Project 2. A Spreadsheet Application with Static Type Inference

In this project, we'll continue building the spreadsheet software. We will work in Racket, and build a static type inferencer and checker for the spreadsheet language to make sure that the spreadsheet values are appropriately typed. Moreover, we will use miniKanren to *infer* the types of columns.

The project is a chance for you to demonstrate that you can:

1. Use racket to solve type checking problems.
2. Use miniKanren and constraint logic programming to solve problems like type inference and type inhabitation.
3. Use good programming style while writing functional code in Racket.

You may work with one partner to write the code for this project. However, each student must independently write a description of how the code works, and their contribution to the project.

As in Project 1 you may share some test cases with other students. However, tests must be shared on Piazza, and each team can only share up to 3 test cases.

Starter code

Download this file from Quercus files:

- `Projects/p2/p2-base.rkt`
- `Projects/p2/p2-spreadsheet.rkt`
- `Projects/p2/p2-starter-tests.rkt`
- `Projects/p2/mk.rkt`

Submission

There will be three Markus assignments set up:

- "p2": Submit the file `p2-base.rkt` and `p2-spreadsheet.rkt` Do not submit `mk.rkt`, since we will be supplying our own. Only one person in each team needs to submit this file.
- "p2-writeup": Submit a PDF file `p2-writeup.pdf` containing your written explanation. Each team member must submit this file separately.

If you choose to work with a partner for an assignment, you must form a group for "p2" on MarkUs. One student needs to invite the other to a group. You should declare a partnership well before the deadline (there is no downside of doing so). Ask your instructor for help if you're having trouble forming a group.

Note that "p2-writeup" deadline is set up two days later than "p2", to avoid double-deducting grace tokens. You may submit p2-writeup up 48 hours past the p2 deadline, but we will not accept writeups submitted any later than that (i.e., you cannot use any grace tokens to further extend the writeup deadline).

A statically typed spreadsheet application

In project 1, we wrote a spreadsheet application that fills in the values of computed columns in a spreadsheet like this one:

Column:	id	name	age	voter	name2	year-until-100
Formula:				`(>= age 18)`	`(++ name name)`	`(- 100 age)`
Values:	1	"adam"	12	#f	"adamadam"	88
	2	"betty"	15	#f	"bettybetty"	85
	3	"clare"	18	#t	"clareclare"	82
	4	"eric"	49	#t	"ericeric"	51
	5	"sam"	17	#f	"samsam"	83

In our Haskell version of the spreadsheet, each spreadsheet value had an implicit associated *type* represented using value constructors: number, string, boolean, and error.

In this part of the project, we will use Racket instead. At the same time, we will add some static type checking to our spreadsheet language DeerLang. In other words, we should be able to see that the expression `(>= x "hello")` will generate a type error without evaluating any code.

Task 1. A Simple Type Inferencer

Recall the spreadsheet language DeerLang definition from project 1; we'll use DeerLang again in this project:

```
<expr> = VALUE
       | ID
       | '(' BUILTIN <expr> ... ')'              ; builtin function call
       | '(' 'lambda' '(' ID ... ')' <expr> ')' ; function definition
       | '(' <expr> ... ')'                      ; function expression

VALUE    = ... numbers, strings, booleans ...
BUILTINS = ... described below ...
ID       = ... valid identifiers ...
```

For task 1 of the project, we'll write a *type inferencer* that takes an expression in DeerLang and determines the type of the result of evaluating that expression. Our language will support three data types: numbers, strings, and booleans. We'll use the Racket symbols `'num`, `'str` and `'bool` to represent these types.

We will be implementing the function `typeof` in the file `p2-base.rkt`. This function takes an expression and a **type environment** (to be explained soon), and returns the type of the expression. Here are some examples of what our function should return:

```
> (typeof 3 '()) ; ignore the second argument for now
'num
> (typeof "hello" '())
'str
> (typeof #t '())
'bool
> (typeof '(= 3 3) '())
'bool
```

The *type environment* will be an association list that stores the types of identifiers. In other words, the type environment is a list of pairs, where the `car` of each pair is the identifier name, and the `cdr` is its type. Using this type environment, we can identify the types of expressions involving identifiers:

```
> (typeof a '((a . num) (b. str)))
'num
> (typeof b '((a . num) (b. str)))
'str
> (typeof '(+ a a) '((a . num) (b. str)))
'num
```

(You might be wondering why we are using an association list with O(n) lookup time, rather than a hash map with O(1) lookup time. The reason is that we will be turning the function `typeof` into a miniKanren *relation* `typeo` in the next part of the project. Since miniKanren only works with simple Racket list structures, we'll consistently use association lists for our type environment.)

