ソフトウェア技法 ミニプロジェクト (2016 年度版)

亀山幸義 (筑波大学情報科学類)

授業の仕上げとして、まとまったプログラムを書いてみよう。OCaml などの関数型言語が最も得意とするのは記号処理であり、たとえば、以下の処理は、非常に書きやすい。

- 数式の処理: 微分、積分、簡単化、変換など
- 論理式の処理: 真理値表の作成、論理式の変形、自動証明など
- プログラムの処理: 構文解析、評価、コンパイル、最適化など
- その他: グラフ、二分木、HTML や XML の文書 (ウェブページ)、自然言語の処理など。

ここでは、必須課題として、四則演算だけからなるプログラムに対する評価器と、命題論理の論理式に対する処理について詳しく解説する。そのあと、いくつかの発展課題について触れる。

1 課題 1: 簡単なインタープリタ (評価器) の作成

OCaml が非常に得意なことの 1 つにプログラム言語の処理系 (インタープリタやコンパイラ) の記述がある。OCaml で OCaml 自身の処理系を書くといったことも可能であるが、ここでは、四則演算だけからなる簡単な「式」の世界に対するインタープリタを作成して、プログラムの処理とはどういうことかを実感してみよう。

1.1 処理対象の式: 逆ポーランド記法

「四則演算だけからなる式」というのは、`1+2*3+4*5` のような式を意味している。これを計算 (評価, evaluate) して、27 を返すプログラムを書こうという趣旨である。このように、式やプログラムを評価して答えを返すプログラムをインタプリタ (評価器、interpreter, evaluator) と呼ぶ。

今回の演習では、式を「逆ポーランド記法」という書き方で入力するものとする。なぜなら、`1+2*3+4*5` といった式は、演算子の優先度をきちんと理解して構文を解析するプログラム (構文解析器) が必要だからである。構文解析も OCaml が得意な分野ではあるが、今回の授業では、構文解析器の作成は時間と意欲のある人むけの課題とすることにして、構文解析の必要がない形式を入力することにした。

結論として、処理対象となる式 (入力とする式) は、「逆ポーランド記法 (Reverse Polish Notation)」で記述された式とする。これは、数式の記述方法の一種であり、「後置記法」とも呼ばれる。

通常の記法の例: `(1+2)*3+4`
逆ポーランド記法の例: `12+3*4+`
(参考) ポーランド記法の例: `+*+1234`

この例でわかるように、逆ポーランド記法は、+ や*といった演算子を、その引数 (オペランド) の後に置く記法である。

逆ポーランド記法の式の意味を説明しよう。通常記法で a+b と書くべきところを、逆ポーランド記法では ab+ と書く、ということであるが、慣れないと間違いやすいので、いくつか例題を出しておく。

逆ポーランド記法での式	通常の記法での式
12345+-*+	1+(2*(3-(4+5)))
1234+-5*+	1+((2-(3+4))*5)
123+-45*+	(1-(2+3))+(4*5)
12+3-4*5+	(((1+2)-3)*4)+5

このような記法は、「かっこ」を使わなくても、構文上の曖昧さがないことがわかっているため、かっこは一切使わない。

本問では、簡単のため、「1 けたの数字」(0 から 9 まで) と、加算と乗算のみを持つ小さな言語を考える。この言語の式を逆ポーランド記法で記述したものを入力として受けつけ、その値を計算するプログラム (評価器) を作ることが目的である。

1.2 処理方法： 1 本のスタックを用いた処理

例として、123456789++++++++ といった式を考えると、9 を入力した瞬間は、そこまでの入力である 1 から 9 までをすべて覚えておく必要があることがわかる。また、これらの数値を使うのは、あとの方の 9 が先で、1 が最後であることもわかる。このような場合の記憶領域としては、スタックを使うと良さそうであるので、ここでもそうしよう。つまり、作るべきものは 1 本のスタックをもつ (小さな) 機械であり、これをスタック機械と呼ぶ。

スタックには、「読みこんだが、まだ計算に使っていないデータ」を積んでおく。また、上記の式で 9 の次の + を処理し終わった瞬間には、8+9 の計算結果である 17 を覚えておく必要があり、これもスタックにいれるとよさそうである。つまり、スタックは、入力されたデータだけでなく計算の途中結果も憶えておく場所となる。

