

ocaml 第四期レポート課題

高口 奨一郎
学籍番号：6319045

2020 年 7 月 30 日

1 解いた問題

問題 1 – 8 まで

2 問題 1

字句解析部の実装を次に示す．省略してある部分にコードを付加し，
‘‘module Lexer = struct … end’’ で括ってプログラムを完成させなさい．
さらに，四則演算同様，gettoken を繰り返し呼び出す ‘‘run’’ 関数を実装し，振舞を確かめなさい．

2.1 プログラムソース

```
1 module Lexer = struct
2
3 module P = Printf
4
5 type token = CID of string | VID of string | NUM of string
6             | TO | IS | QUIT | OPEN | EOF | ONE of char
7
8 exception End_of_system
9
10 let _ISTREAM = ref stdin
11
12 let ch = ref []
13
14 let read () = match !ch with [] -> input_char !_ISTREAM
15                  | h::rest -> (ch := rest; h)
16
17 let unread c = ch := c::!ch
18
```

```

19 let lookahead () = try let c = read () in unread c; c
20                               with End_of_file -> '$'
21 let rec integer i =
22 (* 文字列として数字を構成 *)
23   let c = lookahead () in
24   if (c >= '0' && c <= '9') then
25     integer (i^(Char.escaped (read ())))
26   else i
27
28 and identifier id =
29   let c = lookahead () in
30   if ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ||
31       (c >= '0' && c <= '9') || c == '_') then
32     identifier (id^(Char.escaped (read ())))
33   else id
34
35 and native_token () =
36   let c = lookahead () in
37   if (* CID に対する識別子および予約語 *)
38     (c >= 'a' && c <= 'z') then
39     let id = identifier "" in
40     match id with
41     | "is" -> IS
42     | "quit" -> QUIT
43     | "open" -> OPEN
44     | "'eof'" -> EOF
45     | _ -> CID (id)
46   else if (* VID に対する識別子 *)
47     (c >= 'A' && c <= 'Z') then VID (identifier "")

```

```

48     else if (c >= '0' && c <= '9') then NUM (integer "")
49     else if (* :- を認識して TO を返す *)
50         (c= ':') then (let sub1 = read () in let sub2 = read ()
51                         in if sub2 = '-' then TO
52                             else ONE (sub1))
53     else ONE (read ())
54
55 and gettoken () =
56 try
57 let token = native_token () in
58 match token with
59     ONE ' ' -> gettoken ()
60   | ONE '\t' -> gettoken ()
61   | ONE '\n' -> gettoken ()
62   | _ -> token
63 with End_of_file -> EOF
64
65 let print_token tk =
66 match tk with
67 (CID i) -> P.printf "CID(%s)" i
68 | (VID i) -> P.printf "VID(%s)" i
69 | (NUM i) -> P.printf "NUM(%s)" i
70 | (TO) -> P.printf ":-"
71 | (QUIT) -> P.printf "quit"
72 | (OPEN) -> P.printf "open"
73 | (IS) -> P.printf "is"
74 | (EOF) -> P.printf "eof"
75 | (ONE c) -> P.printf "ONE(%c)" c
76

```

```

77 let rec run () =
78     flush stdout;
79     let rlt = gettoken () in
80     match rlt with
81     (ONE '$') -> raise End_of_system
82     | _ -> (print_token rlt; P.printf "\n"; run())
83
84 end

```

2.2 省略部分のプログラムの説明

2.2.1 21 行目の関数 `integer`

この関数は入力された数字を文字列として返す関数である。まず `lookahead` を用いて文字を見る。その文字が数字 (0 から 9) であれば `read` を用いてその値を `Char.escaped` に与えることで文字列に変更する。また関数 `integer` を再帰的に用いることでバラバラな文字として認識されていた数字をひとつにまとめる。なお、入力された文字が数字でない場合にはそのまま `i` を返す。

