

Strings: a Programming Example

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

`bjorn.lisper@mdh.se`
`http://www.idt.mdh.se/~blr/`

String Processing

String (or text) processing is important

Conversions between different formats: files, documents, XML, web/database, etc.

I think functional programming is good for this kind of application

We will look at a simple example here: how to break a text into a list of words, that can be used for various things like:

- counting the number of words in the text
- printing the text with a given maximal line length in characters (breaking lines when next word does not fit in)

Strings

F# has a data type `string` for strings

We will *not* use this type for now

Rather, we will use lists of characters, of type `char list`

One reason: we then get a good exercise in list programming

Later, we'll bring up the `string` datatype

We will then redo the example using strings rather than lists of characters

Breaking a String Into Words

Words are sequences of characters separated by one or more *whitespace* characters: space, newline, tab

(In F#: ' ', '\n', '\t')

We want a function that converts a list of characters into a list of its words.
Words are also lists of characters

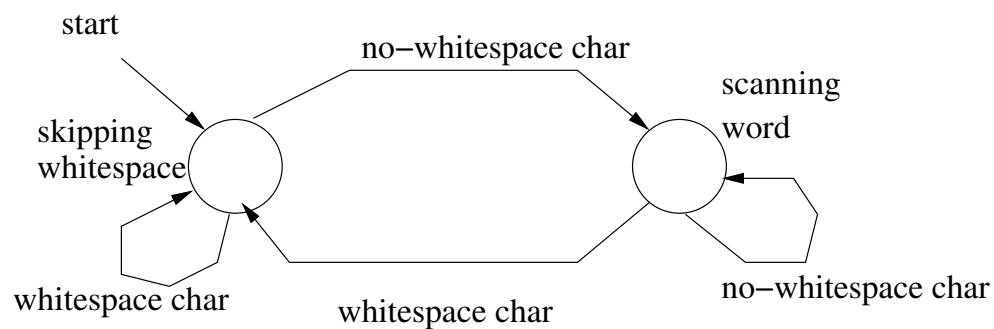
```
string2words : char list -> (char list) list
```

For instance,

```
string2words ['A';'l';'l';'a';'n';' ';'\t';'a';'r';' ';'\n';'k';'a';'k';'a';'n']  
=>  
[['A';'l';'l';'a';'n'];['t';'a';'r'];['k';'a';'k';'a';'n']]
```

How code `string2words?`

We need a mental model. This is a simple parsing problem, which can be solved by a *finite automaton* with two states:



Common design pattern: one function per state. When new character read the function for the new state is called

We'll use a variation of this pattern: in each state we will *look ahead* and count the number of characters before changing to the other state:

- **whitespace**: count characters until non-whitespace char, then drop that number of characters and call the other function on rest of list
- **word**: count characters until whitespace char, then save that number of characters into list of characters and call the other function on rest of list

We can define a general list function `drop` to skip a number of characters:

```
drop 3 [1;4;2;5;6]  $\implies$  [5;6]
```

(`drop n s` returns the list remaining after `take n s`)

Exercise: define `drop!` (A solution on next slide)

```
let rec drop n l =
  if n < 0 then
    failwith "Negative argument"
  else
    match (n,l) with
    | (0,_)      -> l
    | (_,x::xs) -> drop (n-1) xs
    | (n,[])    -> failwith "List too short"
```

(This function is a little inefficient. Why?)

A First Solution

Functions to count characters until next whitespace and next no-whitespace, respectively:

```
let rec find_ws l =
  match l with
  | []      -> 0
  | c::cs  -> if c = ' ' || c = '\n' || c = '\t'
              then 0 else 1 + find_ws cs
```

```
let rec find_nows l =
  match l with
  | []      -> 0
  | c::cs  -> if c <> ' ' && c <> '\n' && c <> '\t'
              then 0 else 1 + find_nows cs
```

Functions `string2words` and `string2words1` corresponding to states “skipping whitespace” and “scanning word”, respectively:

```
let rec string2words s =
  match s with
  | [] -> []
  | _ -> string2words1 (drop (find_nows s) s)
and string2words1 s =
  match s with
  | [] -> []
  | _ -> let n = (find_ws s)
          in take n s :: string2words (drop n s)
```

This is a *mutually recursive* definition. The functions recursively call each other

The keyword “`and`” is used to link mutually recursive declarations (why would it not work with ordinary “`let rec`” for the second declaration?)

