CSC324 Assignment 1: Syntactic Sugar

You will be able to complete this assignment after lecture 2, but complete lab 2 before starting the assignment. Lab 2 (task 2) provides step-by-step guidance on how to write structurally recursive programs that takes code as input. Please use this assignment as an opportunity to apply these techniques and develop your ability to write code in a systematic way. Practice working incrementally on runnable code and test frequently, even if it feels awkward right now, and you will be on your way to becoming an expert programmer!

In this assignment, we will explore the idea of **syntactic sugar**. This term is used to refer to language features created for programmer convenience, but that do not change the semantics or expressiveness of a language. For example, we can think of pattern matching expressions in Racket as *syntactic sugar*, since any code that uses pattern matching can be rewritten using `if` expressions:

```
; this expression uses `match`
(match lst
  ((cons x xs) x)
  (_          0))

; this expression is equivalent to above
(if (pair? lst)
    (let ((x  (car lst))
          (xs (cdr lst)))
      x)
    0)
```

In this assignment, we will perform this conversion: we take code written using language features that are considered syntactic sugar, and **desugar** it into code that does not contain these language features.

To begin, let's consider two grammars: one for a language **MandarinBasic** without syntactic sugar, and another for a more featureful language **MandarinSugar** with syntactic sugar. Here are the grammars for both languages:

```
;; values are the empty list, booleans, numbers
<VALUE> ::= () | #t | #f | NUMBERS

<MandarinBasicProgram> ::= <MandarinBasicExpr>

<MandarinBasicExpr> ::= <VALUE>
                   | IDENTIFIER                                 ; variable names
                   | (lambda (IDENTIFIER) <MandarinBasicExpr>)     ; unary function creation
                   | (<MandarinBasicExpr> <MandarinBasicExpr>)        ; unary function call
                   | (if <MandarinBasicExpr> <MandarinBasicExpr> <MandarinBasicExpr>) ; if expressions
                   | (+ <MandarinBasicExpr> <MandarinBasicExpr>)      ; binary addition
                   | (let ((IDENTIFIER <MandarinBasicExpr>) ...) <MandarinBasicExpr>) ; let expressions
                   | (cons <MandarinBasicExpr> <MandarinBasicExpr>)   ; create pairs
                   | (car <MandarinBasicExpr>)                     ; first
                   | (cdr <MandarinBasicExpr>)                     ; rest
                   | (pair? <MandarinBasicExpr>)                   ; pair? predicate
                   | (= <MandarinBasicExpr> <MandarinBasicExpr>)      ; equality testing
```

```
<MandarinSugarProgram> ::= <MandarinSugarExpr>

<Pattern> ::= <VALUE>                       ; value pattern
            | IDENTIFIER                    ; variable pattern
            | (cons IDENTIFIER IDENTIFIER) ; pair pattern

<MandarinSugarExpr> ::= <VALUE>
                   | IDENTIFIER                                   ; variable names
                   | (lambda (IDENTIFIER IDENTIFIER ...)
                       <MandarinSugarExpr>)                         ; n-ary function creation
                   | (<MandarinSugarExpr>
                       <MandarinSugarExpr> <MandarinSugarExpr> ...)   ; n-ary function call
                   | (if <MandarinSugarExpr> <MandarinSugarExpr> <MandarinSugarExpr>) ; if expressions
                   | (+ <MandarinSugarExpr> <MandarinSugarExpr> ...)    ; n-ary addition
                   | (let ((IDENTIFIER <MandarinSugarExpr>) ...) <MandarinSugarExpr>) ; let expressions
                   | (cons <MandarinSugarExpr> <MandarinSugarExpr>)     ; create pairs
                   | (car <MandarinSugarExpr>)                         ; first
                   | (cdr <MandarinSugarExpr>)                         ; rest
                   | (pair? <MandarinSugarExpr>)                       ; pair? predicate
                   | (= <MandarinSugarExpr> <MandarinSugarExpr>)        ; equality testing
                   | (match <MandarinSugarExpr>                        ; match expressions
                      (<Pattern> <MandarinSugarExpr>) ...
                      (_ <MandarinSugarExpr>))  ; match expressions end with a wildcard
```

Recall that the `...` after the `match` means that there could be zero or more of structures of the form `(<Pattern> <MandarinSugarExpr>)` in a `match` expression.

Your task is to write a function called `desugar` that takes a Racket data structure representing code in MandarinSugar, and convert it to code in MandarinBasic. In particular, n-ary addition, n-ary functions, and match expressions should be converted to MandarinBasic in the following way:

Desugaring Addition

A MandarinSugar expression

```
(+ a b c d ...)
```

should desugared into the expression

```
(+ a (+ b (+ c (+ d ...))))
```

Note: Here, we will assume that `a`, `b`, `c`, ... etc are just identifiers. If `a`, `b`, `c`, ... are themselves MandarinSugar expressions, then those expressions will also need to be desugared.

