

- If $l_1 \Rightarrow$ a proper list of length n_1
and $l_2 \Rightarrow$ a proper list of length n_2
then **(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 .**

EXAMPLES: (APPEND '(A B C) '(D E)) =>
(APPEND '((A B) C) '(D (E))) =>
(APPEND NIL '(A B C)) =>

- If $l_1 \Rightarrow$ a proper list of length n_1
and $l_2 \Rightarrow$ a proper list of length n_2
then **(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 .**

EXAMPLES: (APPEND '(A B C) '(D E)) \Rightarrow (A B C D E)
(APPEND '((A B) C) '(D (E))) \Rightarrow ((A B) C D (E))
(APPEND NIL '(A B C)) \Rightarrow (A B C)

- If $l_1 \Rightarrow$ a proper list of length n_1
 and $l_2 \Rightarrow$ a proper list of length n_2
 then **(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 .**

EXAMPLES: (APPEND '(A B C) '(D E)) \Rightarrow (A B C D E)
 (APPEND '((A B) C) '(D (E))) \Rightarrow ((A B) C D (E))
 (APPEND NIL '(A B C)) \Rightarrow (A B C)

More generally:

- If $l_1, \dots, l_k \Rightarrow k$ proper lists of lengths n_1, \dots, n_k , then
**(APPEND $l_1 \dots l_k$) \Rightarrow a proper list of length $n_1 + \dots + n_k$
 obtained by concatenating those
 k lists.**

EXAMPLE: (APPEND '(A B) '(C D E) '(F)) \Rightarrow

- If $l_1 \Rightarrow$ a proper list of length n_1
 and $l_2 \Rightarrow$ a proper list of length n_2
 then **(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 .**

EXAMPLES: (APPEND '(A B C) '(D E)) \Rightarrow (A B C D E)
 (APPEND '((A B) C) '(D (E))) \Rightarrow ((A B) C D (E))
 (APPEND NIL '(A B C)) \Rightarrow (A B C)

More generally:

- If $l_1, \dots, l_k \Rightarrow k$ proper lists of lengths n_1, \dots, n_k , then
(APPEND $l_1 \dots l_k$) \Rightarrow a proper list of length $n_1 + \dots + n_k$
 obtained by **concatenating those k lists.**

EXAMPLE: (APPEND '(A B) '(C D E) '(F)) \Rightarrow (A B C D E F)

- 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!**

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

(CONS '(1 2 3) '(4 5)) ⇒

(APPEND '(1 2 3) '(4 5)) ⇒

(LIST '(1 2 3) '(4 5)) ⇒

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

`(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)`

`(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)`

`(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))`

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

`(CONS '(1 2 3) '(4 5)) ⇒ ((1 2 3) 4 5)`

`(APPEND '(1 2 3) '(4 5)) ⇒ (1 2 3 4 5)`

`(LIST '(1 2 3) '(4 5)) ⇒ ((1 2 3) (4 5))`

- `(LIST e) = (CONS)`
- `(CONS e l) = (APPEND)`
-

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

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

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

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

- $(\text{LIST } e) = (\text{CONS } e\ \text{NIL})$
- $(\text{CONS } e\ l) = (\text{APPEND } (\text{LIST } e)\ l)$
- Evaluation of $(\text{CONS } e\ l)$, $(\text{APPEND } l_1 \dots l_k)$, 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!

**More Built-in Functions
That Extract Parts of Lists:
SECOND, ..., TENTH,
and C ... R Functions**

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.

C ... R Functions

Each C ... R function is equivalent to the **composition of a certain sequence of CARs and/or CDRs**:

- $(\text{CADR } L) = (\text{CAR } (\text{CDR } L)) = (\text{SECOND } L)$
- $(\text{CADDR } L) = (\text{CAR } (\text{CDR } (\text{CDR } L))) = (\text{THIRD } L)$
- $(\text{CADDRR } L) = (\text{CAR } (\text{CDR } (\text{CDR } (\text{CDR } L)))) = (\text{FOURTH } L)$
- $(\text{CAAR } L) = (\text{CAR } (\text{CAR } L)) = (\text{FIRST } (\text{FIRST } L))$
- $(\text{CDAR } L) = (\text{CDR } (\text{CAR } L)) = (\text{REST } (\text{FIRST } L))$
- $(\text{CDADDR } L) = (\text{CDR } (\text{CAR } (\text{CDR } (\text{CDR } L))))$
 $= (\text{CDR } (\text{CADDR } L)) = (\text{REST } (\text{THIRD } L))$
- $(\text{CADADR } L) = (\text{CAR } (\text{CDR } (\text{CAR } (\text{CDR } L))))$
 $= (\text{CADR } (\text{CADR } L)) = (\text{SECOND } (\text{SECOND } L))$

and similarly for all other sequences of **2 - 4 As** and/or **Ds**.

