情報システム科学実習II第10回
LOGO インタプリタの作成

担当: 山口 和紀・五十嵐 淳

2002年1月9日

参考文献

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986. 日本語訳: A.V. エイホ，R. セシィ，J.D. ウルマン共著，原田賢一訳．コンパイラ：原理・技法・ツール I & II． サイエンス社.

[2] Brian Harvey. *Berkeley logo User Manual*. `http://www.cs.berkeley.edu/~bh/`.

[3] Brian Harvey. *Computer Science Logo Style*, volume 1. MIT Press, 1997.

[4] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.02: Documentation and user's manual*, 2001. `http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`.

今回の課題では，これまでに学習したことをもとに，LOGO という簡単なプログラミング言語の (サブセットの) インタプリタを構成する．

1　LOGO 言語

LOGO 言語 [3, 2] は 1960 年代後半に MIT で開発された (子供向け) 教育用プログラミング言語である．子供向けとはいえ，変数/手続きなどの概念を持っている．LOGO の特徴の一つとして，画面上に置かれた「タートル」を操作することによって描画を行うための，タートルグラフィクス命令群があげられる．

いくつか簡単なタートルグラフィクスプログラムを見てみよう．例えば

```
fd(100) lt(90)
fd(100) lt(90)
fd(100) lt(90)
fd(100) lt(90)
```

で画面に正方形を描くことができる．fd は forward 命令の略記で，タートルを前方に動かすことができる．その時に，タートルの移動に沿ってグラフィクス画面に直線が引かれる．lt は left 命令の略でタートルの位置はそのままで向きだけを左に，(この場合は)90 度回転させる．これを 4 回繰り返すことで正方形を描くことができる．

本来の LOGO の文法では引数のまわりの () は要らないのだが，今回の課題では命令/手続き呼び出しの文法を C 言語風に変更して，引数は (複数ある場合は , で区切り) () で囲むようにする．

その他，タートルグラフィクス命令には表1のようなものがある．座標系は画面の中心が原点で x 軸は水平右方向が正，y 軸は垂直上方向が正，角度は時計回りが正の方向である．

表 1: タートルグラフィクス命令

命令	引数の数	内容
fd または forward	1	タートルを前進させる．
bk または back	1	タートルを後退させる．
lt または left	1	タートルを左 (反時計回り) に回転
rt または right	1	タートルを右 (時計回り) に回転
setxy	2	タートルの位置を与えられた座標に移す
setx	1	タートルの x 座標を与えられた値の位置に移す
sety	1	タートルの y 座標を与えられた値の位置に移す
setheading	1	タートルの向きを，与えられた値の向きに変える．ただし，真上を 0 とし時計回りを正の向きとする．
home	0	タートルをもとの位置 (原点，向きは真上) に移す．
arc	2	タートル位置を中心，半径を第 2 引数として，タートルの向きから第 1 引数度ぶんだけの円弧を描く．
penup	0	ペンをあげて，タートルが移動しても描画されないようにする．
pendown	0	ペンを下げて，タートルの移動とともに描画されるようにする．
clean	0	画面をクリアする．
clearscreen	0	画面をクリアし，タートルをもとの位置に移す．
showturtle	0	タートルを表示する．
hideturtle	0	タートルの表示をやめる．
wrap	0	画面をはみ出したタートル/図形は反対側に描かれるようにする．
window	0	はみ出したものは書かない．タートルは画面の外にいることができる．
fence	0	画面の端に壁があるように，タートルは画面の端より外に移動できない．はみ出したものは書かない．

また，端末への数値の表示として print 命令，四則演算関数として sum, difference, product, quotient が，比較関数として，lessp, greaterp が用意されている．また，これらの関数の代りに，中置オペレータとして +, -, *, /, <, > なども使用することができる．
また，繰り返しの記述は repeat 命令を用いて

```
repeat 4 begin fd(100) lt(90) end
```

