Chapter 4

An Introduction to Functions

Through the agency of `with`, we have added identifiers and the ability to name expressions to the language. Much of the time, though, simply being able to name an expression isn't enough: the expression's value is going to depend on the context of its use. That means the expression needs to be parameterized; that is, it must be a *function*.

Dissecting a `with` expression is a useful exercise in helping us design functions. Consider the program

```
{with {x 5} {+ x 3}}
```

In this program, the expression `{+ x 3}` is parameterized over the value of `x`. In that sense, it's just like a function definition: in mathematical notation, we might write

$$f(x) = x + 3$$

Having named and defined f, what do we do with it? The WAE program introduces `x` and then immediately binds it to `5`. The way we bind a function's argument to a value is to apply it. Thus, it is as if we wrote

$$f(x) = x + 3; f(5)$$

In general, functions are useful entities to have in programming langugaes, and it would be instructive to model them.

4.1 Enriching the Language with Functions

We will initially model the DrScheme programming environment, which has separate windows for Definitions and Interactions. The Interactions window is DrScheme's "calculator", and the part we are trying to model with our calculators. The contents of the Definitions window are "taught" to this calculator by clicking the Run button. Our calculator should therefore consume an argument that reflects these definitions.

To add functions to WAE, we must define their concrete and abstract syntax. In particular, we must both describe a function definition, and provide a means for its use. To do the latter, we must add a new kind of expression, resulting in the language F1WAE.[1] We will presume, as a simplification, that functions consume only one argument. This expression language has the following BNF:

[1] The reason for the "1" will become clear in Section 6.

```
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

(The expression representing the argument supplied to the function is known as the *actual parameter*.)

We have dropped subtraction from the language on the principle that it is similar enough to addition for us to determine its implementation from that of addition. To capture this new language, we employ terms of the following type:

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])
```

Convince yourself that this is an appropriate definition.

Now let's study function definitions. A function definition has three parts: the name of the function, the names of its arguments, known as the *formal parameters*, and the function's body. (The function's parameters may have types, which we will discuss in Chapter X.) For now, we will presume that functions consume only one argument. A simple data definition captures this:

```
(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])
```

Using this definition, we can represent a standard function for doubling its argument as

```
(fundef 'double
        'n
        (add (id 'n) (id 'n)))
```

Now we're ready to write the calculator, which we'll call *interp*—short for *interpreter*—rather than *calc* to reflect the fact that our language has grown beyond arithmetic. The interpreter must consume two arguments: the expression to evaluate, and the set of known function definitions. This corresponds to what the Interactions window of DrScheme works with. The rules present in the WAE interpreter remain the same, so we can focus on the one new rule.

```
;; interp : F1WAE listof(fundef) → number
;; evaluates F1WAE expressions by reducing them to their corresponding values

(define (interp expr fun-defs)
  (type-case F1WAE expr
    [num (n) n]
```

```
    [add (l r) (+ (interp l fun-defs) (interp r fun-defs))]
    [with (bound-id named-expr bound-body)
          (interp (subst bound-body
                         bound-id
                         (num (interp named-expr fun-defs)))
                  fun-defs)]
    [id (v) (error 'interp "free identifier")]
    [app (fun-name arg-expr)
         (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
           (interp (subst (fundef-body the-fun-def)
                          (fundef-arg-name the-fun-def)
                          (num (interp arg-expr fun-defs)))
                   fun-defs))]))
```

The rule for an application first looks up the named function. If this access succeeds, then interpretation proceeds in the body of the function after first substituting its formal parameter with the (interpreted) value of the actual parameter. We see the result in DrScheme:

```
> (interp (parse '{double {double 5}})
          (list (fundef 'double
                        'n
                        (add (id 'n) (id 'n)))))
20
```

To make this interpreter function correctly, we must make several changes. First, we must adapt the parser to treat the relevant inputs (as identified by the BNF) as function applications. Second, we must modify the interpreter itself, changing the recursive calls to take an extra argument, and adding the implementation of app. Third, we must extend *subst* to handle the F1WAE language. Finally, we must write *lookup-fundef*, the helper routine that finds function definitions. The last two changes are shown in Figure 4.1.

