15-150 Fall 2014
Homework 08

Out: Tuesday 4th November, 2014
Due: Tuesday 11th November, 2014 at 23:59 AST

1 Introduction

This homework will get you to practice working with sequences and analyzing the asymptotic complexity of sequence-based code. It will also give you a preview of the kind of code you will be writing in 15-210 *Parallel and Sequential Data Structures and Algorithms*, the follow-up course to 15-150.

This homework has also the purpose of exposing you to the experience, very common in practice, of working with a large body of code written by others. Although the amount of code you will need to write yourself is small, you will have to read a lot documentation, sometimes not written as precisely as you wish. Some of that documentation will not be in this file, but in various signatures. In fact, you will need to do a lot of figuring out on your own. How do you like this for a real-world experience?[1]

1.1 Getting the Homework Assignment

The starter files for the homework assignment have been distributed through Autolab at

`https://autolab.cs.cmu.edu`

Select the page for the course, click on "hw08", and then "Download handout". Uncompressing the downloaded file will create the directory `hw08-handout` containing the starter files for the assignment. The directory where you will be doing your work is called `hw/08` (see below): copy the starter files there.[2]

[1] Well, this is actually a lot better than the real world: the documentation is pretty complete and nothing it says is wrong. Moreover we are giving you a lot of hints throughout the homework. Oh wait, that's what they always say!

[2] The download and working directory have different names so that you don't accidentally overwrite your work were the starter files to be updated. **Do not do your work in the download directory!**

1.2 Submitting the Homework Assignment

Submissions will be handled through Autolab, at

https://autolab.cs.cmu.edu

In preparation for submission, your `hw/08` directory will need to contain the file `hw08.pdf` with your written solutions and the files `sequtil.sml`, `assignment.sml`, `prop.sml`, `cnf.sml`, `prop2.sml` and `test-all.sml` with your code. The starter code for this assignment contains incomplete versions of these files. You will need to complete them with the solution to the various programming tasks in the homework.

To submit your solutions, run

```
make
```

from the `hw/08` directory on any Unix machine (this include Linux and Mac). This will produce a file `hw08-handin.tgz`, containing the files to be handed in for this homework assignment. Open the Autolab web site, find the page for this assignment, and submit your `hw08-handin.tgz` file via the "Handin your work" link. If you are working on AFS, you can alternatively run

```
make submit
```

from the `hw/08` directory. That will create `hw08-handin.tgz`, containing the files to be handed in for this homework assignment and directly submit this to Autolab.

All submission will go through Autolab's "autograder". The autograder simply runs a series of tests against the reference solution. Each module has an associated number of points equal to the number of functions you need to complete. For each such function, you get 1.0 points if your code passes all tests, and 0.0 if it fails at least one test. Click on the cumulative number for a module for details. Obtaining the maximum for a module does not guarantee full credit in a task, and neither does a 0.0 translate into no points for it. In fact, the course staff will be running additional tests, reading all code, and taking into account other aspects of the submitted code such as structured comments and tests (see below), style and elegance.

To promote good programming habits, your are limited to a maximum of 6 submissions for this homework. Use them judiciously! In particular, make sure your code compiles cleanly before submitting it. Also, make sure your own test suite is sufficiently broad.

Remember that your written solutions must be submitted in PDF format—we do not accept MS Word files or other formats.

The SML files `sequtil.sml`, `assignment.sml`, `prop.sml`, `cnf.sml`, `prop2.sml` and `test-all.sml` must contain all the code that you want to have graded for this assignment, and must compile cleanly. If you have a function that happens to be named the same as one of the required functions but does not have the required type, it will not be graded.

1.3 Due Date

This assignment is due on Tuesday 11th November, 2014 at 23:59 AST. Remember that there are no late days for this course and you will get 2 bonus points for every 12 hours that you submit early.

1.4 Code Structure

On this and future assignments, we will be grading your programs on more than just their input-output behavior. It's not enough to have programs that happen to work: they need to clearly state what they do, have some empirical evidence that they work as advertised, and be easy for other people to read and reason about.

You must use the following five step methodology for writing functions, for *every* function you write in this assignment:

1. In the first line of comments, write a call template of the function.
2. In the second line of comments, specify via a `REQUIRES` clause any assumptions about the arguments passed to the function.
3. In the third line of comments, specify via an `ENSURES` clause what the function computes (what it returns).
4. Implement the function (include type annotations for the arguments and result of the function)
5. Provide test cases, generally in the format
   `val <return value> = <function> <argument value>`.

