HLP Worksheet 4 - Programming with Abstraction and Interfaces

Contents

Introduction

This worksheet uses an extended code sample to explore issues you find when putting together small fragments of a program - such as you wrote in the first three worksheets - into a larger application.

The key concepts, taught here from a set of examples, are:

- *functional abstraction*: using functions to hide details and modularise code
- *interfaces*: specifying the inputs and outputs of a module of code so that:
  - It can be written independently of and in parallel with other modules
  - Its implementation need not be understood to use it. Further, its implementation can be changed at some future date without rewriting other code
  - A module can have additional functionality added independently of other modules

These concepts are central to programming in any language: you may already have thought about them in C++ programming. We will see how they work out in the context of a largish program: you will use similar methods in your project work. The ideas here are equally useful in other (non-functional) languages.

> Abstraction is a fundamental idea that has many different manifestations. IEEE defines abstraction as *a view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.* The concept of abstraction can be used in two ways: as a process and as an entity. As a process, it refers to a mechanism of hiding irrelevant details and representing only the essential features of an item so that one can focus on important things at a time. As an entity, it refers to a model or view of an item.

When writing larger programs there are some operations that seem to require non-functional implementation. F# allows this via "assignment" in various ways. We will see examples of how these can be used when designing a general program execution tracing interface to our program.

As you do this worksheet think about how abstraction, and clear specification of interfaces, are used, for discussion etc in the lecture afterwards.

New F# syntax

The new F# language constructs taught here relate to *assignment* and *active patterns*.

- F# is by default functional but has two ways to implement assignment when you need it.
- Active patterns are a clever and very useful way to perform data matching and transformation.

Syntax:

- Mutable let definitions: named local values that can be changed by assignment
- Reference cells (type `T Ref`): cells that can be read or written.
- Active patterns: a way to get programmable matching in `match` expressions.

Syntax	Name	Description
`let (|A|B|) x = ...`	active pattern	Defines patterns **A** and **B** as *active patterns*
`let (|A|_|) x = ...`	partial active pattern	Defines pattern **A** as a *partial active pattern*
`T1 Ref`	*reference cell type* containing type **T**	cell whose contents can be overwritten by assignment

Syntax	Name	Description
`!x`	dereference operator	Returns the current contents of cell `x` which must be a `Ref` type
`x := v`	assignment `x : T1 Ref, v: T1`	write value `v` to cell `x`
`ref v`	ref type constructor	Create a new `Ref` cell containing `v`
`let mutable x = v`	mutable declaration	`x` behaves like a normal mutable variable **assigned** using `<-` **read** as normal (no dereference `!` needed).

Introduction

We will investigate - as an example - code solving the classic "compile and execute a language" problem. This has input a string representing some expression to be evaluated, and output the calculated value of the expression. The *input* could be simple arithmetic: `1+2*(11-4)` or a whole functional language like F#. The output would be a number in the first case, or a more general data value in the second.

On the language side we look in detail at how F# pattern matching works: one of the most sophisticated areas of the language syntax. We also compare two different ways in which F# allows assignment.

The code for this workshop will be "toy" - smaller than any real application. However the same structure (and very similar code) is used in Workshop 5 where it implements a compiler and run-time system for a complete functional language.

EEE and EIE have both seen problems like this, although EIE have studied them in much more detail than EEE. In this worksheet you will be given all the code that is needed and asked to think about how it is coded in F#, and also the above issues of abstraction and interfaces, so this difference will matter less. I've included some details of how the sample code works.

Code used in this Worksheet. This is a Visual Studio project with 5 source files. You can however use it from Visual Studio Code with a little inconvenience.

Modularity in F#

In F# code is divided into independent modules each containing functions and types. Unusually the F# compiler imposes a strict *no forward reference* rule meaning that source files (and hence modules) must be ordered for compilation such that every type and function referenced is previously defined. The only exception is `let ... and` or `type ... and` statements which allows small portions of code to be mutually recursive. You will remember that `type ... and` is considered a *code smell* something normally to be avoided.

There is no technical requirement for the forward reference restriction, except that the benefits of intellisense type inference in an editor are not easily possible with forward references. However many people would argue that the discipline introduced by this rule actually helps large-scale program design by preventing cyclic dependencies. This makes compulsory in F# what is good design practice (but seldom actually adhered to) in other languages. Figure 1 illustrates a typical *layered architecture* as is good practice, without backward links, and *spaghetti-like* organisation, where any part of the code can depend on any other and the modules have complex interconnection.

