CS301 Compiler Design and Implementation **Handout # 18**
Prof. Lyn Turbak **February 19, 2003**
Wellesley College

INTEX: An Introduction to Program Manipulation

1 Introduction

An *interpreter* is a program written in one language (the *target language* or *implementation language*) that executes a program written another language (*the source language*). The source and target languages are typically different. For example, we might write an interpreter for Java in Scheme. We would call such an interpreter a "Java interpreter", naming it after the source language, not the target language. It is possible to write an interpreter for a language in 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. (Of course, a meta-circular interpreter involves the boot-strapping issues we have considered earlier in our high-level discussions of interpretation and compilation.)

One of the best ways to gain insight into programming language design and implementation is 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 intepreter for an extremely simple "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 any number of 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 its *rator*) and two operand expressions (known as *rand1* and *rand2*).
- An **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 for another day.

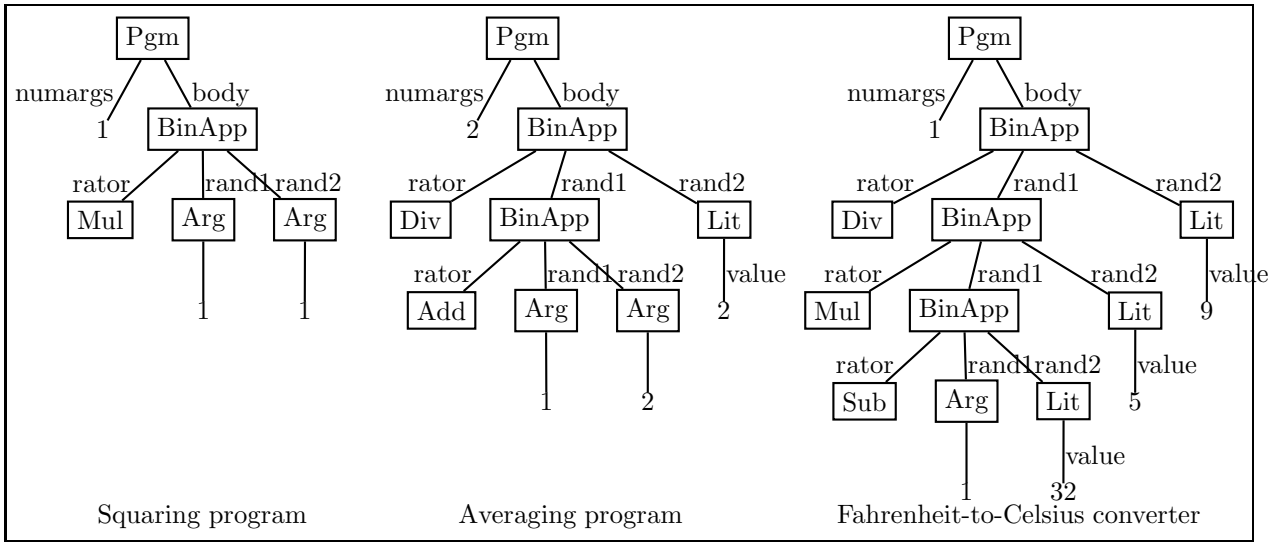

Figure 1: Abstract syntax trees for three sample INTEX programs.

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 possible operator.

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

```
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 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.

```
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.

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 + (expSize 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)
```

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)
```

Again, `argRange` can 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. In certain situations, it is necessary to indicate an error; the `EvalError` exception is used for this.

Even the interpreter can be expressed in terms of `fold`!

```
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)))


  (* direct version *)
  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) ->
        primApply op (eval rand1 args) (eval rand2 args)

  and primApply 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.

```
(* 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 ->
            primApply op (fun1 args) (fun2 args)))
       exp
```

4 S-Expression Form for INTEX Programs

It is easy to represent INTEX programs and expressions in an s-expression format. Fig. 5 shows how to convert between the two.

```
(*************************************************************
   Unparsing to S-Expressions
   *************************************************************)

  let rec pgm2Sexp p =
    match p with
      Pgm (n, e) ->
        Seq ([Sym "intex"; Int n; exp2Sexp e])

  and exp2Sexp e =
    match e with
      Lit i -> Int i
    | Arg i -> Seq [Sym "$"; Int i]
    | BinApp (rator, rand1, rand2) ->
        Seq ([Sym (primop2String rator); exp2Sexp rand1; exp2Sexp rand2])

  and primop2String p =
    match p with
      Add -> "+"
    | Sub -> "-"
    | Mul -> "*"
    | Div -> "/"
    | Rem -> "%"

  and exp2String s = sexp2String (exp2Sexp s)
  and pgm2String s = sexp2String (pgm2Sexp s)

  (*************************************************************
   Parsing from S-Expressions
   *************************************************************)

  let rec sexp2Pgm sexp =
    match sexp with
      Sexp.Seq [Sexp.Sym("intex");
                Sexp.Int(n);
                body] ->
        Pgm(n, sexp2Exp body)
    | _ -> raise (SyntaxError ("invalid Intex program: "
                               ^ (sexp2String sexp)))

  and sexp2Exp sexp =
    match sexp with
      Int i -> Lit i
    | Seq([Sym "$"; Int i]) -> Arg i
    | Seq([Sym p; rand1; rand2]) ->
        BinApp(string2Primop p, sexp2Exp rand1, sexp2Exp rand2)
    | _ ->  raise (SyntaxError ("invalid Intex expression: "
                                ^ (sexp2String sexp)))

  and string2Primop s =
    match s with
      "+" -> Add
    | "-" -> Sub
    | "*" -> Mul
    | "/" -> Div
    | "%" -> Rem
    |  _ -> raise (SyntaxError ("invalid Intex primop: " ^ s))

  and string2Exp s = sexp2Exp (string2Sexp s)
  and string2Pgm s = sexp2Pgm (string2Sexp s)
```