INTEX: An Introduction to Program Manipulation

1 Introduction

An **interpreter** is a program written in one language (the **implementation language**) that executes a program written in another language (the **source language**). The source and implementation languages are typically different. For example, we might write an interpreter for OCAML in JAVA. We will call such an interpreter an "OCAML interpreter", naming it after the source language, not the implementation language. It is possible to write an interpreter for a language in that language itself; such an interpreter is known as a **meta-circular interpreter**. For example, Chapter 4 of Abelson and Sussman's *Structure and Interpretation of Computer Programs* presents a meta-circular interpreter for Scheme. Such an interpreter does not by itself define the meaning of a language; it involves boot-strapping issues we considered earlier in our discussions of interpretation and compilation.

One of the best ways to gain insight into programming language design and implementation is to read, write, and modify interpreters and translators. We will spend a significant amount of time in this course studying interpreters and translators. We begin with an interpreter for an extremely simple "mini" or "toy" language, an integer expression language we'll call INTEX. To understand various programming language features, we will add them to INTEX to get more complex languages. Eventually, we will build up to more realistic source languages.

2 Abstract Syntax for INTEX

An INTEX program specifies a function that takes integer arguments and returns an integer result. Abstractly, an INTEX program is a tree constructed out of the following kinds of nodes:

- A **program** node is a node with two components: (1) A non-negative integer *numargs* specifying the number of arguments to the program and (2) A *body* expression.
- An **expression** node is one of the following:
  - A **literal** specifying an integer constant, known as its **value**;
  - An **argument reference** specifying an integer **index** that references a program argument by position;
  - A **binary application** specifying a binary operator (known as the **rator**) and two operand expressions (known as **rand1** and **rand2**).
- A **binary operator** node is one of the following five operators: **addition**, **subtraction**, **multiplication**, **division**, or **remainder**.

These nodes can be be depicted graphically and arranged into trees. For example, Fig. 1 depicts the trees denoting three sample INTEX programs: (1) a program that squares its single input, (2) a program that averages its two inputs, and (3) a program that converts its single input, a temperature measured in Fahrenheit, to a temperature measured in Celsius. Such trees are known as **abstract syntax trees (ASTs)**, because they specify the abstract logical structure of a program without any hint of how the program might be written down in concrete syntax. We leave discussion of the concrete syntax of INTEX programs until later in this handout (see Sec. 4).

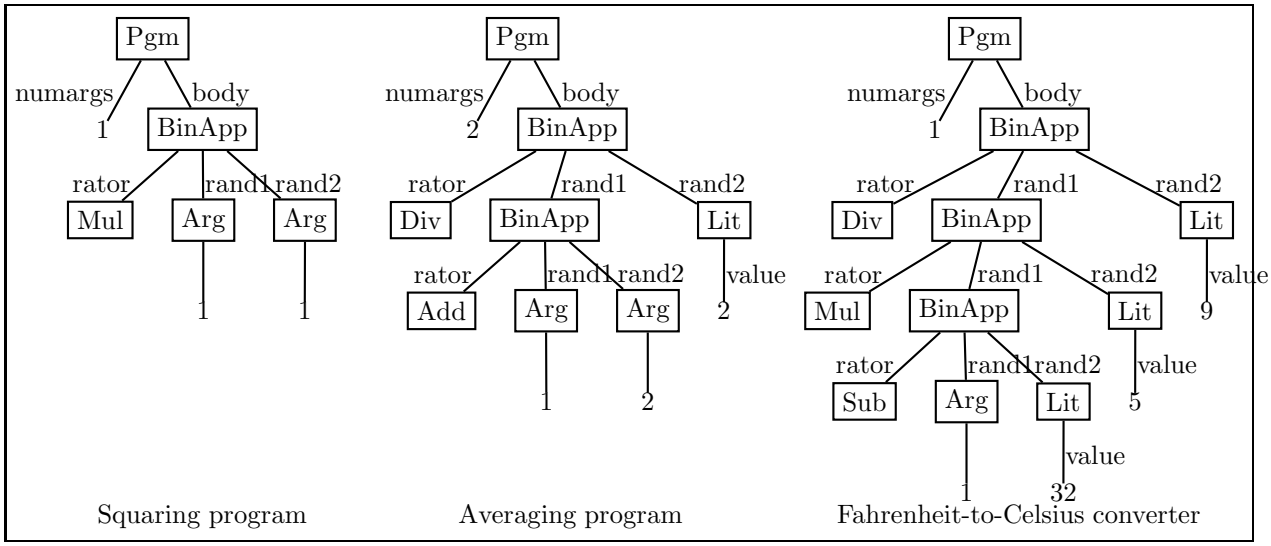


Figure 1: Abstract syntax trees for three sample INTEX programs.