このことを念頭に、123+*4+ という式の処理の過程で、スタックがどう変換すべきかを図示したのが、図 1 である。

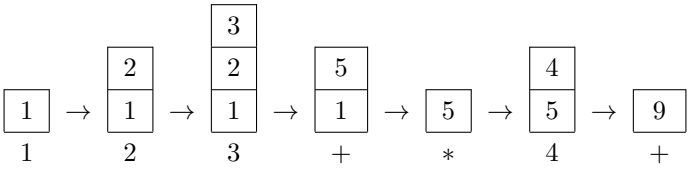

図 1　式 123+*4+ の処理におけるスタックの変化

(1) まず、「123」が入力されると、スタックにこの順番に積まれる。(図において、スタックの下に記載したのは、その時点での入力の文字である。)

(2) 次に「+」を読むと、演算子なので、スタックのトップ (一番最後に積まれたもの) と、トップの次の要素である「3」と「2」を取り出し、足し算を行い、その結果の「5」をスタックに積む。この演算が終わった直後のスタックには「1」と「5」が積まれており、まだ読んでいない式は「*4+」である。

(3) 次の「*」を読むと、上記と同様に「5」と「1」がスタックを取り出し、かけ算を行い、その結果である「5」をスタックに積む。

以下、順次、入力の式を読み込み、数が読まれたらそれをスタックに積み、演算子が読まれたら、スタックからデータを取り出して計算をして結果をスタックに積む、という操作を繰返す。入力の式を読み終わったときにスタックのトップにあるデータが、計算の答である。(ただし、123+ という式のように、演算子の個数が不足している場合は、最終状態でスタックには 2 個以上の数が積まれているので、スタックのトップを返して終わりとするのではなく、エラーとすべきであろう。このあたりは後に議論する。)

1.3 スタック機械の実装

今回の対象言語の処理では、スタックに積む要素は整数だけなので、スタックは「整数リスト (int list 型のデータ)」とするのが良い。リストであれば、「後から追加した要素の方が先に使われる」というスタックの性質を実現するのは容易だからである。

そこで以下の型定義をしておこう。(このような型定義をしてもしなくても、プログラムの動作にはまったく関係ないが、プログラムをわかりやすくする効果がある。)

```
type stack = int list ;;
```

ここで type というキーワードを使ったが、この場合は、ユーザが自分で定義する代数データ型の定義ではなく、単に int list という型に別名をつけただけである。つまり、以下のプログラムにおいて、型 stack と書いてあれば、それをすべて型 int list と置き換えてもよい。

次に、スタックを操作する関数、つまり「データをスタックに追加する push 関数」と、「データをスタックから取り出す pop 関数」を定めよう。

```
let push (n : int) (s : stack) : stack =
   n :: s ;;
let pop  (s : stack) : int * stack =
   match s with
   | [] → failwith "empty stack error in pop"
   | h::s2 → (h,s2)
;;
let is_empty (s : stack) : bool =
   (s = []) ;;
```

関数 push は、スタック s にデータ n を追加して、新しいスタックを返すものである。スタックが整数リストなので、追加操作はリストの cons 操作でよい。

関数 pop は、スタック s のトップからデータを 1 つ取り出す。この場合、取りだしたデータ h だけを返すのではなく、新しいスタック s2 も返しておく必要があるので、h と s2 の組を返すことにした。h::s2 をもらって、(h,s2) を返すというのは、意味がわからない操作に見えるだろうが、そのような理由からである。使い方としては、

```
let (top, newstack) = pop oldstack in
  ... top ... newstack ... ;;
```

という形を想定している。(top がスタックから pop されたデータで、newstack が pop 後の新スタックのつもりである。)

この他に、スタックが空かどうかを判定する関数 is_empty も用意した。これでスタックの操作は完成である。

一方、入力となるデータは、単純な文字列として表現することになる。すなわち、「12+3*4+」という式を入力するときには、"12+3*4+"という string 型のデータとして与えるものとする。

この文字列から文字を取り出す必要がある。OCaml では、文字列を s とすると、その i 番目の文字は、s.[i] という式で得ることができる。ただし、i が 0 からはじまることに注意せよ。つまり、s の 1 文字目は s.[0] である。また、i のところには任意の (整数型の) 式を書けるので、s.[x+37] といった式も書くことができる。もちろん、この x+37 が文字列 s の長さ以上の値であれば、エラーとなる。

