~ Topic III ~

LISP : functions, recursion, and lists

References:

- **Chapter 3** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.

- **Chapters 5(§4.5) and 13(§1)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

- J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, **3**(4):184–195, 1960.[a]

- J. McCarthy. History of LISP. In *History of Programming Languages*. Academic Press, 1981.

[a] Available on-line from `<http://www-formal.stanford.edu/jmc/recursive.html>`.

LISP = LISt Processing
(±1960)

- Developed in the late 1950s and early 1960s by a team led by John McCarthy in MIT.
- McCarthy described LISP as a “a scheme for representing the *partial recursive functions* of a certain class of symbolic expressions”.
- Motivating problems: Symbolic computation (*symbolic differentiation*), logic (*Advice taker*), experimental programming.
- Software embedding LISP: Emacs (text editor), GTK (linux graphical toolkit), Sawfish (window manager), GnuCash (accounting software).

Programming-language phrases

- *Expressions*. A syntactic entity that may be evaluated to determine its value.
- *Statement*. A command that alters the state of the machine in some explicit way.
- *Declaration*. A syntactic entity that introduces a new identifier, often specifying one or more attributes.

Innovation in the design of LISP

- LISP is an expression-based language.

  Conditional expressions that produce a value were new in LISP.

- *Pure* LISP has no statements and no expressions with side effects. However, LISP also supports *impure* constructs.

Some contributions of LISP

- Lists.
- Recursive functions.
- Garbage collection.
- Programs as data.

Overview

- LISP syntax is extremely simple. To make parsing easy, all operations are written in prefix form (*i.e.*, with the operator in front of all the operands).
- LISP programs compute with *atoms* and *cells*.
- The basic data structures of LISP are *dotted pairs*, which are pairs written with a dot between the components. Putting atoms or pairs together, one can write symbolic expressions in a form traditionally called *S-expressions*. Most LISP programs use *lists* built out of S-expressions.
- LISP is an *untyped* programming language.

- Most operations in LISP take list arguments and return list values.

Example:

`( cons '(a b c) '(d e f) )` $\boxed{\text{cons-cell representation}}$

Remark: The function `(quote x)`, or simply `'x`, just returns the literal value of its argument.

$\boxed{?}$ How does one recognise a LISP program?

```
( defvar x 1 )
( defun g(z) (+ x z) )
( defun f(y)
    ( + ( g y )
        ( let
            ( ( x y) )
            (
              g x )
          ) ) )
( f (+ x 1) )
```

```
val x = 1 ;
fun g(z) = x + z ;
fun f(y)
  = g(y) +
    let
      val x = y
    in
      g(x)
    end ;
f(x+1) ;
```

$\boxed{!}$ It is full of parentheses!

Historically, LISP was a *dynamically scoped* language . . .

```
( defvar x T )
( defun test(y) (eq x y) )
( cond
   ( x ( let ( (z 0) ) (test z) ) )
  )
```

vs.

```
( defvar x T )
( defun test(y) (eq x y) )
( cond
   ( x ( let ( (x 0) ) (test x) ) )
  )
```

. . . when *Scheme* was introduced in 1978, it was a *statically scoped* variant of LISP.

Static and dynamic scope

Static scope rules relate references with declarations of names in the program text; *dynamic scope* rules relate references with associations for names during program execution.

There are two main rules for finding the declaration of a global identifier:

- *Static scope*. A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text.
- *Dynamic scope*. A global identifier refers to the identifier associated with the most recent environment.

Renaming of local variables

Lexical scope is deeply related to renaming of variables. It should not matter whether a program uses one name or another one for a local variable. Let us state this supposition as a principle:

> Consistent renaming of local names in the source text has no effect on the computation set up by a program.

This *renaming principle* motivates static scope because a language that obeys the renaming principle uses lexical scope. The reason is that the renaming principle can be applied to rename local variables until each name has only one declaration in the entire program. This one declaration is the one obtained under lexical scope.

The importance of static scope

Static scope rules play an important part in the design and implementation of most programming languages.

- Static scope rules allow many different sorts of connections to be established between references to names and their declarations during translation.

  For instance, relating a variable name to a declaration for the variable and relating a constant name to a declaration for the constant.

  Other connections include relating names to type declarations, relating formal parameters to formal parameter specifications, relating subprogram calls to subprogram declarations, and relating statement labels referenced in `goto` statements to labels on particular statements.

In each of these cases, a different set of simplications may be made during translation that make execution of the program more efficient.

- Static scope rules are also important for the programmer in reading a program because they make it possible to relate a name referenced in a program to a declaration for the name without tracing the course of program execution.

