I217: Functional Programming

5. Tables (or Maps or Dictionaries)

Kazuhiro Ogata, Canh Minh Do

Roadmap

- Tables
- Tables as Lists
- Billing Program

Tables

A *table* from a set A to a set B consists of a subset A' of A and a function $f : A' \rightarrow B$.

The elements of A' are called the *keys* of the table. For $a' \in A'$, $f(a')$ is called the *value* of the key a' in the table. The pair $(a', f(a'))$ is called an *entry* of the table.

An example of tables from the set Qid of quoted IDs, such as 'a to the set Tag of (String,Nat)-pairs, such as ("apple",150):

A' is {'a,'o,'t}

f('a) is ("apple",150)

f('o) is ("orange",100)

f('t) is ("tomato",90)

Tables

Tables can be graphically drawn as follows:

keys	values
k_1	v_1
$\ldots$	$\ldots$
k_n	v_n

The keys $k_1, \ldots, k_n$ are different from each other.

The order is irrelevant.

The example of tables can be graphically drawn as follows:

keys	values
'a	("apple",150)
'o	("orange",100)
't	("tomato",90)

Tables

Tables may be called *maps* (for example in Java) and *dictionaries* (for example in Smalltalk and Python).

In Artificial Intelligence (AI), they are called *association lists* (*a-lists*).

Environments (or stores) used in imperative programming language processors, such as interpreters, can be implemented as tables from the set of variables to the set of values stored in those variables.

Tables as Lists

Tables can be expressed as lists of (key,value)-pairs:

$$(k_1,v_1) \mid \ldots \mid (k_n,v_n) \mid \mathrm{nil}$$

where $k_1,\ldots,k_n$ are different from each other and the order is irrelevant.

The example of tables can be expressed as follows:

('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | nil

Tables as Lists

Entries are implemented as pairs as follows:

```
mod! ENTRY(K :: TRIV, V :: TRIV-ERR-IF) {
  pr(PAIR(K,V) * {sort Pair -> Entry})
  [Entry ErrEntry < Entry&Err]
  op errEntry : -> ErrEntry {constr} .
  op (_,_) : Elt.K Elt&Err.V -> Entry&Err .
  var K : Elt.K .
  eq (K,err.V) = errEntry .
}
```

For error handling because the second element may be an error

If the second element is an error, then the entry is also an error.

Please see the appendices (after the exercise page) for PAIR and TRIV-ERR-IF.

Tables as Lists

Tables are implemented as lists as follows:

```
mod! TABLE {  pr(BOOL-ERR)
 pr(GLIST-ERR(E <= view from TRIV-ERR to ENTRY {
                 sort Elt -> Entry,  sort Err -> ErrEntry,
                 sort Elt&Err -> Entry&Err,  op err -> errEntry } )
   * { sort List -> Table,  sort Nil -> EmpTable,  sort NnList -> NeTable,
       sort ErrList -> ErrTable,  sort List&Err -> Table&Err,
       op errList -> errTable,  op nil -> empTable } )
 vars K K2 : Elt.K .  vars V V2 : Elt.V .  vars VE VE2 : Elt&Err.V .
 var T : Table .  var TE : Table&Err .
 ...  }
```

Since the parameterized module ENTRY is used as an actual parameter of GLIST-ERR, TABLE inherits the parameters from ENTRY.

Please see the appendices for BOOL-ERR, TRIV-ERR and GLIST-ERR.

Tables as Lists

Some functions for tables:

op singleton : Elt.K Elt.V -> Table .
op singleton : Elt.K Elt&Err.V -> Table&Err .
eq singleton(K,err.V) = errTable .
eq singleton(K,V) = (K,V) | empTable .

Given k and v,

keys	values
k	v

is made.

Tables as Lists

op isReg : Table Elt.K -> Bool .
op isReg : Table&Err Elt.K -> Bool&Err .
eq isReg(errTable,K2) = errBool .
eq isReg(empTable,K2) = false .
eq isReg((K,V) | T,K2) = if K == K2 then {true} else {isReg(T,K2)} .

keys	values
…	…
k	*v*
…	…

If *k* is registered ⇒ true is returned

keys	values
…	…

otherwise ⇒ false is returned

Tables as Lists

op lookup : Table Elt.K -> Elt&Err.V .
op lookup : Table&Err Elt.K -> Elt&Err.V .
eq lookup(errTable,K2) = err.V .
eq lookup(empTable,K2) = err.V .
eq lookup((K,V) | T,K2) = if K == K2 then {V} else {lookup(T,K2)} .