The first stage in high level design is identifying distinct modules performing independent parts of the required work. Obviously a layer within a layered architecture exists when subfunctions and types can be extracted into a library module written using lower layers, and used by upper layers. Figure 1 shows two other ways we might seprate modules using functions: *pipeline refinement* and *data-driven refinement*.

In pipeline refinement the problem is broken into a sequence of steps each of which transforms the data. This is what you used in Workshop 1. Where the functions in the pipeline are large, each incorporating lots of subfunctions, this is an effective way to modularise and structure a program. Pipeline refinement is a special very simple case of a layered architecture, where there is a single function connecting each of the layers.

In data-driven refinement the input data is classified into different classes, and each class is then handled by a separate subfunction (class is used here in an English sense, not a technical OOP sense). For example in a programming language compiler it would be normal to have different functions to process different language constructs. The function actually used when processing language input will therefore depend on the data. Following the structure below for data-driven refinement does not always lead to useful modularity, For example where each function a, b, c must recursively call other functions (often true in parsers) we get cyclic dependence which makes independent testing of the data-driven modules impossible. But in some cases it can be very effective.

These two methods of program refinement are orthogonal and often used together.

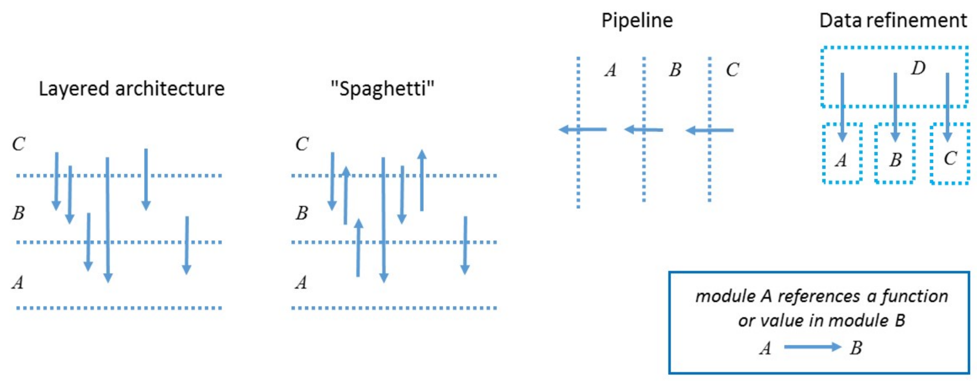

Figure 1

The code fragments below show (in a simplified form) how pipeline and data refinement emerges from different styles of F# code. In order for *useful* modularity to come from this, the constituent functions a,b,c need to be large enough to be worth splitting off as separate modules. Note that the diagrams in Figure 1 show the corresponding A, B, C modules

```
let pipelineRefine x =
    x
    |> a
    |> b
    |> c

let dataRefine x =
    let z =
        match x with
        | Adata y -> a y
        | Bdata y -> b y
        | Cdata y = c y
    d z
```

Note that the arrows in Figure 1 represent data dependency. `A -> B` means that A references on B and therefore B must be calculated before A, and therefore appear in the pipeline before A. So the pipeline order is the opposite of the arrow order.

Pipeline refinement in example code

The traditional pipeline for processing our problem splits work into three parts:

- *Tokeniser* : turns the input string into a sequence of tokens each representing a meaningful item like a number, symbol or operator
- *Parser*: turns the sequence of tokens into an *abstract syntax tree* (AST) - a tree data structure that represents the meaning of the program
- *Execution Engine*: calculates the correct answer from the AST. In the case of a program this involves running the program, in the case of an arithmetic expression this involves evaluating the expression. Note that in a functional language there is no real distinction, a program is simply an expression with some extra complexity (function definitions) added.

Given this the top-level code can be written without knowing the details:

```
    let run progTest =
        progTest
        |> tokenise
        |> parse
        |> execute

```

To make more progress we need the (matching) input and output types:

```
tokenise: string -> Token list
parse: Token list -> AbstractSyntaxTree
execute: AbstractSyntaxTree-> OutputType
```

Note that we have not written the functions, but we are specifying what they do (in part) through there signatures as above, and in part through a detailed description of how the output is related to the input. These two things constitute an interface that ties down the required inputs and outputs of each pipelined subfunction while abstracting away from the detailed implementation that we have not yet considered.

The exact definition of `AbstractSyntaxTree` and `OutputType` (and therefore the exact interfaces) we can leave till later.

Data-driven Refinement

This will be considered later in the sheet.

Exercise 1 - Debug Tracing

At this high level we can consider how this program will be debugged. There are two ways: use built-in tools from a debugger (see later), or add debugging capability to the source code. Both are useful in different ways, but debug code added to your source is the more powerful technique because it can provide output under programmatic control.