For example, for the factorial function presented in lecture:

```
(*  factorial (n) ==> res
 *  REQUIRES:  n >= 0
 *  ENSURES: res is  n!
 *)
fun factorial (0: int): int = 1
  | factorial n = n * factorial (n-1)

(* Tests: *)

val 1 = factorial 0
val 720 = factorial 6
```

1.5 Testing Modules

Because modules encourage information hiding, the way to test SML structures and functors is a bit different from what you did in the past. In fact, outside of a module, you may have no way to view the values of an abstract type. This means you can't compare the result of an operation with the expected value because you have no way to construct this expected value.

So, how to test modular code? There are essentially two ways to proceed.

Inside-the-box testing: You can't build values outside your module, but you can do so inside (typically). Then, what you would do is to put your normal tests inside the structure you are working on. As usual, if a test fails, a `binding non exhaustive` exception will be raised.

This is a bit trickier to do with functors, because you may not have a way to build values that depend on the functor's parameters. In this case, outside-the-box testing is your only option.

Outside-the-box testing: Many modules export a printing function and an equality function (conventionally called `toString`) and `eq`, respectively). You can then use the equality function to test that the value returned by a function is the value you expect. You can use the printing function to visualize returned values of hidden type.

When a module does not provide such functions, it exports operations that interact with each other, somehow. You can leverage these interactions for testing purposes. For example, a dictionary exports `insert` and `lookup` operations. You may test a module implementing dictionaries by populating a dictionary using `insert` and then use `lookup` to check that the expected entries are in it (and that unexpected entries are not).

Best of all, you want to use a combination of inside- and outside-the-box testing. Notice that inside-the-box testing is implementation-dependent, but outside-the-box is not.

1.6 Style

Programs are written for people to read — it's convenient that they can be executed by machines, but their high level text is a way for one person to explain an idea to another person. Your code should reflect this, and we will grade your code on how easy it is to understand.

The published style guide is your primary resource here. Strive to write clear, concise, and elegant code. If you have any questions about style, or just have a feeling that a piece of code could be written more simply, don't hesitate to ask!

1.7 The Compilation Manager

We will be using several SML files in this assignment. In order to avoid tedious and error-prone sequences of `use` commands, we will use SML's *compilation manager*. The compilation manager (CM) is a system that keeps track of what files have been modified and runs just them (and the files that depend on them) through SML. If you have used `make` on a Unix system, the idea is very similar.

Using CM is simple. In fact, there are two ways to do so:

Go to the directory containing your work and run at the terminal prompt (written `#`):

```
# sml -m sources.cm
```

or

Launch SML from the directory containing your work, and then run at the SML prompt (written `-`):

```
- CM.make "sources.cm";
```

Both wil load all the files listed in `sources.cm`[3] and take you to the SML prompt. Do so whenever you change your code. No need to call `use` — in fact you may confuse CM. For large programs, CM offers a better interface to the command line. There is less typing and less of an issue with name shadowing between iterations of your code.

In short, on this assignment, the development cycle will be:

1. Edit your source files.
2. Type either `sml -m sources.cm` at the terminal prompt or `CM.make "sources.cm";` at the SML prompt.
3. Fix errors and repeat.

`CM.make` creates a directory called `.cm` in the current working directory. It gets populated with metadata needed to work out compilation dependencies. The `.cm` directory can safely be deleted at the completion of this assignment (in fact, it can become quite large)

It's sometimes happens that the metadata in the `.cm` directory gets into an inconsistent state — if you run `CM.make` with different versions of SML in the same directory, for example. This often results in bizarre error messages. When that happens, it is safe to delete the `.cm` directory and compile again from scratch.

[3]The file `sources.cm` contains a list of the files tracked by CM. Feel free to take a peek if you are curious!

1.8 Common Sequence Functions

For your convenience, here is a brief description of some of the functions on sequences.

- `Seq.length: 'a Seq.seq -> int`
  `Seq.length s` returns the number of elements in the sequence `s`.

- `Seq.nth: int -> 'a Seq.seq -> 'a`
  `Seq.nth i s` returns the element of sequence `s` at index `i` (starting from 0), assuming it is in bounds.