2.2.2 37 行目の関数 `native __ token` の内部

この関数は関数 `identifier` を用いてそれぞれの文字のトークンが何にあたるかを判別していく。まず、先頭が小文字で始まるものについて考える。小文字で始まる文字列のとき、`is`, `quit`, `open.`, `eof`, `CID` のいずれかであるので `match` 文を用いてそれぞれにあったトークン `IS`, `QUIT`, `OPEN`, `EOF`, `CID(id)` を返す。次に先頭が大文字のアルファベットで始まるものについては `VID` トークン、先頭が数字で始まるものについては `NUM` トークンを返す。先頭が `:` であるときには `:` を読み飛ばしたあとで `:` の次の文字を `read` で見えていれば `TO` を返し、そうでなければただの `コロン` として `ONE(sub1)` を返す。

2.3 実行結果

以下のように正しく動作していることがわかる。

```
val run : unit -> unit = <fun>
# open Lexer;;
# gettoken();;
- :
- : Lexer.token = TO
# gettoken();;
- : Lexer.token = ONE '-'
# gettoken();;
- :
- : Lexer.token = TO
# gettoken();;
- : Lexer.token = ONE '-'
# gettoken();;
open
- : Lexer.token = OPEN
# gettoken();;
quit
- : Lexer.token = QUIT
# gettoken();;
eof
- : Lexer.token = EOF
# gettoken();;
is
- : Lexer.token = IS
# gettoken();;
suj
- : Lexer.token = CID "suj"
# gettoken();;
SUJ
- : Lexer.token = VID "SUJ"
# gettoken();;
+
- : Lexer.token = ONE '+'
# gettoken();;
8
- : Lexer.token = NUM "8"
# gettoken();;
- :
- : Lexer.token = TO
#
```

```
# run();;
1+2;
NUM(1)
ONE(+)
NUM(2)
ONE(;)
father(masuo,tara).
CID(father)
ONE()
CID(masuo)
ONE(.)
CID(tara)
ONE())
ONE(.)
MOTHER
VID(MOTHER)
```

3 問題 2

文法中の, terms, args は, 左再帰の形をしているので, 右再帰に変換する必要がある. 修正した文法を示しなさい. 左再帰を取り除いた文法から作成した構文解析器は, 次のようになる.

3.1 左再帰を除いた文法

新しく terms', args' を定義することで左再帰を除き、右再帰に変換することができる。

3.1.1 terms

terms \rightarrow term terms'
terms' \rightarrow “,” term terms'
terms' \rightarrow

3.1.2 args

args \rightarrow expr args'
args' \rightarrow “,” expr args'
args' \rightarrow

4 問題 3

構文解析器プログラムにある空欄を埋めて、プログラムを完成させなさい。

4.1 プログラムソース

```
1 module Parser = struct
2
3 module L = Lexer
4
5 (* module E = Evaluator *)
6
7 let tok = ref (L.ONE ' ')
8
9 let getToken () = L.gettoken ()
10
11 let advance () = (tok := getToken(); L.print_token (!tok))
12
13 exception Syntax_error
14
15 let error () = raise Syntax_error
16
17 let check t = match !tok with
18   L.CID _ -> if (t = (L.CID "")) then () else error()
19   | L.VID _ -> if (t = (L.VID "")) then () else error()
20   | L.NUM _ -> if (t = (L.NUM "")) then () else error()
21   | tk -> if (tk=t) then () else error()
22
23 let eat t = (check t; advance())
24
```



```

25 let rec clauses() = match !tok with
26     L.EOF -> ()
27     | _ -> (clause(); clauses())
28
29 and clause() = match !tok with
30     L.ONE '(' -> (term(); eat(L.ONE '.'))
31     | _ -> (predicate(); to_opt(); eat(L.ONE '.'))
32
33 and to_opt() = match !tok with
34     L.TO -> (eat(L.TO) ; terms())
35     | _ -> ()
36
37 and command() = match !tok with
38     L.QUIT -> exit 0
39     | L.OPEN -> (eat(L.OPEN);
40         match !tok with
41             L.CID s -> (eat(L.CID ""); check (L.ONE '.'));
42             L.ISTREAM := open_in (s ^ ".pl");
43             advance(); clauses(); close_in (!L.ISTREAM))
44     | _ -> error())
45     | _ -> (term(); check(L.ONE '.'))
46
47 and term() = match !tok with
48     L.ONE '(' -> (eat(L.ONE '(') ; term() ; eat(L.ONE ')'))
49     | _ -> predicate()
50
51 and terms() = (term() ; terms' ())
52
53 and terms' () = match !tok with

