Defining Your Own Functions in Common Lisp

A Simple Common Lisp Function Definition

Here are a Java function and an analogous Lisp function:

Java:

Lisp:

Notes

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:
```

Notes

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

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.
    -
    -

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***.

   -

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.
   -

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***.

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.

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.

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:

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.

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.

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.

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.

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:

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.

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.

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.

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.

   In Java, the only characters that are allowed in the names of variables / parameters, and functions are letters, digits, _, and $. But Lisp also allows many other characters, including + - * / % ^ ! ? > < = [ ] { }.

   +, -, *, and / are the names of Lisp's built-in addition, subtraction, multiplication, and division functions.

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.

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.

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.

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.

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.

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))
```

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.

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

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.

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

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.
-

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

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.
- 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.

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.

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.

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**!)
-

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

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>)

•

•

•

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:

•

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.
-
-

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:

-

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.
-

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:

-

Syntax of Common Lisp Function Definitions and Calls

For any integer *k* ≥ 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:

-

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 ,
  then
  ,
  and then .

Syntax of Common Lisp Function Definitions and Calls

For any integer $k \ge 0$, 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***

  ,

  and then .

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*** .

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;
}
```

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:

•

•

We also note that:

•

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.
-

We also note that:

-

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 e_1 e_2). **Examples:**

We also note that:

•

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 e_1 e_2). **Examples:**
  (if (= 2 3) 4 5) => 5

We also note that:

•

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 e_1 e_2). **Examples:**
  (if (= 2 3) 4 5) => 5   (if (= 2 (+ 1 1)) 4 5) => 4

We also note that:

-

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 e_1 e_2). **Examples:**

    (if (= 2 3) 4 5) => 5   (if (= 2 (+ 1 1)) 4 5) => 4

  Notation: $expr_1$ => $expr_2$ means the *value* of $expr_1$ is $expr_2$.

We also note that:

-

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 e_1 e_2). **Examples:**

  (if (= 2 3) 4 5) => 5   (if (= 2 (+ 1 1)) 4 5) => 4

  Notation: $expr_1$ => $expr_2$ means the *value* of $expr_1$ is $expr_2$.

We also note that:

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

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 e_1 e_2). **Examples:**

  (if (= 2 3) 4 5) => 5   (if (= 2 (+ 1 1)) 4 5) => 4

  Notation: expr_1 => expr_2 means the *value* of expr_1 is expr_2.

We also note that:

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

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 e_1 e_2). **Examples:**

  `(if (= 2 3) 4 5) => 5`   `(if (= 2 (+ 1 1)) 4 5) => 4`

  Notation: $expr_1$ => $expr_2$ means the *value* of $expr_1$ is $expr_2$.

We also note that:

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

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 e_1 e_2). **Examples:**

  ```
  (if (= 2 3) 4 5) => 5      (if (= 2 (+ 1 1)) 4 5) => 4
  ```

  Notation: expr$_1$ => expr$_2$ means the *value* of expr$_1$ is expr$_2$.

We also note that:

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

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 e_1 e_2). **Examples:**

  (if (= 2 3) 4 5) => 5   (if (= 2 (+ 1 1)) 4 5) => 4

  Notation: $expr_1$ => $expr_2$ means the *value* of $expr_1$ is $expr_2$.

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) ))
```

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) ))
```

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)))
```

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!

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!

The QUOTE Special Operator

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.:

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.: The value of **(QUOTE (+ 3 4))** is

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.

Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.
Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).
E.g.:

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.
Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).
E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.

Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).

E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

-
-
-
-

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.
Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).
E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

- Evaluation of **(F '(+ 3 4))** passes the list **(+ 3 4)** as argument to a call of the function F.
-
-
-

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.
Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).
E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

- Evaluation of **(F '(+ 3 4))** passes the list **(+ 3 4)** as argument to a call of the function F.
- Evaluation of **(F (+ 3 4))** passes the number **7** as argument to a call of F.
-
-

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!

E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.

Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).

E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

- Evaluation of **(F '(+ 3 4))** passes the list **(+ 3 4)** as argument to a call of the function F.
- Evaluation of **(F (+ 3 4))** passes the number **7** as argument to a call of F.
- Evaluation of **(F 'X)** passes the symbol **X** as argument to a call of F.
-

QUOTE is a special operator you will use most often!