Like before, the spreadsheet application has several builtin operations. We added two more operations to the list from project A. The builtin definitions and types are shown in the table below:

Racket Symbol	Racket Function	Arguments	Return Type
`'+`	`+`	2 numbers	number
`'-`	`-`	2 numbers	number
`'*`	`*`	2 numbers	number

Racket Symbol	Racket Function	Arguments	Return Type
`'/`	`/`	2 numbers	number
`'>`	`>`	2 numbers	boolean
`'=`	`equal?`	2 numbers	boolean
`'>=`	`>=`	2 numbers	boolean
`'++`	`string-append`	2 strings	string
`'!`	`not`	1 boolean	boolean
`'num->str`	`number->string`	1 number	string
`'len`	`string-length`	1 string	number

Types of builtin operations and functions will be represented as a list, with the first element being a list of argument types, and the second element being the return type. Here's how we will represent function types:

```
; type of the builtin =
'((num num) bool) ; takes 2 numbers and returns a boolean
; type of the builtin !
'((bool) bool)    ; takes 1 boolean and returns a boolean
```

Identifiers may also represent functions, so you can have function types in the type environment:

```
> (typeof 'f '((f . ((num) num)) (g . ((num) str))))
'((num) num)
> (typeof '(g (f 3)) '((f . ((num) num)) (g . ((num) str))))
'str
```

The `typeof` function should also be able to recognize type errors, and return the symbol `'error`.

```
> (typeof '(+ 3 "hello") '())
'error
> (typeof '(f (g 3)) '((f . ((num) num)) (g . ((num) str))))
'error
```

Your task is to complete the function `typeof` that can return the type of a DeerLang expression **that does not not involve function definitions.** You also can assume that we won't test with free identifiers (identifiers whose types are not in the type environment). Here are some example expressions that we don't expect you to be able to handle for task 1:

```
; NOT TESTED: contains lambda expressions
(typeof '(lambda (x) (+ x x)) '())
(typeof '((lambda (x) x) 3) '())

; NOT TESTED: contains free variabels
(typeof '(+ x 3) '())
(typeof '(+ x y) '((x . 3)))

; NOT TESTED: builtins are not identifiers
(typeof '+ '())
```

We recommend starting by writing `typeof` to handle literal values, then identifiers. Then, handle builtin functions, either one at a time or all at once. Finally, handle function calls. Alternatively, you can treat builtin functions as regular function calls. This second approach requires more thinking time, but less coding and debugging time.

Task 2. A Type Inferencer Relation in miniKanren

Handling function definitions is difficult in a functional setting. Consider the expression `((lambda (x) x) 3)`. The function definition does not actually constrain the type of the parameter `x`! However, a function like `(lambda (x) (+ x x))` already fully constrains its parameter and output type. In other words, the type of a function parameter could be constrained by the function body *and* by the argument type.

Problems like type inferencing where constraints can come from multiple places and in somewhat complicated ways is much more easily solved using logic programming. To that end, we'll write a **relation `typeo`** instead of a function.

The `typeo` relation will have three arguments: the DeerLang expression (*including* function definitions!) the type environment (association list), and the output type (`'num`, `'str`, or `'bool`). The relation will succeed if the output type matches the input.

Here are some examples of what running the `typeo` relation could look like. We use a logic variable `out` to represent the output type of an expression, and query the value of `out`:

```
> (run 1 (out) (typeo '(> 4 5) '() out))
'(bool)
> (run 1 (out) (typeo '(> 4 "hi") '() out))
'()
> (run 1 (out) (typeo '((lambda (x) x) 3) '() out))
'(num)
```

Step 1 Handle constants, identifiers, and built-in expressions. Start by converting the `typeof` function into the `typeo` relation. For example, you might have had these lines of code in your `typeof` function:

```
(define (typeof expr env)
  (cond
    [(number? expr) 'num]
    ...))
```

The relational form `typeo` of the function explicitly describes the constraint on the output type.

```
(define (typeo expr env type)
  (conde
    ((number? expr) (== type 'num))
    ...))
```

You may find the relations `numbero`, `stringo`, `boolo`, and `symbolo` useful. You may also need to create fresh logic variables when handling built-ins.

Step 2 Handle function calls. After step 2, your `typeo` relation should be the relational form of your `typeof` function. This step (handling function calls) should be similar to handling builtins, except that you won't know the length of your argument list.

You may find it useful to implement a helper relation `type-listo` that takes a list of expressions, a type environment, and a list of types corresponding to those expressions.

