COL703: Logic for Computer Science
I semester 2018-19

Assignment 3: DPLL for a restricted FOL

1 The actual assignment

In this assignment, you will implement the DPLL algorithm [1] for first order logic, for bounded models. The signatures in SML and Ocaml respectively contain the datatype for the formula.

The SML version of the signature

```
(* SML signature *)
signature Assignment3 =
sig
    datatype term = C of string (* constant symbols *)
                  | V of string (* variable symbols *)
                  | F of string * (term list)

    datatype form = PRED of string * (term list) |
                    NOT of form |
                    AND of form * form |
                    OR of form * form |
                    FORALL of term * form | (* term can only be a variable *)
                    EXISTS of term * form   (* term can only be a variable *)

    exception Not_wff of (term list * form list)

    exception Not_closed of term list

    exception DPLL_unsat of int

    val wff: form -> bool
    val fv: form -> term list
    val closed: form -> bool
    val scnf: form -> form
    val dpll: form * int -> (term list * form list)
    val sat: form * int -> (bool * term list * form list)
end (* sig *)
```

The OCaml version of the signature.

```
(* OCaml signature *)
module type Assignment3 = sig

    type term = C of string | V of string | F of string * (term list)

    type form = PRED of string * (term list)
                  | NOT of form
                  | AND of form * form
                  | OR of form * form
```

```
                | FORALL of term * form (* This term should be a variable only*)
                | EXISTS of term * form (* This term should be a variable only*)
  exception Not_wff of (term list * form list)
  exception Not_closed of term list
  exception DPLL_unsat of int
  val wff: form -> bool
  val fv: form -> term list (* Term list consists of variable only*)
  val closed: form -> bool
  val scnf: form -> form
  val dpll: form -> int -> (term list * form list)
  val sat: form -> int -> (bool * term list * form list)
end;;
```

Input The input to the function `sat` is a formula ϕ and a positive integer k.

Output The output of `sat`(ϕ, k) is either a sequence of k exceptions raised by `sat` or for some $j : 1 \leq j \leq k$, a sequence of $j-1$ exceptions raised by `sat` and a final Herbrand model of size j is output.

1. A formula is *well-formed* if no function or predicate symbol show multiple arities and every quantifier binds only a single variable.

   - $p(f(a), f(a)) \wedge \neg p(a)$ is not well-formed because the two occurrences of the predicate symbol p have arities 2 and 1 respectively.
   - $p(f(a), f(a))$ is a well formed formula, but $p(f(a), f(a, b))$ is not since the two occurrences of the function symbol f have different arities (1 and 2 respectively).

   Implement function `wff` to check whether the terms occurring in a predicate are well-formed and whether the predicate is well-formed. If not raise the exception `not_wff` and give the list of atomic predicates and function symbols that have multiple arities.

2. The function `dpll` only takes closed formulae. Implement a function `fv` which yields the list of free variables in a formula. The function `closed` determines whether the given input formula is closed. If it is not closed raise the exception `not_closed` and give the list of free variables.

3. To apply DPLL, one needs to convert the formula into skolem conjunctive normal form (SCNF) (refer slides for definition). Implement a function `scnf` to convert a closed formula into SCNF.

4. Now assume the body of the SCNF formula ψ is χ and an integer $k > 0$ is given. The aim is determine the smallest $j : 1 \leq j \leq k$ size Herbrand model containing j constants along with the truth assignment (given as a list of atomic predicates which must be assigned *true*) which satisfies the formula ϕ. For each call to `dpll` that does not have an i-sized model produce an exception `dpll_unsat` to show it failed.

   (a) For simplicity, assume that the input formula ϕ contains no function symbols (however you may have to generate constant symbols and/or function symbols while converting formula ϕ into the closed formula ψ in SCNF). Let $\psi \equiv \forall \vec{x}[\chi]$, where χ is the body of the SCNF formula ψ.

   (b) Starting with $j = 1$ determine whether χ has a 1-element Herbrand model. If it does then return the 1-element Herbrand model. If not try $j = 2$ and so on till $j = k$ if necessary.

   (c) For each such value of j the formula will reduce to a j-fold conjunction of ground formulae (which are essentially propositional forms).

   What you have to do is

   (a) Write a function `dpll` such that `dpll`(χ, j) determines whether there is a Herbrand model of ψ with exactly j elements, and if so, returns the model as a list of terms and a list of atomic ground predicates which need to be assigned *true* (the other atomic ground predicates are assumed to be assigned *false*).

(b) Write a function `sat` such that `sat`(ϕ, k) finds the smallest model of size some $j \leq k$ such that `dpll`(χ, j) yields the Herbrand model. `sat` takes the original formula ϕ as input. So `sat` would contain calls to `wff`, `fv`, `closed` and `scnf` before invoking `dpll`.

