CSCI-3300 (Fall 2016) Final Project
Total Points: 400

This course requires a final project instead of a final exam. This will allows the student to focus on the final project before they are required to study for their other final exams; which will reduce the stress during finals week. In addition, I hope the student finds doing a project more fun and rewarding than a standard final exam.

Everything related to the project must be added to the students git repository under the project directory. There will be several milestones associated with the final project. Thus, each deliverable must be checked into the repository by the deadline associated with the milestone.

1 (60 pt) Project Approval (Due on 10/21)

The first milestone is the approval of a project. The students have a choice in the project they will do, but that project must involve the topics of this course. For example, the project can be done using the Haskell programming language, or the project could implement one of the programming languages we discuss during the semester in Java, or in any other programming language. or the student could use any other functional programming language for their project. However, the project must be approved before it is accepted. **There are several example projects that can be chosen in the appendix of this document.**

The approval is to make sure that the project is complex enough to be worth 400 points or 41% of the students grade. The project approval takes place during a scheduled meeting with the students and I. This meeting must take place before the approval meeting deadline (10/21). Before the approval meeting takes place the student is required to complete and turn in (by checking it into the repository) a summary detailing the project they are proposing. This report cannot be longer than three pages, and must be in PDF format. Check in the report to the directory `projects/approval` in your repository and name the file `project-proposal.pdf`. During the approval meeting the students are required to give an oral summary of the project, and ask any questions they may have.

This project may be done in groups of **at most two** students. If two students wish to work together on this project, then this must be made clear in the project proposal, and both students must be present during the project approval meeting.

Students working in groups of two will work on their project inside a group repository created by at the time of the approval meeting. Then each student must commit all of their own work under

their own login. Points will be deducted otherwise.

2 (60 pt) Midpoint Deliverable (Due by on 11/11)

Everyone will be given approximately eight weeks to complete the project. Halfway through this period there is a midpoint deliverable. This milestone is to insure that everyone is making progress on the project. The student is required to write a midpoint report, with a maximum of five pages in PDF format, detailing the progress they have made on their project. This summary should outline what is currently implemented, and should reflect on the challenges the students have experienced. Furthermore, the summary should mention any problems they have ran into. Lastly, all code that has been currently written should be turned in with the report. The code that is turned in must be working. This does not mean that the project must be done, but should at least compile. **At the midpoint deliverable all projects should be half way finished.**

Note that this also must be turned in via the git repository. Add the report to the directory `projects/midpoint-deliverable` and name the file `midpoint-report.pdf`. Finally, when you want to check in your midpoint code, which is required, first make all desired changes, and then create a new branch called `midpoint-release`, and then push that branch to your Bitbucket account. I will then be able to checkout this branch and see your midpoint progress. Adding this to a branch means that after submitting your midpoint code, you can keep making changes and even pushing them to your Bitbucket account without changing your midpoint submission.

3 (220 pt) Final Project Sources (Due by 11:59pm 12/8)

Completed projects are due by the end of the day on 12/8. This includes all working code, and instructions for compiling and running the project. Instructions should include all the necessary software and libraries needed to run the project. Again, the instructions should be in PDF format. **If a project does not compile and run, then it cannot be graded.**

The final project code must be kept in the directory `project/sources`.

4 (60 pt) Final Presentation (Due on 12/15)

Every group is required to give a ten minute final presentation on the day of the final exam. The presentation must include the following:

- A summary of the project,
- Lessons learned:
  - Unexpected results,
  - Problems experienced during development, etc.
- A demo of the project.

The presentation must be at most ten minutes long. An ideal presentation would have content that lasts for about six to eight minutes long, and leaves three to four minutes for questions. All presentations must be given using slides via PowerPoint or some other similar application.

Pre-approved Projects

A The λ-Tutor

One claim I have, or will, made throughout this semester is that some programming languages should not be programmed in directly, but instead, compiled to. The C programming language is largely only compiled to this day in age, but there are other languages that should no longer be programmed in. Javascript is an example of one of these programming languages. There is dialect of Haskell called the Elm programming language which compiles to Javascript. **The goal of this project is to implement a web application using the Elm programming language implementing an interactive tool for learning the λ-calculus.**

A.1 Getting to know Elm

The first step in this project is for the student to introduce themselves to the Elm programming language, and get it installed on their computer. The following resources should be consulted:

- The Elm homepage:
  `http://elm-lang.org/`
- (Example) TODO List:
  `https://github.com/evancz/elm-todomvc`

It would then help to first try some basic examples, and then work the final web application.