Typically, in worksheets so far, to debug code, you have added **`printfn`** statements to functions which print to console as a side effect. For a large program this is not satisfactory. How can we make debug printout more sophisticated? We would like to control *what is done with the printout*. The obvious solution is to add debugging information to the *outputs* of each function. Put the "what has been printed so far" output into a type **`TraceInfo`** and refine the top-level pipeline function signatures so that trace output flows through each top-level function with new output added.

```
tokenise': string * TraceInfo -> Token list * TraceInfo
parse': Token list * TraceInfo -> AbstractSyntaxTree * Traceinfo
execute': AbstractSyntaxTree * Traceinfo -> outputType * TraceInfo
```

We need to design a function **`trace`** that can replace **`printfn`** inside our top-level functions to do the trace printout. A line of debug printout can be generated from any **`trace`** function and threading together all of these outputs functionally would have implications for the structure of the rest of our code. We will consider later on a pure functional solution to this problem, but for now take a pragmatic view by using *mutable variable assignment* (which you used in Worksheet 2 to write the **`memoise`** function).

We will limit the use of assignment (changing the mutable variable to add a new line of printout) to the smallest possible blocks of code, keeping top-level inputs and outputs immutable data. This use of assignment within a functional program does not cause many problems, see the green box below for details.

- The trace printout may have order that changes, but this will only affect the order of the print lines within one top-level traced function. So the main reason for disliking assignment, that meaning changes with order of assignments, is only true if you care about the relative order of such printout lines. Because this is debug printout this relative order is not critical.
- The use of mutability is strictly encapsulated by the trace of each high level function. The result is functional, except for the dependence on execution order noted above.
- The fact that trace causes side effects (of local functions) that are not obvious in the function signatures is not a problem. It is meant to do that, and the printout is obvious in the top-level function signature.

Here is one solution to generate debug printout in this way. The debug printout is implemented by a function **`traceWrap`** that wraps each of our functions **`tokenise`**, **`parse`**, **`execute`** adding trace printout but preserving the rest of the functionality of each function.

```
let tokenise' = traceWrap tokenise
let parse' = tracewrap parse
let execute' = traceWrap execute
```

The signatures of our main functions **`tokenise`** etc must be changed to allow the created **`trace`** function to be passed to them as first parameter. Since each function is being wrapped it no longer needs **`TraceInfo`** as an input (that will be dealt with by the wrapping function). **`traceInfo`** is equally not needed as a main function output - that will also be provided by **`traceWrap`**. The main function signatures are back to their original signature, but with an added first parameter that feeds in the **`trace`** function!

`traceWrap` will accumulate *all* trace output as a list of **`TraceItem`**.

```
type TraceKey = | case1 | case2 | case3 // D.U. specifying all classes of trace printout
type TraceItem = TraceKey * string // Single message stored with key for later printing
type TraceInfo = TraceItem list // all trace messages so far

/// Wraps pFun providing traceOutput to it as 1st parameter to do tracing
let traceWrap pFun (pIn, traceInfo) =
    let mutable newTraceItems: TraceItem list = [] // accumulates trace output
    let traceOutput key str =
        newTraceItems <- (key,str) :: newTraceItems
    let pOut = pFun traceOutput pIn
    (pOut, List.append newTraceItems traceInfo)
```

each string that is output using trace will be given a `key` that classifies the output. This can be used to switch certain classes of output on or off. We will incorporate keys into the type system by using a D.U. type value for the key. This type would typically be a number of single-value alternates corresponding to different classes of output, so that keys are the tags of the D.U.

Inside a top-level function trace printout using the key `Character` might be implemented as:

```
let tokenise trace chars =
   // code
   trace Character <| sprintfn "next char = %c" ch
   // more code
```

Note the use here of `<|`[1]. Note that inside `tokenise` the function used to trace execution: `trace` is a parameter that is passed to `tokenise` by `traceWrap`.

Q1. Why can we not put the `sprintf` inside `traceOutput`, so tracing would be more compact?

```
traceOutput Character "next char = %c" ch
```

Answer

Now the top-level pipeline, with trace output added, looks like:

```
let run progTest =
    progText
    |> traceWrap tokenise
    |> traceWrap parse
    |> traceWrap execute
```

The output of this pipeline is now a tuple, first component of which is the "real" output, second component of which is the trace output.

Q2. What order are the trace messages in the final output list?

Answer

Q3. Write a function `ppT` which when passed the `run` output will extract the trace item data and return all items from a given list of keys only.

