

Exercise sheet 1 for 13 February 2002

2002-02-06

Solve the six first SML exercises, and Exercise 1.8. The solutions to Exercises 1.10, 1.11, 1.12, and 1.16 must be handed in after the exercise classes. If you solve more exercises, you are welcome to hand in those solutions also.

Exercise 1.1 Define an SML function `max2 : int * int -> int` that returns the largest of its two arguments. For instance, `max(99, 3)` should give 99.

Exercise 1.2 Define an SML function `max3 : int * int * int -> int` that returns the largest of its three arguments.

Exercise 1.3 Define a function `isPositive : int list -> bool` so that `isPositive xs` returns true if all elements of `xs` are greater than 0, and false otherwise.

Exercise 1.4 Define a function `isSorted : int list -> bool` so that `isSorted xs` returns true if the elements of `xs` appear sorted in non-decreasing order, and false otherwise. For instance, the list `[11, 12, 12]` is sorted, but `[12, 11, 12]` is not. Note that the empty list `[]` and all one-element lists such as `[23]` are sorted.

Exercise 1.5 Define an SML function `count : inttree -> int` that counts the number of internal nodes (Br constructors) in an `inttree`, where the type `inttree` is defined in the lecture. That is, `count (Br(37, Br(117, Lf, Lf), Br(42, Lf, Lf)))` should give 3, and `count Lf` should give 0.

Exercise 1.6 Define an SML function `depth : inttree -> int` that measures the depth of an `inttree`, that is, the maximal number of internal nodes (Br constructors) on a path from the root to a leaf. For instance, `depth (Br(37, Br(117, Lf, Lf), Br(42, Lf, Lf)))` should give 2, and `depth Lf` should give 0.

Exercise 1.7 Extend the lecture's expression language with the additional operators: `max`, `min`, and `equals`. They all take two arguments. The `equals` operator should return 1 when true, and 0 when false.

Exercise 1.8 Write more example expressions in the expression language (in abstract syntax), and evaluate them using the `eval` function.

Exercise 1.9 Rewrite one of the `eval` functions to evaluate the arguments of a dyadic (two-argument) primitive before branching out on the kind of primitive, in this style:

```
fun eval e env : int =
  case e of
    ...
  | Prim(ope, [e1, e2]) =>
    let val i1 = ...
        val i2 = ...
    in
      case ope of
        "*" => i1 * i2
      | ...
    end
```

Exercise 1.10 Extend the expression language with conditional expressions `If(e1, e2, e3)` corresponding to Java's expression `e1 ? e2 : e3` or SML's expression `if e1 then e2 else e3`.

The datatype (abstract syntax) for expressions may be extended as follows:

```
datatype expr =
  CstI of int
| Var of string
| If of expr * expr * expr
| Prim of string * expr list
```

Extend the interpreter function `eval` correspondingly. It should evaluate `e1`, and if `e1` is non-zero, then evaluate `e2`, else evaluate `e3`.

Note that various strange and non-standard interpretations of the conditional expression is possible. For instance, the interpreter might start by testing whether expressions `e2` and `e3` are syntactically identical, in which case there is no need to evaluate `e1`, only `e2` (or `e3`). Although possible, this is hardly very useful.

Exercise 1.11 Declare an alternative datatype `aexpr` for a representation of arithmetic expressions without let-binding. The datatype should have constructors `CstI`, `Var`, `Add`, `Mul`, `Div`, `Sub`, for constants, variables, addition, multiplication, division, and subtraction.

The idea is that we can represent $x * (y + 3)$ as `Mul(Var "x", Add(Var "y", CstI 3))` instead of `Prim("**", [Var "x", Prim("+", [Var "y", CstI 3])])`.

Write the representation of the expressions $v - (w + z)$ and $2 / (3 + x * y)$. How should $x + y + z + v$ be represented?

Exercise 1.12 Write an SML function `fmt : aexpr -> string` to format expressions as SML strings. For instance, it may format `Sub(Var "x", CstI 34)` as the SML string `"(x - 34)"`.

Exercise 1.13 Write an SML function `simplify : aexpr -> aexpr` to perform expression simplification. For instance, it should simplify $(1 + 0) * (x + 0)$ to x .

Exercise 1.14 (Only for people with fond recollections of differential calculus). Write an SML function to perform symbolic differentiation of simple arithmetic expressions (such as those above) with respect to a single variable.

Exercise 1.15 Write a version of the formatting function `fmt` which avoids producing excess parentheses. For instance,

```
Mul(Sub(Var "a", Var "b"), Var "c")
```

should be formatted as `"(a-b)*c"`, whereas

```
Sub(Mul(Var "a", Var "b"), Var "c")
```

should be formatted as `"a*b-c"`. Also, it should be taken into account that operators associate to the left, so that

```
Sub(Sub(Var "a", Var "b"), Var "c")
```

is formatted as `"a-b-c"` whereas

```
Sub(Var "a", Sub(Var "b", Var "c"))
```

is formatted as `"a-(b-c)"`.

This can be achieved by declaring the formatting function to take an extra parameter `ctxpre` that indicates the precedence or binding strength of the context. The formatting function then has type `fmt : int * expr -> string`.

Higher precedence means stronger binding. When the top-most operator of an expression to be formatted has higher precedence than the context, there is no need for parentheses around the expression. A left associative operator of precedence 6 (in SML), such as minus (`-`), provides context precedence 5 to its left argument, and context precedence 6 to its right argument. As a consequence, `-(a, -(b, c))` will be parenthesized `a - (b - c)` but `-(-(a, b), c)` will be parenthesized `a - b - c`.

Exercise 1.16 The purpose of this exercise is to use Java classes and methods to do what we have done using the SML datatype `aexpr` from Exercise 1.11 above. Design a Java class hierarchy to represent arithmetic expressions: it could have an abstract class `Expr` with subclasse `CstI`, `Var`, and `Binop`, where the latter is itself abstract and has concrete subclasses `Add`, `Mul`, `Sub`, and `Div`. All classes should implement the `toString()` method to format an abstract syntax tree as a Java `String`. The class declarations may look like this:

```
abstract class Expr {
    abstract public String toString();
}

class CstI extends Expr {
    private int i;

    public CstI(int i) {
        this.i = i;
    }

    public String toString() { ... }
}

class Var extends Expr {
    private String name;

    public Var(String name) {
        this.name = name;
    }

    public String toString() { ... }
}

abstract class Binop extends Expr {
    private String oper;
    private Expr e1, e2;

    public Binop(String oper, Expr e1, Expr e2) {
        this.oper = oper; this.e1 = e1; this.e2 = e2;
    }

    public String toString() { ... }
}

class Add extends Binop { ... }
class Mul extends Binop { ... }
class Sub extends Binop { ... }
class Div extends Binop { ... }
```

The classes may be used like this:

```
Expr e = new Add(new CstI(17), new Var("z"));
System.out.println(e.toString());
```

Create some abstract syntax trees in Java and print them.

Exercise 1.17 How would you add facilities to evaluate arithmetic expressions represented in Java as in Exercise 1.16?