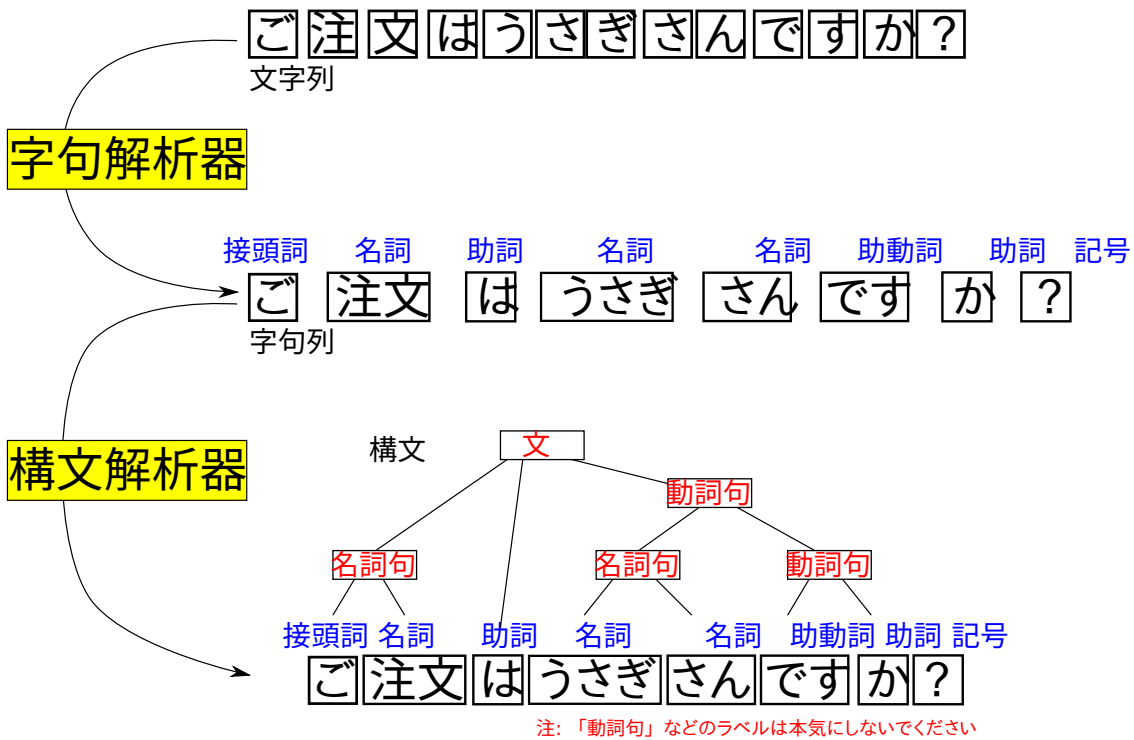
- あらゆるプログラミング言語処理系は，最初にプログラムを読み込み，文法のチェックを行う
  - 字句解析器 (*“lexer”* または *“tokenizer”*)
  - 構文解析器 (*“parser”*)
- それらは,「言語処理系」でなくてもあらゆる場面で必要
  - Web Page (HTML や XML) の読み込み
  - CSV, SVG, . . . ファイル. . .
  - ソフトの config file. . .
- それらを「さっと作れる」ことは実践的にも重要なスキル
  - アドホックに文字列処理をやるだけではきっとうまく行かない
  - そのための便利なツ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

Introduction

- All programming language implementations first read a program and check its grammar
  - lexical analyzer (*"lexer"* or *"tokenizer"*)
  - syntax checker (*"parser"*)
- they are necessary not only in programming language implementations but in many other circumstances
  - web pages (HTML or XML)
  - CSV, SVG, . . . files . . .
  - config files of software . . .
- it's an important skill to be able to make them quickly
  - you'd better not process strings in an ad-hoc manner
  - there are useful tools for them (parser generators)
  - it never hurts to have an experience with them