- Authors sometimes write C ... R function names containing **> 4 As** and/or **Ds**
--see [Touretzky's solution to 2.15 on p. C-11](#)--but functions with such names are not built-in functions of most Common Lisp implementations!
- C ... R function names are pronounceable: This is one reason the nondescriptive names CAR and CDR are still used.

From p. 48
of Touretzky:

CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	<i>kar</i>	FIRST
CDR	<i>cou-der</i>	REST
CAAR	<i>ka-ar</i>	
CADR	<i>kae-der</i>	SECOND
CDAR	<i>cou-dar</i>	
CDDR	<i>cou-dih-der</i>	
CAAAR	<i>ka-a-ar</i>	
CAADR	<i>ka-ae-der</i>	
CADAR	<i>ka-dar</i>	
CADDR	<i>ka-dih-der</i>	THIRD
CDAAR	<i>cou-da-ar</i>	
CDADR	<i>cou-dae-der</i>	
CDDAR	<i>cou-dih-dar</i>	
CDDDR	<i>cou-did-dih-der</i>	
CADDDR	<i>ka-dih-dih-der</i>	FOURTH

and so on

Defining Your Own Functions in Common Lisp

- The slides in this section show, with examples, how to define functions in Common Lisp.
- For more examples, see sec. 3.5 of Touretzky (pp. 82–3):
<https://www.cs.cmu.edu/~dst/LispBook/book.pdf#page=94>
- Touretzky calls Lisp notation “*eval* notation”, as Lisp uses its *eval* function to evaluate expressions.

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java: `static 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 Go) each variable / formal parameter has a type that can be determined by a compiler, and *should never store a value / reference that isn't of that type or a subtype*. Similarly, if a function in a statically typed language returns a value, then that function has a fixed return type that can be determined by a compiler, and *should always return a value / reference of that type or a subtype*.
 - A good compiler for a statically typed language detects and rejects almost all code that performs operations on, returns, or stores values of inappropriate types--e.g., code that adds an int to a list--*so type checking rarely has to be done at run-time*.

Here are a Java function and an analogous Lisp function:

Java: `static 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 Go) each variable / formal parameter has a type that can be determined by a compiler, and *should never store a value / reference that isn't of that type or a subtype*. Similarly, if a function in a statically typed language returns a value, then that function has a fixed return type that can be determined by a compiler, and *should always return a value / reference of that type or a subtype*.
 - In a dynamically typed language (e.g., Lisp, Python, or Javascript), a variable / formal parameter *may store values or references of entirely different types at different times*. A function in such a language *may return values / references of entirely different types when it is called with different arguments*.

Here are a Java function and an analogous Lisp function:

Java: `static 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 *may store values or references of entirely different types at different times*. A function in such a language *may return values / references of entirely different types when it is called with different arguments*.

Here are a Java function and an analogous Lisp function:

Java: `static 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 *may store values or references of entirely different types at different times*. A function in such a language *may return values / references of entirely different types when it is called with different arguments*.

When we use a dynamically typed language, type-checking occurs during code execution--an error is reported when an operation is performed on values of incorrect types.

Here are a Java function and an analogous Lisp function:

Java: `static 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 *may store values or references of entirely different types at different times*. A function in such a language *may return values / references of entirely different types when it is called with different arguments*.

When we use a dynamically typed language, 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, a Lisp function definition does not need to declare the types of its formal parameters and does not need to declare its return type!

Here are a Java function and an analogous Lisp function:

Java: **static 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, a Lisp function definition does **not** need to declare the types of its formal parameters and does **not** need to declare its return type!

Remark Lisp does allow “FYI” type declarations of formal parameters and other variables, but such declarations are unlike type declarations in statically typed languages because:

- A Lisp implementation is **not** required to warn the user when a variable is given a value of the wrong type.
- There’s no good way to reliably detect such errors without performing type checks **during code execution** (whereas in a statically typed language almost all such errors will be detected by a good compiler and won't have to be checked for at run time).

Expert programmers can use Lisp type declarations to tell a compiler it can *assume* certain values are of specified types to generate more efficient code; the code will work if the assumptions are correct.

Here are a Java function and an analogous Lisp function:

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

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

Notes


1. As Lisp is dynamically typed, a Lisp function definition does not need to declare the types of its formal parameters and does not need to declare its return type!
2. `(+ n x)` in Lisp is analogous to `n+x` in Java.

