Elements of Programming Languages
Coursework 1
Version 1.10 (last updated October 21, 2015)
Due: October 23, 2015, 4pm

Overview

In this assignment, you will implement a simple interpreter and typechecker for a language called Giraffe with integers, strings, booleans, pairs, let-binding, and functions. In addition, Giraffe includes some higher-level constructs (as discussed in class) such as `let fun`, `let rec`, and `let pair`. You will implement capture-avoiding substitution and desugaring for these constructs. Finally, you will implement some simple Giraffe programs.

The syntax of Giraffe is as follows:

$\mathit{Expr} \ni e$	$::=$	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \mathtt{*} e_2$	Numbers
	$\mid$	$b \in \mathbb{B} \mid e_1 == e_2 \mid \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$	Booleans
	$\mid$	$s \in \mathit{String} \mid \mathtt{length}(s) \mid \mathtt{index}(e_1, e_2) \mid \mathtt{concat}(e_1, e_2)$	Strings
	$\mid$	$x \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	Variables and let-binding
	$\mid$	$(e_1, e_2) \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e$	Pairs
	$\mid$	$e_1\ e_2 \mid \backslash x{:}\tau.\ e \mid \mathtt{rec}\ f(x : \tau) : \tau'.\ e$	Functions
	$\mid$	$\mathtt{let}\ (x, y) = e_1\ \mathtt{in}\ e_2$	Syntactic sugar
	$\mid$	$\mathtt{let}\ \mathtt{fun}\ f(x{:}\tau) = e_1\ \mathtt{in}\ e_2 \mid \mathtt{let}\ \mathtt{rec}\ f(x{:}\tau){:}\tau' = e_1\ \mathtt{in}\ e_2$	
$\mathit{Type} \ni \tau$	$::=$	$\mathtt{int} \mid \mathtt{bool} \mid \mathtt{str} \mid \tau_1 \mathtt{*} \tau_2 \mid \tau_1 \mathtt{->} \tau_2$	

The syntax of Giraffe is intentionally chosen to be close to concrete syntax, so there are slight differences from the languages presented in the lectures. Parentheses and conventional precedence conventions are used, and multiplication, lambda-abstraction, pair types, and function types are represented slightly differently:

Abstract syntax	Concrete syntax
$e_1 \times e_2$	$e_1 \mathtt{*} e_2$
$\lambda x{:}\tau.\ e$	$\backslash x{:}\tau.\ e$
$\tau_1 \times \tau_2$	$\tau_1 \mathtt{*} \tau_2$
$\tau_1 \to \tau_2$	$\tau_1 \mathtt{->} \tau_2$

Getting started with Giraffe

We provide a Scala file `CW1.scala` that defines the abstract syntax of Giraffe and provides a simple parser and REPL for Giraffe that you may use to write tests or examples.

You can start the interactive interpreter as follows:

```
$ scala CW1.scala
Welcome to Giraffe!
Enter expressions to evaluate, :load <filename.gir> to load a file, or
:quit to quit.
Giraffe> 1 + 1
AST:  Plus(Num(1),Num(1))

Type Checking...Done!
Type of Expression: IntTy
```

```
Result: NumV(2)
Giraffe>
```

Initially the interpreter only supports a few of the above constructs, such as numbers and addition.

The `CW1.scala` file provides functions for parsing Giraffe code to abstract syntax trees, which you can also use from within the Scala read-eval-print loop after loading `CW1.scala`.

- `CW1.parser.parseStr: String => Expr` which takes a string with some code and attempts to parse it.
- `CW1.parser.parse: String => Expr` which takes a path to a file and reads the file contents and attempts to parse them.

In addition, the following functions can be used to typecheck or evaluate a closed expression:

- `CW1.Main.evaluate: Expr => Value`, which calls the evaluation function you are to implement in part 2 with an empty initial environment
- `CW1.Main.typecheck: Expr => Type`, which calls the typechecking function you are to implement in part 3 with an empty initial context

We include four example programs: a pair swapping function, factorial, exponentiation, and a function to test whether two strings have the same last character. These are provided both as files and embedded in `CW1.scala`.

