Racket

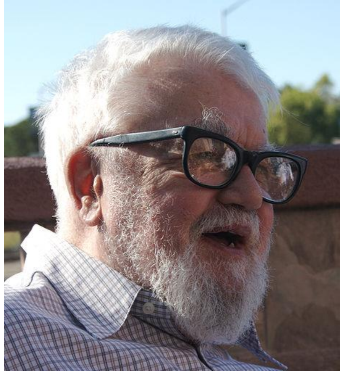
CS A250 – C++ Programming Language 2

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 <u>***run***</u> faster

  - An **interpreter** translates high-level instructions into an *intermediate form*, and then executes
    - Interpreters <u>***execute***</u> 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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

- **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.)

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) ())
```

CAUTION!

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
```

Caution! (cont.)

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
This will give you an error, because
the first element cannot be removed.
```

Caution! (cont.)

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
This will give you an error, because
the first element cannot be removed.

> (cons 'Jane a)
```

Caution! (cont.)

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
This will give you an error, because
the first element cannot be removed.

> (cons 'Jane a)
`(Jane)
```

Caution! (cont.)

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
This will give you an error, because
the first element cannot be removed.

> (cons 'Jane a)
‘(Jane)

> (cons 'Jane (rest a))
```

Caution! (cont.)

- The **rest** function **removes** the first element. If there is **no** first element (empty list), the result will be an **error**.

```
> (define a '())

> (rest a)
This will give you an error, because
the first element cannot be removed.

> (cons 'Jane a)
‘(Jane)

> (cons 'Jane (rest a))
Error
```

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

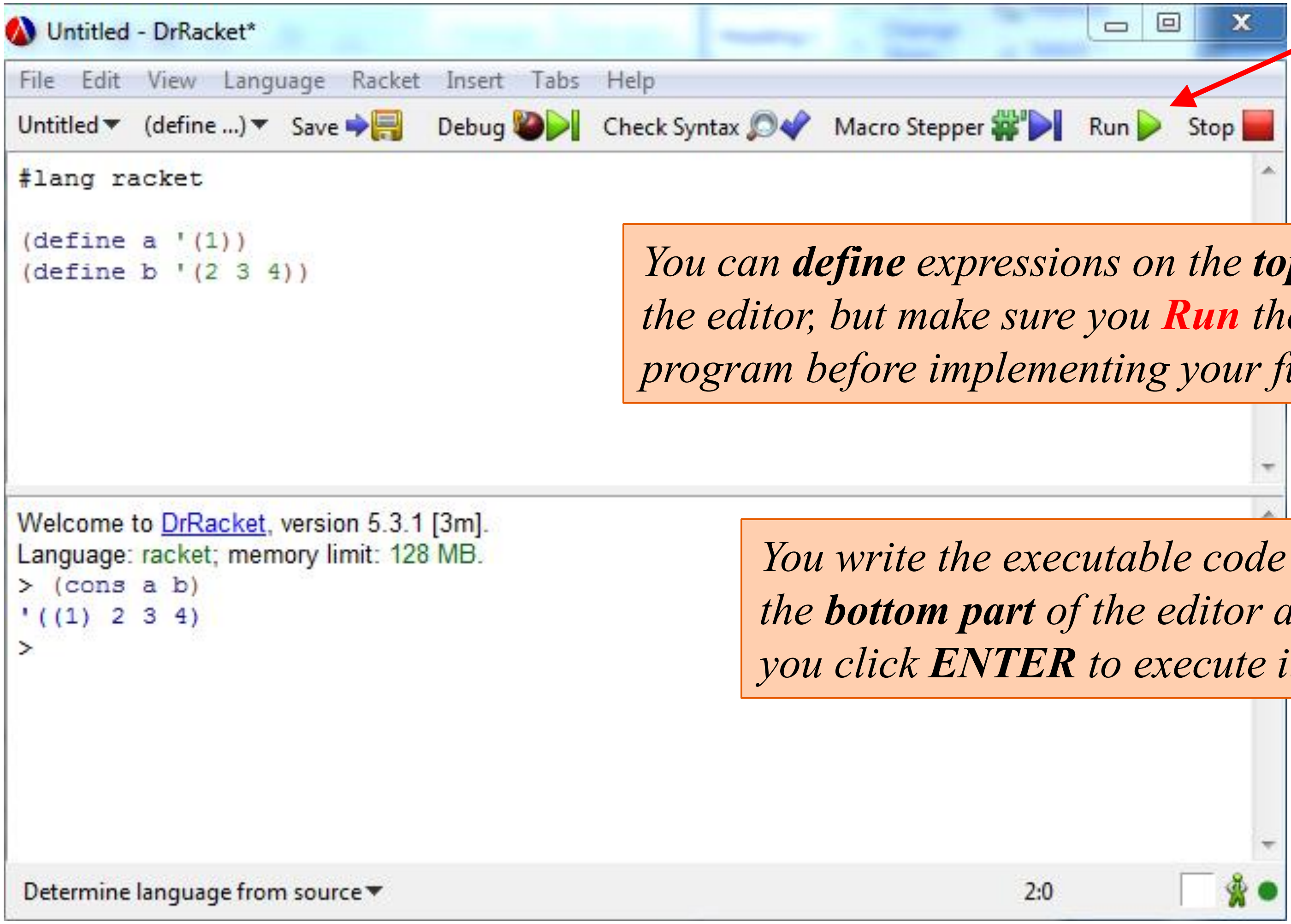
Racket Software

- 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



```
#lang racket

(define a '(1))
(define b '(2 3 4))
```

```
Welcome to DrRacket, version 5.3.1 [3m].
Language: racket; memory limit: 128 MB.
> (cons a b)
'((1) 2 3 4)
>
```

Run

*You can **define** expressions on the **top part** of the editor, but make sure you **Run** the program before implementing your functions.*

*You write the executable code in the **bottom part** of the editor and you click **ENTER** to execute it.*

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)