Here are a Java function and an analogous Lisp function:

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

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

Notes

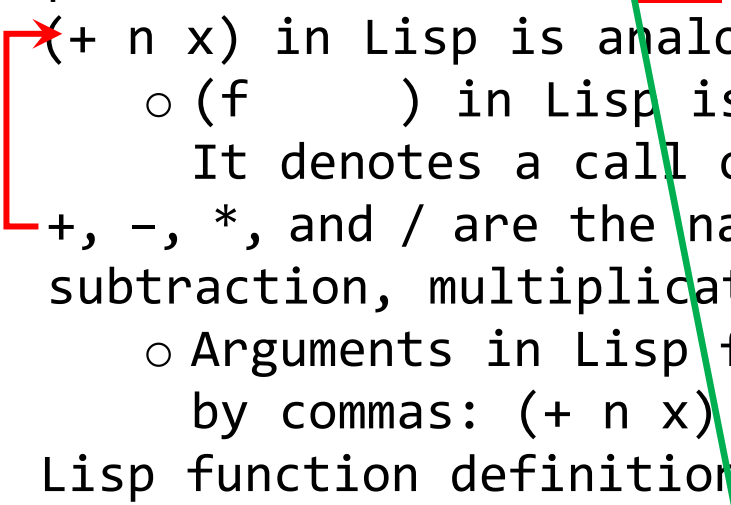
1. As Lisp is dynamically typed, a Lisp function definition does not need to declare the types of its formal parameters and does not need to declare its return type!
2.  `(+ n x)` in Lisp is analogous to `n+x` in Java.
 - `(f ...)` in Lisp is analogous to `f(...)` in Java:
It denotes a call of the function `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`.

Here are a Java function and an analogous Lisp function:

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

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

Notes

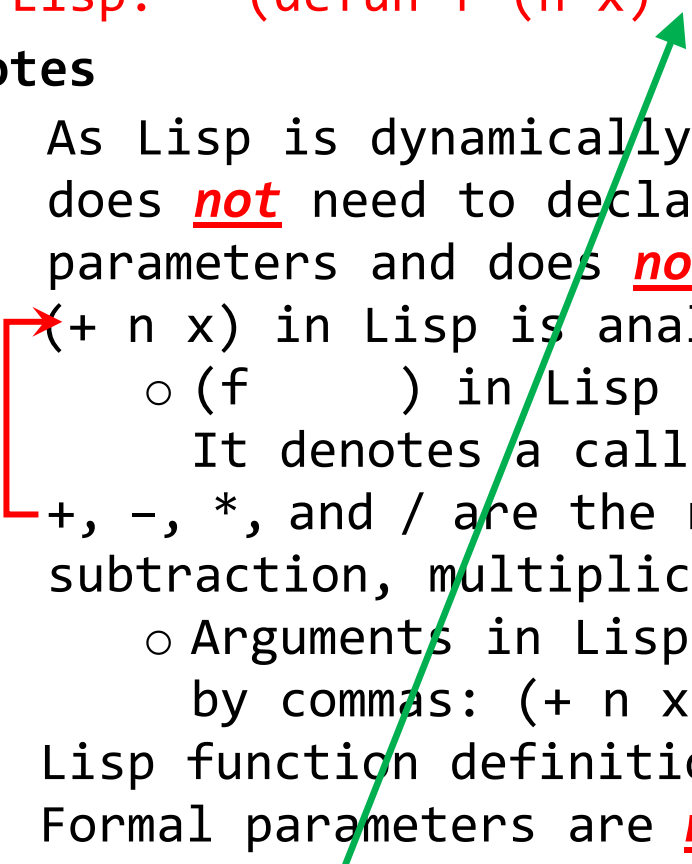
1. As Lisp is dynamically typed, a Lisp function definition does not need to declare the types of its formal parameters and does not need to declare its return type!
2.  $(+ \ n \ x)$ in Lisp is analogous to $n+x$ in Java.
 - $(f \ \ \ \)$ in Lisp is analogous to $f(\ \ \ \)$ in Java:
It denotes a call 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.

Here are a Java function and an analogous Lisp function:

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

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

Notes


1. As Lisp is dynamically typed, a Lisp function definition does not need to declare the types of its formal parameters and does not need to declare its return type!
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.

Here are a Java function and an analogous Lisp function:

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

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

Notes

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**.

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. (4.2 in Clisp is analogous to 4.2F in Java.)
- The call **(f (- 8 2) (f 3 4.2))** returns 13.2; this is analogous to the Java call **f(8-2, f(3, 4.2F))**.

Here are a Java function and an analogous Lisp function:

Java: **static 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. (4.2 in Clisp is analogous to 4.2F in Java.)
- The call (f (- 8 2) (f 3 4.2)) returns 13.2; this is analogous to the Java call f(8-2, f(3, 4.2F)).
- The call (f 3.0 4.0) returns 7.0. (Note that the analogous Java call f(3.0F, 4.0F) is **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. (Note that the Java call f(3, 4) returns a **float** value: 4 is coerced to 4.0F.)