It is easy to express INTEX ASTs using OCAML datatypes. Fig. 2 introduces three types to express the three different kinds of INTEX AST nodes:

1. The `pgm` type has a single constructor, `Pgm`, with two components (numargs and body);
2. The `exp` type is a recursive type with three constructors: `Lit` (for integer literals), `Arg` (for argument references), and `BinApp` (for binary applications).
3. The `binop` type has five constructors, one for each of the five possible operators.

Using these datatypes, the three sample trees depicted in Fig. 1 can be expressed in OCAML as shown in Fig. 3.

3 Manipulating INTEX Programs

INTEX programs and expressions are just trees, and can be easily manipulated as such. Here we study three different programs that manipulate INTEX ASTs.

```
type pgm = Pgm of int * exp (* numargs, body *)

and exp =
    Lit of int (* value *)
  | Arg of int (* index *)
  | BinApp of binop * exp * exp (* rator, rand1, rand2 *)

and binop = Add | Sub | Mul | Div | Rem (* Arithmetic ops *)
```

Figure 2: OCAML datatypes for INTEX abstract syntax.

3.1 Program Size

Define the *size* of an INTEX program as the number of boxed nodes in the graphical depiction of its AST. Then the size of an INTEX program can be determined as follows:

```
let rec sizePgm (Pgm(_,body)) = 1 + (sizeExp body)

and sizeExp e =
  match e with
    Lit i -> 1
  | Arg index -> 1
  | BinApp(_,r1,r2) -> 2 + (sizeExp r1) + (sizeExp r2)
    (* add one for rator and one for BinApp node *)
```

For example:

```
# open Intex;;
# sizePgm sqr;;
- : int = 5
# sizePgm avg;;
- : int = 8
# sizePgm f2c;;
- : int = 11
```

The tree manipulation performed by `sizeExp` is an instance of a general `fold` operator on INTEX expressions that captures the essence of divide, conquer, and glue on these expressions:

```
let rec fold litfun argfun appfun exp =
  match exp with
    Lit i -> litfun i
  | Arg index -> argfun index
  | BinApp(op,rand1,rand2) ->
      appfun op
             (fold litfun argfun appfun rand1)
             (fold litfun argfun appfun rand2)
```

```
let sqr = Pgm(1, BinApp(Mul, Arg 1, Arg 1))

let avg = Pgm(2, BinApp(Div,
                        BinApp(Add, Arg 1, Arg 2),
                        Lit 2))

let f2c = Pgm(1, BinApp(Div,
                        BinApp(Mul,
                               BinApp(Sub, Arg 1, Lit 32),
                               Lit 5),
                        Lit 9))
```

Figure 3: Sample programs expressed using OCAML datatypes.

Using `fold`, we can re-express `sizeExp` as:

```
(* fold-based version *)
let sizeExp e =
  fold (fun _ -> 1) (fun _ -> 1) (fun _ n1 n2 -> 2 + n1 + n2) e
```

3.2 Static Argument Checking

We can statically (i.e., without running the program) check if all the argument indices are valid by a simple tree walk that determines the minimum and maximum argument indices:

```
let rec argCheck (Pgm(n,body)) =
 let (lo,hi) = argRange body
  in (lo >= 1) && (hi <= n)

and argRange e =
  match e with
    Lit i -> (max_int, min_int)
  | Arg index -> (index, index)
  | BinApp(_,r1,r2) ->
      let (lo1, hi1) = argRange r1
      and (lo2, hi2) = argRange r2
       in (min lo1 lo2, max hi1 hi2)
```

For example:

```
# argCheck f2c;;
- : bool = true
# argCheck (Pgm(1,Arg(2)));;
- : bool = false
```

`argRange` can also be expressed in terms of `fold`:

```
(* fold-based version *)
let argRange e =
  fold (fun _ -> (max_int, min_int))
       (fun index -> (index, index))
       (fun _ (lo1,hi1) (lo2,hi2) -> (min lo1 lo2, max hi1 hi2))
       e
```