Answer

This trace functionality is flexible and works well for debugging but has a higher than necessary overhead. The problem is that all trace messages must be computed as strings and added to the list of messages, even if they are discarded by ppT. There are two solutions, when higher performance is required:

- Use F# conditional compilation directives, like those of C but more limited, so that all trace statements are deleted from the source when not needed as indicated by a preprocessor symbol passed to the compiler. `#if TRACE_ON trace Character <| sprintfn "char = %c" ch #endif`
- Change the trace code so that a function is passed to trace instead of a string. The function is only executed to return the correct string if the given message is needed. There is a cost to passing such a function but it is smaller than the cost of generating the string. `trace Character <| fun () -> sprintfn "char = %c" ch`

Q4. Rewrite type `TraceInfo` and function `traceWrap` so that `trace` is used as above, and trace output is *collected* for a specified subset of keys only.

Answer

Q5. Function `trace` is defined as a subfunction of `traceWrap` and then passed to the top-level functions in which it is used. Alternatively it could be defined at top level before all other functions, no passing as parameter needed. However, this does not work. Why not?

Answer

Q6. The use of an (unspecified) D.U. type in `traceWrap` represents a form of abstraction. Why is this? Consider how you would add trace functionality in which messages are classified according to an integer level, where less serious messages have a lower level and all messages higher than a given level can be printed.

Answer

Trace and exception processing: example

A simple implementation satisfying the top-level specification introduced above is given in the code for this worksheet. Download and open it in Visual Studio by double clicking the `expr.sln` file.

This example shows how a `tracewrap` function can be used as in Exercise 1. In addition to the trace functionality this implements `try with` *exception processing*.

You have already used *exceptions* generated by `failwithf` to abort execution on errors. The basic idea of an exception is that any function may terminate abnormally by *raising* an exception in which case its value becomes an *exception* object, that contains typically the exception name and a and error string. The exception object is unlike any normal value, its existence causes all outer functions to return immediately with the exception as returned value until some function *catches* the exception. If no user code catches the exception there will be top-level error.

For a brief description of F# exceptions see here. In this Worksheet the use of exceptions will be mostly self-explanatory.

The `try with` construct (lines 85-95 `common.fs`) allows a program to *catch* its own errors and process them itself. In this case the error processing prints out all trace messages using a console print function `cprintf` that prints in color for an easier read trace of execution up to the point of the error. Check this by temporarily changing `src1` to `src2` on line 4 of `program.fs` and running the program. The code in lines 89-94 of `common.fs` will be executed from the `with` part of the exception catcher `try with`. This code prints out the current trace and waits for a key press. After a key press the exception will be restarted with `reraise` and Visual Studio will enter its own debugger exception handling code. This highlights the code that generated the exception (in this case `reraise()`). In the debugger dialog tick *break when this exception type is thrown*. Rerun the code and note that now the debugger exception handling happens first. However, if you click continue, the user defined exception handler will then be executed. Therefore in this mode you can use both the debugger information, which includes highlighting where the exception was generated, and user-defined information.

When working on large programs (as for your project) take time to fine-tune exception and trace handling so that it is convenient, with useful printout both during normal execution and if an unexpected exception happens. In functional programming you will find that most of your code works first time, and that unexpected exceptions are impossible from core F# code (but can happen from .NET libraries you interoperate with). Even in this relatively benign run-time environment, when there are problems, good trace and exception handling is valuable.

Both tracing, and exceptions, are not quite functional operations, and would never happen in a production program. They are not a good way to implement *normal* code. The question of what is the best way to instrument and debug errors is a big topic. Generally the approach in this module is to find ways of writing and structuring code so that errors are less likely to happen in the first place, so sophisticated error handling is not relevant. The mechanisms described here suffice for the work done in this module, and extend fairly well to larger projects.

Exercise 2 Module structure

F# uses *modules* to organise code, avoid name clashes, and (optionally) hide local information. The code you are given for this worksheet is arranged in five source files each of which contains a single *module* named after the file.

file	module	description
common.fs	common	definitions used by more than one module
tokenise.fs	tokenise	definitions used by the tokenise function
parse.fs	parse	defінbitions used by the parse function
evaluate.fs	eval	definitions used by the eval function
program.fs	program	top-level code

There is no fixed relationship between modules and files: any file can have any number of modules (which may be nested). However the one module per file organisation works well and is recommended. F# modules are described in detail here.

