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
- **HOILEC**: HOFL + SML-like explicit mutable cells
- **HOILIC**: HOFL + Racket-like implicit mutable cells

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
```

(* Sample AST as SOP tree *)

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

(* Sample AST as s-expression *)

```
(intex 2 (/ (+ ($ 1) ($ 2)) 2))
```

How do we write this Intex program as SML SOP tree? As s-expression?

Solutions

```sml
(val f2c = (* Fahrenheit to Celsius converter *)

Intex(1, BinApp(Div, BinApp(Mul, BinApp(Sub, Arg 1, (Int 32)), Int 5)))
```

```sml
(* As s-expression *)

(intex 1 (/ (* (- ($ 1) 32) 5) 9))
```
Intex Implementation #1: Intex Interpreter in SML

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

avg machine (l)
 Holt avg-in-Intex program
 Holt Intex interpreter machine (l)
 Holt Holt Intex-interpreter-in-SML program
 Holt Holt SML interpreter machine in csenv/wx VM (ignore details)

Intex Interpreter Without Error Checking: Solution

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

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

(* val binopToFun: 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

```ocaml
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
```

Intex programs as S-expression strings

```ocaml
* Intex programs as S-expression strings

(val testRun: 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

```ocaml
* Running Intex programs as S-expression strings

 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

```ocaml
* A Read-Eval-Print Loop (REPL) in Intex

- repl();
  Intex> (+ 1 2) 3
  Intex> (#args 6 7)
  Intex> (+ ($ 1) ($ 2)) 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!
```

Intex> (+ 1 2)
```
```ocaml
Intex> (+ 1 2)
```
```ocaml
3
```
What do we know about this program?

```
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 subexps without arg indices
2b. Return *false* if any Arg index fails test.
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 program
    - Intex-to-Postfix-compiler-in-SML program
    - SML interpreter machine in csenv/wx VM (ignore details)
- Postfix interpreter machine (I)
  - Postfix-interpreter-in-SML program
  - SML interpreter machine in csenv/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 translateString intexPgmString = PostFix.pgmToString (intexToPostFix (Intex.stringToPgm intexPgmString))
```

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)))

Solutions:

```sml
(postfix 0 7 4 sub 8 2 div mul)
(postfix 4 1 nget $1
  3 nget $2, know that $1 on stack
  sub ; (- $1 $2)
  4 nget $3, know that (- $1 $2) on stack
  6 nget $4, know that $3 and (- $1 $2) on stack
  div ; (/ $3 $4)
  mul ; (* (- $1 $2) (/ $3 $4)))
```

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

```sml
intexToPostFix Solution

(* val intexToPostFix: Intex.pgm -> PostFix.pgm *)
fun intexToPostFix (Intex.Intex(numargs, exp)) = PostFix.PostFix(numargs, expToCmds exp 0)
(* 0 is the initial depth argument *)

(* val expToCmds: Intex.pgm -> PostFix.cmd list *)
(* depth arg statically tracks how many values are on stack above the program arguments *)
and expToCmds (Intex.Int i) depth = [PostFix.Int i] (* specified argument is on stack at index + depth *)
| expToCmds (Intex.Arg index) depth = [PostFix.Nget (index + depth)] (* 1st rand is at same depth as whole binapp *)
| expToCmds (Intex.BinApp (binop,exp1,exp2)) depth = (expToCmds exp1 depth) (* 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
```