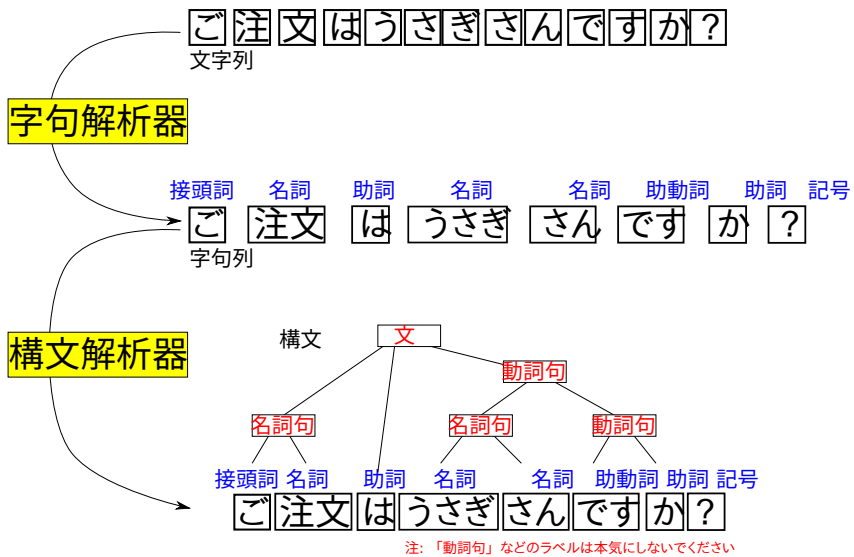
字句解析と構文解析

- 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」 (式, 文, プログラム全体など)
  - 文にならない字句の列が来たらエラー

