Metaprogramming in SML: PostFix and S-expressions

CS251 Programming Languages
Spring 2017, Lyn Turbak

Department of Computer Science
Wellesley College

Recall the Racket PostFix Interpreter

```
;; Contents of postfix-fancy-transform.rkt
(define (postfix-run pgm args) … )
(define (postfix-exec-commands cmds init-stk) … )
(define (postfix-exec-command cmd stk) … )
(define (postfix-program? sexp) … )
(define postfix-arithops … )
(define postfix-relops … )
… many more definitions …

;; Sample program from lecture
(define pf1 '(postfix 2 2 nget 0 gt (sub)
               (swap 1 nget mul add) sel exec))
```

```
> (postfix-run '(postfix 2 1 nget mul swap 1 nget mul add)
               '(3 4))
25

> (map (λ (args) (postfix-run pf1 args)) '((3 5) (3 -5)))
'(2 28)
```

Our Goal is Something Similar in SML

```
- testRun' "(postfix 2 1 nget mul swap 1 nget mul add)" "(3 4)";
val it = "25" : string

- val pf1String = "(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul add)
sel exec)";
val pf1String = "(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul add)
sel exec)"  : string

- map (testRun' pf1String) ["(3 5)", "(3 -5)"];
val it = ["2","28"] : string list
```

Along the way we will see:

- Representing PostFix programs with sum-of-product datatypes
- Leveraging pattern matching in the PostFix interpreter
- Converting between string and sum-of-product representations of a Racket-like S-expression datatype.

PostFix Syntactic Data Types

All PostFix code in these slides is from `~wx/cs251/sml/postfix/PostFix.sml`

```
datatype pgm = PostFix of int * cmd list
and cmd = Pop | Swap | Nget | Sel | Exec
        | Int of int
        | Seq of cmd list
        | Arithop of arithop
        | Relop of relop
and arithop = Add | Sub | Mul | Div | Rem
and relop = Lt | Eq | Gt
```

```
(* SML syntax corresponding to s-expression syntax
   (postfix 2 2 nget 0 gt
              (sub) (swap 1 nget mul add) sel exec)) *)

val pf1 = PostFix(2, [Int 2, Nget, Int 0, Relop Gt,
                      Seq[Arithop Sub],
                      Seq[Swap, Int 1, Nget,
                          Arithop Mul, Arithop Add],
                      Sel, Exec])
```

PostFix Interpreter

```
(* Stack values are either ints or executable seqs *)
datatype stkval = IntVal of int | SeqVal of cmd list

exception ConfigError of string * cmd * stkval list (* config errors *)
exception ExecError of string (* other runtime errors *)

(* val run : pgm -> int list -> int *)
fun run (PostFix(numargs, cmds)) args =
  if numargs = List.length args
  then case execCmds cmds (map IntVal args) of
     (IntVal v) :: _ => v
   | _  => raise ExecError
             "Command sequence on top of final stack"
  else raise ExecError
        "Mismatch between expected and actual”
        ^  "number of args”

(* val execCmds : cmd list -> stkval list -> stkval list *)
and execCmds cmds vs = foldl (fn (cmd,stk) => execCmd cmd stk) vs cmds

(* val execCmd : cmd -> stkval list -> stkval list *)
and execCmd … see the next page …
```

```
(* Perform command on given stack and return resulting stack *)
and execCmd (Int i) vs = (IntVal i) :: vs
  | execCmd (Seq cmds) vs = (SeqVal cmds) :: vs
  | execCmd Pop (v :: vs) = vs
  | execCmd Swap (v1 :: v2 :: vs) = v2 :: v1 :: vs
  | execCmd Nget (stk as (IntVal index) :: vs) =
      if index <= 0 orelse index > List.length(vs)
      then raise ConfigError("Invalid index", Nget, stk)
      else (case List.nth(vs, index-1) of
              (v as IntVal(_)) => v :: vs
            | SeqVal(_) => raise ConfigError("…”, Nget, stk))
  | execCmd Nget ((IntVal index) :: vs) =
     (case List.nth(vs, index-1) of
       (v as IntVal(_)) => v :: vs
     | SeqVal(_) => raise ExecError "Nget can't get a command sequence")
  | execCmd Nget ((IntVal index) :: vs) = List.nth(vs, index-1) :: vs
  | execCmd Sel (v_else :: v_then :: (IntVal v_test) :: vs) =
     (if v_test = 0 then v_else else v_then) :: vs
  | execCmd Exec ((SeqVal cmds) :: vs) = execCmds cmds vs
  | execCmd (Arithop a) ((IntVal i1) :: (IntVal i2) :: vs) =
     (IntVal ((arithopToFun a)(i2, i1)) ) :: vs
  | execCmd (Relop r) ((IntVal i1) :: (IntVal i2) :: vs) =
     (IntVal (boolToInt( ((relopToFun r)(i2, i1)) ) ) ) :: vs
  | execCmd cmd stk = raise ConfigError("Illegal configuration", cmd, stk)

and arithopToFun Add = op+ | arithopToFun Mul = op* | arithopToFun Sub = op-
  | arithopToFun Div = (fn(x,y) => x div y)| arithopToFun Rem = (fn(x,y) => x mod y)

and relopToFun Lt = op< | relopToFun Eq = op= | relopToFun Gt = op>

and boolToInt false = 0 | boolToInt true = 1
```

execCmd

Try it out

```
- run pf1 [3,5];
val it = 2 : int

- run pf1 [3,~5];
val it = 28 : int
```