Types and values `x` defined in a module e.g. `Ymod` can be accessed outside the module using the module name as `Ymod.x`. Alternatively the module can be opened, to make `x` available without the qualifier:

```
open Ymod // open statements are usually at the start of the file.

printfn "%A" Ymod.x // always allowed
printfn "%A" x // allowed because module Ymod is open
```

Q7. What are the types of modularity (as discussed in the Modularity section) shown by these modules?

Answer

Modules allow you to package definitions and avoid name clashes. They support *information hiding* - a definition declared **`private`** in a module is not visible from outside the module. [2]

Exercise 3 - Active patterns

The code in **`tokeniser.fs`** defines a set of functions of type: **`char->bool`** that classify characters into different classes. F# provides a convenient syntax that can often be used for this called an *active pattern* illustrated below.

```
let (|EVEN|ODD|) n =
    if n % 2 = 0 then EVEN else ODD

// using active patterns in a match statement
match x with
| EVEN -> "I win"
| ODD -> "You win"
```

Active patterns can partition the values of a type into any number of disjoint sets, the union of the sets must equal the set of all values in the type, so that every value is matched by exactly one of the active pattern identifiers. Active patters are defined by an arbitrary function, given in the definition, which returns one of the partition identifiers for any input.

Q8. Why is it not possible to encode all the boolean functions on lines 25-37 of **`tokeniser.fs`** as a single active pattern?
Answer

Q9. Write an active pattern for as many of these functions as possible:
Answer

Q10. Replace line 67 of **`tokeniser.fs`**

```
| ch :: r when isWhiteSpace ch -> tokenise1 r
```

By an equivalent construction using your active patterns
Answer

The **`isAlphaNum`** function - an **`OR`** connection of two cases, can be implemented using two active pattern cases with the same expression:

```
    | Alpha | Digit -> "character is alphanumeric"

```

Not used here, but active pattern cases can be **`AND`** connected with a single **`&`** operator:

```
    | Alpha & Digit -> // this will never happen because Alpha and Digit are never both true

```

Q11. How could you add an active pattern for **`isNewline`**?
Answer

Active patterns allow flexible *matching* of data. In the next section we look at *partial active patterns* which are very different, more powerful, and allow both *matching* and *transformation* of data.

Tokeniser: using partial active patterns

Tokens are determined, like words, by processing the input characters sequentially and noting delimiters (e.g. whitespace). The **`tokenise`** function transforms the input string to a list of characters which is then recursively processed to extract tokens (meaningful words in the programming language). The sequential list of output tokens is output as the value of **`tokenise`**.

```
"10 + 2" ->
[ '1' ; '0' ; '+' ; ' ' ; '2'] ->
[ TokIntLit 10 ; TokSpec "+" ; TokIntLit 2 ]
```

Finding the next token (*lexical analysis*) can be seen as an operation of matching and transformation of an input list of characters into an output list of tokens.

The `tokenise1` function (lines 64-71 of tokenise.fs) performs this function using partial active patterns.

> **Q12**. What does the `::` (list cons) operator on line 67 match and how does it transform the matched value?
>
> Answer

> **Q13**. What is the function here of the `when` part of line 68?
>
> Answer

In this example `opMatch` is written very compactly using library functions `Option.map` `List.skip`, `List.tryFind` and user-defined function (lines 43-50) `charListStartsWith`. The 1st component of the output tuple `t` is the new token (an operator `TokSpec` token) and the second component `r` is the rest of the unmatched characters from `lst`.

> **Q14**. Check that you understand how `opMatch` works by using it in a code fragment and testing this.
>
> (Q14 Answer not supplied)

> **Q15**. Write a partial active pattern `IntMatch` (like `OpMatch`) that will match with non-negative integers and allow string `src` to be tokenised correctly.
>
> Answer

When an identifier such as `OpMatch` is defined as an active pattern the definition is therefore always a function of type `T1 -> T2 Option` where t1 is the type of the matched value, and T2 is the type of the transformed output. The definition function can be used as a normal function outside a match expression if this is needed:

```
let x = (|OpMatch|_|) ['*' ; '*']
```

> **Q16**. What is `x` set equal to by this definition?
>
> Answer

More information on active patterns including some useful new system functions for matching strings[3].