```

```

54     L.ONE ' , ' -> (eat(L.ONE ' , ' ) ; term() ; terms'())
55     | _ -> ()
56
57 and predicate() = (eat(L.CID "" ) ; eat(L.ONE ' ( ' ) ;
58                      args() ; eat(L.ONE ' ) '))
59
60 and args() = (expr() ; args'())
61
62 and args'() = match !tok with
63     L.ONE ' , ' -> (eat(L.ONE ' , ' ) ; expr() ; args'())
64     | _ -> ()
65
66 and expr() = match !tok with
67     L.ONE ' ( ' -> (eat(L.ONE ' ( ' ) ; expr() ; eat(L.ONE ' ) '))
68     | L.ONE ' [ ' -> (eat(L.ONE ' [ ' ) ; list() ; eat(L.ONE ' ] '))
69     | L.CID s -> (eat(L.CID "" ) ; tail_opt())
70     | L.VID s -> (eat(L.VID "" ))
71     | L.NUM n -> (eat(L.NUM "" ))
72     | _ -> error()
73
74 and tail_opt() = match !tok with
75     L.ONE ' ( ' -> (eat(L.ONE ' ( ' ) ; args() ; eat(L.ONE ' ) '))
76     | _ -> ()
77
78 and list() = match !tok with
79     L.ONE ' ] ' -> ()
80     | _ -> (expr() ; list_opt())
81
82 and list_opt() = match !tok with

```

```

83      L.ONE '|' -> (eat(L.ONE '|'); id())
84      | L.ONE ', ' -> (eat(L.ONE ', '); list())
85      | _ -> ()
86
87 and id() = match !tok with
88     L.CID _ -> (eat(L.CID ""))
89     | L.VID _ -> (eat(L.VID ""))
90     | L.NUM _ -> (eat(L.NUM ""))
91     | _ -> error()
92
93 end
94
95 let rec run() =
96   print_string "?- ";
97   while true do
98     flush stdout; Lexer.ISTREAM := stdin;
99     Parser.advance(); Parser.command(); print_string "\n?- "
100  done

```

4.2 省略部分のプログラムの説明

4.2.1 33 行目の関数 `to __ opt`

`match` 文を用いて入力が入力が `TO` であれば `eat` した上で `terms` を実行、そうでなければ何もしない。

4.2.2 47 行目の関数 `term`

`match` 文を用いて入力の先頭の文字が `' (` であれば `' (` を `eat` した上で `term` の実行、`)` の `eat` を行う。そうでなければ `predicate` を実行する。

4.2.3 51 行目の関数 terms

左再帰を除くため新たに terms' を定義する。問 2 で考えた文法の規則に従ってプログラムする。terms では term を実行した後に terms' を、terms' では match 文を用いて入力の前頭の文字が ' (' であれば ' (' を eat した上で term の実行、terms' の実行を行う。そうでなければ何もしない。

4.2.4 57 行目の関数 predicate

CID、' (' の eat、args の実行、')' の eat を順に行う。

4.2.5 60 行目の関数 args

terms と同じようにして問 2 の文法を参考にしてプログラムを設定する。args では expr を実行した後に args' を、args' では match 文を用いて入力の前頭の文字が ' (' であれば ' (' を eat した上で expr の実行、args' の実行を行う。そうでなければ何もしない。

4.2.6 66 行目の関数 expr

match 文を用いて入力の前頭の文字が ' (' であれば ' (' を eat した上で expr の実行、')' の eat を行う。入力の前頭の文字が [' であれば [' を eat した上で list の実行、']' の eat を行う。入力が CID であれば CID を eat した上で tail __ opt を実行、入力が VID、NUM であれば VID の eat、NUM の eat をそれぞれ実行する。そうでなければエラーメッセージを返す。

