1 Project 2

Due: Mar 11, before midnight.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes how it can be submitted.

1.1 Aims

The aims of this project are as follows:

- To get you to think recursively.
- To expose you to more Scheme programming.

1.2 Requirements

Implement all except the last of the following functions in a file `prj2-sol.scm` (the file may also contain any necessary auxiliary functions).

The last function should be implemented in a file called `gram.scm`.

Your solution may not use any of Scheme's imperative features; i.e., you may not use any Scheme built-in with a name ending with the `!` character.

Questions 1-5 deal with implementing unary arithmetic based on *Peano's definition* of the *natural numbers*:

- `0` is a natural number.
- If `n` is natural number, then so is its successor.

Hence we will represent natural numbers with a unary notation in Scheme syntax:

- The Scheme symbol `'z` will be used to represent the natural number 0.
- If n is the representation of natural number N, then the representation of the successor of N will be `(cons 's` *n*`)`, alternately `'(s .` *n*`)`

Example representations would be `'z` for 0, `'(s . z)` for 1, `'(s s s . z)` for 3. In general, the representation for natural number N will contain N occurrences of `s`.

1. Write a function `(int->unary n)` which when given a non-negative integer argument `n`, produces its unary representation.

   Examples:

   ```
   > (int->unary 0)
   'z
   > (int->unary 4)
   '(s s s s . z)
   > (int->unary 6)
   '(s s s s s s . z)
   >
   ```

   See the provided tests for more examples.

   Hint: Recurse on `n`. *5-points*

2. Write a function `(unary->int n)` which when given a number `n` in unary representation, produces the corresponding integer.

   Examples:

   ```
   > (unary->int '(s s s s . z))
   4
   > (unary->int '(s s s s s . z))
   5
   > (unary->int 'z)
   0
   >
   ```

   See the provided tests for more examples.

   Hint: Recurse on `n`. *5-points*

3. Define a function `(unary-add m n)` which when given two numbers `m` and `n` in unary representation, produces the unary representation of their sum. You implementation may not use your solutions to (1) and (2) directly or indirectly; instead it should be based on the hint below.

   Examples:

   ```
   > (unary-add '(s s . z) '(s s s . z))
   '(s s s s s . z)
   > (unary-add 'z '(s s s . z))
   '(s s s . z)
   >
   ```

   See the provided tests for more examples.

   Hint: Recall that addition is merely repeated incrementing and that $(1+m)+n = 1+(m+n)$. You may also want to consider the definition of `my-append` in the slides. *10-points*

4. Define a tail-recursive function `(unary-add-tr m n)` which when given two numbers `m` and `n` in unary representation, produces the unary representation of their sum. All recursion must be tail recursive. Your implementation may not use your solutions to the previous exercises directly or indirectly; instead it should be based on the hint below.

   See the provided tests for examples.

   Hint: $(1 + m) + n = m + (1 + n)$. *10-points*

5. Define a function `(unary-mul m n)` which when given two numbers `m` and `n` in unary representation, produces the unary representation of their product. Your implementation may use either of your earlier functions (3) or (4).

   Examples:

   ```
   > (unary-mul (int->unary 3) (int->unary 0))
   'z
   > (unary-mul (int->unary 3) (int->unary 4))
   '(s s s s s s s s s s s s . z)
   > (unary->int (unary-mul (int->unary 3) '(s s . z)))
   6
   >
   ```

   See the provided tests for more examples.

   Hint: Recall from elementary school, that multiplication is nothing but repeated addition. *10-points*

6. Write a function `(contains-empty-lists-only? ls)` which returns `#t` if all entries in `ls` are `'()`, `#f` otherwise.

   See the provided tests for examples.

   Hint: Recurse on the list `ls`. *5-points*

7. Write a function `(split-firsts ls)` which when given a non-empty proper list `ls` of proper sub-lists, return a pair. The first element of the returned pair is a proper list containing the first elements of each sub-list (if a sub-list is empty, the corresponding element is returned as '()). The second element of the returned pair is a proper list containing the remaining elements of each sub-list (again, if a sub-list is empty, then corresponding element is returned a '()). It follows that each element of the returned pair will have length equal to that of ls.