What About Errors?

```
-  run (PostFix(1,[Arithop Add])) [3]
;uncaught exception ExecError  raised at: postfix.sml:
49.25-49.61

-  run (PostFix(1,[Seq [Arithop Add]])) [3]
;uncaught exception ExecError  raised at: postfix.sml:
33.17-33.59

- run (PostFix(1,[Exec])) [3]
;uncaught exception ExecError  raised at: postfix-
solns.sml:49.25-49.61

- run (PostFix(1,[Int 0, Arithop Div])) [3]
;uncaught exception Div [divide by zero]  raised at:
postfix-solns.sml:57.38-57.41
```

Problems:

1. No error message printed
2. Stops at first error in a sequence of tests

SML Exception Handling with `handle`

```
fun testRun pgm args =
  Int.toString (run pgm args) (* nonerrors & errors must be same type! *)
  handle ExecError msg => "ExecError: " ^ msg
       | General.Div => "Divide by zero error"
        (* General.Div from SML General basis structure;
           Need explicit qualification to distinguish
           from PostFix.Div *)
       | other => "Unknown exception: " ^ (exnMessage other)
```

```
- testRun pf1 [3,~5];
val it = "28" : string (* no error here; returns int as string *)

- testRun (PostFix(1,[Arithop Add])) [3];
val it = "ExecError: Unexpected Configuration" : string
(* could modify program to give configuration details *)

- testRun (PostFix(1,[Seq [Arithop Add]])) [3];
val it = "ExecError: Sequence on top of final stack" : string

- testRun (PostFix(1,[Exec])) [3];
val it = "ExecError: Unexpected Configuration" : string

- testRun (PostFix(1,[Int 0, Arithop Div])) [3];
val it = "Divide by zero error" : string
```

Errors no longer halt execution/testing

```
- map (fn args => testRun (PostFix(2, [Arithop Div])) args)
=     [[3,7], [2,7], [0,5], [4,17]];
val it = ["2","3","Divide by zero error","4"] : string list
```

Exception Handling in other Languages

SML's `raise` & `handle` like

- Java's `throw` and `try/catch`
- JavaScript's `throw` and `try/catch`
- Python's `raise` & `try/except`

No need for `try` in SML; you can attach `handle` to any expression (but might need to add extra parens).

Result types of expression and its handlers must be the same!

S-expression vs SOP program representations

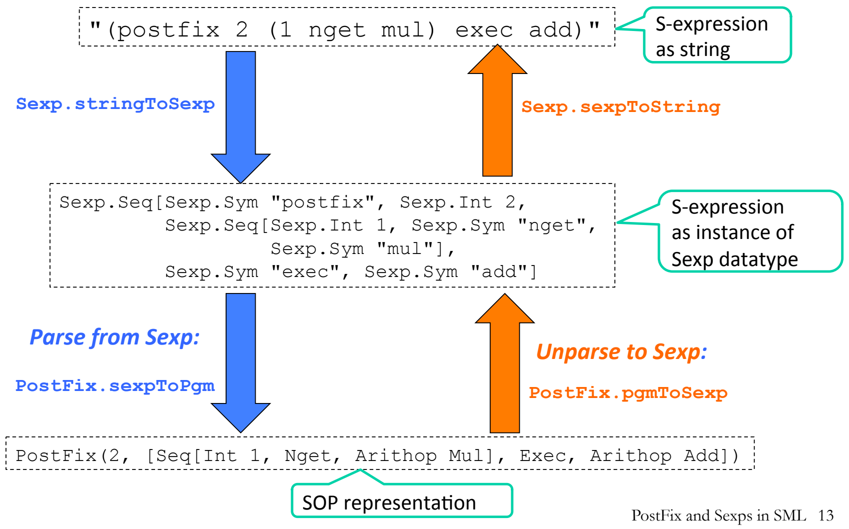
```
'(postfix 2 2 nget 0 gt
            (sub)
            (swap 1 nget mul add)
            sel exec)
```

```
PostFix(2, [Int 2, Nget, Int 0, Relop Gt,
                   Seq[Arithop Sub],
                   Seq[Swap, Int 1, Nget,
                       Arithop Mul, Arithop Add],
                   Sel, Exec])
```

- S-expression notation is more compact
- Sum-of-product notation allows writing program directly as instance(s) of program datatype(s), which supports interpretation based on pattern matching

Can we somehow get the advantages of both?