4.2.7 74 行目の関数 tail __ opt

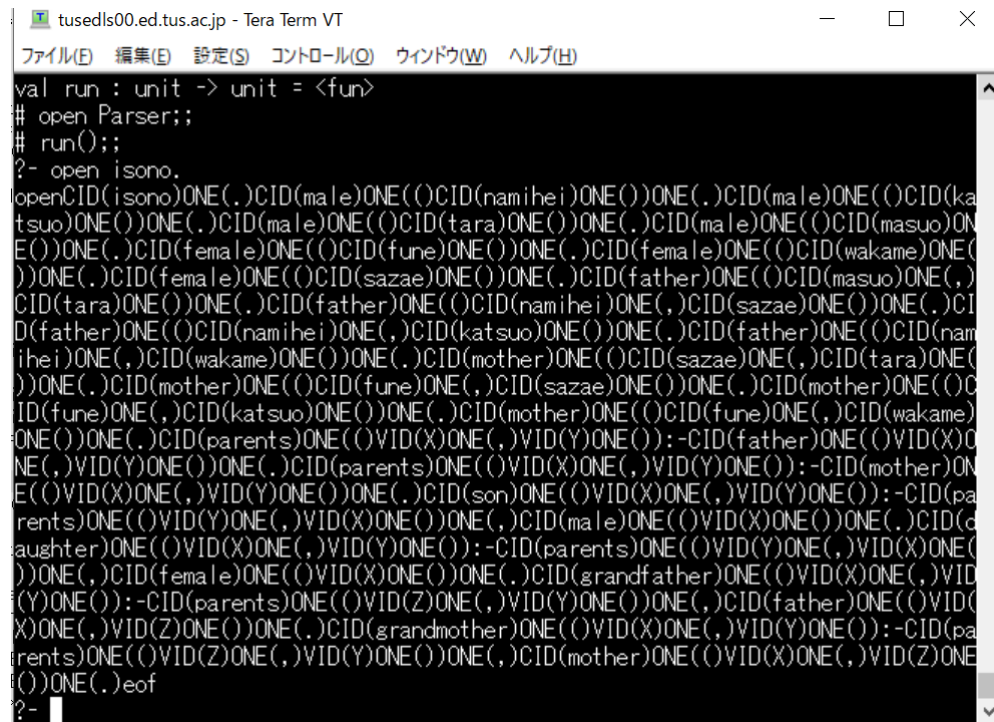
match 文を用いて入力の前頭の文字が ' (' であれば ' (' を eat した上で args の実行、')' の eat を行う。そうでなければ何もしない。

4.2.8 87 行目の関数 id

match 文を用いて入力が CID であれば CID の eat、入力が VID であれば VID の eat、入力が NUM であれば NUM の eat を行う。そうでなければエラーメッセージを返す。