- `Seq.tabulate: (int -> 'a) -> int -> 'a Seq.seq`
  `Seq.tabulate f n` computes a sequence of length `n` such that the value of each element of the sequence is the result of applying the function `f` to its index.

- `Seq.map: ('a -> 'b) -> 'a Seq.seq -> 'b Seq.seq`
  `Seq.map f s` returns a sequence whose elements are the result of applying the function `f` to the corresponding element in the sequence `s`.

- `Seq.reduce: ('a * 'a -> 'a) -> 'a -> 'a Seq.seq -> 'a`
  `Seq.reduce g e s` combines all the elements of the sequence `s` using the binary function `g` and base value `e`.

- `Seq.mapreduce: ('a->'b) -> 'b -> ('b * 'b -> 'b) -> 'a Seq.seq -> 'b`
  `Seq.mapreduce f e g s` chains the functionalities of `Seq.map` and `Seq.reduce` by applying the function `f` to each element of the sequence `s` before combining them as in `Seq.reduce` with `g` and `e`.

- `Seq.filter: ('a -> bool) -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.filter f s` returns that sequence obtained by keeping only the elements of `s` on which `f` returns `true`.

- `Seq.empty: unit -> 'a Seq.seq`
  `Seq.empty ()` returns the empty sequence.

- `Seq.singleton: 'a -> 'a Seq.seq`
  `Seq.singleton x` returns the one-element sequence containing the value of `x`.

- `Seq.append: 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`
  `Seq.append s1 s2` returns the sequence obtained by appending sequence `s2` to the end of sequence `s1`.

- `Seq.flatten: 'a Seq.seq Seq.seq -> 'a Seq.seq`
  `Seq.flatten S` returns the sequence obtained by appending all sequences in `S`.

Function	**Inputs**	**Work**	**Span**	**Note**
`Seq.length s`	$n =$ size of `s`	$O(1)$	$O(1)$	
`Seq.nth i s`	$n =$ size of `s`	$O(1)$	$O(1)$	
`Seq.tabulate f n`		$O(n)$	$O(1)$	[1]
`Seq.map f s`	$n =$ size of `s`	$O(n)$	$O(1)$	[2]
`Seq.reduce g e s`	$n =$ size of `s`	$O(n)$	$O(\log n)$	[2]
`Seq.mapreduce f e g s`	$n =$ size of `s`	$O(n)$	$O(\log n)$	[2]
`Seq.filter f s`	$n =$ size of `s`	$O(n)$	$O(1)$	[2]
`Seq.empty ()`		$O(1)$	$O(1)$	
`Seq.singleton x`		$O(1)$	$O(1)$	
`Seq.append s1 s2`	$n_1 =$ size of `s1` $n_2 =$ size of `s2`	$O(n_1 + n_2)$	$O(1)$	
`Seq.flatten s`	$n =$ sum of sizes of `s`	$O(n)$	$O(1)$	

Notes:

[1] assuming $O(1)$ work and span for `f`.

[2] assuming $O(1)$ work and span for `f`. Multiply by $W_{\texttt{f}}(n)$ or $S_{\texttt{f}}(n)$ if work and span of `f` is independent of its input, but dependent on n. Same thing for `g`.

Printing Sequences

To print a sequence, you can use the function `Seq.toString: ('a -> string) -> 'a Seq.seq -> string`. The first argument is a function that prints the individual elements of the sequence. For example,

```
Seq.toString Int.toString (Seq.tabulate (fn i => i+1) 12)
```

Testing with Sequences

To test code returning sequences, you can use the function `Seq.eq: ('a * 'a -> bool) -> ('a Seq.seq * 'a Seq.seq) -> bool`. The first argument is a function that tests the equality of pairs of elements. For example,

```
Seq.eq (op=) (Seq.tabulate (fn i => i+1) 12, Seq.tabulate (fn j => 1+j) 12)
```

Conversion to Lists

The functions `Seq.toList: 'a Seq.seq -> 'a list` and `Seq.fromList: 'a list -> 'a Seq.seq` allow converting to and from a list. Their span is $O(n)$ where n is the size of the input sequence or list. They should therefore be avoided except for actual list conversion.

2 Warmup

This section has the purpose of giving you familiarity with working with sequences. You can use any function in the `SEQUENCE` signature, with one exception: you are not allowed to reduce these problems to their `list` equivalents.

