Computer Science 131
Programming Languages

September 5, 2000

Examples, Examples, Examples

Environments

- Also known as:
  - Lookup tables
  - Finite mappings
  - Dictionaries
- Here: associating keys with values
  - Today, keys are always strings

Specification

```
type key = string
type 'a env

val empty  : 'a env
val insert : 'a env * key * 'a -> 'a env
val lookup : 'a env * key      -> 'a option
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list
```

For example,

```
[("x",3), ("y",7), ("z",2)]  :  int env
[("x",3.0), ("x",4.0)]       :  real env
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

val empty : 'a env = ...
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

val empty : 'a env = []
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

val empty : 'a env = []
fun insert (env,k,v) = ...
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

val empty : 'a env = []
fun insert (env,k,v) = (k,v) :: env
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

fun lookup(env,k) = ...
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

fun lookup([],k) = ...
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

fun lookup([],k) = NONE
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

fun lookup([],k) = NONE
  | lookup((k',v)::rest, k) =
        ...
```

Implementation 1: Association Lists

```
type key = string
type 'a env = (string * 'a) list

fun lookup([],k) = NONE
  | lookup((k',v)::rest, k) =
       if (k' = k) then
          SOME v
       else
          lookup(rest,k)
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = ...
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

val empty : 'a env = ...
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

val empty : 'a env = (fn _ => NONE)
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun empty _ = NONE
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun empty _ = NONE
fun lookup (env,k) = ...
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun empty _ = NONE
fun lookup (env,k) = env(k)
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun insert (env,k,v) =
        ...
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun insert (env,k,v) =
        (fn k' => ...)
```

Implementation 2: Partial Functions

```
type key = string
type 'a env = key -> 'a option

fun insert (env,k,v) =
       (fn k' => if (k=k') then
                          SOME v
                 else
                          env(k'))
```

Expressions

```
type var = string
datatype exp = Num of real
             | Var of var
             | Plus of exp * exp
             | Times of exp * exp
```

Evaluation Specification

```
type var = string
datatype exp = Num of real
             | Var of var
             | Plus of exp * exp
             | Times of exp * exp

val eval : exp * real env -> real
```

Evaluation Example

```
val e = Times(Num 2.0,
              Plus(Var "x", Var "y"))
val env = empty
val env = insert(env,"x",3.0)
val env = insert(env,"y",4.0)
```

Then `eval(e,env)` evaluates to ...

Evaluation Example

```
val e = Times(Num 2.0,
              Plus(Var "x", Var "y"))
val env = empty
val env = insert(env,"x",3.0)
val env = insert(env,"y",4.0)
```

Then `eval(e,env)` evaluates to 14.0

Expressions

```
fun eval(Num r, env) = ...
```

Expressions

```
fun eval(Num r, env) = r
```

Expressions

```
fun eval(Num r, env) = r
  | eval(Var v, env) = ...
```

Expressions

```
fun eval(Num r, env) = r
  | eval(Var v, env) =
      (case lookup(env,v) of
         NONE => 0.0  (* or exception *)
       | SOME r => r)
```

Expressions

```
fun eval(Num r, env) = r
  | eval(Var v, env) =
      (case lookup(env,v) of
         NONE => 0.0  (* or exception *)
       | SOME r => r)
  | eval(Plus(e1,e2),env) =
      ...
```

Expressions

```
fun eval(Num r, env) = r
  | eval(Var v, env) =
      (case lookup(env,v) of
         NONE => 0.0  (* or exception *)
       | SOME r => r)
  | eval(Plus(e1,e2),env) =
       eval(e1,env) + eval(e2,env)
```

Expressions

```
fun eval(Num r, env) = r
  | eval(Var v, env) =
      (case lookup(env,v) of
         NONE => 0.0  (* or exception *)
       | SOME r => r)
  | eval(Plus(e1,e2),env) =
       eval(e1,env) + eval(e2,env)
  | eval(Times(e1,e2),env) =
       eval(e1,env) * eval(e2,env)
```

Expressions

- To think about: how would you write a (partial) differentiation function?

```
val diff : exp * var -> exp
```

Regular Expressions

- Descriptions of (regular) sets of strings
  - We say "regexp **`r`** accepts a string **`s`**" if **`s`** is in the set described by **`r`**.

