FUNCTIONAL PROGRAMMING (1)  
PROF. SIMON PARSONS

- Imperative programming is concerned with “how”.
- Functional or applicative programming is, by contrast, concerned with “what”.
- It is based on the mathematics of the lambda calculus (Church as opposed to Turing).
- “Programming without variables”.
- It is inherently concise, elegant, and difficult to create subtle bugs in.

### Referential transparency

- The main (good) property of functional programming is referential transparency.
- Every expression denotes a single value.
- This value cannot be changed by evaluating an expression or by sharing it between different parts of the program.
- There can be no reference to global data.
- (Indeed there is no such thing as global data.)
- There are no side-effects, unlike in referentially opaque languages.

---

Program example (output)

```plaintext
program example(output)
var flag: boolean

function f(n:int): int
begin
if flag then f:= n
else f: 2*n
flag := not flag
end

begin
flag := true
writeln(f(1) + f(2))
writeln(f(2) + f(1))
end
```

- What is the output?

Okay, so the answer is 5 followed by 4.
- This is odd since if these were mathematical functions, 
\[ f(1) + f(2) = f(2) + f(1) \]
for any \( f \).
- But this is because mathematical functions are functions only of their inputs.
- They have no memory.
- We can always tell what the value of a mathematical function will be just from its inputs.

---

Simple functional programming in HOPE

- At the heart of the “problem” is fact that the global data `flag` controls the value of `f`.
- In particular the assignment:

```
flag := not flag
```

is the thing that gives this behaviour.
- If we eliminate assignment, we eliminate this kind of behaviour.
- Variables are no longer placeholders for values that change.
- (They are much less variable than variables in imperative programs).

- HOPE is strongly typed.
- Other functional languages aren’t typed (LISP for example).
- We call the function by:

```
square(3)
```
- Which evaluates to \( 3 \times 3 \) by definition, and then to 9 by the definition of \( \times \).
- Note only that, it will always evaluate to 9.
More complex functions:

\[ \text{dec max : num \# num \rightarrow num;} \]
\[ \quad \text{--- max(m, n) = if m > n then m else n;} \]

and:

\[ \text{dec max3 : num \# num \# num \rightarrow num} \]
\[ \quad \text{--- max3(a, b, c) = max(a, max(b, c));} \]

The type definitions indicate that the functions take two and three arguments respectively.

Saying that these functions take two and three arguments is slightly misleading. Instead they both have one argument—they are both tuples. One is a two-tuple and one is a three-tuple. This has one neat advantage—you can get functions to return a tuple, and thus several values.

\[ \text{dec IntDiv : num \# num \rightarrow num \# num;} \]
\[ \quad \text{--- IntDiv(m, n) = (m \div n, m \mod n);} \]

And we can compose \( \text{max(IntDiv(11, 4))} \), which will give 13.

Another function:

\[ \text{dec analyse : real \rightarrow char \# trueval \# num;} \]
\[ \quad \text{--- analyse(r) = (if r < 0 then \text{'-'} else \text{'}+' ,} \]
\[ \quad \quad \text{(r \geq -1.0) and (r \leq 1.0),} \]
\[ \quad \quad \text{round(r));} \]

Applying

\[ \text{analyse(-1.04)} \]
\[ \quad \text{will give ('-', false, -1)} \]

Note the overloading of >.

Recursion

Without variables, we can’t write functional programs with loops. So to get iteration, we need recursion.

\[ \text{dec sum : num \rightarrow num;} \]
\[ \quad \text{--- sum(n) = if n = 0 then 0 else sum(n - 1) + n;} \]

Which works in the same way as recursion normally does. Recursion fits in perfectly with the functional approach. Each application of the recursive function is referentially transparent and easy to establish the value of.

Here is a classic recursive function, with a twist. We can define functions to be infix.

\[ \text{infix} \hat{\text{;}} \]

Here is the power function as an infix function:

\[ \text{dec} \hat{\text{;}} : \text{num \# num \rightarrow num;} \]
\[ \quad \text{--- x \hat{\times} y = if y = 0 then 1 else x \times (y - 1);} \]

Again, HOPE gives us a very elegant way of defining the function.

Qualified expressions

Because we don’t have variables, sometime it seems we have to do unnecessary work when evaluating functions:

\[ \text{dec f : num \rightarrow num;} \]
\[ \quad \text{--- f(x) = g(square(max(x, 4))) +} \]
\[ \quad \quad \text{(if x \leq 1 then 1 else g(square(max(x, 4))));} \]

Here we have to evaluate \( g(square(max(x, 4))) \) twice in some situations.

With variables, of course, we would have to do this just once.

Once way around this would be to define the repeated bit as a new function:

\[ \text{dec f : num \rightarrow num;} \]
\[ \quad \text{--- f(x) = f1(g(square(max(x, 4))));} \]

\[ \text{dec f1 : num \rightarrow num;} \]
\[ \quad \text{--- f1(a, b) = a + (if b \leq 1 then 1 else a);} \]

Efficiency here relies on efficient evaluation in the language. Another way is to use qualified expressions.

Consider:

\[ \text{dec f : num \rightarrow num;} \]
\[ \quad \text{--- f(x) = let a = g(square(max(x, 4)))} \]
\[ \quad \quad \text{in a + (if x \leq 1 then 1 else a);} \]

The let construct allows us to extend the set of parameters of a function.

In general:

\[ \text{let <name> = <expression1> in <expression2>} \]

The first expression defines <name> and the second uses it.

