

## A completely different look at abstraction

Rewriting the line-breaking program in ML

## The purpose of this lecture

- Examine an ML version of the line-breaking program
  - ML is a mostly-functional language that is dramatically different from C++
  - The program is therefore going to be very different as well
- Understand how the same abstractions can work in very different languages

## What is ML?

- ML stands for “MetaLanguage”
- Originally part of a theorem-proving system, but outgrew its beginnings
- Started about the same time as BCPL (the precursor to C), literally next door
- Presently the work of an informal collaboration between Bell Labs and several universities (including Princeton)

## Properties of ML

- Mostly functional
  - Functions are first-class values
  - Mutable objects legal, but discouraged
- High level, semantically safe
  - All memory is garbage collected
  - All operations are checked for validity
- Strongly typed, with compile-time type inference

## The point of this program

- We are going to try to implement abstractions in ML that are similar to the ones we used in C++
- If we were setting out from scratch to write this program in ML, we would have used different abstractions
- We will leave many details of ML unexplained

## Meta-points

- Abstractions can mostly transcend any one language ...
- ... but total language independence is hard ...
- ... and independence from language and environment is even harder

## Design strategy

- The program you are about to see is an ML implementation of a design that was originally intended for C++
- If we were setting out to write this program from scratch in ML, we would probably design it differently
- We intend to stick with our original C++ abstractions to the extent possible

## Defining the abstractions

- ML has what is called a *signature*, which is a formal way of expressing an interface to a family of types and functions
- We will define a signature that corresponds to each of our `Token` and `Line` classes
- After that, we will implement them

## The TOKEN signature

```
signature TOKEN =
sig
  datatype Toktype =
    WORD of string | BREAK | END
  val construct: TextIO.instream -> Toktype
end
```

The value of `construct` is a function whose argument is a `TextIO.instream` and whose result is a `Toktype`

A `Toktype` value is either a `WORD` (in which case it contains a `string`), or a `BREAK` or an `END` (in which case it contains no additional data).

## The LINE signature

```
signature LINE =
sig
  type T;
  val construct: int -> T
  val reset: T -> T
  val canfit: T * string -> bool
  val append: T * string -> T
  val print: T * TextIO.outstream -> unit
end
```

- Note that we haven't said anything about what the type `T` is yet: That's part of the implementation.

## Implementing a signature

- We will implement the `Token` signature by writing

```
structure token : TOKEN =
struct
  (* definitions will go here *)
end
```
- Every name that matches the signature will be type-checked against it
- Every name that doesn't will be hidden

## Defining `Token.Toktype`

- ```
datatype Toktype = WORD of string | BREAK | END
```
- This definition must match the one in the signature
    - We defined the details of `Toktype` in the signature because it is part of the interface
    - We have to define it again in the structure because we define everything in the structure; the compiler verifies the definitions but doesn't invent them

## Checking for white space

- ML doesn't have a built-in `isspace` function, so we must write our own

```
fun isspace("#" ") = true
  | isspace("#"\n") = true
  | isspace("#"\t") = true
  | isspace(_) = false
```

- Writing `#"c"` in ML is analogous to writing `'c'` in C or C++

## Using | in definitions

- The usual way of defining functions in ML is to give a number of alternatives, where the first ones are often constants
- The alternatives are tested in order
- These tests, and recursion, are the main control structures:

```
fun fact(0) = 1
  | fact(n) = n * fact(n-1)
```

## Counting newlines

- Our first job in constructing a Token will be to read white space, counting newlines, to see if we have a paragraph break
- As in C++, we must avoid reading too far
- ML uses a slightly different abstraction

## Reading ahead

- The ML I/O library doesn't let you put characters back in the input
- Instead, it lets you peek ahead in the input to see what the next character is (and whether you're at end of file)