(c) To generate constants for each value of j starting from 1, use the column names used in spread-sheets (but lower-case) in that order. Hence the constants are of form `C(‘‘a’’)`, `C(‘‘b’’)`, ..., `C (‘‘z’’)`, `C(‘‘aa’’)`, `C(‘‘ab’’)` ... and so on.

(d) Appropriate exceptions using the name of the function from which exception was raised also need to be declared and used.

References [1] DPLL algorithm. https://en.wikipedia.org/w/index.php?title=DPLL_algorithm, Novem- ber 2016. Page Version ID: 748891330.

2 Instructions

General Instructions:

- This is an individual assignment. No consultation or sharing is allowed.
- You are free to use either Ocaml or SML.
- The example instructions and code that will be used for auto-grading refer to “Assignment3”. The “3” needs to be uniformly replaced throughout by some n for Assignment n.
- Your submission will be auto-graded, please ensure your submission conforms to the format specified below in the submission instructions. If auto-grade fails due to format mismatch, you will be awarded zero.

Submission Instructions:

Submission. Submit one .zip file. The name of the file should be <*yourKerberosLoginId*>`.zip`. On unzipping the file, it should produce one folder called <*yourEntryNumber*> and should contain all the source files as required. In addition, provide a `README.txt` that contains any instruction for the evaluator. Implementation details (beyond the usual comments in the program) may be provided in a separate file called <*yourKerberosLoginId*> with the appropriate extension `.docx`, `.rtf`, `pdf`, `.tex` etc.

Grading criteria. Depending upon the assignment different weightages will be assigned to the following.

- Correctness
- Efficiency
- Code readability (and documentation if any).
- Ease of access through `README.txt`

Implementation instructions for Ocaml. Your submission should have

1. Interface file named `signature.mli`. This file defines an interface for the datatype and functions. You may use the interface template provided below.

```
module type Assignment3 = sig
(* Your code goes here *)
end
```

2. Implementation file named `structure.ml`. This file implements the interface.

```
open Signature
module A3 : Assignment3 = struct
(* Your code goes here *)
end
```

3. **Note for auto grading**. The auto-grader script will unzip your submission and move one folder down. Ocaml files will be executed as shown below

   ```
   ocamlc signature.mli structure.ml testcase.ml
   ./a.out
   ```

   Sample testcase.ml file:

   ```
   open structure.Assignment3;
   (* The test cases go here.
      Below is a sample call to wff function *)

   (* FOL formula e.g: loves(romeo, juliet)  *)
   let t1 = C "romeo";;
   let t2 = C "juliet";;
   let fml = PRED("loves", [t1; t2]);;
   wff fml;;
   ```

4. References for OCaml:
   - https://v1.realworldocaml.org/v1/en/html/files-modules-and-programs.html
   - https://ocaml.org/learn/tutorials/modules.html

Implementation instructions for SML Your submission should have an

1. Interface file named `signature.sml`. This file defines an interface for the datatype and functions. You may use the interface template provided below.

   ```
   signature Assignment3 =
   sig
   (* Your code goes here *)
   end;
   ```

2. Implementation file named `structure.sml`. This file implements the interface as "A3".

   ```
   use "signature.sml";
   structure A3 : Assignment3 =
   struct
     (* Your code goes here *)
   end;
   ```

3. **Note for Auto grading**. The auto-grader script will unzip your submission and move one folder down. SML files will be executed as shown below.

   ```
   sml structure.sml testcase.sml
   ```

   Sample testcase.ml file:

   ```
   use "structure.sml";
   open A3;
   (* The test cases go here.
      Below is a sample call to wff function *)

   (* FOL formula e.g: loves(romeo, juliet)  *)
   val t1 = C "romeo";
   ```

```
val t2 = C "juliet";
val fml = PRED("loves", [t1, t2] );

A3.wff fml;
```

4. References SML:
   - https://www.cs.cmu.edu/ rwh/introsml/modules/sigstruct.htm
   - https://jozefg.bitbucket.io/posts/2015-01-08-modules.html

3 Notes, caveats, warnings

Note:

1. You are *not* allowed to change any of the names given in the signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.
2. You may define any new functions you like in the structure besides those mentioned in the signature.
3. Your program should work with the given signature.
4. You need to think of the *most efficient way* of implementing the various functions given in the signature so that the function results satisfy their definitions and properties.
5. Since the evaluator may want to look at your source code before evaluating it, you must explain your algorithms in the form of comments at appropriate places, so that the evaluator can understand what you have implemented.
6. Do *not* add any more decorations or functions or user-interfaces in order to impress the evaluator of the program. Nobody is going to be impressed by it.
7. There is a serious penalty for code similarity. If it is felt that there is too much similarity in the code between any two persons, then both are going to be penalized equally. So please set permissions on your directories, so that others cannot copy your programs.