Note how the words are collected into separate lists by `take`

Also note that “`::`” in `string2words1` puts the list of characters as *element* into the list, so the returned list is a list of *lists of characters* (not list of characters)

A More Elegant Slution

This solution works fine, but is a bit clumsy

In particular, `find_ws` and `find_nows` are very similar

They do precisely the same, but with negated conditions!

Can we “factor out” the common structure?

Yes, if we can make the condition a *parameter* to a more general function!

Let's see on next slide how to do this . . .

A More General Character Count Function

F# has *higher order functions*

They are functions that take other functions as arguments, or return functions as result

We can thus define a function `find` that takes a *predicate* `p` on characters as first arguments and counts the number of characters up to the first character `c` such that `p c = true`:

```
let rec find p l =  
    match l with  
    | []      -> 0  
    | x::xs  -> if p x then 0 else 1 + find p xs  
find : (char -> bool) -> (char list) -> int
```

(`find` will actually have a more “general” type. More on this later)

Predicate to check for whitespace:

```
let ws c =  
  match c with  
  | ' '   -> true  
  | '\n' -> true  
  | '\t' -> true  
  | _     -> false  
ws : char -> bool
```

Then simply:

```
let find_ws s = find ws s
```

For `find_nows`, we must have a negated whitespace-predicate:

```
let not_ws c = not (ws c)
```

We get:

```
let find_nows s = find not_ws s
```

(A more elegant solution, avoiding these declarations, would be to use nameless functions but we haven't introduced them yet)

Final Solution

```
module String2words
  let ws c =
    match c with
    | ' ' -> true
    | '\n' -> true
    | '\t' -> true
    | _ -> false

  let not_ws c = not (ws c)

  let rec find p l =
    match l with
    | [] -> 0
    | x::xs -> if p x then 0 else 1 + find p xs

  let find_ws s = find ws s

  let find_nows s = find (not_ws) s
```

Final Solution, Part 2

```
let rec string2words s =  
  match s with  
  | [] -> []  
  | _   -> string2words1 (drop (find_nows s) s)  
and   string2words1 s =  
  match s with  
  | [] -> []  
  | _   -> let n = (find_ws s)  
            in take n s :: string2words (drop n s)
```

Applications of `string2words`

Let's do the two applications mentioned before:

- counting the number of words in the text
- printing the text with a given maximal line length in characters (breaking lines when next word does not fit in)

Can you figure out how to do them?

How to do them

The first is easy: use the `List.length` function from the `List` module

```
let wordcount s = List.length (string2words s)
```

The second is more interesting ...

A function `words2lines` `linelen` `ws`, where `linelen` is the line length and `ws` is a list of words to be printed

Idea: keep a current position on the line, check length of next word, if greater than `linelen` then start new line else output word on current line and update position

Current position passed as argument

Local function to do this, so `words2lines` does not need to have this extra argument

We will use the *append* (or *concatenate*) operation “@” on lists:

$$[1;2;3] @ [4;2] \implies [1;2;3;4;2]$$

The Solution

```
let words2lines linelen ws =
  let rec
    w2l l pos =
      match l with
      | []      -> []
      | w::ws  -> if pos + List.length w < linelen
                  then w @ [' '] @ w2l ws (pos + List.length w + 1)
                  else '\n' :: w @ [' '] @ w2l ws (List.length w + 1)
  in w2l ws 0
```

Not perfect. Leaves space at end of each line. Somewhat poor treatment of words longer than line length – always new line even if the long word is first in list

Exercise: write a new solution that handles these cases better

Strings

`string` is a one of the builtin datatypes in F#

Strings are really a kind of immutable arrays, holding characters

There is a `String` module with operations on strings

Basic syntax for string constants: a string of characters inside `" . . . "`:

```
"abc is bcd"
```

Familiar syntax for control characters: `\n` (newline), `\t` (tab), `\\` (backslash), etc.

```
"Line 1\nSecond line"
```

Empty string: `" "`

Operations on Strings

Concatenation, or append: +

"abc" + "xyz" \implies "abcxyz"

Concatenation requires copying (potentially expensive). Use with care!

