コンパイラ実験 第 1 回課題

理学部情報科学科 3 年 05-181023 服部桃子

問 1

動作例

test/fib.mlに対するSyntax.print_t, KNormal.print_tの結果はそれぞれ以下のようになる。

Syntax.print_t:

```
LET REC fib (n) =
  IF
    LE
      VAR n
      INT 1
  THEN
    VAR n
  ELSE
    ADD
      VAR fib
        SUB
          VAR n
          INT 1
      VAR fib
        SUB
          VAR n
          INT 2
IN
  VAR print_int
    VAR fib
      INT 30
```

KNormal.print_t:

```
LET REC fib (n) =
  LET Ti3 =
    INT 1
  IN
    IF ( n <= Ti3 ) THEN
      VAR n
    ELSE
      LET Ti6 =
```

```
        LET Ti5 =
          LET Ti4 =
            INT 1
          IN
            SUB n Ti4
        IN
          fib Ti5
      IN
        LET Ti9 =
          LET Ti8 =
            LET Ti7 =
              INT 2
            IN
              SUB n Ti7
          IN
            fib Ti8
        IN
          ADD Ti6 Ti9
IN
  LET Ti2 =
    LET Ti1 =
      INT 30
    IN
      fib Ti1
  IN
    print_int (Ti2)
```

考察

まず、main.mlの関数lexbufを以下のように変更した。

Syntax.print_t, KNormal.print_tが今回実装した、それぞれSyntax.t, KNormal.tを綺麗に出力する関数である。関数lexbuf内でこれらを呼ぶようにしている。

Syntax.tをプリントした後にKNormal.tがプリントされる。

```
let lexbuf outchan l =
  Id.counter := 0;
  Typing.extenv := M.empty;
  Emit.f outchan
    (RegAlloc.f
       (Simm.f
          (Virtual.f
             (Closure.f
                (iter !limit
                   (Alpha.f
                      (* [WEEK1 Q1] add pretty-print for KNormal.t and Syntax.t *)
```

```
                        (let kexp =
                            (KNormal.f
                                (Typing.f
                                    (let exp = (Parser.exp Lexer.token l) in
                                     Syntax.print_t exp; exp))) in
                        print_endline "------------------------";
                        KNormal.print_t kexp; kexp)))))))
```

Syntax.print_tは、syntax.mlに次のように定義した。引数depthは、インデントをどのくらいするかを指定するもので、depthが 1 深くなるごとにインデントが 2 増える。Syntax.tの各ラベルが持つ tuple の最後の要素 (Notの第二要素など) は問 2 のためのもので、ここでは無視できる。

```
let rec string_of_expr ?(do_indent = true) ?(endline = "\n") (exp : t) (depth : int) : string =
  let indent = (String.make (depth * 2) ' ') in
  let prefix = (match do_indent with
      | true -> indent
      | false -> "") in
  match exp with
  | Unit -> prefix ^ "()" ^ endline
  | Bool b -> (match b with
      | true  -> prefix ^ "BOOL TRUE" ^ endline
      | false -> prefix ^ "BOOL FALSE" ^ endline)
  | Int n   -> prefix ^ "INT " ^ (string_of_int n) ^ endline
  | Float f -> prefix ^ "FLOAT " ^ (string_of_float f) ^ endline
  | Not (e, _) -> prefix ^ "NOT\n" ^ (string_of_expr e (depth + 1))
  | Neg (e, _) -> prefix ^ "NEG\n" ^ (string_of_expr e (depth + 1))
  | Add (e1, e2, _)  -> prefix ^ "ADD\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | Sub (e1, e2, _)  -> prefix ^ "SUB\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | FNeg (e, _)      -> prefix ^ "FNEG\n" ^ (string_of_expr e (depth + 1))
  | FAdd (e1, e2, _) -> prefix ^ "FADD\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | FSub (e1, e2, _) -> prefix ^ "FSUB\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | FMul (e1, e2, _) -> prefix ^ "FMUL\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | FDiv (e1, e2, _) -> prefix ^ "FDIV\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | Eq (e1, e2, _)   -> prefix ^ "EQ\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | LE (e1, e2, _)   -> prefix ^ "LE\n" ^ (string_of_expr e1 (depth + 1)) ^ (string_of_expr e2 (depth + 1))
  | If (b, e1, e2, _) -> prefix ^ "IF\n" ^ (string_of_t b (depth + 1)) ^
                         prefix ^ "THEN\n" ^ (string_of_t e1 (depth + 1)) ^
                         prefix ^ "ELSE\n" ^ (string_of_t e2 (depth + 1))
  | Let ((x, _), e1, e2, _) -> prefix ^ "LET " ^ x ^ " =\n" ^ (string_of_expr e1 (depth + 1)) ^ (indent ^ "IN\n") ^ (string_of_expr e2 (depth + 1))
  | Var x -> prefix ^ "VAR " ^ x ^ endline
  | LetRec (f, e, _) -> prefix ^ "LET REC " ^ (string_of_fundef f (depth + 1)) ^ (indent ^ "IN\n") ^ (string_of_expr e (depth + 1))
  | App (e1, e2, _) -> (string_of_expr e1 depth) ^ String.concat "" (List.map (fun e -> string_of_expr e (depth + 1)) e2)
  | Tuple e -> prefix ^ "( " ^
               String.concat ", " (List.map (fun ex -> string_of_t ex (depth + 1) ~do_indent:false ~endline:"") e) ^ " )" ^ endline
  | LetTuple (l, e1, e2, _) -> prefix ^ "LET (" ^ (String.concat ", " (List.map fst l)) ^ ") =\n" ^
                               (string_of_expr e1 (depth + 1)) ^ (indent ^ "IN\n") ^ (string_of_expr e2 (depth + 1))
  | Array (e1, e2, _) -> prefix ^ "[ " ^ (string_of_expr e1 depth ~do_indent:false) ^ (string_of_expr e2 (depth + 1) ~endline:" ]\n")
  | Get (e1, e2, _) -> (string_of_expr e1 depth ~endline:"[ ") ^ (string_of_expr e2 (depth + 1) ~do_indent:false ~endline:" ]") ^ endline
  | Put (e1, e2, e3, _) -> (string_of_expr e1 depth ~endline:"[ ") ^ (string_of_expr e2 (depth + 1) ~do_indent:false ~endline:" ] <-\n") ^
                           (string_of_expr e3 (depth + 1)) ^ endline
and
  string_of_fundef (f : fundef) (depth : int) =
  (fst f.name) ^ " (" ^ (String.concat ", " (List.map fst f.args)) ^ ") =\n" ^ (string_of_expr f.body depth ~endline:"")

let print_expr (exp : t) =
  print_string (string_of_expr exp 0)
```

