CS301 Compiler Design and Implementation **Handout # 25**
Prof. Lyn Turbak **January 6, 2004**
Wellesley College

Problem Set 2 Solutions

A PostFix Interpreter

A complete PostFix interpreter is presented in Figs. 1–2. Note the use of pattern matching (e.g., in `exec`) and auxiliary functions (e.g. `arithop`, `relop`) to simplify the structure of the interpreter.

Testing the PostFix Interpreter

The tester for the PostFix interpreter is presented in Fig. 3. The test entries used by the tester are shown in Figs. 4–9. Many students did not include tests for many of the error cases. Note how `makeOpTest` in Fig. 7 abstracts over the generation of error cases tests.

```
module PostFixInterp : sig

    type sval = IntVal of int | StrVal of string | SeqVal of PostFix.com list
    type state = PostFix.com list * sval list
    type ans = IntAns of int | ErrAns of string
    val run : PostFix.pgm -> int list -> ans
    val exec : state -> ans

end = struct

  open PostFix

  type sval =  (* stack values *)
      IntVal of int
    | StrVal of string
    | SeqVal of com list

  type ans = (* answers to executing PostFix programs *)
      IntAns of int    (* integer answer *)
    | ErrAns of string (* error answer *)

  type state = com list * sval list

  let sval2String sv =
    match sv with
      IntVal(i) -> string_of_int i
    | StrVal(s) -> "\"" ^ s ^ "\""
    | SeqVal(_) -> "<seq>"

  let stk2String stk =
     "[" ^ (String.concat ";" (List.map sval2String stk)) ^ "]"

  let boolToInt b = match b with true -> 1 | false -> 0

  let rec updateNth xs n v =
    match xs with
      [] -> raise (Failure ("updateNth"))
    | (x::xs') ->
        if n = 0 then
          (v::xs')
        else x::(updateNth xs' (n - 1) v)

  let arithop aop x y =
    match aop with
      Add -> x + y | Sub -> x - y | Mul -> x * y
    | Div -> x / y | Rem -> x mod y
    | _ -> raise (Failure "can't happen") (* make type checker happy *)

  let relop rop x y =
    match rop with
      LT -> x < y | LE -> x <= y | EQ -> x = y
    | NE -> x != y | GE -> x >= y | GT -> x > y
    | _ -> raise (Failure "can't happen") (* make type checker happy *)
```

Figure 1: PostFix interpreter, part 1.

```
(* run : prog -> int list -> answer *)
  let run pgm args =
     match pgm with
       (Pgm(n,coms)) ->
          let len = List.length(args) in
            if n != len then
              ErrAns("Program expected " ^ (string_of_int n)
                   ^ " arguments but got " ^ (string_of_int len))
            else
              exec (coms, List.rev(List.map (fun x -> IntVal x) args))

  (* exec : state -> ans *)
  (* execute commands until done *)
  and rec exec s =
    match s with
    | ([], []) -> ErrAns("Stack empty at end of program")
    | ([], IntVal(i)::_) -> IntAns(i)
    | ([], v::_) -> ErrAns("Non-int at top of stack when program ends: "
                          ^ (sval2String v))
    | (Int(i)::cs, vs) -> exec(cs, IntVal(i)::vs)
    | (Str(s)::cs, vs) -> exec(cs, StrVal(s)::vs)
    | (Seq(coms)::cs, vs) -> exec(cs, SeqVal(coms)::vs)
    | (Exec::cs, SeqVal(coms)::vs) -> exec(coms@cs, vs)
    | (Pop::cs, _::vs) -> exec(cs, vs)
    | (Swap::cs, v0::v1::vs) -> exec(cs, v1::v0::vs)
    | (Sel::cs, v0::v1::IntVal(0)::vs) -> exec(cs, v0::vs)
    | (Sel::cs, v0::v1::IntVal(i2)::vs) -> exec(cs, v1::vs) (* i3 is non-zero *)
    | (Get::cs, IntVal(i)::vs) ->
        if (0 <= i) && i < List.length(vs) then
          exec(cs, (List.nth vs i)::vs)
        else
          ErrAns("get -- index out of range: "
                 ^ (string_of_int i))
    | (Put::cs, IntVal(i)::v::vs) ->
        if (0 <= i) && i < List.length(vs) then
          exec(cs, (updateNth vs i v))
        else
          ErrAns("put -- index out of range: "
                 ^ (string_of_int i))
    | (Prs::cs, StrVal(s)::vs) -> (StringUtils.unbufferedPrint s; exec(cs, vs))
    | (Pri::cs, IntVal(i)::vs) -> (StringUtils.unbufferedPrint (string_of_int i);
                                   exec(cs,vs))
    | (Div::cs, IntVal(0)::IntVal(i1)::vs) ->
         ErrAns("Division by 0: " ^ (string_of_int i1))
    | (Rem::cs, IntVal(0)::IntVal(i1)::vs) ->
         ErrAns("Remainder by 0: " ^ (string_of_int i1))
    | (((Add | Sub | Mul | Div | Rem) as aop)::cs,
        IntVal(i0)::IntVal(i1)::vs) -> exec(cs, IntVal(arithop aop i1 i0)::vs)
    | (((LT | LE | EQ | NE | GE | GT) as rop)::cs,
        IntVal(i0)::IntVal(i1)::vs) -> exec(cs, IntVal(boolToInt(relop rop i1 i0))::vs)
    | (c::cs, vs) ->
        ErrAns("Invalid stack for " ^ com2String(c) ^ ": " ^ (stk2String vs))

end
```