Selection of character from strings is done by indexing, `s[i]`. String indices start from 0

"abc".[0] \implies 'a', "abc".[1] \implies 'b', "abc".[2] \implies 'c'

Note that a character is returned, not a string

More Operations on Strings

Selection of substring, `s.[i..j]`:

`"abc".[0..1] ⇒ "ab"`

Also `s.[i..]` (all elements in `s` from `i` and up), `s[..i]` (all elements in `s` up to `i`):

`"abc".[1..] ⇒ "bc"`, `"abc".[..1] ⇒ "ab"`

Note that here a string is returned, not a character

Selection of substring requires copying. Use with care!

Length of string: `String.length`

`String.length "abc" ⇒ 3`

Even More Operations on Strings

Some operations on strings use “dot” notation (object method style), some examples:

```
s.Length (same as String.length s)
```

```
s.ToUpper(), s.ToLower()
```

```
"abc".ToUpper()  $\implies$  "ABC"
```

```
"AbC".ToLower()  $\implies$  "abc"
```

This syntax is really quite alien to functional languages, but is present in F# due to its connections to the .NET with its object-oriented nature

If we don't like it, we can easily define wrapper functions to hide it:

```
let toupper s = s.ToUpper()
```

String Programming Example Revisited

We now redo the example of breaking a string into words, with strings

It turns out we can still keep the same design

Our function `string2words` will map from `string` to `string list`

A Straightforward Solution

A straightforward scheme to convert from lists to strings:

1. Replace pattern-matching on lists with calls to `List.head`, `List.tail`, `List.isEmpty`
2. Define versions of these functions that work on strings. Do the same for `take` and `drop`
3. Replace each list function with its corresponding string function

Replacing Pattern-Matching with Function Calls

```
match l with  
| [] -> ...  
| x::xs -> ...x...xs...
```

=>

```
if List.isEmpty l  
  then ...  
  else ...List.head l...List.tail l...
```

String Functions

Declarations of string counterparts for `head`, `tail`, etc:

```
let s_head s = s.[0]
let s_tail s = s.[1..]
let s_take n s = s[..n-1]
let s_drop n s = s.[n..]
let s_isempty s = s = ""
```

However, using them yields a problem!

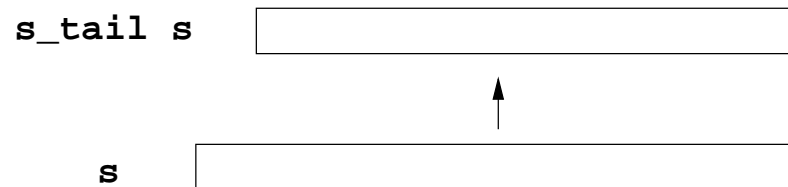
Can you see what the problem is?

String Functions (II)

Some of them are *terribly inefficient*

Substring selection requires copying of the selected substring

`s_tail` is the worst culprit. Almost all of the string argument must be copied to yield its tail. This is especially devastating when the tail is taken recursively



A More Efficient Solution

Rather than using substring selection, we can add a number for the input string that keeps track of the current position

It will act as a pointer to the position where the current substring starts



The substring of string `s` at position `pos` is represented by the pair (pos, s)

Versions of `head`, `tail`, etc. for this representation on next page

Functions for the Efficient Representation

```
type pstring = int*string // alias for pair of int and string

let ps_head (pos,s) = s.[pos]
let ps_tail (pos,s) = (pos+1,s)
let ps_take n (pos,s) = s.[pos..pos+n-1] // returns string
let ps_drop n (pos,s) = (pos + n,s)
let ps_isempty (pos,s) = pos >= String.length s
```

(`ps_take` returns a `string` rather than a `pstring`. For our purposes this is appropriate. A version that returns a `pstring` is perfectly possible, but would require `pstrings` with two pointers rather than one.)

New declarations for our string programming solution, using these functions, on next page

```
let rec find p ps =
  if ps_isempty ps
  then 0
  else if p (ps_head ps) then 0 else 1 + find p (ps_tail ps)

let find_ws ps = find ws ps
let find_nows ps = find (not_ws) ps

let rec string2words ps =
  if ps_isempty ps
  then []
  else string2words1 (ps_drop (find_nows ps) ps)
and string2words1 ps =
  if ps_isempty ps
  then []
  else let n = find_ws ps
        in ps_take n s :: string2words (ps_drop n s)
```