Exercise 4.1.1 *Why is the argument expression of an application of type* F1WAE *rather than of type* WAE*? Provide a sample program permitted by the former and rejected by the latter, and argue that it is reasonable.*

Exercise 4.1.2 *Why is the body expression of a function definition of type* F1WAE *rather than of type* WAE*? Provide a sample definition permitted by using the former rather than the latter, and argue that it is reasonable.*

4.2 The Scope of Substitution

Suppose we ask our interpreter to evaluate the expression

(app 'f (num 10))

in the presence of the solitary function definition

(*fundef* 'f

```
    'n
    (app 'n (id 'n)))
```

What should happen? Should the interpreter try to substitute the *n* in the function position of the application with the number 10, then complain that no such function can be found (or even that function lookup fundamentally fails because the names of functions must be identifiers, not numbers)? Or should the interpreter decide that function names and function arguments live in two separate "spaces", and context determines in which space to look up a name? Languages like Scheme take the former approach: the name of a function can be bound to a value in a local scope, thereby rendering the function inaccessible through that name. This latter strategy is known as employing *namespaces* and languages such as Common Lisp adopt it.

4.3 The Scope of Function Definitions

Suppose our definition list contains multiple function definitions. How do these interact with one another? For instance, suppose we evaluate the following input:

```
(interp (parse '{f 5})
    (list (fundef 'f 'n (app 'g (add (id 'n) (num 5))))
        (fundef 'g 'm (sub (id 'm) (num 1)))))
```

What does this program do? The main expression applies `f` to `5`. The definition of `f`, in turn, invokes function `g`. Should `f` be able to invoke `g`? Should the invocation fail because `g` is defined after `f` in the list of definitions? What if there are multiple bindings for a given function's name?

We will expect this program to evaluate to `9`. We employ the natural interpretation that each function can "see" every function's definition, and the natural assumption that each name is bound at most once so we needn't disambiguate between definitions. Is is, however, possible to define more sophisticated scopes.

Exercise 4.3.1 *If a function can invoke every defined function, that means it can also invoke itself. This is currently of limited value because the language* F1WAE *lacks a harmonious way of terminating recursion. Consider adding a simple conditional construct (such as* `if0`*, which succeeds if the term in the first position evaluates to* `0`*) and writing interesting programs in this language.*

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])

(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])

;; lookup-fundef : symbol listof(fundef) ⟶ fundef
(define (lookup-fundef fun-name fundefs)
  (cond
    [(empty? fundefs) (error fun-name "function not found")]
    [else (if (symbol=? fun-name (fundef-fun-name (first fundefs)))
              (first fundefs)
              (lookup-fundef fun-name (rest fundefs)))]))

;; subst : F1WAE symbol F1WAE → F1WAE
(define (subst expr sub-id val)
  (type-case F1WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]
    [app (fun-name arg-expr)
         (app fun-name (subst arg-expr sub-id val))]))
```

Figure 4.1: Implementation of Functions: Support Code

```
;; interp : F1WAE listof(fundef) → number
(define (interp expr fun-defs)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs) (interp r fun-defs))]
    [with (bound-id named-expr bound-body)
          (interp (subst bound-body
                         bound-id
                         (num (interp named-expr fun-defs)))
                  fun-defs)]
    [id (v) (error 'interp "free identifier")]
    [app (fun-name arg-expr)
         (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
           (interp (subst (fundef-body the-fun-def)
                          (fundef-arg-name the-fun-def)
                          (num (interp arg-expr fun-defs)))
                   fun-defs))]))
```

Figure 4.2: Implementation of Functions: Interpreter