A New Mini-Language: Intex

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

Intex is the first in a sequence of mini-languages that can be extended to culminate in something that is similar to Racket. At each step along the way, we can add features that allow us to study different programming language dimensions.

This semester, due to lack of time, we will see only the earliest stages of this sequence.

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))
Intex Implementation #1: Intex Interpreter in SML

Given an avg-in-Intex program, how can we execute it?

avg machine (I)
q avg-in-Intex program
q Intex interpreter machine (I)
   ￨ Intex-in-SML-interpreter program
   ￨ SML interpreter machine in wx VM (ignore details)

Intex Interpreter Without Error Checking

(* Intex.pgm -> int list -> int *)
fun run (Intex(numargs, exp)) args =
eval exp args

(* Intex.exp -> int list -> int *)
and eval (Int i) args = i
| eval (Arg index) args = List.nth(args, index-1)
| eval (BinApp(binop, exp1, exp2)) args =
  (binopToFun binop)(eval exp1 args, eval exp2 args)

(* Intex.binop -> int * int -> int *)
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)

Try it out

- run (Intex(1, BinApp(Mul, Arg 1, Arg 1))) [5];
  val it = 25 : int

- run (Intex(1, BinApp(Div, Arg 1, Arg 1))) [5];
  val it = 1 : int

- run (Intex(1, BinApp(Div, Arg 1, Arg 1))) [0];
  uncaught exception EvalError

- run avg [5,15];
  val it = 10 : int

- map (run f2c) [[~40], [0], [32], [98], [212]]
  val it = [~40,~18,0,36,100] : int list
Handling Errors

(* Intex.pgm -> int list -> string *)
fun testRun pgm args =
  Int.toString (run pgm args) (* Convert to string so
  same type as error messages below *)
handle EvalError msg => "EvalError: " ^ msg
| other => "Unknown exception: " ^ (exnMessage other)

val it = "1": string
-
val it = [5];
-
val it = [0];
-
val it = ["~40", "~18", "0", "36", "100"];
-
Running Intex programs as S-expression strings

(* string -> string -> string *)
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)

val test = Intex(2,
  BinApp(Sub,
    BinApp(Mul, Int 32),
    Int 9))
-
val it = ["1", "EvalError: Division by 0", "1"];
-
val it = ["~40", "~18", "0", "36", "100"]
-
What do we know about this program?

val test = Intex(2,
  BinApp(Sub,
    BinApp(Mul, Int 32),
    Int 9))
**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 the 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: 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 subexp Without arg indices

Return false if any Arg index fails test.

---

**Static Arg Index Checking: Bottom Up**

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

2. Check if in inclusive range (1, numargs)

---

**Intex Implementation #2: Intex-to-Postfix-compiler in SML**

Given an avg-in-Intex program, how can we execute it?

- avg machine (I)
  - avg-in-PostFix program
  - avg-in-Intex program
  - Intex-to-PostFix-compiler machine
    - Intex-to-PostFix-compiler-in-SML program
    - SML interpreter machine in wx VM (ignore details)
  - PostFix interpreter machine (I)
    - PostFix-in-SML-interpreter program
    - SML interpreter machine in wx VM (ignore details)
Hand-Compiling Intex to PostFix

Manually translate the following Intex programs to Equivalent PostFix programs

```haskell
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?

Can we automate this process?

Yes! We can define an `intexToPostFix` function with type `Intex.pgm -> PostFix.pgm` and then use it like this:

```haskell
fun translateString intexPgmString = PostFix.pgmToString
                                  (intexToPostFix (Intex.stringToPgm intexPgmString))
```

How to define `intexToPostFix`?

```haskell
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 (Intex.Int i) depth = [PostFix.Int i]
  | expToCmds (Intex.Arg index) depth =
      [PostFix.Int (index + depth), PostFix.Nget]
  (* specified argument is on stack at index + depth *)
  | expToCmds (Intex.BinApp(binop,exp1,exp2)) depth =
    (expToCmds exp1 depth)@ (expToCmds exp2 (depth + 1))
  (* for 2nd rand, add 1 to depth to account for 1st rand *)
  @ [PostFix.Arithop (binopToArithop binop)]
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
```