5 問題 4

isono.pl を用いて振舞いを確認しなさい。このように正常に動作している



```
tusedls00.ed.tus.ac.jp - Tera Term VT
ファイル(F) 編集(E) 設定(S) コントロール(C) ウィンドウ(W) ヘルプ(H)
val run : unit -> unit = <fun>
# open Parser;;
# run();;
?- open isono.
openCID(isono)ONE().CID(male)ONE().CID(namihei)ONE().ONE().CID(male)ONE().CID(katsuo)ONE().ONE().CID(male)ONE().CID(tara)ONE().ONE().CID(male)ONE().CID(masuo)ONE().ONE().CID(female)ONE().CID(fune)ONE().ONE().CID(female)ONE().CID(wakame)ONE().ONE().CID(female)ONE().CID(sazae)ONE().ONE().CID(father)ONE().CID(masuo)ONE().CID(tara)ONE().ONE().CID(father)ONE().CID(namihei)ONE().CID(sazae)ONE().ONE().CID(father)ONE().CID(namihei)ONE().CID(katsuo)ONE().ONE().CID(father)ONE().CID(namihei)ONE().CID(wakame)ONE().ONE().CID(mother)ONE().CID(sazae)ONE().CID(tara)ONE().ONE().CID(mother)ONE().CID(fune)ONE().CID(sazae)ONE().ONE().CID(mother)ONE().CID(fune)ONE().CID(katsuo)ONE().ONE().CID(mother)ONE().CID(fune)ONE().CID(wakame)ONE().ONE().CID(parents)ONE().VID(X)ONE().VID(Y)ONE().-CID(father)ONE().VID(X)ONE().VID(Y)ONE().ONE().CID(parents)ONE().VID(X)ONE().VID(Y)ONE().-CID(mother)ONE().VID(X)ONE().VID(Y)ONE().ONE().CID(son)ONE().VID(X)ONE().VID(Y)ONE().-CID(parents)ONE().VID(Y)ONE().VID(X)ONE().ONE().CID(male)ONE().VID(X)ONE().ONE().CID(daughter)ONE().VID(X)ONE().VID(Y)ONE().-CID(parents)ONE().VID(Y)ONE().VID(X)ONE().ONE().CID(female)ONE().VID(X)ONE().ONE().CID(grandfather)ONE().VID(X)ONE().VID(Y)ONE().-CID(parents)ONE().VID(Z)ONE().VID(Y)ONE().ONE().CID(father)ONE().VID(X)ONE().VID(Z)ONE().ONE().CID(grandmother)ONE().VID(X)ONE().VID(Y)ONE().-CID(parents)ONE().VID(Z)ONE().VID(Y)ONE().ONE().CID(mother)ONE().VID(X)ONE().VID(Z)ONE().ONE().eof
?- 
```

ことがわかる。

6 問題 5

上記で示した文法とプログラムは、質問として 1 つの述語しか与えることができない。「,」で区切って複数の述語を記述できるようにしなさい。

6.1 アルゴリズムの説明

述語を与える predicate を”,” で区切ったときにもう一度再帰的に呼び出せばよい。

6.2 プログラム

```
1 and predicate() = (eat(L.CID "") ; eat(L.ONE '(') ; args() ;
2                               eat(L.ONE ')') ; predicate'())
3
4 and predicate'() = match !tok with
5   L.ONE ', ' -> (eat(L.ONE ', ') ; eat(L.CID "") ;
6     eat(L.ONE '(') ; args() ; eat(L.ONE ')') ; predicate'())
7   | _ -> ()
```

6.3 プログラムの説明

関数 predicate の最後に関数 predicate' を加える。関数 predicate に加えて新しい関数 predecate' を定義する。関数 predicate' は match 文を用いて入力先頭の文字が’,’であれば’,’を eat した上で関数 predicate と同じ操作を行う。そうでなければ入力がこれ以上ないということなので何もしない。

6.4 実行結果

以下のように正しく動作していることがわかる。

```
?- grandfather(namihei,tara),father(namihei,katsuo).
CID(grandfather)ONE(,)CID(namihei)ONE(,)CID(tara)ONE(,)ONE(,)CID(father)ONE(,)CID(namihei)ONE(,)CID(katsuo)ONE(,)ONE(,)
?- 
```

7 問題 6

文法中にエラーが生じた際に、エラーメッセージを印字するようにしなさい。エラーメッセージには、エラーが生じた箇所の行番号を含むこと。元のプログラムでは、文法エラーに対して例外を発生させるので、この例外を捕まえて処理する必要があるだろう。OCaml で例外処理を記述するには、次のようにする。

7.1 アルゴリズムの説明

改行を示すトークンを判別するたびにカウントを増やし、文法エラーが生じたタイミングで箱に格納しておいたカウントをエラーが発生した行数としてエラーメッセージと共に出力すればよい。

7.2 プログラムの説明

```
1 let count = ref 1
2
3 let sub () = count := !count + 1
4
5 let reset () = count := 0
```