keys	values
…	…
k	*v*
…	…

If *k* is registered

v is returned

keys	values
…	…

otherwise

an error is returned

Tables as Lists

```
op update : Table Elt.K Elt.V -> Table .
op update : Table&Err Elt.K Elt&Err.V -> Table&Err .
eq update(errTable,K2,VE2) = errTable .
eq update(TE,K2,err.V) = errTable .
eq update(empTable,K2,V2) = (K2,V2) | empTable .
eq update((K,V) | T,K2,V2)
  = if K == K2 then {(K,V2) | T} else {(K,V) | update(T,K2,V2)} .
```

keys	values
…	…
k_2	v
…	…

If k_2 is registered

keys	values
…	…
k_2	v_2
…	…

keys	values
…	…

otherwise

keys	values
…	…
k_2	v_2

Tables as Lists

```
op insert : Table Elt.K Elt.V -> Table&Err .
op insert : Table&Err Elt.K Elt&Err.V -> Table&Err .
eq insert(errTable,K2,VE) = errTable .
eq insert(TE,K2,err.V) = errTable .
eq insert(T,K2,V2) = if isReg(T,K2) then {errTable} else {(K2,V2) | T} .
```

keys	values
…	…
k_2	v
…	…

If k_2 is registered ⇒ errTable is returned

keys	values
…	…

otherwise ⇒

keys	values
k_2	v_2
…	…

Tables as Lists

```
op remove : Table Elt.K -> Table .
op remove : Table&Err Elt.K -> Table&Err .
eq remove(errTable,K2) = errTable .
eq remove(empTable,K2) = empTable .
eq remove((K,V) | T,K2)
   = if K == K2 then {T} else {(K,V) | remove(T,K2)} .
```

keys	values
…	…
k_2	v
…	…

If k_2 is registered

keys	values
…	…
…	…

keys	values
…	…

otherwise

keys	values
…	…

Tables as Lists

op delete : Table Elt.K -> Table&Err .
op delete : Table&Err Elt.K -> Table&Err .
eq delete(errTable,K2) = errTable .
eq delete(T,K2)
    = if isReg(T,K2) then {remove(T,K2)} else {errTable} .

keys	values
…	…
k_2	v
…	…

If k_2 is registered

keys	values
…	…
…	…

keys	values
…	…

otherwise

errTable is returned

Tables as Lists

How to use tables:

```
mod! STRING-ERR principal-sort String {
  pr(STRING)
  [String ErrString < String&Err]
  op errStr : -> ErrString {constr} .
  op if_then{_}else{_} : Bool String&Err String&Err -> String&Err .
  vars SE1 SE2 : String&Err .
  eq if true then {SE1} else {SE2} = SE1 .
  eq if false then {SE1} else {SE2} = SE2 .
}
```

Tables as Lists

```
view TRIV2QID from TRIV to QID {
  sort Elt -> Qid
}
```

The built-in module in which quoted IDs are defined

```
view TRIV-ERR-IF2STRING-ERR
       from TRIV-ERR-IF to STRING-ERR {
  sort Elt -> String,
  sort Err -> ErrString,
  sort Elt&Err -> String&Err,
  op err -> errStr,
  op (if_then{_}else{_}) -> (if_then{_}else{_}),
}
```

Mix-fix operators should be enclosed with (and) when they appears in views.

Tables as Lists

```
open TABLE(K <= TRIV2QID, V <= TRIV-ERR-IF2STRING-ERR) .
  op t : -> Table .
  eq t = update(update(singleton('java,"Java"),'obj,"OBJ3"),'c,"C") .
  red t .
  red isReg(t,'obj) .
  red isReg(t,'mk) .
  red lookup(t,'obj) .
  red lookup(t,'mk) .
  red update(t,'mk,"SML#") .
  red update(t,'obj,"CafeOBJ") .
  red insert(t,'mk,"SML#") .
  red insert(t,'obj,"CafeOBJ") .
  red remove(t,'mk) .
  red remove(t,'obj) .
  red delete(t,'mk) .
  red delete(t,'obj) .
close
```

Billing Program

We will develop a billing program as an application of tables. The billing program makes a bill based on a catalog and a shopping cart.

A catalog contains some information on items that can be ordered, associating each item ID to the item name and price. It is expressed as a table from the set of item IDs to the set of (name,price)-pairs. Item IDs, names and prices are expressed as quoted IDs, strings and natural numbers.