Parser: more active patterns

The next phase of the code is `parsing` from file `parse.fs`. The `parser` function takes a (linear) list of tokens and returns an *abstract syntax tree* (AST) representing the expression. This operates as a *recursive descent parser* which identifies *items* and *multiplicative terms* in the abstract syntax. As below each term is defined recursively from sub-terms

AST node	description	content	F# in AST
item	literal or bracketed expression	*literal* `"("` *mult-term* `")"`	`Literal (TokIntLit n)` n/a (use *mult-term* node)
mult-term	items joined together with `"*"` or `"/"`	*item* *item* `"*"` *mult-term*	n/a (use *item* node) `Apply [op ; lItem ; rTerm]`

For example the expression: `1*(6/4)` would be parsed as:

```
     Apply [
            Literal (TokSpec "*")
            Literal (TokIntLit 1)
            Apply [ Literal (TokSpec "/")
                    Literal (TokIntLit 6)
                    Literal (TokIntLit 4) ]
       ]

```

The parser contains two mutually recursive functions, one for each of the above AST nodes. Each function works by matching the input stream to determine which of its options applies. For example:

- `PITEM (TokSpec "(")` → parse bracketed expression
- `PITEM (TokIntLit n)` → parse item

In `PITEM` (line 20) the combined pattern is:

```
         | TokSpec "(" :: PMULTERM (term, (TokSpec ")" :: r)) ->

```

This match decodes as follows:

- The first token in the matched token list must equal `TokSpec "("`
- The tail of the token list is matched by `PMULTERM`. If it matches, its output `x` is then matched with `term , TokSpec ")" :: r)`. `term` will be set to the output *mul-term* from `PMULTERM` and if the remaining tokens match `TokSpec ")" :: r` the whole match succeeds. Thus if the whole match succeeds `term` and `r` are defined and can be used on the RHS of the pattern match. If any part of the match fails, the next match pattern `|` is tried.

Exercise 4 - Evaluater module

Recursive evaluation

The `evaluate` function has interface defined by the AST type as input and type `Data` as output. It takes an AST and returns the correct evaluated (numeric) value encoded as a `Data` value.

Given this interface there are several ways we could write the evaluater. The first implementation is as a recursive function that calls itself to evaluate the entire AST as in lines 18-36 of `evaluate.fs`.

The advantage of having clearly specified function with a well-defined interface is that the implementation can be completely changed in one module without affecting the rest of the code[4].

Graph Reduction

A different implementation of evaluation uses the technique of *graph reduction*. This technique can be extended to arbitrary functions, so although it is not an easy way to implement binary arithmetic we will see in the next worksheet that it is highly expandable. We represent a binary operation as a curried function and hence two *function applications*:

$$a + b \equiv \mathtt{apply}(\mathtt{apply}(\mathtt{plus}, a), b)$$

Q17. What type would

$$\mathtt{apply}(\mathtt{plus}, a)$$

have in an F#-like type system?

Answer

This can be represented as a *graph* in which each node is either an *apply* operation or an integer *leaf* or an operator *leaf* as in type `Data` below.

Q18. Draw a graph (as a set of binary *apply* nodes) for the expression: `1 + 2*3`.

(Q18 Answer not supplied)

We will evaluate an expression, stored as a graph, by *graph reduction*. That is, overwriting nodes in the graph corresponding to binary arithmetic operations by the calculated result. $\mathtt{DCell}, \mathtt{DName}, \mathtt{DInt}$ represent binary (apply) cell, operator, and integer constructors:

$$\mathtt{DCell}(\mathtt{DCell}(\mathtt{DName}\ "+", a), b) \rightarrow \mathtt{DInt}(a + b)$$

During *graph reduction* nodes are repeatedly changed to a new equivalent (usually beta-reduced) value. After all such chnages are implemenetd the final graph represents the evaluated answer.

This could be done without mutability (by returning a new tree structure) but for efficiency we need a representation of the tree in which specific apply nodes can be overwritten in F#: The `Ref` type constructor provides a writable (mutable) location in memory with contents the given type and operations as here.

Based on this the data type we need is:

```
// A graph node
type Data =
    | DInt of int // literal integrers
    | DName of string // used for operators "*" etc
    | DCell of Hd : Loc * Tl : Loc

// a mutable cell used so that graphs can be mutated during beta reduction
and Loc = Data ref
```

Q19. Suppose `x: Loc` contains a graph representing the operation `+ a b` where `a` and `b` are two integer literals. Write a fragment of F# code that will correctly perform *reduction* on this graph, replacing the graph by its numeric value encoded as a **`Loc`**. You may find the following partial active pattern, which always matches and returns the contents of a ref cell, useful:

```
let (|R|_|) x = Some (! x)
```

Given this pattern, the match expression:

```
match DCell( ref(DInt 1), ref(DInt 2)) with
| DCell( R(DInt x), R(DInt y)) -> printfn "x=%d, y=%d" x y
```

Will match and bind **`x`** and **`y`** to the integers **`1`** and **`2`**.