このような文字列が与えられたとき、逆ポーランド記法の式を適切に処理して答えを得るというのが、ここでの課題である。スタックは処理の過程でどんどん変化していくので、処理の中心となる関数は、「現在のスタック (処理前のスタック)」と「入力 (文字列)」と「入力の何文字目を処理しているか」の 3 つの情報を引数に取り、「その文字の処理が終わったあとのスタック」を返す関数として実装する。この関数を run という名前にしよう。

```
let rec run (stk : stack) (str : string) (i : int) : stack =
  if i < String.length str then
    match str.[i] with
    | '0' →  let newstk = push 0 stk in run newstk str (i+1)
    | '1' →  ...
    | ...
    | '+' →  (* スタックからデータを2つ pop して、それらの加算結果を push した
                ものを newstk とする。run newstk str (i+1) を呼ぶ。*)
    | '*' →  ...
    | _   →  failwith "illegal input" (* 8/3 に修正 16:02 *)
  else stk ;;
```

入力された文字列を str という引数とし、その i 文字目の処理をする関数として run を実装した。ここでは、入力文字が '0' だっ

た場合の処理しか書いていないが、ほかのケースも同様に書けるであろう。なお、足し算である'+' の処理では、スタックに要素が 2 つ以上含まれていなければ (pop できなければ) エラーとすべきである。

1 つ大事なことは、run の再帰呼び出しは末尾再帰になっている点である。このため、仮に入力の文字列が非常に長くて 1 万文字あったとしても、OCaml システムが、Stack overflow で止まってしまうことはない。

さて、上記の run は、処理の中核だけを書いたものであり、これを使うためには、処理の最初にスタックの初期値等を与える処理と、処理の最後にスタックから「答え」を取り出す処理が必要である。このトップレベル関数 (run を呼び出す関数) を、eval という名前にする。これは evaluator (評価器) の最初の 4 文字である。

```
let eval (str : string) : int =
  let stk = run [] str 0 in
  match pop stk with
  | (top, _) → top
;;
```

eval 関数の仕事は、run 関数を適当な初期値で呼び出すことと、run が返すスタックのトップ要素を「返り値」としていることである。

なお、上記の eval の実装ではさぼっているが、本来ならば、「計算終了後のスタック」は、返す値だけからなる (要素が 1 つだけのスタックである) べきである。つまり、既に書いたように、`"12345+"`のように、式でないもの (演算子が不足しているもの) を入力したときにはエラーとしたい。このようなチェックを上記プログラムに追加するのは、皆さんの課題とする。

演習課題 1-1 (必須) 上記のインタープリタ (run と eval) を完成させなさい。

例: `eval "123+*4+"` = `9`

例: `eval "123+*4++"` = (スタックが途中で不足してしまうのでエラー)

例: `eval "123+*4"` = (スタックに要素が 2 つ以上残ってしまうのでエラー)

演習課題 1-2 (発展) 対象となる言語を拡張して、run/eval も対応させなさい。拡張としては、引き算やべき乗演算をいれる、スタック操作関数として dup(スタックのトップの要素をコピーしてスタックに積む)、swap (スタックのトップと 2 番目の要素をいれかえる) が考えられる。

なお、ここではすべての命令を 1 文字としているので、dup や swap は d や s という 1 文字にするとよい。

演習課題 1-3 (発展) さらに余力があれば、普通の記法の式 (たとえば `1+2*3+4*5`) を逆ポーランド記法に変換するプログラムを書きなさい。これは構文解析をすることになるので、やや難問である。自分で文献等を調べてチャレンジしてほしい。

(参考: プログラム言語の処理系を自作したい人にとって構文解析器を毎回ゼロから作るのは大変であるので、C 言語では lex と yacc というツールがある。OCaml ではそれとほぼ同等の機能をもつ ocamllex と ocamlyacc というツールがあり、自分プログラム言語を実装する場合には、これらのツールを使うのが良い。詳細は、3 年次主専攻実験「関数プログラミング」の資料等を参考にしてほしい。)

2 課題 2: 論理式の処理

命題論理の論理式は、原子論理式 p,q,... を、「かつ (and)」、「または (or)」、「ならば (imp)」、「でない (not)」といった論理記号で結合したものである。ただし、括弧を適宜使ってよいものとする。また、ここでは、簡単のため、3 つ以上の論理式の「かつ」や「または」は考えず、必ず「A かつ B」のように、引数が 2 つであるものとする。

このような論理式は、OCaml の代数データ型で簡単にあらわすことができる。

```
type formula =
  | Atom of string             (* example  Atom("p"), Atom("q123")  *)
  | Not  of formula            (* example  Not(f1) [8/3; コメントの誤植修正 17:23 *)
  | And  of formula * formula  (* example  And(f1, f2)  *)
  | Or   of formula * formula  (* example  Or(f1, f2)  *)
;;
```