Idea: convert between SOP and S-expression reps using intermediate Sexp datatype



```
"(postfix 2 (1 nget mul) exec add)"
```

```
Sexp.Seq[Sexp.Sym "postfix", Sexp.Int 2,
         Sexp.Seq[Sexp.Int 1, Sexp.Sym "nget",
                  Sexp.Sym "mul"],
         Sexp.Sym "exec", Sexp.Sym "add"]
```

```
PostFix(2, [Seq[Int 1, Nget, Arithop Mul], Exec, Arithop Add])
```

Sexp datatype

```
signature SEXP = sig
    datatype sexp = Int of int
                  | Flt of real
                  | Str of string
                  | Chr of char
                  | Sym of string
                  | Seq of sexp list

    exception IllFormedSexp of string
    val isEqual : sexp * sexp -> bool
    val stringToSexp : string -> sexp
    val stringToSexps : string -> sexp list
    val fileToSexp : string -> sexp
    val fileToSexps : string -> sexp list
    val sexpToString : sexp -> string
    val sexpToString' : int -> sexp -> string
    val sexpsToString : sexp list -> string
    val sexpToFile : sexp -> string -> unit
    val readSexp : unit -> sexp
end
```

This SEXP signature and Sexp structure can be found in ~wx/cs251/sml/sexp

```
structure Sexp :> SEXP =
   struct … end
```

You can treat the Sexp structure As a black box. You needn't Understand how it works.

Sexp examples

```
- Sexp.stringToSexp "(17 3.141 'c' \"foo bar\" (\"baz quux\" 1.5 42))";
(* Need to escape nested double quotes *)
val it =
  Seq
    [Int 17,Flt 3.141,Chr #"c",Str "foo bar”,
     Seq [Str "baz quux",Flt 1.5,Int 42]] : Sexp.sexp

- Sexp.sexpToString it;
val it = "(17 3.141 'c' \"foo bar\" (\"baz quux\" 1.5 42))" : string

- Sexp.stringToSexps "5 2.7 'Q' \"cs251\" () (1) (2 3)";
val it = [Int 5,Flt 2.7,Chr #"Q",Str "cs251",Seq [],Seq [Int 1],Seq
[Int 2,Int 3]]  : Sexp.sexp list
```

Can read sexps from files:

```
; Contents of pgms.sexp

(postfix 2 1 nget mul) ; simple PostFix program

{ ; Curly braces are nestable block comments
  (postfix 1) ; silly program
  { (intex 0 17) ; Another silly program }
}

(intex 2 (/ (+ ($ 1) ($ 2)) 2)) ; Intex averaging program
```

```
- Sexp.fileToSexps "pgms.sexp";
val it =
  [Seq [Sym "postfix",Int 2,Int 1,Sym "nget",Sym "mul”],
   Seq [Sym "intex",Int 2,
        Seq [Sym "/",
             Seq [Sym "+",Seq [Sym "$",Int 1], Seq [Sym "$",Int 2]],
             Int 2]]]  : Sexp.sexp list
(* The above output has been reformatted to enhanced readability.
   Note that line and block comments are ignored *)
```

Parsing sexps to PostFix.cmd and PostFix.pgm

```
exception SyntaxError of string

fun sexpToPgm (Sexp.Seq(Sexp.Sym "postfix" :: Sexp.Int n :: cmdxs)) =
    PostFix(n, map sexpToCmd cmdxs)
| sexpToPgm sexp = raise (SyntaxError ("invalid PostFix program: "
                                        ^ (Sexp.sexpToString sexp))

and sexpToCmd (Sexp.Int i) = Int i
  | sexpToCmd (Sexp.Seq cmdxs) = Seq (map sexpToCmd cmdxs)
  | sexpToCmd (Sexp.Sym "pop") = Pop
  | sexpToCmd (Sexp.Sym "swap") = Swap
  | sexpToCmd (Sexp.Sym "nget") = Nget
  | sexpToCmd (Sexp.Sym "sel") = Sel
  | sexpToCmd (Sexp.Sym "exec") = Exec
  | sexpToCmd (Sexp.Sym "add") = Arithop Add
  | sexpToCmd (Sexp.Sym "sub") = Arithop Sub
  | sexpToCmd (Sexp.Sym "mul") = Arithop Mul
  | sexpToCmd (Sexp.Sym "div") = Arithop Div
  | sexpToCmd (Sexp.Sym "rem") = Arithop Rem
  | sexpToCmd (Sexp.Sym "lt") = Relop Lt
  | sexpToCmd (Sexp.Sym "eq") = Relop Eq
  | sexpToCmd (Sexp.Sym "gt") = Relop Gt

and stringToCmd s = sexpToCmd (Sexp.stringToSexp s)
and stringToPgm s = sexpToPgm (Sexp.stringToSexp s)
```

