CSE341 Spring '05 Due Friday, May 20
Assignment 5

For this assignment you will implement a small procedural programming language called DysFun by converting it into Scheme expressions. This uses the same principles introduced in part 2 of homework 4: Scheme programs are merely lists and can be easily constructed by other Scheme programs. This is a *difficult* assignment, so start early. The sample solution is about 200 lines of liberally-indented code. You will be provided a skeleton code file with some useful included functions and the rest ready to be filled in. Be sure to read the skeleton code so you know what provided functions are at your disposal. You are also encouraged to use some of your functions from homework 4, particularly `contains?` and `split-by`.

1 DysFun overview

Like the infix expressions in homework 4, DysFun programs are represented as Scheme S-expressions and can *always* be written as a Scheme quoted list, despite looking quite unlike Scheme code. See the provided code for an example.

Expressions

DysFun expressions are like infix expressions, with the addition of C/Java-style function calls. A function call takes the form:

```
f (arg1, arg2, arg3, ...)
```

Of course, function calls can be mixed with infix operators as you would expect:

```
42 - foo(bar * bob, 24)
```

The provided function `application?` will detect when a DysFun expression is a function application.

Statements

In the simplest case a DysFun statement is just a DysFun expression, although there are several special forms that will be discussed later. Because semi-colons are used to begin Scheme comments, DysFun statements must be separated with exclamation marks (`!`). Remember that spaces must still be used liberally in Scheme S-expressions, so a typical sequence of statements might look like:

```
display (2 * 2) !
newline () !
display ("Hello, world!") !
```

A sequence of statements will be referred to as a statement block.

Special forms

In addition to plain expressions, there are several special forms that can appear as statements. Unlike in Java and C, they must still be terminated with an exclamation mark.

Assignment

An assignment takes the form:

```
VAR := EXPR
```

Where `VAR` is a Scheme symbol and `EXPR` is a DysFun expression. For example:

```
x := 2 + 2 !
y := x * x !
```

Assignments are always mutating, just like in C and Java. The included function `assign?` will detect when an expression is an assignment statement.

If

An **if** statement takes the form:

```
if CONDITION STATEMENT_BLOCK
```

Where `CONDITION` is a DysFun expression and `STATEMENT_BLOCK` is a block of `!`-separated statements. Because DrScheme allows curly braces to be used as parentheses, you can write **if** statements in a (somewhat) familiar form:

```
if (x < 5) {
    x := x + 5 !
    display ("This does nothing useful") !
} !
```

Take notice that **if** statements are like any other statement and must be terminated with with `!`. Don't be fooled by curly braces or formatting; this piece of code is represented as a list in Scheme and can be processed with normal list-processing functions. The included function `if?` detects when an expression is an **if** statement. **if** statements may be nested.

If-else

An **if-else** statement takes the form:

```
if CONDITION BLOCK1 else BLOCK2
```

As in:

```
if (x < 5) {
    display ("X is < 5") !
} else {
    display ("X is >= 5") !
} !
```

The included function `if-else?` detects when an expression is an **if-else** statement. The semantics are the same as in C or Java. **if-else** statements may be nested.

While

A **while** statement takes the form:

```
while CONDITION BLOCK
```

As in:

```
while (x < 10) {
    x := x + 1 !
    display (x) !
} !
```

The included function `while?` detects when an expression is a **while** statement. **while** statements may be nested.

Functions

A DysFun function definition takes the form:

```
func FUNCNAME (arg1, arg2, arg3, ...) BLOCK
```

For example, a function of no arguments that loops forever, printing numbers would look like:

```
func useless () {
    x := 0 !
    while (#t) {
        display (x) !
        newline () !
        x := x + 1 !
    } !
}
```

Notice that variables can be used without any prior declaration; in Scheme (and C or Java, for that matter), this is not the case. Variables used in this way have scope local to the function. Functions may not be nested (unless you do the extra credit).