An example of catalogs expressed as a table:

```
('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | empTable
```

Billing Program

A shopping cart contains items to be ordered and their numbers, expressed as lists of (item IDs,natural number)-pairs.

An example of shopping carts expressed as such a list:

('o,4) | ('t,10) | ('o,6) | nil

Billing Program

A bill item consists of an item name, the number of the item to be ordered and the sub-total for this item, expressed as a triple of a string, a natural number and a natural number.

A bill item list is a list of bill items.

A bill is a pair of a bill item list and the total.

An example of bills expressed as such pairs:

$$(\overbrace{\underbrace{(\text{"orange"},10,1000)}_{\text{A bill item}} \mid (\text{"tomato"},10,900) \mid nil}^{\text{A bill item list}},\ \underbrace{1900}_{\text{The total}})$$

Billing Program

The billing program takes

('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | empTable

('o,4) | ('t,10) | ('o,6) | nil

and makes the bill

A bill item list

(("orange",10,1000) | ("tomato",10,900) | nil, 1900)

A bill item

The total

Billing Program

```
mod! TAG {
  pr(PAIR(STRING-ERR,NAT-ERR) * {sort Pair -> Tag} )
  [Tag ErrTag < Tag&Err]
  op errTag : -> ErrTag {constr} .
  op (_,_) : String&Err Nat&Err -> Tag&Err .
  op if_then{_}else{_} : Bool Tag&Err Tag&Err -> Tag&Err .
  var SE : String&Err .
  var NE : Nat&Err .
  vars TE1 TE2 : Tag&Err .
  -- (_,_)
  eq (errStr,NE) = errTag .
  eq (SE,errNat) = errTag .
  -- if_then{_}else{_}
  eq if true then {TE1} else {TE2} = TE1 .
  eq if false then {TE1} else {TE2} = TE2 .
}
```

```
('a,("apple",150)) |
('o,("orange",100)) |
('t,("tomato",90)) |
empTable
```

Billing Program

```
mod! CATALOG {
 pr(TABLE(K <= TRIV2QID, V <= TRIV-ERR-IF2TAG) * {
     sort Table -> Catalog,  sort EmpTable -> EmpCatalog,
     sort NeTable -> NeCatalog,  sort ErrTable -> ErrCatalog,
     sort Table&Err -> Catalog&Err,  op empTable -> empCatalog,
     op errTable -> errCatalog } )  }

view TRIV-ERR-IF2TAG from TRIV-ERR-IF to TAG {
 sort Elt -> Tag,  sort Err -> ErrTag,
 sort Elt&Err -> Tag&Err,  op err -> errTag,
 op (if_then{_}else{_}) -> (if_then{_}else{_})  }

open CATALOG .
 op cat : -> Catalog .
 eq cat = ('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | empCatalog .
 red cat .
close
```

empTable has been renamed to empCatalog.

Billing Program

```
mod! CART-ITEM {
  pr(PAIR(QID,NAT-ERR) * {sort Pair -> CItem})
  [CItem ErrCItem < CItem&Err]
  op errCItem : -> ErrCItem {constr} .
}
```

('o,4) | ('t,10) | ('o,6) | nil

```
view TRIV-ERR2CART-ITEM from TRIV-ERR to CART-ITEM {
  sort Elt -> CItem,
  sort Err -> ErrCItem,
  sort Elt&Err -> CItem&Err,
  op err -> errCItem,
}
```

Billing Program

```
mod! CART {  pr(GLIST-ERR(E <= TRIV-ERR2CART-ITEM) * {
                    sort List -> Cart,  sort Nil -> EmpCart,
                    sort NnList -> NeCart,  sort ErrList -> ErrCart,
                    sort List&Err -> Cart&Err,  op nil -> empCart,
                    op errList -> errCart } )
  op norm : Cart -> Cart .
  op mkCart : Cart CItem -> Cart .
  vars I I2 : Qid .  vars N N2 : Nat .  var C : Cart .
  -- norm
  eq norm(empCart) = empCart .
  eq norm((I,N) | C) = mkCart(norm(C),(I,N)) .
  -- mkCart
  eq mkCart(empCart,(I,N)) = (I,N) | empCart .
  eq mkCart((I2,N2) | C,(I,N))
     = if I == I2 then {(I,N + N2) | C} else {(I2,N2) | mkCart(C,(I,N))} .
}
```

Billing Program

nil has been renamed to empCart.

```
open CART .
  op c : -> Cart .
  eq c = ('o,4) | ('t,10) | ('o,6) | empCart .
  red norm(c) .
close
```

('o,10) | ('t,10) | nil is returned as the result.

