

- If $L \Rightarrow$ a proper list of length n
then $(\text{CONS } x \ L) \Rightarrow$ a proper list of length $n+1$
obtained by inserting the value of
 x at the beginning of the list of
length n .

Examples: $(\text{CONS } 'A \ '(B \ C \ D)) \Rightarrow (A \ B \ C \ D)$

- $(\text{CAR } (\text{CONS } x \ L)) \Rightarrow$ the value of x
- $(\text{CDR } (\text{CONS } x \ L)) \Rightarrow$ the value of L
- A call of CONS must have exactly two arguments:
Otherwise there will be an evaluation error.
- If the 2nd argument value passed to CONS is not a proper list (i.e., if it is an atom other than NIL or a dotted list), then CONS returns a dotted list. But in this course you are not expected to call CONS this way!

The name "CONS" is a shortened form of "construct":

- CONS is the most basic way to construct a new proper list;
but the CDR of that new list will be an existing list.

- If $l_1 \Rightarrow$ a proper list of length n_1
and $l_2 \Rightarrow$ a proper list of length n_2
then $(\text{APPEND } l_1 \ l_2) \Rightarrow$ a proper list of length $n_1 + n_2$
obtained by **concatenating the lists given by l_1 and l_2 .**
- If $l_1, \dots, l_k \Rightarrow k$ proper lists of lengths n_1, \dots, n_k , then
 $(\text{APPEND } l_1 \ \dots \ l_k) \Rightarrow$ a proper list of length $n_1 + \dots + n_k$
obtained by **concatenating those k lists.**

When APPEND is called with $k \geq 2$ arguments:

- If any of the first $k-1$ argument values is **not** a proper list, then there will be an evaluation error.
EXAMPLE: Evaluation of $(\text{APPEND } (+ \ 3 \ 4) \ '(C \ D \ E))$ produces an error, because 7 is not a list.
- If the first $k-1$ argument values are proper lists but the k^{th} argument value is not, then APPEND returns a *dotted* list (unless the first $k-1$ argument values are all NIL). **But in this course you are not expected to call APPEND this way!**

$(\text{LIST } e_1 \dots e_k) \Rightarrow$ a proper list of length k whose i^{th} element ($1 \leq i \leq k$) is the value of e_i .

EXAMPLES: $(\text{LIST 'P '(A B) (+ 3 4) '(F)}) \Rightarrow (\text{P (A B) 7 (F)})$
 $(\text{LIST (CAR '(A B)) 6 (CDR '(X Y))}) \Rightarrow (\text{A 6 (Y)})$
 $(\text{LIST '(U V W)}) \Rightarrow ((\text{U V W}))$

Don't confuse the functions CONS, APPEND, and LIST.

$(\text{CONS '(1 2 3) '(4 5)}) \Rightarrow ((\text{1 2 3}) 4 5)$

$(\text{APPEND '(1 2 3) '(4 5)}) \Rightarrow (1 2 3 4 5)$

$(\text{LIST '(1 2 3) '(4 5)}) \Rightarrow ((\text{1 2 3}) (4 5))$

- $(\text{LIST } e) = (\text{CONS } e \text{ NIL})$
- $(\text{CONS } e \text{ } L) = (\text{APPEND } (\text{LIST } e) \text{ } L)$
- Evaluation of $(\text{CONS } e \text{ } L)$, $(\text{APPEND } L_1 \dots L_k)$, and $(\text{LIST } e_1 \dots e_k)$ does not change the values of the e 's and the L 's.
 - If this were not the case, we would not be able to use these functions in functional programming!

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `float f (int n, float x) { return n+x; }`

Lisp: `(defun f (n x) (+ n x))`

Notes

1. Java is statically typed but Lisp is dynamically typed.
 - In a statically typed language (e.g., Java, C++, or C#) each variable / formal parameter has a type that can be determined by a compiler, and *is only allowed to store values of that type or a subtype*. Compilers can detect and reject code that performs operations on values of incorrect types--e.g., code that adds an int to a list.
 - In a dynamically typed language (e.g., Lisp, Python, or Javascript), a variable / formal parameter does not have a fixed type and *may well store values of different types at different times*. Type-checking occurs during code execution--an error is reported when an operation is performed on values of incorrect types.