The interpreter can be run from the command line as follows:

```
$ scala CW1.scala example1.gir # runs the interpreter on a file
```

Finally, we provide a JAR file that contains a sample solution called `CW1Solution.jar`. You can run this as follows:

```
$ scala CW1Solution.jar              # starts the interactive interpreter
$ scala CW1Solution.jar example1.gir # runs the interpreter on a file
```

(You are welcome to try to decompile this code if you think that will be easier than solving the exercises directly.)

Objectives

The rest of this handout defines exercises for you to complete, building on the partial implementation in `CW1.scala`. You may add your own function definitions or other code, but please use the existing definitions/types for the functions we ask you to write in the exercises, to simplify automated testing we may do. Also, please do not change code in the `CW1.CWParser` and `CW1.Main` submodules.

Your solutions may make use of Scala library operations, such as the list and list map operations that have been covered in the lab. However, your solution should not make use of any features of Scala that have not been covered so far, particularly side-effects (with the exception of the `Gensym` object provided as part of `CW1.scala` for your use in exercise 1).

This assignment relies on material covered up to Lecture 6 (October 13). The four sections of this assignment are independent and can be attempted in any order. Partial credit is given for progress on each part, so we suggest implementing (and testing) the more straightforward cases of each part first before attempting the remaining cases.

This assignment is graded on a scale of 20 points, and amounts to 10% of your final grade for this course. You may not work in groups on this assignment and must document any meaningful discussions you have about this assignment (or external sources you consult) in the process of constructing your solutions.

Submission instructions You should submit a single file, called `CW1.scala`, with missing code filled in as specified in the exercises in the rest of this handout. To submit, use the following DICE command:

```
$ submit epl 1 CW1.scala
```

The submission deadline is 4pm on Friday, October 23.

Additional notes

- There are several lines that look like this in the provided `CW1.scala` code:

```
    case _ => sys.error("subst:␣todo")
```

  These catch any cases you don't handle and raise an error. You should remove these lines in your solution (if you leave them in and the appear before your code, then your code won't get run since the wildcard _ matches any pattern.)

- The additional file `SubstTest.scala` includes several tests of substitution in combination with evaluation. The tests can be run as follows:

```
scala> :load CW1.scala
...
scala> :load SubstTest.scala
... should show several variables being bound to "true"
    if test passes, "false" otherwise...
```

- If you load `CW1.scala` and want to test your solution, you should be aware that the definition of `subst` and other things in `CW1.scala` is in an **`object`**, which means that to access it you'd need to write `CW1.subst` and so on. `SubstTest.scala` includes a line `import CW1._` which makes all of the components of `CW1` available for use without the `CW1.` prefix. You can also type this import declaration into the scala REPL directly.

- Substitution does not have to handle the syntactic sugar forms — if so, however, you should make sure that substitution is only called on desugared expressions (e.g. in the `desugar` function).

- `CW1Solution.subst` implements substitution for let rec, let pair and let fun so you can compare its behavior to yours.

- A function `SubstTest.aequiv` that tests alpha-equivalence of two Giraffe ASTs is provided in `SubstTest.scala`. Many of the tests now just use that instead of `eval` and you can use it to write your own substitution tests that don't depend on `eval`.

- In defining substitution, it is always safe to rename a bound variable to a completely fresh name. Therefore, you may find it easier to simply rename all bound names as you encounter them, rather than trying to avoid unnecessary renaming (especially in cases that bind multiple names). The correctness of `subst` is all that matters, so do not worry about minimizing the amount of renaming.

- For the string primitive operations, you are free to use Scala's built in string library operations. (They are exactly the standard Java ones, since Scala reuses Java's String library class.)

- For part 4, you should construct expressions in Giraffe as specified in this handout without adding any features. If Giraffe seems too limited to solve these problems, please have a look at the provided examples for ideas. Also note that you may assume that the input values are sensible (this is now clarified in the exercise statements.)

1 Syntactic transformation

In this section you will implement some techniques for transforming the abstract syntax of a program.

1.1 Capture-avoiding substitution

In this part, you are to implement a Scala function `subst: (Expr,Expr,Variable) => Expr` so that `subst(e,e',x)` returns $e[e'/x]$, that is, the result of capture-avoiding substitution of e' for x in e.

