

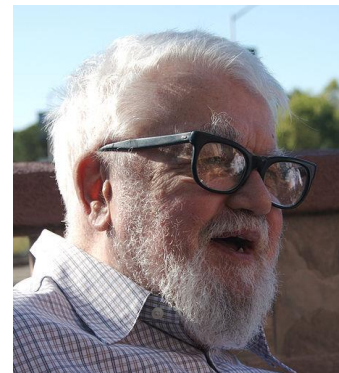


RACKET

CS A250 – C++ Programming Language 2

THE LISP FAMILY

- **LISP** programming language
 - **LIS**t **P**rocessor
 - The *first* **functional programming language** for list processing
 - An **artificial intelligence** language
 - Based on logic and mathematics concepts
 - Developed by **John McCarthy** at the MIT Research Laboratory of Electronics in 1958



SOME FACTS

- Interest in **artificial intelligence** started in the 50's
 - **Linguists** were concerned with natural language processing.
 - **Psychologists** were interested in modeling human information storage and retrieval.
 - **Mathematicians** were interested in mechanizing certain intelligence processes, such as theorem proving.
 - All of these investigations came to a conclusion:
 - Some method must be developed to allow computers to process symbolic data in **linked lists** (at the time, all computation was on numeric data in arrays)
- **John McCarthy** developed the **MIT Artificial Intelligence** project

RACKET

- **Racket** (or **DrRacket**) is a “dialect” of LISP
 - Formerly known as **Scheme** (or **DrScheme**)
 - Very small language with simple syntax
 - Well-suited to educational applications
- The original LISP is called “pure Lisp”
- Another dialect of LISP is COMMON LISP

INTERPRETER VS. COMPILER

- **Racket** is an **interpreter** (*not* a compiler)
 - A **compiler** is a program that translates high-level language source code into *machine language*
 - It collects and organizes instructions
 - Compilers run faster
 - An **interpreter** translates high-level instructions into an *intermediate form*, and then executes
 - Interpreters execute faster
 - So you can run small pieces of your program.

DATA STRUCTURES

- **Racket** (and any **LISP** language) has *only two* data structures:
 - **Atoms**
 - Symbols
 - Numeric literals
 - **Lists**
 - Specified by delimiting their elements with parentheses
- Both **symbols** and **lists** are preceded by an apostrophe (')
 - **Numeric literals** are **not** preceded by an apostrophe

LISTS

- The symbol () denotes a **list**
- A list is preceded by an **apostrophe**
 - '(1 2 (3))
- Examples:
 - '() ← **Null** or **empty list**
 - '(Jane)
 - '(Jane Jill)
 - '((a) (((56) (Jane Jill)) ()) (b) 1)
- To be a list the opening and closing parenthesis ***must*** match.

ATOMS AND LISTS

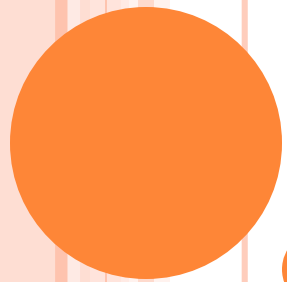
○ Example:

‘(A (B C) D (E (F G)))’

○ **Four** elements

- The first is the atom A
- The second is the sublist (B C)
- The third is the atom D
- The fourth is the sublist (E (F G)), which has its
 - first element in E and
 - second element in the sublist (F G)

‘(A (B C) D (E (F G)))’



PRIMITIVE FUNCTIONS

PRIMITIVE FUNCTIONS

- The following are three **primitive functions** we will use:
 - **first**
 - Outputs the first data expression in a **non-null list**.
 - **rest**
 - Takes a **non-null list** and *erases* the *first* data expression, outputting the **rest** of the input list.
 - **cons**
 - Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))
```

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))  
1
```

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))
```

```
1
```

```
> (first '((4) (5) ((6 7))))
```

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))
```

```
1
```

```
> (first '((4) (5) ((6 7))))
```

```
'(4)
```

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))
```

```
1
```

```
> (first '((4) (5) ((6 7))))
```

```
'(4)
```

```
> (first '(((8 9)) (10) ((11 12))))
```

PRIMITIVE FUNCTIONS (CONT.)

○ first

- Outputs the **first data expression** (*not* the first atom) in a **non-null list**.