Figure 2: PostFix interpreter, part 2.

```
module PostFixInterpTest = struct

  open PostFixInterp

  module PostFixTester =
    MakeTester (
      struct
        type prog = string
        type arg = int
        type res = ans

        let trial progString args =
          try run (PostFix.string2Pgm progString) args
          with
           Sexp.IllFormedSexp s ->
             (ErrAns ("\n***ERROR: S-EXPRESSION PARSING ERROR***\n"
              ^ s ^ "\n"))
          | PostFix.SyntaxError s ->
             (ErrAns ("\n***ERROR: POSTFIX PARSING ERROR***\n"
              ^ s ^ "\n"))
          | unhandledException ->
             (StringUtils.unbufferedPrint
                "\n***TESTING TERMINATED BY UNHANDLED EXCEPTION***\n";
              raise unhandledException)

        let arg2String = string_of_int
        let resEqual = (=)

        let res2String ans =
          match ans with
            IntAns(i) -> string_of_int i
          | ErrAns(s) -> s

      end
    )
```

Figure 3: PostFix interpreter tester.

```
let handout10Entries = (* Examples from Handout #10 *)
    [
     ("simple1","(postfix 0  1 2 3)",[([],IntAns(3))]);
     ("simple2", "(postfix 0  1 2 3 pop)", [([],IntAns(2))]);
     ("simple3", "(postfix 0  1 2 swap 3 pop)", [([],IntAns(1))]);
     ("simple4", "(postfix 1 swap)", [([5], ErrAns("Invalid stack for swap: [5]"))]);
     ("simple5", "(postfix 1 pop)",[([17], ErrAns("Stack empty at end of program"))]);

     ("arith1", "(postfix 0 3 4 sub)", [([],IntAns(-1))]);
     ("arith2", "(postfix 0 3 4 add 5 mul 6 sub 7 div)", [([],IntAns(4))]);
     ("arith3", "(postfix 0 3 4 5 6 7 add mul sub swap div)", [([],IntAns(-20))]);
     ("arith4", "(postfix 0 1 20 300 4000 swap pop add)", [([],IntAns(4020))]);
     ("arith5", "(postfix 0 17 4 rem)", [([],IntAns(1))]);
     ("arith6", "(postfix 0 3 4 lt)", [([],IntAns(1))]);
     ("arith7", "(postfix 0 3 4 gt)", [([],IntAns(0))]);
     ("arith8", "(postfix 0 3 4 lt 10 add)", [([],IntAns(11))]);
     ("arith9", "(postfix 1 4 mul add)", [([3],ErrAns("Invalid stack for add: [12]"))]);
     ("arith10", "(postfix 2 4 sub div)", [([3;4],ErrAns("Division by 0: 3"))]);

     ("arg1", "(postfix 1 2 sub)", [([5],IntAns(3))]);
     ("arg2", "(postfix 2 sub)", [([5;4],IntAns(1))]);
     ("arg3", "(postfix 2)", [([5;4],IntAns(4))]);
     ("arg4", "(postfix 2 sub)",
      [([5], ErrAns("Program expected 2 arguments but got 1"))]);
     ("arg5", "(postfix 2 sub)",
      [([5;4;3],ErrAns("Program expected 2 arguments but got 3"))]);
     ("get1", "(postfix 2 0 get)", [([5;6],IntAns(6))]);
     ("get2", "(postfix 2 1 get)", [([5;6],IntAns(5))]);
     ("get3", "(postfix 2 2 get)", [([5;6],ErrAns("get -- index out of range: 2"))]);
     ("get4", "(postfix 2 -1 get)", [([5;6],ErrAns("get -- index out of range: -1"))]);
     ("get5", "(postfix 1 0 get mul)", [([5],IntAns(25))]);
     ("get6", "(postfix 4 Given args a, b, c, x, calculates ax^2 + bx + c.
           0 get get x
           1 get get x again, at new index
           mul push x^2 on stack
           4 get mul multiply x^2 by a
           swap swap ax^2 with x
           3 get mul multiply x by b
           add ax^2 + bx
           add ax^2 + bx + c
           )", [([3;4;5;10],IntAns(345))]);

     ("put", "(postfix 2 Given args a and b, calculate 2a - b
           1 get Push a
           2 mul Replace a by 2a on top of stack
           1 put Replace a by 2a on bottom of stack
           sub Calculate 2a - b
           )", [([5;6],IntAns(4))]);
```

