# Strings

CS135 Lecture 12

# L12.0 Characters

# Information Interchange

Internally, a computer represents everything as numbers. All the values we use in Racket are represented internally by one or more numbers, including lists.

```
16      1/3      true      'hello      (list 'a 'b 'c)
```

When we are just computing the value of functions and expressions, we don’t care how the values are represented internally, but when we want to display the results of a computation we need to turn the numbers into pictures, sound, text, etc.

When you hit the “A” key on your keyboard, we need a standard number that means “A”, so that it is stored correctly in files and displayed correctly on this screen. That number is 65.

# Unicode

An international standard (called “Unicode”) assigns a unique number (called a “code point”) to almost every character in almost every written human language.

Lowercase English letter a = 97
Greek small letter alpha (α) = 945
East Asian character 日 = 26,085
Ancient Egyptian hieroglyph 𓂀 = 78,323

English letters, punctuation, etc. are assigned code points less than 128. They were assigned these values before Unicode was invented, under a earlier system called ASCII (American Standard Code for Information Interchange).

Normally code points are written in base 16 prefaced by a U+ (e.g., U+0061 = 97).

# What is a “character”?

These glyphs represent the same character. They use different fonts but have the same code point: A 𝐴 𝑨 𝓐 𝔸

These glyphs represent different characters. They look the same but have different code points:

A — Latin Capital Letter A (65)
A — Greek Capital Letter Alpha (913)
A — Cyrillic Capital Letter A (1040)
Ꭺ — Cherokee Letter A (5034)

A Unicode code point represents a character in an abstract sense. A “glyph” is the graphical shape that visually represents that character in a particular font or style.

# When are two characters the “same character”?

Lowercase “a” and uppercase “A” are in some sense the “same character”, but they are assigned different code points, SO THAT THIS SENTENCE DOESN’T HAVE TO BE ENTIRELY IN CAPITAL LETTERS.

In some sense the following characters are the “same character”, but they are assigned different code points due to the historical evolution of East Asian writing systems: 驿 (simplified Chinese), 驛 (traditional Chinese), 駅 (Japanese).

Unless you are writing very low level font or display code for an operating system, web browser, etc., you can generally assume that two characters are the “same character” if they have the same code point.

# When are characters the “same character”?

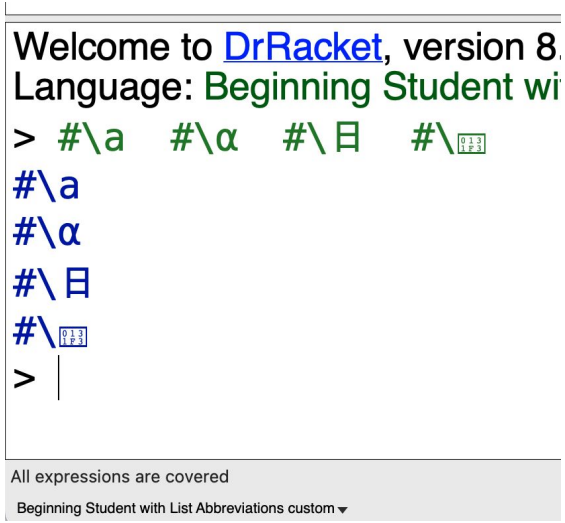
These three characters share the codepoint (39592) but are not the “same character” in exactly the same sense that 驿, 驛 and 駅 are not the “same character”.

In some cases, characters that look distinct in different locales are represented by the same codepoint. This example, and others like it, are essentially mistakes in Unicode. See https://en.wikipedia.org/wiki/Han_unification

# Characters in Racket

In Racket, when you want to refer to a character as the character itself, it is written in a special format, starting with `#\`, followed by the character. For example:

```
#\a   #\α   #\日   #\□
```




DrRacket doesn't have a font for □, so it is written as a little box containing the base-16 number corresponding to the code point: U+0131f3 = 78,323. If you are reading this as a PDF, it may also not appear correct.

However this is still an acceptable character.