A.2 The web app

The web application must run in the browser, and be supported by either Google Chrome or Apple Safari. The application must have the following features:

- An intuitive and elegant design.
- Be able to prompt the user for a λ-calculus expression. The syntax should be derived from the following grammar:

```
t ::= x | \x.t | t1 t2 | ( t )
```

- Given a λ-calculus expression generate the parse tree of the expression.
- Given a λ-calculus expression, generate the parse tree, and then allow the user to choose redexes in the parse tree to contract producing a new parse tree.

- Given a λ-calculus expression, generate the parse tree, highlight every free variable one color, and highlight every bound variable another. Allow for the user to compute either just the free variables, just the bound variables, or both.

The design choices are up to the discretion of the student.

B A Linear SAT Solver

In the most general sense automated theorem proving consists of an algorithm that takes as in input an arbitrary mathematical statement and outputs whether or not the input statement is true for false. This general task is at the present time an open problem, but there are some classes of mathematical statements where this can be done. **The goal of this project is to implement an automated theorem prover called a SAT solver.**

A SAT solver, for satisfiability solver, is a program that takes as input a propositional formula and outputs whether or not the input formula is satisfiable or not. If it is, then it should output, what is called, a satisfiable assignment.

Propositional Formulas. The following grammar defines the syntax of propositional formulas:

$$A, B, C ::= p \mid A \wedge B \mid A \vee B \mid A \Rightarrow B \mid \neg A$$

The variable p is called a propositional variable, that is a variable that can either be true or false, and we will also denote these by q, r, and s. An example formula is $\neg(p \Rightarrow q) \Rightarrow (p \wedge \neg q)$. A satisfiable assignment of a propositional formula, P, is an assignment of true and false to each propositional variable of P that result in P being true. For example, setting $p = T$, and $q = F$ is a satisfiable assignment to the example formula given above. However, setting $p = F$ and $q = T$ is not.

Now we can translate propositional formulas of the previous grammar into a fragment called the conjunction-negation normal form (CNNF) which are formulas defined by the following grammar:

$$X, Y, Z ::= p \mid X \wedge Y \mid \neg A$$

The following recursive algorithm does the conversion:

$$\begin{aligned}
\mathsf{prop2CNNF}(p) &= p \\
\mathsf{prop2CNNF}(A \wedge B) &= \mathsf{prop2CNNF}(A) \wedge \mathsf{prop2CNNF}(B) \\
\mathsf{prop2CNNF}(A \vee B) &= \neg((\neg\mathsf{prop2CNNF}(A)) \wedge (\neg\mathsf{prop2CNNF}(B))) \\
\mathsf{prop2CNNF}(A \Rightarrow B) &= \neg(\mathsf{prop2CNNF}(A) \wedge (\neg\mathsf{prop2CNNF}(B))) \\
\mathsf{prop2CNNF}(\neg A) &= \neg\mathsf{prop2CNNF}(A)
\end{aligned}$$

Implement in Haskell this conversion, and then use it to implement the linear SAT algorithm described by the attached section called "Simple, Incomplete SAT solvers."[1] Finally, come up with twenty test cases showing the correctness of your implementation.

[1] These notes are by James Hook and Tim Sheard of Portland State.

C RecCalc: A Simply Typed Functional Programming Language

To fully understand the theory of programming languages (PLs) it is important to understand the design aspect of PLs, but it is equally important to understand how to implement them. **The goal of this project is to implement – in Haskell – a simple but powerful programming language called RecCalc.** The RecCalc PL is based off of a simply typed λ-calculus called Gödel's system T, which will be introduced at some point during the semester. It is very powerful, in fact, any amount of arithmetic can be carried out with in it.

C.1 RecCalc's Specification

The RecCalc syntax is defined as follows:

$$\begin{aligned} \text{(Types)} \quad & T, A, B, C ::= \mathsf{Nat} \mid A \to B \\ \text{(Terms)} \quad & t ::= x \mid 0 \mid \mathsf{suc}\ t \mid \mathsf{fun}\ x : T \Rightarrow t \mid \mathsf{app}\ t_1\ \mathsf{to}\ t_2 \mid \mathsf{rec}\ t\ \mathsf{with}\ t_1 \parallel t_2 \\ \text{(Term Contexts)} \quad & \Gamma ::= \cdot \mid x : T \mid \Gamma_1, \Gamma_2 \end{aligned}$$

Substitution is denoted:

$$\mathsf{replace}\ x\ \mathsf{with}\ t\ \mathsf{in}\ t'$$

This syntax has been implemented in Syntax.hs. Compare the definition of the concrete syntax above with the abstract syntax defined in that file.