ここで、Atom("p") というのは、原子論理式 p のことである。また、Not, And, Or は論理記号のことである。

たとえば、以下のものは、formula 型の要素である。

```
let f1 = And(Not(Atom("p")),Or(Atom("q"),Atom("p"))) ;;
let f2 = Not(And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) ;;
```

なお、この形式は、内部処理には適しているが、人間が入力するのはとても大変である。人間は、たとえば、$\neg p \vee (q \wedge p)$ といった表記を好むが、こういった自由な表記を許すためには、構文解析器が必要になるので、この授業では、そこまではやらない。つまり、皆さんの「論理式処理プログラム」に対する入力は、上記の f1 や f2 といった formula 型のデータとする。

2.1 論理式に対する簡単な処理

与えられた論理式に含まれる原子命題の名前 (文字列"p"など) を集めたリストを返す関数 get_atom を定義しよう。たとえば、

get_atom (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) = ["p"; "q"]

といった答えが返ってくるとよい。

ここで大事なのは、原子命題 p は 2 回含まれているが、出力のリストには 1 回だけ出てくことである。つまり、出力のリストは集合であって欲しい (要素の重複がないリストであってほしい)。集合であればよいので、要素の順番はどうでもよい。なお、2 つの文字列が等しいかどうかの判定は、=を使えばよい。

演習課題 2-1 (必須) このような関数 get_atom を定義せよ。

例. get_atom (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) = ["p"; "q"]

この例では、["q"; "p"] を返しても良い。

ヒント: リストの中に特定の要素が出現するかどうかを調べるには、List.mem という関数を使うとよい。たとえば、List.mem "p" ["q"; "p"; "r"] = true となる

2.2 論理式に対する割当て

次に論理式の真理値を計算する前提として、原子命題に対する「割当て (assignment)」を考える。これは、1 つ 1 つの原子命題に対して true か false を定めたものであり、以下の形のリストで表現することにする。

割当ての例 1: [("p", true); ("q", false)]
割当ての例 2: [("p", true); ("q", true)]

つまり、原子命題の名前 (文字列) と、true/false の値を組にしたもののリストである。このような形にしておくと嬉しい理由は、「原子命題"p"の真理値を計算したい」といときに、List.assoc という関数を使うだけで答えが得られるからである。

```
let _ = List.assoc "p" [("p", true); ("q", false)] ;;
  - : bool = true

let _ = List.assoc "q" [("p", true); ("q", false)] ;;
  - : bool = false
```

一般に、[(キー 1、値 1); (キー 2, 値 2); ...; (キー n, 値 n)] という形のリストを連想リスト (association list) と呼び、よく使うデータ構造である。連想というのは、キーと値の「対応表」という程度の意味である。assoc 関数は、連想リストにキーを投げると対応する値を返すものであり、OCaml の List モジュールが提供してくれている。

ここでは、キーとして原子命題の名前 (文字列)、対応する値として bool 型の値 (真理値) を用いた連想リストを用いる。

2.3 論理式の真理値

論理式の真理値を計算しよう。論理式の真理値は、その論理式に含まれる全ての原子命題に対して true/false という値が決まれば、論理式全体の true/false が決まるというものである。つまり、論理式と割当てが与えられると真理値が定まる。

例 1. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p",true);("q",false)]` = `false`

例 2. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p",false);("q",true)]` = `true`

演習課題 2-2 (必須) このような性質をもつ関数 `eval` を定義せよ。なお、与えられた割当てにおける原子命題が不足しているときは、エラーを出すようにせよ。逆に、割当てにおける原子命題が多過ぎるのは問題ない。

> 例 3. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p", true)]` = `false` またはエラー
> ([2016/8/4 修正; この例では、q の値がわからなくても、論理式の真理値が false であることがわかる例になっており、その場合、エラーでなく false を返してもよいことにした。)
> 例 3b. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p", false)]` = エラー
> ([2016/8/4 追加; この例では、q の値がわからなければ、論理式の真理値がどうなるかわからないので、必ずエラーを返すべきである。)
> 例 4. `eval (And(Not(Atom("p")),Or(Atom("q"),Atom("p")))) [("p", false); ("q", true); ("r", true)]` = `true`