3.3 Interpretation

An interpreter for INTEX is presented in Fig. 4. It determines the integer value of an INTEX program given a list of integer arguments. For example:

```
# run sqr [5];;
- : int = 25
# run sqr [-7];;
- : int = 49
# run avg [5;15];;
- : int = 10
# run f2c [212];;
- : int = 100
# run f2c [32];;
- : int = 0
# run f2c [98];;
- : int = 36
```

In certain situations, it is necessary to indicate an error; the `EvalError` exception is used for this. For example:

```
# run sqr [2;3];;
Exception: IntexInterp.EvalError "Program expected 1 arguments but got 2".
# run sqr [];;
Exception: IntexInterp.EvalError "Program expected 1 arguments but got 0".
# run (Pgm(1,BinApp(Div, Arg 1, Lit 0))) [5];;
Exception: IntexInterp.EvalError "Division by 0: 5".
```

The `run` function checks that number of supplied arguments matches the number of expected arguments. It then defers to the `eval` function, which is the real workhorse of the interpreter. `eval` defines the meaning of an expression relative to the argument list (`args`) of the program. The argument list is needed to evaluate argument references. Although binary application expressions do not directly refer to the argument list, it must be passed down to both rands in case they contain argument references. Binary applications are evaluated via divide/conquer/glue: both rands are recursively evaluated, and their values are combined by the `primapply` helper method, which defines

```
module IntexInterp = struct

  open Intex

  exception EvalError of string

  let rec run (Pgm(n,body)) ints =
    let len = List.length ints in
      if n = List.length ints then
        eval body ints
      else
        raise (EvalError ("Program expected " ^ (string_of_int n)
                          ^ " arguments but got " ^ (string_of_int len)))

  and eval exp args =
    match exp with
      Lit i -> i
    | Arg index ->
        if (index <= 0) || (index > List.length args) then
          raise (EvalError("Illegal arg index: " ^ (string_of_int index)))
        else
          List.nth args (index - 1)

    | BinApp(op,rand1,rand2) ->
        binApply op (eval rand1 args) (eval rand2 args)

  and binApply binop x y =
    match binop with
      Add -> x + y
    | Sub -> x - y
    | Mul -> x * y
    | Div -> if y = 0 then
               raise (EvalError ("Division by 0: " ^ (string_of_int x)))
             else
               x / y
    | Rem -> if y = 0 then
               raise (EvalError ("Remainder by 0: " ^ (string_of_int x)))
             else
               x mod y

end
```

Figure 4: An interpreter for INTEX.

the binary operators.

Even the `eval` function can be expressed in terms of `fold`!

```
(* fold-based version *)
let eval exp =
  fold (fun i -> (fun args -> i))
       (fun index ->
         (fun args ->
           if (index <= 0) || (index > List.length args) then
             raise (EvalError("Illegal arg index: " ^ (string_of_int index)))
           else
             List.nth args (index - 1)))
       (fun op fun1 fun2 ->
         (fun args ->
            binApply op (fun1 args) (fun2 args)))
       exp
```

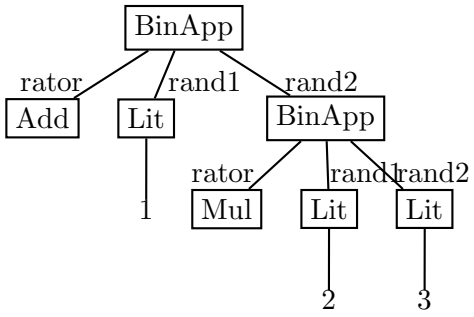
4 A Concrete Syntax for INTEX

4.1 A Plethora of Options

We have seen that abstract syntax trees (ASTs) make it easy to write OCAML programs that manipulate INTEX programs. But it's rather awkward to use the AST functions from Sec. 2 to create INTEX program trees. Moreover, we can't very well expect that INTEX programmers are going to use AST functions written in OCAML to write INTEX programs!