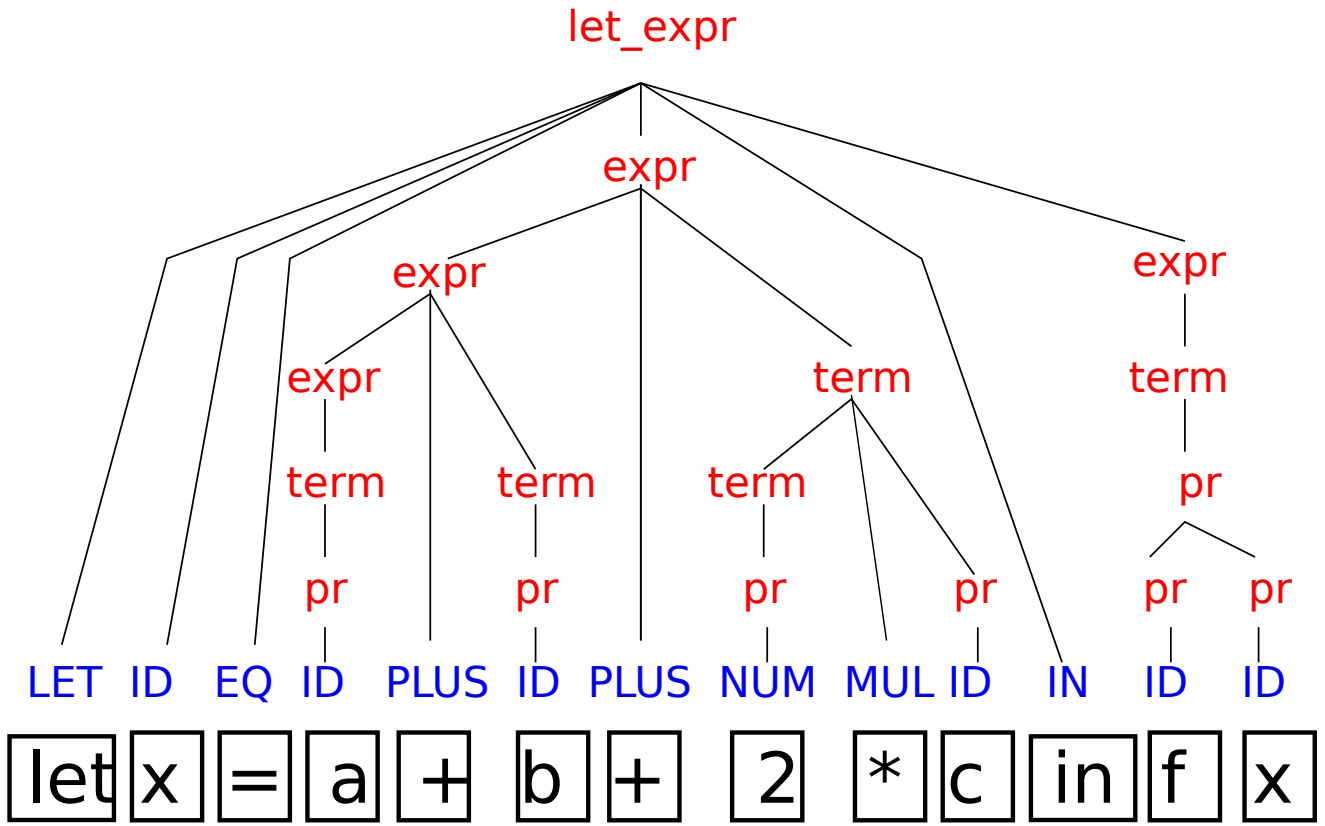


Lexer and parser

- lexer ≈
  - converts a sequence of "characters" → a sequence of "tokens" (≈ words)
  - rejects when characters do not constitute a valid token
- parser ≈
  - converts a sequence of "tokens" → a "sentence" (expression, statement, whole program, etc.)
  - rejects tokens that constitute a valid sentence

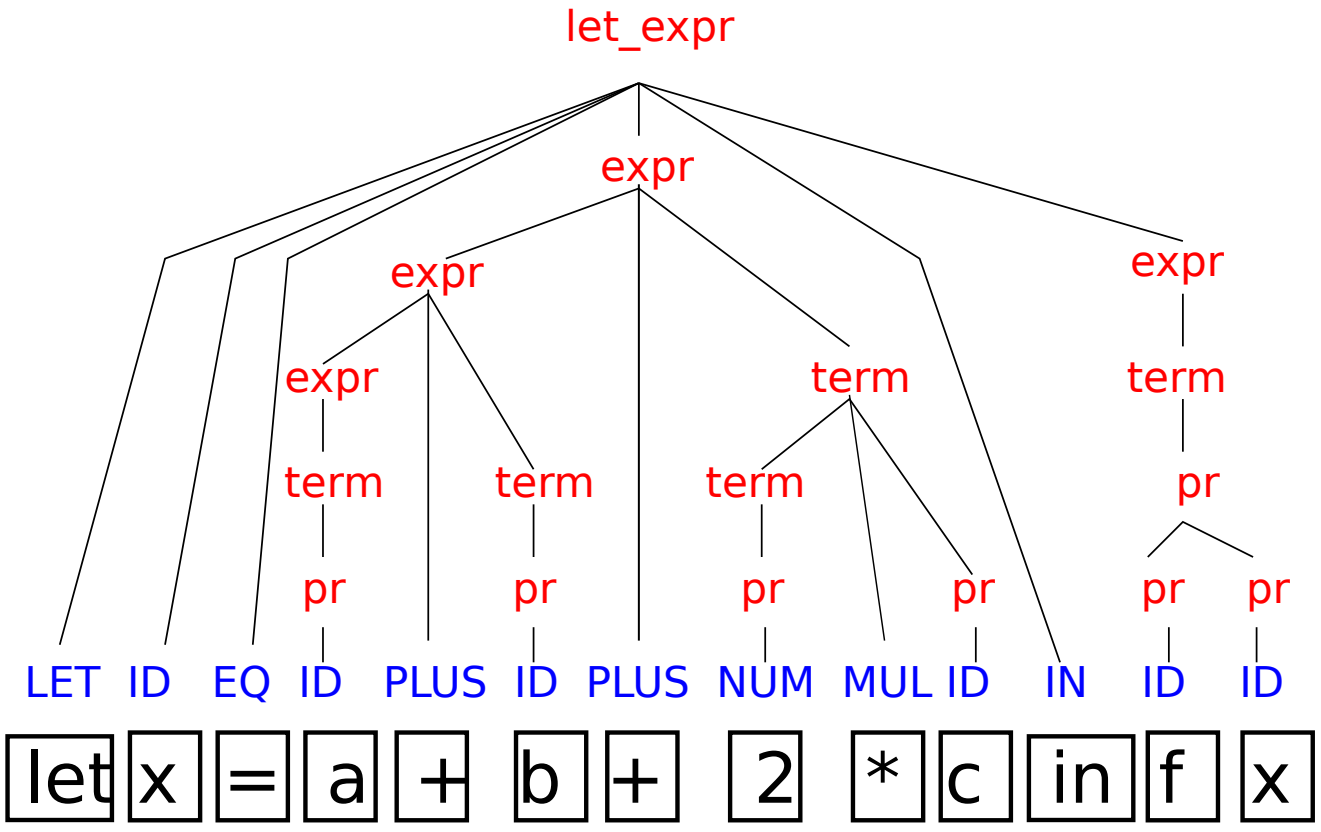


プログラミング言語の例



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

An example in a programming language



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

How to define a token and a sentence?