Write your code and its specs in the file `sequtil.sml` and test it in `test-all.sml`.

Task 2.1 (5 points) SML was designed as a sequential language which means that no pure SML program can take advantage of parallelism. In particular, when evaluating a pair `(e1,e2)`, an SML interpreter will first evaluate `e1` and only when it gets a value for it does it start evaluating `e2`.[4] By contrast, sequences are parallel data structures, which means that a multi-processor architecture could process most operations on them in parallel on their elements.

Using sequences, define the SML function

```
ppar: ('a -> 'b) -> 'a * 'a -> 'b * 'b
```

such that `ppar f (x,y)` ≅ `(f x, f y)` but so that the two applications of `f` are carried out in parallel. Said differently, the span of evaluating `ppar f (x,y)` shall be the maximum of the span of evaluating `f x` and `f y`, while the span of evaluating `(f x, f y)` is their sum.

Task 2.2 (5 points) Implement the SML functions

```
all:  bool Seq.seq -> bool
some: bool Seq.seq -> bool
```

such that, given a sequence `B` of Boolean values, `all B` returns `true` if each value in `B` is `true` and `false` otherwise, and `some B` returns `true` if at least one value in `B` is `true` (and `false` otherwise). *Your code should not be recursive.*

Task 2.3 (3 points) Determine the work and span of `all` and `some`. Justify all steps.

[4] Therefore, we misled you when using pairs to motivate the notion of span: pairs are evaluated sequentially — we will see why they can't be evaluated in parallel towards the end of the course.

3 Propositional Formulas

A *propositional formula* φ is a logical expression built out of *propositional letters* (A, B, C, $\ldots$), the truth values *true* (written $\top$) and *false* (written $\bot$), and the logical connectives *conjunction* ($\wedge$), *disjunction* ($\vee$) and *negation* ($\neg$). Propositional formulas are expressed in SML by the following type declaration:

```
datatype fla = Var of R.letter
             | True
             | False
             | And of fla * fla
             | Or  of fla * fla
             | Not of fla
```

where `R.letter` is the type of propositional letters and `R` is a structure that ascribes to signature `LETTER`. Two implementations are provided in the starter code.

There are a lot interesting questions that one can ask about a propositional formula φ:

- If we assign specific truth values to the propositional letters in φ, what is the truth value of φ overall?
- Is φ *valid*? I.e., does every assignment of truth values to its propositional letters make φ *true*?
- Is φ *satisfiable*? I.e., are there assignments of truth values to its propositional letters that make it *true*? If so, what are these assignments?
- Is φ *unsatisfiable*? I.e., does every assignment of truth values to its propositional letters make it *false*?

The questions in this exercise will explore these questions.

3.1 Truth Value Assignments

Rather than using `True` and `False` of type `fla` as our truth values, it will be convenient to use the two Boolean values of SML (`true` and `false`).

Therefore, we can model a *truth value assignment* as a sequence of pairs, with the first component being a propositional letter and the second component the truth value this letter is assigned to:

```
type assignment = (R.letter * bool) Seq.seq
```

This representation is subject to the invariant that each propositional letter appears at most once in an assignment. The use of sequences will allow us to process assignments in parallel.

For example, the truth value assignment that make `A` *true*, `B` *false* and `C` *true* will be the sequence ⟨`("A",true)`, `("B",false)`, `("C",true)`⟩.

The signature `assignment.sig` defines the following operations on assignments:

```
val toString: assignment -> string
val eq: assignment * assignment -> bool
val valid: assignment -> bool
val fromList: R.letter list * R.letter list -> assignment
val toList: assignment -> R.letter list * R.letter list
val eval: R.letter -> assignment -> bool
val genAll: R.letter Seq.seq -> assignment Seq.seq
```

Here is what they do:

- `toString A` returns a string representation of assignment `A`.
- `eq (A1,A2)` tells if two assignments are equal, i.e., if they assign the same truth value to each of the propositional letters in them.
- `valid A` checks if a truth assignment is valid, i.e., if no propositional letter occurs more than once in it. If the representation invariant is satisfied, it should always return `true`.
- `fromList (P,N)` returns the truth assignment such that each propositional letter in `P` is *true* and each letter in `N` is *false*. You may assume that `P` and `N` are disjoint.
- `toList A` returns a pair of lists `(P,N)` such that all letters assigned truth value *true* in `A` appear in `P` and all letters assigned *false* appear in `N`.
- `eval l A` returns the truth value of letter `l` in assignment `A`. You may assume that `l` occurs in `A`.
- `genAll L` returns the sequence containing all the possible assignments of truth values to all the letters in `L`. You may assume that `L` does not contain duplicates.