To make programs easier to read and write, we can develop a **concrete syntax** that specifies how program trees can be written as sequences of characters. There are many design choices to make when definining concrete syntax. For example:

- How should argument references be written? Many languages have so-called positional arguments that are specified by a numeric index. For example:
  - the `bash` shell programming language uses `$`i to denote the ith argument (1-indexed) to a shell script.
  - the LATEX formatting language uses `#`i to denote the ith argument (1-indexed) to a macro.
  - The `main()` method in a JAVA class refers to the ith command line argument as an index (0-based) of an array argument.
- How are applications of binary operators written? Suppose we want to represent the following INTEX expression tree:



There are many possible notations. For instance:

- In **infix notation**, the operator is placed between the operands, as in `1 + (2 * 3)`. If we specify that `*` has a higher **precedence** than `+` (as is usually assumed in mathematical notation), we can dispense with the explicit parentheses and write `1 + 2 * 3`.
- In **prefix notation**, the operator is placed before the operands. This is the convention in LISP dialects, in which the example would be written `(+ 1 (* 2 3))`. But there are other variants as well; e.g., `+(1,*(2,3))`.
- In **postfix notation**, the operator is placed after the operands, as in `1 2 3 * +` or `2 3 * 1 +`. Postfix notation is unambiguous and can be written without any explicit parentheses.

- What should the top-level program structure be? How should the number of program arguments be declared? Again, there are many possibilities; here are a few:

  - We could use C and Java's approach of using a distinguished `main()` function to specify the entry point to a program:

    ```
    int main(int a, int b)
    {
        return (a + b) / 2;
    }
    ```

    However, this uses named parameters rather than positional parameters. It would be more in line with INTEX to specify only the number of parameters:

    ```
    int main(int, int)
    {
        return ($1 + $2) / 2;
    }
    ```

    In this case the `main()` name, `int` type annotations, `return` keyword, and semi-colon are "noise" in the sense that they don't have any meaning in INTEX and would be ignored. A syntax that mapped more straightforwardly onto INTEX would omit these:

```
main(2) { ($1 + $2) / 2 }
```

– We could use a notation similar to that used to define mathematical functions:

```
avg[2] = div(+($1,$2),2)
```

Here, the name `avg` has no semantic import (it is ignore by INTEX), but it is a helpful comment.

– We can represent the INTEX tree in s-expression notation, where each tree node is explicitly labelled by its type:

```
(program 2
  (binapp div
          (binapp add
                  (arg 1)
                  (arg 2))
          (lit 2)))
```

Here we have adopted the simple **prefix convention** from Handout #23 for representing sum-of-product trees as s-expressions: the first s-expression in a parenthesized sequence is a token that is the summand node label and the remaining s-expressions are arbitrary s-expressions that represent the product components.

– There are other notations for representing sum-of-product trees. The most popular of these are the XML and XML document description languages. In these languages, summand tags appear in begin/end markups and product components are encoded both in the association lists of markups as well as in components nested within the begin/end markups. For instance, Fig. 5 shows how the averaging program might be encoded in XML. The reader is left to ponder why XML, which at one level is a verbose encoding of s-expressions, is a far more popular standard for expressing structured data than s-expressions.

4.2 Compressed S-expression Format

Because s-expressions are easy to parse, we will adopt an s-expression notation for our concrete INTEX syntax. However, the notation used in the example above is more verbose than we'd like, so we'll use some tricks to make the notation more concise/readable:

- If we assume that numbers stand for themselves, we can avoid the explicit `lit` tag. Thus, we will shorten `(lit 2)` to `2`.
- Since we must be able to distinguish argument references from integer literals, we *cannot* similarly shorten `(arg 1)` to `1`. But we can use a shorter tag name, such as `$`, in which case `(arg 1)` becomes `($ 1)`.[1]

[1] If we're willing to analyze the characters in a symbol name, we could shorten this even more to `$1`.

```
<program>
  <numargs num=2>
  <body>
    <binapp>
      <op name="/">
      <rand1>
        <binapp>
          <op name="+">
          <rand1>
            <arg index=1>
          </rand1>
          <rand2>
            <arg index=2>
          </rand2>
        </binapp>
      </rand1>
      <rand2>
        <lit num=2>
      </rand2>
    </binapp>
  </body>
</program>
```

Figure 5: The averaging program expressed in XML notation.

- The only non-leaf node is a binary application, so we can dispense with the `binapp` tag without introducing ambiguity. Using traditional operator symbols (`+`, `-`, `*`, `/`, `%`) in place of names (`add`, `sub`, `mul`, `div`, `rem`) further shortens the notation. For example, `(binapp add ($ 1) ($ 2))` becomes `(+ ($ 1) ($ 2))`.
- To distinguish INTEX programs from other programs in other mini-languages we will study, we will replace `program` by `intex`. This is not much shorter, but helps to disambiguate programs from different languages.

