CS 152
Programming Paradigms

Scheme

Today

- Evaluation Rule
- Lists
- Branching
- Function Definitions

Course Learning Outcomes

2. Have a basic knowledge of the procedural, object-oriented, functional, and logic programming paradigms.
11. Produce programs in a functional programming language in excess of 200 LOC.

Evaluation Rule

A parenthesized expression (list) is evaluated as follows:

- If the first item is a keyword, a special rule is applied to evaluate the rest of the expression. An expression starting with a keyword is called a special form. (define, quote, if, cond, lambda)
- Otherwise, it is a function application. Each expression within the parentheses is evaluated recursively. The first expression must evaluate to a function, which is then applied to remaining values (its arguments)

Function Applications

- All expressions must be written in prefix form: (+ 2 3)
- + is a function, and it is applied to the values 2 and 3, to return the value 5
- A function is represented by the first expression in an application: +
- A function call is surrounded by parentheses (+ 2 3)

Applicative Order

- Evaluation rule represents applicative order evaluation:
  - All subexpressions are evaluated first
  - A corresponding expression tree is evaluated from leaves to root

Simple Arithmetics

```
> (+ 1 2 3)
6
> (/ 10 2)
5
> (* 2 (+ 1 2 1))
8
> (4 + 2)
    application: not a procedure;
     expected a procedure that can be applied to arguments
      given: 4
```

Each expression within the parentheses is evaluated recursively. The first expression **must evaluate to a function**, which is then applied to remaining values (its arguments).

iClicker

- What does the following Scheme expression evaluate to?

```
(+ 5 (* 2 3 2) (/ 8 4))
```

iClicker

How do we write ((1+2) * 5 + 3 + 10 / 5) in Scheme?
The expression evaluates to 20.

A. ((+ 1 2) * 5) + (3 10 (/ 10 2))
B. (+ (* (+ 1 2) 5 3) (/ 10 5))
C. (+ (* (+ 1 2) 5) 3 (/ 10 5))
D. (* (+ (+ 1 2) 5) 3 (/ 10 5))

Special Forms

- If the first item in a parenthesized expression is a keyword, it is called a special form:
  - quote
  - define
  - if
  - cond
  - lambda

Quote Special Form

- A problem arises when data are represented directly in a program, such as a list of numbers: (2 3 4)
- Scheme will try to evaluate it as a function call
- We prevent this using a special form with the keyword quote
- (quote (2 3 4))
- Rule for evaluating a quote special form is to simply return the expression following quote without evaluating it

