Exam 2 Study Guide
CPSC 3520 - Brendan McGuire

Disclaimer: while I attempted to be as accurate as possible, I am a student just like you. If you see anything wrong or unclear here, please let me know and I will be sure to correct it!

Good luck today!

Prolog

A declarative programming language based on defining a series of statements, and querying the database of statements to find solutions. Prolog will then try all possible combinations to satisfy the problem.

Syntax & Semantics

A prolog file is composed of a number of *clauses*, which are statements that either define something to be true (fact) or declare a rule by which something *could be true* (rules).

```
% Facts                    % Rules
is_red(apple).             fruit(X) :- is_red(X), from_plant(X).
from_plant(apple).
```

- ➔ The red and orange sections together compose the predicate
- ➔ The purple section is the body of the clause. If no body is present, the statement is a fact, and the predicate is assumed to be true.

Note that the comma indicates that both predicates in the body must be true for the rule to be satisfied. Similarly, delimiting the predicates in the body with a semicolon indicates that ANY of the predicates may be true for the rule to be true. This statement would require fruit to match EITHER is_red OR from_plant:

```
fruit(X) :- is_red(X) ; from_plant(X).
```

Variables

In a clause, the predicate may specify one or more variables that apply to the particular clause. For example, X is a variable of fruit in the above example. If you do not care about what value the variable takes on, then use the anonymous variable by naming the variable underscore (_)

The Goal & Unification

After the predicates, a goal should be entered into the Prolog REPL. Prolog will attempt to unify this goal with the database (usually find all values of the argument which make the predicate true). In SDE1, the goal was specified to be the following.

```
bedplan(F1, F2, F3, F4, F5, F6).
```

In order to find the solutions to the stated, the prolog interpreter undergoes a process known as *unification*. The process of unification is shown below:

1. Clauses are tested in the order in which they appear in the database
2. When a subgoal matches the left side (head of the rule), the right side (tail) becomes a new set of subgoals to unify
3. The unifier proceeds from left to right in attempting to unify the predicates in the tail. When the subgoal is spawned, the unification/search process described in 1 repeats.
4. A goal is satisfied when a matching fact is found in the database for all leaves in the goal tree (all parts of the predicate have been satisfied)
5. When 2 or more clauses in the database with the same predicate name are identified as possible matches, the first one to appear is tested for unification. The remaining matches are marked for backtracking if the first case fails.

You may occasionally desire to prevent backtracking beyond a certain point in your predicates. This can be done with the cut (denoted with !), which is a goal that is always true but cannot be backtracked. It should be used sparingly.

Prolog Special Topics

Lists

In Prolog lists consist of elements separated by commas and enclosed with brackets like the following.

```
[a, b, c, d]
```

Just like constant values, lists may be used as arguments to predicates. Inside of predicates, list manipulation is based on the head (first) and tail (rest of the elements) of the list. This is denoted in statements like [H | T]

```
contains(X, [X | _]).
contains(X, [_ | Y]) :- contains(X, Y).
```

Logic Grammar Notation

Also known as Definite Clause Grammar is a special mode of Prolog that allows you to rapidly define grammars that can be used to construct parsers. The only significant difference between Logic Grammar Notation and regular Prolog clauses is the use of `-->` instead of`:-` for predicate definition.

For example, consider the set of productions below, which can be entered directly into the Prolog Logic Grammar Notation on the right.

```
S → AB
A → t1
B → t2
```

```
s --> a,b
a --> [t1]
b --> [t2]
```

Prolog will convert this notation into a set of actual predicates which can be used to build parsers, shown below.

```
s(A, B) :-
      a(A, C),
      b(C, B).
```

Prolog Functions

write – Text output to terminal

The write function can be used to pass output to the terminal. See below the rule print, which prints the given text, and adds a newline (nl).

```
print(X) :- write(X), nl.
?- print(“Hello World”).
      “Hello, World”
```

consult – Load a database from a file

Loads a database (set of predicates) from a file. Can then use these queries to form the goal of unification.

```
?- consult(“sde1.pro”).
?- bedplan(F1, F2, F3, F4). % Uses bedplan predicate defined in sde1.pro
```

protocol/noprotocol – Logging to a file

The protocol function call allows you to log the contents of the REPL session to a file. Similarly, call noprotocol to end logging.

```
?- protocol("output.txt").
?- noprotocol.
```

Lambda Calculus

An underlying model of computation (way to execute algorithms) based on the work of Alanzo Church. Despite its extreme simplicity, it is extremely expressive due to its ability to have functions applied to themselves, which allows for recursion. Similar to how the Turning Machine is a model of computation for imperative languages, Lambda Calculus is the underpinning for functional programming, especially for languages like LISP and ML.

The syntax of Lambda Calculus itself is extremely simple. For example, consider the identity function, which just returns its argument.

$$\lambda n.n$$

Functions can be called (applied) by using the following syntax.

$$((\lambda n.n)\ 4)$$

Polymorphism

Lambda calculus does not really have the concept of types, so functions often can be applied to many different types, each performing similar behavior. This concept is known as polymorphism (poly = many, morph = types)

Reduction Examples

Let's reduce the following lambda expressions to simpler forms by applying the operators.

