ソフトウェア技法: 補足 2 (夏の虫取り)

亀山幸義 (筑波大学情報科学類)

OCaml でプログラムを書くことに慣れてくると、ディバッグ (debug、虫取り) の方の時間が長くなってくる。特に、他のプログラム言語における IDE に慣れた人には、今回の演習のように素朴にエディタだけを使ったディバッグは大変かもしれない。

ここでは、OCaml プログラムの非常に基本的なディバッグ方法をいくつか述べる。

1 エラーの種類

OCaml プログラムのエラーの主なものには、以下のものがある。

- 字句エラー
- 構文エラー
- 型エラー
- Warning
- 実行時エラー
- 実行時のスタックオーバーフロー
- 無限ループ
- プログラムの意味的なエラー (意図した結果にならない)

これらは (最後の 2 つを除けば) それぞれ異なるエラーメッセージ (あるいは Warning のメッセージ) が出るので、処理系が出すメッセージをよく見れば、どれであるかわかる。そこで、どの種類であるかわかったとして、対処方法を見てみよう。

1.1 字句エラー

```
#  3 +   1 ;;
Characters 5-6:
   3 +   1 ;;
       ^
Warning 3: deprecated: ISO-Latin1 characters in identifiers
Characters 6-7:
   3 +   1 ;;
        ^
Error: Illegal character (\128)
```

これは Illegal character (不正な文字) とあるので、「文字」のレベルでおかしいということである。実は、上記のプログラムで、3+ の直後には、(日本語の) 全角の空白をいれてしまったので、「そのような文字は知らない」と言われてしまったのである。

このようなメッセージであることさえわかれば、このバグの取りかたは簡単であろう。自分のプログラムに、おかしな文字がはいっていないかを見て (あるいは、再度タイプして) 不正な文字を取りのぞけばよい。

1.2 構文エラー

```
# 3 + (let x = 4) ;;
Characters 14-15:
  3 + (let x = 4) ;;
               ^
Error: Syntax error: operator expected.
#
```

これは Syntax error とあるように、まさに構文エラーである。OCaml の式としての構文になっていないということで、上記の場合は、よくみると、式の中で (トップレベル以外で)「in のない let」を使ってしまっているので、構文があわない。実際エラーメッセージでは、「ここに operator (演算子) があることが期待されている」と書いてあるので、補えばよい。

1.3 型エラー

```
# 3 + "3" ;;
Characters 4-7:
  3 + "3" ;;
      ^^^
Error: This expression has type string but an expression was expected of type
         int
```

型エラーである。これは「型エラー」とは書いていないが、「ここには int 型のデータが来ることが期待されたのに string 型のデータである」と書いてあることから、型が違っているというメッセージであることがわかる。

なお、型エラーは、時として、「本当に間違った場所」ではない場所を指して「間違っている」と言うことがあるので注意したい。たとえば、

```
let rec foo x =
   if x = 0 then 1.0
   else x * (foo (x - 1)) ;;
   Characters 53-66:
     else x * (foo (x - 1)) ;;
              ^^^^^^^^^^^^^
Error: This expression has type float but an expression was expected of type
         int
#
```

というプログラムでは、波線の式は実は int 型で問題なく、そのずっと前に書いてある「1.0」がいけなかった (これを int 型の 1 にすると、プログラム全体の型が整合する) のであるが、OCaml 処理系は、そんな人間の思惑は知らないので、彼の好きな順番に型を推論していき (今の場合は、上の方から順番に推論していき) 型が食い違ったところでエラーをだしてしまう。

これは、OCaml の型システムがまずいのではなく、本質的な問題であり研究論文がいまでも出ている分野である。もし、上記のように「本当のエラーの原因となる場所」が食いちがってしまう問題が気になる人は、是非研究をしてほしい。

とりあえずは、そのような素晴しい研究がでるまでは、「型エラーにおける「エラー原因箇所の指定」は、ずれていることがある」ということを覚えておいてほしい。(型エラーが存在する、という指摘は大いに役立つし、エラー箇所の指定もある程度役に立

つが、ずれていることもある、という話である。)

1.4 Warning

Warning はいくつかあるが、パターンマッチで「網羅的でない」ケースや、変数を定義したのに使っていないケースなどがある。これらはエラーではないので、無視してもよいが、プログラムの意味的なエラーを暗示していることもあるので、有用な情報であることも多い。

```
# match [1;2;3] with
  h ::t → 10 ;;
Characters 0−30:
  match [1;2;3] with
  h ::t → 10...
Warning 8: this pattern−matching is not exhaustive.
Here is an example of a value that is not matched:
[]
− : int = 10
```

網羅的でない (not exhaustive) と明記されているので、メッセージは明快であろう。

1.5 実行時エラー

コンパイルが成功してプログラムが動きだしても、実行時にエラーとなることがある。この場合、OCaml はかなり素気ないエラーメッセージであることが多い。

```
# 1 / 0 ;;
Exception: Division_by_zero.
```