Now we define the type checking algorithm:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \mathsf{fun}\ x : T_1 \Rightarrow t : T_1 \to T_2}\ \text{T\_Lam}$$

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash \mathsf{app}\ t_1\ \mathsf{to}\ t_2 : T_2}\ \text{T\_App} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \text{T\_Zero} \qquad \frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\ t : \mathsf{Nat}}\ \text{T\_Suc}$$

$$\frac{\begin{array}{l}\Gamma \vdash t : \mathsf{Nat} \\ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \to \mathsf{Nat} \to T\end{array}}{\Gamma \vdash \mathsf{rec}\ t\ \mathsf{with}\ t_1 \parallel t_2 : T}\ \text{T\_Rec}$$

At this point we have seen the syntax and how terms are typed using the type checking rules next we define the evaluation rules:

$$\frac{}{\mathsf{app}\,(\mathsf{fun}\,x : T \Rightarrow t)\,\mathsf{to}\,t' \rightsquigarrow \mathsf{replace}\,x\,\mathsf{with}\,t'\,\mathsf{in}\,t}\ \text{E\_Beta} \qquad \frac{}{\mathsf{rec}\,0\,\mathsf{with}\,t_1 \parallel t_2 \rightsquigarrow t_1}\ \text{E\_RecBase}$$

$$\frac{}{\mathsf{rec}\,(\mathsf{suc}\,t)\,\mathsf{with}\,t_1 \parallel t_2 \rightsquigarrow \mathsf{app}\,(\mathsf{app}\,t_2\,\mathsf{to}\,(\mathsf{rec}\,t\,\mathsf{with}\,t_1 \parallel t_2))\,\mathsf{to}\,t}\ \text{E\_RecStep}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{fun}\,x : T \Rightarrow t \rightsquigarrow \mathsf{fun}\,x : T \Rightarrow t'}\ \text{E\_Lam} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{app}\,t_1\,\mathsf{to}\,t_2 \rightsquigarrow \mathsf{app}\,t_1'\,\mathsf{to}\,t_2}\ \text{E\_App1}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{app}\,t_1\,\mathsf{to}\,t_2 \rightsquigarrow \mathsf{app}\,t_1\,\mathsf{to}\,t_2'}\ \text{E\_App2} \qquad \frac{t \rightsquigarrow t'}{\mathsf{suc}\,t \rightsquigarrow \mathsf{suc}\,t'}\ \text{E\_Suc}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{rec}\,t\,\mathsf{with}\,t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\,t'\,\mathsf{with}\,t_1 \parallel t_2}\ \text{E\_Rec1} \qquad \frac{t_1 \rightsquigarrow t_1'}{\mathsf{rec}\,t\,\mathsf{with}\,t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\,t\,\mathsf{with}\,t_1' \parallel t_2}\ \text{E\_Rec2}$$

$$\frac{t_2 \rightsquigarrow t_2'}{\mathsf{rec}\,t\,\mathsf{with}\,t_1 \parallel t_2 \rightsquigarrow \mathsf{rec}\,t\,\mathsf{with}\,t_1 \parallel t_2'}\ \text{E\_Rec3}$$

Again, the goal is to implement this language which amounts to taking the above algorithm descriptions and translating them in Haskell. The syntax and parser are provided. The next section gives an overview of what is given to you as an initial code base. A few more functions that are provided to you are given in the description of each project task given below.

C.2 Overview of the Initial Code Base

Located in the course git repository is a directory called Project, and within it are several Haskell files:

Syntax.hs	The implementation of RecCalc's syntax.
Parser.hs	The RecCalc parser.
Pretty.hs	The file that will contain the pretty printer.
TypeCheck.hs	The file that will contain the type checker.
Eval.hs	The file that will contain the evaluator.
Main.hs	The file that will contain the main functions.

The syntax and parser for RecCalc have already been implemented and tested, these implementations are provided. Everyone should first read through the Syntax.hs file before attempting any of the project tasks. It is important to understand the definition of terms and types.

C.2.1 Syntax.hs

The syntax is implemented using a Haskell library called Unbound. It is not important for us to fully understand this library, but only understand what is necessary for this project. The unbound library provides Haskell with support for handling binding when implementing programming languages. There are three functions it provides that we will need to use:

`bind`	Takes a term and a name, and then returns a term with the name bound in it.
`unbind`	Takes a term with a name bound in it, and returns a pair of the name and term.
`aeq`	Tests two terms for α-equivalence.