The result of applying all of the above tricks on our three sample INTEX programs yields:

```
(intex 1 (* ($ 1) ($ 1))) ; Squaring program

(intex 2 (/ (+ ($ 1) ($ 2)) 2)) ; Averaging program

(intex 1 (/ (* (- ($ 1) 32) 5) 9)) ; Temperature conversion program
```

These are significantly shorter that the OCAML AST notation or the unoptimized s-expression notation. This is the s-expression notation that we will adopt for INTEX. We will make similar abbreviations in other languages. It's worth noting that the syntax of LISP dialects is effectively determined by this process — prefix tags are used everywhere except for literals (e.g., numbers, booleans, strings, characters) and for applications (which are written without an explicit `apply` tag, as in `(fact 5)` rather than `(apply fact 5)`).

There are several advantages to using parenthesized prefix s-expression notation for concrete syntax of the mini-languages we will study:

- It is easy to parse s-expressions into trees using the functions in the `Sexp` module (Handout #23). We needn't worry about defining lexical analyzers and parsers for each new language we consider.
- The notation is unambiguous and extremely regular.
- Emacs provides lots of nice features for manipulating s-expressions (such as parethesis matching and automatic indentation).

There are some drawbacks as well:

- The mini-languages will end up looking a lot like a LISP dialect, such as SCHEME. We must be careful to keep in mind that they are very different languages from LISP, even though they look like LISP on the surface.
- The mini-languages will all have a very similar syntax even though their semantics (meaning) may differ significantly. This can make it hard to differentiate between languages.
- For those used to programming in non-LISP langauges like JAVA, C, and OCAML, the parenthesized prefix s-expression form can seem rather unnatural at first. Hopefully they will become natural with practice. (And you will be getting *lots* of practice!)

4.3 Unparsing and Parsing Intex Programs

It is relatively straightforward to convert between Intex ASTs (specified via Ocaml data type constructors), `sexp` trees (also specified via Ocaml data type constructors), and s-expression notation (strings with matching parenthesis). Fig. 6 presents functions that **unparse** an Intex AST into an s-expression. For example:

```
# open Intex;; (* Unparsing functions are in Intex module *)

# let f2c = Pgm(1, BinApp(Div,
                          BinApp(Mul,
                                 BinApp(Sub,Arg(1),Lit(32)),
                                 Lit(5)),
                          Lit(9)));;
        val f2c : Intex.pgm =
  Pgm (1,
   BinApp (Div, BinApp (Mul, BinApp (Sub, Arg 1, Lit 32), Lit 5), Lit 9))

# pgmToSexp f2c;;
- : Sexp.sexp =
Sexp.Seq
 [Sexp.Sym "intex"; Sexp.Int 1;
  Sexp.Seq
   [Sexp.Sym "/";
    Sexp.Seq
     [Sexp.Sym "*";
      Sexp.Seq
       [Sexp.Sym "-"; Sexp.Seq [Sexp.Sym "$"; Sexp.Int 1]; Sexp.Int 32];
      Sexp.Int 5];
    Sexp.Int 9]]

# pgmToString f2c;;
- : string = "(intex 1 (/ (* (- ($ 1) 32) 5) 9))"
```

Fig. 7 presents functions for **parsing** an s-expression into an Intex AST. Let's give them a spin:

```
# open Intex;; (* Parsing functions are in Intex module *)
```

```
let rec pgmToSexp p =
  match p with
    Pgm (n, e) ->
      Seq ([Sym "intex"; Int n; expToSexp e])

and expToSexp e =
  match e with
    Lit i -> Int i
  | Arg i -> Seq [Sym "$"; Int i]
  | BinApp (rator, rand1, rand2) ->
      Seq ([Sym (primopToString rator); expToSexp rand1; expToSexp rand2])

and primopToString p =
  match p with
    Add -> "+"
  | Sub -> "-"
  | Mul -> "*"
  | Div -> "/"
  | Rem -> "%"

and expToString s = sexpToString (expToSexp s)
and pgmToString s = sexpToString (pgmToSexp s)
```