Step 3 Handle function definitions. This step will be the most challenging. You will need to manipulate the type environment here, and possibly add logic variables in the type environment. You might also find it useful to write helper relations, including relations like `lookupo` and `appendo` that we wrote during lecture.

Task 3. Type Checking a Spreadsheet

With `typeof` and `typeo` available to us, we can type check an actual spreadsheet that includes type annotations. The input to the type checker will be similar to project 1, but **each column will have a type annotation**:

```
<spreadsheet> = '(' 'spreadsheet'
                   '(' 'def'     (ID <expr>  ) ... ')'
                   '(' 'columns' (ID <column>) ... ')'
                ')'
<column> = '(' 'values'   <type> VALUE ... ')'
         | '(' 'computed' <type> <expr> ')'
<type> = 'num' | 'str' | 'bool'
```

Here is an example spreadsheet:

```
(define sample-spreadsheet
    '(spreadsheet
       (def (voting-age 18)
            (concat (lambda (x y) (++ x y)))
            (canvote (lambda (x) (>= x voting-age))))
       (columns
         (id    num (values 1 2 3 4 5))
         (name  str (values "adam" "betty" "clare" "eric" "sam"))
         (age   num (values 12 15 18 49 17))
         (voter bool (computed (canvote age)))
         (name2 str (computed (concat name name))
         (year-until-100 num (computed (- 100 age)))))))
```

Complete the function `type-check-spreadsheet` in the file `p2-spreadsheet.rkt`. This function takes a spreadsheet, and returns a list of booleans representing whether the annotated type of each column is correct. Here is one example:

```
> (type-check-spreadsheet
    '(spreadsheet
       (def (voting-age 18)
            (canvote (lambda (x) (>= x voting-age))))
       (columns
         (name  str (values "adam" "betty" "clare" "eric" "sam"))
         (age   num (values 12 15 18 49 17))
         (voter bool (computed (canvote age)))
         (voter2 bool (computed (>= age name))))))
'(#t #t #t #f)
```

This output shows that type checking succeeded for the first three columns, but not for the last one. The last column uses the formula `(>= age name)` where `name` is a string rather than a number.

Note: If you wish for us to test this function with our own correct implementation of `typeo` and `typeof`, change the import in `p2-spreadsheet.rkt`. The instructions are at the top of that file.

As before, you can assume that:

- computed columns will only depend on columns appearing before it
- there will always be at least 1 column with raw data initially and that definitions+identifiers names will never be the same.
- definitions **will not be recursive or mutally recursive**
- all value columns have the same number of elements, consistent with the number of rows in the spreadsheet.

You may not assume that all value columns have the correct types. For example, the "name" field below has a number in the first row rather than the string, so the type checking for that column should fail. Any other computed column that depends on "name" will also fail the type check.

```
> (type-check-spreadsheet
    '(spreadsheet
       (def (voting-age 18)
            (canvote (lambda (x) (>= x voting-age))))
       (columns
         (name  str (values 4 "betty" "clare" "eric" "sam"))
         (age   num (values 12 15 18 49 17))
         (voter bool (computed (canvote age 18)))
         (voter2 bool (computed (>= age name))))))
'(#t #t #t #f)
```

Task 4. Synthesizing Programs (Type Inhabitor)

One of the neat features of miniKanren is that logic variables can go anywhere! In `p2-spreadsheet.rkt`, write a macro that takes an expression with the logic variable `BLANK` in the abstract syntax tree expression structure, the type you would like the expression to be, and the maximum number of results `n`, and returns a list of possible ways to replace `BLANK` in the expression so that it would satisfy the type checker. Here are some examples.

It is okay if your list of examples are different, or if they do not appear in the same order!

```
> (fill-in BLANK `(+ 3 ,BLANK) 'num 5)
'((_.0 (num _.0))                   ; BLANK can be any number
  ((+ _.0 _.1) (num _.0 _.1))       ; BLANK can be the sum of two numbers
  ((- _.0 _.1) (num _.0 _.1))       ; BLANK can be the difference of two numbers
  (((lambda () _.0)) (num _.0))     ; BLANK can be a call to a thunk that returns a number
  ((+ _.0 (+ _.1 _.2)) (num _.0 _.1 _.2))) ; BLANK can be the sume of three numbers...
```

In this example, we are filling in the blank in the expression `(+ 3 BLANK)` so that the resulting expression is of type `'num`. We are returning 5 results.

Note: If you wish for us to test this operation with our own correct implementation of `typeo` and `typeof`, change the import in `p2-spreadsheet.rkt`. The instructions are at the top of that file.