Answer

Q20. Why is it not possible to perform similar reduction on a graph: **`x:Data`**?

Answer

Q21. Matching more than one nested **`DCell`** with **`R`** is annoying because of the large number of brackets. Write a partial active pattern that will match a **`DCell`** and transform it into its two components de-referenced. Rewrite your answer to the last question using this.

Answer

Exercise 5 - Interfaces

Having looked in detail at this code it is possible to analyse what are its interfaces. This code is much shorter than any normal program divided into 5 modules, but it shows how larger samples can be written. A much larger version of similar code, with the same structure, is used in Worksheet 5.

Each of the three main modules: Tokeniser, Parser, Evaluater, represents code that can also be independently owned and implemented.

A *module dependency graph* shows you what are the dependencies between modules. It has an arrow from module A to module B if A uses any type or value defined in B. In F# cyclic dependency between modules is not allowed so the module dependency graph will be a tree.

Q22. What is the module dependency graph for this code?

Answer

What is an interface?

The word *interface* is commonly used as a specific programming construct (F# has an **`interface`** keyword that is used when defining interfaces in an OOP world).

Interface is also used more generally, as in this worksheet, to mean the features of a module - both programmatic, e.g. types, and non-programmatic, such as documentation about module functionality - that must be communicated between the module owner and the module user. If these interfaces are clearly and completely defined then it is possible to develop code independently in parallel without introducing bugs.

Programmatic interfaces in F#

Programmatic interfaces - used in some OOP languages - are in fact one example of this since they define how code in one OOP *class* can be used in the context of another different class. For example in F# a string is defined as a literal object - not an array of **`char`** so that:

```
    let x = "cat"
    let y = [|'c' ; 'a' ; 't'|]
    x = y // this expression will give a type error

```

gives type error:

```
C:\\Program Files (x86)\\Microsoft VS Code\\Untitled-1(3,3): error FS0001: This expression was expected to have type
    string
but here has type
    char []
```

Thus in the above F# fragment `x` and `y` have completely different types.

How can we decode a literal `string` into its constituent `char`? In F# `string` implements the .NET `IEnumerable<char>` interface which means that strings can be used as sequences of characters. In F# this is very useful, and also a bit surprising. Something of one type (`string`) will magically turn into another type `seq<char>` when this is required by the type context.

The type constructor `seq` is similar to `list` and `array`. It constructs *sequences* - like lists but items in the sequence are not computed until they are required. The uncomputed iterms in the sequence are held as a *continuation* which is a function that when evaluated will return the next item in the list. This is called *lazy evaluation*. F# has module `Seq` with functions on `seq` just as `array` and `List` have functions on arrays and lists. Thus, after the previous definitions of `x` and `y`:

```
    (x |> Seq.toArray) = y

```

will work and returns `true`.

Since `IEnumerable` is used commonly by .NET library code written in other languages it is important for F# to interoperate with it. In fact any F# class can be defined to implement any interface, so that programmers can implement types that operate like `string` themselves by making the type a class and giving it an interface. That is beyond the normal scope of this module, but for those interested here is how F# `seq<T>` (or `T seq`) type implements `IEnumerable<T>`.

What is abstraction?

Abstraction can be anything that hides information (e.g. details of implementation) from higher levels of code. The obvious abstraction used here is that the source text is converted to a list of tokens before it is parsed. The tokens abstract what is important from the details (such as white space and comments) of the input string.

A useful type of abstraction in programming comes from generalisation. In all programming the common form is functional abstraction where some operation used in multiple places is generalised and written as a single function: e.g. `isOpInList` from `tokeniser.fs:40`.

An *interface* is an abstraction of the functionality being interfaced: leaving out everything except what is needed to use that functionality.

Summary - managing a large project

In addition to introducing some new F# syntax this worksheets provides background for some of the issues you will encounter when writing a large program in your group project:

- How do you split it up?
- How do you track interfaces and interface changes?
- How do you ensure that changes in one module do not break another?
- How do you manage errors and debugging?