(QUOTE *e*) evaluates to *e*; *e* is not evaluated!
E.g.: The value of **(QUOTE (+ 3 4))** is the list **(+ 3 4)**.
Note: (QUOTE e_1 … e_n) cannot be evaluated if $n \neq 1$;
evaluation produces an error!

For any S-expression *e*, **'*e* is equivalent to (QUOTE *e*)**
and we usually write 'e instead of (QUOTE e).
E.g.: We'd write **'(+ 3 4)** instead of **(QUOTE (+ 3 4))**.

If F is the name of some Lisp function, then:

- Evaluation of **(F '(+ 3 4))** passes the list **(+ 3 4)** as argument to a call of the function F.
- Evaluation of **(F (+ 3 4))** passes the number **7** as argument to a call of F.
- Evaluation of **(F 'X)** passes the symbol **X** as argument to a call of F.
- Evaluation of **(F X)** passes the *value of the variable denoted by* **X** as argument to a call of F.

●

●

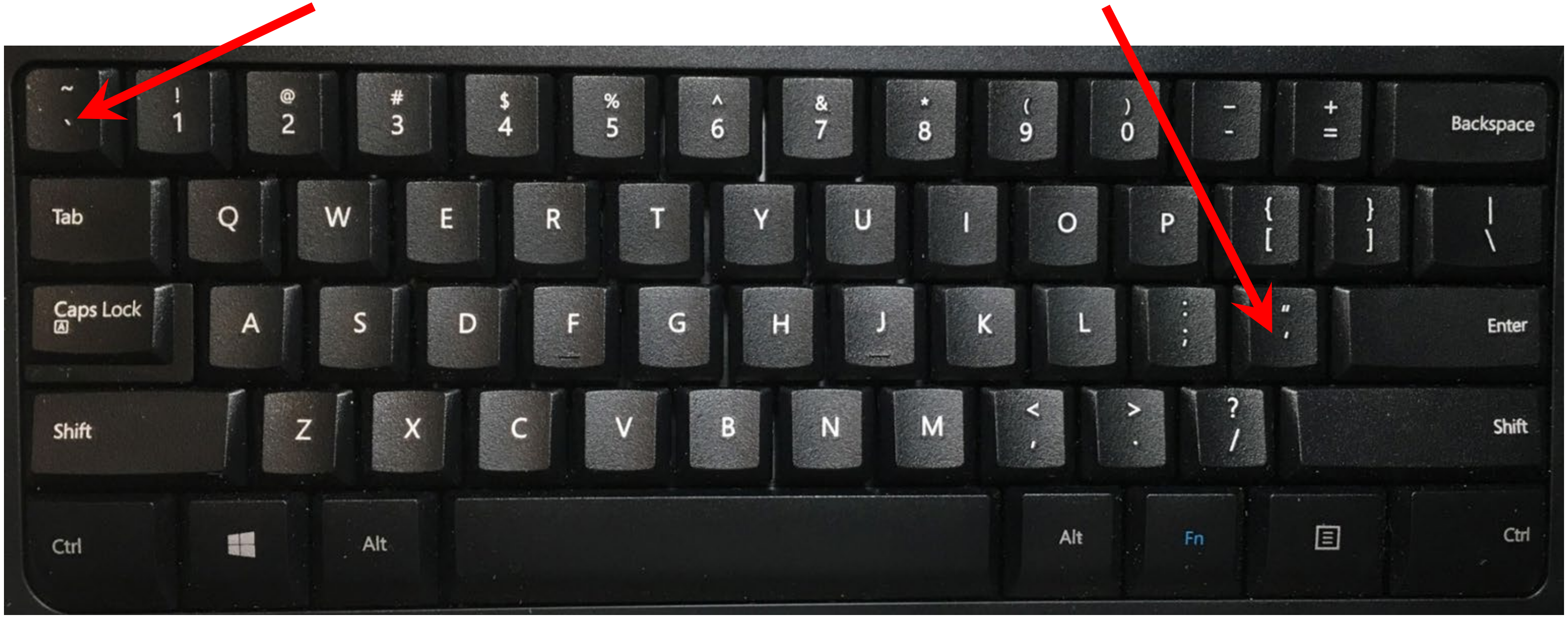
●

●

- Numbers, NIL, and T evaluate to themselves, so you don't need to QUOTE them!
-
-
-