Abstract machines

The terminology *abstract machine* is generally used to refer to an idealised computing device that can execute a specific programming language directly.

Typically an abstract machine may not be fully implementable. However, an abstract machine should be sufficiently realistic to provide useful information about the real execution of programs.

An important goal in discussing abstract machines is to identify the mental model of the computer that a programmer uses to write and debug programs.

LISP abstract machine

The abstract machine for Pure LISP has four parts:

1. A *LISP* expression to be evaluated.
2. A *continuation*, which is a function representing the remaining of the program to evaluate when done with the current expression.
3. An *association list*, also know as the *A-list*.

   The purpose of the A-list is to store the values of variables that may occur either in the current expression to be evaluated or in the remaining expressions in the program.
4. A *heap*, which is a set of *cons cells* (or *dotted pairs*) that might be pointed to by pointers in the A-list.

Recursion

McCarthy (1960)

```
( defun subst ( x y z )
  ( cond
    ( ( atom z ) ( cond ( ( eq z y ) x ) ( T z ) ) )
    ( T ( cons (subst x y (car z) ) (subst x y (cdr z)) ) )
    )
  )
```

In general . . . , the routine for a recursive function uses itself as a subroutine. For example, the program for `subst x y z` uses itself as a subroutine to evaluate the result of substituting into the subexpression `car z` and `cdr z`. While `subst x y (cdr z)` is being evaluated, the result of the previous evaluation of `subst x y (car z)` must be saved in a temporary storage register. However, `subst` may need the same register for evaluating `subst x y (cdr z)`. This possible conflict is resolved by the SAVE and UNSAVE routines that use the *public push-down list*[a]. The SAVE routine has an index that tells it how many registers in the push-down list are already in use. It moves the contents of the registers which are to be saved to the first unused registers in the push-down list, advances the index of the list, and returns to the program form which control came. This program may then freely use these registers for temporary storage. Before the routine exits it uses UNSAVE, which restores the contents of the temporary registers from the push-down list and moves back the index of this list. The result of these conventions is described, in programming terminology, by saying that the recursive subroutine is transparent to the temporary storage registers.

[a] 1995: now called a stack

Garbage collection
McCarthy (1960)

...When a free register is wanted, and there is none left on the free-storage list, a reclamation† cycle starts.

---

† We already called this process “garbage collection”, but I guess that I chickened out of using it in the paper—or else the Research Laboratory of Electronics grammar ladies wouldn’t let me.

Garbage collection

In computing, *garbage* refers to memory locations that are not accessible to a program.

> At a given point in the execution of a program P, a memory location ℓ is *garbage* if no completed execution of P from this point can access location ℓ. In other words, replacing the contents of ℓ or making this location inaccessible to P cannot affect any further execution of the program.

Garbage collection is the process of detecting garbage during the execution of a program and making it available.

Programs as data

- LISP data and LISP program have the same syntax and internal representation. This allows data structures to be executed as programs and programs to be modified as data.

- One feature that sets LISP apart from many other languages is that it is possible for a program to build a data structure that represents an expression and then evaluates the expression as if it were written as part of the program. This is done with the function `eval`.

Reflection

A language mechanism is said to be *reflective* if it allows to reason about its own structure.

A language is said to be *fully reflective* if it allows a program to reason about all aspects of its current structure and state.

Fully reflective languages are still just research prototypes (*e.g.* Maude), but limited forms of reflection appear in several languages (*e.g.* LISP, Prolog, Python).

Parameter passing in LISP

The *actual parameters* in a function call are always expressions, represented as lists structures.

LISP provides two main methods of *parameter passing*:

- *Pass/Call-by-value.* The most common method is to evaluate the expressions in the actual-parameter list, and pass the resulting values.
- *Pass/Call-by-name.** A less common method is to transmit the expression in the actual parameter list *unevaluated*, and let the call function evaluate them as needed using `eval`.

  The programmer may specify transmission by name using `nlambda` in place of `lambda` in the function definition.

Strict and lazy evaluation

Example: Consider the following function definitions with parameter-passing by value.

```
( defun CountFrom(n) ( CountFrom(+ n 1) ) )

( defun FunnyOr(x y)
    ( cond ( x 1) ( T y ) )
  )

( defun FunnyOrelse(x y)
    ( cond ( (eval x) 1) ( T (eval y) ) )
  )
```

? What happens in the following calls?

```
( FunnyOr T (CountFrom 0) )
( FunnyOr nil T )

( FunnyOrelse 'T '(CountFrom 0) )
( FunnyOrelse 'nil 'T )
```