# Operations on characters

For CS135 we will only need three operations on characters (unless we indicate otherwise on an assignment or exam): **`char?`**, **`char=?`** and **`char<?`**

The following expressions are **`true`**:

```
(char? #\a)   (char=? #\日 #\日)   (char<? #\a #\日)
```

Generally, characters with a natural “alphabetical order” are assigned numbers that reflect that order. The following expressions are **`true`**:

```
(char<? #\a #\b)   (char<? #\α #\β)   (char<? #\A #\Z)
```

But things don’t always work the way you might expect:

```
(char<? #\a #\B) ⇒ false
```

# L12.1 Strings

# Strings

In Racket, a string is a sequence of zero or more characters enclosed in double quotes:

```
""
"Hello world!"
"Chat, j'ai pété!"
"An investment in knowledge pays the best interest."
"L'instruction est la clé du succès."
"好好学习,天天向上。"
"Ich verstehe nur Bahnhof"
```

We can test if a value is a string with the predicate `string?`

# “Escaping” with \

One obvious problem we encounter here is how to have a string with a double quote in it. How is Racket supposed to know that the double quote is part of your string and not a marker to indicate the end of your string? We avoid this confusion by using the backslash (\) as a special "escape" character: it combines with the character that follows it to tell Racket you want to include something special that would otherwise confuse it:

```
"\"Repent, Harlequin!\" Said the Ticktockman"

"This string has a single backslash (\\) in it."
```

# Strings are composite values

A string can be viewed as a composite value, i.e., as a sequence of characters. While Racket has lots of builtin string functions, in CS135 we will usually work with strings by converting them to/from lists with **`string->list`** and **`list->string`**.

```
(string->list "L'instruction est la clé du succès.")
   ⇒(list #\L #\' #\i #\n #\s #\t #\r #\u #\c #\t #\i
          #\o #\n #\space #\e #\s #\t #\space #\l #\a
          #\space #\c #\l #\é #\space #\d #\u #\space
          #\s #\u #\c #\c #\è #\s #\.)

(list->string (list #\h #\e #\l #\l #\o))
   ⇒"hello"
```

# Contracts and data definitions with characters and strings

In contracts and data definitions, we write **`Char`** to indicate a character and **`Str`** to indicate a string.

```
;; convert a string to a list of characters
;; string->list: Str -> (listof Char)

;; convert a list of characters to a string
;; list->string: (listof Char) -> Str
```

# Testing strings for equality

Racket has a built-in predicate **`string=?`** that tests two strings for equality, which you can use after this lecture. How would we write an equivalent function with the tools we have available?

```
;; Are two strings equal?
;; str=?: Str Str -> Bool
(define (str=? s0 s1) (...))

(check-expect (str=? "hello" "hello") true)
(check-expect (str=? "hello" "ciao") false)
(check-expect (str=? "" "") true)
```

# Testing strings for equality

```
;; Are two lists of characters the same?
;; char-list=?: (listof Char) (listof Char) -> Bool
(define (char-list=? loc0 loc1)
  (cond [(empty? loc0) (empty? loc1)]
        [(empty? loc1) false]
        [else (and (char=? (first loc0) (first loc1))
                   (char-list=? (rest loc0) (rest loc1)))]))

;; Are two strings equal?
;; str=?: Str Str -> Bool
(define (str=? s0 s1)
  (char-list=? (string->list s0) (string->list s1)))
```

# Pattern for string processing

In CS135, many string processing problems are solved with a basic pattern:

1. Convert each string to a list of characters (`string->list`)
2. Process the lists of characters in a helper function
3. If needed, convert lists of characters back to strings (`list->string`)

Steps #1 and #3 typically happen in a wrapper function.

# Lexicographical order

Strings are typically sorted in “lexicographical order”, as follows:

- Strings are compared character by character from left to right.
- If characters differ, the order is determined by the value of their code points.
- If one string is a prefix of another, the shorter string comes first.

Racket has a built-in predicate (`string<?`) that compares strings in lexicographical order, which you can use after this lecture.