Figure 4: Test entries, part 1.

```
("exec1", "(postfix 1 (2 mul) exec)", [([7],IntAns(14))]);
     ("exec2", "(postfix 0 (0 swap sub) 7 swap exec)", [([],IntAns(-7))]);
     ("exec3", "(postfix 0 (7 swap exec) (0 swap sub) swap exec)", [([],IntAns(-7))]);
     ("exec4", "(postfix 0 (2 mul))",
      [([],ErrAns "Non-int at top of stack when program ends: <seq>")]);
     ("exec5", "(postfix 0 3 (2 mul) gt)",
      [([], ErrAns "Invalid stack for gt: [<seq>;3]")]);
     ("exec6", "(postfix 0 3 exec)", [([],ErrAns "Invalid stack for exec: [3]")]);
     ("exec-nested",
      "(postfix 1 ((2 get swap exec) (3 mul swap exec) swap) (5 sub) swap exec exec)",
      [([0],IntAns(-5));
       ([1],IntAns(-2));
       ([2],IntAns(1));
       ([3],IntAns(4))]);

     ("sel1", "(postfix 1 2 3 sel)", [([1],IntAns(2))]);
     ("sel2", "(postfix 1 2 3 sel)", [([0],IntAns(3))]);
     ("sel3", "(postfix 1 2 3 sel)", [([17],IntAns(2))]);
     ("sel4", "(postfix 4 lt (add) (mul) sel exec)", [([3;4;5;6],IntAns(7))]);
     ("sel5", "(postfix 4 lt (add) (mul) sel exec)", [([3;4;6;5],IntAns(12))]);
```

Figure 5: Test entries, part 2.

```
("fact-iter",
      "(postfix 1
         (factorial loop code
          1 get 0 eq is n = 0?
          () if yes, we're done; ans is on top of stack
          (1 get mul if no, update state vars: ans <- n*ans
          1 get 1 sub 1 put and n <- n-1;
         2 get exec)   then execute loop again
         sel exec)
         swap swap n with factorial loop code
         1 push initial answer = 1
         2 get exec execute loop
         )",
      [([0],IntAns(1));
       ([1],IntAns(1));
       ([5],IntAns(120))]);

     ("fact-iter-print",
      "(postfix 1 (1 get 0 eq
             (\"\n\" prs)
             (\"\n  n=\" prs 1 get pri \"; ans=\" prs 0 get pri
              1 get mul 1 get 1 sub 1 put 2 get exec)
             sel exec)
            swap
            1 2 get exec)",
      [([0],IntAns(1));
       ([1],IntAns(1));
       ([5],IntAns(120))]);

     ("fact-rec",
      "(postfix 1 (factorial recursion code
            1 get 0 eq is n = 0?
            (pop pop 1) if yes, return 1
            (1 get 1 sub push n-1
             1 get 0 get exec call fact recursively
             swap pop         delete fact code
             mul              multiply on way out
            )
            sel exec)
           0 get exec call fact initially
           )",
      [([0],IntAns(1));
       ([1],IntAns(1));
       ([5],IntAns(120))]);

    ("print",
     "(postfix 2
        \"\nAdding \" prs Display ``Adding '' after a newline
        1 get pri Display 1st arg
        \" and \" prs Display `` and ''
        0 get pri Display 2nd arg
        \"\n\" prs  Display newline
        add  Return sum of args
        )",
       [([3;7],IntAns(10))])

    ]
```