## The countn1 function

```
fun countn1(strm, cnt) =
  case TextIO.lookahead(strm) of
  NONE => cnt
  | SOME(c) =>
    if isspace(c)
    then (TextIO.input1(strm);
          countn1(strm, if c = #"\n"
                        then cnt+1
                        else cnt))
    else cnt
```

- We will pass an initial value for the counter as a parameter when we call it

## The readword function

- This function assumes that the very next character in the input is nonblank

```
fun readword(strm) =
  case TextIO.lookahead(strm) of
  NONE => ""
  | SOME(c) =>
    if isspace(c)
    then ""
    else (TextIO.input1(strm);
          Char.toString(c)^readword(strm))
```

## The construct function

- There's nothing special about this function in ML, but we've given it a name that suggests its purpose

```
fun construct(strm) =
  if countnl(strm, 0) >= 2
  then BREAK
  else if TextIO.lookahead(strm) = NONE
       then END
       else WORD(readword(strm))
```

## The Line structure

- You'll be happy to know that the functions in this one are simpler
- The outline:

```
structure Line : LINE =
  struct
    (* definitions go here *)
  end
```

## The type Line.T

- As in C++, we will use a string and an int to represent a line
  - ML gives us a way to define simple structures as ordered pairs (or triples, etc.) without having to make up names
- ```
type T = string * int
```

## construct, reset, and append

- Note that these functions take a Line as input and yield a new Line as output (with the same maximum width)

```
fun construct(n) = ("", n)
fun reset(s, n) = ("", n)
fun append((s, n), s') =
  (if s = ""
   then s'
   else s ^ " " ^ s',
   n)
```

## canfit and print

```
fun canfit((s, n), s') =
  if s = ""
  then size(s') <= n
  else size(s) + size(s') + 1 <= n
fun print((s, n), strm) =
  if s = ""
  then ()
  else TextIO.output(strm, s ^ "\n")
```

## Now for the reformat function

- We want to avoid using a mutable variable
- As with the countnl function, we will do so by passing the value of the variable as an argument in a recursive call
- We will therefore make reformat call an auxiliary function

## The top-level definition of reformat

```
fun reformat(istrm, ostrm, n) =  
  let fun f(l) = (* definition of f *)  
        in f(Line.construct(n))  
    end
```

## The definition of f

```
fun f(l) =  
  case Token.construct(istrm) of  
    Token.BREAK =>  
      (Line.print(l, ostrm);  
       TextIO.output(ostrm, "\n");  
       f(Line.reset(l)))  
  | Token.WORD(w) =>  
      if Line.canfit(l, w)  
      then f(Line.append(l, w))  
      else (Line.print(l, ostrm);  
           f(Line.append(Line.reset(l), w)))  
  | Token.END =>  
      Line.print(l, ostrm)
```

## How do we execute it?

- For example:  
    reformat(TextIO.openIn("myfile"),  
              TextIO.stdout, 60)

## So what's the point?

- Most of the design translated right into ML, even though the languages are so different
- Although the design translated easily, the implementation did not
- Moreover, one seemingly small difference made a large difference in the program

## The biggest little difference

- C and C++ share the notion of reading an extra character from the input and putting it back if you decided you didn't like it
- ML lets you peek ahead one character without reading it

## Why was this difference important?

- If you can't put back a character after reading it, you must save it somewhere that will let the rest of your program get at it
- In the ML version, that would require passing "the next character" from each function to the next
- So we had to use lookahead instead

## Another systematic difference

- Although ML does allow mutable values, using them in a program such as this one would go against the spirit of ML
- Instead, we introduced extra arguments to `countn1` and `reformat` so that we could pass the state explicitly from one iteration to the next

## What about I/O?

- Input and output are the only side effects in this program
- We could have rewritten the program to avoid side effects altogether, but it would have looked much different
- Changing the shape of your foundation often changes what you build on it

## The moral of the story

- A clean, abstract design can transcend any one language
- However, it is hard to avoid depending on the environment altogether
- Sometimes a small change in the environment can mean a large change in the program