CSE 413 23sp Final Exam, June 5, 2023

Name ___________________________________ **UW NETID** _______________@uw.edu

There are 9 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, no electronic devices, signal flags, tin-can telephones, smoke signals, telepathy, tattoos, implants, or other signaling or communications apparatus allowed. However, you may have a two 5x8 notecards with any hand-written notes you wish on both sides.

Style and indenting matter, within limits. We're not overly picky about details like an extra or a missing parenthesis, but we do need to be able to follow your code and understand it.

If you have questions during the exam, raise your hand and someone will come to you. ***Don't*** leave your seat. Please **wait** to turn the page until everyone has their exam and you have been told to begin.

Advice: The solutions to several of the problems are quite short. Don't be alarmed if there is a lot more room on the page than you actually need for your answer.

More gratuitous advice: Be sure to get to all of the questions. If you find you are spending a lot of time on a question, move on and try others, then come back to the question that was taking the time.

There is an extra blank page at the end if you need more space for an answer. Be sure to indicate that your answer is continued on the extra page, and be sure to indicate on the extra page which question the answer refers to.

After the blank pages with extra space for answers, there is a sheet of paper with MUPL information. You **should remove this page from the exam** and use it for reference while working those problems.

Relax. ☺ You are here to learn.

1 (streams)	/ 14
2 (MUPL)	/ 15
3 (ruby)	/ 15
4 (regexp/DFA)	/ 14
5 (CFGs)	/ 14
6 (calculator)	/ 14
7 (typing)	/ 8
8 (memory)	/ 4
9 (free)	/ 2
Total	/ 100

CSE 413 23sp Final Exam, June 5, 2023

Question 1. (14 points) Streams and thunks – the question that wasn't on the midterm! Recall that in Racket we can represent a stream as a thunk that, when called, produces a pair (*first-element-of-stream* **.** *thunk-that-represents-the-remainder-of-the-stream*). For example, the infinite stream of natural numbers 1, 2, 3, … beginning with 1 can be defined as follows:

```
(define nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
```

Complete the definition of function `get` below such that it returns the nth element of the stream `st` . For example, `(get nats 1)` should evaluate to 1; `(get nats 10)` should evaluate to 10. Sample solution: 3 lines of Racket code. Your solution, of course, can be longer or shorter but this might give you an idea of what to expect.

```
;; return the nth item in stream st

(define (get st n)    ;; write rest of the function definition below
```

Question 2. (15 points) The MUPL Memorial Question (feel free to go on to other questions and come back to this later). Also, DON'T PANIC! The question description is longer than the answer.

Several pages of MUPL information appear on the last sheet of this exam for you to use as reference material while you answer this question. They are taken from the assignment (hw5), giving the details of the MUPL language, and the starter code for hw5, including the Racket struct definitions used to implement MUPL. You should remove that sheet of paper from the exam and use it while you work this problem.

We would like to add a new `abs` expression to our MUPL interpreter. The `abs` expression is defined as follows:

- If *e* is aMUPL expression, then `(abs e)` is a MUPL expression. The value of `(abs e)` is either the value *e* if *e* is a non-negative MUPL integer (i.e., *e* ≥ 0), or is the value -*e* if *e* is a negative MUPL integer. If *e* is not a MUPL integer (i.e., something not created by `(int ...)`), then execution should be terminated with a suitable error message.

Modify the MUPL interpreter `eval-under-env` function by adding this new kind of expression to the MUPL language. Write code below to add an interpreter case for `abs` to `eval-under-env`. You should assume the following structure has been added to MUPL to represent this new expression:

```
(struct abs (e) #:transparent) ;; if e≥0 then e else -e
```

Reminder: The Racket function `(error "message")` can be used to terminate evaluation with the given message. (Hint: Sample solution length is about 6 lines of code.)

```
(define (eval-under-env e env)
  (cond [(var? e)
         (envlookup env (var-string e))]
        ;; remaining cases omitted
        ;; CHANGE add your code for abs below
```

CSE 413 23sp Final Exam, June 5, 2023

Question 3. (15 points) A bit of Ruby. Write a small Ruby program that will open a text file whose name is given as a command-line argument, then read the file and print the number of lines and number of words in the file. For example, suppose the Ruby code is contained in the file `wc.rb`, and the file `be.txt` contains the following:

```
to be or not to be
to do is to be
to be is to do
do be do be do
```

Then the command `ruby wc.rb be.txt` should print the following:

```
4
21
```

You should assume that words are any non-blank sequences of characters in an input line that are separated by one or more blanks. You should avoid reading and storing the entire input file before processing it – process each line of input as you read it.

Some possibly useful facts:

- `File.open(filename)` can be used to open a file.
- The command-line arguments to a program can be accessed as `ARGV[0], ARGV[1], …`
- If `s` is a string, `s.trim` is a copy of `s` with any leading or trailing blanks omitted.
- The string `split` method returns an array of the blank-separated words in a string. Example: `"  one two three ".split` returns `["one", "two", "three"]`. If the entire string consists of blanks or has no characters in it, split will return an empty array `[]`.

Write your code below or on the next page. Sample solution is about 10-12 lines, including trivial initialization statements like `nlines = 0`.

Question 3. (cont.) Additional space for answer if needed.

CSE 413 23sp Final Exam, June 5, 2023

Question 4. (14 points) Regular expressions and DFAs. In the US, dollar currency amounts are normally written with a leading $, one or more digits, optionally with one or more leading 0's, giving the number of dollars. If there are more than 3 digits in the dollar part, they are separated into groups of three digits with commas, counting from the right.

