Final project ideas

CS3490: Programming Languages

This document provides some suggestions for a final project.

Each project is annotated with a difficulty rating between 1 and 5. If you want to work in a group, the difficulty should be higher the more people you have in the group. For each member of the group, you should add at least one point to the difficulty level. These ratings however are approximate and the difficulty of a project can often be adjusted.

You will need to formulate your own proposal that makes precise what features you intend to implement. The proposal must be approved before you start the final project.

Your final submission will include the complete program code, one or two sample input files or interaction sessions, and a final report. The report should include a basic description of the algorithm(s) you implemented, explanations of the most difficult parts of the code, and a discussion of the things you had to learn, trade-offs you had to make, and any other points of interest.

Your project should be written in Haskell. Exceptions to this rule may be granted if the project idea requires the use of a specific language or functionality unavailable in Haskell. In any case, your program should use functional programming concepts and techniques you have learned in this course. Extra points will be awarded to code that is clean, well-organized, modular, robust, and well-documented.

The principal requirement is that your project should demonstrate mastery of language processing tasks, such as parsing, lexical analysis, representation of structured data in abstract datatypes, etc.

Some project ideas are more research-oriented, which means that a larger part of the project would be dedicated to learning a new topic in depth. In such a project, you are expected to produce a longer report, that summarizes your research findings. For example, in the case of implementing an advanced algorithm or data structure, your report should actually explain how it works. Note that your program should still demonstrate mastery of the skills learned in class, for example, as part of a UI frontend.

Finally, topics marked with a star ($\star$) represent Dr. Polonsky's recommendations, involving particularly interesting interplay of concepts and programming techniques.

Contents

1. Extending a language implemented in class

This section contains the most straightforward projects — if you are not sure what you want to do, I recommend you to pick one of these.

1.1. IMP with procedures (1-2)

The language IMP implemented in Homework 9 can be extended with *procedures*, which are *named blocks of code*. The input file should allow a syntax for introducing such blocks. The set of instructions should be extended to allow to `call` a block by giving its name.

For an extra point of difficulty, you can implement a mechanism for passing variables to the named blocks, so that they can be used as functions. Whether you use a call stack or some other approach, you should make sure the variables do not clash when the function is called recursively.

1.2. FUN with functions (1)

This project should provide a front-end to the call-by-value evaluator of arithmetic and boolean expressions implemented in Lecture 10.31. It should provide for the possibility of defining *functions* directly in the input file, in the format

```
f x1 ... xk = e
```

where `f` is a function name (which can be just a string), `x1` through `xk` are variables, and `e` is an expression that can make use of those variables.

Your interpreter should read in the input file, parse the list of function definition, then allow the user to interactively evaluate arbitrary expressions, which can call functions in the list of definitions.

1.3. CL with definitions (1)

This is similar to the previous project, but you should begin with the implementation of combinatory logic in Lecture 10.14.

You are to provide a front-end that will allow the user to input a file where new combinators can be defined, for example

```
B x y z = x (y z)
C x y z = x z y
W x y = x y y
```

Then you should extend the syntax of combinators to allow previously defined combinators to be invoked. The reduction function should similarly be updated to reflect this new addition to the language.

1.4. PCF or λT with new types (1)

You can start with either PCF, implemented in midterm2 take-home, or λT, implemented in class.

You should add new type and term constructors to the language, which should at the very least include Booleans, Pairs, and one of String, Lists of Integers, or Lists of `a`.

1.5. λT with binary numbers (1-2)

One problem with programming λT is that the integers are represented with a *unary* successor.

A thorough solution to the problem of efficiency would thus be to switch to the binary notation. This would involve switching to a datatype encoding binary strings. This requires two unary constructors like the successor, and a single constant to mark the end of string.

The recursor should also be adjusted to fold over the new datatype instead.

For an extra difficulty point, after you implement this language, you should test your interpreter by writing a λT-program for insertion sort, following the strategy in Lecture 11.18. (Some things, for example, pairing, can actually be simpler if you work with binary numerals.) If everything works correctly, you should get a much better performance, and should be able to test it on a decent-sized list.

1.6. Your own variation

If you have a specific idea for extending one of the languages implemented in class, you can suggest it as well. The most important thing is to make sure you formulate the new language construct in such a way that it interacts nicely with existing infrastructure (substitution, evaluation, etc.)

2. Practical Applications

2.1. ⋆ Regular expression search-and-replace (1-2)

Regex search-and-replace is a very useful tool in an experienced programmer's toolkit.

The classic `sed` command-line tool offers this functionality, but is line-based, and thus can only perform one replacement per line.

Modern text editors all support some version of regex search-and-replace, but they each have their own syntax, increasing the effort required to learn to use it properly.

In this project, you have to implement your own regex search-and-replace program, written in Haskell. It should take as input a filename, a matching string and a replacement string. Then it should perform the replacements, and write the output to a new file or to standard out.

See the I/O section of the Haskell tutorial on how to access command-line arguments.

The most important feature to implement, which makes regex search a thousand times more powerful, is *saving matched patterns into a variable*. The saved variable can be

referred to later in the matched string or the replacement string. A project that fully implements this feature has a difficulty rating of 2, so two people can work on it.

2.2. Advanced calculator (1)

The purpose of this project is to create an advanced calculator. You can start with the one we have implemented in class, and extend into with a rich set of features, including

- Upgrade from `Integer` to `Double` datatype
- Interactive UI
- Scientific functions
- Statistical/Financial functions
- Saving expressions into variables to be accessed later
- Saving expressions with a designated variable into a function name that can be used in later expressions.
- Converting between units, angles, bases, currencies, etc.

The datatype should use double-precision floating point as the underlying number type — which is also the evaluator's return type.

The program should include some basic parsing capability, so that the user could input the expression to be evaluated on the command line. There should also be a way to define new functions from existing ones, like in a graphing calculator.

For extra difficulty, you can consider adding operations that work with these functions, like computing their derivative. You can also consider writing a GUI front-end, although that can constitute a topic of its own.

2.3. Markdown to HTML converter (1)

Markdown is a style of writing plain-text files with consistent notation for headings, emphasized words, lists, etc. It is very convenient for writing things like to-do lists, event reminders, shopping lists, structured plain-text emails, etc.