The functions `toString`, `eq` and `valid` will be your main tools to debug your code. The first two are pre-implemented for you in the starter file `assignment.sml`.

Task 3.1 (35 points) Give an SML implementation for all of the above functions, save `toString` and `eq`, in file `assignment.sml` and test it in `test-all.sml`. *Your code shall not use recursion*, not even in any helper function you may define. Except in `toList` and `fromList`, your code shall not make use of lists.

Task 3.2 (10 points) Determine the work and span of each function in `assignment.sml`. Justify all steps.

3.2 CNF-Conversion

Now that we know about truth value assignments, it is fairly easy to write functions that answer the questions posed at the beginning of this section. Because propositional formulas can be highly nested, doing so can be achieved at best in time proportional to the *nesting depth* of the formula, defined as the largest number of constructors encountered before we reach a propositional variable or the logical constants $\top$ or $\bot$. We can do better by first converting the formula to conjunctive normal form.

A formula in *conjunctive normal form* (abbreviated *CNF*) is a conjunction of disjunctions of either propositional letters or their negation. A propositional letter or its negation is called a *literal*. A disjunction of literals is called a *clause*. The logical constant $\bot$ is understood as the empty clause (the clause with zero literals), and the logical constant $\top$ is understood as the empty conjunction of clauses. Therefore, a CNF formula is a conjunction of clauses, a very simple, two-level structure.

We represent literals in SML as follows:

```
datatype literal = Pos of R.letter
                 | Neg of R.letter
```

where `Pos l` and `Neg l` corresponds to `Var l` and `Not (Var l)`, respectively. Then a CNF formula is represented by the type `literal Seq.seq Seq.seq`. Literals and some operations over them are defined signature `LITERAL` in `literal.sig`.

Two formulas φ_1 and φ_2 are *logically equivalent*, written $\varphi_1 \equiv \varphi_2$, if they both evaluate to the same truth value for any truth value assignment. Every propositional formula can be transformed into a logically equivalent CNF formula by using a few simple equivalences:

$$\begin{array}{rcll}
\neg(\varphi_1 \wedge \varphi_2) & \equiv & \neg\varphi_1 \vee \neg\varphi_2 & \text{(De Morgan law)} \\
\neg(\varphi_1 \vee \varphi_2) & \equiv & \neg\varphi_1 \wedge \neg\varphi_2 & \text{(De Morgan law)} \\
\neg\top & \equiv & \bot & \text{(Definition of negation)} \\
\neg\bot & \equiv & \top & \text{(Definition of negation)} \\
\neg\neg\varphi & \equiv & \varphi & \text{(Double negation elimination)} \\
\varphi \vee (\varphi_1 \wedge \varphi_2) & \equiv & (\varphi \vee \varphi_1) \wedge (\varphi \vee \varphi_2) & \text{(Distributivity)} \\
\varphi \vee \top & \equiv & \top & \text{(Absorption)}
\end{array}$$

as well as the versions of the last two where the sides of $\vee$ have been swapped.

In the following tasks, you will be converting an arbitrary propositional formula φ into CNF. As you do so, your code shall be as parallel as possible, yielding an overall span of $O(d)$ where d is the nesting depth of φ — recall that pairs are *not* evaluated in parallel in SML. Write the code for these functions in file `prop.sml` and test them in `test-all.sml`.

Task 3.3 (5 points) Implement the SML function

```
pushNot: fla -> fla
```

such that `pushNot F` is a formula `F'` that is logically equivalent to `F` but such that all negations are attached to propositional letters. Thus, `F'` consists of literals, `True` and `False` combined by conjunctions and disjunctions. Some of the above equivalences will come useful for this purpose. If `F` has nesting depth d, the span of `pushNot F` will be in $O(d)$.

Task 3.4 (8 points) Implement the SML function

```
distribute: fla -> fla
```

such that, given a formula `F` where negations appear only attached to propositional letters, the call `distribute F` returns a formula `F'` where no conjunction or truth occurs below a disjunction. `F'` is to all effects in CNF. Again, some of the above equivalences will come useful and, if `F` has nesting depth d, the span of `distribute F` will be in $O(d)$.

