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.

- **Intex**: integer expressions, positional program arguments
- **Bindex**: Intex + named arguments & local naming
- **Valex**: Bindex + conditionals, multiple kinds of values (booleans, strings, lists), dynamic type checking, and syntactic sugar
- **HOFL** (Racket-like language): Valex + first-class functions
- **HOFLEC**: HOFL + SML-like mutable cells

Intex Syntax Trees & Syntactic Data Types

```plaintext
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))

(* Sample AST as s-expression *)
(intex 2 (/ (+ $1 $2)) 2)

; can even write:
(intex 2 (/ (+ $1 $2)) 2)
```

How do we write this Intex program as SML SOP tree?

As s-expression?
Intex Implementation #1: Intex Interpreter in SML

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

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

Intex Interpreter Without Error Checking: Skeleton

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

(* Intex.exp -> int list -> int *)
and eval (Int i) args = ???
  | eval (Arg index) args = ???
  | eval (BinApp(binop, exp1, 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

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 it = "10" : string

A Read-Eval-Print Loop (REPL) in Intex

- repl();
  intex> (+ 1 2)
    3
  intex> (#args 6 7)
    13
  intex> (* $1 $2)
    42
  intex> (#run (intex 2 (/ (+ ($ 1) ($ 2)) 2)) 5 15)
    10
  intex> (#run "avg.itx" 5 15)
    10
  intex> (#run avg.itx 5 15)
    10
  intex> (#quit)
  Moriturus te saluto!
What do we know about this program?

```plaintext
val test = Intex(2,
    BinApp(Sub,
        BinApp(Mul, Arg 1, Arg 3),
        Arg 2))
```

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 PS9.

---

**Static Arg Index Checking: Top Down (PS9)**

1. In top-down phase, pass numargs to every subexpression in program.
2. Check numargs against every Arg index.
   2a. Return `true` for Arg indices that pass test and subexprs without arg indices
   2b. Return `false` if any Arg index fails test.

**Static Arg Index Checking: Bottom Up (PS9)**

1. Calculate (min,max) index value for every subexpression in tree in bottom-up fashion
2. Check if in inclusive range (1, numargs)
   2. Compare (min,max) value at root of program body expression to (1, numargs), returning true if a subrange and false otherwise
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)

Can we automate this process?

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

```sml
fun intexToPostFix (Intex.Int exp) = PostFix.Int (exp)
| intexToPostFix (Intex.Arg index) = PostFix.Nget
| intexToPostFix (Intex.BinApp (binop, exp1, exp2)) = intexToPostFix exp1 @ (intexToPostFix exp2) @ (binopToArithop binop)
```

Hand-Compiling Intex to PostFix

Manually translate the following Intex programs to equivalent PostFix programs:

- (intex 0 (* (- 7 4) (/ 8 2)))
- (intex 4 (* (- $1 $2) (/ $3 $4)))

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

IntexToPostFix Skeleton

```sml
(* val intexToPostFix: Intex.pgm -> PostFix.pgm *)
fun intexToPostFix (Intex.Int exp) = PostFix.Int (exp)
| intexToPostFix (Intex.Arg index) = PostFix.Nget
| intexToPostFix (Intex.BinApp (binop, exp1, exp2)) = intexToPostFix exp1 @ (intexToPostFix exp2) @ (binopToArithop binop)
```

```sml
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
```