Figure 6: Test entries, part 3.

```
let argTestEntries =
    let makeOpTest op =
      [
        (op ^ "Args1", "(postfix 0 " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": []"))]);
        (op ^ "Args2", "(postfix 0 17 " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [17]"))]);
        (op ^ "Args3", "(postfix 0 17 (sub) " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [<seq>;17]"))]);
        (op ^ "Args4", "(postfix 0 (sub) 17 " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [17;<seq>]"))]);
        (op ^ "Args5", "(postfix 0 (sub) (mul) " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [<seq>;<seq>]"))]);
        (op ^ "Args6", "(postfix 0 17 \"foo\" " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [\"foo\";17]"))]);
        (op ^ "Args7", "(postfix 0 \"foo\" 17 " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [17;\"foo\"]"))]);
        (op ^ "Args8", "(postfix 0 \"foo\" \"bar\" " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [\"bar\";\"foo\"]"))]);
        (op ^ "Args9", "(postfix 0 (sub) \"foo\" " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [\"foo\";<seq>]"))]);
        (op ^ "Args10", "(postfix 0 \"foo\" (sub) " ^ op ^")",
         [([],ErrAns("Invalid stack for " ^ op ^ ": [<seq>;\"foo\"]"))])
      ]
      @ (* Special 0 test for div and rem *)
        (if List.mem op ["div"; "rem"] then
          [(op ^ "Args11", "(postfix 0 17 0 " ^ op ^")",
            [([],ErrAns((if op = "div" then "Division" else "Remainder")
                        ^ " by 0: 17"))])]
         else
          [])
```