Examples of legal strings using these rules: $1, $01, $1,234, $17, $01, $00,017, $1,024, $8,820,000,000 .

Examples of illegal strings: 12 (no leading $), $12,34 (must be a group of three digits after each ","), $1234 (leading digits not separated into groups of 3 with commas), $1.25 (".25" factional amount not allowed).

(a) (7 points) Give a regular expression that generates all strings that are legal currency amounts according to these rules. (Hint: you may want to work on both parts (a) and (b) at the same time and use the solution to one to inform your solution to the other.)

Fine print: You must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*|*s* , *r** , *r*+ , *r*?, character classes like [a-cxy] and [^aeiou], abbreviations *name*=*regexp*, and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various programming language libraries like Python or Ruby or Linux commands are allowed.

(b) (7 points) Draw a DFA that accepts all valid currency strings described above and generated by the regular expression from part (a).

(additional space for answer, if needed, on the next page)

Question 4. (cont) Additional space for answer if needed.

CSE 413 23sp Final Exam, June 5, 2023

Question 5. (14 points) Context-free grammars. Consider the following grammar for arithmetic expressions involving + and *:

> *exprs* ::= *exprs* + *expr* | *exprs* * *expr* | *expr*
> *expr* ::= x

(a) (7 points) Is this grammar ambiguous or unambiguous? If it's ambiguous, show that by giving two different parse trees or two different leftmost (or rightmost) derivations for some string generated by the grammar. If it's not ambiguous, give an informal argument why it is not.

(b) (7 points) Does this grammar properly capture the normal precedence of arithmetic operators * and + (i.e., * should have higher precedence than +)? If it does, give a brief argument as to why, if not, give an example that shows where it fails to capture the correct precedence relationship.

Question 6. (14 points) This question concerns the calculator language from the last two assignments. If you recall, the grammar for the calculator language was as follows:

program ::= *statement* | *program statement*
statement ::= *exp* | *id* = *exp* | `clear` *id* | `list` | `quit` | `exit`
exp ::= *term* | *exp* + *term* | *exp* − *term*
term ::= *power* | *term* * *power* | *term* / *power*
power ::= *factor* | *factor* ** *power*
factor ::= *id* | *number* | (*exp*) | `sqrt` (*exp*)

We would like to add a random number generator to the language by adding a new `random()` function to the language. The meaning of this new function is the obvious one – it evaluates to a new random number each time it is executed.

(a) (3 points) We propose to add this new function to the language by adding the following additional production to the grammar rule for factor:

factor ::= … | `random ( )`

Does this additional rule make the grammar ambiguous? If so, give an example that shows that it does. If not, give a short, but convincing explanation of why not.

(b) (3 points) What changes or additions are needed in the calculator's scanner and in the `Token` class to add this new `random` function expression?

(continued next page)

Question 6. (cont.) (c) (8 points) Below, write the additional Ruby code needed in method `factor` to parse and evaluate this new `random` function. Your answer should be guided by your answers to the previous parts of the question, but adjusted as needed for use in a recursive-descent parser. You should make the following assumptions (if needed):

- All methods in the parser for components of expressions, like `exp`, `term`, `power`, or `factor` return the value of the expression that they parse and evaluate.
- The ruby method `rand` returns a new random number each time it is called. You should return the result of calling this method each time `random()` is evaluated in an expression in order to return a new random number result.
- The scanner and `Token` class have been modified as described above in your solution to part (b). You can call `next_token` to return the next `Token` input object whenever you need it.
- The `kind` method of a `Token` object returns a string that contains the literal text that represents the token class, like ")", or "+", or "exit". It returns "ID" or "NUMBER" for an identifier or number, respectively. For an id or number token, the `value` method returns the specific identifier or number.
- There is a global variable named `$current_token` that contains the next unprocessed `Token` read from the input at all times. Your code **must** update this variable appropriately as it parses the input and it **must** always contain the next unprocessed token. There is no "look-ahead" or "peek" function in the scanner.
- Parser functions like `exp` and `term` exist to parse each grammar nonterminal and return its value, if any. These functions have no parameters and expect `$current_token` to be the first token in the grammar nonterminal they are parsing when they are called.
- You may assume there are no syntax errors, missing or extra tokens or other errors in the calculator input.

Write the additional code that needs to be added to `factor` below. If needed, state any extra assumptions you need to make in your solution. Do not worry too much about details – as long as your intent is clear and correct you will receive credit.

(additional space for your answer on the next page)

Question 6. (cont.) Additional space for your answer to part (c) if needed.

CSE 413 23sp Final Exam, June 5, 2023

A few short answer questions to wrap up. For each of these questions, please keep your answers **brief** and **concise**. A couple of sentences should usually be enough.

Question 7. (8 points, 4 each) Ruby is a dynamically typed language ("duck typing"). Languages like Java and C++ are statically typed languages.

(a) Describe one distinct advantage that static type systems have over dynamic typing.

(b) Describe one distinct advantage that dynamic typing has over static type systems.

Question 8. (4 points) Reference counting is one system for automatically reclaiming memory resources when they are no longer in use. Yet implementations of languages like Java use garbage collection algorithms instead to reclaim free storage on the heap. Describe the most important reason that reference counting cannot be used instead of garbage collection as a general solution to reclaim storage in Java.

CSE 413 23sp Final Exam, June 5, 2023

Question 9. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

Draw a picture of something you plan to do this summer.

Have a great summer break and best wishes for the future!
The CSE 413 staff

CSE 413 23sp Final Exam, June 5, 2023

Additional space for your answers, if needed. Please be sure to write the question number below, and indicate on the original page that the answer is continued on the last page so we don't miss it.