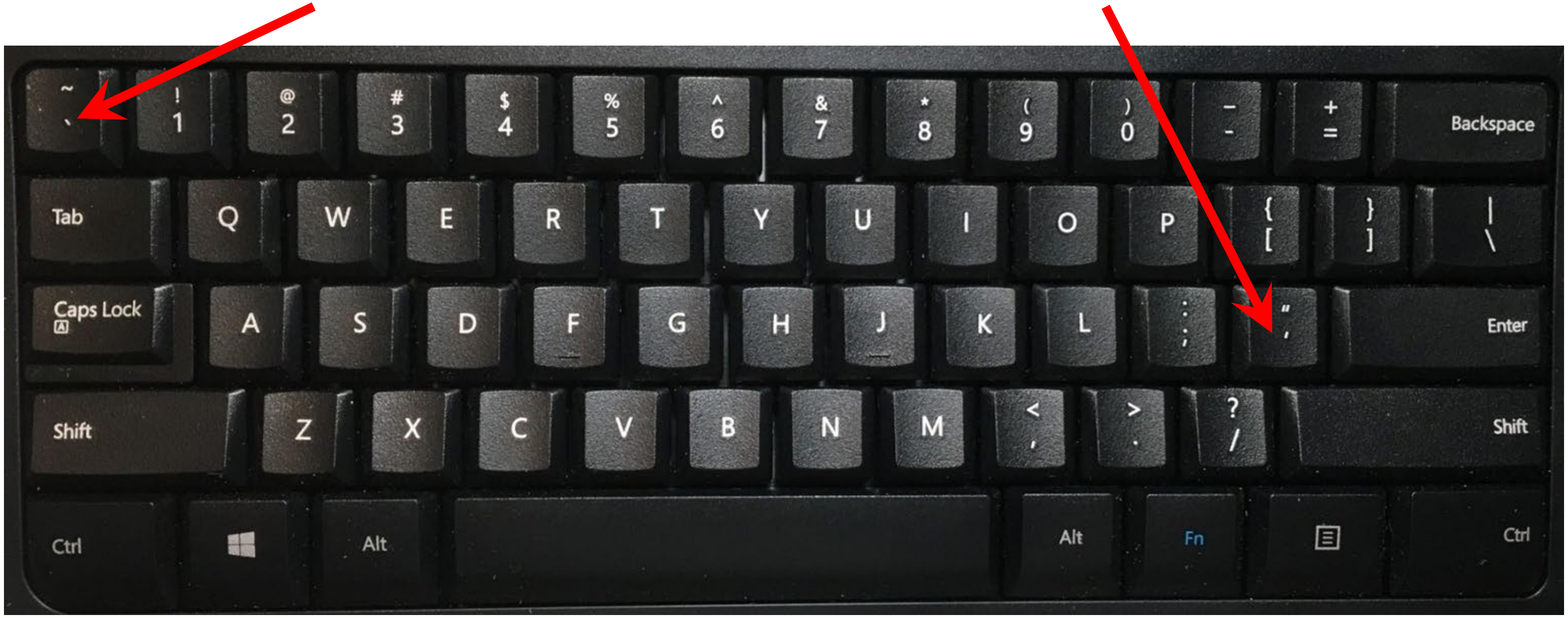
- Numbers, NIL, and T evaluate to themselves, so you don't need to QUOTE them!
- As a matter of style, you should ***not*** type any spaces between ' and the S-expression *e* when you type '*e*.
-
-

- Numbers, NIL, and T evaluate to themselves, so you don't need to QUOTE them!
- As a matter of style, you should ***not*** type any spaces between ' and the S-expression *e* when you type '*e*.
- Don't confuse the ` and ' characters:
  ` is ***this*** character!   ' is ***this*** character!


~ ` ! 1 @ 2 # 3 $ 4 % 5 ^ 6 & 7 * 8 (9) 0 _ - + = Backspace
Tab Q W E R T Y U I O P { [}] | \
Caps Lock A S D F G H J K L : ; " ' Enter
Shift Z X C V B N M < , > . ? / Shift
Ctrl Alt Alt Fn Ctrl

-

- Numbers, NIL, and T evaluate to themselves, so you don't need to QUOTE them!
- As a matter of style, you should ***not*** type any spaces between ' and the S-expression *e* when you type '*e*.
- Don't confuse the ` and ' characters:
  ` is ***this*** character!   ' is ***this*** character!