To deal with variable renaming, you will need to generate fresh names when crossing a binder, in order to avoid *variable capture*. For example, if we are substituting x for y in $\texttt{let}\ x = 1\ \texttt{in}\ x + y$ then we need to rename the bound name x in the `let`-expression to avoid getting the wrong result $\texttt{let}\ x = 1\ \texttt{in}\ x + x$.

To be safe, we suggest renaming all bound names to fresh ones before recursively processing subexpressions in the scope of the bound names. We provide an object `Gensym` object for this purpose; this object encapsulates a counter that is used to generate unique ids. Your solution may use `Gensym` to generate fresh names and must not use assignment or mutable variables in any other way.

Exercise 1. *Finish the definition of capture-avoiding substitution by filling in the remaining cases for `subst`. This should handle all abstract syntax cases and should rename bound variables to avoid capture. You may use the function `Gensym.gensym` to create a new variable name that (you may assume) is not already in use elsewhere in the expression.*

[3 marks]

1.2 Desugaring

Consider the following desugaring rules, where the left-hand side describes a Giraffe expression form that can be defined in terms of other Giraffe constructs, shown on the right-hand side:

$$\begin{aligned}
\texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 &\quad\longrightarrow\quad \texttt{let}\ p = e_1\ \texttt{in}\ e_2[\texttt{fst}\ p/x, \texttt{snd}\ p/y] \\
\texttt{let fun}\ f(x{:}\tau) = e_1\ \texttt{in}\ e_2 &\quad\longrightarrow\quad \texttt{let}\ f = \backslash x{:}\tau.\ e_1\ \texttt{in}\ e_2 \\
\texttt{let rec}\ f(x{:}\tau){:}\tau' = e_1\ \texttt{in}\ e_2 &\quad\longrightarrow\quad \texttt{let}\ f = \texttt{rec}\ f(x{:}\tau){:}\tau'.\ e_1\ \texttt{in}\ e_2
\end{aligned}$$

In the first rule, the variable p should be a fresh variable not already in use in the expression. Again, you may use `Gensym.gensym` to generate a fresh variable name.

The function `desugar: Expr => Expr` in `CW1.scala` is intended to traverse an expression and replace all occurrences of the above defined forms with their definitions. Some easy cases are already written for you.

Exercise 2. *Complete the definition of `desugar`. Your implementation should replace all of the defined forms (let–pair, let–fun and let–rec) above in one pass over the expression.*

[3 marks]

2 Interpretation

2.1 Primitive operations

Giraffe includes several primitive data types and operations on them. It is convenient to define these operations on the `Value` type that represents the results of evaluation. In `CW1.scala` you will find a definition of two examples, `add` and `subtract`, which implement integer addition on the `Value` type (as shown in the lectures). Several additional primitive operations are needed:

- `multiply` takes two integer values and multiplies them
- `eq` compares two values of the same base type (`int`, `str`, `bool`)
- `length` returns the integer value of a string's length (which we write as $|s|$ in mathematical notation, e.g. $|\texttt{abc}| = 3$)
- `index` given string value s and integer value i, returns the one-character string at position i of s. Positions start at zero. We write this in mathematical notation as follows: $s[i]$. For example, `("abc")[0] = "a"` and `("abc")[1] = "b"`. The behavior on out-of-range indices is undefined (that is, you may return a dummy value or raise an error using `sys.error` in this case).
- `concat` concatenates two string values. We write this mathematically as $s_1 \cdot s_2$. For example, `("abc")` $\cdot$ `("def") = "abcdef"`.

The behavior of these operations is unspecified if applied to values of unexpected types. In these cases you may use Scala's `sys.error()` function to signal an error: for example, adding an integer to a string, or comparing a string and a boolean for equality. Alternatively, you may choose to return some dummy value in these cases, or leave them unhandled in pattern matching (which will also result in a Scala run-time error).

We have covered operations like multiplication and equality testing (for integer and boolean values) in lectures, and you may reuse or adapt this code. We have not covered string equality, or the other string operations, in lectures, but it should be straightforward to implement these, following the same pattern.

