Lecture 10:
Functional Languages (Part II)

LISP

Elements of LISP

LISP (LISt Processing Language) was designed for manipulating symbols, rather than numbers. Therefore, it has become one of the most popular languages in the area of Artificial Intelligence (AI).

Several dialects of LISP have been developed for different goals

- Common LISP, aims at rich data structures and flexible constructs;
- Scheme, aims at simplicity;

I. LISP Basics

LISP's strength and weakness lie in its simple syntax:

```
<expression>  ::= <atom> | (<expressions>) | '(<expressions>)
<expressions> ::= <expression> | <expression> <expressions>
```

An atom can be an identifier, operator, or a number.

This minimal syntax helps to expose concepts of the programming language.

Some dialects, especially Common Lisp, add "syntactic sugar" to make programming easier without obscuring these concepts.

Because of its simplicity, LISP is often considered the "assembler language of AI".

Examples of Program Structures

All functions use *prefix* notation.

Compute the sum of a sequence of numbers:

```
(+ 1 2 3)
```

The arguments to a function call can be arbitrary LISP expressions, for example other functions. Thus, LISP supports *higher-order functions*.

Function definitions look like this:

```
> (defun sq (x)
      (* x x))
SQ
> (sq 3)
9
```


Examples of Program Structures (cont'd)

A conditional expression looks like an ordinary function called `if`. It has three arguments: the head, the then-branch and the else branch:

```
> (defun my-max (x y)
   (if (> x y) x y))
MY-MAX
> (my-max 1 2)
2
```

Recursion is heavily used in LISP:

```
> (defun fac(n)
         (if (< n 1) 1 (* n (fac (- n 1)))))
FAC
> (fac 5)
120
```

Examples of Data Structures

The basic compound data type is a *list*.

The empty list is denoted by `()` or `NIL`.

The constructor function `CONS` combines an element and a list into a larger list.

The destructor functions `CAR` and `CDR` (don't ask!) extract the head and the tail of a list.

```
> (defun my-length (l)
       (if (eq l NIL) 0 (+ 1 (my-length (cdr l)))))
MY-LENGTH
> (my-length (CONS 1 (CONS 2 NIL)))
2
```


II. Reflection in LISP

Whenever a list is found, the system tries to evaluate it, using the first list element as the function and the rest as its arguments. Therefore,

```
> (my-length (1 2 3 4 5))
*** - EVAL: 1 is not a function name
```

This implicit evaluation can be avoided by *quoting* a list. For example:

```
> (my-length '(1 2 3 4 5))
5
```

Separating Definition from Evaluation

LISP programs themselves are lists.

Therefore, we can assign an unevaluated LISP program to a variable.

```
> (setq p '(+ 1 2 3))
(+ 1 2 3)
```

The function `eval` takes a list and executes it.

```
> (eval p)
6
```

This shows how the definition of a computation can be separated from its evaluation.

An Example of Reflection in LISP

The combination of lists-as-programs and quoting gives the programmer the possibility to build and execute LISP expressions as data structures in a LISP program.

This technique is called *reflection*.

```
> (setq prg '(funcall my-fun 3 4))
(FUNCALL MY-FUN 3 4)
> (let ((my-fun (lambda (x y) (+ x y)))) (eval prg))
7
> (let ((my-fun (lambda (x y) (* x y)))) (eval prg))
12
```

III. Lexical vs Dynamic Scoping

Lexical (or static) scoping means that the context for an expression is determined by the place where its value is *defined*.

This is what you get in most programming languages.

As an example take the following definitions:

```
> (setq inc 1)
INC
> (defun up (x) (+ x inc))
UP
> (let ((inc 2))
       (up 5))
6
```


Lexical vs Dynamic Scoping (cont'd)

Dynamic scoping means that the context for an expression is determined by the place where its value is *required*.

This is what you typically get by *macro expansion*.

In Common Lisp both scoping rules are available. Variables that are introduced via the `defvar` command are dynamically scoped, and by convention use names enclosed by two `*`'s.

```
> (defvar *inc* 1)
*INC*
> (defun up (x) (+ x *inc*))
UP
> (let ((*inc* 2))
       (up 5))
7
```

Lexical vs Dynamic Scoping: Summary

Originally, LISP was defined to use dynamic scoping.

Dynamic scoping can be useful for global variables.

In general, dynamic scoping is hard to understand since you have to understand the dynamic behaviour of the program (its "unfoldings") in order to decide which variable is in scope.

The more recent LISP variants, such as Common LISP and Scheme, use lexical scoping.

IV. LISP in Every-day Use

Elisp is a dialect of LISP used in the EMACS editor.

It provides the full power of LISP, with higher-order functions etc, and a library of text processing functions to the programmer.