Example #1

`((λx.x) ((λy.y) z))`	Original Expression
`((λx.x) z)`	Apply the argument z to the identity function `(λy.y)`
z	Apply the argument z to the identity function `(λx.x)`

Example #2

`(((λx.x) (λy.y))z)`	Original Expression
`((λy.y) z)`	Apply the argument `(λy.y)` to the identity function `(λx.x)`
z	Apply the argument z to the identity function `(λy.y)`

Currying (Multiple Arguments)

In Lambda Calculus, functions cannot have multiple arguments. Instead, this is resolved by having a function return *another function* which can then take its own argument. This process is repeated for as many arguments are needed.

```
((λx.(λy.x + y) 5) 8)
```

This is an anonymous function that takes an argument x, which returns a function that takes an argument y, that returns x + y. In this case, we apply 5 to `(λx.(λy.x + y)` to get `(λy.5 + y)`. We then apply 8 to get a final answer of 13

Lisp

Lisp extends the idea of Lambda Calculus into a more complete language and is the basis for many functional programming languages and paradigms. Traditionally, recursion and function application is preferred over loops and assignments, but there are some exceptions to make the language more practicable.

Interpreter (REPL)

A common way to interact with Lisp is via the interpreter, which is a *Read, Eval, Print Loop*. This means you will enter the text as input, the code will be run, and the output is shown on the screen. It will then prompt again, and usually preserves context like defined functions, global variables, etc.

```
* (print “Hello World”)

  “Hello World”
  “Hello World”
* (exp 1)
  2.7182817
```

Syntax and Semantics

The syntax of Lisp is intentionally desired to look like a developed version of Lambda Calculus, with similar syntax for function definition and evaluation.

Functions & Control Flow

As a descendant of Lambda Calculus, functions are the underpinning of Lisp. To start off, consider the following factorial function below.

```
(defun factorial (n)
(cond
      ((equal n 1) 1)
      (t (* n factorial (n - 1)))
)
```

This function uses the defun keyword to define a function called factorial, which accepts a single argument n. A cond statement is used for control flow. The structure of cond statements is shown below and can be extended with many different conditional statements.f

```
(cond
      (condition1 action)
      (condition2 action)
)
```

Note in the function above, we use the literal t (which presents a true value), to force the second conditional to always be true, essentially serving as an else. This conditional multiples n by the recursive call to the factorial of n-1, as per usual.

Assignment

One of the ways that Lisp breaks from complete functional programming orthodoxy is by set, which allows you to imperatively declare *lexical* variables.

```
[1]> (setq threehalfs 1.5)
1.5
[2]> (setq onepointfive `threehalfs)
THREEHALFS
[3]> onepointfive
THREEHALFS
[4]> (eval onepointfive)
```

There are few ways to set variables using Common Lisp, described below:

set – Sets a dynamic variable (or can be used to reassign a variable that already exists)

setq – Means set quoted. Allows you to create a lexical variable and set lvalues like a traditional language

setf – Often backfilled as set field, allows you to set an individual value through the lvalue

Example	Result	Explanation
`(set ls '(1 2 3 4))`	Error	Because ls is not quoted, Lisp looks for the value at ls and attempts to set it, but the variable ls does not exist.
`(set 'ls '(1 2 3 4))`	Create lexical variable ls	Because we quoted ls, Lisp considers it to be a name, so we can use it as a variable name
`(setq ls '(1 2 3 4))`	Create lexical variable ls	The same as above, setq just means set quoted
`(setf (car ls) 10)`	Changes the first element of ls to 10	The value being set in this statement is `(car ls)` the first element of the list ls

Lists, Single Quote, CAR, and CDR

An important concept in Lisp is the idea of a list. In Lisp, lists are notated in parentheses, shown below. *Note, the quote here is used to prevent Lisp from evaluating the list (i.e. trying to recognize 1 as a function)*

```
[1]> ‘(1 2 3 4 5)
   (1 2 3 4 5)
[2]> (car ‘(1 2 3 4 5))
   1
[3]> (setq asc ‘(1 2 3 4 5))
   (1 2 3 4 5)
[4]> (car asc)
   1
[5]> (cdr asc)
  (2 3 4 5)
[6]> (cadr asc)
   2
[7]> (caddr asc)
   3
[8]> (cadddr asc)
   4
```

In this example, we see a few different important concepts. First, notice the single quote before the list declarations. This single quote instructs lisp to not evaluate the expression given, and instead consider it a named symbol instead. Were this quote not present, the interpreter would attempt to evaluate that statement as a function 1 with arguments.

See below for an explanation for each function call:

car – returns the head of the list.
cdr – returns the tail (everything except for the head) of the list
cadr – gets the 2nd element of the list
caddr – gets the 3rd element of the list
cadddr – gets the 4th element of the list

ML, SML, and OCAML

The ML family of languages (SML, ML, OCAML) are defined by being strongly-typed functional languages, with a few imperative escape hatches (like setq in Lisp). Like Lisp, it focuses on having a strong *Function Basis Language* arranged hierarchically. These functions are defined *polymorphically*, akin to the Standard Template Library in C++.