These are all `true`:

```
(string<? "planet" "plants")
(string<? "plan" "plants")
(string<? "" "plants")
```

# Comparing strings lexicographically

```
;; Compare lists of characters in lexicographical order
;; char-list<?: (listof Char) (listof Char) -> Bool
(define (char-list<? loc0 loc1)
  (cond [(empty? loc0) (not (empty? loc1))]
        [(empty? loc1) false]
        [(char=? (first loc0) (first loc1))
         (char-list<? (rest loc0) (rest loc1))]
        [else (char<? (first loc0) (first loc1))]))

;; Compare strings in lexicographical order
;; str<?: Str Str -> Bool
(define (str<? s0 s1)
  (char-list<? (string->list s0) (string->list s1)))
```

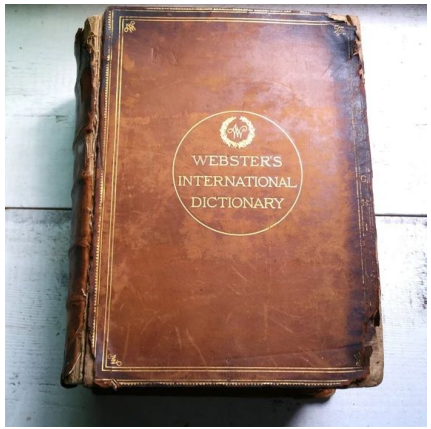
# L12.3 Dictionaries

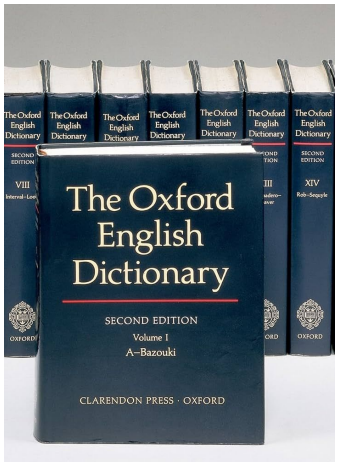
# Dictionaries

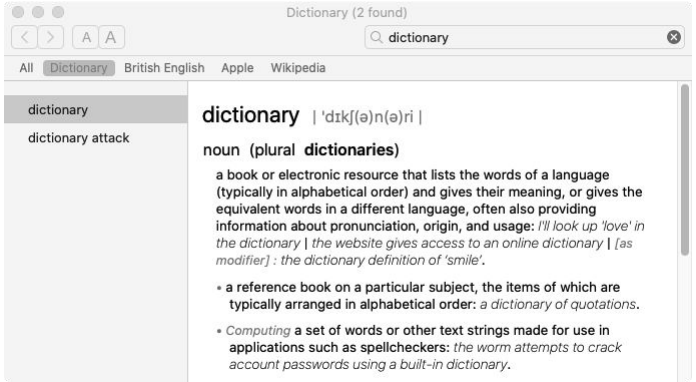
Once upon a time, a *dictionary* was a book in which you look up a word to find a definition.

Now we use the word to mean a structure for organizing data (a “data structure”).

A dictionary is a data structure that stores “key-value pairs”.



# Key-value pairs

More generally, a dictionary contains a number of unique *keys*, each with an associated *value*.

- A book of word definitions: keys are words; values are definitions.
- Your contacts list: keys are names; values are telephone numbers
- Your seat assignment for midterms: keys are userids; values are seat locations.
- Course marks:  keys are student numbers; values are marks.
- Stocks:  keys are symbols; values are prices.

Key-value pairs are a fundamental “abstraction” for organizing data.

# Keys are unique

An important aspect of dictionaries is that keys are **unique**.

Given a key, we can look it up in the dictionary and get, at most, one value.  We say "at most" because it's possible the key isn't there.  But there won't be two of them.

Values, on the other hand, may be duplicated: for example, each student has a unique student ID (key), but multiple students might have the same grade (value).

| Userid | Mark |
|---|---|
| asmith | 87 |
| bjones | 64 |
| cwang | 87 |

# Dictionary operations

What operations might we wish to perform on dictionaries?

- **`lookup`**: given a key, produce the corresponding value
- **`add`**: add a (key,value) pair to the dictionary
- **`delete`**: given a key, delete it and its associated value

The “**`lookup`**” operation is also commonly called “**`find`**” or “**`search`**” .
The “**`add`**” operation is sometimes called “**`insert`**”.
The “**`delete`**” operation is sometimes called “**`remove`**”.