Desugaring Functions

To desugar functions with multiple arguments, we need to consider both function creation (lambdas) and function calls. A function (lambda) expression with 2 arguments can be desugared into a lambda in terms of the first argument that returns a function. In other words, a MandarinSugar expression

```
(lambda (x y) expr)
```

should be desugared into the expression

```
(lambda (x)
   (lambda (y) expr))
```

This means that when we call a two-argument expression `f`, we will need to first supply the first argument to get a function in terms of the resulting argument, and then call that function. In other words, a MandarinSugar expression

```
(f a b)
```

should be desugared into the expression

```
((f a) b)
```

Putting these desugaring rules together, the MandarinSugar expression:

```
((lambda (a b c) (+ a b c)) 1 2 3)
```

should be desugared into this MandarinBasic expression:

```
((((lambda (a)
     (lambda (b)
       (lambda (c) (+ a (+ b c)))))
   1) ; a
  2) ; b
 3) ; c
```

Note: MandarinBasic and MandarinSugar contain two structures which resemble 2-ary function application: `cons` and `=`. These should *not* be desugared in the same way as other functions. In particular, `(cons e1 e2)` and `(= e1 e2)` should be desugared to `(cons e1' e2')` and `(= e1' e2')` respectively, assuming `e1` desugars to `e1'` and `e2` desugars to `e2'`.

Desugaring Match Expressions

Match expressions should be desugared into nested if expressions where each condition tests whether the corresponding pattern matches. There are three types of patterns, which we will describe in turn.

The `<VALUE>` pattern is matched when the pattern contains a literal value (e.g., #t, #f, numbers and `()`). This pattern is matched when the expression's value matches the literal value exactly. Thus, the expression

```
(match x
  (1 5)
  (_ 6))
```

should be desugared to:

```
(if (= x 1)
    5
    6)
```

The `IDENTIFIER` pattern is matched when the pattern contains a single identifier, to be interpreted as a variable binding. Thus, the expression

```
(match x
  (a (+ a 1))
  (_ 2))
```

should create a local variable binding that binds the identifier `a` to the value of the expression `x`. The above `match` expression thus desugars to:

```
(if #t          ; an identifier pattern always matches
   (let ((a x))
      (+ a 1))
   2)
```

Finally, the pair pattern `(cons IDENTIFIER IDENTIFIER)` is matched when the pattern contains a `cons` pair, with identifiers intended to bind to the `car` and `cdr` of the pair. The expression

```
(match lst
  ((cons x xs) (+ x 1))
  (_           0))
```

should desugar into an expression that tests whether `lst` is a pair, and if so, creates the appropriate local bindings. The above `match` expression thus desugars to:

```
(if (pair? lst)
    (let ((x  (car lst))
          (xs (cdr lst)))
       (+ x 1))
    0)
```

In addition, unlike in Racket and in accordance with the MandarinSugarExpr grammar, all `match` expression will end with a "wildcard" pattern `_`. Thus, for all MandarinSugar `match` expressions, the desugared `if` expressions will always have a specified "else" branch.

If there are multiple patterns, each of them should be processed sequentially. For example, consider the below expression,

```
(match x
    (1          expr1)
    ((cons a d) expr2)
    (y          expr3)
    (_          expr4))
```

There are three non-wildcard patterns, so there should be three if expressions. Since we want to execute the case corresponding to the first pattern which matches, these if expressions should be nested.

The first if expression should test whether `x` matches the pattern `1`, which is a simple equality check: `(= x 1)`. If this condition is true, we execute `expr1`, otherwise we move on to the next pattern.

For `x` to match the second pattern, `(cons a d)`, we must have `(pair? x)`. Moreover, in executing `expr2`, we need `a` to be bound to the `car` of `x` and `d` to be bound to the `cdr` of `x`, so we must introduce `let` bindings accordingly.

The third pattern is an identifier pattern which always matches, so the condition here is simply `#t`. As above, though, we must introduce a let binding to ensure `y` is equal to `x`.

At the end, we have the following final result:

```
(if (= x 1)         ; literal pattern 1 matches if x = 1
    expr1
    (if (pair? x)   ; cons pattern matches if x is a pair
        ; we must bind a and d to their appropriate values
        (let ((a (car x)) (d (cdr x)))
             expr2)
        (if #t      ; an identifier pattern always matches
            ; we must bind y to its appropriate value
            (let ((y x))
                 expr3)
            expr4)))
```

You might also notice that since the identifier pattern always matches, we can simplify the above expression without changing the way the program is intended to behave. However, for the purpose of this assignment, please **do not desugar away redundant patterns**.

Note that the MandarinSugar expression `(match x (_ expr))` should just desugar into `expr`.