```
> (first '(1 2 3))
```

```
1
```

```
> (first '((4) (5) ((6 7))))
```

```
'(4)
```

```
> (first '(((8 9)) (10) ((11 12))))
```

```
'((8 9))
```


PRIMITIVE FUNCTIONS (CONT.)

o **rest**

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest ' (1 2 3))
```

PRIMITIVE FUNCTIONS (CONT.)

o **rest**

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest ' (1 2 3))  
' (2 3)
```

PRIMITIVE FUNCTIONS (CONT.)

o rest

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest ' (1 2 3))  
' (2 3)
```

```
> (rest ' ((4) (5) ((6 7))))
```

PRIMITIVE FUNCTIONS (CONT.)

o rest

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest '(1 2 3))  
'(2 3)
```

```
> (rest '((4) (5) ((6 7))))  
'((5) ((6 7)))
```

PRIMITIVE FUNCTIONS (CONT.)

o rest

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest ' (1 2 3))  
' (2 3)
```

```
> (rest ' ((4) (5) ((6 7))))  
' ((5) ((6 7)))
```

```
> (rest ' (((8 9)) (10) ((11 12))))
```

PRIMITIVE FUNCTIONS (CONT.)

o rest

- Takes a **non-null list** and **removes** the **first data expression** (*not* the first atom), outputting the **rest** of the input list.

```
> (rest '(1 2 3))  
'(2 3)
```

```
> (rest '((4) (5) ((6 7))))  
'((5) ((6 7)))
```

```
> (rest '(((8 9)) (10) ((11 12))))  
'((10) ((11 12)))
```

PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input ***must*** be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))
```

PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))  
'(bob jen jill)
```


PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))  
'(bob jen jill)  
  
> (cons 1 '(2 3))
```

PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))  
'(bob jen jill)
```

```
> (cons 1 '(2 3))  
'(1 2 3)
```

PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))  
'(bob jen jill)
```

```
> (cons 1 '(2 3))  
'(1 2 3)
```

```
> (cons 'a '(((b) c (((d))))))
```

PRIMITIVE FUNCTIONS (CONT.)

o cons

- Takes **two** inputs, a **data expression** and a **list** (the second input *must* be a list), and **constructs a new list** by inserting the **first input** as the **first data expression in the second input**.

```
> (cons 'bob '(jen jill))  
'(bob jen jill)
```

```
> (cons 1 '(2 3))  
'(1 2 3)
```

```
> (cons 'a '(((b) c (((d))))))  
'(a ((b) c (((d)))))
```

DEFINING EXPRESSIONS

- Using the keyword **define**, we can define expressions to re-use them.

```
> (define x '(a b c))
```

```
> (first x)
```

```
'a
```

```
> (rest x)
```

```
'(b c)
```

```
> (cons 'm x)
```

```
'(m a b c)
```

MIXED EXPRESSIONS

- Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))
```

MIXED EXPRESSIONS (CONT.)

- Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))  
  
> (rest a)
```

MIXED EXPRESSIONS (CONT.)

○ Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))  
  
> (rest a)  
'(B C)
```


MIXED EXPRESSIONS (CONT.)

○ Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))  
  
> (rest a)  
'(B C)  
  
> (first (rest b))
```

MIXED EXPRESSIONS (CONT.)

○ Example

```
> (define a '(A B C))
> (define b '((Bob) and Peterson))
> (define c '((1 2 3)))
> (define d '(( ) ( )))

> (rest a)
'(B C)

> (first (rest b))
'and
```

MIXED EXPRESSIONS (CONT.)

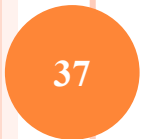
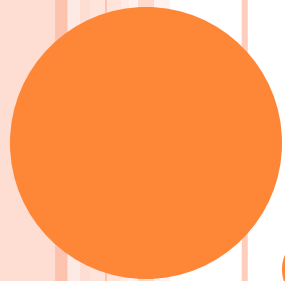
○ Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))  
  
> (rest a)  
'(B C)  
  
> (first (rest b))  
'and  
  
> (cons (first c) (rest d))
```

MIXED EXPRESSIONS (CONT.)

○ Example

```
> (define a '(A B C))  
> (define b '((Bob) and Peterson))  
> (define c '((1 2 3)))  
> (define d '(( ) ( )))  
  
> (rest a)  
'(B C)  
  
> (first (rest b))  
'and  
  
> (cons (first c) (rest d))  
'((1 2 3) ( ))
```



37



PREDICATE FUNCTIONS

PREDICATE FUNCTIONS

- Three important **predicate functions** among **Racket**'s **primitive functions** are:
 - **equal?**
 - Outputs *true* if the two inputs are equal, *false* otherwise
 - Different from *eq?*
 - Note that *eq?* will *not* output *true* if two lists are the same, because some implementations consider it a pointer
 - **list?**
 - Outputs *true* if it is a list, *false* otherwise
 - **empty?**
 - Outputs *true* if the list is empty, *false* otherwise

PREDICATE FUNCTIONS (CONT.)

