CS109B ML Notes for the Week of 5/8/95

Structures

ML's way of encapsulating concepts such as a data structure and functions that operate on it.

- Keyword `structure`, the structure name, and an = sign begin a structure definition.
- The structure itself is defined between `struct...end` and consists of definitions such as `val`, `fun`, or `exception`.

Example: Stacks with operations `push`, `pop`, `top`, and `create`; the last takes an element and returns a stack of that element alone.

- In our selected implementation, a stack is a reference to a list.

Here is a structure for stacks:

```
structure Stack = struct
    exception EmptyStack;

    val create(x) = ref [x];

    fun push(x,s) = s := x::(!s);

    fun top(ref nil) = raise EmptyStack
    |   top(ref(x::xs)) = x;

    fun pop(ref nil) = raise EmptyStack
    |   pop(s) = s := tl(!s);
end;
```

Using a Structure

Like `Array`, we need to open the structure, with

```
open Stack;
```

Then, we can use its operations to create and manipulate stacks, as:

```
val myStack = create(0);
push(1,myStack);
push(2,myStack);
top(myStack);
pop(myStack);
top(myStack);
```

Signatures

ML's description of a structure.

- The signature includes names and types of things, but not their values, much like the description of a function talks about what its input and output types are without addressing what the function *does*.

Example: Here is ML's response to that stack structure:

```
structure Stack :
    sig
        exception EmptyStack
        val create : '1a → '1a list ref
        val pop : 'a list ref → unit
        val push : 'a * 'a list ref → unit
        val top : 'a list ref → 'a
    end
```

Creating Your Own Signatures

- Use keyword `signature`, the name and an = sign.
- The signature itself comes between `sig`... `end`.

Example: Suppose we wanted a signature like that for structure `Stack`, but without

1. Without the function `top`.
2. With stacks restricted to integers.

We could use signature

```
signature NOTOP = sig
    exception EmptyStack;
    val create: int -> int list ref;
    val pop: int list ref -> unit;
    val push: int * int list ref -> unit;
end;
```

Information Hiding With Signatures

We can define a new structure that is like an old one, but with a different, more restrictive signature.

- You can't make functions appear by magic, but you can make them go away (i.e., become invisible to the user) and you can restrict types.

Example: We could define a structure `IntStack` which is a stack of integers only, and without the `top` function.

```
structure IntStack: NOTOP = Stack;
```

Why Information Hiding?

Modern software design emphasizes the use of "modules" or "classes" that give the user a well-defined, limited interface, usually functions to apply to some hidden data structure.

- The theory is that these modules are less likely to cause bugs if their data structure cannot be manipulated in a way that is a surprise to the designer.

- Information hiding lets the module implementer use his/her own functions that the user cannot access.

- Writing a structure with all needed functions and then giving the user only a subset is one way to achieve this goal.

Information Hiding by Local Elements

Instead of hiding elements by a special signature, we can define local elements of a structure by `local...in...end`.

- Definitions between `local` and `in` are only accessible to the definitions after `in`.

- Definitions after `in` are accessible to anyone who opens the structure.

  - ☐ Compare `local` with `let`: the difference is that `let` requires `in` to be followed by an evaluatable expression, not definitions.

```
open Array;

structure Random = struct
    local
        val register = array(10,0);

        fun feedback1(nil) = ()
        |   feedback1(x::xs) = (
            update(register,x,1-sub(register,x));
            feedback1(xs));

        fun feedback() = feedback1([0,2,4,7]);

        fun shift1(0) = update(register,0,0)
        |   shift1(i) = (
            update(register,i,sub(register,i-1));
            shift1(i-1));

        fun shift() = shift1(9);

        fun init1(0) = (
            update(register,9,1);
            update(register,0,0))
        |   init1(i) = (
            update(register,i,0);
            init1(i-1))
    in
        fun init() = init1(9);

        fun getBit() =
            let val bit = sub(register,9);
            in (
                shift();
                if bit=1 then feedback() else ();
                bit)
            end;
    end
end;
```

Fig. 1. Structure `Random`.

Example: Here is an example of some interest in its own right. A structure called `Random` in Fig. 1 offers the user two functions; `init` to initialize a random bit generator, and `getBit` to return the next random bit.

- The method is a *feedback shift register*, an array of bits (10 in this case).
  - ☐ To get the next bit, read the last (9th) and shift all bits up one.
  - ☐ Bring a 0 into the first (0th) position.
  - ☐ If the bit just generated is 1, complement certain fixed positions of the array. Which positions are complemented determines how good the random-number generator is.

Here is an example of how `Random` might be used to print 1000 random bits.

```
open Random;

fun random1(0) = print("\n")
|   random1(i) = (
        if i mod 72 = 0 then print("\n") else ();
        print(getBit());
        random1(i-1));

fun random(i) = (
        init();
        random1(i));

random(1000);
```

- Function `random1` counts down, calling `getBit` the appropriate number of times and printing a newline every 72 bits.
- Function `random` initializes the register to 0000000001 and then calls for 1000 bits.