Metaprogramming in SML: PostFix and Intex

CS251 Programming Languages
Spring 2016, Lyn Turbak

Department of Computer Science
Wellesley College

PostFix Syntactic Data Types

```
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
             (mul) (swap 1 nget mul add) sel exec)) *)

val pf1 = PostFix(2, [Int 2, Nget, Int 0, Relop Gt,
                      Seq[Arithop Mul],
                      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 ExecError of string (* runtime errors *)

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

and execCmd … we’ll flesh this out …
```

execCmd: Flesh this out

```
(* 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
  (* Flesh out other cases *)
  | execCmd _ _ = raise ExecError "Unexpected Configuration"

(* Perform all commands on given stack and return resulting stack *)
and execCmds cmds vs = raise ExecError ”Flesh out"

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
```

Try it out

```
- run pf1 [3,5];
val it = 15 : 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)
  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 (PostFix(1,[Arithop Add])) [3];
val it = "ExecError: Unexpected Configuration" : string

- 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`

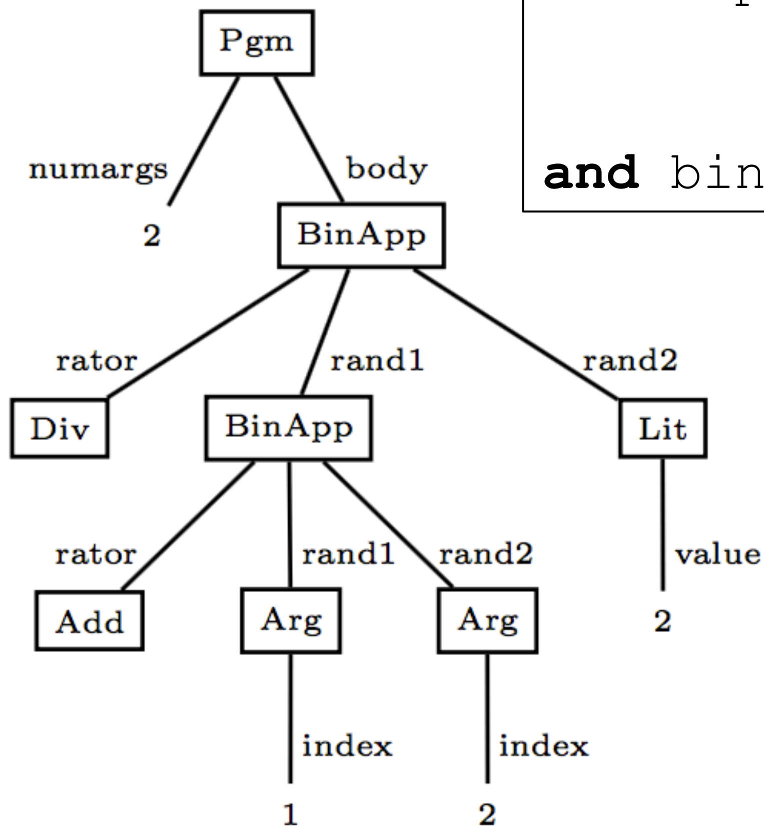
No need for `try` in SML; you can attach `handle` to any expression (but might need to add extra parens.

A New Mini-Language: Intex

Intex programs are simple arithmetic expressions on integers that can refer to integer arguments.

We will extend Intex in a sequence of mini-languages that will culminate in something that is similar to Racket. Each step along the way, we will add features that allow us to study different programming language dimensions.

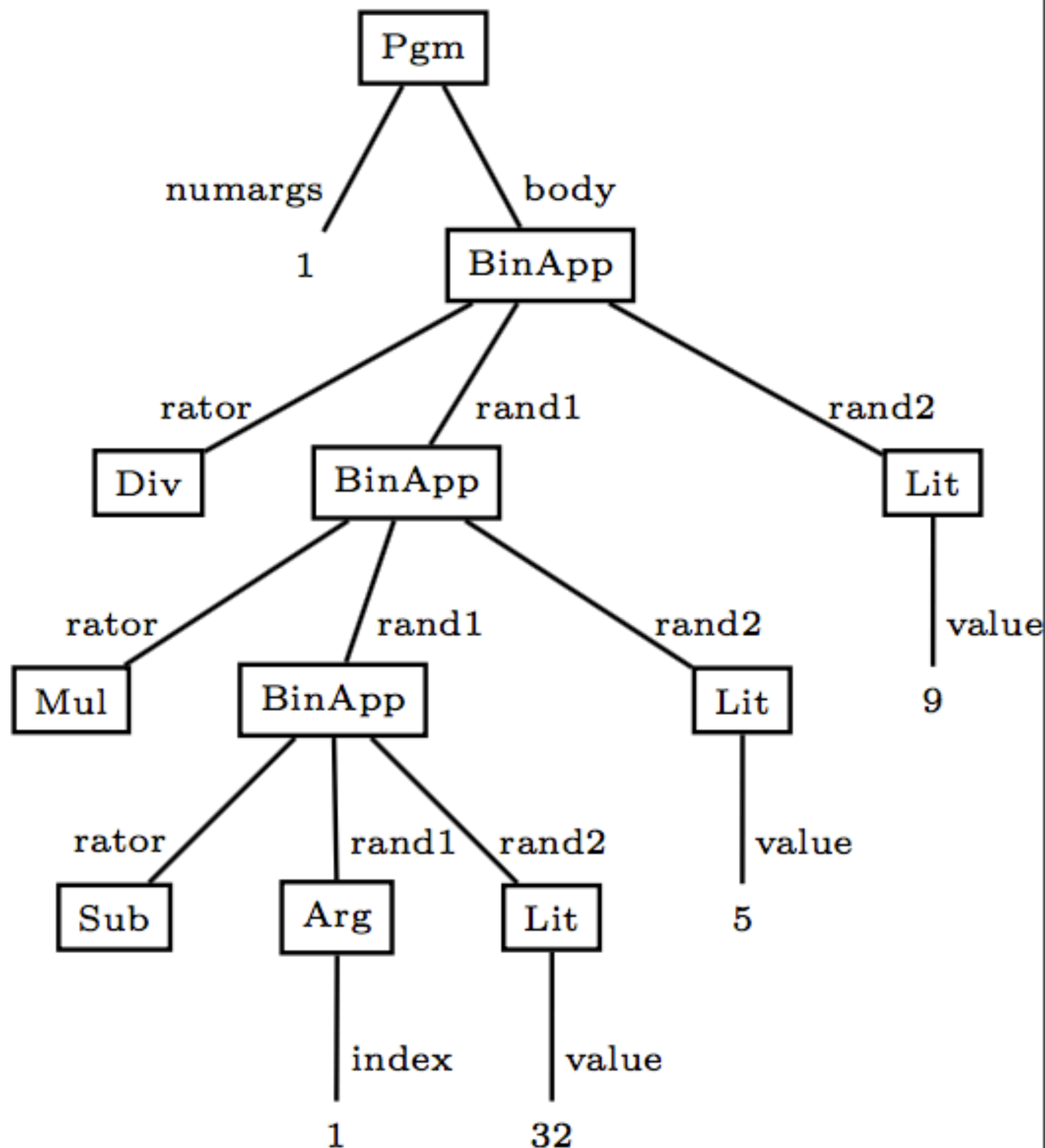
Intex Syntax Trees & Syntactic Data Types




