1 Evaluation Steps

The addition of functions to the language requires the evaluation of two constructs: function definitions and function calls. The general steps required for evaluation are given here and an example is given in the next section.

1.1 Function Definition

To support static scoping in a language that allows functions defined within other functions and that allows these nested functions to escape from their defining context (the enclosing function) we will implement functions as closures. A closure is a run-time representation that encapsulates the code for the function (i.e., what the function should do when called) and the defining context (the variables in the outer scopes that the function may access). We will refer to this defining context as the defining environment.

It is this need to capture the defining environment that drives the use of an implementation that deviates from the use of a traditional stack. Specifically, in a stack-based implementation, returning from a function initiates a pop of the local scope (stack frame). If a nested function is returned that later needs access to the just popped local scope, then that function cannot safely access the needed variables.

In short, the evaluation of a function definition is the creation of a closure encapsulating the code for the function and the defining environment (the *currently active environment* at the time the function definition is evaluated).

1.2 Function Call

The evaluation of a function call requires multiple steps. The conceptual complexity in evaluating a function call stems from a context switch between the calling environment (i.e., the environment active where the call takes place) and the callee or function environment (i.e., the environment used to execute the body of the function). This context switch is what will enable the implementation of static scoping and is the entire reason for storing the defining environment within a closure.

The steps are as follows.

1. Evaluate the "function expression" to gain access to a closure.
2. Evaluate the actual arguments in the context of the *currently active environment* to compute their values.
3. Create a new frame to store the parameters and local variables for the invoked closure.
4. Bind the values of the actual arguments with the corresponding formal parameters in this new table.
5. **Set up the new context for the pending switch:** Link this new table to the *defining environment* stored within the closure.
6. **Context switch:** evaluate the body of the function with this new environment (that from the previous step with the new table at the head) as the now *currently active environment*.
7. Upon return from the function call, restore the previous environment.

2 Example

When the interpreter begins the initial top-level environment (Fig. 1) contains only bindings for the built-in symbols such as `hd` and `tl`. The interpreter prompts the user for input and waits until an expression is typed. When an expression is typed it will be evaluated in this top-level environment.

Assume that the user has typed the following expression:

```
prompt> fun f x =
             fn y =>
               x + y
        ;
```

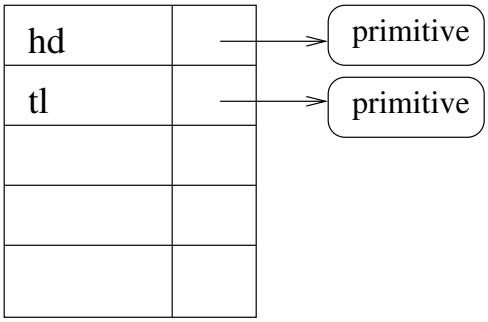

Figure 1: Top-level environment

This expression binds to f result of a function definition. Doing so creates a closure containing a reference to the code and a pointer to the **defining environment** (the code is excluded from the diagrams to focus on the environment chains). In this case the defining environment is the top-level environment because the function is defined at the top-level (i.e., the top-level environment is the *currently active environment*). The binding of f can be seen in Figure 2.

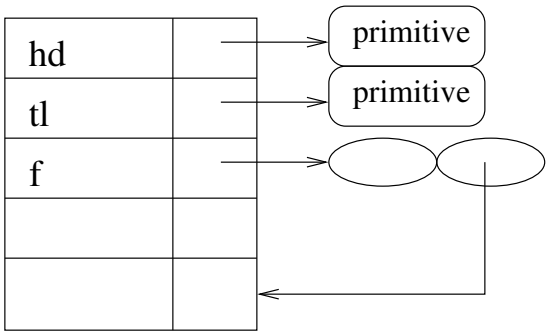

Figure 2: Definition of f

Now assume that the user types the following expression:

```
prompt> val g = f 3;
```

As part of the evaluation of this expression, f is applied to 3. Since f is bound to a closure, this application creates a new frame to contain the parameters. This new frame will point to the environment in which the closure was defined; this is the same environment that was originally stored in the closure bound to *f*. See Figure 3.

The application of f to 3 then continues with the evaluation of the body of the function using this new environment (which now becomes the *currently active environment*). The body is an anonymous function expression which evaluates to a new closure. The defining environment for this new closure is the *currently active environment* (which is the environment that includes x (the parameter) bound to 3).

A closure is created that references the code for the anonymous function and the defining environment. This closure is the result of the call `f 3`. The evaluation of the `val` then binds the closure to *g*. This `val` was evaluated in the top-level environment so the binding for g is added to the top-level environment. See Figure 4.

An application of `g` (e.g., `(g 5)`) will create a new frame to contain the parameters because `g` is bound to a closure. This new frame will point to the environment in which the closure was defined. This environment is found in the closure bound to *g*. See Figure 5.

Evaluation of the code for this function (`x + y`) continues by searching through the chain of environment frames to find `x` and `y` and then returns the sum.

