CIS 343 - Structure of Programming Languages

Nathan Bowman

Based on slides by Ira Woodring

Control Flow in Scheme (Follows the Sebesta Text Chapter 15)

In imperative programming, control flow generally consists of `if` statements and loops

Functional programming also has `if` statements, but we will not be using loops

Instead, functional programming relies on recursion for repetition

We will learn about:

- `if`
- `cond`/`else`
- recursion

Scheme also has `case` and `match` statements that somewhat resemble `cond`, but we will not discuss those. Feel free to look them up and use them if you are curious.

Predicate functions are functions that return Boolean values. Some predicate functions provided by Scheme are

=, eq?, NOT, >, <, >=, <=, EVEN?, ODD?, ZERO?

Functions that return #t or #f (true or false) but contain words end with the '?' character.

If a list is being examined, a non-empty list returns #t and an empty list returns #f.

IF:

```
(IF predicate then_expression else_expression)
(if (<> n 1701)
  #f  ;; What to do if the statement is true.
  #t  ;; What to do if the statement evaluated to false.
)
```

A more complicated example:

```
(define (fib n)
  ;; Calculate the nth Fibonacci number recursively
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

COND is used for multiple conditionals. For instance:

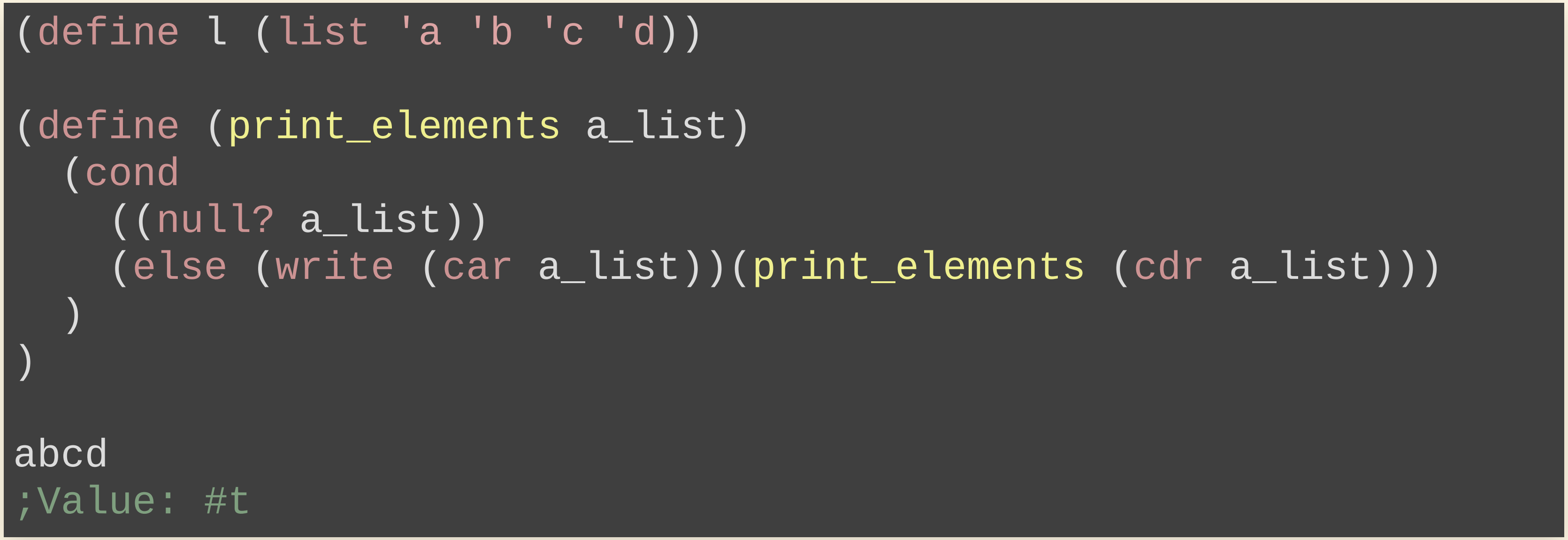
```
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))
```

Notice there are multiple predicates here.

A good understanding of recursion is essential in functional languages, as they lack iteration. For instance, if we want print all members of a list in an imperative language, we would do something like this:

```
for(Member m:list){
  print m;
}
```

In a functional language though, we use recursion:



```
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (write (car a_list))(print_elements (cdr a_list)))
  )
)

abcd
;Value: #t
```

You may have noticed above that the return value from the function was `#t`. We never explicitly stated this, so where did that come from?

Like Ruby, Scheme doesn't require a `return` statement to return a value. The final value in an expression is what is returned. So what happened above? The list was recursively broken into two parts - the first element, and the rest. The first element was printed, then we recursively printed the rest. When there were no more parts the `null`? function returned `#t`.

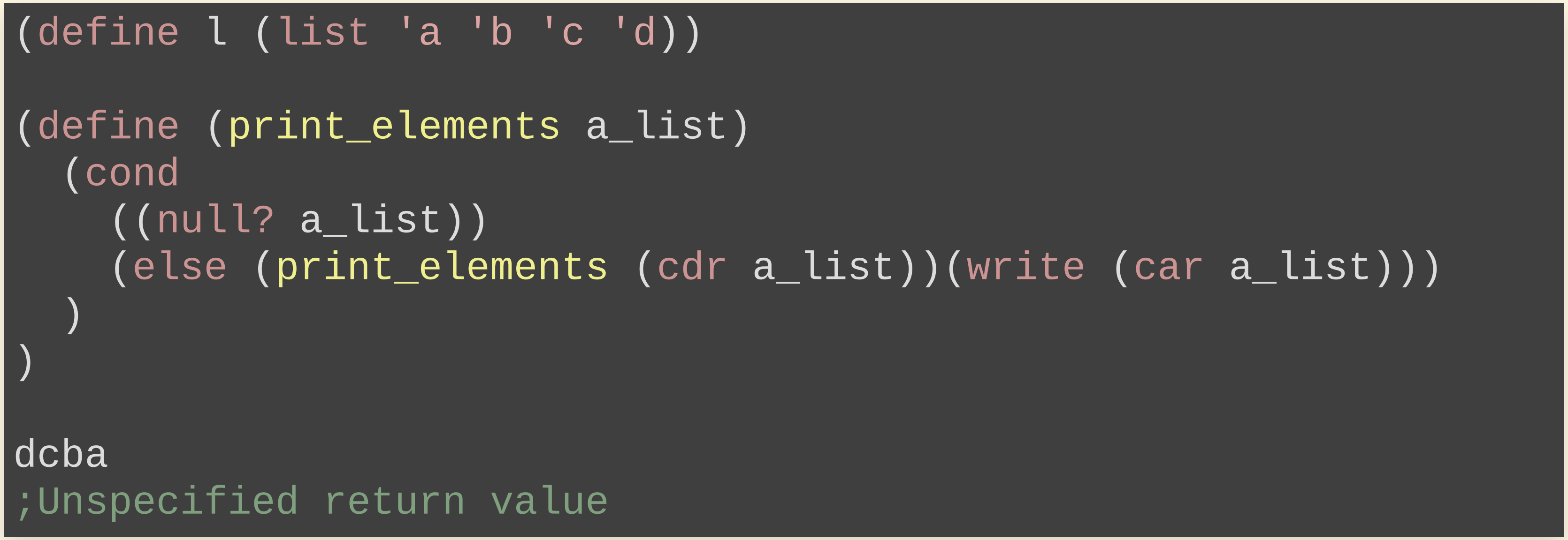
Coding

How could we change this code to print the list backward?

```scheme
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (write (car a_list))(print_elements (cdr a_list)))
  )
)

abcd
;Value: #t
```



```
(define l (list 'a 'b 'c 'd))

(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (print_elements (cdr a_list))(write (car a_list)))
  )
)

dcba
;Unspecified return value
```

Why unspecified return value?

The last line that is run is not the null check anymore. It is a write statement! How could we fix it to return a #t or #f value?

If we needed it to return #t or #f we could do this:

```scheme
(define (print_elements a_list)
  (cond
    ((null? a_list))
    (else (print_elements (cdr a_list))(write (car a_list))#t)
  )
)
```

What will this code do?

```
(define l (list 1 2 3 4))
(define (new_func a_list)
  (cond
    ((null? a_list) 0)
    (else (+ (car a_list)(new_func (cdr a_list))))
  )
)
(new_func l)
```

Control flow:

- `if` statements are used for simple `if` or single `if/else`
- `cond` statements are used when there is more than one conditional
- No looping -- all repetition done through recursion