おいおい! プログラムのどの部分がエラーかも教えてくれない。というわけで、実行時エラーの場合は、ディバッグを頑張らないといけない。

そこで、一番素朴な対処方法は、「プログラムに print をいっぱい挿入する」という作戦である。このためには、まだ教えていなかった ; の用法が役に立つ。

```
# print_int 13; print_float 3.14; print_newline (); 10 ;;
133.14
− : int = 10
```

ここでわかるのは、 a; b; c というように、いくつかの式をセミコロン 1 つ (;) で区切って並べると、全体として 1 つの式になることである。また、この式を実行すると、a, b, c の順番に計算して、最後に c の値を返すことである。なお、c の型は何でもよいが、a,b の型は、いずれも unit 型でないと Warning がでる。つまり、a,b のところに「計算結果が意味のある計算」を書くことは期待されていなくて、print 式など、「副作用だけに意味がある」式を書くのがよい。

このようにセミコロンで区切った式を英語で sequence とか sequence expression と言う。

念のために言っておくと、このセミコロンの用法は、リストの要素を区切るセミコロンとはまるで別である。つまり、リストの中で、sequence の意味でのセミコロンをそのまま使うことはできない。かっこで囲えば使うことはできる。

```
# [ 1; (print_int 13; 2); 3];;
13- : int list = [1; 2; 3]
```

このように改行 (newline) を印刷しないと、13 という印刷結果と、「計算結果がはじまることをあらわす - の記号」がくっついてしまって、非常に見にくい。

ともかく、式 e を、(print_int n; e) といった式に変形すると、「この瞬間に n の値を印刷する」ということになるので、ディバッグに使うことができる。

1.6 スタックオーバーフロー

実行時エラーの一種であるが、スタックオーバーフロー (stack overflow) について取りあげよう。

```
# let rec foo x = 1 + (foo x) in
    foo 100;;
Stack overflow during evaluation (looping recursion?).
```

これは無限ループしてしまうプログラムを実行したのであるが、無限に止まらないのではなく、その前に「スタック」があふれた、といって止まってしまったケースである。

これは、再帰関数を間違えて書いて実行したときによくあるケースである。1 回関数呼び出しごとに、一定のデータを新たに記憶する必要があり、それらのデータを置いておく場所は、処理系内部の「スタック」で管理している。(なぜスタックかといえば、関数 f が関数 g を呼びそこから関数 h を呼んだ場合、戻っていくときは h から g にもどり そして f にもどる。つまり、first-in last-out だからである。)

このような「関数呼びだしごとに使われるスタック」の領域は、当然有限なので、関数呼び出しをものすごい回数すると、当然不足してしまう。これが上記のエラーである。

このようなエラーは、ほとんどの場合、再帰関数の書きかたを失敗して、「止まらない計算」にしてしまった場合であるので、自分のプログラムをよく見て、修正するしかない。なお、どの再帰関数が止まらないかわからないときは、前の項でのべた print 式を適度にはさみこむ技法を使うとよい。(が、print 式が 10000 回も実行されると人間はよめないので、うまく工夫しないといけない。たとえば、ループ 1000 回ごとに印刷する、等である。)

ともかく「スタックオーバーフロー」は、関数呼びだしの回数が「処理系がもっている上限」を越えてしまった、という意味であることを覚えておこう。

1.7 無限ループ

前述のように、関数呼びだしが無限に連鎖した場合はスタックオーバーフローになることもあるが、スタックがあふれずに、無限に止まらないこともある。

```
# let rec goo x = goo x in
    goo 100;;
(* 止まらないで、ずっとこの状態のまま *)
```

これは無限ループであるので、ユーザが強制的に止めるしかない。Unix 系 OS なら C-c (control と c を同時に押す) をすると、たいてい止まる。

ところで、1 つ前の項の foo と この項の goo は何が違うのだろうか？ foo は、スタックオーバーフローなのに、なぜ goo はスタックがオーバーフローせずに無限に走りつづけるのだろうか？その答えは、「末尾再帰」ということにあるが、これはこの授業で

後に説明する予定である。

1.8 実行時エラーの解析

OCaml ではこのほかの trace という機能がある。前述のように、関数呼びだしが無限に連鎖した場合はスタックオーバーフローになることもあるが、スタックがあふれずに、無限に止まらないこともある。

```
# let rec foo x = 1 + (foo x) ;;
  val foo : 'a → int = <fun>
# #trace foo ;;
foo is now traced.
# foo 10;;
foo <-- <poly>
foo <-- <poly>
foo <-- <poly>
(* ずっと印刷される *)
```

ここで #trace というのが、関数をトレースするための命令である。これは #use などと同様に、OCaml の式ではなく、OCaml 処理系への命令であるので、#で始まっている。

トレースをすると、その対象となっている関数が呼び出されるごとに、その状況が印刷されるので、ある種のディバッグには便利である。