A dictionary is an abstraction of a data structure, usually called an “abstract data type”. There multiple ways to implement it.

# Association lists

One simple implementation of a dictionary uses an *association list*, which is just a list of (key, value) pairs.

We store the pair as a two-element list. For our example, we will use strings as the keys and numbers as the values, but keys and values can be other types.

```
;; An association list (AL) is a (listof (list Str Num))
;; Requires: each key (Str) is unique

(define marks
  (list (list "asmith" 87)
        (list "bjones" 64)
        (list "cwang" 87)))
```

# The `lookup` operation

Recall that `lookup` consumes a key and a dictionary (association list) and produces the corresponding value when it's found. But what should `lookup` produce if it fails?

When the key is not found in the association list, we cannot produce a number. Every number is a valid value and might be the result of a successful lookup. The "not found" condition needs to be distinguishable, from successful searches. We'll use `empty` to indicate failure.

```
(check-expect (lookup "bjones" marks) 64)
(check-expect (lookup "dleeted" marks) empty)
```

# The `lookup` operation

```
;; Lookup a key-value pair in an association list
;; lookup: Str AL -> (anyof Num empty)
(define (lookup key al)
  (cond [(empty? al) empty]
        [(string=? (first (first al)) key)
         (second (first al))]
        [else (lookup key (rest al))]))

(check-expect (lookup "bjones" marks) 64)
(check-expect (lookup "dleeted" marks) empty)
```

# The **`delete`** operator

For reasons that will become clear, we will implement **`delete`** next.

```
;; Delete a key-value pair from an association list
;; delete: Str AL -> AL
(define (delete key al)
  (cond [(empty? al) empty]
        [(string=? (first (first al)) key) (rest al)]
        [else (cons (first al) (delete key (rest al)))]))

(check-expect
  (delete "bjones" marks)
  (list (list "asmith" 87) (list "cwang" 87)))
(check-expect (delete "a37858w" empty) empty)
```

# The `add` operator

The add operator adds a key-value to pair to the dictionary.

Since keys must be unique, what do we do if the key is already in the dictionary?

We could:

1. Ignore the new key-value pair and keep the existing one.
2. Delete the existing pair and add the new one.
3. Treat it as an error

In practice, #2 is the most convenient, since it allows us to *update* a value.

# The **add** operator

```
;; Add a key-value pair to an association list
;; add: Str Num al -> al
(define (add key value al)
  (cons (list key value) (delete key al)))

(check-expect (add "eyefaild" 48 marks)
              (list (list "eyefaild" 48) (list "asmith" 87)
              (list "bjones" 64) (list "cwang" 87)))
(check-expect (add "cwang" 100 marks)
              (list (list "cwang" 100) (list "asmith" 87)
                    (list "bjones" 64)))
```

# Lecture 12 Summary

# L12: You should know

- How Unicode facilitates information interchange
- The relationship between characters and strings
- How to convert between strings and lists of characters
- How to process strings as lists of characters
- The concept of a dictionary as an abstract data type
- How to use an association list to implement a dictionary

# L12: Allowed constructs

Newly allowed constructs:

```
Char char? char=? char<? Str string? string=? string<?
string->list list->string
```

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos add1 and append asin atan boolean? check-expect
check-within cond cons cons? cos define e else empty empty?
even? exp expt false first inexact? integer? length list
list? log max min not number? odd? or pi quotient rational?
remainder rest reverse second sin sqr sqrt sub1 symbol?
symbol=? tan third true zero?
listof Any anyof Bool Int Nat Num Rat Sym
```

# Other comparison predicates on characters and strings

The allowed constructs only list: **`char=? char<? string=? string<?`**

Given that it's straightforward to construct them from these predicates, we will also allow you to use other character and string comparison functions without special permission (or rather we will silently ignore your usage):

```
char<=? char>? char>=? string<=? string>? string>=?
```

In general, if both = and < forms of comparison predicates are on the allowed list, so are the other comparison predicates.

Other string and character functions require explicit permission on assignments and exams.