The following example illustrates some features of Markdown:

```
This is a heading.
------------------
Double asterisks indicate words written in **bold**.
Underscores indicate _italic font_.
1. First item
2. Second item
3. Third item.  We have several choices:
   * First bullet
   * Second bullet
```

```
   * etc.
4. Fourth item.
```

Markdown can be converted into HTML in a straightforward way. The above example would then look more like

This is a heading

Double asterisks indicate words written in **bold**.

Underscores indicate *italic font*.

1. First item
2. Second item
3. Third Item. We have several choices:
   - First bullet
   - Second bullet
   - etc.
4. Fourth item.

In this project, you have to write a Haskell program that takes as input a Markdown file and converts it into an HTML file so that it can be displayed in a browser.

The most tricky part of this project is implementing lists, which requires the parser to keep track of content spread over multiple lines. You should think in advance how you would go about solving this problem.

Read more: `https://en.wikipedia.org/wiki/Markdown`.

Warning! Many people underestimate the difficulty of this project. You cannot implement this converter using replacements on the level of strings; you really have to parse the file into a datatype encoding internal representation of a "Markdown structure". Parsing nested lists is particularly a challenge, and you will need to adapt the parsing techniques learned in class for this purpose.

Only choose this project if you have a pretty clear idea/outline of how you will accomplish these tasks.

3. Advanced Functional Programming

3.1. Functional programming with algebraic datatypes (1)

In class, we learned about basic *algebraic* data types. These include both recursive and non-recursive data types like

```
data List  a = Nil | Cons a (List a)
data Maybe a = Nothing | Just a
data Tree  a = Leaf a | Node (Tree a) (Tree a)
data LTree a = LLeaf  | LNode a (LTree a) (LTree a)
```

```
data RTree a = RLeaf  | RNode [Tree a]
data Either a b = Left a | Right b
data AExpr a =  AVar a | Const Integer | Add (AExpr a) (AExpr)
             | Sub (AExpr a) (AExpr a) | Sub (AExpr a) (AExpr a)
data Prop a =        Var a | And (Prop a) (Prop a) | Or (Prop a) (Prop a)
            | Not (Prop a) | Imp (Prop a) (Prop a) | Iff (Prop a) (Prop a)
```

We also learned about `fold`s, which capture generic recursion patterns over such types. For example, `foldr` captures generic recursion over lists, and `treeFold` captures generic recursion over trees.

But such generic combinators exist for every datatype like above.

In this project, you have to define such folds for arithmetic expressions, propositional formulas, and at least one more recursive datatype, which implement generic patterns for traversing such datatypes. Then you have to redefine previously implemented recursive functions on these types — such as `fv`, `eval`, `subst`, etc. — *purely* in terms of these generic "Aexpr folds", "Prop folds", etc. That is, you must reimplement these functions *without* recursion, using instead the generic recursion combinators you defined for each.

3.2. Nested types and other advanced datatypes (2)

All of the algebraic datatypes listed in the previous topic are *homogenous*, in that the *recursive* reference in the definition is always applied to the same type variable, `a`.

But Haskell also supports *non-homogeneous* datatypes, also known as *nested types*. Here are some examples:

Perfect Trees. `data PTree a = PLeaf a | PNode (PTree (a,a))`

This datatype encodes "full binary trees", having leaves with exactly 2^n data elements at depth n.

Lambda terms. `data Lam a = Var a | App (Lam a) (Lam a) | Abs (Lam (Maybe a))`

This is an encoding of lambda terms as a (nested) *type constructor*. The type parameter `a` encodes the type of *variables* that can occur inside the term.

Here the unary `Abs` constructor *increases* the number of variables that the lambda terms can contain.

One can think of terms of this datatype as trees with binary or unary nodes, in which the data at the leaves depends on how many unary nodes are above it. Every unary node between the root and the leaf adds one more possible value to the data at the leaf — represented by the `Nothing` constructor of the `Maybe` datatype.

Bushes. `data Bush a = BLeaf | BNode a (Bush (Bush a))`

This is an example of a "truly" nested type, where the recursive occurrence of the type constructor is applied to *itself!*

Nested types can be tricky to think about. However, many of them still support the same `map` functions, generic `fold`s, `bind`s, and so on.

In this project, you have to implement such functions for the above nested types. You should start with a selection of functions that are available for lists, and see how many of them can be generalized to the three datatypes above.

You can also experiment with defining your own nested type.

3.3. Advanced I/O (1)

In class, we learned some very basic I/O. You could make a project in which you explore more advanced input/output programming in Haskell. For example, you could write a program that searches the filesystem starting from the current directly for files that match a pattern, opening each of them, making some small replacements or extracting some data to be displayed to the user, then saving and closing them.

3.4. An application of Zipper (1-2)

The `Zipper` is a very useful concept and technique in functional programming, which encapsulates many patterns of traversing complex data types.

For more information, see

- The last chapter of the Haskell tutorial is dedicated to the Zipper:
  `http://learnyouahaskell.com/zippers`
- The original paper that introduced the concept:
  `https://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf`
- Derivatives of datatypes. (Not strictly about programming with zipper, but it goes a long way to explaining what it represents conceptually.):
  `http://strictlypositive.org/diff.pdf`

3.5. ⋆ Write a "Double Quine" in Haskell (1-2)

If *Hello World* is often one of the very first programs one writes when learning a new language, requiring minimal familiarity with the syntax and basic I/O, being able to write the *Quine* is a major milestone on the other end of the spectrum of mastery, and it often requires deep understanding of the language's constructs and possibilities.

A *Quine* is a program which prints its own *source code* to standard output. The program is not allowed to actually open and read its own source file (that's cheating!)

By the Church–Turing thesis, every Turing-complete language has a Quine. But it can be quite a tricky conceptual exercise to actually find one.

(If you are not familiar with this idea, take a moment to imagine how you might write a Quine in your favorite programming language.)

Unfortunately, there are many Quines that can be found for Haskell by googling around, so this could not serve as a project on its own right.

However, there are a number of variations of the same idea that could serve as an idea for a project. One interesting twist is to have a program that prints out the source code of *another, different* program, where running *that* program prints out the source code of the original one.