We also have:

\[ <expression2> \text{ where <name> = <expression1>} \]

So we could also write:

\[ \text{dec f : num \rightarrow num;} \]
\[ \quad \text{--- f(x) = a + (if x \leq 1 then 1 else a)} \]
\[ \quad \text{where a = g(square(max(x, 4))));} \]

Note that == associates a name with an expression, it does not do assignment.
To see this:
let x == E1 in
  if (let x == E2 in E3)
    then x
  else 1 + x

The first let associates E1 with x.
The second let doesn’t change this.
Instead it renames E2 as x within E3.
Outside E3 x has its original meaning.
So far we have used qualified expressions to save on evaluation.

We also use them to clarify functions.
A third use is to decompose tuples.
dec quot : num # num -> num;
  --- quot(q, r) <= q;
dec rem : num # num -> num;
  --- rem(q, r) <= r;
let pair == IntDiv(x, y) in quot(pair) * y + rem(pair)

This latter expression pattern matches (q, r) with the result of calling IntDiv.

User defined data
As in most languages, we can’t do much interesting stuff in HOPE without defining data.
This is way simpler in HOPE than in other languages.
Consider handling lists.
In C, we have to use structs, and pointers and worry about memory.
Even in Java we have to use the right constructors.
In HOPE we just deal with the recursive definition of a list.

A list is either empty or an element followed by a list.
data NumList == nil ++ cons(num # NumList)
Here nil and cons are constructors.
A single element list is then:
  cons(3, nil)
And the list comprising 1, 2 and 3 is:
  cons(1, cons(2, cons(3, nil)))

With this definition we can build lists of any type:
  AnyCons(1, AnyCons(2, nil))
AnyCons(‘a’, AnyCons(‘b’, nil))
AnyCons(AnyNil, AnyCons(AnyCons(1, nil)))
AnyNil
The last two are a list of lists, and a list of unspecified type.

Lists are so common that they are built into HOPE.
i infix :: : ?
data list(alpha) == nil ++ alpha :: list(alpha)
We can also write lists as, for example [1, 2, 3].
Strings are lists of characters.
With this information it is easy to write functions to handle lists.

dec join : list(alpha) # list(alpha) -> list(alpha);
  --- join(nil, L) <= L;
  --- join(x::y, L) <= x :: join(y, L)
dec rev : list(alpha) -> list(alpha);
  --- rev(nil) <= nil;
  --- rev(x::l) <= rev(l) join [x];
Note that join is predefined in HOPE as the infix function «."
Higher order functions

- Consider
  
  \[
  \begin{align*}
  \text{dec IncList} & : \text{list(num)} \to \text{list(num)}; \\
  & \quad \text{--- IncList(nil) <= nil;} \\
  & \quad \text{--- IncList(x::l) <= (x + 1)::IncList(l);} \\
  \text{dec MakeStrings} & : \text{list(char)} \\
  & \quad \to \text{list(list(char));} \\
  & \quad \text{--- MakeStrings(nil) <= nil;} \\
  & \quad \text{--- MakeStrings(c::l) <= [c]::MakeStrings(l);} \\
  \end{align*}
  \]

- While doing different things, these two functions have the same basic form.

Both operate on a list and apply a function to every member of the list.

- The two functions are:

  \[
  \begin{align*}
  \text{dec Inc} & : \text{num} \to \text{num} \\
  & \quad \text{--- Inc(n) <= n + 1} \\
  \text{dec Listify} & : \text{char} \to \text{list(char)} \\
  & \quad \text{--- Listify(c) <= [c]} \\
  \end{align*}
  \]

- We can capture this by defining a higher order function

  \[
  \begin{align*}
  \text{map} & : (\alpha \to \beta) \# \text{list}(\alpha) \\
  & \quad \to \text{list}(\beta); \\
  & \quad \text{--- map(f, nil) <= nil;} \\
  & \quad \text{--- map(f, x :: l) <= f(x) :: map(f, l);} \\
  \end{align*}
  \]

- We can then write down the equivalent of our two earlier functions.

  \[
  \text{map}(\text{Inc}, L) \\
  \text{map}(\text{Listify}, L)
  \]

Of course, this relies on us having defined \text{Listify} and \text{Inc}.

- However, we don’t even have to do this.

- \text{HOPE} provides us with the means to write anonymous function bodies when and where we need them.

- For example:

  \[
  \text{lambda } x \Rightarrow x + 1
  \]

- Here we have to use the word \text{lambda}.

In general, we can replace any function with a \text{lambda} expression.

- We replace:

  \[
  \begin{align*}
  & \quad \text{--- f(x) <= } E \\
  & \quad \text{with} \\
  & \quad \text{lambda } x \Rightarrow E \\
  \end{align*}
  \]

- Thus the function \text{IncList} is the same as:

  \[
  \text{map}(\text{lambda } x \Rightarrow x + 1, L)
  \]

- Note that we have problems defining a recursive \text{lambda} because there is no name to use in the recursion.

- Instead we have to use a \text{let} or \text{where}.

- For example:

  \[
  \begin{align*}
  & \quad \text{let } f \equiv \text{lambda } x \Rightarrow \text{if } x = 0 \text{ then } 0 \\
  & \quad \text{else } x + f(x - 1) \\
  \end{align*}
  \]

  (which computes the sum of the first 3 numbers.)

- Such constructs are called \text{recursive let} and \text{recursive where}.

Some functional languages make these separate constructs (e.g., \text{letrec}).

- In \text{HOPE} \text{lambda} expressions can also contain a number of parts.

  \[
  \begin{align*}
  & \quad \text{--- IsEmpty(nil) <= true;} \\
  & \quad \text{--- IsEmpty(_,_:_) <= false;} \\
  \end{align*}
  \]

- becomes

  \[
  \text{lambda nil => true | _,_: => false}
  \]