◦ **equal?**

- Outputs **true** if the two inputs are equal, **false** otherwise

```
> (equal? 'a 'a)
```

PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)  
#t
```


PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)
```

```
#t
```

```
> (equal? 'a 'b)
```

PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)
```

```
#t
```

```
> (equal? 'a 'b)
```

```
#f
```

PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)
#t

> (equal? 'a 'b)
#f

> (equal? '() '(()))
```

PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)
#t

> (equal? 'a 'b)
#f

> (equal? '() '(()))
#f
```

PREDICATE FUNCTIONS (CONT.)

◦ `equal?`

- Outputs `true` if the two inputs are equal, `false` otherwise

```
> (equal? 'a 'a)
```

```
#t
```

```
> (equal? 'a 'b)
```

```
#f
```

```
> (equal? '() '(()))
```

```
#f
```

```
> (equal? '() (first '(())))
```

PREDICATE FUNCTIONS (CONT.)

◦ equal?

- Outputs **true** if the two inputs are equal, **false** otherwise

```
> (equal? 'a 'a)
```

```
#t
```

```
> (equal? 'a 'b)
```

```
#f
```

```
> (equal? '() '(()))
```

```
#f
```

```
> (equal? '() (first '(())))
```

```
#t
```

PREDICATE FUNCTIONS (CONT.)

◦ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))
```

PREDICATE FUNCTIONS (CONT.)

○ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))  
#t
```


PREDICATE FUNCTIONS (CONT.)

○ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))
```

```
#t
```

```
> (list? '())
```

PREDICATE FUNCTIONS (CONT.)

◦ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))
```

```
#t
```

```
> (list? '())
```

```
#t
```

PREDICATE FUNCTIONS (CONT.)

○ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))
```

```
#t
```

```
> (list? '())
```

```
#t
```

```
> (list? 'a)
```

PREDICATE FUNCTIONS (CONT.)

○ list?

- Outputs **true** if it is a **list**, **false** otherwise

```
> (list? '(x y))
```

```
#t
```

```
> (list? '())
```

```
#t
```

```
> (list? 'a)
```

```
#f
```

PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )
```

PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )  
#t
```

PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )
```

```
#t
```

```
> (empty? ' (1 2) )
```

PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )
```

```
#t
```

```
> (empty? ' (1 2) )
```

```
#f
```


PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )
```

```
#t
```

```
> (empty? ' (1 2) )
```

```
#f
```

```
> (empty? (first ' ( ( ) (1 2 3) ) ) )
```

PREDICATE FUNCTIONS (CONT.)

◦ empty?

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? ' ( ) )
```

```
#t
```

```
> (empty? ' (1 2) )
```

```
#f
```

```
> (empty? (first ' ( ( ) (1 2 3) ) ) )
```

```
#t
```

PREDICATE FUNCTIONS (CONT.)

◦ empty?

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? '())
```

```
#t
```

```
> (empty? '(1 2))
```

```
#f
```

```
> (empty? (first '(() (1 2 3))))
```

```
#t
```

```
> (empty? '(()))
```

PREDICATE FUNCTIONS (CONT.)

◦ **empty?**

- Outputs **true** if the input is a **null list**, **false** otherwise

```
> (empty? '())
```

```
#t
```

```
> (empty? '(1 2))
```

```
#f
```

```
> (empty? (first '(() (1 2 3))))
```

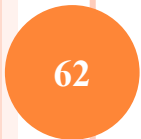
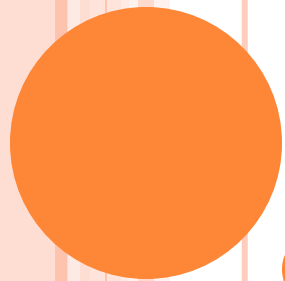
```
#t
```