```
datatype pgm = Intex of int * exp
and exp = Int of int
        | Arg of int
        | BinApp of binop * exp * exp
and binop = Add | Sub | Mul | Div | Rem
```

```
val avg = Intex(2, BinApp(Div, BinApp(Add, Arg 1, Arg 2), Int 2))
```

How do we write this Intex program in SML?



Intex Interpreter Without Error Checking

```
fun run (Intex(numargs, exp)) args =
   eval exp args

and eval … flesh this out …

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

Intex Interpreter With Error Checking

```
exception EvalError of string

fun run (Intex(numargs, exp)) args =
  if numargs <> length args
  then raise EvalError
          "Mismatch between expected and actual number of args"
  else eval exp args

and eval (Int i) args = i
  | eval (Arg index) args =
    if (index <= 0) orelse (index > length args)
    then raise EvalError "Arg index out of bounds"
    else List.nth(args, index-1)
  | eval (BinApp(binop, exp1, exp2)) args =
    let val i1 = eval exp1 args
        val i2 = eval exp2 args
    in (case (binop, i2) of
         (Div, 0) => raise EvalError "Division by 0"
       | (Rem,0) => raise EvalError "Remainder by 0"
       | _ => (binopToFun binop)(i1, i2))
    end
```

Dynamic vs. Static Checking: Arg Indices

Dynamic check (at runtime) :

```
| eval (Arg index) args =
   if (index <= 0) orelse (index > length args)
   then raise EvalError "Arg index out of bounds"
   else List.nth(args, index-1)
```

Static check (at compile time or checking time, before runtime) :