Quote Special Form

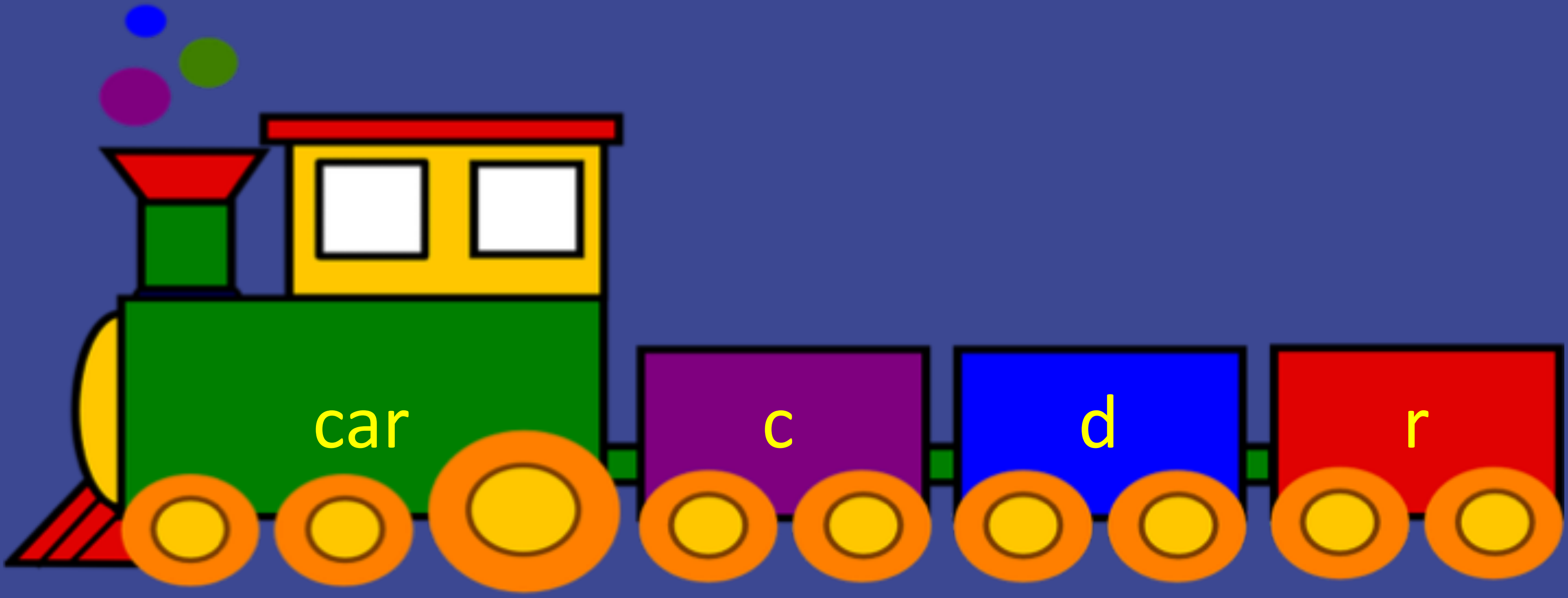
- Shorthand notation for quote special form: '
  - 'grade
  - '(2 3 4)
  - '(red yellow blue green orange)
  - '("cs 152" "cs 151" "cs 146")

Lists

```
>'(red yellow blue green orange)
(red yellow blue green orange)
>(define colors '(red yellow blue green orange))
>colors
(red yellow blue green orange)
```

List of symbols

Lists: car and cdr



Lists: car and cdr

```
>(define colors '(red yellow blue green orange))
>colors
(red yellow blue green orange)
>(car colors) ; first item in the list colors
red
>(cdr colors) ; the rest - the list without the first item
(yellow blue green orange)
```

car: Contents of Address Register
cdr: Contents of Data Register

Lists: car and cdr

```
>(define colors '(red yellow blue green orange))
>colors
(red yellow blue green orange)
>(car (cdr colors))
yellow
>(cadr colors) ; short for (car (cdr colors))
yellow
```

Lists: car and cdr

```
>colors
(red yellow blue green orange)
>(cdr (cdr colors))
(blue green orange)
 >(cddr colors)
(blue green orange)
>(cdddr colors)
(green orange)
```

iClicker

```
>(define xs '(1 2 3 4 5))
```

(cadddr xs) is a:

A. Number
B. List

iClicker

```
>(define xs '(1 2 3 4 5))
```

(cadddr xs) is :

A. 1
B. 2
C. 3
D. 4
E. 5

cons and Pairs

A pair (sometimes called a dotted pair) is a data abstraction that can be constructed with the Scheme function *cons*.

cons takes two arguments and returns a compound object that contains the two arguments as parts.

```
> (cons 1 2)
(1 . 2)
```

The dot indicates that this a pair – not a list.

cons and Pairs

Given a pair, we can extract its parts using car and cdr.

```
> (define p (cons 1 2))
> p
(1 . 2)
> (car p)
1
> (cdr p)
2
```

Pairs

cons can be used to form pairs whose elements are pairs.

```
> (define p (cons (cons 1 2) 3))
> p
> ((1 . 2) . 3)
> (car p)
> (1 . 2)
> (cdr p)
3
```

the cdr of a list is always a list but the cdr of a pair may be of any type.

Why Pairs?

- Pairs can be used as general-purpose building blocks to create all sorts of complex data structures.
- Lists are built with pairs.

From Pairs to Lists

A list is a pair whose second part (cdr) is a list.