演習課題 2-3 (選択必修課題; これと 3-1 のどちらかが必修) 上記の関数 `eval` を利用して、与えられた論理式の真理値表を作成せよ。

これは、たとえば、`And(Not(Atom("p")),Or(Atom("q"),Atom("p")))` という論理式が与えられると、以下の表を印刷するというものである。

```
p q: target
t t: f
f t: t
t f: f
f f: f
```

この表の 2 行目は、(q,p)=(true,true) の時、与えられた論理式の真理値が false であること、また 3 行目は、(q,p)=(false,true) の時、与えられた論理式の真理値が true であることを表している。

ヒント: 原子命題のリスト atoms をもらって、その原子命題リストに対する「すべての割当てを並べたリスト」を返す関数は、以下のように実装できる。

```
let rec walk atoms aenv =
  match atoms with
    | [] → [aenv]
    | h::t → (walk t ((h,true)::aenv)) @ (walk t ((h,false)::aenv))
;;

let make_aenv (atoms : string list) =
  walk atoms []
;;

let _ = make_aenv ["p"; "q"];;
- : (string * bool) list list =
[[("q", true); ("p", true)]; [("q", false); ("p", true)];
 [("q", true); ("p", false)]; [("q", false); ("p", false)]]
```

演習課題 2-4 (発展課題) 論理式 `And(Not(Atom("p")),Or(Atom("q"),Atom("p")))` を `(~ p) & (q \/ p)` といった形で印刷してくれると、人間にとってわかりやすい。そのような関数 `print_formula` を実装せよ。

2.4 論理式の同値変形

命題論理式を CNF (Conjunctive Normal Form, 論理積標準形) に変形しよう。変形後の論理式は、ある特殊な形をした命題論理式であり、かつ、もとの命題論理式と同値である。

論理積標準形というのは、

$$(L_1 \vee L_2 \vee \cdots) \wedge (L_1' \vee L_2' \vee \cdots) \wedge \cdots$$

の形をした論理式のことである。ただし L_i 等はすべて、原子命題であるか、または、原子命題に `Not` をつけた形の論理式 (リテラルと呼ぶ) である。つまり、論理積標準形というのは、NOT-OR-AND がこの順番に階層化された論理式のことである。この形に変形すると、論理式が証明可能かどうかの判定がとても簡単にできるという利点がある。

CNF への同値変形のためには、以下の変換をおこなえばよい。

- 否定の記号を、一番内側に移動する。このためには、`Not(Not(f))->f`, `Not(And(f1,f2))->Or(Not(f1),Not(f2))`、`Not(Or(f1,f2))->And(Not(f1),Not(f2))` といった変形を繰返し適用し、これ以上変形できなくなったら終了すればよい。
- And を Or の外に出す。このためには、`Or(And(f1,f2),f3)-> And(Or(f1,f3),Or(f2,f3))`、`Or(f3,And(f1,f2))-> And(Or(f3,f1),Or(f3,f2))` といった変形を繰返し適用し、これ以上変形できなくなったら終了すればよい。

ここで、繰返しの終了判定は、「変換によって論理式が変化しなかった」という条件による。なお、OCaml では formula 型のようなデータ構造の要素同士の比較も「=」を使えばできる。(たとえば、`Atomic("p")=Atomic("p")` と書くことができる。)

演習課題 3-1 (選択必修課題; これと 2-3 のどちらかが必修) 上記の 2 つの変換のうち Not を内側にいれる変換を実装せよ。関数名は `to_nnf` とする。(NNF というのは Negation-Normal Form、つまり否定に関する標準形という意味である)。

例: `to_nnf (Not(And(Not(Atom "p"), Or(Atom "q", Atom "p")))) =`
`Or(Atom "p", And(Not (Atom "q"), Not(Atom "p")))`

演習課題 3-2 (発展) 上記の 2 つの変換のうち後半 (否定に関する処理がおわった論理式に対して、それを CNF に変換) を実装せよ。関数名は `to_cnf` とする。

演習課題 3-3 (発展) 与えらえた命題論理式を、CNF に変換することにより、証明可能かどうかを判定せよ。

(たとえば、$(A \vee \neg A) \wedge (\neg B \vee C \vee B)$ は証明可能であり、$(A \vee \neg A) \wedge (\neg B \vee C)$ は証明可能でない。)