Return

Within the body of a function, the special function `return`, taking a single argument, causes the function to immediately return that argument, no matter where the `return` occurs. Essentially, it works the same way as return in C and Java. Consider:

```
func double (x) {
    return (2 * x) !
    display ("This is never executed") !
}
```

2 Implementation Strategy

We have provided a file `hw5-skel.scm` on the course web that contains several useful functions as well as an example of a DysFun function.

Plain Expressions

A plain expression can be converted to Scheme much like infix expressions in homework 4, so `infix->prefix` is an excellent starting point. There are two major differences, though:

- The set of infix operators (and the order in which to test for them) is dictated by the provided top-level definition `infix-operators`. This list is ordered from lowest precedence to highest precedence, which is the order in which you want to split an expression into sub-expressions.
- You need an additional case to handle function calls. Remember that the argument list can be empty for a call to a no-argument function.

Unfortunately, comma (`,`) has a special meaning in quoted Scheme expressions and does not show up in argument lists as you would expect. A function has been provided for you called `unquote->comma` which will give you a list suitable for splitting by the symbol `|,|`. You can think of it as a black box and use it without worrying too much about what it's doing, as long as you do use it. For example:

```
(unquote->comma '(1, 2 * (1 + 1), foo)) => (1 |,| 2 * (1 + 1) |,| foo)
```

The symbol `|,|` might look strange, but it can be quoted in your implementation (for example, to pass to `split-by`) as `'|,|`

Assignment

You may assume that anything on the left-hand side of an assignment has already been bound. Thus, assignment translates rather naturally into Scheme `set!`

If

An **if** expression translates easily into a Scheme `if` expression. Since there is no alternate (else branch), you can simply omit it from the resulting Scheme expression, i.e.:

```
(if CONDITION
    CONVERTED_CODE_BLOCK_GOES_HERE)
```

Keep in mind that a DysFun statement block can translate to more than one Scheme expression. You will have to use `begin` to cope with having multiple expressions in the consequent of a Scheme `if`.

If-else

if-else translates almost the same as **if**, except that there is an alternate block that must be represented in the resulting `if` expression in Scheme.

While

while is somewhat tricky. Although there are looping constructs in Scheme, you are *not* allowed to use them on this assignment. Rather, you must convert the loop into a function that calls itself tail-recursively. In order for the function to be able to call itself, it will have to be introduced in a `letrec` and given a name. Choosing this name is non-trivial; if you pick the name of a variable used elsewhere in the DysFun program it could cause it to behave incorrectly. Luckily, DrScheme has a function, `gensym`, which returns a symbol guaranteed to be different from every other symbol ever encountered. You can use this to automatically generate a name for your looping function.

Functions

Implementing functions is likely the most difficult part (conceptually) of this assignment. The Scheme `lambda` expression you produce must:

- Initially bind all variables that might later be used to dummy values. This is because a DysFun function does not need to declare its local variables before using them. (Use `let` and the `unbound-vars` function which you will write).
- Have a special function in its environment, `return`, which when called with a value causes the function to immediately return that value (use a continuation with `let/cc`)

Keep in mind that a DysFun argument list has commas while a Scheme `lambda` does not. You can use `unquote->comma` and the built-in function `filter` to aid you here.

3 Problems

Find the following function stubs in the skeleton code and complete them to work as specified. For functions that process individual DysFun statements, you should assume that the statement

will be passed in by itself without the trailing ! terminator. Many of your functions will have to recursively call each other, so be prepared to use functions that you haven't written yet.

1. Write a function `union` of the form `(union ls1 ls2)`, where `ls1` and `ls2` are lists. The result should be the union of the two lists treated as sets; in other words, the result should not have any duplicate elements. You may assume that `ls1` and `ls2` do not contain duplicate elements to begin with. For example, `(union '(a b) '(a c)) => (a b c)`, although the order of the result can differ.

