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

---

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

As s-expression?

```plaintext
val f2c = (* Fahrenheit to Celsius converter *)
Intex(1,
  BinApp(Div,
    BinApp(Mul,
      BinApp(Sub,
        Arg 1,
        Int 32),
      Int 5),
    Int 9))
```

---

**Interpreting and Compiling Intex**

SOLUTIONS

CS251 Programming Languages
Spring 2018, Lyn Turbak

Department of Computer Science
Wellesley College

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))
```

```
val f2c = (* Fahrenheit to Celsius converter *)
Intex(1,
  BinApp(Div,
    BinApp(Mul,
      BinApp(Sub,
        Arg 1,
        Int 32),
      Int 5),
    Int 9))
```

---

```
val f2c = (* Fahrenheit to Celsius converter *)
Intex(1,
  BinApp(Div,
    BinApp(Mul,
      BinApp(Sub,
        Arg 1,
        Int 32),
      Int 5),
    Int 9))
```

```
val f2c = (* Fahrenheit to Celsius converter *)
Intex(1,
  BinApp(Div,
    BinApp(Mul,
      BinApp(Sub,
        Arg 1,
        Int 32),
      Int 5),
    Int 9))
```
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 With Error Checking

```sml
exception EvalError of string

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

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

val eval = fn (BinApp(binop, exp1, exp2)) =>
  let
    val i1 = eval exp1
    val i2 = eval exp2
  in case (binop, i2) of
    (Div, 0) => raise EvalError
    "Division by 0"
    (Rem, 0) => raise EvalError
    "Remainder by 0"
    _ => (binopToFun binop)(i1, i2)
  end
```

---

Intex Interpreter Without Error Checking: Solution

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

val eval = fn (Int i) args => i
val eval = fn (Arg index) args => List.nth(args, index-1)
val eval = fn (BinApp(binop, exp1, exp2)) args =>
  (binopToFun binop)(eval exp1 args, eval exp2 args)
```

---

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

```ml
(* 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)
```

```ml
testRun (Intex(1, BinApp(Div, Arg 1, Arg 1))) [5];
val it = "1" : string
- testRun (Intex(1, BinApp(Div, Arg 1, Arg 1))) [0];
val it = "EvalError: Division by 0" : string
- map (testRun f2c) [~40, 0, 32, 98, 212];
val it = ["-40", "-18", "0", "36", "100"] : string list
```

Running Intex programs as S-expression strings

```ml
(* 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)
```

```ml
testRun' "(intex 2 (/ (+ ($ 1) ($ 2)) 2))" ((5 15));
val it = "10" : string
- map (testRun' "(intex 1 (/ (* (- ($ 1) 32) 5) 9))")
  = ["(-40)", "(0)", "(32)", "(98)", "(212)"];
val it = ["-40", "-18", "0", "36", "100"] : string list
- map (testRun' "(intex 1 (/ ($ 1) ($ 1)))")
  = ["(-17)", "(0)", "(42)"];
val it = ["1", "EvalError: Division by 0", "1"] : string list
```

Intex programs as S-expression strings

```ml
Intex(1, BinApp(Mul, Arg 1, Arg 1))
"(intex 1 (* ($ 1) ($ 1)))"
```

```ml
Intex(2, BinApp(Div,
    BinApp(Add, Arg 1, Arg 2),
    Int 2))
"(intex 2 (/ (+ ($ 1) ($ 2)) 2))"
```

```ml
Intex(1, BinApp(Div,
    BinApp(Mul, Sub, Arg 1, Int 32),
    Int 9))
"(intex 1 (/ (* (- ($ 1) 32) 5) 9))"
```

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

```ml
- 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?

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.

Static Arg Index Checking: Bottom Up (PS9)

1. Calculate (min,max) index value for every subexpression in tree in bottom-up fashion
2. Compare (min,max) value at root of program body expression to (1, numargs), returning true if a subrange and false otherwise
Hand-Compiling Intex to PostFix
Manually translate the following Intex programs to equivalent PostFix programs:

\[
\begin{align*}
\text{(intex 0 (* (- 7 4) (/ 8 2)))} \\
\text{(intex 4 (* (- $1 $2) (/ $3 $4))})
\end{align*}
\]

Solutions:

\[
\begin{align*}
\text{(postfix 0 7 4 sub 8 2 div mul)} \\
\text{(postfix 4 1 nget ; $1 3 nget ; $2, know that $1 on stack} \\
\text{sub ; (- $1 $2)} 4 nget ; $3, know that (- $1 $2) on stack \\
\text{div ; (/ $3 $4)} mul ; (* (- $1 $2) (/ $3 $4))}
\end{align*}
\]

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

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