`[SEE HANDOUT]`

Regular Expressions Specification

```
datatype regexp = Zero  (* Never accepts *)
                | One   (* Accepts empty string *)
                | Char of char
                | Plus  of regexp * regexp
                | Times of regexp * regexp
                | Star  of regexp

val accept : regexp * string -> bool
```

Attempt 1

```
val acc : regexp * char list -> bool

fun accept(r,s) = acc(r, String.explode s)
```

Attempt 1

```
fun acc(Zero,cs) = ...
```

Attempt 1

```
fun acc(Zero,cs) = false
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = ...
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = ...
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = (c = d)
  | acc(Char d, _) = false
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = (c = d)
  | acc(Char d, _) = false
  | acc(Plus(r1,r2), cs) =
        ...
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = (c = d)
  | acc(Char d, _) = false
  | acc(Plus(r1,r2), cs) =
       acc(r1,cs)  orelse  acc(r2,cs)
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = (c = d)
  | acc(Char d, _) = false
  | acc(Plus(r1,r2), cs) =
       acc(r1,cs)  orelse  acc(r2,cs)
  | acc(Times(r1,r2), cs) = ...
```

Attempt 1

```
fun acc(Zero,cs) = false
  | acc(One,cs) = (cs = [])
  | acc(Char d, [c]) = (c = d)
  | acc(Char d, _) = false
  | acc(Plus(r1,r2), cs) =
       acc(r1,cs)  orelse  acc(r2,cs)
  | acc(Times(r1,r2), cs) = (* OOPS *)
```

Attempt 2

```
val acc : regexp * char list ->
              (char list) option

fun accept(r,s) =
      (case (acc(r, String.explode s)) of
          NONE => false
        | SOME cs => (cs = []))
```

Attempt 2

```
fun acc(Zero,cs) = ...
```

Attempt 2

```
fun acc(Zero,cs) = NONE
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | acc(One,cs)  = ...
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | acc(One,cs)  = SOME cs
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | acc(One,cs)  = SOME cs
  | acc(Char d, []) = NONE
  | acc(Char d, (c::cs)) =
     if (c=d) then SOME cs else NONE
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | acc(One,cs)  = SOME cs
  | acc(Char d, []) = NONE
  | acc(Char d, (c::cs)) =
     if (c=d) then SOME cs else NONE
  | acc(Times(r1,r2), cs) =
     ...
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | acc(One,cs)  = SOME cs
  | acc(Char d, []) = NONE
  | acc(Char d, (c::cs)) =
     if (c=d) then SOME cs else NONE
  | acc(Times(r1,r2), cs) =
     (case acc(r1,cs) of
        NONE => NONE
        SOME cs' => acc(r2,cs'))
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | ...
    (* Is this right? *)
  | acc(Plus(r1,r2), cs) =
     (case acc(r1,cs) of
        NONE => acc(r2,cs)
        SOME cs' => SOME cs')
```

Attempt 2

```
fun acc(Zero,cs) = NONE
  | ...
    (* Is this right? *)
  | acc(Plus(r1,r2), cs) =
     (case acc(r1,cs) of
        NONE => acc(r2,cs)
        SOME cs' => SOME cs')

    (* OOPS *)
```

Attempt 3

```
(* Specification: acc(r,cs,k) returns
   true if r matches some prefix of cs, and
   k applied to the remainder of the list = true *)
val acc : regexp * char list *
             (char list -> bool) -> bool

fun isNil [] = true
  | isNil _ = false

fun accept(r,s) =
      acc(r, String.explode s, isNil)
```

Attempt 3

```
fun acc(Zero, cs, k) = false
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = ...
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = k cs
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = k cs
  | acc(Char d, [], k) = false
  | acc(Char d, c::cs, k) = ...
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = k cs
  | acc(Char d, [], k) = false
  | acc(Char d, c::cs, k) =
       if (c=d) then (k cs) else false
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = k cs
  | acc(Char d, [], k) = false
  | acc(Char d, c::cs, k) =
      if (c=d) then (k cs) else false
  | acc(Plus(r1,r2), cs, k) =
      ...
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | acc(One, cs, k) = k cs
  | acc(Char d, [], k) = false
  | acc(Char d, c::cs, k) =
       if (c=d) then (k cs) else false
  | acc(Plus(r1,r2), cs, k) =
       acc(r1,cs,k) orelse acc(r2,cs,k)
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | ...
  | acc(Times(r1,r2), cs, k) =
      ...
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | ...
  | acc(Times(r1,r2), cs, k) =
      acc(r1, cs,
          fn cs' => acc(r2, cs', k))
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | ...
  | acc(Times(r1,r2), cs, k) =
      acc(r1, cs,
          fn cs' => acc(r2, cs, k))
  | acc(Star r1, cs, k) =
      ...
```

Attempt 3

```
fun acc(Zero, cs, k) = false
  | ...
  | acc(Times(r1,r2), cs, k) =
      acc(r1, cs,
          fn cs' => acc(r2, cs, k))
  | acc(Star r1, cs, k) =
      (k cs) orelse
      (acc(Times(r1,Star r1), cs, k)
```

Attempt 3

There is still a subtle bug here...see handout.

```
| acc(Star r1, cs, k) =
      (k cs) orelse
      (acc(Times(r1,Star r1), cs, k)
```