Exercise 3. *Implement the remaining primitive operations for multiplication, equality testing, string length, string indexing and string concatenation over `Value`s.*

[2 marks]

$\boxed{\sigma, e \Downarrow v}$

$$\frac{n \in \mathbb{N}}{\sigma, n \Downarrow n} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 - e_2 \Downarrow v_1 -_{\mathbb{N}} v_2} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, e_1 * e_2 \Downarrow v_1 \times_{\mathbb{N}} v_2}$$

$$\frac{b \in \mathbb{B}}{\sigma, b \Downarrow b} \qquad \frac{\sigma, e_1 \Downarrow v \quad \sigma, e_2 \Downarrow v}{\sigma, e_1 == e_2 \Downarrow \texttt{true}} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2 \quad v_1 \neq v_2}{\sigma, e_1 == e_2 \Downarrow \texttt{false}}$$

$$\frac{\sigma, e \Downarrow \texttt{true} \quad \sigma, e_1 \Downarrow v}{\sigma, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v} \qquad \frac{\sigma, e \Downarrow \texttt{false} \quad \sigma, e_2 \Downarrow v}{\sigma, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow v}$$

$$\frac{s \in \mathit{String}}{\sigma, s \Downarrow s} \qquad \frac{\sigma, e \Downarrow s}{\sigma, \texttt{length}(e) \Downarrow |s|} \qquad \frac{\sigma, e_1 \Downarrow s \quad \sigma, e_2 \Downarrow n}{\sigma, \texttt{index}(e_1, e_2) \Downarrow s[n]} \qquad \frac{\sigma, e_1 \Downarrow s_1 \quad \sigma, e_2 \Downarrow s_2}{\sigma, \texttt{concat}(e_1, e_2) \Downarrow s_1 \cdot s_2}$$

$$\frac{}{\sigma, x \Downarrow \sigma(x)} \qquad \frac{\sigma, e_1 \Downarrow v_1 \quad \sigma[x := v_1], e_2 \Downarrow v_2}{\sigma, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2}$$

$$\frac{\sigma, e_1 \Downarrow v_1 \quad \sigma, e_2 \Downarrow v_2}{\sigma, (e_1, e_2) \Downarrow (v_1, v_2)} \qquad \frac{\sigma, e \Downarrow (v_1, v_2)}{\sigma, \texttt{fst}\ e \Downarrow v_1} \qquad \frac{\sigma, e \Downarrow (v_1, v_2)}{\sigma, \texttt{snd}\ e \Downarrow v_2}$$

$$\frac{}{\sigma, \backslash x{:}\tau.\ e \Downarrow \langle \sigma, \backslash x.\ e \rangle} \qquad \frac{}{\sigma, \texttt{rec}\ f(x{:}\tau){:}\tau'.\ e \Downarrow \langle \sigma, \texttt{rec}\ f(x).\ e \rangle} \qquad \frac{\sigma, e_1 \Downarrow \langle \sigma_0, \backslash x.\ e \rangle \quad \sigma, e_2 \Downarrow v_2 \quad \sigma_0[x := v_2], e \Downarrow v}{\sigma, e_1\ e_2 \Downarrow v}$$

$$\frac{\sigma, e_1 \Downarrow \langle \sigma_0, \texttt{rec}\ f(x).\ e \rangle \quad \sigma, e_2 \Downarrow v_2 \quad \sigma_0[f := \langle \sigma_0, \texttt{rec}\ f(x).\ e \rangle, x := v_2], e \Downarrow v}{\sigma, e_1\ e_2 \Downarrow v}$$

Figure 1: Evaluation rules for Giraffe

2.2 An environment-based interpreter

The Scala file contains a skeleton of an interpreter based on the rules in Figure 1. Unlike the interpreters covered in class, this evaluator uses an explicit *environment* to record the values of variables. An environment is a mapping from variable names to values. The following grammar rules describe environments and values:

$$\begin{array}{rcl} \mathit{Val} \ni v & ::= & n \mid b \mid s \mid (v_1, v_2) \mid \langle \sigma, \backslash x.\ e \rangle \mid \langle \sigma, \texttt{rec}\ f(x).\ e \rangle \\ \mathit{Env} \ni \sigma & ::= & [x_1 = v_1, \ldots, x_n = v_n] \end{array}$$

We write $\sigma(x)$ for the result of looking up the value of x in σ and $\sigma[x := v]$ for the environment obtained by adding (or replacing) the binding of x to v in σ. In Scala, we can use `ListMap[Variable,Value]` to represent environments.