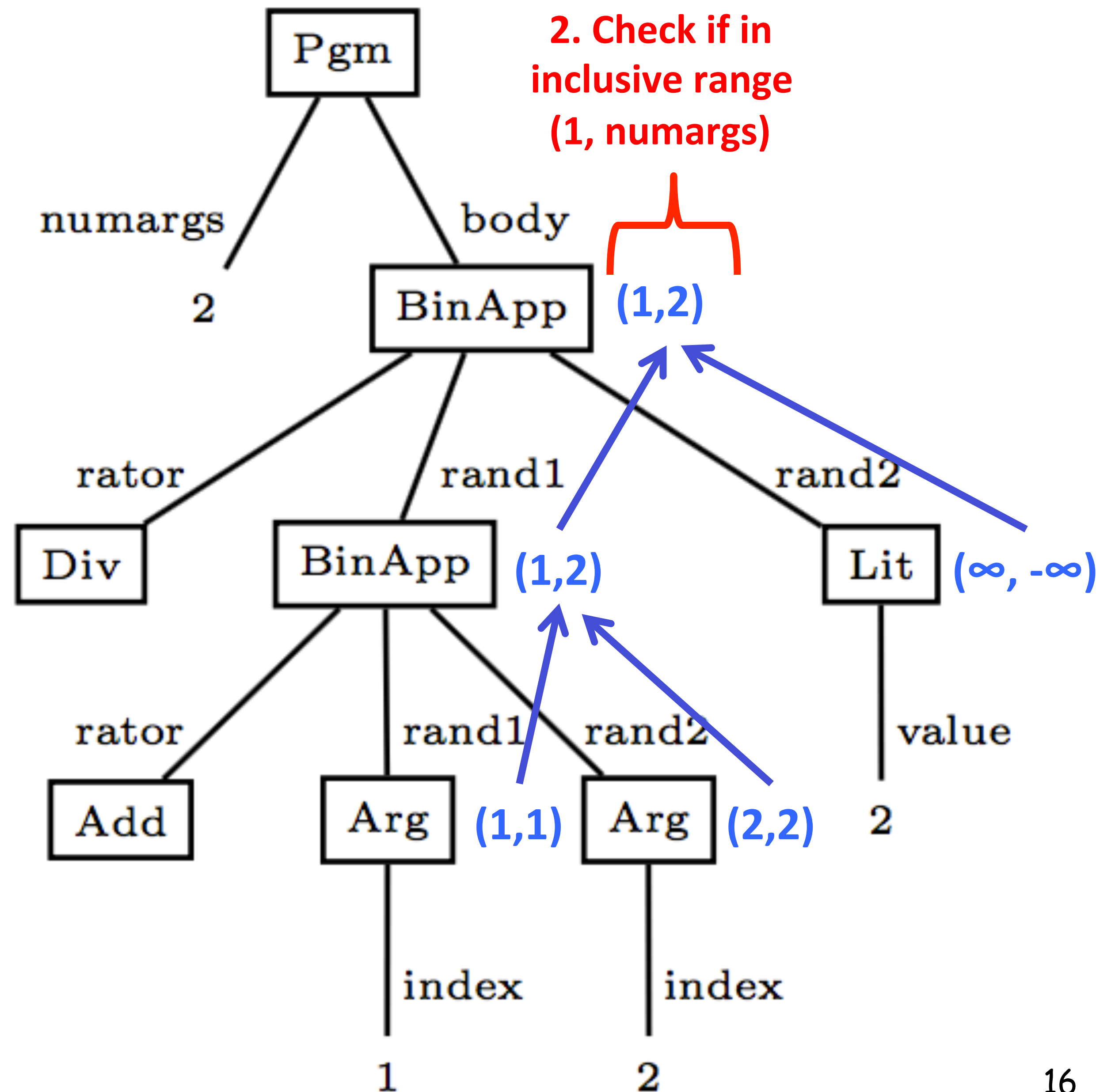
Idea: We know numargs from program, so can use this to check all argument references without running the program.

Such checks are done by examining thee program syntax tree. Often there is a choice between a *bottom-up* and *top-down* approach to processing the tree.