Figure 7: Test entries, part 4.

```
in

    (List.concat (List.map makeOpTest
                           ["add"; "sub"; "mul"; "div"; "rem";
                            "lt"; "le"; "eq"; "ne"; "ge"; "gt"]))
    @
    [
     ("popArgs", "(postfix 0 pop)", [([],ErrAns("Invalid stack for pop: []"))]);
     ("swapArgs1", "(postfix 0 swap)", [([],ErrAns("Invalid stack for swap: []"))]);
     ("swapArgs2", "(postfix 0 1 swap)", [([],ErrAns("Invalid stack for swap: [1]"))]);
     ("selArgs1", "(postfix 0 sel)", [([],ErrAns("Invalid stack for sel: []"))]);
     ("selArgs2", "(postfix 0 1 sel)", [([],ErrAns("Invalid stack for sel: [1]"))]);
     ("selArgs3", "(postfix 0 1 2 sel)", [([],ErrAns("Invalid stack for sel: [2;1]"))]);
     ("selArgs4", "(postfix 0 (sub) 1 2 sel)",
      [([],ErrAns("Invalid stack for sel: [2;1;<seq>]"))]);
     ("selArgs5", "(postfix 0 \"foo\" 1 2 sel)",
      [([],ErrAns("Invalid stack for sel: [2;1;\"foo\"]"))]);
     ("getArgs1", "(postfix 0 get)", [([],ErrAns("Invalid stack for get: []"))]);
     ("getArgs2", "(postfix 0 17 (sub) get)",
      [([],ErrAns("Invalid stack for get: [<seq>;17]"))]);
     ("getArgs3", "(postfix 0 17 \"foo\" get)",
      [([],ErrAns("Invalid stack for get: [\"foo\";17]"))]);
     ("getArgs4", "(postfix 0 0 get)", [([],ErrAns("get -- index out of range: 0"))]);
     ("getArgs5", "(postfix 0 17 1 get)", [([],ErrAns("get -- index out of range: 1"))]);
     ("getArgs6", "(postfix 0 17 19 2 get)", [([],ErrAns("get -- index out of range: 2"))]);
     ("getArgs7", "(postfix 0 17 -1 get)", [([],ErrAns("get -- index out of range: -1"))]);
     ("putArgs1", "(postfix 0 put)", [([],ErrAns("Invalid stack for put: []"))]);
     ("putArgs2", "(postfix 0 1 put)", [([],ErrAns("Invalid stack for put: [1]"))]);
     ("putArgs3", "(postfix 0 17 23 (sub) put)",
      [([],ErrAns("Invalid stack for put: [<seq>;23;17]"))]);
     ("putArgs4", "(postfix 0 17 23 \"foo\" put)",
      [([],ErrAns("Invalid stack for put: [\"foo\";23;17]"))]);
     ("putArgs5", "(postfix 0 23 0 put)", [([],ErrAns("put -- index out of range: 0"))]);
     ("putArgs6", "(postfix 0 17 23 1 put)", [([],ErrAns("put -- index out of range: 1"))]);
     ("putArgs7", "(postfix 0 17 19 23 2 put)",
      [([],ErrAns("put -- index out of range: 2"))]);
     ("putArgs8", "(postfix 0 17 23 -1 put)", [([],ErrAns("put -- index out of range: -1"))]);
     ("prsArgs1", "(postfix 0 prs)", [([],ErrAns("Invalid stack for prs: []"))]);
     ("prsArgs2", "(postfix 0 17 prs)", [([],ErrAns("Invalid stack for prs: [17]"))]);
     ("prsArgs3", "(postfix 0 (sub) prs)", [([],ErrAns("Invalid stack for prs: [<seq>]"))]);
     ("priArgs1", "(postfix 0 pri)", [([],ErrAns("Invalid stack for pri: []"))]);
     ("priArgs2", "(postfix 0 (sub) pri)", [([],ErrAns("Invalid stack for pri: [<seq>]"))]);
     ("priArgs3", "(postfix 0 \"foo\" pri)",
      [([],ErrAns("Invalid stack for pri: [\"foo\"]"))]);
     ("execArgs1", "(postfix 0 exec)", [([],ErrAns("Invalid stack for exec: []"))]);
     ("execArgs2", "(postfix 0 17 exec)", [([],ErrAns("Invalid stack for exec: [17]"))]);
     ("execArgs3", "(postfix 0 \"foo\" exec)",
      [([],ErrAns("Invalid stack for exec: [\"foo\"]"))])
   ]
```

Figure 8: Test entries, part 5.

```
let otherEntries = (* Other Tests *)
    [
     ("sqr", "(postfix 1 0 get mul)",
      [([5],IntAns(25));
       ([-7], IntAns(49));
       ([3;4], ErrAns("Program expected 1 arguments but got 2"));
       ([], ErrAns("Program expected 1 arguments but got 0"))]);

     ("avg", "(postfix 2 add 2 div)",
      [([5;15],IntAns(10));
       ([5;10],IntAns(7));
       ([6;-8],IntAns(-1))]);

     ("poly", "(postfix 4 0 get 1 get mul 4 get mul swap 3 get mul add add)",
      [([1;2;3;1],IntAns(6));
       ([1;2;3;10],IntAns(123));
       ([3;2;1;1],IntAns(6));
       ([3;2;1;10],IntAns(321))]);

     ("prs", "(postfix 0 \"yes\\no\" prs 0)",
      [([],IntAns(0))])

    ]

  let gcdEntries = (* GCD function *)
    [
     ("gcd",
      "(postfix 2 (0 get 0 eq
                   (pop)
                   (swap 1 get rem 2 get exec)
                   sel exec)
                  2 get swap 2 put swap 2 get exec)",
      [([36;60],IntAns(12));
       ([40;60],IntAns(20));
       ([60;18],IntAns(6));
       ([60;9],IntAns(3));
       ([60;8],IntAns(4));
       ([17;60],IntAns(1))])
    ]

  let entries = handout10Entries @ argTestEntries @ otherEntries @ gcdEntries

  let test () = PostFixTester.testEntries entries

end
```

Figure 9: Test entries, part 6.