2. Write a function `unbound-vars` of the form `(unbound-vars bound expr)`, where `expr` is an arbitrary piece of DysFun code and `bound` is a list of variables explicitly bound in the code already (for example, the arguments to a function). The result should be the set of all unbound variables (symbols) in the expression, with no duplicates (*hint*: use `union`). A symbol is unbound if it not in `bound`, the list of explicitly bound variables, and is also not in the provided top-level binding `reserved-symbols`, a list of symbols that have special meanings and should not be considered local variables. Remember that DysFun code is represented as a Scheme S-expression. An S-expression can either be an atom (the only kind of which you care about are symbols) or a pair (a cons cell) of two other S-expressions. You should be able to write the function taking into account only 3 different cases. Use built-in type predicates such as `symbol?` and `pair?`

3. Write a function `dysfun-expr->scheme-expr` of the form `(dysfun-expr->scheme-expr expr)`, where `expr` is a plain DysFun expression. The result should be an equivalent Scheme expression. Remember that `infix->prefix` from homework 4 can be used as a starting point.

4. Write a function `dysfun-assign->scheme-set` of the form `(dysfun-assign->scheme-set expr)`, where `expr` is a DysFun assignment statement. The result should be an equivalent Scheme `set!` expression.

5. Write a function `dysfun-if->scheme-if` of the form `(dysfun-if->scheme-if expr)`, where `expr` is a DysFun if statement. The result should be an equivalent Scheme `if` expression.

6. Write a function `dysfun-if-else->scheme-if` of the form `(dysfun-if-else->scheme-if expr)`, where `expr` is a DysFun if-else statement. The result should be an equivalent Scheme `if` expression.

7. Write a function `dysfun-while->scheme-expr` of the form `(dysfun-while->scheme-expr expr)`, where `expr` is a DysFun while statement. The result should be an equivalent Scheme expression.

8. Write a function `dysfun-stmt->scheme-expr` of the form `(dysfun-stmt->scheme-expr expr)`, where `expr` is a DysFun statement. This function should simply use the provided predicates to test what kind of statement `expr` is and dispatch one of the above functions to convert it to a Scheme expression.

9. Write a function `dysfun-stmts->scheme-exprs` of the form `(dysfun-stmts->scheme-exprs expr)`, where `expr` is a DysFun `!`-separated statement block. The result should be a list of equivalent Scheme expressions. Remember that your other routines expect the `!` to be stripped off.

10. Write a function `dysfun-func->scheme-lambda` of the form `(dysfun-func->scheme-lambda expr)`, where `expr` is a DysFun function definition. The result should be an equivalent Scheme `lambda` expression.

11. Write a *DysFun* function of the form `func map-loop (f, ls) { ... }` that works the same as the Scheme function `map`; that is, it returns a new list which is the result of applying `f` to each element in `ls`. All the built-in Scheme functions listed in the provided binding `snarfed-functions` are available to be used directly in your DysFun code (with DysFun syntax, of course). Your function must be implemented as a loop rather than recursively. Bind it to the top-level binding `map-loop-dysfun`. Run `dysfun-func->scheme-lambda` on it and bind the result to `map-loop-scheme` so that you can examine the Scheme code your implementation produces. Finally, run `eval` on `map-loop-scheme` and bind the result to `map-loop`. You may now call the function `map-loop` like any other Scheme function to test if it works. An example of how to do this is provided in the skeleton code with a DysFun function that calculates Fibonacci numbers.

Extra credit:

Clearly indicate in comments at the top of your program whether you have attempted the extra and which parts you have attempted.

1. Modify your implementation of while loops so that the no-argument functions `continue` and `break` are available within the loop body. `Continue` causes the loop to immediately start over (testing the condition and breaking out of the loop if necessary, of course), while `break` causes the loop to immediately terminate.

2. Modify your implementation to allow nested functions to be declared and called within other functions. This means that you will need to handle function declarations as possible statements. In addition, you should not use `unbound-vars` to determine which variables are implicitly bound in the function; it is too naive to properly handle nested functions. Feel free to write a smarter version, but give it a different name and leave `unbound-vars` unchanged.