4. Lambda calculus and type theory

4.1. Lambda calculus/combinatory logic calculator (1)

The *lambda calculus* is the absolute minimal core of Haskell, which has just three types of syntactic constructs:

- Variables: `x`, `y`, `n`, `f`...
- Function application: `f x`, `fact n`, `append l1 l2`...
- Lambda abstraction: `\x -> x`, `\f x -> f (f x)`, `\x -> x x`, ...

The lambda calculus is *untyped*: application `f e` is allowed between any two terms `f` and `e`, without even considering the type of the function `f`.

In fact, this *type-free lambda calculus* is **the** first true Turing-complete programming language, which motivated the Church–Turing thesis on universal computation, and the very concept of Turing machines in the first place.

In this project you have to write a simulator of the lambda calculus extended with some other operations.

4.2. Programming in λ-calculus (1-2)

The untyped lambda calculus is a Turing-complete language: any algorithm can be encoded in it.

The purpose of this project is to implement a non-trivial algorithm in the *pure* lambda calculus.

You should start by writing an interpreter for the lambda calculus. This is more or less already done in the lectures — see code file from Lecture 11.14. But you should clean up that code, make it geared toward the pure lambda calculus, and provide a basic front-end to test your programs.

Next, you should actually encode some datatypes, like pairs and lists, in the syntax Λ. There are a number of ways to do this, see `https://en.wikipedia.org/wiki/Lambda_calculus#Encoding_datatypes`. See if you can encode the insertionSort by continuing the approach given in `programmingInT.hs` from Lecture 11.18.

4.3. A self-interpreter in LC or CL (2-3)

This is a variation of the previous topic.

You should find a way to encode trees in the lambda calculus, and then encode the tree type of lambda terms. You should be able to write an evaluator that converts the

code of a term to the real term. Furthermore, this evaluator should be programmable in lambda calculus itself.

For extra difficulty, implement the following operation, with a pure lambda term. Given a term M, if M is the code of another lambda term N, output another term P such that, evaluating P gives back M. (In other words, it "requotes" the term a second time.)

4.4. Implement translation from LC to CL

Lambda Calculus and combinatory logic are very closely related: you can define explicit translations between them.

You can use the following datatypes:

```
type Vars = String
data Comb = Var | I | K | S | Comb Comb
data Lam  = Var Vars | App Lam Lam | Abs Vars Lam
```

Obviously, a combinator like `I`, `K`, `S` etc. can be easily translated into the lambda term $\lambda x.x$, $\lambda xy.x$, $\lambda xyz.xz(yz)$, etc.

Going the other way is actually one of the reasons why combinators were invented in the first place.

The following "abstraction algorithm" is essentially due to Moses Schönfinkel, who invented Combinatory Logic. First, we define a transformation on combinators — terms that only contain variables, constants, and applications. Let x be a given variable. The transformation is denoted by $\lambda^* x(-)$.

$$\begin{aligned}
\lambda^* x(x) &= \mathtt{I} \\
\lambda^* x(y) &= \mathtt{K}y \\
\lambda^* x(c) &= \mathtt{K}c \qquad c \in \{\mathtt{I}, \mathtt{K}, \mathtt{S}\} \\
\lambda^* x(st) &= \mathtt{S}\,\lambda^*(s)\,\lambda^*(t)
\end{aligned}$$

Finally, to define the transformation `[-] :: Lam -> Comb` from lambda terms to combinators, this transformation is applied inside out:

$$\begin{aligned}
[x] &= x \\
[st] &= [s][t] \\
[\lambda x.s] &= [\lambda^* x(s)]
\end{aligned}$$

Implement the above transformations in Haskell as well as a text front-end to test the result.

Note. The translation can be optimized by the following conditions:

$$\begin{aligned}
\lambda^* x(s) &= \mathtt{K}s \qquad \text{if } x \notin s \\
\lambda^* x(sx) &= s \qquad \text{if } x \notin s
\end{aligned}$$

You should implement these optimizations as well.

4.5. ⋆ A study of type isomorphisms (1-3)

Whereas in propositional logic one is often interested to find out which formulas are equivalent, leading to the laws of *Boolean algebra*, the same question about types leads to the problem of *type isomorphism:* when are two types "essentially equivalent", up to permutation of their elements?

Formally, two types A and B are *isomorphic* if there is a bijection between them. That is, A and B are isomorphic if there exist functions $f : A \to B$, $g : B \to A$ such that the following equations hold for all $x \in A$ and $y \in B$:

$$g(f(x)) = x \qquad f(g(y)) = y$$

When A and B are isomorphic, we write $A \simeq B$.