PostFix parsing examples

```
- map stringToCmd ["3", "pop", "add", "lt", "(1 nget mul)"];
val it = [Int 3,Pop,Arithop Add,Relop Lt,Seq [Int
1,Nget,Arithop Mul]] : cmd list

- stringToPgm "(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul
  add) sel exec)";
val it =
  PostFix
    (2,
     [Int 2,Nget,Int 0,Relop Gt,Seq [Arithop Sub],
      Seq [Swap,Int 1,Nget,Arithop Mul,Arithop Add],
      Sel,Exec]) : pgm
```

testRun' takes sexp strings

```
exception SexpError of string * Sexp.sexp

fun testRun' pgmSexpString argsSexpString =
  testRun (stringToPgm pgmSexpString)
          (sexpStringToIntList argsSexpString)
  handle SexpError (msg, sexp) => ("SexpError: " ^ msg ^ " "
                                   ^ (Sexp.sexpToString sexp))
  | Sexp.IllFormedSexp msg => ("SexpError: Ill-formed sexp "
                               ^ msg)
  | other => "Unknown exception: " ^ (exnMessage other)

and sexpStringToIntList str =
  let val sexp = Sexp.stringToSexp str
   in case sexp of
         Sexp.Seq xs => map sexpToInt xs
       | _  => raise SexpError("expected sexp sequence but got", sexp)
  end

and sexpToInt (Sexp.Int i) = i
  | sexpToInt sexp = raise SexpError("expected sexp int but got",
                                      sexp)
```

We've achieved our goal from beginning of lecture

```
- testRun' "(postfix 2 1 nget mul swap 1 nget mul add)" "(3 4)";
val it = "25" : string

- val pf1String = "(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul add) sel
exec)";
val pf1String = "(postfix 2 2 nget 0 gt (sub) (swap 1 nget mul add) sel
exec)"  : string

- map (testRun' pf1String) ["(3 5)", "(3 -5)"];
val it = ["2","28"] : string list

(* The following examples illustrate some error cases *)
- testRun' "(postfix 1 1 get mul)" "(3)";
val it = "SyntaxError: unknown command get" : string

- testRun' "(postfix 1 1 nget mul" "(3)"
val it =  "SexpError: Ill-formed sexp Sexp: end of input before matching right
paren -- (postfix 1 1 nget mul"

- testRun' "(postfix nget mul)" "(3)";
val it = "SyntaxError: invalid PostFix program: (postfix nget mul)" : string

- testRun' "(postfix 1 1 nget mul)" "3";
val it = "SexpError: expected sexp sequence but got 3" : string
```

Unparsing PostFix.pgm and PostFix.cmd to sexps

```
fun pgmToSexp (PostFix(n,cmds)) =
  Sexp.Seq (Sexp.Sym "postfix" :: Sexp.Int n :: map cmdToSexp cmds)

and cmdToSexp (Int i) = Sexp.Int i
  | cmdToSexp (Seq cmds) = Sexp.Seq (map cmdToSexp cmds)
  | cmdToSexp Pop = Sexp.Sym "pop"
  | cmdToSexp Swap = Sexp.Sym "swap"
  | cmdToSexp Nget = Sexp.Sym "nget"
  | cmdToSexp Sel = Sexp.Sym "sel"
  | cmdToSexp Exec = Sexp.Sym "exec"
  | cmdToSexp (Arithop Add) = Sexp.Sym "add"
  | cmdToSexp (Arithop Sub) = Sexp.Sym "sub"
  | cmdToSexp (Arithop Mul) = Sexp.Sym "mul"
  | cmdToSexp (Arithop Div) = Sexp.Sym "div"
  | cmdToSexp (Arithop Rem) = Sexp.Sym "rem"
  | cmdToSexp (Relop Lt) = Sexp.Sym "lt"
  | cmdToSexp (Relop Eq) = Sexp.Sym "eq"
  | cmdToSexp (Relop Gt) = Sexp.Sym "gt"

and cmdToString s = Sexp.sexpToString (cmdToSexp s)
and pgmToString s = Sexp.sexpToString (pgmToSexp s)
```

PostFix unparsing example

```
- pgmToString(PostFix(2, [Int 1, Nget, Int 3, Nget, Relop Lt,
=                          Seq[Arithop Sub],
=                          Seq[Swap, Int 1, Nget, Arithop Mul, Swap,
Arithop Add],
=                          Sel, Exec]));

val it =  "(postfix 2 1 nget 3 nget lt (sub) (swap 1 nget mul swap
add) sel exec)"  : string
```