と書くこともできる．一命令以上を繰り返すには begin end で囲む．(本来の文法では [] で囲む．) 他の制御構造として， if 文 (else 節は省略可能) で条件分岐を行うことができる．
手続き定義は，proc (本来の文法は to) キーワードで行う．まず，宣言の 1 行目として，

proc ⟨ 手続き名 ⟩ ⟨ 変数名$_1$⟩ . . . ⟨ 変数名$_n$⟩

のように手続き名とパラメータ (なくてもよい) を宣言する．変数名は最初が : で始まる (アルファベットを使った) 名前でなくてはならない．すると，プロンプトが % に変って，手続き本体を入力するモードに切り替わる．手続き本体は end だけからなる行で入力を終わらせることができる．手続きの実行を途中で抜けるためには，返り値を返さない stop 命令，返り値を返す output(...) 命令を用いることができる．
以下は木を描く有名なプログラムである．

```
proc tree :n
if :n < 5 then stop
fd(:n)
lt(30) tree(product(:n, 0.7))
rt(60) tree(:n * 0.7)
lt(30) bk(:n)
end

tree(100)
```

この課題では,

1. タートルグラフィクス命令・制御構造命令・算術演算/比較関数を備えたインタプリタ
2. 手続き定義が可能なインタプリタ
3. 中置オペレータを使った式の記述が可能なインタプリタ

を段階的に作成してゆく．

2 インタプリタの構成

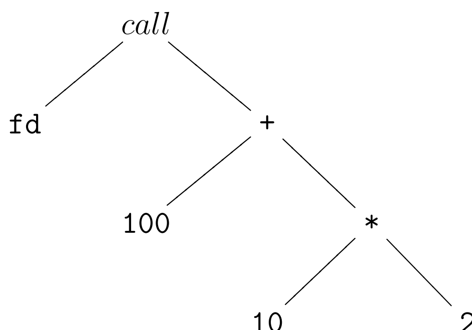
基本的なインタプリタの仕事は，3 ステップに分けられる．

字句解析 入力文字列を，識別子，算術演算記号，括弧記号など，プログラム上で最小の意味単位である**トークン**(*token*) と呼ばれる記号の列に変換する，

構文解析 トークン列を解析して，ひとまとまりの意味を持つ単位で構造化する．また，このような構造を定義する規則 (一種の文法規則と見なせる) は，具体的なプログラムに現れる括弧などの区切り記号を含まず，プログラムの意味の本質的な部分のみを抽象化したものであることから，**抽象構文**(*abstract syntax*) と呼ばれる．これに対し，区切り記号なども含んだプログラム文面の構文は **具象構文**(*contrete syntax*) と呼ぶことがある．抽象構文は木構造をしていることが多いため，構文解析をした結果を**抽象構文木**(*abstract syntax tree*) と呼ぶことがある．

解釈器による解釈 抽象構文木にしたがって，プログラムを実行する．

たとえば，`fd(100 + 10 * 2)` という文字列を考えてみよう．まず，字句解析部分が，この文字列を，識別子 `fd`, 開き括弧記号 `(`, 数字 `100`，加算記号 `+`, 数字 `10`, 乗算記号 `*`, 数字 2, 閉じ括弧記号 `)` というトークン列に分解する．次に，このトークン列を構文解析で次のような構造に変換する．



最後に，この構文木を解釈器がボトムアップに乗算，加算，手続き呼び出しを行って描画を行う．

以下で説明を行うプログラムは，授業の web ページ `http://www.graco.c.u-tokyo.ac.jp/~igarashi/class/syslabII01.html` からリンクしてあるので，各自ダウンロードすること．課題は，プログラムの一部抜けている部分を補うものや，インタプリタに新たな機能を付け足すものなどになる．

プログラムファイルはモジュールごとにファイルに分割されているが，`Makefile` を用意したので，すべてのファイルを一つのディレクトリにいれて，端末で

```
touch .depend
make dep
make
```

とすれば，`logo` という名前でインタプリタがコンパイルされる．

ただ，開発途中では，インタラクティブに関数をテストしたい場合もあるだろう．kterm などの端末から

```
ocamlmktop -o test str.cma graphics.cma scanner.cmo ...
```

と必要なモジュールをコンパイルした `.cmo` ファイルを並べると，`Str`, `Graphics`, `Scanner` などのモジュールをあらかじめ読み込んだインタラクティブコンパイラ `test` を生成することができる．

注意　ただし，これらのモジュールは以下のように open されていない状態でコンパイラが開始される．

```
igarashi@quena:logo> ./test
        Objective Caml version 3.02

# proc_id;;
Unbound value proc_id
# Parser.proc_id;;
- : string Parser.phrase = <fun>
```

また，第 8 回 で学んだようにモジュールのシグネチャ (.mli) に書いていない関数は .ml に書いてあっても利用できない．この場合，対応する .mli にその関数の型を書くか，.mli と .cmi を削除してから，再コンパイルするかしなくてはならない．