3 発展課題: 多倍長演算

整数型 (int 型) や浮動小数点型 (float 型) は、32bit あるいは 64bit といった固定長であるので、大きさあるいは精度に限界がある。しかし、可変長であるリストを使えば、このような固定長データの限界を越えることができる。たとえば Lisp 言語の多くの方言では、bignum というデータ構造 (big number の意味) が用意されており、これは「(メモリが許す限り) いくらでも大きな整数」を表せるデータ構造である。

多倍長計算は、たくさんの固定長データを束ねて、可変長のデータを表す手法である。仮に、0 から 9 までの整数しか持っていないが、リストを使えるプログラム言語があったとする。すると、8141592653 という大きな整数を、

[3; 5; 6; 2; 9; 5; 1; 4; 1; 8]

と表すことができる。(下のけたから並べたのは、その方がプログラミングしやすいという理由である。)

上記の数を、141421356 と加えるには、

[3; 5; 6; 2; 9; 5; 1; 4; 1; 8]
[6; 5; 3; 1; 2; 4; 1; 4; 1]

を、桁ごとに加えて

[9;10; 9; 3;11; 9; 2; 8; 2; 8]

次に、繰り上げ処理をすればよい。

[9; 0; 0; 4; 1; 0; 3; 8; 2; 8]

上記では「1 けた分」を 0 から 9 の整数にしたが、OCaml の整数として表せる数なら (OCaml で加減乗除ができる限りは) 「1 けた分」は、もっと大きな数でよい。これを R とする。x_i が 0〜R-1 の整数だとすると、

[x0; x1; x2; ...; xk]

という整数リストは、$x_0 + x_1 R + x_2 R^2 + \ldots x_k R^k$ という非負の整数をあらわす。ただし、最上位の桁である x_k は 0 でないものとする。(従って、整数の 0 は、多倍長整数として表すと [0] でなく空リスト [] となる。)

演習問題 (発展課題)

- 上記の多倍長整数に対する、加算をおこなう関数 add を定義せよ。
  ただし、「0 から 9 の数字」ではなく、「0 から R-1 までの数」とせよ。(R は、プログラムの最初の方で `let r = power 2 31` などとして与えることにして、プログラム中では、R の具体的な値には依存しないようにせよ。)

```
type bigint = int list ;;

(* bigint 同士の加算 *)
let rec bi_add (b1 : bigint) (b2 : bigint) : bigint =
   ...
;;
```

- 上記の多倍長整数に対する、減算をおこなう関数 sub を定義せよ。sub においては、「最上位の桁が 0 でない」という条件を満たすように作成せよ。なお、0 が最上位にいくつかあらわれる場合、すべて除去する操作が必要である。また、答えが負の数になる場合は、failwith でエラーにせよ。
- 上記の多倍長整数に対する、乗算をおこなう関数を定義せよ。(OCaml の整数は 64bit なので、$R = 2^{31}$ にしておくことにより、0 から R-1 までの数 2 つの積は、OCaml の整数に対する乗算をそのまま使ってよい。)

- 多倍長整数として、N! (N の階乗) の計算を行いなさい。実行時間 10 秒以内で何番目の階乗まで求まるか、限界まで挑戦せよ。(実行時間を測定するため、Unix モジュールの time 関数などが使える。)

4 自分で設定する課題 (発展課題)

余力がある人は、以下の題材を参考に自分で課題を設定して、プログラムを作成して提出しなさい。内容がよければ、加点対象とする。

- 浅井先生の OCaml 入門書では、東京のメトロネットワークにもとづいて、「駅 A から駅 B への最短経路を求める問題」が取りあげられている。アルゴリズムとしては、Dijkstra 法として知られる古典的な課題であるが、OCaml でグラフなどのデータ構造を扱う演習としては、格好の問題である。
- 上記で、命題論理の論理式が証明可能かどうかを判定するため、CNF に変形するアルゴリズムを考えたが、これ以外に Wang のアルゴリズムとよばれる方法で、直接証明をする方法も知られている。アルゴリズム自体は難しくないので、プログラミング能力に自信がある人はやってみてほしい。
- このほか、命題論理の論理式が「充足」可能かどうかを判定する問題を SAT (Satisfiability) と呼び、SAT を高速に解くアルゴリズムとして、DPLL という方法がよく知られている。これは、ネット上に多くの参考文献があるので、それを参考に自分で実装してみると良いだろう。SAT ソルバは、猛烈な開発競争により高速になり、広く利用されるようになっている。
- この他の題材としては、構文解析器 (parser)、文字列照合の KMP 法や BM 法、ラムダ計算のインタープリタの実装などの題材が考えられる。