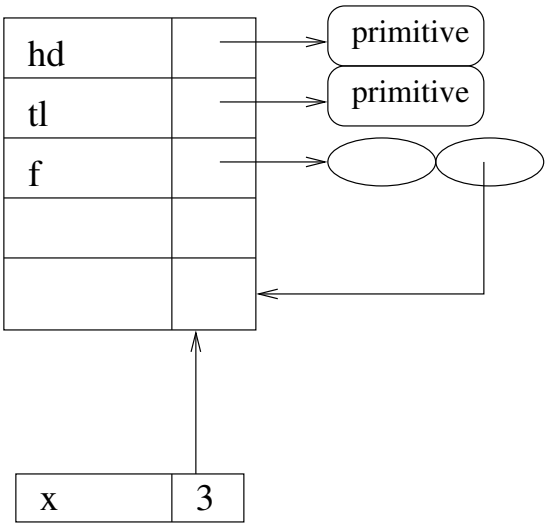

Figure 3: Invocation of f – new frame

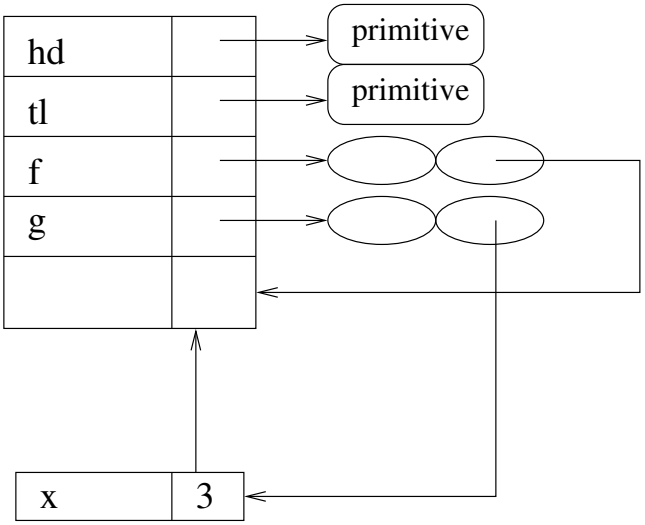

Figure 4: Definition of g

An example of another definition can be seen in Figure 6. This definition is the result of evaluating 'val h = f 4;' in the top-level environment.

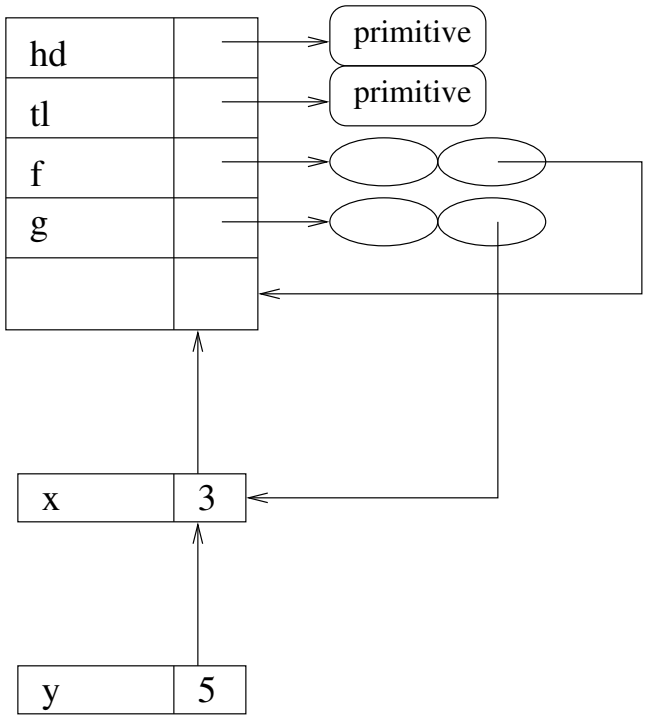

Figure 5: Invocation of g – new frame

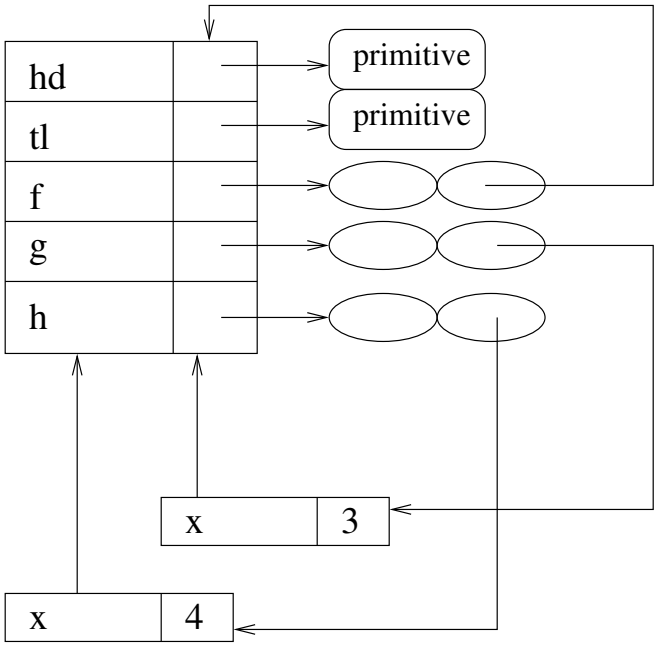

Figure 6: Definition of h