2.1　Scanner モジュール

これは，字句解析を行うためのモジュールである．token はトークンを表現するヴァリアント型で，コンストラクタは手続き名，変数名，数字，記号の 4 種類のトークンを表現している．

文字列からの抽出は scan 関数が行っており，ライブラリの Str モジュールの string_match, regexp, matched_string といった，正規表現を使って文字列マッチを行う関数 (詳細はマニュアル [4] 参照) を使っている．

実行例

```
# Scanner.scan "fd(100,200) ( + 2.3(";;
- : Scanner.token list =
[Scanner.ProcId "fd"; Scanner.Sym "("; Scanner.Num 100; Scanner.Sym ",";
 Scanner.Num 200; Scanner.Sym ")"; Scanner.Sym "("; Scanner.Sym "+";
 Scanner.Num 2.3; Scanner.Sym "("]
```

Exercise 10.1 (* missing *) に入る適当な式を入れて scanner.ml を完成させよ．また，負の数 -10 などにもマッチするように変更せよ．

Exercise 10.2 LOGO 言語のコメントは ; から行末までとなっている．scanner.ml を変更し，コメントを受理するようにせよ．(コメントはトークンを返さないものとせよ．)

2.2　Parser モジュール

このモジュールは構文解析を行うためのツールとなる関数群である．具体的な LOGO 言語の文法にそった構文解析ルーチンは syntax.ml で定義される．

構文解析関数は，トークンの列 (リスト) を受け取って，先頭からいくつかのトークンを消費して解析を行った結果と，残りのトークン列を返す関数として定義する．この型を，解析を行った結果の型をパラメータ 'a として 'a phrase として定義する．

```
type 'a phrase = Scanner.token list -> 'a * Scanner.token list
```

- proc_id は string phrase で，トークン列の先頭にある手続き名の文字列を返す．(先頭が手続き名トークンでない場合は例外 SyntaxErr を raise する．)var_id も同様に，変数名を解析する．num も同様である．
- key は高階関数で，key s とするとトークンの先頭がキーワード s であるかチェックする 構文解析関数を生成する．解析結果は重要でないため型は unit phrase になっている． sym も同様である．
- ⟨*phrase*$_1$⟩ ||| ⟨*phrase*$_2$⟩ で，(同じ型の) 二つの解析関数を “or” で組み合わせることができる．つまり ⟨*phrase*$_1$⟩ を試して，だめ (例外発生) なら ⟨*phrase*$_2$⟩ を試すような解析関数となる．
- ⟨*phrase*$_1$⟩ ++ ⟨*phrase*$_2$⟩ で，二つの解析関数を連続するように組み合わせられる．つまり，まず ⟨*phrase*$_1$⟩ を試して，残りのトークン列を ⟨*phrase*$_2$⟩ に渡して解析を行う．全体の解析結果は，ふたつの結果の組になる．
- ⟨*phrase*⟩ @>> ⟨ 関数 ⟩ で，⟨*phrase*⟩ の結果に ⟨ 関数 ⟩ を適用するような解析関数を表す．構文木を構成するときにしばしば用いられる．
- repeat ⟨*phrase*⟩ は ⟨*phrase*⟩ をできる限り適用して，得られた結果をリストとして返す解析関数を表す.
- after_sym ⟨ 記号文字列 ⟩ ⟨*phrase*⟩ は，⟨ 記号文字列 ⟩ の後に ⟨*phrase*⟩ が来ることを期待する解析関数である．結果は，⟨*phrase*⟩ の結果をそのまま返す．after_key も同様である．

実行例　先頭の open は ++ などを中置オペレータとして使うために必要なものである．

```
# open Parser;;

# (proc_id ++ key "if") (Scanner.scan "fd if (");;
- : (string * unit) * Scanner.token list = ("fd", ()), [Scanner.Sym "("]
# (repeat proc_id @>> List.rev) (Scanner.scan "a b c :a (");;
- : string list * Scanner.token list =
["c"; "b"; "a"], [Scanner.VarId ":a"; Scanner.Sym "("]
# (num ++ sym "+" ++ num @>> fun ((x, _), z) -> x +. z)
#   (Scanner.scan "10.2 + 3.4");;
- : float * Scanner.token list = 13.6, []
```