The rules for environment-based evaluation are defined in Figure 1. Most of the cases of evaluation are similar to those for the evaluator covered in class, except with the addition of the environment parameter σ.

One important difference is that there is now an explicit rule for variable evaluation, which looks up the value of the variable in the environment. Similarly, constructs that bind variables (such as `let`) now need to add the value of the variable to the environment, instead of substituting it into the expression. (You will want to use Scala's built-in `ListMap` lookup and update operations for these cases.)

Another major difference is the way function values are handled: when we evaluate a lambda-abstraction, we need to construct a value that pairs up the lambda-abstraction expression with the environment present at the time the value was created. This structure, written $\langle \sigma, \backslash x.\ e \rangle$, is called a *closure*[1] because it "closes off" the function by providing the values of any free variables. Likewise, for a recursive function $\texttt{rec}\ f(x).\ e$ the corresponding value is a closure $\langle \sigma, \texttt{rec}\ f(x).\ e \rangle$. Importantly, in both cases, when we evaluate the *body* of a called function, we use the environment stored in the closure, not the environment present when the function call is evaluated.

Finally, notice that the rules in Figure 1 do not include rules for the "syntactic sugar" let-binding forms.

[1]The term *closure* is sometimes used as a generic term to refer to first-class functions, but really closures are an implementation technique for first-class functions with static scope. Without closures, an environment-based interpreter would look for the values of local variables in the environment in which the function is called, i.e. we would have dynamic scope instead.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \star e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash b : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{int}, \texttt{bool}, \texttt{str}\}}{\Gamma \vdash e_1 == e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\frac{s \in \mathit{String}}{\Gamma \vdash s : \texttt{str}} \qquad \frac{\Gamma \vdash e : \texttt{str}}{\Gamma \vdash \texttt{length}(e) : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash \texttt{index}(e_1, e_2) : \texttt{str}} \qquad \frac{\Gamma \vdash e_1 : \texttt{str} \quad \Gamma \vdash e_2 : \texttt{str}}{\Gamma \vdash \texttt{concat}(e_1, e_2) : \texttt{str}}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \star \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \star \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \star \tau_2}{\Gamma \vdash \texttt{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \texttt{->} \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash \backslash x{:}\tau.\ e : \tau \texttt{->} \tau'} \qquad \frac{\Gamma, f : \tau \texttt{->} \tau', x{:}\tau \vdash e : \tau'}{\Gamma \vdash \texttt{rec}\ f(x{:}\tau){:}\tau'.\ e : \tau \texttt{->} \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \star \tau_2 \quad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ (x, y) = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e_1 : \tau_2 \quad \Gamma, f{:}\tau_1 \texttt{->} \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let fun}\ f(x{:}\tau_1) = e_1\ \texttt{in}\ e_2 : \tau} \qquad \frac{\Gamma, f{:}\tau_1 \texttt{->} \tau_2, x{:}\tau_1 \vdash e_1 : \tau_2 \quad \Gamma, f{:}\tau_1 \texttt{->} \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let rec}\ f(x{:}\tau_1){:}\tau_2 = e_1\ \texttt{in}\ e_2 : \tau}$$

Figure 2: Typing rules for Giraffe

You do not need to implement evaluation rules for these; instead, they are translated away to equivalent forms by `desugar`.

Exercise 4. *Complete the definition of* `eval: (ListMap[Variable,Value],Expr) => Value`*, following the rules in Figure 1.*

[4 marks]

---

3 Typechecking

Figure 2 summarizes the typechecking rules covered in lectures, plus some rules for string constructs. In Scala, we can use `ListMap[Variable,Type]` to represent type environments Γ. These rules can be read as an algorithm for computing a type for an expression e, given a typing context Γ, or determining that the expression e does not typecheck. Specifically, it is the case that given an expression e and environment Γ, there is at most one type τ satisfying $\Gamma \vdash e : \tau$.