A list of length 1 is a pair whose second part is the empty list.

```
> (cons 1 '())
(1)
```

'() is the empty list

From Pairs to Lists

```
>(define a (cons 1 '()))
> a
(1)
>(define b (cons 2 a))
> b
(2 1)
>(define c (cons 3 b))
> c
(3 2 1)
```

From Pairs to Lists

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

Or we can write:

```
> (list 3 2 1) ; shorthand for (cons 3 (cons 2 (cons 1 '())))
(3 2 1)
```

list vs quote

```
>(define a 1)
>(define b 2)
>(list a b)
(1 2 )
> '(a b)
(a b)
```

quote: no evaluation

Growing Lists with cons

```
> (define scores '(90 80))
> (cons 100 scores)
(100 90 80)
```

This is functional programming. We do not mutate the existing list, we create a new one.

iClicker: Growing Lists with cons

(define colors '(red yellow blue green orange))

How do we use *cons* to create a new list with the symbol purple and all the existing colors in the list colors?

A. (cons 'purple colors)
B. (cons 'purple 'colors)
C. (list 'purple colors)

iClicker: Growing Lists with cons

(define colors '(red yellow blue green orange))

How do we use *cons* to create a new list with the symbol purple and all the existing colors in the list colors?

A. (cons 'purple colors) ✓
B. (cons 'purple 'colors) -> (purple . colors)
C. (list 'purple colors) -> (purple (red yellow blue green orange))

Empty lists

null? is a predicate thar returns #t if its argument is the empty list.

It is very useful in checking the base case of recursive functions.

```
> (null? '())
#t
> (null? '(1 2 3))
#f
> (define seq '(1))
> (null? (cdr seq))
#t
```

Lists: Summary

- Empty list: '()
- List head (element): car
- List tail (list): cdr
- Check for empty list: null?
- Contructor: cons – to create a list we *cons* an element and a list.

Review: Scheme Expressions

Parenthesized expressions can be:

- A function application: each expression within the parentheses is evaluated recursively. The first expression must evaluate to a function, which is then applied to remaining values (its arguments)
- A special form

Special Forms

- If the first item in a parenthesized expression is a keyword, it is called a special form:
  - ✓ quote
  - ✓ define
  - if
  - cond
  - lambda

Review: Binding with *define*

```
> (define grade 75)
> grade
75
> (+ 2 grade)
77
```

if Special Form

```
(if <condition> <true_result> <false_result>)
```

Example:

```
> (define grade 85)
> (if (>= grade 70) "Credit"  "No Credit")
"Credit"
```

if Special Form

(if <condition> <true_result> <false_result>)

Letter grades?

```
(define grade 85)
(if (>= grade 90)
    "A"
    (if (>= grade 80)
        "B"
        (if (>= grade 70)
            "C"
            (if (>= grade 60)
                "D"
                "F"))))
```

cond Special Form

Multi-branch conditionals:

```
(cond (<c1> <e1>)
      (<c2> <e2>)
       …
      (<cn> <en>)
      (else <else-expression>))
```

cond Special Form

Multi-branch conditionals:

```
(cond ((>= grade 90) "A")
      ((>= grade 80) "B")
      ((>= grade 70) "C")
      ((>= grade 60) "D")
      (else "F"))
"B"
```

lambda Special Form

Lambda expressions evaluate to anonymous functions

(lambda (<formal-parameters>) <body>)

(lambda (x) (+ 1 x))

Function Application

We can write:

((lambda (x) (+ 1 x)) 5) ; apply the anonymous function

6

We can also bind the function :

(define increment (lambda (x) (+ 1 x))) ; bind the function

(increment 5) ; apply the named function

6

Named Function Definitions

Two equivalent expressions:

```
(define increment (lambda (x) (+ 1 x)))
(define <name> (lambda (<formal-parameters>) <body>))

(define (increment x) (+ 1 x)))
(define (<name> <formal parameters>) <body>)
```

Example: square

Expected Behavior:

> (square 8)

64

Example: square

```
;;; Function square:  number -> number
;;; Returns the square of a given number
(define
  (square x)
  (* x x))
> (square 8)
64
> (square 2  3)
square: arity mismatch; the expected number of arguments does not
match the given number  expected: 1  given: 2
```

```
(define
 (<name> <formal parameters>)
  <body>)
```

To Do

- Homework 2
  - Individual assignment
  - Questions? Canvas discussion forum
  - Due September 8 at 5PM