Exercise 10.3 任意の長さの加減算式 (例: `1.0 + 2.0 - 3.0` や `4.2 - 41.4`) を表す文字列を受け取って字句解析，構文解析を行って，計算結果を返すような関数 (型は `string -> float` となるもの) を定義せよ．

2.3 Syntax モジュール

つぎに，`Parser` モジュールを使って，LOGO の文法を定義する．まず，抽象構文木の型，ユーザ入力 (今は式だけであるが，後に手続き定義を導入したときには手続き定義の 1 行目も含まれる．`Decl` コンストラクタがそれを表現する．`of` 以下の `string * string list` は手続き名とパラメータ名を表す．) の型を `t`, `toplevel` としてそれぞれ定義する．あとの拡張を考えて，算術演算なども `t` のコンストラクタとして用意しておく．

まずは，算術式や，手続き定義 (および変数参照) のない LOGO プログラムの具象文法を示す．

⟨ユーザ入力⟩ ::= ⟨式⟩

⟨式⟩ ::= ⟨数値⟩ | ⟨変数⟩ | ⟨*repeat* 文⟩ | ⟨ブロック⟩ | ⟨条件文⟩ | ⟨手続き呼出⟩

⟨*repeat* 文⟩ ::= `repeat` ⟨式$_1$⟩ ⟨式$_2$⟩

⟨ブロック⟩ ::= `begin` ⟨式$_1$⟩...⟨式$_n$⟩ `end`

⟨条件文⟩ ::= `if` ⟨式$_1$⟩ `then` ⟨式$_2$⟩ `else` ⟨式$_3$⟩ | `if` ⟨式$_1$⟩ `then` ⟨式$_2$⟩

⟨手続き呼出⟩ ::= ⟨手続き名⟩ `(` `)` | ⟨手続き名⟩ `(` ⟨引数リスト⟩

⟨引数リスト⟩ ::= ⟨式⟩ `)` | ⟨式⟩ `,` ⟨引数リスト⟩

各文法カテゴリー (⟨式⟩ のようなもの) についてひとつ解析関数を用意する．たとえば `expr` は，トークン列が ⟨式⟩ であることを期待して解析を行う関数である．その内容は，`|||` を使って，可能性のある式の形を次々に試していくものである．文法定義と非常に形が似ていることがわかるだろう．また `rep` 関数をみると，`@>>` を使って構文木が構成するやり方がわかると思う．これらの関数は相互再帰的に呼ばれるために，`let rec` と `and` を使って定義されている．

2.4 ProcEnv と ValEnv モジュール

このふたつのモジュールは，評価する時点での，各手続き定義や，各変数の内容を保持するためのデータ構造を定義している．ここでは `ProcEnv` を主に説明する．型 `t` の定義からわかるように，手続き定義は名前と定義 (`proc`) の組リストで表現される．`proc` 型の値は，タートルグラフィクス命令や算術関数などプリミティブとなる ML の関数，(コンストラクタ `Prim`) もしくは，ユーザが定義する手続き (コンストラクタ `Def`) である．前者が `float list` を引数として受け取るようにしてあるのは，引数の数の違う関数をまとめて扱いたいからである．後者は，パラメータ名のリストと手続き本体となる構文木のリストとして表現される．

関数 add_def は新しい (ユーザによる手続き) 定義を追加するためのもので，手続き名，パラメータ名リスト，手続き本体と古い環境を受け取って，新しい環境を返す．ただし，LOGO の仕様では，同じ名前の手続きの再定義が禁止されているため，そのチェックを行っている．

Exercise 10.4 valEnv.mli のシグネチャに従って，valEnv.ml を実装せよ．add 関数は，変数名のリストとその値のリストと古い環境を受け取って，新しい束縛を付加した環境を返す．ただし，ふたつのリストの長さが一致しない場合は，パラメータの数と実引数の数が一致してないわけなので，エラー (例外発生) となってよい．また，手続き定義の環境とは異なり，同じ変数名で二度定義されてもよい．(これを許さないと，再帰呼び出しの実装が複雑になるだろう．)

2.5 Primitives モジュール

このモジュールはタートルグラフィクス命令，算術関数などを定義したモジュールである．グラフィクスにはライブラリの Graphics モジュールを用いている．詳しいことは OCaml マニュアル [4] を参照してもらいたい．

forward 以下の定義がそれらの命令を実装した関数である．(ただし，一部の命令はきちんと実装されていない．) 各命令は，float list -> Syntax.t 型の関数として定義されており，タートル操作が起こるたびに， 現在位置などを記録している (primitives.mli には現れていない) モジュール内でのみ参照できる変数 turtle の内容を代入で書き換えている．

