Ordered structures

Dr. Constantinos Constantinides, P.Eng.

Department of Computer Science and Software Engineering
Concordia University Montreal, Canada

December 31, 2021

Introduction

- Ordered structures include tuples, lists and strings.

Tuples

- A *tuple* (also called *sequence*, or *vector*) is a structure which contains a collection of elements.
- Unlike sets and bags, the ordering of the elements matters in a tuple. Unlike a set, repetitions are allowed in a tuple.
- Note that since order is important and repetitions are allowed,

$$\boxed{\begin{array}{c}(a, b, b, c) \neq (c, a, b, b) \\ \\ (a, b, c) \neq (c, a, b, b)\end{array}}$$

- The length of a tuple denotes the number of elements that it contains. A tuple of length n is called an *n-tuple*. A 2-tuple is often called an *ordered pair* and a 3-tuple is often called an *ordered triple*.

Cartesian product of sets

- For two sets A and B, an ordered pair is an object of the form (a, b) (alternative notations are $< a, b >$ and $a \mapsto b$) where $a \in A$ and $b \in B$.
- The *product* (or *Cartesian product*) of two sets A and B is defined by

$$A \times B = \{(x, y) : x \in A, y \in B\}$$

which is the set of all ordered pairs in which the first element comes from A and the second element comes from B.

Lists

- A *list* is a finite ordered sequence of zero or more elements that can be repeated.
- The elements of a list can be any kind of objects, including lists themselves in which case a list is said to be *nested* (as opposed to being *flat*).
- The number of elements in a list is called the *length* of the list.
- For example, the list $L = \langle a, b, c, d \rangle$ has length 4, its head is a and its tail is $\langle b, c, d \rangle$.
- The empty list, denoted by $\langle \rangle$ does not have a head or tail. Note that

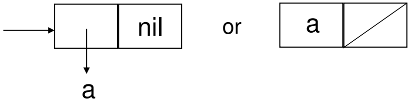
$$a \neq \langle a \rangle \neq \langle \langle a \rangle \rangle$$

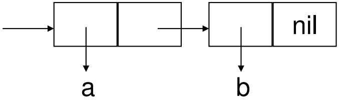
Accessing the head and tail of a list

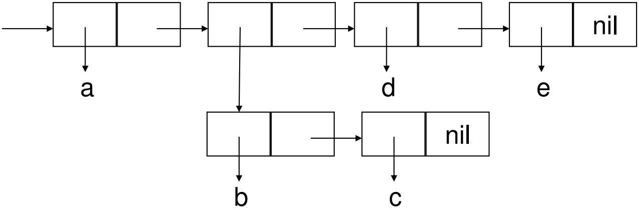
- We have two operations that can access two things in a list in one step.
- Operation *head* returns the first element of the list.
- Operation *tail* returns a list made up of all elements of the initial list except the head element.

L	$head(L)$	$tail(L)$
$\langle a, \langle b \rangle \rangle$	a	$\langle \langle b \rangle \rangle$
$\langle \langle a \rangle, \langle b, c \rangle \rangle$	$\langle a \rangle$	$\langle \langle b, c \rangle \rangle$
$\langle a \rangle$	a	$\langle \rangle$

Visualization of lists







Constructing lists

- We have three operations to create a list which are summarized below:

1. *cons*: It takes two arguments, the second of which must be a list, and returns a new list whose head is the first argument and whose tail is the second argument.
2. *list*: It creates a list comprised of its arguments.
3. *concat*: It creates a list by concatenating existing lists.

Constructing lists with *cons*

- Operation *cons* is binary (i.e. it has arity 2).
- Given an element h and a list L, $cons(h, L)$ denotes a list whose head is h and whose tail is L.
- Consider the following examples:

$$cons(a, \langle\rangle) = \langle a \rangle$$

$$cons(a, \langle b, c \rangle) = \langle a, b, c \rangle$$

$$cons(\langle a, b \rangle, \langle c, d \rangle) = \langle \langle a, b \rangle, c, d \rangle$$

- For any non-empty list L, the three operations are related:

$$cons(head(L), tail(L)) = L$$

Constructing lists with cons in Common Lisp

```
(cons ’a ’())
(A)

> (cons ’a (cons ’b ’()))
(A B)

> (cons ’a
   (cons (cons ’b (cons ’c ’())) (cons ’d (cons ’e ’()))))
(A (B C) D E)
```

Constructing lists with *list*

- Operation `list` has variable arity, i.e. it can take any number of arguments.
- The operation constructs a list comprised of these arguments.
- For example:

$$\mathit{list}(a, \langle b, c \rangle, d) = \langle a, \langle b, c \rangle, d \rangle$$

Constructing lists with `list` in Common Lisp

```
> (list ’a)
(A)

> (list ’a ’b)
(A B)

> (list ’a (list ’a ’b) ’d ’e)
(A (A B) D E)
```

Constructing lists with *concat*

- Much like *list*, operation *concat* has variable arity, i.e. it can take any number of arguments.
- Note that there is a restriction on the types of its arguments: they must all be lists. Consider the following examples:

$$concat(\langle a, b \rangle, \langle c, d \rangle) = \langle a, b, c, d \rangle$$

$$concat(\langle a, b \rangle, \langle c, \langle d \rangle \rangle) = \langle a, b, c, \langle d \rangle \rangle$$

Constructing lists with append in Common Lisp

- In LISP, function *append* implements *concat*.

```
> (append (list 'a))
(A)

> (append (list 'a) (list 'b))
(A B)

> (append (list 'a)
          (list (list 'a 'b)) (list 'd) (list 'e))
(A (A B) D E)
```

Strings and languages

- A *string* is a finite sequence of zero or more elements.
- The elements are taken from a finite set called an *alphabet*.
- For example, if $A = \{a, b, c\}$ then some strings over A are $a, ba, aabb$.
- A string with no elements is called the *empty string*, denoted by Λ.
- The number of elements in a string s is called the *length* of s denoted by $|s|$.
- For example, $|aabb| = 4$.

Strings and languages /cont.

- Concatenation is an operation to placing two strings s_1 and s_2 next to each other to form a new string. For example,

$$concat(aa, bb) = aabb$$

- Given an alphabet A, a *language* is a set of strings over A.
- Any language over A is a subset of A^*, the set of all strings over A.

Summary of ordered and unordered structures

Collection	Order	Can be infinite	Repetitions allowed
Set	No	Yes	No
Bag	No	Yes	Yes
Tuple	Yes	No	Yes
List	Yes	No	Yes
String	Yes	No	Yes