- ` can do things ' can't. It's useful for writing macros, but you won't need to use ` in this course.

Built-in
Common Lisp Functions
for Taking Lists Apart:
CAR/FIRST and CDR/REST

Notation: For S-expressions e and e', we write $e \Rightarrow e'$ to mean that e evaluates to e'.

Examples:

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: `(+ 4 5) ⇒`   `(SQRT (+ 4 5.0)) ⇒`

`'(SQRT (3 SQRT)) ⇒`

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: `(+ 4 5)` ⇒ **9** `(SQRT (+ 4 5.0))` ⇒ **3.0**
`'(SQRT (3 SQRT))` ⇒

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**

'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:

  (**CAR** *e*) ⇒ the *first* element of that list L.

  Examples:

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:
(CAR '(DOG CAT (AT (3 +)))) ⇒
(CAR '(DOG)) ⇒
(CAR '((AT (3 +)) DOG CAT)) ⇒

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: `(+ 4 5)` ⇒ **9** `(SQRT (+ 4 5.0))` ⇒ **3.0**

`'(SQRT (3 SQRT))` ⇒ **`(SQRT (3 SQRT))`**

- If *e* ⇒ a nonempty list L, then:

  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒
(CAR '((AT (3 +)) DOG CAT)) ⇒
```

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**

'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:

  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

(CAR '(DOG)) ⇒ DOG

(CAR '((AT (3 +)) DOG CAT)) ⇒

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9** (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:
```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

Examples:

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: (+ 4 5) ⇒ **9** (SQRT (+ 4 5.0)) ⇒ **3.0**

'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:

  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:

  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

Examples:

```
(CDR '(DOG CAT (AT (3 +)))) ⇒
(CDR '(DOG)) ⇒
(CDR '((AT (3 +)) DOG CAT)) ⇒
(CDR (CAR '((A B C) D E))) ⇒
(CAR (CDR '((A B C) D E))) ⇒
```

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

Examples:

```
(CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
(CDR '(DOG)) ⇒
(CDR '((AT (3 +)) DOG CAT)) ⇒
(CDR (CAR '((A B C) D E))) ⇒
(CAR (CDR '((A B C) D E))) ⇒
```

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9**   (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:
```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

Examples:
```
(CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
(CDR '(DOG)) ⇒ NIL
(CDR '((AT (3 +)) DOG CAT)) ⇒
(CDR (CAR '((A B C) D E))) ⇒
(CAR (CDR '((A B C) D E))) ⇒
```

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples:

```
(+ 4 5) ⇒ 9   (SQRT (+ 4 5.0)) ⇒ 3.0
'(SQRT (3 SQRT)) ⇒ (SQRT (3 SQRT))
```

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by *omitting* its first element.

Examples:

```
(CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
(CDR '(DOG)) ⇒ NIL
(CDR '((AT (3 +)) DOG CAT)) ⇒ (DOG CAT)
(CDR (CAR '((A B C) D E))) ⇒
(CAR (CDR '((A B C) D E))) ⇒
```

Notation: For S-expressions *e* and *e'*, we write

e ⇒ *e'* to mean that *e* evaluates to *e'*.

Examples: (+ 4 5) ⇒ 9   (SQRT (+ 4 5.0)) ⇒ 3.0

'(SQRT (3 SQRT)) ⇒ (SQRT (3 SQRT))

- If *e* ⇒ a nonempty list L, then:

  (**CAR** *e*) ⇒ the *first* element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:

  (**CDR** *e*) ⇒ the list obtained from L by *omitting* its first element.

Examples:

```
(CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
(CDR '(DOG)) ⇒ NIL
(CDR '((AT (3 +)) DOG CAT)) ⇒ (DOG CAT)
(CDR (CAR '((A B C) D E))) ⇒ (B C)
(CAR (CDR '((A B C) D E))) ⇒
```

Notation: For S-expressions *e* and *e'*, we write
e ⇒ *e'* to mean that *e* evaluates to *e'*.
Examples: (+ 4 5) ⇒ **9** (SQRT (+ 4 5.0)) ⇒ **3.0**
'(SQRT (3 SQRT)) ⇒ **(SQRT (3 SQRT))**

- If *e* ⇒ a nonempty list L, then:
  (**CAR** *e*) ⇒ the ***first*** element of that list L.

Examples:

```
(CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
(CAR '(DOG)) ⇒ DOG
(CAR '((AT (3 +)) DOG CAT)) ⇒ (AT (3 +))
```

- If *e* ⇒ a nonempty list L, then:
  (**CDR** *e*) ⇒ the list obtained from L
  by ***omitting*** its first element.

Examples:

```
(CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
(CDR '(DOG)) ⇒ NIL
(CDR '((AT (3 +)) DOG CAT)) ⇒ (DOG CAT)
(CDR (CAR '((A B C) D E))) ⇒ (B C)
(CAR (CDR '((A B C) D E))) ⇒ D
```

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- If *e* ⇒ a nonempty list L, then:

  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- If *e* ⇒ a nonempty list L, then:

  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  ○

-

Q.

A.

- If *e* ⇒ a nonempty list L, then:

  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  - In *Scheme*, the car and cdr of an empty list are ***undefined***.

-

Q.

A.

- If *e* ⇒ a nonempty list L, then:

  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  - In *Scheme*, the car and cdr of an empty list are ***undefined***.

- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

  E.g.:

Q.

A.

- If *e* ⇒ a nonempty list L, then:

      (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

      (CDR *e*) ⇒ the list obtained from L
                  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.

  This is illogical, but sometimes convenient!

  ○ In *Scheme*, the car and cdr of an empty list are ***undefined***.

- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

  E.g.: We get an ***error*** if (CAR (+ 3 4)), (CDR (+ 3 4)), (CAR (CAR '(A B))), or (CDR (CAR '(A B))) is evaluated!

Q.

A.

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the *first* element of that list L.
  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.
  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  ○ In *Scheme*, the car and cdr of an empty list are ***undefined***.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!
  E.g.: We get an ***error*** if (CAR (+ 3 4)), (CDR (+ 3 4)), (CAR (CAR '(A B))), or (CDR (CAR '(A B))) is evaluated!

Q. What happens if (CAR '(+ 3 4)) is evaluated?
What happens if (CDR '(+ 3 4)) is evaluated?

A.

- If *e* ⇒ a nonempty list L, then:

  (CAR *e*) ⇒ the *first* element of that list L.

  Example: (CAR '(DOG CAT (AT (3 +)))) ⇒ DOG

- If *e* ⇒ a nonempty list L, then:

  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.

  Example: (CDR '(DOG CAT (AT (3 +)))) ⇒ (CAT (AT (3 +)))

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
  This is illogical, but sometimes convenient!