- normally, we define
  - tokens: by *regular expression (regex)*
  - sentences: by *context free grammar (CFG)*
- there are tools that generate programs (lexers and parsers) from their declarative descriptions
- "practice makes perfect." Let's see it working

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 }
```

`ocamllex` : lexer generator

- input to `ocamllex` = .mll file (similar to but different from OCaml)
- ex. (`calc_lex.mll`)

```
1  { (* preamble: any OCaml code; can be omitted *)
2  type token =
3    NUM of (int)
4  | PLUS
5  | EOF
6  }
7
8  (* the real part *)
9  rule lex = parse
10 | [' ' '\t' '\n'] { lex lexbuf }          (* skip whitespaces *)
11 | "+"              { PLUS }
12 | ['0'-'9']+ as s { NUM(int_of_string s) }
13 | eof              { EOF }
14
15 { (* any OCaml code; can be omitted *)
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) }
```

.mll file format

```
{
  any OCaml code
  typically type definition for tokens
}

(* define regexes *)
let id = regex
  ...

(* rule *)
rule lex = parse
| regex { expr }
| regex { expr }
   ...
| regex { expr }

{
  any OCaml code
}
```

- semantics of "| *regex* { *expr* }" :

  when a prefix of the input matches "regex", *evaluate* "expr" *and make it a token*

- "*regex* as *var*" binds the matched string to the variable *var*

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

- you can name regular expressions for later use

```
1 let digit = ['0'-'9']
2 ...
3 | 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

ocamllex regex examples

regex	semantics (strings that match it)
`_`	any character
`'a'`	`a`
`[ 'a' 'b' 'c' ]`	`a`, `b`, or `c`
`['0'-'9']`	any of `0`, `1`, `...`, `9`
`"abc"`	`abc`
`"abc"|"def"`	`abc` or `def`
`"abc"*`	zero or more repetitions of `abc`
`"abc"+`	one or more repetitions of `abc`
`("abc"|"def")+`	one or more (`abc` or `def`)
`eof`	end of input

- see `http://caml.inria.fr/pub/docs/manual-ocaml-400/manual026.html` (English) or `http://ocaml.jp/archive/ocaml-manual-3.06-ja/manual026.html` (Japanese) for more

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)

Note: a formal definition of regular expressions

- let A be the set of characters constituing a sequence (*alphabet*)
- the following is *regular expressions over A*

expression		semantics (strings that match it)
ϵ		empty string
a	$(a \in A)$	a
RS	(R, S : regex)	concatination of strings matching R and strings matching S
$R \mid S$	(R, S : regex)	strings matching R or S
$R*$	(R : regex)	zero or more repetitions of a string matching R

- use parens as necessary (例: `(abc|def)+`)
- all previous examples are combinations of the above (or an abbreviation thereof)

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

```
$ ocamllex calc_lex.mll
6 states, 267 transitions, table size 1104 bytes
$ ls
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)

```
$ ocaml -init calc_lex.ml
        OCaml version 4.01.0

# lex ;;
- : 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

A file and function generated by ocamllex

- ocamllex generates an OCaml file (.ml) from a lexer definition file (.mll)

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

- the `.ml` file defines a function `lex` (as `rule lex = parse ...` was in the `.mll`)

```
$ ocaml -init calc_lex.ml
        OCaml version 4.01.0

# lex ;;
- : Lexing.lexbuf -> token = <fun>
```

- `Lexing.lexbuf` is a type of buffers to read characters from (≈ `FILE*` of C); a mutable record
- `lex` *buf* consumes characters from *buf* and returns a token

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")
```

Creating a `Lexing.lexbuf` buffer

- from a string

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

- from a standard input

```
1 Lexing.from_channel stdin
```

- from a file

```
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 }
```

Using a lexer

```
$ 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 =
  { ... (snip) ... }
# 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 : parser generator

```
/* declarations +
   any OCaml code */
%{  (* any OCaml code *)
%}
/* token definitions */
%token <int> NUM
%token PLUS EOF

/* the start symbol and
   its type (mandatory) */
%start program
%type <int> program
%%  /* grammar definitions
       and evaluation rules */
expr :
| NUM            { $1 }
| expr PLUS NUM { $1 + $3 }

program :
| expr EOF       { $1 }
%%
  (* any OCaml code *)
```