また，Graphics ライブラリの座標系とこの座標系が原点の位置や角度の単位/正の向きが違っているために変換を行わなければならない．rad_of_deg などを参照してもらいたい．

また，最後にインタプリタ起動時の手続き環境 init_env を定義している．

Exercise 10.5 補助的に使われている move 関数を完成させよ．この関数は，距離と方向 (単位は度) を受け取って，Graphics ライブラリの座標系で x 軸，y 軸方向への移動量を返す．

2.6 Eval モジュール

これが解釈器本体である．eval 関数は，手続きと値の環境，構文木を受け取って，評価を行い，最終的に値 (の構文木) を返す．以下，構文木のコンストラクタによって場合わけの説明を行う．

- すでに値である場合 (Number, Bool, Void) はそれをそのまま返すだけである．
- 制御構造 begin ... end, repeat の解釈はわりと自明であろう．命令列の実行では，途中の命令が返り値を返す物でないことを ensure_void を使って確かめながら実行している．

- 手続き呼出は，まず手続き名を環境から参照して，引数を評価，apply 関数を呼び出している．apply 関数では，手続きがプリミティブであれば (ProcEnv.Prim にマッチすれば) その ML 関数を直接呼出している．

関数 read_eval_loop は，実際に端末からキーボード入力を受け取って解釈することを繰り返し行う．ignore は 'a -> unit 型を持つ関数で，関数の返り値を必要とせず次の動作 (; の後) を行う場合に用いる．ここでは解釈器から返る結果 Void を無視するために使っている．(ちなみに OCaml では ; で区切られた式全体を囲むのに (...;...;...) と書いてもよいが begin, end を使うことができる．)

Exercise 10.6 if 文の実行部の実装を行え．

2.7 Main モジュール

このモジュールは基本的に，画面の初期化などを行って，read_eval_loop を呼び出しているだけである．ここまでのプログラムでも，repeat などは使えるので多少のおもしろいことはできるはずである．

3 手続き定義と呼び出し

手続き定義と呼び出しを実装するためには，

- Syntax.toplevel, Eval.read_eval_loop を変更して，手続き定義をキーボードから入力できるようにする．また，stop 文，output 文を受理するように構文解析関数を変更する．
- Eval.apply を変更して，proc で定義された手続き呼び出しの実行を行えるようにする．
- Eval.eval で変数，stop 文，output 文の解釈を行えるようにする．

ことが必要になる．

手続き定義の入力 まず，Syntax.toplevel 関数を，手続き定義の 1 行目を構文解析できるように変更する (キーボード入力は行ごとに字句/構文解析されるので定義全体を一度に解析することはできない)．文法は以下の通りである．

$$\begin{array}{rcl}\langle\,\text{ユーザ入力}\,\rangle & ::= & \langle\,\text{式}\,\rangle \mid \langle\,\text{手続き定義}\,\rangle \\ \langle\,\text{手続き定義}\,\rangle & ::= & \texttt{proc}\ \langle\,\text{手続き名}\,\rangle\,\langle\,\text{変数}_1\rangle \ldots \langle\,\text{変数}_n\rangle\end{array}$$

解析結果は，Decl (⟨ 手続き名 ⟩,⟨ 変数名リスト ⟩) と構成される．定義の 2 行目以降を入力する部分は，read_eval_loop で定義することになる．match 式に以下のように Decl を処理する節を付加する．

```
match Syntax.reader (read_line ()) with
  Exps es -> ...
| Decl (f, params) ->
    let rec read_body () =
      print_string "% ";
      match Scanner.scan (read_line()) with
        [] -> read_body ()
      | toks ->
          match
            (key "end" @>> (fun _ -> []) ||| repeat expr) toks
          with
            ([], []) -> []
          | (es, []) -> es @ (read_body ())
          | _ -> failwith "Syntax error in proc"
    in
    let body = read_body () in
    Printf.printf "procedure %s defined.\n" f;
    read_eval_loop (ProcEnv.add_def procenv f params body)
```

局所関数 read_body が 2 行目以降の入力を受け付ける部分である．プロンプトとして % を表示して，1 行ごとに字句解析，構文解析を行う．字句解析の結果が空リストであれば，実質空行であるので繰り返す．構文解析では式の列を受理するために repeat expr を使っている．構文解析関数は解析できない部分を第 2 要素として返すので，ふたつめの match 式はそれが空であることを確認している．入力が end で終了したら，環境を拡張して read_eval_loop に戻る．