Lexer モジュール内に以上の 3 つの関数を定義する。関数 `count` は行番号をカウントする関数である。関数 `sub` は呼び出されるたびに `count` の値を 1 増加させる。関数 `reset` は `count` の初期値を 0 にする。

```
1 and gettoken () =
2   try
3     let token = native_token () in
4     match token with
5     | ONE ' ' -> gettoken ()
6     | ONE '\t' -> gettoken ()
7     | ONE '\n' -> sub () ; gettoken ()
```

```

8      | _ -> token
9 with End_of_file -> EOF

```

改行を判別するたびに count の値を 1 増加させる関数 sub を呼び出している。

```

1 let rec run() =
2     print_string "?- ";
3     while true do
4         flush stdout; Lexer.ISTREAM := stdin;
5         Parser.advance(); try Parser.command() with Parser.Syntax__
6             print_int (!Lexer.count); print_string 行目"Syntax_error" ;
7             print_string "\n?- " ;
8             Lexer.reset () ;

```

Parser.command() によって発生したエラーを捕まえて print によって ? 行目 Syntax__ error と出力されるように設定した。

7.3 実行結果

このように文法エラーが生じた行とエラーメッセージを返す。



```

?- father(X,Y),
mother.
CID(father)ONE(VID(X)ONE(,VID(Y)ONE())ONE(,CID(mother)ONE(,2行目 Syntax_error
?- father(X,Y),
mother(X, Y).
CID(father)ONE(VID(X)ONE(,VID(Y)ONE())ONE(,CID(mo)CID(th)3行目 Syntax_error
?- father(X,Y),
mother(X, Y),
mother(X, Y).
CID(father)ONE(VID(X)ONE(,VID(Y)ONE())ONE(,CID(mother)ONE(VID(X)ONE(,3行
目 Syntax_error
?- father(X,Y),
mother(X, Y),
mother(X, Y),
mother(X, Y).
CID(father)ONE(VID(X)ONE(,VID(Y)ONE())ONE(,CID(mother)ONE(VID(X)ONE(,3行
目 Syntax_error
?-

```


8 問題 7

SProlog の文法に算術式を加え、それに基づいて構文解析器を拡張しなさい。付加する生成規則は次のようになる。元の文法において、`expr` の生成規則の右辺を、`arithmexp` で置き換えるとよい。

8.1 アルゴリズムの説明

生成規則に従ってプログラムを構成すればよい。ただし、生成規則をそのまま反映させると `arithmexp` と `arithmterm` は左再帰になっているので右再帰に変換した文法を考える。以下に右再帰に変換した文法を示す。`arithmfactor` に関しては左再帰を含んでいないので生成規則通りにプログラム記述する。

8.1.1 `arithmexp`

```
arithmexp → arithmterm arithmexp'
arithmexp' → "+" arithmterm arithmexp'
arithmexp' → "-" arithmterm arithmexp'
arithmexp' →
```

8.1.2 `arithmterm`

```
arithmterm → arithmfactor arithmterm'
arithmterm' → "*" arithmfactor arithmterm'
arithmterm' → "/" arithmfactor arithmterm'
arithmterm' →
```

8.2 プログラム

```
1 and arithmexp() = (arithmterm() ; arithmexp'())
2
3 and arithmexp'() = match !tok with
4   L.ONE '+' -> (eat(L.ONE '+') ; arithmterm() ; arithmexp'())
5 | L.ONE '-' -> (eat(L.ONE '-') ; arithmterm() ; arithmexp'())
```

```

6 | - -> ()
7
8 and arithmterm() = (arithmfactor() ; arithmterm'())
9
10 and arithmterm'() = match !tok with
11 | L.ONE '*' -> (eat(L.ONE '*') ; arithmfactor() ; arithmterm'())
12 | L.ONE '/' -> (eat(L.ONE '/') ; arithmfactor() ; arithmterm'())
13 | - -> ()
14
15 and arithmfactor() = match !tok with
16 | L.ONE '(' -> (eat(L.ONE '(') ; arithmexp() ; eat(L.ONE ')'))
17 | L.ONE '-' -> (eat(L.ONE '-') ; arithmexp())
18 | L.ONE '[' -> (eat(L.ONE '[') ; list() ; eat(L.ONE ']'))
19 | L.CID s -> (eat(L.CID s) ; tail_opt())
20 | L.VID s -> (eat(L.VID s))
21 | L.NUM n -> (eat(L.NUM n))
22 | - -> error()