This file also has several functions that will be needed to handle replacing variables in terms (substitution function). These are as follows:

`replace`	Takes a name and two terms, and replaces the name in the second term with the first.

More details of each of the above functions will be given in class.

C.2.2 Parser.hs

The goal of a parser is to take a string representation of a program and turn it into a program in the abstract syntax. Our abstract syntax is the language defined in Syntax.hs. This file provides four parsers:

`parseTerm`	Parses a string into a term.
`parseType`	Parses a string into a type.
`parseCtx`	Parses a string into a list of pairs of variable names and their corresponding types.

The initial code base complies right away. Each project task (given in the next section) are already present in their respective files. They have simply been left undefined. For example, in the file TypeCheck.hs there are the following lines:

```
typeCheck :: Fresh m => Ctx -> Term -> ErrorT String m Type
typeCheck ctx t = undefined
```

It will then be your job to fill in these functions with their respective definitions using the definitions given in Section C.1. Thus, you will not be responsible for filling in the types of the functions, but only their definitions. In addition, every external library module needed for this project has already been imported.

C.3 Project Tasks

This section simply lists the required work.

C.3.1 Pretty Printer

The syntax of terms (or programs) of RecCalc are represented by the datatype called `Term`, and the syntax of types are represented by the datatype called `Type` in the file Syntax.hs. Now this is the internal representation of terms and types – often called the abstract syntax – and not the representation we as users use to program in RecCalc – often called the concreate syntax (see Section C.1).

A pretty printer takes a program or type in the abstract syntax and outputs its equivalent form in the concreate syntax. Lets consider a couple of examples:

Input	$\Rightarrow$	Output
`Fun Nat (bind (s2n "y") (Var (s2n "y")))`	$\Rightarrow$	$\mathsf{fun}\ y : \mathsf{Nat} \Rightarrow y$
`Arr Nat Nat`	$\Rightarrow$	$\mathsf{Nat} \to \mathsf{Nat}$

So your task – if you choose to accept it – is to fill in the following two functions:

```
prettyType :: Fresh m => Type -> m String
prettyType = undefined

prettyTerm :: Fresh m => Term -> m String
prettyTerm = undefined
```

The first takes a `Type` and needs to output its equivalent form in the concrete syntax as a string. Then the second does the same thing, but for terms (programs).

C.3.2 Type Checker

This task simply asks you to implement the type construction algorithm outlined in Section C.1. Thus, you must implement the following function:

```
typeCheck :: Fresh m => Ctx -> Term -> ErrorT String m Type
typeCheck = undefined
```

C.3.3 Evaluator

Similarly, this task asks you to implement the evaluation algorithm given in Section C.1. This requires you to implement the following:

```
eval :: Fresh m => Term -> m Term
eval = undefined
```

C.3.4 Type Preservation

This task requires you to implement the following function:

```
typePres :: Term -> Bool
typePres = undefined
```

The type preservation function should take in a term as input, and then first construct the input terms type using `runTypeChecker` (provided), and then evaluate the input term using `runEval` (provided). Finally, construct the type of the term returned by the evaluator (`runEval`), and then output `True` if it is the same type as the original input, and output `False` otherwise.

C.3.5 Main Functions

There are three main functions you must implement:

```
mainCheck :: IO ()
mainCheck = undefined

mainEval :: IO ()
mainEval = undefined

mainPres :: IO ()
mainPres = undefined
```

The first, `mainCheck`, should do the following:

- Ask the user for a context as a string,
- Ask the user for a term as a string,
- Parse the input using `parseCtx` and `parseTerm`,
- Type check (using `runTypeChecker`) the input term using the input context, and
- Finally, output the resulting type using `runPrettyType` (provided).

Note that typing contexts are to be entered by the user as comma separated lists of variable names, that is, in the form `x : T1, y : T2, ..., z : Ti`, where `x`, `y`, and `z` are term variable names, and `T1` through `Ti` are a bunch of types in the concreate syntax.

The second, `mainEval`, should do the following:

- Ask the user for a term,
- Parse the input using `parseTerm`,
- Evaluate the input term using the evaluator (`runEval`), and
- Output the resulting term from the evaluator using `runPrettyTerm`.

Finally, the third, `mainPres`, should do the following:

- Ask the user for a term,
- Parse the input using `parseTerm`,
- Test the term for type preservation using `typePres`, and
- Output the result of `typePres` using `show`.