Exercise 10.7 Syntax.toplevel 関数を変更して，上で示した文法にしたがって手続き定義の 1 行目を受理できるようにせよ．

Exercise 10.8 stop 文と output 文の受理ができるように Syntax モジュールを変更せよ．具体的には Syntax.expr の定義を以下のように変更し，

```
and expr toks =
  (num @>> (fun n -> Number n) ||| var_id @>> (fun s -> Var s) |||
   rep ||| begin_end ||| conditional ||| output ||| stop ||| proc)
    toks
```

以下の文法で示される stop 文，output 文を受理できるように Syntax.stop, Syntax.output 関数を定義せよ．

$$
\begin{array}{rcl}
\langle \text{式} \rangle & ::= & \cdots \mid \langle \textit{output}\,\text{文} \rangle \mid \langle \textit{stop}\,\text{文} \rangle \mid \langle \text{手続き呼出} \rangle \\
\langle \textit{output}\,\text{文} \rangle & ::= & \texttt{output}\ (\ \langle \text{式} \rangle\ ) \\
\langle \textit{stop}\,\text{文} \rangle & ::= & \texttt{stop}
\end{array}
$$

また，Syntax.expr 関数の定義で，||| で結合する各フレーズの順番を変え，以下のようにすると構文解析がうまくいかなくなる理由を説明せよ．

```
and expr toks =
  (num @>> (fun n -> Number n) ||| var_id @>> (fun s -> Var s) |||
  proc ||| rep ||| begin_end ||| conditional ||| output ||| stop)
    toks
```

手続き呼び出し　Eval.apply 関数は，手続きとして与えられるものが，プリミティブだけではなくユーザ定義関数の場合もでてくる．コンストラクタが Def の場合は，基本的には関数本体の実行をすればよいのだが，本体からパラメータの値が参照できるように変数環境を拡張する必要がある．

Exercise 10.9 Eval.apply 関数を完成させよ．

解釈器の変更　変数の評価は，変数環境から変数名を使って値をとってくればよい．

手続きの終了は，残りの命令を無視して手続きから脱出するために例外を用いて処理する．例外 Return を raise することによってそれを行う．つまり，stop, output を処理する部分は以下のように書ける．

```
let return v = raise (Return v)

let rec apply valenv procenv args = function
    ...
and eval valenv procenv = function

    ...

  | Output exp ->
      let v = ensure_nonvoid (eval valenv procenv exp) in
      return v
  | Stop ->
      return Syntax.Void
```

Exercise 10.10 変数参照部分を完成させるとともに，stop, output 命令によって raise された Return を処理する適当な場所を考え，Eval モジュールのどこかを変更して，手続き呼び出しが正しく行えるようにせよ．

4　中置オペレータの導入

中置オペレータのある文法を構文解析するのには工夫がいくらか必要である．単純に，括弧，加減乗除算のある式の文法を

$$\langle 式 \rangle ::= \langle 式 \rangle + \langle 式 \rangle \mid \langle 式 \rangle - \langle 式 \rangle \mid \langle 式 \rangle * \langle 式 \rangle \mid \langle 式 \rangle / \langle 式 \rangle \mid \langle 数値 \rangle \mid (\langle 式 \rangle)$$

と与え，この文法を直接構文解析関数としてプログラムすると，

```
let rec expr toks =
  (expr ++ after_key "+" expr ||| ...) toks
```

のようになるが，これはうまく動作しない．演算子の優先規則が正しく反映されてないこともあるが，最大の問題は，一度の expr の呼び出しが無限ループを引き起こすためである．これは，⟨ 式 ⟩ の規則が，定義される文法カテゴリー自身がそれを定義するものの一番最初の部分式としてでてくる**左再帰**(*left recursive*) と呼ばれるものになっていることが問題である[1]．左再帰を除去するように文法規則を書き換える方法が確立されており，この文法のように簡単であれば，除去することができる．詳しくはコンパイラの教科書 [1] などを参考にしてほしい．

