Notes Lecture 7

1. **Basic objects:**
   * **Atoms**
     + Numbers
     + Symbols (=variables) = string
   * **Lists** = sequence of atoms and / or lists

the way *S-expressions* are:

* + *created* = language syntax
  + *evaluated* = language semantics

1. **Dynamic data structures**

**Symbolic processing** has two main features

1. each *variable* - a *symbolic expression* and not a numerical value
2. main operations 🡪 creation of new expressions from existing ones

**Lists**

A close-up of a white background

Description automatically generated

**Binary trees** = if C1 contains a pointer

**Dotted pairs** = if C2 is anything other than a pointer

Any list has an equivalent in dotted pair notation,

but not every dotted pair has an equivalent in list notation.

**The recursive definition of equivalence** between lists and dotted pairs:

* + If A atom, then the list is equivalent to the dotted pair
  + If the list is equivalent to the dotted pair

then the list is equivalent to the dotted pair

**Examples**: [L07c/pg4](https://ubbcluj-my.sharepoint.com/:b:/r/personal/horia_pop_ubbcluj_ro/Documents/Teaching%20Resources/Functional%20and%20Logic%20Programming/lectures/w07-Lisp/Lecture-07c-Lisp-Introduction.pdf?csf=1&web=1&e=8Qennf)

1. **Syntactic rules**
   * **S-expression**
     + **atom**
       - Numerical atom: string of digits
         * whether or not followed by the dot character
         * preceded or not by a + or -
       - String atom = quoted string (“example”)
       - Symbol= String of characters
     + **list** **=** construction of the form:
       - () or
       - (e) or
       - (e1 e2 ... en)
     + **dotted** **pair**
       - (e1 . e2), where e1 and e2 are S-expressions
   * **Form** = evaluable S-expression
   * **Lisp program** = sequence of forms
2. **Evaluation rules**
   * The following rules for evaluating S-expressions apply:
     + a numerical atom is evaluated by that number;
     + a string is evaluated by its text itself (including quotation marks)
     + a list is *evaluable* (= form) only if its first element is the name of a function, in which case all the arguments are evaluated first, after which the function is applied to these values and the result is determined.
   * The QUOTE function (or ‘ ) returns the S-expression argument itself

= stops the attempt to evaluate the argument.

The character ' can be used instead of QUOTE.

1. **Classification of Lisp functions**

|  |  |  |
| --- | --- | --- |
|  | Fixed no of params | Argument evaluation |
| subr | **Yes** | **Before function eval** |
| nsubr | **Yes** | **Part of function eval** |
| lsubr | **No** | **Before function eval** |
| fsubr | **No** | **Part of function eval** |

1. **Primitive Lisp functions**
   * **CONS**
     + CONS subr 2 (e1 e2): l or pp
     + Creates a dotted pair
   * **CAR**
     + CAR subr 1 (l or pp): e
   * **CDR** 
     + CDR subr 1 (l or pp): e
2. **Assignment - SET, SETQ, SETF**
   * SET lsubr 2, ... (s1 e1 ... sn en): e
3. Evaluates both and
4. Assigns the evaluated value of to the evaluated value of
   * SETQ fsubr 2, ... (s1 f1 ... sn fn): e
     + = “set quoted”
5. Evaluates
6. Assigns the evaluated value of to
   * SETF macro 2, ... (p1 e1 ... pn en): e
     + = “set field”
     + The value at location described by is set to the evaluated value of
   * SET & SETQ side effects are used to give values to the symbols in Lisp
   * Side effect = action by which a function, in addition to calculating its value, makes changes to the data structures in memory
7. **Other list constructors**
   * **LIST**
     + LIST lsubr 0,1, ... (... e ...): l
     + E.g.: (LIST ‘(a b) ‘(c d)) = ((a b) (c d))
   * **APPEND**
     + APPEND lsubr 0,1, ... (... l ...): l
     + E.g.:
       - (APPEND ‘(A B) ‘(C D)) = (A B C D)
       - (APPEND ‘(A B) ‘C) = (A B . C)
       - (APPEND ‘A ‘B ‘(C D) ‘E ‘F) = (C D . F)