A shopping cart may contain multiple pairs whose item IDs are the same. norm modifies a given shopping cart such that it contains at most one pair for each item ID, preserving the number of each item to be ordered.

Billing Program

Please see the Appendices.

```
mod! BILL-ITEM {  pr(TRIPLE(STRING-ERR,NAT-ERR,NAT-ERR)
                      * {sort Triple -> BItem})
  [BItem ErrBItem < BItem&Err]
  op errBItem : -> ErrBItem {constr} .
  op (_,_,_) : String&Err Nat&Err Nat&Err -> BItem&Err .
  var SE : String&Err .  vars NE1 NE2 : Nat&Err .
  eq (errStr,NE1,NE2) = errBItem .
  eq (SE,errNat,NE2) = errBItem .
  eq (SE,NE1,errNat) = errBItem .
}
```

(("orange",10,1000) | ("tomato",10,900) | nil, 1900)

```
view TRIV-ERR2BILL-ITEM from TRIV-ERR to BILL-ITEM {
  sort Elt -> BItem,
  sort Err -> ErrBItem,
  sort Elt&Err -> BItem&Err,
  op err -> errBItem  }
```

Billing Program

```
mod! BILIST principal-sort BIList {
  pr(GLIST-ERR(E <= TRIV-ERR2BILL-ITEM) * {
      sort List -> BIList,  sort Nil -> NilBIList,  sort NnList -> NnBIList,
      sort ErrList -> ErrBIList,  sort List&Err -> BIList&Err,
      op nil -> nilBIL,  op errList -> errBIL, } )
  op total : BIList -> Nat .
  op total : BIList&Err -> Nat&Err .
  var S : String .  vars N ST : Nat .  var BIL : BIList .
  eq total(errBIL) = errNat .
  eq total(nilBIL) = 0 .
  eq total((S,N,ST) | BIL) = ST + total(BIL) .
}
```

nil has been renamed to nilBIL.

(("orange",10,1000) | ("tomato",10,900) | nilBIL, 1900)

Billing Program

```
mod! BILL { pr(CATALOG)  pr(CART)
  pr(PAIR(BILIST,NAT-ERR) * {sort Pair -> Bill})
  [Bill ErrBill < Bill&Err]
  op errBill : -> ErrBill {constr} .
  op (_,_) : BIList&Err Nat&Err -> Bill&Err .
  op mkBill : Catalog Cart -> Bill&Err .
  op mkSubBill : BIList&Err -> Bill&Err .
  op mkBIL : Catalog Cart -> BIList&Err .
  op mkSubBIL : Tag&Err Catalog Cart Nat -> BIList&Err .
  var BILE : BIList&Err .  var BIL : BIList .  var NE : Nat&Err .
  var CAT : Catalog .  var I : Qid .  vars N P : Nat .
  var IN : String .  var C : Cart .
```

(("orange",10,1000) | ("tomato",10,900) | nilBIL, 1900)

Billing Program

('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | empTable

('o,4) | ('t,10) | ('o,6) | empCart

```
-- (_,_)
eq (errBIL,NE) = errBill .
eq (BILE,errNat) = errBill .
-- mkBill
eq mkBill(CAT,C) = mkSubBill(mkBIL(CAT,norm(C))) .
-- mkSubBill
eq mkSubBill(errBIL) = errBill .
eq mkSubBill(BIL) = (BIL,total(BIL)) .
-- mkBIL
eq mkBIL(CAT,empCart) = nilBIL .
eq mkBIL(CAT,(I,N) | C) = mkSubBIL(lookup(CAT,I),CAT,C,N) .
-- mkSubBIL
eq mkSubBIL(errTag,CAT,C,N) = errBIL .
eq mkSubBIL((IN,P),CAT,C,N) = (IN,N,N * P) | mkBIL(CAT,C) .
}
```

mkBIL

(("orange",10,1000) | ("tomato",10,900) | nilBIL, 1900)

mkBill

("orange",10,1000) | ("tomato",10,900) | nilBIL

Billing Program

```
open BILL .
  op cat : -> Catalog .
  eq cat = ('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) | empCatalog .
  op c : -> Cart .
  eq c = ('o,4) | ('t,10) | ('o,6) | empCart .
  red mkBill(cat,c) .
close                 (("orange",10,1000) | ("tomato",10,900) | nilBIL, 1900)

open BILL .
  op cat : -> Catalog .
  ops c1 c2 : -> Cart .
  eq cat = ('a,("apple",150)) | ('o,("orange",100)) | ('t,("tomato",90)) |
          ('b,("banana",140)) | ('p,("potato",30)) | empCatalog .
  eq c1 = ('p,3) | ('o,2) | ('a,3) | ('p,10) | ('b,10) | ('o,10) | ('t,20) | empCart .
  eq c2 = ('p,3) | ('o,2) | ('f,10) | ('a,3) | empCart .
  red mkBill(cat,c1) .
  red mkBill(cat,c2) .
close
```