For example, let's develop some Elisp code to automatically launch specific applications when starting your EMACS.

We want to have one variable containing the list of applications to be launched on startup:

```
; define the commands to execute
(defvar my-invoke-list '(to-do calendar gomoku))
```

LISP in Every-day Use (cont'd)

Here is a list of all "launchable" applications:

```
; the function used to print log messages
(defvar my-log (function message))

; Applications that can be started from this init file
; Format of each entry: (NAME . CMD)
(defvar my-invoke-cmds
  '((test . my-startup-test)         ; for testing startup procedure only
    (w3 . w3)                         ; www browser
    (mh-e . mh-rmail)                 ; mh mail reader
    (vm . vm)                         ; vm mail reader
    (rmail . rmail)                   ; rmail mail reader
    (gnus . gnus)                     ; news reader
    (calc . calc)             ; GNU calc desktop calculator
    (to-do . todo-show)               ; GNU calc desktop calculator
    (calendar . calendar)
```


LISP in Every-day Use (cont'd)

```
    (diary . diary)
    (gomoku . gomoku)
    (get-comic . (lambda ()
                   (interactive)
                   (get-daily-comic "dilbert")))
    (escreen . (lambda ()
                 (interactive)
                 (mapcar (lambda (n) (escreen-create-screen))
                         (fromTo 2 (- escreen-max-screens
                                      escreen-current-screen-number)))))
    (multi-term . (lambda ()
                    (interactive)
                    (multi-term ?x t)))
   ))
```

LISP in Every-day Use (cont'd)

Finally, we use Elisp's version of `map`, called `mapcar`, and the concept of *higher-order functions*, to start these applications:

```
;; Now actually do the computations defined above
;; mapcar traverses the list my-invoke-list; for each list elementit executes
;; the corresponding command

(mapcar (function (lambda (x)
          (let ( (oldbuf (current-buffer))
                 (y (assoc x my-invoke-cmds)) )
           (if (null y)
              (funcall my-log (format "Error: No cmd found for package %s" x))
             (if (not (commandp (cdr y)))
              (funcall my-log (format "Error: %s is not an interactive fct" (cdr y)))
              (funcall my-log (format "= Starting  %s (cmd: %s)" (car y) (cdr y)))
             (funcall (cdr y)))))))
        my-invoke-list
)
```


Summary of Concepts in LISP

- LISP supports *higher-order functions*, i.e. functions can be passed as parameters to other functions and be returned as results.
- LISP allows to write *polymorphic functions*, i.e. functions that operate over data structure of different type (e.g. `my-length`).
- LISP programs are themselves lists, which provides the powerful tool of *reflection*.
- LISP distinguishes between two kinds of scoping for variables: *lexical scoping* and *dynamic scoping*.
- LISP is weakly typed, i.e. no compile-time type checking is performed.
- LISP contains purely functional and impure features such as destructive update (e.g. `setq`).

SML

An Overview of SML

"ML" stands for *Meta-Language*.

Originally used as a language for the LCF theorem proofer at Edinburgh University.

One of the first formally designed languages ⇒ good for formal reasoning.

Consists of a purely functional core, and some imperative features.

Several robust implementations are available: SML/NJ, CAML.

Main Conceptual Differences to Haskell

- SML is a **strict** language using eager evaluation.
- SML consists of a purely functional core language, extended with **imperative constructs**.
- SML has a different mechanism for modularisation and overloading.

I. Pattern Matching in SML

Our square function can be defined like this:

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

Here is the definition of factorial in SML:

```
> fun fact 0 = 1
   |  fact n = n * (fact (n-1));
val fact = fn : int -> int
```

II. Higher-order Functions in SML

The basic set of pre-defined higher-order functions is similar to those in Haskell.

```
> fun sqs xs = map sq xs;
val sqs = fn : int list -> int list

> sqs [1,2,3,4,5];
val it = [1,4,9,16,25] : int list
```

The SML version of a function computing the list of squares of even numbers from an input list can be written like this:

```
> val sq_even = map (fn x => x*x) o filter (fn x => x mod 2 = 0);
val sq_even = fn : int list -> int list

> sq_even [1,2,3,4,5];
val it = [4,16] : int list
```


III. Eager Evaluation

Let's revisit our example of extracting the n-th element from an infinite list.

The SML versions of getting the n-th element of a list (`get`) and generating a list of increasing integers starting with n (`enumFrom`) can be written like this:

```
> fun get 0 (x::xs) = x
   | get i (x::xs) = get (i-1) xs;
val get = fn : int -> 'a list -> 'a

> fun enumFrom n = n::(enumFrom (n+1));
val enumFrom = fn : int -> int list
```

An Example of Lazy Evaluation (revisited)

With **lazy evaluation** we *delay* all evaluation of arguments until their value is needed in the body of the function. We then evaluate the argument to *weak head normal form* (WHNF), i.e. we evaluate the top-level constructor only ("call-by-name", "normal order evaluation to weak normal form with sharing").