For example, in Haskell, there exists the following isomorphism between A and B, where $A =$ `(a,b) -> c` and $B =$ `a -> b -> c`:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f = \x y -> f (x,y)
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry g = h where h (x,y) = g x y
```

It is easy to see that `curry . uncurry = id` and `uncurry . curry = id`.

(That is, `curry` and `uncurry` are each other's inverses.)

In this project, you will go on a hunt for as many type isomorphisms as you can find.

One place to look for them is of course Boolean algebra. To translate a Boolean formula into a potential type isomorphism, you should take the Curry–Howard interpretation of the formula, see section 7.3.

For example, Boolean laws on the left give rise to type isomorphisms on the right:

$A \wedge B$	$=$	$B \wedge A$	`(a, b)`	$\simeq$	`(b, a)`
$\top \Rightarrow A$	$=$	A	`() -> a`	$\simeq$	`a`
$A \Rightarrow B \wedge C$	$=$	$(A \Rightarrow B) \wedge (A \Rightarrow C)$	`a -> (b , c)`	$\simeq$	`(a -> b, a->c)`
$A \vee B \Rightarrow C$	$=$	$(A \Rightarrow C) \wedge (B \Rightarrow C)$	`Either a b -> c`	$\simeq$	`(a -> c, b -> c)`

(Here $\Rightarrow$ is implication. $X \Rightarrow Y$ is true unless $X = \top$ and $Y = \bot$.)

Beware that *not all* of Boolean algebra identities will give rise to type isomorphisms! Understanding this difference better is part of the research component of this project. You will explore which Boolean algebra laws hold only because the variables are restricted to the two-element domain `Bool`, and which are more fundamental and hold true even when propositions are replaced by types.

(For example, which of the De Morgan laws are valid about types?)

For more on this topic, see:

`https://kseo.github.io/posts/2016-12-25-type-isomorphism.html`
`http://www.dicosmo.org/Articles/POPL92.pdf`
`http://www.dicosmo.org/ResearchThemes/ISOS/ISOShomepage.html`
`http://math.ucr.edu/home/baez/week202.html`

4.6. Implementing a type-checking engine (1-2)

Implement the Hindley–Milner type-inference algorithm for a significant extension of λT or PCF.

5. Learning a Haskell library

5.1. Advanced Parsing library

Haskell has many functional libraries dedicated to parsing. Especially interesting are the ones that are built around *parser combinators*. This is a very general way to approach parsing that exploits Haskell's higher-order functional features. Parser combinators may allow one to automatically generate a parser for a datatype directly from its definition.

One specific project idea could be to implement such a *parser generator*. This program should take a signature of terms or a context-free grammar. You should design a mini-language for defining such specifications. Then your program could generate a parser for this grammar.

This could be a potentially useful tool to have if you ever wish to implement your own language in the future.

5.2. Lens

Lens is a relatively recent addition to the Haskell toolkit, however it has found to be an extremely useful library for all sorts of applications.

In this project, you should learn about this library and use it with some specific application in mind.

This page has an introduction to this library: `https://www.fpcomplete.com/haskell/tutorial/lens/`

5.3. GUI in Haskell (1)

Here you are to learn a GUI library for Haskell and use it to implement a front-end to a Haskell application, such as an arithmetic or boolean calculator implemented in class.

There are a number of GUI libraries for Haskell, but they are all under rapid development and are thus somewhat of a moving target.

Here is an overview: `https://en.wikibooks.org/wiki/Haskell/GUI`

5.4. Other libraries

If you are interested in how to do achieve a specific task in Haskell, you could volunteer to learn a library dedicated to that task. Examples include:

- Aeson (a library for parsing JSON files)
- Yesod (Haskell web development framework)
- Scrapper (Web scraping library)

6. Theory topics

6.1. (N)DFA simulator: simulate a (Non-)deterministic finite accepter (1-2)

Write a Haskell program that simulates a deterministic finite accepter (DFA), or a non-deterministic finite accepter (NDFA).

The first has difficulty level 1, the second difficulty level 2.

The program should read the specification of the (N)DFA from a file, in the format of your own choosing, which will include the alphabet, the set of states, the initial and final states, and the transition function (or transition relation in the case of nondeterministic automata).

It should then prompt the user for the input string, simulate execution of the (N)DFA on that string, and output whether the string was accepted or not.

Then the program should prompt the user for another string and simulate the same (N)DFA again until the process is killed or a special "quit" instruction is given.

6.2. PDA simulator: simulate a Pushdown Automaton (2)

Write a Haskell program that simulates a Pushdown automaton (PDA).

The program should read the specification of the PDA from a file, in the format of your own choosing, which will include the input alphabet, the stack alphabet, initial and final states, transition function, etc.

It should then prompt the user for the input string, simulate execution of the PDA on that string, and output whether the string was accepted or not.

Then the program should prompt the user for another string and simulate the same PDA again until the process is killed or a special "quit" instruction is given.

6.3. TM simulator: simulate a Turing Machine (3)

Write a Haskell program that simulates a Turing machine (TM).

The program should read the specification of the TM from a file, in the format of your own choosing, which will include the input alphabet, the tape alphabet, the set of states, the initial and final states, the transition function, etc.

It should then prompt the user for the input string, simulate execution of the TM on that string, and output whether the string was accepted or not.

Then the program should prompt the user for another string and simulate the same TM again until the process is killed or a special "quit" instruction is given.

6.4. DFA minimizer: remove unnecessary states from a DFA (1)

Given a deterministic finite automaton, find an equivalent automaton with as few states as possible. Here "equivalent" means that it accepts the same language.

The program should read the specification of the DFA from a file in the format of your choosing, and then write the specification of the equivalent reduced DFA in the same format — either to a new file or to standard out.