KNormal.print_tは、kNormal.mlに同様に定義した。

問 2

動作例

```
let x = 1 in
print_int (x + )
```

という入力を与えると、

```
Fatal error: exception Failure("parse error near line 2, characters 15-16")
```

のようなエラーが表示される。

```
let x = 1 in
let y = 4.0 in
print_int (x + y)
```

という入力を与えると、

```
Fatal error: exception Typing.Error("Type error in line 3, from character 11:
↪  unable to unify int and float")
```

のようなエラーが表示される。

考察

パースエラーや型エラーを報告するときに、その位置も報告するようにする課題。
まず、現在の行番号を正しく得るためには、字句解析の際に、改行文字を読むたびにLexing.new_lineを呼ぶ必要がある。そのため、spacesから'\n 'をのぞき、以下のような規則を追加した。

```
rule token = parse
(* 中略 *)
| '\n'  (* [WEEK1 Q2] improve parse/typing error messages *)
    { Lexing.new_line lexbuf; token lexbuf }
```

そして、パーサーでエラーが生じたときに、次のようにParsing.symbol_start_posやParsing.symbol_end_posを利用して位置を得て、エラーメッセージ中で使うようにした。
レコード型Lexing.positionのフィールドpos_lnumが現在の行番号、pos_cnumがファイルの先頭から現在の場所までのバイト数、pos_bolが直前の行までに含まれるバイト数を表すので、現在の位置が行の先頭から何バイト目であるかはpos_cnumとpos_bolの差で表せる。

```
exp:
(* 中略 *)
| error
    { let start_pos = Parsing.symbol_start_pos () in
      let end_pos = Parsing.symbol_end_pos () in
      failwith
        (* [WEEK1 Q2] improve parse-error message *)
        (Printf.sprintf "parse error near line %d, characters %d-%d"
```

```
            start_pos.pos_lnum
            (start_pos.pos_cnum - start_pos.pos_bol)
            (end_pos.pos_cnum - end_pos.pos_bol)) }
```

また、型エラーが生じたときにその位置を取得するためには、次のように、Syntax.tで定義されている演算子のラベルに、それが書かれていたファイル中での位置を持たせるようにすればよい。

```
type pos = Lexing.position

type t =
  | Unit
  | Bool of bool
  | Int of int
  | Float of float
  | Not of t * pos
  | Neg of t * pos
  | Add of t * t * pos
  | Sub of t * t * pos
  | FNeg of t * pos
  | FAdd of t * t * pos
  | FSub of t * t * pos
  | FMul of t * t * pos
  | FDiv of t * t * pos
(* 略 *)
```

そしてこの位置は、次のようにパーサー内で取得できる。以下はparser.mly内のコードである。位置を取得する関数get_pos (unit -> Lexing.position)を定義して使っている。

```
%{
open Syntax
let addtyp x = (x, Type.gentyp ())

(* [WEEK1 Q2] improve error message with reporting the position in the input
   ↪ program *)
let get_pos () = Parsing.symbol_start_pos ()
%}

(* 中略 *)

exp:
| simple_exp
    { $1 }
```

```
14  | NOT exp
15      %prec prec_app
16      { Not($2, get_pos ()) }
17  | MINUS exp
18      %prec prec_unary_minus
19      { match $2 with
20      | Float(f) -> Float(-.f) (* -1.23 won't raise type error *)
21      | e -> Neg(e, get_pos ()) }
22  | exp PLUS exp
23      { Add($1, $3, get_pos ()) }
24  | exp MINUS exp
25      { Sub($1, $3, get_pos ()) }
26  (* 略 *)
```