You will do both approaches for Arg index checking in PS6 Problem 5

Static Arg Index Checking: Bottom Up

1. Calculate (min,max) Index value for every Subexpression in tree In bottom-up fashion



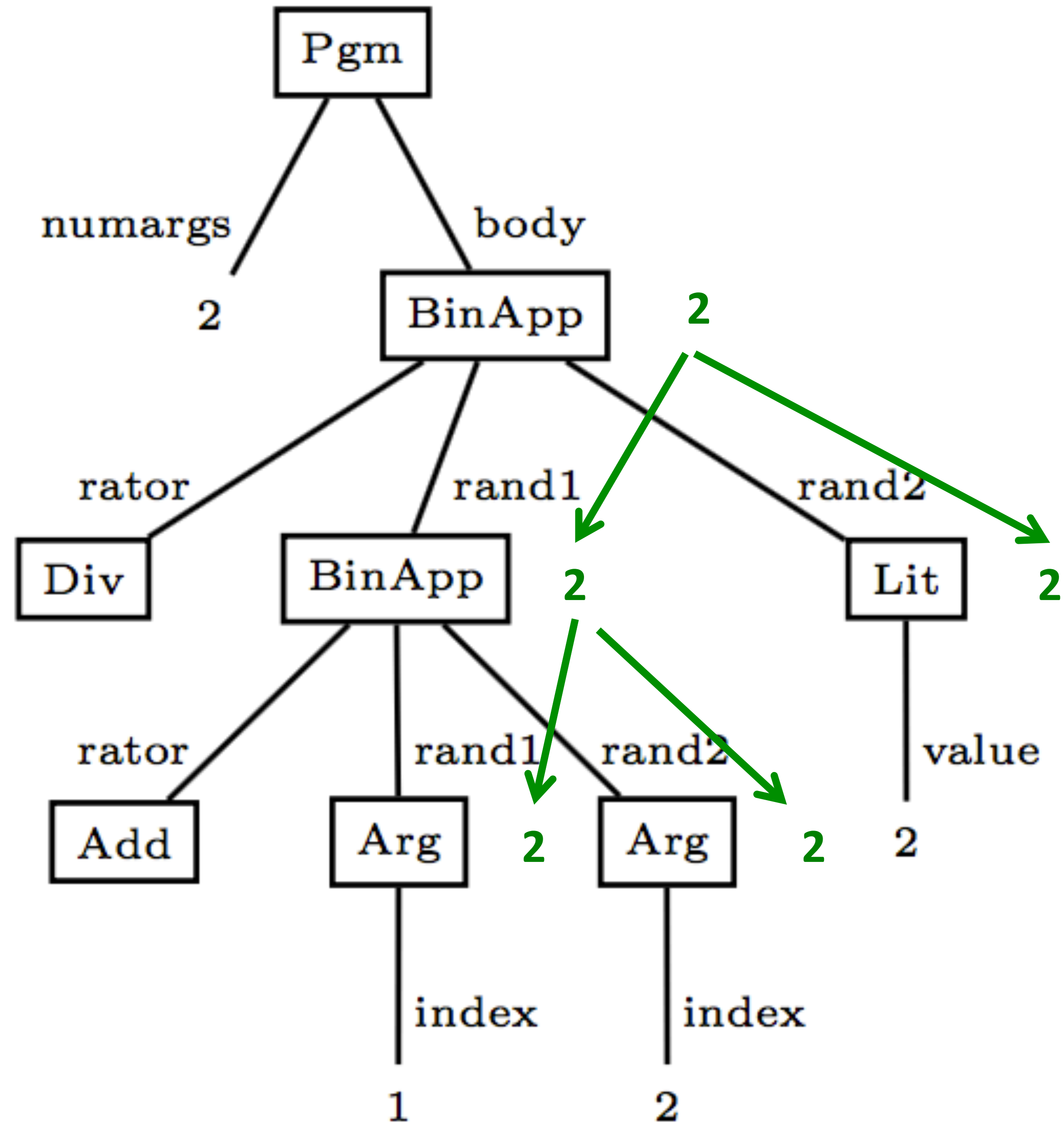
Static Arg Index Checking: Top Down

In top-down phase, pass numargs to every subexpression in program.

Check against every arg Index.

Return true for Arg indices that pass test and subexps Without arg indices

Return false if any Arg index fails test.



Hand-Compiling Intex to PostFix

Manually translate the following Intex programs to
Equivalent PostFix programs

```
val intexP1 = Intex(0, BinApp(Mul,
                        BinApp(Sub, Int 7, Int 4),
                        BinApp(Div, Int 8, Int 2)))

val intexP2 = Intex(4, BinApp(Mul,
                        BinApp(Sub, Arg 1, Arg 2),
                        BinApp(Div, Arg 3, Arg 4)))
```

Reflection: How did you figure out how to translate Intex Arg
indices into PostFix Nget indices?

Automating Intex to PostFix Compilation

```
fun intexToPostFix (Intex.Intex(numargs, exp)) =
  PostFix.PostFix(numargs, expToCmds exp 0)
   (* 0 is a depth argument that statically tracks
      how many values are on stack above the arguments *)

and expToCmds exp depth = … Flesh this out …

and binopToArithop Intex.Add = PostFix.Add
  | binopToArithop Intex.Sub = PostFix.Sub
  | binopToArithop Intex.Mul = PostFix.Mul
  | binopToArithop Intex.Div = PostFix.Div
  | binopToArithop Intex.Rem = PostFix.Rem
```