   See the provided tests for examples.

   Hint: Add two additional arguments (with default value `'()`) to accumulate each element of the returned pair. Recurse through the elements of `ls`. At each step of the recursion look at the first element in `ls`; if it is a pair, add its head and tail to the accumulating arguments and recurse; if it is not a pair (it must be `'()`), recurse with `'()` added to the accumulating arguments. *15-points*

8. Write a function `(list-tuples ls)` which when given a proper-list containing n proper-list's, returns a list of n-tuples formed by picking up elements of each tuple from each of the n-lists. If a particular list does not have any elements, then its tuple entry should be '().

   Examples:

   ```
   > (list-tuples '((a b) (1 2) (x y)))
   '((a 1 x) (b 2 y))
   > (list-tuples '((a b c) (1) (x y)))
   '((a 1 x) (b () y) (c () ()))
   ```

   See the provided tests for more examples.

   Hint: Use the previously written `split-firsts` function to recurse through each element of `ls` in parallel, bottoming out the recursion when `ls` does not contain any non-empty sublists. *10-points*

9. Write a function `(parse gram toks)` which returns `#t` iff the tokens list `toks` are a sentence in the language defined by grammar `gram`; return `#f` otherwise.

   The token list `toks` consists of a list of 2-element lists; the first element of each two element list is a Scheme symbol (starting with an uppercase letter) representing the token kind and the second element is a Scheme string representing the token lexeme.

   The grammar is represented as a nested list structure. Specifically:

   - A grammar is represented as a proper list of rules.
   - Each rule is represented as a Scheme pair.
     - The first element of a rule pair is a non-terminal symbol.
     - The second element of a rule pair is a list of rule RHSs.
   - A rule RHS is represented as a (possibly empty) proper list of terminal and non-terminal symbols.

   Terminal symbols are represented as Scheme symbols starting with an uppercase letter whereas non-terminal symbols are Scheme symbols which are not terminal symbols. *30-points*

   Hint: See the `IFDEF-GRAM` in the provided ifdef.scm for an example grammar.

   Implement the top-level `parse` function using auxiliary parsing functions, where each auxiliary parsing function will be given a token list to parse. Each auxiliary parsing function should return `#f` if it is unable to parse a prefix of the token list. OTOH, if it is able to parse a prefix of the token list, it should return the rest of the token list.

Note that the code can be simplified greatly because Scheme's `and` and `or` functions not only have short-circuit semantics, but also return the value of the last operand evaluated.

```
> (and #f (/ 1 0))
#f
> (and (> 1 2) (/ 1 0))
#f
> (and (< 1 2) (+ (* 2 3) 3))
9
> (or (+ 1 2) (+ (* 2 3) 3))
3
> (or (> 1 2) (+ (* 2 3) 3))
9
>
```

(This behavior is similar to that in many untyped languages like Python, JavaScript, Perl and Ruby).

Hence it possible to implement `parse` using the auxiliary parsing functions as possible:

- An attempt to `parse` token list `toks` according to grammar `gram` should succeed (return `#t`) if no tokens are leftover after parsing the start symbol of the grammar.

  This can be achieved checking that no tokens are left over after calling an auxiliary parsing function `(parse-nonterm gram toks nonterm)` where `nonterm` will be the start symbol which will be `(caar gram)`.

- To parse a non-terminal, it is necessary to lookup all the RHS's for that non-terminal in the grammar and then attempt each RHS on the token list.

  Specifically, `(parse-nonterm gram toks nonterm)` should simply delegate to `(parse-rhss gram toks rhss)` where `rhss` is a list of RHS's for `nonterm`.

- To parse a token list via multiples RHS's, it is necessary to try parsing the token list using the first RHS; otherwise try parsing it using the remaining RHS's.