6.5. NFA-to-DFA converter: determinize a given NDFA (2)

Given a non-deterministic finite automaton, find an equivalent deterministic finite automaton. Here "equivalent" means that it accepts the same language.

The program should read the specification of the NFA from a file in the format of your choosing, and then write the specification of the equivalent DFA in the same format — either to a new file or to standard out.

6.6. CFG transformer: convert a grammar to Chomsky Normal Form (1-2)

Given a context-free grammar, simplify it by removing redundant rules until Chomsky's Normal Form is reached.

The program should read the grammar from a file in the format of your choosing, and then write the reduced grammar in the same format — either to a new file, or to standard out.

Read more: `https://en.wikipedia.org/wiki/Chomsky_normal_form`.

6.7. NDFA to Regex converter (1)

You already know how to convert a regular expression to a NDFA: that is basically what a lexer does.

What about the converse? Given the specification of a (N)DFA, can you find a regular expression that describes the same language as that accepted by the finite automaton?

Your program should read the specification of NDFA from a file in the format of your choosing, and then write out on screen a regular expression generating the language accepted by the NDFA.

7. Logic

The topics in this section may require additional information on how to approach them. Feel free to contact me with questions or requests for clarification.

7.1. ⋆ Implementation of the unification algorithm (1-2)

The *unification algorithm* is a fundamental algorithm in artificial intelligence, used in theorem proving, type inference, knowledge representation, and other tasks.

Unification takes two expressions with variables as input, and computes *the most general* substitution for the variables that makes them identical. Examples:

- $x + y = 2 + 3$ has the only unifier $[x = 2, y = 3]$.
- $x + x = 2 + y$ has the only unifier $[x = 2, y = 2]$.
- $x + x = 2 + 3$ has no unifiers: there is nothing that can be substituted to make the two expressions equal.

- $x + x = y + z$ has as unifiers $[x = 1, y = 1, z = 1]$ and $[x = 2, y = 2, z = 2]$.

  These seem incompatible. But there is a *most general unifier*, or mgu for short, which is a substitution that leaves one of the variables unset: $[x = x, y = x, z = x]$.

- $x + x = 2 * y + z * 3$ has the only unifier $[x = 2 * 3, y = 3, z = 2]$

- $x + 3 * x = 2 + y$ has the only unifier $[x = 2, y = 3 * 2]$

- $x + 3 * x = 2 * y + y$ has no unifier, because it leads to a circular constraint $x = 2 * y = 2 * (3 * x)$, from where $x = 2 * (3 * x)$, which has no (finite) solution.

In this project, you are to implement an algorithm that computes this *most general unifier*. If your algorithm works for a particular datatype of terms, such as Prop or AExpr enriched with additional operations, this project would have *one difficulty point.*

For *two difficulty points*, you should implement a "generic" unification algorithm, which works for an arbitrary signature of terms

$$\Sigma = \{f_1, f_2, \ldots, f_k\}$$

where function symbol f_i has a specified arity $a_i \geq 0$.

Your program should be able to read the signature specification from a separate file. The (set of) equations to be unified could also be stored in the same file, or queried interactively from the user.

Example. Let $\Sigma = \left\{f^2, g^1, h^3, a^0, b^0\right\}$ be a signature with one binary, one unary, one ternary operator, and two constants. This can be seen as a vector of *function symbols* $(f_1, f_2, f_3, f_4, f_5) = (f, g, h, a, b)$ with *arities* $(a_1, a_2, a_3, a_4, a_5) = (2, 1, 3, 0, 0)$.

The most general unifier of a given equation or system of equations is computed as

follows. (In fact, the algorithm will work with systems of equations.[1])

$$\{f(g(f(a,x)),h(y,x,y)) \quad = \quad f(g(w),h(v,b,w))\}$$

$$\xrightarrow{\mathsf{split}f} \begin{cases} g(f(a,x)) & = & g(w) \\ h(y,x,y) & = & h(v,b,w) \end{cases}$$

$$\xrightarrow{\mathsf{split}g} \begin{cases} f(a,x) & = & w \\ h(y,x,y) & = & h(v,b,w) \end{cases}$$

$$\xrightarrow{\mathsf{swap}} \begin{cases} w & = & f(a,x) \\ h(y,x,y) & = & h(v,b,w) \end{cases}$$

$$\xrightarrow{\mathsf{subst}[w=f(a,x)]} \{h(y,x,y) \quad = \quad h(v,b,f(a,x))\}$$

$$\xrightarrow{\mathsf{split}h} \begin{cases} y & = & v \\ x & = & b \\ y & = & f(a,x) \end{cases}$$

$$\xrightarrow{\mathsf{subst}[y=v]} \begin{cases} x & = & b \\ v & = & f(a,x) \end{cases}$$

$$\xrightarrow{\mathsf{subst}[x=b]} \begin{cases} v & = & f(a,b) \end{cases}$$

$$\xrightarrow{\mathsf{subst}[v=f(a,b)]} \{\}$$

The path of this process produces the substitution

$$[w = f(a,x), y = v, x = b, v = f(a,b)]$$

This is the most general unifier. By applying the chain of substitutions backwards, one obtains the ground instance

$$[w = f(a,b), y = f(a,b), x = b, v = f(a,b)]$$

Notice that your algorithm will need to implement the *occurs check* to catch circular constraints — otherwise it will loop forever.

To read more, google Martelli–Montanari algorithm, or read the Wikipedia page about unification.

The following page contains a rather short description of the algorithm.
`https://www.cmi.ac.in/~madhavan/courses/pl2009/lecturenotes/lecture-notes/node113.html`

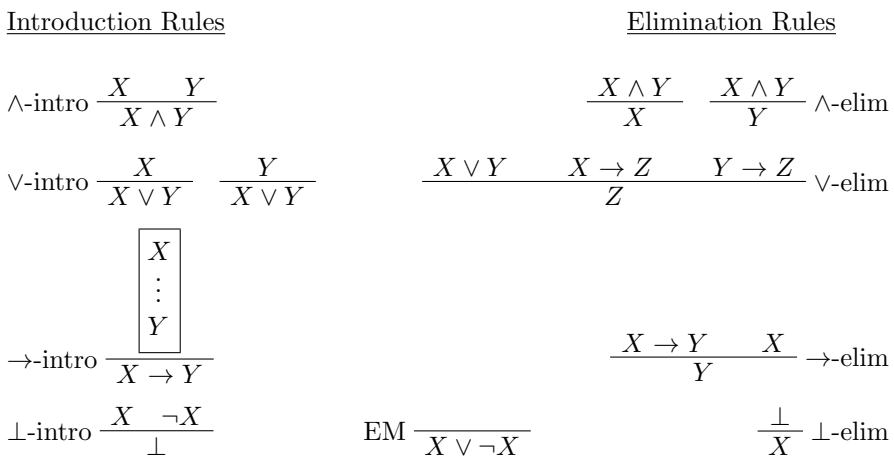
[1] A system is a list of equations. An equation is a pair of terms. A term is a tree whose nodes are labelled with function symbols, and the number of children in the node matches the arity of the function.

7.2. Propositional prover (2)

In class, you implemented algorithms that decide satisfiability, tautology, contradiction, and equivalence of propositional formulas.

If you are familiar with proofs, you may take the next step and implement a *theorem prover*: a program which, given a formula that is a tautology, will construct a representation of a proof of the formula using sound inference rules of propositional logic.

Such a proof can be represented as a sequence of formulas, where each new formula is derived from the existing ones using one of the propositional inference rules. Here is a list of such rules with which every propositional tautology can be derived:



7.3. ⋆ The proofs-as-programs paradigm (2-3)

Better yet, you could use the *Curry–Howard isomorphism* to encode propositional formulas *directly* into Haskell's type system!

The following transformation from formulas to types is applied:

```
|A -> B|     ==>     |A| -> |B|
|A /\ B|     ==>     (|A| , |B|)
|A \/ B|     ==>     Either |A| |B|
| !A |       ==>     |A -> False|
|True|       ==>     ()
|False|      ==>     Empty
```

Here `Empty` is like `void`: a type with no inhabitants. It is not built into Haskell by default, but it can be simulated by defining the following datatype:

```
data Empty = Null Empty
```

Notice how there is absolutely no way to ever construct any data of this type, because there is a recursive case, but no base case. At the same time, one can implement the ⊥-elim rule associated to the `False` proposition, known as the "Principle or Explosion", or, in Latin, *ex falso quodlibet*: From the False proposition, everything is provable!

```
exFalso :: False -> a
exFalso (Null e) = exFalso e
```

In this project, you are to implement a *constructive* prover: Given a formula A, which is a tautology, you are to *automatically generate* a Haskell program of type $|A|$.

As a first step, you should implement every inference rule above when the logical connectives are translated into types. Most of them are quite easy, for example $\rightarrow$-elim (Modus Ponens) is just application:

```
MP :: (a -> b) -> a -> b
MP f x = f x
```

There is only one rule that you will not be able to implement: the Excluded Middle rule. Logic without this rule is known as *constructive logic*.

Since the proof script that results with the above encoding is a Haskell program, such proof is completely *algorithmic*: You can actually *run the proof*! The intuitive interpretation of this dynamical nature of proofs is that a proof of a theorem is a procedure, or a function, that *transforms* any evidence that the assumptions are true into evidence that the conclusion is true.

7.4. Playing with Dependent types (2-4)

Agda is a programming language built on top of Haskell which implements a powerful new feature: *dependent types*.

This is a fundamental change, which raises the expressivity level of the language from mere programming to representing abstract mathematics. With dependent types, the above *proofs-as-programs* analogy is generalized to first-order logic. This allows us to prove both abstract tautologies in predicate logic as well as the properties of Haskell programs themselves!

For example, you could prove the following fact in Agda:

```
(l1 : List a) -> (l2 : List a) -> Equal (len (append l1 l2)) (len l1 + len l2)
```

Here `Equal` is a type constructor that *depends* on values of type `Integer`. It can be defined generically for every type just like an ordinary datatype with only one constructor:

```
data Equal :: a -> a -> type where
  refl :: (x : a) -> Equal x x
```

Thus, predicates can be encoded just like other datatypes, and proving predicate logic formulas them is just like programming any other Haskell function — by pattern matching and recursion. (In this context, recursion encodes various forms of induction.)

In this project, you will learn some Agda and prove nontrivial theorems using dependent types.

A good place to start is the Agda tutorial: `http://learnyouanagda.liamoc.net/pages/introduction.html`
`http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf`

7.5. First-order prover (Resolution or Tableau-based) (5)

In this project, you must implement a prover for predicate logic, which supports substitution of terms for variables, universal and existential quantifiers, and equality. This project will be a lot of work and you should only take it up if you have a rough idea of what it will entail.

(NB. Unlike Propositional logic, provability of first-order formulas is *not* decidable.)

8. Computer algebra system functionality

Most mathematical algorithms you have encountered thus far work with numbers: they perform *numerical computation*. Computer algebra systems do mathematics as well, but they are primarily about manipulating expressions algebraically. That is, they perform *symbolic computation*.

8.1. Linear systems solver (1-2)

The input is a system of linear equations, for example:

$$\begin{cases} x + y = 5 \\ 3x - 2y = 10 \end{cases}$$

The output is `x=4, y=1`. The output can also be "no solution", or "infinitely many solutions" (in which case two distinct solutions should be output, or an expression parametrizing this infinite set of solutions).

Your program should be able to handle systems having any number of variables and equations. The input should be read from a file specified by the user. You can define your own language for representing systems of equations.

8.2. ⋆ Polynomial division (2-3)

Did you know you can use long division to divide *polynomials* just like you use it to divide numbers?

For example, let

$$P(x) = x^4 + 2x^3 - 2x - 1$$
$$Q(x) = x - 1$$

Then $P(x)/Q(x)$ can be computed as follows:

$$\begin{array}{r} x^3 + 3x^2 + 3x + 1 \\ x - 1 \overline{)\, x^4 + 2x^3 + 0x^2 - 2x - 1} \\ \underline{x^4 - x^3} \\ 3x^3 - 0 \\ \underline{3x^3 - 3x^2} \\ 3x^2 - 2x \\ \underline{3x^2 - 3x} \\ x - 1 \\ \underline{x - 1} \\ 0 \end{array}$$

At every step, we record above the line what the divisor $x - 1$ must be multiplied by to match the highest exponent in the current stage. Then we multiply and subtract, which reduces the exponent by one.

With higher-degree polynomials, it works the same way:

$$\begin{array}{r} x^2 - x - 2 \\ x^2 - 3x + 2 \overline{)\, x^4 - 4x^3 + 3x^2 + 4x - 4} \\ \underline{x^4 - 3x^3 + 2x^2} \\ \quad - x^3 + x^2 + 4x \\ \quad \underline{-x^3 + 3x^2 - 2x} \\ \quad\quad - 2x^2 + 6x - 4 \\ \quad\quad \underline{-2x^2 + 6x - 4} \\ \quad\quad\quad 0 \end{array}$$

If the first polynomial does not divide perfectly into the second one, there will be a remainder below the last line:

$$\begin{array}{r} x^2 - x - 5 \\ x - 2 \overline{)\, x^3 - 3x^2 - 3x + 1} \\ \underline{x^3 - 2x^2} \\ \quad - x^2 - 3x \\ \quad \underline{- x^2 + 2x} \\ \quad\quad - 5x + 1 \\ \quad\quad \underline{- 5x + 10} \\ \quad\quad\quad - 9 \end{array}$$

Thus, we conclude that

$$x^3 - 3x^2 - 3x + 1 = (x - 2)(x^2 - x - 5) - 9$$

which can also be written as

$$\frac{x^3 - 3x^2 - 3x + 1}{x - 2} = (x^2 - x - 5) - \frac{9}{x - 2}$$

In this project, you have to implement the above "long division for polynomials". The program should prompt the user for two polynomials, and print out their quotient and the remainder (if any).

You should think about the best way to represent polynomials in Haskell. One simple solution could be to just use a list of numbers — of either `Rational` or `Double` type — to represent the coefficients. The length of the list would then be one more than the degree of polynomial.

This project can be scaled up in difficulty by building on the division function to get a full-fledged "polynomial calculator". In addition to adding, subtracting, multiplying, or dividing polynomials, the calculator could support computation of gcd of polynomials, finding derivatives of polynomials, substitution of one for the variable of another, and functions for simplifying polynomial expressions down to some standard form.

8.3. Symbolic differentiation (1-2)

In Calculus, you learned the rules for computing the derivative of *elementary functions*. These are all the functions you can get by combining polynomials, fractions, exponentials, and trigonometric

functions. Using the Chain Rule, any composition of such functions can be differentiated: to find the derivative of a product, you use Produce Rule, to find the derivative of a fraction, you use Quotient Rule, etc.

In this project, you should define a datatype of functions, which extends the type of arithmetic expressions defined in class, to include these additional unary operators (`sqrt(x)`, `sin(x)`, `cos(x)`,...) and binary operators (`x^y`, `log_x(y)`,...). The type of constants should include all real numbers, represented by double-precision floating point.

Then you should define a function which, given an expression, will recursively apply the appropriate differentiation rules to construct a new expression, representing the derivative of the given one.

The difficulty level of this project can be adjusted by implementing more functions, and/or by allowing differentiation with respect to more than one variable.

As with other proposals, a complete project should have a datatype of expressions to be differentiated, a format for entering such expressions in plain text, a parser which reads the text into the internal datatype, and a UI to interact with the user.

8.4. Integration of rational functions (2-4)

As is known, integration is much more difficult than differentiation. Many elementary functions, like e^{x^2} and $\frac{sin(x)}{x}$, do not have elementary anti-derivatives.

However, for *rational* functions $r(x)$ — which are quotients of polynomials $r(x) = \frac{p(x)}{q(x)}$, where $p(x)$ and $q(x)$ are polynomials — one can use the method of *partial fractions* to compute any integral and express it in terms of elementary functions.

In this project, you have to implement such a symbolic integrator of rational functions.

Read more: `https://www.math24.net/integration-rational-functions/`

8.5. ⋆ Linear recurrence solver (3-5)

In Computer Science, one often needs to solve recurrence relations. For example, the sequence defined by $a_0 = 1$, $a_{n+1} = 2a_n$ gives rise to the sequence

$$(a_0, a_1, a_2, a_3, \ldots) = (1, 2, 4, 8, \ldots)$$

This recurrence has an *explicit solution* $a_n = 2^n$, which is a formula involving only n, with no recursive reference to a_n for smaller values of n.

(For example, such calculations can be useful if you need to estimate the worst-case running time of a tree traversal algorithm, and thus need to compute the maximum number of nodes in a binary tree of height n.)

A *second-order* linear recurrence depends on not one, but two preceding values. The most famous second-order recurrence is probably the Fibonacci sequence

$$a_0 = 0, \quad a_1 = 1, \quad a_{n+2} = a_{n+1} + a_n$$

As another example, consider the recurrence relation

$$a_0 = 2 \qquad a_1 = 1 \tag{1}$$

$$a_{n+2} = a_{n+1} + 2a_n \tag{2}$$

which generates the sequence

$$(a_0, a_1, a_2, a_3, \ldots) = (2, 1, 5, 7, 17, 31, \ldots) \tag{3}$$

This sequence does not have appear to have any obvious pattern. (Or does it?)
In such a situation, how can we ever find an explicit formula for it?

It turns out there is a pretty simple algorithm to find solutions to such problems.

1. First, convert the recurrence relation above into the so-called *characteristic polynomial*, by applying the transformation $a_{n+k} \longmapsto x^k$ to recurrence (2):

$$\begin{aligned} a_{n+2} = a_{n+1} + 2a_n &\longmapsto x^2 = x^1 + 2x^0 \\ &\longmapsto x^2 = x + 2 \\ &\longmapsto x^2 - x - 2 = 0 \end{aligned}$$

2. Next, we find the roots of the characteristic polynomial.

   Since $x^2 - x - 2 = (x-2)(x+1)$, we get that $x = 2$ and $x = -1$ are the roots i.e. the values of x that make the equation equal 0.

   For general second-order recurrences, the biggest exponent of x is 2, and so we can always find the roots using the quadratic formula if necessary.

3. The general solution to the recurrence (2) has the form

$$a_n = \alpha \cdot (2)^n + \beta \cdot (-1)^n \tag{4}$$

   For general second-order recurrences, if there are two distinct roots of the equation r_1 and r_2, then the general solution will be $a_n = \alpha r_1^n + \beta r_2^n$.

