CS403: Programming Languages

Scam/Scheme Basics

Key ideas

- running the interpreter
- evaluating Scam expressions
- binding names to things (definitions)
- function definitions
- conditional expressions
- nesting functions

Running the interpreter

Once Scam is installed, simply type *scam* at the system prompt. Once inside the interpreter, you can load a file containing Scam definitions thusly:

```
(include "defs.scm")
```

where *defs.scm* is the name of the file to be loaded. You can also start up the interpreter and read stuff from a file by the command:

```
scam defs.scm
```

I automate this task by placing the following macro definition in my .exrc (or .vimrc) file:

```
map @ :!scam % ^M
```

The `^M` is entered as `ctl-v ctl-m`. This will automatically load the file I'm editing into the Scam interpreter when I press the `'@'` key. Remember to save your file before running the macro. To exit *scam*, type `ctl-c q`.

Scheme basics

Every expression in Scheme has a value and every non-trivial expression is expressed in prefix notation. The value of the expression 6 is 6 while the value of the expression (* 4 6) is 24. Note that a non-trivial expression is enclosed in parentheses. Expressions can be nested, as in:

```
(* (+ 2 2) (+ 3 3))
```

which has a value of 24 as well. The value of an expression is generally obtained by evaluating the operator and the operands, then applying the operator to the operands.

Definitions in Scheme

Names can be bound to values in Scheme. For example,

```
(define x 2)
```

binds the name x to the value 2. If x is ever evaluated, it returns the value 2. Note that the definition is not evaluated as most expressions. The x is not evaluated (in fact, it may be undefined and not have a value). Expressions that are not evaluated in the normal fashion are called special forms. Scheme has a very limited number of special forms, which makes Scheme interpreters very easy to write. Names can be bound to functions as well as values. For example

```
(define (sqr x)
    (* x x)
    )
```

binds the name *sqr* to the function which squares its argument. Thus

```
(sqr 5)
```

evaluates to the value 25.

Conditionals

The *cond* special form has the following syntactic structure

```
(cond
    (<pred_1> <expr_1_a> <expr_1_b> ... <expr_1_z>)
    (<pred_2> <expr_2_a> <expr_2_b> ... <expr_2_z>)
      .
      .
      .
    (<pred_n> <expr_n_a> <expr_n_b> ... <expr_n_z>)
    )
```

The value of a *cond* exprression is the expression associated with the first true predicate. Like switch statements in C and C++, the cond special form has a default clause

```
(cond
    (<pred_1> <expr_1_a> <expr_1_b> ... <expr_1_z>)
    (<pred_2> <expr_2_a> <expr_2_b> ... <expr_2_z>)
      .
      .
      .

    (else <expr_n_a> <expr_n_b> ...  <expr_n_z>)
    )
```

The value of a *cond* expression is the expression associated with the *else* if none of the previous predicates evaluate to true.

```
(define (maximum a b)
    (cond
        ((> a b) a)
        (else b)
        )
    )
```

A sometimes handier version of conditional evaluation is the *if* special form. Its syntax is

```
(if <pred>
    <expr_t>
    <expr_f>
    )
```

The expression `<expr\_t>` is evaluated if `<pred>` evaluates to true; `<expr\_f>` is evaluated otherwise. Let's use *if* to compute the absolute value of a number.

```
(if (< num 0)
    (- num)
    num
    )
```

The return value is again the last expression evaluated. The downside of the if is that only a single expression can serve as the true or false expression. To evaluate multiple expressions, a begin expression (similar to a C block) is needed:

```
(if (< num 0)
    (begin
        (showln "debug: flipping the sign")
        (- num)
        )
    num
    )
```

Logical expressions can be combined using *and* and *or* special forms and logical values can be reversed with *not*.

```
(and <pred_1> <pred_2> ...  <pred_n>)
(or  <pred_1> <pred_2> ...  <pred_n>)
(not <pred>)
```

Of course, these special forms can be nested. Scheme short-circuits the evaluation of logical expressions in the same fashion as C.

Functions

Functions are easily defined in Scheme. The syntax of a definition is

```
(define (<name> <arg1> <arg2> ... <argn>)
    <exp1> <exp2> ... <expn>)
```

Here is the definition of a function which performs addition

```
(define (add a b)
     (+ a b)
     )
```

It would be called thusly...

```
(add 3 5)
```

The value 3 would be bound to the name *a* and the value 5 would be bound to the name *b*. The interpreter would return a value of 8 since the value of a function call is the value of the last expression in the function body.

Defining functions - finding square roots

Scheme is well adapted to implemented recursive and iterative searches. Newton's method for finding square roots can be easily implemented in Scheme. The text uses a top down approach, which I term programming by procrastination (never write a function today what you can write tomorrow). Here is the top-level code

```
(define (sqrt-search guess target)
    (if (stop? guess target)
        guess
        (sqrt-search (improve guess target) target)
        )
    )
```

Note that *stop?* and *improve* have been postponed. Let's write those functions now (instead of tomorrow).

```
(define (stop? guess target)
    (< (abs (- (* guess guess) target)) .00001)
    )

(define (improve guess target)
    (* 0.5 (+ guess (/ target guess)))
    )
```

We assume that a proper binding for `abs` exists. It turns out that our *sqrt-search* procedure has two helper functions, both of which are polluting the namespace. Scheme allows nested procedures (as does Pascal) to localize the names of the helper functions. A revised implementation looks like...

```
(define (sqrt-search guess target)
    (define (stop? guess target)
        (< (abs (- (* guess guess) target)) .00001)
        )
    (define (improve guess target)
        (* 0.5 (+ guess (/ target guess)))
        )

    (if (stop? guess target)
        guess
        (sqrt-search (improve guess target) target)
        )
    )
```

Now names *stop?* and *improve* are bound only within the body of the procedure and are not visible outside the function. Note that the guess parameter of the stop? function is an alias for the guess parameter of the sqrt-search routine. Since the body of stop? is within scope of the sqrt-search parameters, we can eliminate the redundant parameters in the nested functions and modify the calls accordingly:

```
(define (sqrt-search guess target)
    (define (stop?)
        (< (abs (- (* guess guess) target)) .00001)
        )
    (define (improve)
        (* 0.5 (+ guess (/ target guess)))
        )

    (if (stop?)
        guess
        (sqrt-search (improve) target)
        )
    )
```

Note that we can perform the same manipulation for the *improve* function and the target parameters.