```
> (empty? '(()))
```

```
#f
```

NAMING CONVENTIONS

- **Racket** is **case-sensitive**
 - **BUT** some implementations of **Lisp sub-languages** are *not* case-sensitive.
- **Identifiers** *can* begin with a digit
 - **BUT** some implementations of **Lisp sub-languages** *cannot* begin with a digit.
- The **conventional way** in **Racket** is to use **lower case** and separate words with a **dash (-)**
 - For example: list-of-names
 - *We will use the conventional way (of course)*



RACKET SOFTWARE

62

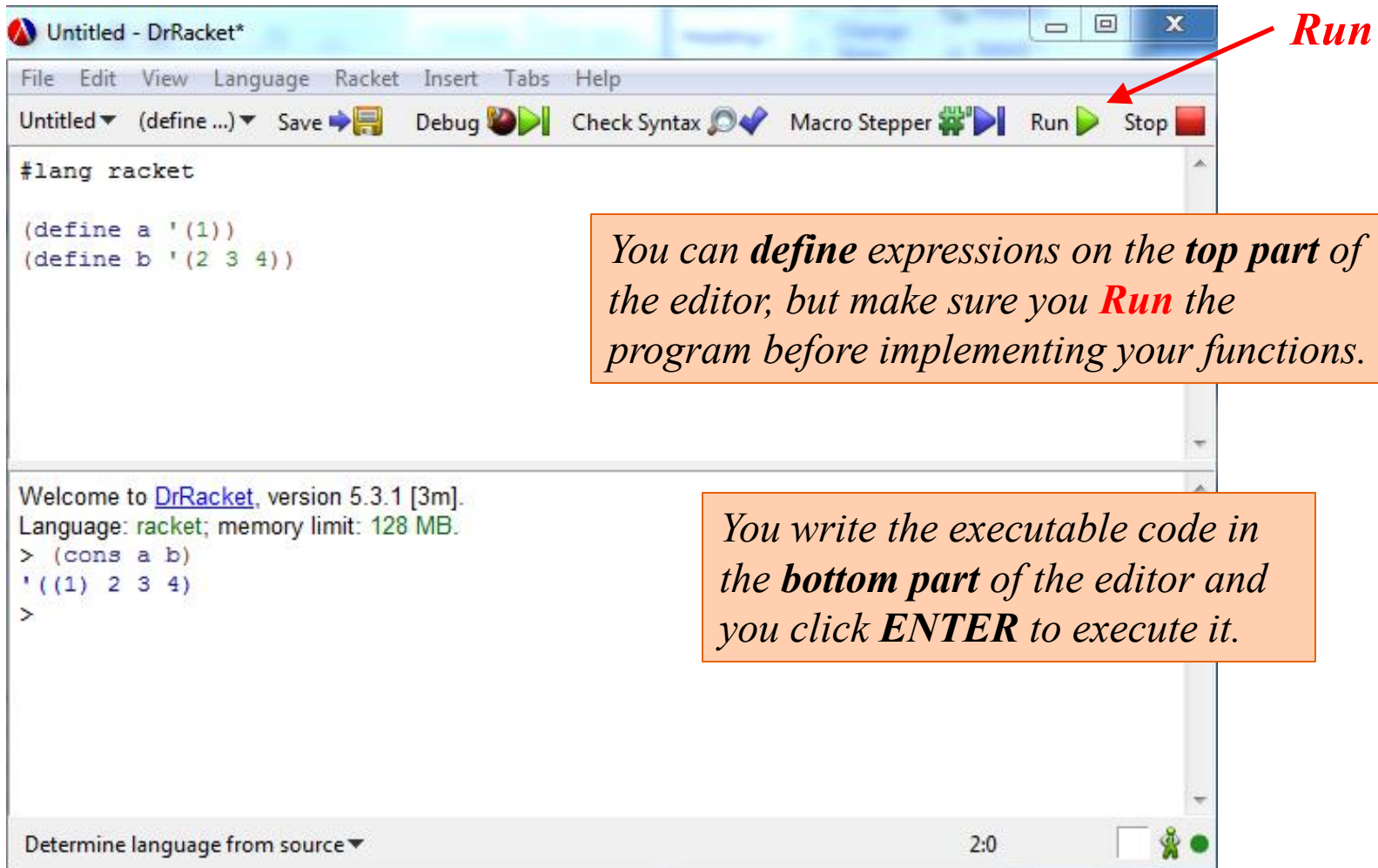
RACKET SOFTWARE

- Where to download from:
 - PLT Racket <http://racket-lang.org/>
 - For both PC and Mac

RACKET SOFTWARE (CONT.)

- How to start the program:
 - Open **DrRacket**
 - Select **Language** on the top menu
 - Click on **Choose Language...**
 - Depending on the version of the software, select:
 - **Use the Racket Language (ctl-R)**
 - Click **OK**
 - This will show **#lang racket** in the top section of the window
 - Click **Run** on the top menu to validate the new language
 - **Run** will also clear all the code in your editor.

HOW TO USE THE RACKET SOFTWARE



COMMENTS

- ... And, of course, you **ALWAYS** need to write a **name header** and comments...
- Use a **semicolon (;)** before comments

```
;some comment here  
(define a '(1 2 3))
```

- Note that comments are placed in the ***upper*** portion of the editor



RACKET 1 (END)

67