Here is a trace of the evaluation of `[1..]!!2` with lazy evaluation:

`[1..] !! 2`	⇒	*is the list empty?* (no: evaluate `[1..]` to WHNF)
`(1:[2..]) !! 2`	⇒	*is the index 0?*
`[2..] !! 1`	⇒	*is the list empty?*
`(2:[3..]) !! 1`	⇒	*is the index 0?*
`[3..] !! 0`	⇒	*is the list empty?*
`(3:[4..]) !! 0`	⇒	*is the index 0?*
`3`		

An Example of Eager Evaluation

SML uses *eager evaluation*, i.e. in a function application we first evaluate all arguments to normal form (to plain data) and then enter the function ("call-by-value", "applicative order evaluation to full normal form").

Here is the trace of evaluating `get 2 (enumFrom 1)` with eager evaluation:

`get 2 (enumFrom 1)`	⇒ *evaluate all arguments*
`get 2 (1::(enumFrom 2))`	⇒ *evaluate all arguments*
`get 2 (1::2::(enumFrom 3)`	⇒ *evaluate all arguments*
...	

We have a non-terminating program!

The Relationship between Semantics and Evaluation

Distinguish properties of the *semantics* of a language from the *evaluation* mechanism, which implements this property!

Semantics: In the semantics of a programming language the special value $\perp$ stands for an error or for non-termination (e.g. 1/0).

If for all user defined functions f the following holds

$$f \perp = \perp$$

then the language is strict.

Note: In a non-strict language a function may return a value even though one of its arguments produces an error.

Evaluation: Determines in which order and to what degree expressions are evaluated.

Which Evaluation for Which Semantics?

The evaluation mechanism that is used to implement a language must match its semantics.

In a **strict language** it is safe to perform **eager evaluation**, i.e. to evaluate all arguments before calling the function itself. If one of the arguments produces an error, then the semantics of the language guarantees that the function call should produce an error, too.

In a **non-strict language** we cannot use eager evaluation, since it would produce an error, where the language demands a proper value. Therefore, non-strict languages are typically implemented using **lazy evaluation**, i.e. the arguments of a function are only evaluated when their value is needed in the body of the function.

However, other evaluation mechanisms could be used to implement a non-strict semantics, e.g. parallel eager evaluation.

The Relationship between Semantics and Evaluation

The following table summarises the relationship between the semantics of the language and the evaluation mechanism used to implement the semantics:

	strict semantics	non-strict semantics
eager evaluation	$\surd$ (e.g. SML implementations)	$\times$ (error where well-defined)
lazy evaluation	$\surd$ (value where erroneous)	$\surd$ (e.g. Haskell implementations)

Using eager evaluation on a non-strict language will return errors where the result is well-defined (e.g. `foo 1 2 1/0`).

Using lazy evaluation on a strict language will return result values where the execution should report an error.

IV. Imperative Constructs in SML

SML can use imperative-style variables by providing the concept of a *reference* to a value as a separate datatype.

For example:

```
- val x = ref 1;
> val x = ref 1 : int ref
```

associates `x` to a changeable reference to an integer with initial value 1.

Imperative Constructs in SML

The operators on this reference type are

- `ref` (of type `'a -> 'a ref`) for generating a new reference,
- `:=` (of type `'a ref * 'a -> unit`) for changing a value, and
- `!` (of type `'a ref -> 'a`) for getting the value of a reference.

For example the value of `x` can be modified as follows:

```
- x := 2;
> () : unit
- x;
> ref 2 : int ref
```

Note that the value of `x` has *not* change. The value of the `ref` value has changed!

Imperative Constructs in SML (cont'd)

Since `:=` does not change the value of a variable, the contents can be shared with other variables of the same `ref` type:

```
- val y = x;
> val y = ref 2 : int ref
- y := 3;
> () : uniy
- y;
> ref 3 : int ref
- x;
> ref 3 : int
```

`x` and `y` both refer to the same value.

`ref` values in SML are similar to pointer types in C or Pascal. They are very powerful but can be tricky to handle.

Summary

The characteristic property of declarative languages is **referential transparency**, allowing to replace equals-for-equals.

This makes *formal reasoning* about programs much simpler.

For programming in-the-large declarative languages offer

- better modularity, via lazy evaluation
- better computation abstraction, via higher-order functions.
- better data abstraction, via algebraic data types