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `float f (int n, float x) { return n+x; }`

Lisp: `(defun f (n x) (+ n x))`

Notes

1. Java is statically typed but Lisp is dynamically typed.
 - In a dynamically typed language (e.g., Lisp, Python, or Javascript), a variable / formal parameter does not have a fixed type and *may well store values of different types at different times*. Type-checking occurs during code execution--an error is reported when an operation is performed on values of incorrect types.

As Lisp is dynamically typed, the above Lisp definition does not declare the types of the formal parameters `n` and `x`, and also does not declare the return type of `f`: The two argument values we pass into a call of `f` can be numbers of any type; the type of the result returned by `f` will depend on the types of the argument values.

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `float f (int n, float x) { return n+x; }`

Lisp: `(defun f (n x) (+ n x))`

Notes

1. Java is statically typed but Lisp is dynamically typed.

As Lisp is dynamically typed, the above Lisp definition does not declare the types of the formal parameters `n` and `x`, and also does not declare the return type of `f`: The two argument values we pass into a call of `f` can be numbers of any type; the type of the result returned by `f` will depend on the types of the argument values.

For example, if one argument value is an integer and the other a floating-point number, the returned result will be a floating-point number. But if both argument values are integers, then the result will be an integer.

If either argument value isn't a number, a type-mismatch error will be reported *when the call* `(+ n x)` is executed.

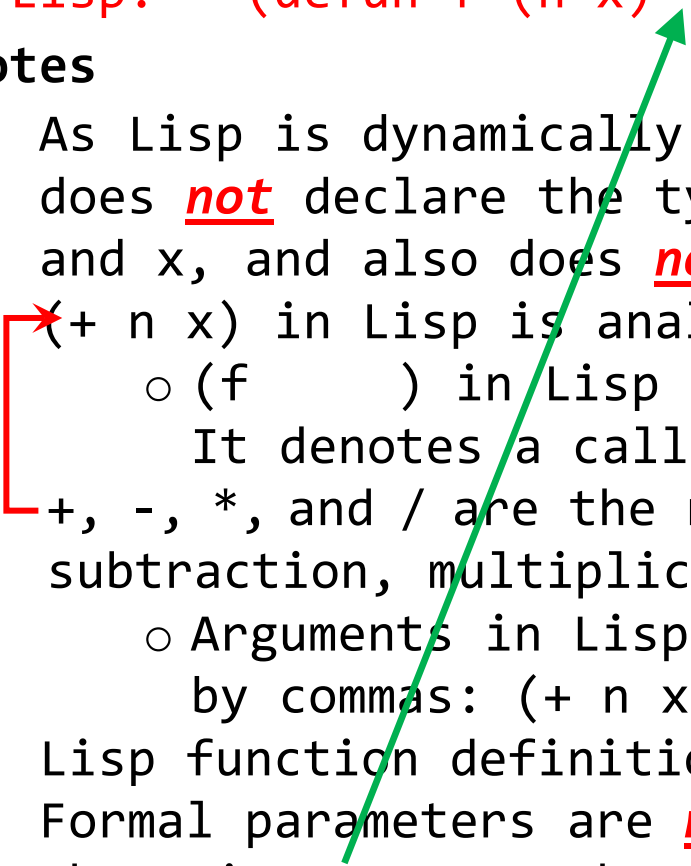
A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `float f (int n, float x) { return n+x; }`

Lisp: `(defun f (n x) (+ n x))`

Notes

1. As Lisp is dynamically typed, the above Lisp definition does not declare the types of the formal parameters `n` and `x`, and also does not declare the return type of `f`.
2.  `(+ n x)` in Lisp is analogous to `n+x` in Java.
 - `(f ...)` in Lisp is analogous to `f(...)` in Java:
It denotes a call of the function `f`.
 - `+`, `-`, `*`, and `/` are the names of Lisp's built-in addition, subtraction, multiplication, and division functions.
 - Arguments in Lisp function calls are not separated by commas: `(+ n x)` is correct; `(+ n, x)` is wrong.
3. Lisp function definitions begin with the word `DEFUN`.
4. Formal parameters are not separated by commas.
5. There is no `RETURN` keyword before the expression whose value is to be returned.

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `float f (int n, float x) { return n+x; }`

Lisp: `(defun f (n x) (+ n x))`

- `(f ...)` in Lisp is analogous to `f(...)` in Java:
It denotes a call of the function `f`.
- Arguments in Lisp function calls are **not** separated by commas: `(+ n x)` is correct; `(+ n, x)` is wrong.