  Specifically, `(parse-rhss gram toks rhss)` will fail (return `#f`) if there are no RHS left in `rhss`. Otherwise, it can succeed if it can parse `toks` using the first RHS in `rhss`, or if it can parse `toks` using the remaining RHS's in `rhss`.

Note that Scheme's `or` has the necessary semantics.

- To parse a token list via a single RHS it is necessary that a prefix of the token list match **all** the symbols in the RHS.

  Specifically, `(parse-rhs gram toks rhs)` will succeed if `rhs` is empty (in which case it will return `toks`), or it is possible to parse the first symbol in `rhs` and parse the rest of `rhs` using the tokens leftover after parsing the first symbol. Note that `parse-rhs` should return the tokens leftover after parsing the rest of the `rhs`.

- To parse a grammar symbol:

  - If the grammar symbol is a terminal, then the first token in the token list must match.
  - If the grammar symbol is a non-terminal then it must be necessary to parse that non-terminal using one of its RHSs.

  Specifically, `(parse-sym gram toks sym)` will return the tail of `toks` if `sym` is a terminal and `toks` is not empty and the `kind` of the first element in `toks` matches `sym`. When `sym` is a non-terminal, it will simply return the result of `(parse-nonterm gram toks sym)`.

1.3 Provided Files

You should use the provided prj2-sol directory as a starting point for your project by copying it into your `i471?` directory. It contains the following files:

prj2-sol.scm A skeleton file which contains skeleton functions for all except the last function you are required to run.

The file is executable so that it can be executed directly via the command line as a racket script. It also contains unit tests for each of the functions you are required to implement.

gram.scm A skeleton file which contains skeleton functions for implementing the `parse` function you are required to implement.

ifdef.scm A driver program for the `parse` function. Specifically, it:

- Provides a Scheme representation of a `IFDEF-GRAM` grammar compatible with the grammar from your previous project.
- Provides a Scheme scanner compatible with your previous project.
- Provides a `main` function which is used when `ifdef.scm` is invoked from the command-line:

  1. If the first argument is `scan`, then it transforms its standard input to JSON tokens on standard output according to the specifications for your previous project.

In fact, it is possible to run the *Project 1* scan tests:

```
$ ~/cs471/projects/prj1/extras/run-all-tests.sh \
       scan-tests-only
 scan ifdef-elif2.defs ... ok
 scan ifdef-else.defs ... ok
 scan ifdef.defs ... ok
 scan nested.defs ... ok
 scan parse0.defs ... ok
 scan scan.defs ... ok
 scan scan0.defs ... ok
 scan text.defs ... ok
```

2. If the first argument is `parse`, then it will invoke the `parse` function to be implemented by you with the `IFDEF-GRAM` and the token list resulting from scanning standard input using the aforementioned scanner and output the result on standard output.

scan.sh A trivial shell script to meeting the requirements from your previous project to make it possible to test the scanner using the test scripts from that project.

The extras directory contains a shell script test-ifdef-parse.sh which can be used to test your `parse` function using the test data from your previous project. You can run the script from the directory containing your `gram.scm` file:

```
$ ~/cs471/projects/prj2/extras/test-ifdef-parse.sh
usage: /home/.../extras/test-ifdef-parse.sh TESTS_DIR
$ ~/cs471/projects/prj2/extras/test-ifdef-parse.sh \
    ~/cs471/projects/prj1/extras/tests
parse ifdef-elif2.defs ... ok
parse ifdef-else.defs ... ok
parse ifdef.defs ... ok
parse nested.defs ... ok
parse parse0.defs ... ok
parse text.defs ... ok
parse no-sym-err.defs ... ok: failed as expected
parse top-elif-err.defs ... ok: failed as expected
parse top-else-err.defs ... ok: failed as expected
parse top-endif-err.defs ... ok: failed as expected
parse unclose-err.defs ... ok: failed as expected
$
```

1.4 Submission

Before submitting your project, update your README to specify the status of your project. Document any known issues.

Submit using a procedure similar to that used in your previous project.