Task 3.5 (5 points) Implement the SML function

```
toCNF: fla -> L.literal Seq.seq Seq.seq
```

such that, given a formula `F` where no conjunction or truth occurs below a disjunction and all negations are attached to propositional letters, `toCNF F` returns the CNF representation of `F`. If `F` has nesting depth d, the span of `distribute F` should be in $O(d)$.

Task 3.6 (5 points) Determine the work of each of the above functions and verify that they all have span $O(d)$ where d is the nesting depth of the input.

3.3 Validity and All That

We are now in a position to answer the questions we asked at the beginning of this section. The signature `CNF` specifies the following functions on CNF formulas:

```
val toString: cnf -> string
val eq: cnf * cnf -> bool
val toLists: cnf -> L.literal list list
val fromLists: L.literal list list -> cnf
val letters: cnf -> L.R.letter Seq.seq
val eval: cnf -> A.assignment -> bool
val valid: cnf -> bool
val satisfiable: cnf -> A.assignment Seq.seq
val unsat: cnf -> bool
```

where `A` and `L` are structures that ascribe to signatures `ASSIGNMENT` and `LITERAL`, respectively. These functions have the following behavior:

- `toString F` returns a string representation of CNF formula `F`.

- `eq (F1,F2)` determines whether `F1` and `F2` are equal CNF formulas.
- `toLists F` returns a list of all the clauses in `F` where each clause is represented as a list of literals.
- `fromLists Ls` returns the CNF formula whose clauses are the conjunction of the literals in each list in `Ls`.
- `letters F` returns the sequence of the propositional letters occurring in `F`, without duplicates.
- `eval F A` returns the truth value of CNF formula `F` with respect to truth value assignment `A`. Each propositional letter in `F` is required to occur in `A`.
- `valid F` determines whether `F` is valid, i.e., whether it evaluated to *true* on every truth value assignment.
- `satisfiable F` returns the sequence of all the truth value assignments for which `F` is satisfiable.
- `unsat F` determines whether `F` is unsatisfiable.

The functions `toString` and `eq` are pre-implemented in file `cnf.sml` to facilitate debugging.

Task 3.7 (30 points) Give an SML implementation for all of the above functions, save `toString` and `eq`, in file `cnf.sml` and test it in `test-all.sml`. *Your code shall not use recursion*, not even in any helper function you may define. Except in `toLists` and `fromLists`, your code shall not make use of lists.

Task 3.8 (10 points) Determine the work and span of each function `cnf.sml`. Justify all steps.

3.4 Generalized Conjunction and Disjunction

Propositional formulas used in practice often contain long chains of conjunctions of the form $\varphi = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$. Using a binary operator like $\wedge$ is cumbersome as we need to choose a certain associativity, for example $((\ldots(\varphi_1 \wedge \varphi_2) \wedge \ldots) \wedge \varphi_n)$. It also hinders performance because, in this example, we need to traverse $n-1$ conjunctions to get to φ_n. We would rather view φ as a flat collection of the conjuncts $\varphi_1, \ldots, \varphi_n$, something that is sometimes written

$$\varphi \quad = \quad \bigwedge_{i=1}^{n} \varphi_i$$

thereby making $\wedge$ an n-ary operator, for any n. The same considerations apply to $\vee$.

We can capture this idea in SML by having the constructors `And` and `Or` take a sequence of formulas as arguments, rather than just two formulas. This leads to the following type definition:

```
datatype fla = Var of R.letter
             | And of fla Seq.seq
             | Or  of fla Seq.seq
             | Not of fla
```

for an appropriate structure `R` that ascribes to signature `LETTER`. Observe that `True` and `False` are absent as they can be expressed as the empty conjunction and disjunction respectively.

We can now implement all the functionalities seen earlier for propositional formulas on this new definition.

Task 3.9 (25 bonus points) Give an SML implementation of the operations in file `prop2.sml` and test them in `test-all.sml`. Your code shall not use recursion except to traverse values of type `fla` above. Except in the operations that explicitly convert to and from lists, your code shall not make use of lists.

Task 3.10 (8 bonus points) Determine the work and span of each function in `prop2.sml`. Justify all steps.