Examples of Calls of the Above Lisp Function `f`:

- The call `(f 3 4.2)` is analogous to the Java call `f(3, 4.2F)` and returns `7.2`.
- The call `(f (- 8 2) (f 3 4.2))` is analogous to the Java call `f(8-2, f(3, 4.2F))` and returns `13.2`.
- The call `(f 3.0 4.0)` returns `7.0`. (Note that the Java call `f(3.0F, 4.0F)` would be **illegal** as the 1st argument of the Java function `f` is declared to have type `int`!)
- The call `(f 3 4)` returns the **integer** `7`, as the values of the parameters `n` and `x` are integers.

Syntax of Common Lisp Function Definitions and Calls

For any integer $k \geq 0$, a new Common Lisp function that takes k arguments can be defined as follows:

```
(defun <func name> (<param>1 ... <param>k)  
  <body-expr>)
```

- <func name> is a symbol (e.g., G or FACTORIAL) that will be the name of the new function.
- <param>₁, ..., <param>_k are k different symbols that will be the k formal parameters of the new function.
- <body-expr> is an S-expression that can be evaluated.

A call of the function can be written as follows:

```
(<func name> <arg>1 ... <arg>k)
```

Here <arg>₁, ..., <arg>_k are the argument S-expressions:

- This call is evaluated by **evaluating** <arg>₁, ..., <arg>_k, **then evaluating** <body-expr> in an environment in which the value of <param>_i ($1 \leq i \leq k$) is the value of <arg>_i, **and then returning** the value of <body-expr>.

Another Example of a Common Lisp Function Definition

The following Java function was considered earlier:

```
// factorial(n) returns 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{ return (n == 1) ? 1 : factorial(n-1) * n; }
```

We now write a **Common Lisp** analog of this function.

To do this, we use the following facts:

- In Lisp, the **=** function can be used to test whether two numbers are equal.
- The Lisp analog of $c ? e_1 : e_2$ is **(if c e₁ e₂)**. **Examples:**
(if (= 2 3) 4 5) => 5 **(if (= 2 (+ 1 1)) 4 5) => 4**
Notation: **expr₁ => expr₂** means the *value* of **expr₁** is **expr₂**.

We also note that:

- The Lisp analog of the Java expression **factorial(n-1) * n** is:
(* (factorial (- n 1)) n)

Here is a Common Lisp version of the above function:

```
(defun factorial (n)
  (if (= n 1) 1 (* (factorial (- n 1)) n) ))
```

Another Example of a Common Lisp Function Definition

The following Java function was considered earlier:


```
// factorial(n) returns 1 * 2 * ... * (n-1) * n if 1 ≤ n ≤ 20
static long factorial (int n)
{ return (n == 1) ? 1 : factorial(n-1) * n; }
```

Here is a Common Lisp version of the above function:

```
(defun factorial (n)
  (if (= n 1) 1 (* (factorial (- n 1)) n) ))
```

As the (if ...) expression in this definition is quite long, it may be better to split it into 3 lines:

```
(defun factorial (n)
  (if (= n 1)
      1
      (* (factorial (- n 1)) n)))
```



Note: Do not put these last two closing parentheses on separate lines! That would *waste screen space* and also serve no good purpose because Lisp programmers read and write code in editors that match parentheses for them!

SECOND, THIRD, etc.

Recall: If $l \Rightarrow$ a nonempty list, then
 $(\text{FIRST } l) = (\text{CAR } l)$ evaluates to the
1st element of the list given by l .

Also, $(\text{FIRST NIL}) = (\text{CAR NIL}) \Rightarrow \text{NIL}$.

Similarly:

- If $l \Rightarrow$ a list of length ≥ 2 , then
(SECOND l) $= (\text{CAR } (\text{CDR } l))$ evaluates to
the 2nd element of the list given by l .
 $(\text{SECOND } l) \Rightarrow \text{NIL}$ if $l \Rightarrow$ a proper list of length ≤ 1 .
- If $l \Rightarrow$ a list of length ≥ 3 , then
(THIRD l) $= (\text{CAR } (\text{CDR } (\text{CDR } l)))$ evaluates to
the 3rd element of the list given by l .
 $(\text{THIRD } l) \Rightarrow \text{NIL}$ if $l \Rightarrow$ a proper list of length ≤ 2 .
- **(FOURTH l)**, ..., **(TENTH l)** are defined analogously.