Some interesting early papers on programming language design principles are due to Landin [2], Hoare [1] and Wirth [8]. Building on Landin's work, Tennent [6, 7] proposed the language design principles of *correspondence* and *abstraction*.

The principle of correspondence requires that the mechanisms of name binding (the declaration and initialization of a variable `let x = e`, or the declaration of a type `type T = int`, and so on) must behave the same as parametrization (the passing of an argument `e` to a function with parameter `x`, or using a type `int` to instantiate a generic type parameter `T`).

The principle of abstraction requires that any construct (an expression, a statement, a type definition, and so on) can be named and parametrized over the identifiers that appear in the construct (giving rise to function declarations, procedure declarations, generic types, and so on).

Tennent also investigated how the programming language Pascal would have looked if those principles had been systematically applied. It is striking how well modern languages, such as Scala and C#, adhere to Tennent's design principles, but also that Standard ML [3] did so already in 1986.

1.7 Exercises

Exercise 1.1. (i) File `Intro2.fs` contains a definition of the `expr` expression language and an evaluation function `eval`. Extend the `eval` function to handle three additional operators: `"max"`, `"min"`, and `"=="`. Like the existing operators, they take two argument expressions. The equals operator should return 1 when true and 0 when false.
(ii) Write some example expressions in this extended expression language, using abstract syntax, and evaluate them using your new `eval` function.
(iii) Rewrite one of the `eval` functions to evaluate the arguments of a primitive before branching out on the operator, in this style:

```
let rec eval e (env : (string * int) list) : int =
    match e with
    |  ...
    | Prim(ope, e1, e2) ->
      let i1 = ...
      let i2 = ...
      match ope with
        | "+" -> i1 + i2
        | ...
```

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

You need to extend the `expr` datatype with a new constructor `If` that takes three `expr` arguments.
(v) Extend the interpreter function `eval` correspondingly. It should evaluate `e1`, and if `e1` is non-zero, then evaluate `e2`, else evaluate `e3`. You should be able to

evaluate th expression `If(Var "a", CstI 11, CstI 22)` in an environment that binds variable `a`.

Note that various strange and non-standard interpretations of the conditional expression are 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 shortcut is rarely useful.

Exercise 1.2. (i) Declare an alternative datatype `aexpr` for a representation of arithmetic expressions without let-bindings. The datatype should have constructors `CstI`, `Var`, `Add`, `Mul`, `Sub`, for constants, variables, addition, multiplication, and subtraction.

Then $x * (y + 3)$ is represented as `Mul(Var "x", Add(Var "y", CstI 3))`, not as `Prim("*", Var "x", Prim("+", Var "y", CstI 3))`.
(ii) Write the representation of the expressions $v - (w + z)$ and $2 * (v - (w + z))$ and $x + y + z + v$.
(iii) Write an F# function `fmt : aexpr -> string` to format expressions as strings. For instance, it may format `Sub(Var "x", CstI 34)` as the string `"(x - 34)"`. It has very much the same structure as an `eval` function, but takes no environment argument (because the *name* of a variable is independent of its *value*).
(iv) Write an F# function `simplify : aexpr -> aexpr` to perform expression simplification. For instance, it should simplify $(x+0)$ to x, and simplify $(1+0)$ to 1. The more ambitious student may want to simplify $(1+0)*(x+0)$ to x. Hint: Pattern matching is your friend. Hint: Don't forget the case where you cannot simplify anything.

You might consider the following simplifications, plus any others you find useful and correct:

$$\begin{array}{l} \hline 0 + e \longrightarrow e \\ e + 0 \longrightarrow e \\ e - 0 \longrightarrow e \\ 1 * e \longrightarrow e \\ e * 1 \longrightarrow e \\ 0 * e \longrightarrow 0 \\ e * 0 \longrightarrow 0 \\ e - e \longrightarrow 0 \\ \hline \end{array}$$

(v) Write an F# function to perform symbolic differentiation of simple arithmetic expressions (such as `aexpr`) with respect to a single variable.

Exercise 1.3. Write a version of the formatting function `fmt` from the preceding exercise that avoids producing excess parentheses. For instance,

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

should be formatted as `"(a-b)*c"` instead of `"((a-b)*c)"`, and

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

should be formatted as `"a*b-c"` instead of `"((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"`, and

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

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

Hint: This can be achieved by declaring the formatting function to take an extra parameter `pre` that indicates the precedence or binding strength of the context. The new 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, such as minus (`-`), provides context precedence 5 to its left argument, and context precedence 6 to its right argument.

As a consequence, `Sub(Var "a", Sub(Var "b", Var "c"))` will be parenthesized `a-(b-c)` but `Sub(Sub(Var "a", Var "b"), Var "c")` will be parenthesized `a-b-c`.

Exercise 1.4. This chapter has shown how to represent abstract syntax in functional languages such as F# (using algebraic datatypes) and in object-oriented languages such as Java or C# (using a class hierarchy and composites).
(i) Use Java or C# classes and methods to do what we have done using the F# datatype `aexpr` in the preceding exercises. Design a class hierarchy to represent arithmetic expressions: it could have an abstract class `Expr` with subclasses `CstI`, `Var`, and `Binop`, where the latter is itself abstract and has concrete subclasses `Add`, `Mul` and `Sub`. All classes should implement the `toString()` method to format an expression as a `String`.

The classes may be used to build an expression in abstract syntax, and then print it, as follows:

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

(ii) Create three more expressions in abstract syntax and print them.
(iii) Extend your classes with facilities to evaluate the arithmetic expressions, that is, add a method `int eval(env)`.
(iv) Add a method `Expr simplify()` that returns a new expression where algebraic simplifications have been performed, as in part (iv) of Exercise 1.2.

References

1. Hoare, C.: Hints on programming language design. In: ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, Massachusetts 1973. ACM Press (1973)