Exercises

1. Write all programs in the slides including the appendices and feed them into the CafeOBJ system. Moreover, write some more test code and do some more testing for the programs.

2. Revise the programs used in the slides such that binary-search trees are used instead of lists.

3. Write a program of balanced binary search trees based on the following:
Arne Andersson: Balanced Search Trees Made Simple. WADS 1993: 60-71
https://link.springer.com/chapter/10.1007/3-540-57155-8_236

Exercises

4. Write a program that finds the shortest path on a directed weighted graph based on the Dijkstra shortest path finding algorithm.

5. Write a program that finds the shortest path on a directed weighted graph based on A*, an extended version of the Dijkstra shortest path finding algorithm.

6. Write a program that finds the shortest path on a directed weighted graph based on LPA* (Lifelong Planning A*), an incremental version of A*. See the following:
Sven Koenig, Maxim Likhachev: D*Lite. AAAI/IAAI 2002: 476-483
https://www.aaai.org/Library/AAAI/2002/aaai02-072.php

Exercises

7. Implement all data structures and algorithms you have learned so far in your life.

Appendices

```
mod! PAIR(FE :: TRIV, SE :: TRIV) {
  [Pair]
  op (_,_) : Elt.FE Elt.SE -> Pair {constr} .
}

mod! TRIPLE(FE :: TRIV, SE :: TRIV, TE :: TRIV) {
  [Triple]
  op (_,_,_) : Elt.FE Elt.SE Elt.TE -> Triple {constr} .
}

 mod* TRIV-ERR-IF {
   [Elt Err < Elt&Err]
   op err : -> Err .
   op if_then{_}else{_} : Bool Elt&Err Elt&Err -> Elt&Err .
 }
```

Appendices

```
mod! GLIST-ERR(E :: TRIV-ERR) {
  [Nil NnList < List]
  [List ErrList < List&Err]
  op errList : -> ErrList {constr} .
  op nil : -> Nil {constr} .
  op _|_ : Elt.E List -> List {constr} .
  op _|_ : Elt&Err.E List&Err -> List&Err .
  op _@_ : List List -> List .
  op _@_ : List&Err List&Err -> List&Err .
  op if_then{_}else{_}
     : Bool List&Err List&Err -> List&Err .
  var X : Elt.E .
  var XE : Elt&Err.E .
  vars L L2 : List .
  vars LE LE2 : List&Err .
  -- _|_
  eq err.E | LE = errList .
  eq XE | errList = errList .
  -- _@_
  eq nil @ L2 = L2 .
  eq (X | L) @ L2 = X | (L @ L2) .
  eq errList @ LE = errList .
  eq LE @ errList = errList .
  -- if_then{_}else{_}
  eq if true then {LE} else {LE2} = LE .
  eq if false then {LE} else {LE2} = LE2 .
}
```

Appendices

```
mod! BOOL-ERR {
  [Bool ErrBool < Bool&Err]
  op errBool : -> ErrBool {constr} .
  op if_then{_}else{_} : Bool Bool Bool -> Bool .
  vars B1 B2 : Bool .
  -- if_then{_}else{_}
  eq if true then {B1} else {B2} = B1 .
  eq if false then {B1} else {B2} = B2 .
}
```

Appendices

```
mod! NAT-ERR principal-sort Nat {
  pr(NAT)
  [Nat ErrNat < Nat&Err]
  op errNat : -> ErrNat {constr} .
  op _*_ : Nat&Err Nat&Err -> Nat&Err .
  op if_then{_}else{_} : Bool Nat&Err Nat&Err -> Nat&Err .
  vars NE NE1 NE2 : Nat&Err .
  eq errNat * NE = errNat .
  eq NE * errNat = errNat .
  eq if true then {NE1} else {NE2} = NE1 .
  eq if false then {NE1} else {NE2} = NE2 .
}
```