- input = `.mly` file
- format: separated by %% into 3 parts
  - declarations + any OCaml code
  - grammar defs and evaluation rules
  - any OCaml code
- declarations
  - `%token` : token names and their types
  - `%start` : the start symbol (the symbol representing the whole input)
  - `%type` : a type corresponding to a symbol (mandatory for the start symbol)
- Remark: both `.mll` and `.mly` define tokens (a better method comes later)

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:

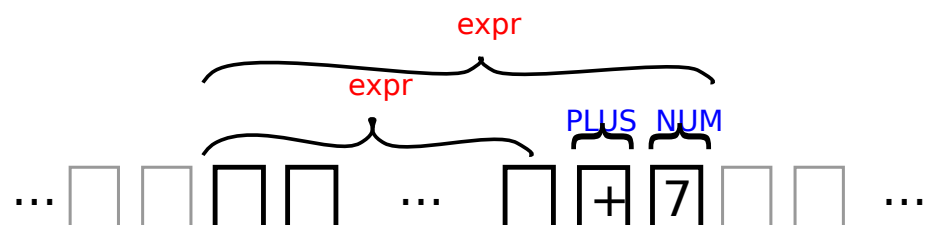
```
1 expr :
2 | ...
3 | expr PLUS NUM  { ... }
```

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)



Grammar definition in ocamlyacc

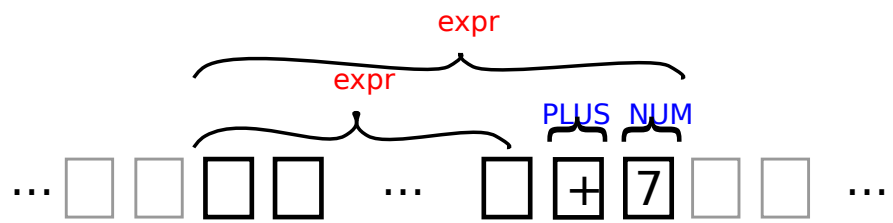
- based on *Context Free Grammar (CFG)*
- ex.

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

reads that concatinating
  - (a string matching) `expr`,
  - (a token) `PLUS`, and
  - (a string matchin) `NUM`

will match (make a string matching) `expr`

- `|` indicates there are multiple possibilities
- { ... } is an evaluation rule (later)
- Remarks:
  - the righthand side can reference the symbol being defined itself (recursive definitions)
  - multiple symbols can refer to each other (mutually recursive definitions)



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

A formal definition of CFG

- set of terminal symbols (tokens): T
- set of non-terminal symbols: NT
- the start symbol: $S \in NT$
- set of rules. each rule is of a form

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

  $(n \geq 0,\ a \in NT,\ b_i \in NT \cup T)$.
  - which reads that concatinating
    - a token sequence matching b_1,
    - ...
    - a token sequence matching b_n,

    will match (make a string matching) a
- just for the sake of formality:
  - if b_i is a token, b_i matches (a token sequence consisting only of) the token

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

Evaluation rule

- any OCaml expression, which can use `$1, $2, ...` to reference values of the corresponding symbol on the righthand side
- semantics: if a subsequence of tokens matches a due to the rule $a = b_1 \ \ldots \ b_n$, evaluate the corresponding evaluation rule and associate the resulting value with the subsequence
- ex.

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

reads that, if a subsequence of tokens matches `expr PLUS NUM`, the value of that subsequence is the sum of

  - the value of (the string matching) the first term `expr` and
  - the value of (the string matching) the third term `NUM`

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

A file generated by ocamlyacc

- ocamlyacc generates two OCaml files (`.ml` and `.mli`) from a `.mly` file
  - what is `.mli`? ⇒ later

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

- a function is defined in `.ml` with the name of the start symbol
  - that is, due to `%start program` in the `.mly` file, a function `program` will be defined

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

a parser (function) generated by ocamlyacc

```
$ ocaml -init calc_parse.ml
        OCaml version 4.01.0

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

- `Lexing.lexbuf -> token` is the type of lexers
- `int` is what `.mly` file (`%type <int> parse`) specified
- the parser
  - takes a lexer and a character buffer,
  - repeats calling the lexer to get tokens one after another from the character buffer,
  - checks if the entire token sequence matches the start symbol, and
  - if it does, returns the value associated with the entire sequence according to the evaluation rule

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

```
# 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

Combining the lexer and the parser

- a lexer (`lex`) and a parser (`program`) are now ready
- we would expect we can combine them as follows