Figure 6: Unparsing INTEX programs to s-expressions.

```
# let f2cSexp = Sexp.stringToSexp "(intex 1 (/ (* (- ($ 1) 32) 5) 9))";;
val f2cSexp : Sexp.sexp =
  Sexp.Seq
   [Sexp.Sym "intex"; Sexp.Int 1;
    Sexp.Seq
     [Sexp.Sym "/";
      Sexp.Seq
       [Sexp.Sym "*";
        Sexp.Seq
         [Sexp.Sym "-"; Sexp.Seq [Sexp.Sym "$"; Sexp.Int 1]; Sexp.Int 32];
        Sexp.Int 5];
      Sexp.Int 9]]

# sexpToPgm f2cSexp;;
- : Intex.pgm =
Pgm (1,
 BinApp (Div, BinApp (Mul, BinApp (Sub, Arg 1, Lit 32), Lit 5), Lit 9))

# stringToPgm "(intex 1 (/ (* (- ($ 1) 32) 5) 9))";;
- : Intex.pgm =
Pgm (1,
 BinApp (Div, BinApp (Mul, BinApp (Sub, Arg 1, Lit 32), Lit 5), Lit 9))
```

```
let rec sexpToPgm sexp =
  match sexp with
    Sexp.Seq [Sexp.Sym("intex");
              Sexp.Int(n);
              body] ->
      Pgm(n, sexpToExp body)
  | _ -> raise (SyntaxError ("invalid Intex program: "
                             ^ (sexpToString sexp)))

and sexpToExp sexp =
  match sexp with
    Int i -> Lit i
  | Seq([Sym "$"; Int i]) -> Arg i
  | Seq([Sym p; rand1; rand2]) ->
      BinApp(stringToPrimop p, sexpToExp rand1, sexpToExp rand2)
  | _ ->  raise (SyntaxError ("invalid Intex expression: "
                              ^ (sexpToString sexp)))

and stringToPrimop s =
  match s with
    "+" -> Add
  | "-" -> Sub
  | "*" -> Mul
  | "/" -> Div
  | "%" -> Rem
  |  _ -> raise (SyntaxError ("invalid Intex primop: " ^ s))

and stringToExp s = sexpToExp (stringToSexp s)
and stringToPgm s = sexpToPgm (stringToSexp s)
```

Figure 7: Parsing s-expressions into INTEX programs.

5 All Together Now: A Read-Eval-Print Loop

We now have all the pieces to create an interactive **read-eval-print loop (REPL)** similar to those used in an OCAML interpreter, a SCHEME interpreter, or a `bash` shell. Fig. 8 presents the the `repl` function, which launches a REPL for INTEX.

```
let repl () =
  let print = StringUtils.print in
  let sexpToint sexp =   (* sexpToint : sexp -> int *)
    match sexp with
      Sexp.Int(i) -> i
    | _ -> raise (Failure "Not an int!") in
  let rec loop args =
    let _ = print "\n\nintex> " in
    let line = read_line () in
      match (Sexp.stringToSexp line) with
        Sexp.Seq [Sexp.Sym "#quit"] -> print "\nMoriturus te saluto!\n"
      | Sexp.Seq ((Sexp.Sym "#args") :: ints) -> loop (List.map sexpToint ints)
      | _ ->
          try
           (print (string_of_int (eval (stringToExp line) args));
            loop args)
           with
            EvalError s -> (print ("Error: " ^ s); loop args)
          | SyntaxError s -> (print ("Error: " ^ s); loop args)
  in loop []
```

Figure 8: A read-eval-print loop (REPL) for INTEX.

At its core, the INTEX REPL:

1. *Reads* an INTEX expression the user types at a prompt;
2. *Evaluates* the INTEX expression;
3. *Prints* the resulting integer value.

For example:

```
# repl();;


intex> (+ 1 2)
3

intex> (* (+ 3 4) (- 5 6))
-7
```

The `try ... with` construct specifies an **exception handler** similer to JAVA's `try ... catch`. In this case, it catches errors without leaving the REPL:

```
intex> (/ 5 0)
Error: Division by 0: 5

intex> (% 7 (- 3 3))
Error: Remainder by 0: 7
```

So far, we have seen how the REPL evaluates literals and binary applications. But how can argument lists be specified? The `#args` directive introduces an argument list that can be used to evaluate the following expressions:

```
intex> (#args 10 20 30)
intex> (+ ($ 1) (* ($ 2) ($ 3)))
610
intex> (#args 5 2 7)
intex> (+ ($ 1) (* ($ 2) ($ 3)))
19
intex> ($ 4)
Error: Illegal arg index: 4
intex> ($ 0)
Error: Illegal arg index: 0
```

Finally, we need a way to exit the interpreter. The `#quit` directive does this:

```
intex> (#quit)

Moriturus te saluto!
- : unit = ()
# (* Now we're back in the OCaml interpreter *)
```