4. It remains to find α and β. This can be done by plugging (4) into the initial conditions (1) and solving the resulting linear system:

$$\begin{aligned} &\begin{cases} 2 = a_0 = \alpha \cdot (2)^0 + \beta \cdot (-1)^0 = \alpha \cdot 1 + \beta \cdot 1 & = \alpha + \beta \\ 1 = a_1 = \alpha \cdot (2)^1 + \beta \cdot (-1)^1 = \alpha \cdot 2 + \beta \cdot (-1) = 2\alpha - \beta \end{cases} \\ \implies &\begin{cases} \alpha + \beta = 2 \\ 2\alpha - \beta = 1 \end{cases} \implies \begin{cases} (\alpha + \beta) + (2\alpha - \beta) = 2 + 1 \\ 2 \cdot (\alpha + \beta) - (2\alpha - \beta) = 2 \cdot 2 - 1 \end{cases} \\ \implies &\begin{cases} (\alpha + 2\alpha) + (\beta - \beta) = 3 \\ (2\alpha - 2\alpha) + (2\beta + \beta) = 3 \end{cases} \implies \begin{cases} 3\alpha = 3 \\ 3\beta = 3 \end{cases} \implies \begin{cases} \alpha = 1 \\ \beta = 1 \end{cases} \end{aligned}$$

5. Plugging these values of α, β into (4) yields the solution:

$$a_n = 2^n + (-1)^n$$

   You can check this formula against the original sequence (3).

To check your understanding of the above procedure, you can try to find an explicit formula for a much more famous second-order recurrence, the Fibonacci numbers.
(*Hint.* The characteristic polynomial for the Fibonacci sequence cannot be factored easily. If you find roots involving the square root of 5, you are on the right track!)

In this project, you have to implement a solver of such linear recurrences. If you wish to work in a large group, there are many additional features you could implement to make the project even more challenging. These include:

- Allow the possibility of repeated roots.

  For example, if the recurrence is $a_{n+2} = 4a_{n+1} - 4a_n$, then the characteristic polynomial is

  $$x^2 - 4x + 4 = (x-2)^2$$

  which has both roots equal to 2. In this case, the solution is of the form $a_n = \alpha(2)^n + \beta n(2)^n$. In general, if r is a repeated root of the equation, the solution is

  $$a_n = \alpha r^n + \beta n r^n$$

- Allow the possibility of complex roots.

  The characteristic equation of recurrence $a_{n+2} = -a_n$ is $x^2 + 1 = 0$, which does not cross the x-axis, so it has no real solution. Nevertheless, it has complex solutions $r_1, r_2 = \pm i$, which can be treated just as the real ones. Imaginary roots signal periodic behavior. For example, if the initial data is $a_0 = 1$, $a_1 = 2$, then the sequence generated by $a_{n+2} = -a_n$ is

  $$(1, 2, -1, -2, 1, 2, -1, -2, \ldots)$$

- Allow non-homogeneous terms in the recurrence, e.g., $a_{n+1} = a_n + n$

  Such recurrences can be tricky to solve, but there are techniques that can handle many special cases.

- Allow recurrences of degree higher than 2. This is by far the hardest extension, due to difficulty of finding roots of cubic equations and other higher-degree polynomials.

You can read more about this topic online by googling "solving linear recurrences".

9. Pass the tests! Marry the princess! (3-5)

The following puzzle was invented by American logician Raymond Smullyan, and appears in his book "The Riddle of Scheherazade, and other amazing puzzles". It is quoted directly from there in the following passage.

> A certain prince Al-Khizr was in love with the sultan's daughter and asked for her hand in marriage.
>
> "My daughter is very choosy," said the sultan, "and wants to marry only someone who shows extraordinary intelligence. So if you want to marry her, you must first pass eight tests."
>
> "What are the tests?" asked the suitor.
>
> **THE FIRST TEST** "Well, for the first test, you have to write down a number that will be sent to the princess. She will then send back a number to you. If she sends back the very same number that you have sent her, then she will allow you to take the second test. But if her number is different from yours, then you are out."
>
> "Now, how can I possibly know what number to write?" asked the suitor. "How can I guess what number the princess has in mind?"
>
> "Oh, she doesn't have a number in mind," said the sultan. "The number she sends back is dependent on the number you send. The number you send completely determines what number she will send back. And if you send the right number, then she will send back the same number."

"Then how can I guess the right number?" asked the suitor.

"It's not a matter of guessing," said the sultan. "You must deduce the correct number from the rules I am about to give you. For any numbers x and y, by xy I mean not x times y but x followed by y, both numbers, of course, written in base ten Arabic notation. For example, if x is 5079 and y is 863, then by xy I mean 5079863. Now here are the rules:

Rule 1: For any number x, if you write her 1x2, then she will send you back the number x. For example, if you write 13542, she will write back 354.

Rule 2: For any number x, the repeat of x means xx. For example, the repeat of 692 is 692692. And now, the second rule is that if x brings back y, then 3x will bring back the repeat of y. For example, since 15432 brings back 543, then 315432 will bring back 543543. From which it further follows that if you send her 3315432, you will get back 543543543543 (since 315432 brings back 543543).

Rule 3: The reverse of a number means the number written backwards. For example, the reverse of 62985 if 58926. The third rule is that if x brings back y, then 4x brings back the reverse of y. For example, since 172962 brings back 7296, then 4172962 brings back 6927. Thus, if you send her the number 4172962, you will get back 6927. Or, combining Rules 1, 2, and 3, since 316982 brings back 698698 (by Rules 1 and 2), then 4316982 brings back 896896.

Rule 4 (The Erasure Rule): If x brings back y, and if y contains at least two digits, then 5x brings back the result of erasing the first digit of y. For example, since 13472 brings back 347, then 513472 brings back 47.

Rule 5 (The Addition Rule): If x brings back y, then 6x brings back 1y and 7x brings back 2y. For example, since 15832 brings back 583, then 615832 brings back 1583, and 715832 brings back 2583.

"Those are the rules," said the sultan, "and from them can be deduced a number x that will bring back the very number x. There are actually an infinite number of solutions, but any single one will suffice for passing the first test."

"Are there any meanings to these numbers?" asked the suitor.

"Ah, that is the princess' secret, but fortunately you don't have to know the meaning in order to pass the first test."

THE SECOND TEST For the second test, the suitor had to send the princess a number, x, such that she would send back the repeat of x—the number xx. What x would work?

THE THIRD TEST

For the third test, the suitor had to send the princess a number x such that she would send back the reverse of x. What number x would work? An extra bonus would be given if the number x contains no more than twelve digits. What x would work?

THE FOURTH TEST For this test, the suitor had to send a number x such that the princess would send back the number x with its last digit erased. What x would work?

THE FIFTH TEST For this test, the suitor had to send a number x such that the princess would send back a different number y, which the suitor was to send back to

the princess, and she would (hopefully) send back the first number x. What number x would work?

THE SIXTH TEST The suitor now had to send a number x, get back a number y, return y to the princess, and get back the reverse of the original number x. What number x would work?

THE SEVENTH TEST For this test, the suitor was to send a number x, get back a number y, return y to the princess, and get back the number x with the first and last digits interchanged. What number x would work?

THE EIGHTH TEST For the final test, the suitor was to send a number x; the princess would then send back a number y; the suitor was then to send her the reverse of this y; the princess would then send back a number of the form zz (a number z repeated); the suitor was then to break zz in half (so to speak) and send her back z. The princess would then (hopefully) send back the original number x. What number x would work?

In this project, you should do the following:

1. Solve the eight tests above;
2. Define a minimal programming language with the only datatype being strings and the only operations representing the five rules given at the begining of the puzzle.
3. Write an interpreter for this language. (This includes parsing and evaluation, but the whole thing can be done in about 20 LoC.)
4. Test your solutions by running them with your interpreter.
5. For extra points, investigate whether this language is Turing-complete. If so, encode the factorial function in it. Otherwise, give example of an algorithm that cannot be encoded in the language.