`CW1.scala` contains a preliminary definition of `tyOf` for Giraffe with some cases filled in already. Given a type context `ctx` and an expression `e`, a call to `tyOf(ctx,e)` should either terminate and return the type of `e`, or raise an error. (You can use `sys.error` to flag such errors.) The provided code shows how to do this for a few simple cases.

Notice that Figure 2 does include typechecking rules for the syntactic sugar let-binding forms. It is often helpful to typecheck programs prior to desugaring, so that the error messages will relate to the original source program. Therefore, you should implement these typechecking cases.

Exercise 5. *Complete the definition of the typechecker* `tyOf: (ListMap[Variable,Type],Expr) => Type`*, following the rules in Figure 2.*

[4 marks]

---

4 Some simple programs

In this section you will implement some simple programs in Giraffe. You can use the provided sample solution `CW1Solution.jar` to test these programs, and once you have completed other parts of this assignment you may also use them to test your implementation of Giraffe. You may use the parser provided in `CW1.scala` to construct the required programs (how to do this is illustrated using other examples in `CW1.scala`), or you can just write the abstract syntax directly in Scala. If these programs don't run or typecheck correctly in your interpreter, you may also want to write your own, smaller tests to isolate the problem.

Both exercises in this section ask for expressions of function type (unlike the examples we provided, which show functions defined and then applied to representative arguments). You should construct these expressions in Giraffe as specified in this handout without adding any features. If Giraffe seems too limited to solve these problems, please have a look at the provided examples for ideas.

4.1 Fibonacci numbers

Recall that the Fibonacci sequence is defined as follows:

$$\begin{array}{rcl} x_1 & = & 1 \\ x_2 & = & 1 \\ x_3 & = & x_1 + x_2 \\ & \vdots & \\ x_{n+2} & = & x_n + x_{n+1} \end{array}$$

Exercise 6. *Define a Scala expression* `fib: Expr` *whose* Giraffe *type is* `int -> int` *that computes the n-th Fibonacci number. This implementation does not have to be efficient (i.e. a naive, exponential implementation is fine). You may assume that the input n is a positive integer.*

[2 marks]

4.2 Substrings

We say that string s is a *substring* of another string t if t is of the form $t_1 \cdot s \cdot t_2$, i.e. if s appears inside t. For example, `"ab"` and `"bc"` are substrings of `"abcd"`, but `"ad"` is not.

Exercise 7. *Define a Scala expression* `substring: Expr` *whose* Giraffe *type is* `str * str -> bool` *that determines whether its first argument is a substring of the second. You may assume that the arguments to* `substring` *are sensible, that is, that the length of the first argument is less than that of the second.*

[2 marks]

A Change log

- V1.1 (October 5)
  - Fixed some mistakes in Figure 1, and clarified notation for σ
  - Added some clarification on how to read the typing rules as a function.
  - Added some clarification on how this assignment relates to the lecture material.
- V1.2 (October 7)
  - Fixed a typo in the evaluation rule for let-binding.
- V1.3 (October 8)
  - Added information about tests for substitution correctness
- V1.4 (October 11)

  - Added information about wildcard cases
  - Added information about imports.

- V1.5 (October 13)
  - Clarified that substitution should either handle all cases, or only be called on desugared expressions.

- V1.6 (October 15)
  - `CW1Solution` now implements substitution for syntactic sugar forms.
  - `SubstTest` now includes an alpha-equivalence checker.

- V1.7 (October 15)
  - Fixed the inevitable bugs in substitution for let rec and let fun.
  - Added a note regarding fresh renaming and substitution

- V1.8 (October 15)
  - Interpreter (and sample solution) now desugar after typechecking and print both raw and desugared syntax
  - Added a note regarding using String library functions

- V1.9 (October 20)
  - Added a hint about using let pair for multi-argument functions
  - Fixed SubstTest tests to deal with the fact that parsing no longer desugars
  - Added some more complex examples, illustrating multiple arguments
  - REPL now handles exceptions during loading (e.g. parse errors)
  - Clarified that the expressions in exercises 6 and 7 should be the functions, not examples applied to arguments.

- v1.10 (October 21)
  - Fixed problem with `SubstTest.aequiv`
  - Clarified that Giraffe REPL can't handle multi-line programs