中置オペレータを導入した文法規則は以下のように与えられる．

⟨ ユーザ入力 ⟩	::=	⟨ 式 ⟩ | ⟨ 手続き定義 ⟩
⟨ 手続き定義 ⟩	::=	`proc` ⟨ 手続き名 ⟩ ⟨ 変数$_1$⟩ ... ⟨ 変数$_n$⟩
⟨ 式 ⟩	::=	⟨ 算術式 ⟩ `<` ⟨ 算術式 ⟩ | ⟨ 算術式 ⟩ `>` ⟨ 算術式 ⟩ | ⟨ 算術式 ⟩
⟨ 算術式 ⟩	::=	⟨ 乗除式 ⟩ `+` ⟨ 乗除式 ⟩ | ⟨ 乗除式 ⟩ `-` ⟨ 乗除式 ⟩ | ⟨ 乗除式 ⟩
⟨ 乗除式 ⟩	::=	⟨ 原子式 ⟩ `*` ⟨ 原子式 ⟩ | ⟨ 原子式 ⟩ `/` ⟨ 原子式 ⟩ | ⟨ 原子式 ⟩
⟨ 原子式 ⟩	::=	`(` ⟨ 式 ⟩ `)` | ⟨ 数値 ⟩ | ⟨ 変数 ⟩ | ⟨*repeat* 文 ⟩ | ⟨ ブロック ⟩
		| ⟨ 条件文 ⟩ | ⟨*output* 文 ⟩ | ⟨*stop* 文 ⟩ | ⟨ 手続き呼出 ⟩
⟨*repeat* 文 ⟩	::=	`repeat` ⟨ 式$_1$⟩ ⟨ 式$_2$⟩
⟨ ブロック ⟩	::=	`begin` ⟨ 式$_1$⟩... ⟨ 式$_n$⟩ `end`
⟨ 条件文 ⟩	::=	`if` ⟨ 式$_1$⟩ `then` ⟨ 式$_2$⟩ `else` ⟨ 式$_3$⟩ | `if` ⟨ 式$_1$⟩ `then` ⟨ 式$_2$⟩
⟨*output* 文 ⟩	::=	`output` `(` ⟨ 式 ⟩ `)`
⟨*stop* 文 ⟩	::=	`stop`
⟨ 手続き呼出 ⟩	::=	⟨ 手続き名 ⟩ `( )` | ⟨ 手続き名 ⟩ `(` ⟨ 引数リスト ⟩
⟨ 引数リスト ⟩	::=	⟨ 式 ⟩ `)` | ⟨ 式 ⟩ `,` ⟨ 引数リスト ⟩

Exercise 10.11 構文解釈関数 (群) を変更して，上の文法を受理できるようにせよ．この際，`+`, `-`, `*`, `/`, `<`, `>` に対応するコンストラクタを使用して構文木を構成せよ．例えば加算式 `1 + 2` は `Plus (Number 1.0, Number 2.0)` という構文木を構成する．また，`Plus` などを解釈できるように `eval` 関数を変更せよ．

5 その他の練習問題

Exercise 10.12 インタプリタの機能拡張を行え．例としては

- 描画色を変えるための命令の追加
- 文法エラーの箇所の表示，エラー理由の表示など，よりよいエラー処理機能

[1] 左再帰文法規則は，ここで扱った**トップダウン構文解析**(*top down parsing*) と呼ばれる方法では解析できないだけであり，左再帰の文法を扱える構文解析を実装する**ボトムアップ構文解析**(*bottom up parsing*) と呼ばれる方法もある．

- 手続きのパラメータとして以外の変数宣言
- `wrap`, `window`, `fence` 命令の実装
- `showturtle`, `hideturtle` 命令の実装
- ファイルからのプログラム入力命令の追加

などがあげられる．

Exercise 10.13 自分の作成したインタプリタで何か図形を描く LOGO プログラムを作成せよ．(図形/プログラムの主観的な美しさを採点対象とする．) また，`http://ajlogo.com/` に Java アプレットによる LOGO インタプリタを用いた様々な LOGO プログラムがあるので，(もちろんプリミティブが足りなかったりするのでそのままでは動かないが) 参考にしてもよい．

A　最終レポート: 締め切り 2/8

通常の課題の他に，授業の感想などを書いてください．また，この締め切りは教務への成績報告の関係上，ほぼ**延長不可能**ですので注意してください．(以前のレポートを遅れて提出する人も同様です．) それでも，まにあわん何とかしてくれ，という人は，早めに (遅くとも 1 月末までに) 相談してください．

必修課題: 10.1, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 10.10

オプション課題: 第 10 回の資料の残り全部の問題.