Remark: In the above, we have assumed `x`, `lst`, `expr`, `expr1`, etc. to be basic expressions. If they were MandarinSugar expressions, they would themselves need to be desugared!

Invalid match expressions

Although the grammar does not explicitly forbid this, repeating the same identifier in a single `cons` pattern will **not be allowed** in a MandarinSugarExpr. In other words, something like

```
(match x
    ((cons a a) a)    ; not a valid pattern!
    (_          324))
```

is an **invalid** MandarinSugar program, and will not be tested. Your code does *not* need to handle this case. (Though this case is not terribly difficult to handle. Come to Randy if you have an implementation for this case!)

Repeating multiple, redundant patterns is acceptable; your code should be able to properly desugar something like this:

```
(match x
    ((cons a b) a)
    ((cons a b) b)   ; is a valid pattern!
    (_          324))
```

Our grammar also explicitly forbid recursive patterns like ones that languages like Racket support. For example, your code will *not* need to support patterns like this:

```
(match x
    ((cons a (cons b c)) b)    ; not a valid pattern!
    (_                   324))
```

Supporting these kinds of patterns is beyond the scope of the course, but if you are interested, refer to chapters 4-5 of the book The implementation of functional programming languagesPermalink.

How to get started

The best way to get started is by following the strategies provided in the lab 2 handout.

This means starting by understanding (1) what your input looks like, and (2) understanding what the output should be for these inputs. Trace through and understand the provided cases first. Write and verify test cases (and remember that these test cases should *not* be shared with anyone else!).

Maintain running code at all times, and consider writing prototype code in a separate file. Work on the different cases separately and incrementally, in the order we discussed them:

1. convert n-ary `+` expressions into binary `+` expressions.
2. convert `match` expressions into `if` expressions: start by handling the case where there is exactly one pattern and one wildcard into `if` expressions, then consider how to break down the problem into parts in the case with multiple patterns.
3. convert n-ary function creation and calls into binary ones: again start by handling the simplest cases before moving on to more complex ones, always thinking about how to break down the problem into parts.
4. consider what else needs to be done, but was not explicitly stated in this handout.

In order to break down the problem into small, manageable parts, remember to use structural recursion like in lab 2. As always, *trust the recursion.* You can use helper functions freely.

More advice on getting started

In this course, it is normal to spend a long time thinking/prototyping. It's okay to work with examples for a long time. It's okay to think for a long time about a very short piece of code. It's also okay to write and then throw out a bunch of prototype code as you understand the problem better.

Write code incrementally, and test your code incrementally.

Finally, when coding, keep track of the *type* of quantities you are working with: are you working with a MandarinBasic expression that has already been desugared, or a MandarinSugar expression yet to be desugared? Are you converting a MandarinSugar expression into a MandarinBasic expression, or another (simpler) MandarinSugar expression?

And again, remember to **trust the recursion**!

Even more advice

> Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis. - Alan J. Perlis

Beyond practising good programming strategies, this assignment will be a test of three main things: (1) do we understand syntax/grammar? (2) do we understand structural recursion and how to trust the recursion? (3) how complete was our case analysis/testing? Understanding of (1) and (2) should get you to the ~70% range in terms of grade. But true mastery of the materials is demonstrated by (3).

For example, consider: is this a valid MandarinSugar expression? Why or why not? Can you make even more complex examples of MandarinSugar expressions, e.g., where `w` is replaced by another MandarinSugar expression?

```
(match (+ w (match x (1 x) (_ 0)))
  ((cons a b) (lambda (m) a))
  (_          z)))
```

If you followed the strategy that we use in lab 2, your code should be able to handle these types of more complex expressions without excessive code.

Submission and Instructions

Submit the files `a1.rkt` on Markus. On assignments, there will be a small number of tests available for you on Markus. Those tests are meant to help you verify that your code is runnable on the Markus autotesting machine. Please test your code early. Markus sometimes hangs when running tests, particularly if you submit code that does not terminate. If so, you will not be able to continue to submit additional tests until the test times out.

The Markus tests available to you will not tell you whether your code is correct. That is up to you, since testing is an important skill to develop. Please do not share the tests that you write with anyone else. If you are not sure about the expected output for a particular input, please ask on Piazza.

For all labs and assignments:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" unless explicitly told otherwise.
- In Racket, you may not use any iterative or mutating functionality unless explicitly allowed. Iterative functions in Racket are functions like `for` and `for*`. Mutating functions in Racket have a `!` in their names, like `set!`. If use the materials discussed in class and do not go out of your way to find these functions, your code should be okay.
- The `(provide ...)` code in your files are very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.

Breaking any of these above rules can result in a grade of 0.

Moreover, code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero. Please make sure to run and test all of your code before your final submission.

You are permitted (and encouraged!) to write helper functions freely.