```

8.3 プログラムの説明

関数 `arithmexp` は `arithmterm()` を実行した後に `arithmexp'()` を実行する。関数 `arithmexp'` は `match` 文を用いて先頭の文字が+であれば'+'を `eat` した上で `arithmterm()` を実行し、`arithmexp'()` を実行する。先頭の文字が-であれば '-' を `eat` した上で `arithmterm()` を実行し、`arithmexp'()` を実行する。そうでなければ何もしない。ここがループが停止するところである。

関数 `arithmterm` は `arithmfactor()` を実行した後に `arithmterm'()` を実行する。関数 `arithmterm'` は `match` 文を用いて先頭の文字が*であれば '*' を `eat` した上で `arithmfactor()` を実行し、`arithmterm'()` を実行する。先頭の文字が/であれば '/' を `eat` した上で `arithmfactor()` を実行し、`arithmterm'()`

を実行する。そうでなければ何もしない。ここがループが停止するところである。

関数 `arithmfactor` は `match` 文を用いて先頭の文字が `' (` であれば `' (` を `eat` し、`arithmexp()` を実行、`)'` を `eat` する。先頭の文字が `' -` であれば `' -` を `eat` し、`arithmexp()` を実行、先頭の文字が `' [` であれば `' [` を `eat` し、`list()` を実行、`]'` を `eat` する。先頭の文字が `CID` であれば `CID` を `eat` し、`tail __ opt()` を実行、先頭の文字が `VID`、`NUM` であれば `VID`、`NUM` を `eat` する。そうでなければエラーを返す。

8.4 実行結果

```
1 and expr() (*= match !tok with
2   L.ONE '(' -> (eat(L.ONE '(') ; expr() ; eat(L.ONE ')'))
3   | L.ONE '[' -> (eat(L.ONE '[') ; list() ; eat(L.ONE ']'))
4   | L.CID s -> (eat(L.CID s) ; tail_opt())
5   | L.VID s -> (eat(L.VID s))
6   | L.NUM n -> (eat(L.NUM n))
7   | _ -> error()
8
9 and arithmexp() *)=
```

この部分をコメントアウトする。

以下のように正しく動作していることがわかる。

```
val id : unit -> unit
end
val run : unit -> unit = <fun>
# open Parser;;
# run();;
?- mother (x+y).
CID(mother)ONE(CID(x)ONE(+)CID(y)ONE())ONE(.)
?- father ([3]).
CID(father)ONE(ONE([)NUM(3)ONE(])ONE())ONE(.)
?-
```

9 問題 8

変数への代入を記述できるようにしなさい。必要な生成規則は次のようになる。

9.1 アルゴリズムの説明

生成規則に従って term に新たな規則を追加すればよい。

9.2 プログラム


```
1 and term() = match !tok with
2   L.ONE '(' -> (eat(L.ONE '(') ; term() ; eat(L.ONE ')'))
3   | L.VID s -> (eat(L.VID s) ; eat(L.IS) ; arithmexp())
4   | _ -> predicate()
```

9.3 プログラムの説明

問題 2 で作成した関数 term に次のプログラムを加える。先頭の文字が VID であれば VID を eat し、IS を eat、arithmexp() を実行する。

9.4 実行結果

このように正常に動作していることがわかる。

 tusedls00.ed.tus.ac.jp - Tera Term VT

ファイル(F) 編集(E) 設定(S) コントロール(O)

```
^CInterrupted.  
# open Parser;;  
# run();;  
?- KOTAE is 2.  
VID(KOTAE)isNUM(2)ONE(.)  
KAITOU is 15.  
VID(KAITOU)isNUM(15)ONE(.)  
SEISEKI is 9.  
VID(SEISEKI)isNUM(9)ONE(.)
```