  - ○ In *Scheme*, the car and cdr of an empty list are ***undefined***.

- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

  E.g.: We get an ***error*** if (CAR (+ 3 4)), (CDR (+ 3 4)), (CAR (CAR '(A B))), or (CDR (CAR '(A B))) is evaluated!

Q. What happens if (CAR '(+ 3 4)) is evaluated?
What happens if (CDR '(+ 3 4)) is evaluated?

A. (CAR '(+ 3 4)) ⇒ + (CDR '(+ 3 4)) ⇒ (3 4)

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the ***first*** element of that list L.

- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by ***omitting*** its first element.

- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.

- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the *first* element of that list L.
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the ***first*** element of that list L.
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

Alternative Names for CAR and CDR

- **FIRST** is another name for **CAR** in Common Lisp.
- **REST** is another name for **CDR** in Common Lisp.

- If *e* ⇒ a nonempty list L, then:
  (CAR *e*) ⇒ the *first* element of that list L.
- If *e* ⇒ a nonempty list L, then:
  (CDR *e*) ⇒ the list obtained from L
  by *omitting* its first element.
- (CAR NIL) ⇒ NIL and (CDR NIL) ⇒ NIL.
- If *e* ⇒ an atom other than NIL, then evaluation of (CAR *e*) or (CDR *e*) will produce an ***error***!

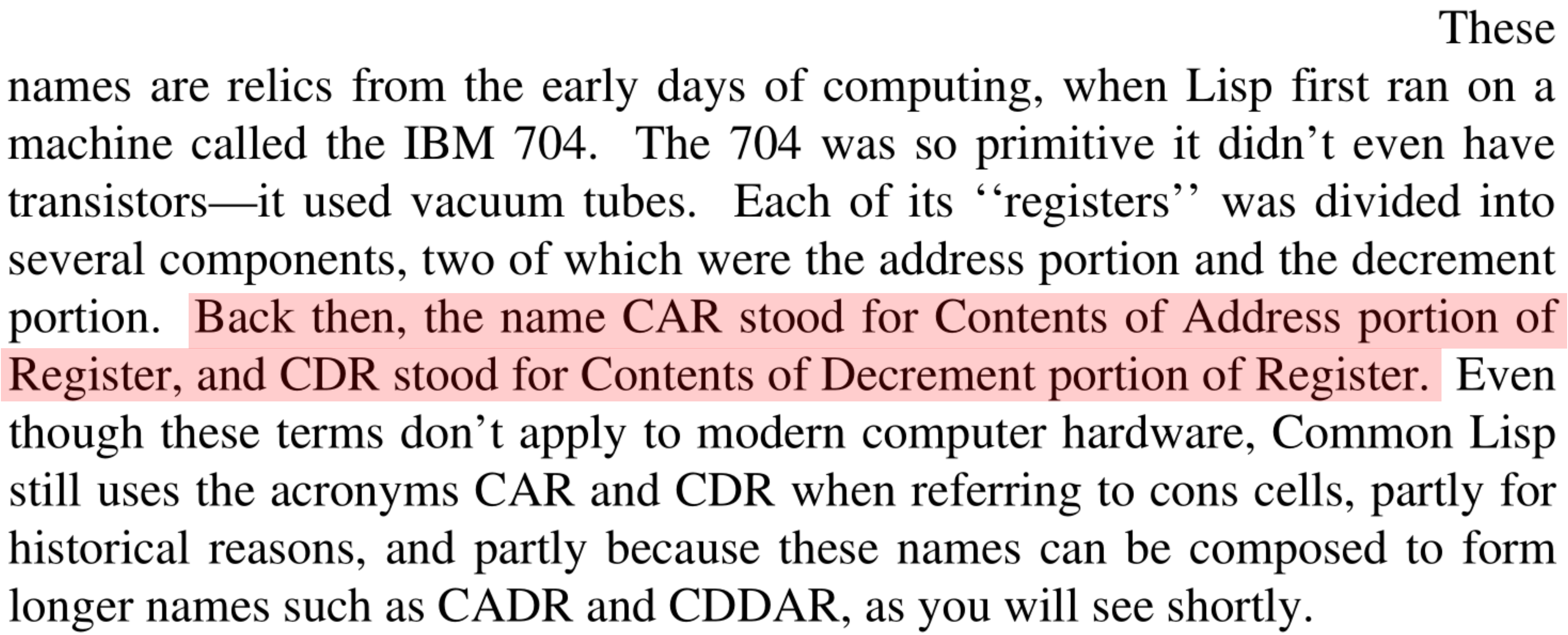
Alternative Names for CAR and CDR

- **FIRST** is another name for **CAR** in Common Lisp.
- **REST** is another name for **CDR** in Common Lisp.

Thus: (FIRST *e*) = (CAR *e*)   (REST *e*) = (CDR *e*)

The names FIRST and REST have the advantage of being descriptive, but "CAR" and "CDR" provide the basis for the C…R function names we will consider later.

The following remarks regarding the origin of the names **CAR** and **CDR** are from pp. 42 – 3 of Touretzky:



These names are relics from the early days of computing, when Lisp first ran on a machine called the IBM 704. The 704 was so primitive it didn't even have transistors—it used vacuum tubes. Each of its ''registers'' was divided into several components, two of which were the address portion and the decrement portion. Back then, the name CAR stood for Contents of Address portion of Register, and CDR stood for Contents of Decrement portion of Register. Even though these terms don't apply to modern computer hardware, Common Lisp still uses the acronyms CAR and CDR when referring to cons cells, partly for historical reasons, and partly because these names can be composed to form longer names such as CADR and CDDAR, as you will see shortly.