Languages in the ML Family

SML (Standard ML) – Developed by Bell Labs and INRIA
CAML – A descendant of SML, developed by INRIA
OCAML – A descendant of CAML that added Object Oriented capabilities and a modules system

The REPL and Assignment

Just like Lisp, ML languages often operate from a REPL. In *OCaml* phrases are either simple expressions, let definitions, or identifiers (either constant values or functions)

Type System

There are a few basic types in ML, which form the basis of the *Function Basis Language*, and from which other types (like function signatures) can be formed.

int – Represents a 32bit integer, just like int in c
bool – Just like in C++, represents a boolean value
real – Represents an IEEE 32-bit floating-point number, just like float in C.
unit – The unit type is similar to void in c, and is used in situations where the type doesn't matter.
string – Collection of characters, just like in c++

Function Signatures & Tuples

Because of the strong type system and polymorphic bent, the nature and structure of function signatures in ML languages are worth particular attention. For starters, consider the following function definition enter at the REPL:

```
- fun twice x:int = 2 * x;
  val twice = fn: int -> int

- twice 10
  val it = 20 : int
```

The function signature of `twice` looks like `fn: int -> int` which means a function that takes an int as an argument and returns an int. Functions that need to take multiple arguments may use either using *Currying* to take a tuple as an argument. The latter is shown below.

```
- val a = (2.0, 3);
  val a = (2.0, 3) : real * int

- fun power(x, 0) = 1.0
  = power(x, n) = x * power(x, n-1)
  val power = fn real * int -> real
```

The first statement declares a tuple whose first type of real and second type is int. *Important to note: the star in the tuple type definition just separates the types of the tuple, it is not indicative of multiplication*. The second statement declares a function named power, which takes a real and an int as a tuple.

Control Flow

Unlike lisp, ML languages have a concept of if-else, but it is slightly different in the functional context than a more traditional imperative condition statement. For example, consider the following function definition for factorial. Notice how the value inside of the then and else is returned from the function.

```
val fact n:int = if n = 0 then 1 else x*fact(x-1)
```

Match

One unique feature of ML-based languages is the match statement, which is like a switch statement with superpowers. The syntax of match generally looks like the following:

```
match expr
with pattern_1 -> expr_1
|    pattern_2 -> expr_2
|    pattern_3 -> expr_3
|    _ -> default_expr
```

The key distinction between match and switch is more forms with which the cases can take on. For example, the following patterns can be in the case statements:

Constants (just like switch cases)
Wildcard: _ (can take on any value, like default in switch)
Variables: x (similar to wildcard, but you can reference the value in the resulting expression)
Tuples: (true, _) (can match individual parts of the tuple)
Constructors (which may contain other patterns):
    Lists: x::xs
    Other Datatypes: Node(L, x, R)

Like switch, you should try to make your match statements *exhaustive* (that is, they should handle every possible value of the argument). This is usually done with a wildcard match at the end. See below for some examples of how match can be used in ways that switch normally wouldn't allow.

```
match (x, y)
with (0, 0) -> 0 (* Match the origin *)
|    (x, 0) -> x (* matches all points on the x-axis *)
|    (0, y) -> y (* matches all points on the y-axis *)
|    (x, y) -> 1 (* matches all other points *)
```

Lists (OCAML)

Just like in Lisp, in ML languages, lists primarily operate based on the head and the tail of the list, and internally use linked lists. To see the syntax, consider the following function definition which checks for list membership.

```
let rec member = function
    (x, []) -> false
    | (x, h::t) -> if (h = x) then true else member (x, t) ;;
```

Here we define a recursive function member (see below for more information about recursive functions), which accepts a tuple with the first argument x, the item to test for, and the second argument being the list. The match statement has 2 cases. In the first case, if the list is empty then we return false. In all other cases, we first check for head equality, and then recurse on the tail.

Generics (CAML)

Consider the following function which returns the head of the given list.

```
let first input = List.hd(input);
val first : ‘a list -> ‘a = <fun>
```

This is an example of *polymorphic behavior,* as this function can work on all sorts of lists. In order to represent this, we define the list type to be ‘a, which means it can take on any value. Generics in ML languages are quite similar to templates in C++.

Recursive Function

As discussed above, if you have a function which is recursive, you must mark it with the rec keyword. This is used to prevent optimizations (like inlining) that might make recursion impossible.

Objects (OCAML)

The key distinction between CAML and OCAML is the introduction of Object-Oriented Principles. Here we define a class point.

```
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

Some key things to notice about this declaration:

1. The `int ->` specifies a constructor that can be used to obtain a point.
2. The value x is marked mutable, which means its value can be changed. *By default, these values are immutable*
3. The methods have definitions here just like normal

Module System (OCAML)

You can load code from another file by the function use (`use: string -> unit`), shown below.

```
use(“bedplan.sml”);
```