Grading

- 25 points: Task 1
- 30 points: Task 2
- 20 points: Task 3
- 5 points: Task 4
- 10 points: Style
- 10 points: Quality of the Written Explanation

Working with a partner You are encouraged to work with a partner. We expect both partners to contribute equally to the project, and to understand **all** the code that you submit.

Style Rubric (10 points) We'll be grading for the following:

1. (3 pts) Are you indenting your code appropriately? How many code indentation issues can the TA find in a 5 minute period?
2. (3 pts) Are you using `let*` to effectively reduce duplicate code? How many code duplications can the TA find in a 5 minute period?
3. (4 pts) Are you documenting your code effectively? Are you choosing good variable names? Can the TA choose a handful of variable names, and understand what the variables store?

Style: Indentation Appropriately indenting Racket code makes the code easier to read. If you are using Dr. Racket, you can use Dr. Racket to automatically indent your code for you. Here's an example where having the wrong indentation can make your code more difficult to read:

```
(define (foo x y z)
    (cond [(equal? x 1) y]
          [else (let* ([m (+ x y)]
          [n (+ y z)])      ; <-- looks like a branch of the "cond"!
          (+ m n))]))
```

Properly align your code so that sibling sub-expressions have the same indentation level. That way, your code becomes much easier to scan and understand.:

```
(define (foo x y z)
    (cond [(equal? x 1) y]
          [else (let* ([m (+ x y)]
                       [n (+ y z)]) ; better!
                  (+ m n))]))
```

Style: Repeated Code Good Racket code liberally use `let*` to define local variables to prevent repetition. In fact, expert Racket code have very little indentation, and a lot of `define` and `let*` expressions!

You should not be afraid to define new helper functions, and to use many, many local variables! It is better to use a local a variable called `defs` and assign it to a very simple expression like `(rest (second spreadsheet))`, than to have that expression repeat in many places.

Written Explanations (10 points) Submit a one-page written (max 500 words, Times New Roman, font size 12) explanation answering the below questions. The explanations should be written and submitted independently, even if you worked with a partner.

1. (3 pt) If working in a team, describe your contribution to the project. Otherwise, please declare that you wrote all the code.
2. (3 pt) Describe the strategy behind your `type-check-spreadsheet` function. Why can't we use the same strategy in the `typeof` function to handle function definitions?
3. (4 pt) You have two options for this question.

Option 1:

We used **macros** to implement the logic programming language features "-<" and "?-", and we wrote an **interpreter** to implement DeerLang in project 1. Explain the advantages and limitations in using these approaches to build language features. You will be graded for correctness and insight.

Option 2:

Many people in the programming languages research community strive to make their talks accessible to people who are new to the field. These are some of the talk recordings to give you a sense of what people are working excited about. Choose **one of the videos**, watch it, and write a paragraph describing what the talk is about. You will be graded on how convinced TA is that you watched the talk, based purely on your write up.

1. **On the Expressive Power of Programming Languages (2019)** https://www.youtube.com/watch?v=43XaZEn2aLc We discussed the Church-Turing thesis in the first lecture. But if all Turing-complete language have the same computational power, then how can we compare the expressiveness of different languages?
2. **Four Languages from Forty Years Ago (2018)** https://www.youtube.com/watch?v=0fpDlAEQio4 This talk is a gentle introduction to the "big ideas" in programming languages, including logic programming, algebraic data types, and others.
3. **"I See What You Mean" (2015)** https://www.youtube.com/watch?v=R2Aa4PivG0g This talk is about the connection between logic programming and Structured Query Language, but continues the discussion by adding the notion of time.
4. **Barliman: trying the halting problem backwards, blindfolded (2016)** https://www.youtube.com/watch?v=er_lLvkklsk&feature=youtu.be&t=95 This talk is about using miniKanren to build an IDE that takes test-driven development one step further.
5. **Boundaries (2012)** https://www.destroyallsoftware.com/talks/boundaries This video is a little old, but the ideas are still relevant even if the technology isn't. This talk is the most "industry focused" out of the list of talks.
6. **Propositions as Types (2015)** https://www.youtube.com/watch?v=IOiZatlZtGU How the simply typed lambda calculus corresponds to logic.
7. **A Little Taste of Dependent Types (2018)** https://www.youtube.com/watch?v=VxINoKFm-S4

Here's our recommendations on how to choose the talks:

- If you enjoyed the week 9 materials on miniKanren, watch talk #4.
- If you want to see a broad view of programming languages, watch talk #1 or #2
- If you want to see a different view of logic programming, watch talk #3
- If you want to understand what type theory is and why there is so much research in this area, watch talk #6 or #7
- If you want a talk that is more about software engineering, watch talk #5.