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

- this is not the case, unfortunately, because
  1. they are written in seprate files, which must follow a rule to refererence each other
  2. a more complicated issue: `token` generated from `.mll` and `token` generated from `.mly` are not considered the same thing

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` in lexer ≠ `token` in parser

- - .mll

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

  this is `token` in `calc_lex.ml`

  - .mly

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

  this is `token` in `calc_parse.ml`

- they are different even if their names are the same; they are different even if their definitions are ientical
- it is inconvenient but , but OCaml in general requires appears unreasonable 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

How to solve the issue of separate token definitions

- Method 1: instruct `.mll`
  "you do not define token but use the one in `.mly`"
- Method 2: instruct `.mly`
  "you do not define token but use the one in `.mll`"
- you can do either one, but the following explains the method 1

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

How to solve the separate token definitions

- change `calc_lex.mll` as follows:

```
1  {
2    (* remove token definition here *)
3    (* {\it with the following magic, you now reference {\tt PLUS} etc. in}
4       {\tt calc_parse.ml} *)
5  open Calc_parse
6  }
7  rule lex = parse
8  | [' ' '\t' '\n']  { lex lexbuf }
9  | "+"              { PLUS }
10 | ['0'-'9']+ as s  { NUM(int_of_string s) }
11 | eof              { EOF }
```

- I will explain the "magic" shrotly

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

How to run programs consisting of multiple files

- even with that, any of the following does not work

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

- reason: `ocaml` command does not take *multiple* `.ml` *files*
- *for your piece of mind, do not consider* `ocaml` *a command that directly executes an* `.ml` *file*
- you instead compile `.ml` files with `ocamlc` and pass generated files (.cmo)

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
```

How to run programs consisting of multiple files

```
$ ocamllex calc_lex.mll
$ ocamlyacc calc_parse.mly
  # compile with ocamlc (the order is important)!
  # parse must come before lex
$ ocamlc -c calc_parse.mli calc_parse.ml calc_lex.ml
  # pass .cmo to ocaml
$ 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>
```

- they finally combine successfully

```
# 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

Summary

- in OCaml, directly executing `.ml` is an exception
- the norm is to *compile* `.ml` *into* `.cmo` *(byte code) and give them to* `ocaml`
- it is *"must"* when a program consists of multiple files
- to create `.cmo`, compile `.ml` files with `ocamlc -c`, but their order in the command line is *important*
- rule: a depender must come after dependees
  - `calc_lex.ml` refererences `token` defined in `calc_parse.ml`
    → `calc_parse.ml calc_lex.ml`

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/calc_lex.cmo _build/calc_parse.cmo
        OCaml version 4.01.0
#
```

a convenient command : ocamlbuild

- a build tool for OCaml
- a lengthy series of incomprehensible command lines is unpleasant but still convenient

```
$ 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
  # generated files are in _build folder
  # you still need -I _build
$ ocaml -I _build _build/calc_lex.cmo _build/calc_parse.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
#
```

ocamlmktop

- after creating `*.cmo`'s, you could give them to `ocaml` every time like

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

but you can also generate an interactive command (`calc.top` below) they are "burned in" as follows

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

```
$ ./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

Summary of what you must know when developing multi-file programs in OCaml

- when you refer to a name (function/variable names, type names, constructor names, etc.) defined in another file (ex: `abc.ml`)
  - method 1: qualify names like "`Abc.`*name*"
  - method 2: write `open Abc` in the file that references them
- compile them with `ocamlc`, in the order of dependencies
- `ocaml` and programs generated by `ocamlmktop` specify with `-I` directories to search for `*.cmi` and `*.cmo`

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:

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

Common issues in grammar definitions (1) left and right associativity

- what if we change the above grammar

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

into this?

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

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

- the original rule reflects the fact that addition (+) is "left associative"
- left associative:

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

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

Common issues in grammar definitions (2) precedence

- say we want to include * (multiplication)
- any issue with the following?

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

- with this definition

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

- we like to incorporate the fact that "a multiplication binds its operands more strongly than an addition"

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

- one way is to introduce more symbols
  - * binds "number"s to make a "term"
  - + then binds "term"s to make an "expr"

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

Remark:

- `ocamlyacc` has mechanisms to solve these issues declaratively
  - `%left` (left associative)
  - `%right` (right associative)
  - their order indicates their precedences
- refer to the manual if you want to use them
- if you don't know what they exactly mean, a more straightforward approach is to introduce more symbols