These are things you will discover through practice but there are some ground rules:

- Look for ways to modularise the code. Typical standard ways are using *data-driven* and/or *pipeline* refinement. In this example the pipeline refinement is obvious: tokeniser, parser, evaluator form a pipeline with each stage mostly independent of the others. Data-driven refinement could be related to different token types where the code to implement a new type of tone - for example 64 bit integer literals - is written as a partial active pattern `INT64TOK` . in a separate module `LongIntTokeniser` that is a dependency of `Tokeniser`. You might think that the parser could similarly be modularised, isolating specific partial active patters for given constructs in independent modules, but there is a problem. In the parser many of the different active patterns that represent expression constructs are *mutually recursive*. This is inevitable because expressions are recursively defined in terms of similar subexpressions. Modularising the parser this way would therefore lead to code with circular dependencies - not allowed in F#[5].
- Future enhancement, if you are lucky, is either data-driven refinement, with new D.U. cases added, or pipeline refinement with a whole new module added. If you are unlucky it is making changes across all modules!
- Be clear about responsibility for types that define module interfaces:, in this example: Token, AST, Data. Who *owns* the type definition? Who needs to know if it changes? Typically type changes are adding D.U. cases to implement data-driven refinement, for example extra AST cases for more complex expressions.
- Be clear about changes in specification of functions that define an interface:
  - adding a parameter
  - changing function definition
  - altering exceptional behaviour. Because we don't normally worry about when things go wrong this tends to be less well defined in an interface, therefore it is a big source of bugs. Define your function for all inputs, and make it a total function (no exceptions) if possible.

- Minimise and localise side effects. In pure functional code there will not be any. Where side effects exist make this very clear and if possible localised to a specific small part of code. This rule actually helps simplify code written in any language, not just a functional language.
- Be clear about dependencies. E.g. here **`common.fs`** is dependency of multiple modules. Use large scale organisation which minimises "spaghetti-like" dependency between modules. In F# the lack of forward references encourages good structure from the start. The code in this example is very well modularised with almost no interdependence between the main modules. In larger projects that can be more difficult to preserve. Having a small set of common definitions is a usual way to keep things clean.
- Include test code. One (very helpful) way to specify a module is to provide a set of (automatic) tests that it must meet. Many programming methodologies require tests to be written before the module code that passes them. Certainly tests should be written and used as part of the specification of a module. One of the key advantages to adding good tests is that later changes that break earlier written features will usually be picked up from failing tests.
- Restructure if needed. Restructuring is changing the module interfaces. Sometimes the original structure of a project turns out to be wrong when additional functionality is added. In this case altering module interfaces, moving code to a better place, may be the best solution especially if it is early in the project. But remember this is high cost, unless the changes are just additions to D.U. cases orthogonal to existing as in data-driven refinement!
- Refactor if needed. Within a module if the implementation gets too messy it may be necessary to refactor. This can be done without changing the interface, tests, or affecting other parts of the project. It is therefore lower cost. Here are some guidelines for when you should do this.

Ticked Exercise

Implement types **`Command`** and **`Response`** as below, and a function:

`environment: Command->Response`

in F# which holds as a local data information about variable-value associations, accepts commands as input, and returns variable values:

```
type Command =
    | Read of string
    | Write of (string, int)
    | Assign of (string,string)
    | Parse of (string)

type Response =
    | VarValue of int // for commands that return data
    | ParseError // if command string is invalid
    | DataError // if the required data does not exist
    | OK // for valid commads that return no data
```

input F# value	operation	output
`Read v`	Read the value of variable **`v`**	**`VarValue n`** where **`n`** is the current value of variable **`v`** or **`DataError`** if **`v`** has never been written
`Write(v,n)`	Write 'n' into **`v`**	OK
`Assign(v1, v2)`	Write the current value of **`v2`**into **`v1`**	OK or DataError if v2 has never been written
`Parse s`	interpret **`s`** as in Table 3 execute it	

Table 2 - meaning of commands

Variable names are case sensitive, so **`"Aardvark"`** and **`"aardvark"`** are distinct variables.

input string	operation
`"READ v"`	Read(v)
`"WRITE v n"`	Write(v,n)
`"ASSIGN v1 v2"`	Assign(v1,v2)

Table 3 - valid commands for `Parse`

- *v*, *v1*, *v2* are alphabetic (underscore not allowed, upper and lower case allowed).
- *n* consists of 1 or more digits **`0-9`** with optional **`-`** immediately before digits.
- Any amount of whitespace is allowed before, or after, each significant part of the parsed string except between the optional **`'-'`** and the digits of *n*.
- Whitespace is defined to be **`' '`**, **`'\t'`**,**`'\n'`**,**`'\r'`**,**`'\f'`**.
- Any characters other than explicitly allowed here are a parse error.

Examples of valid strings are:

```
"READ     xvv"
"    WRITE   XxX -234    "
"WRITE cat 1"
"\n ASSIGN  A   \t   \n   b\t"
```

Any invalid string must result in **`ParseError`**.

Note that **`environment`** must be a function with encapsulated mutable state to hold the variables and their values, which can change. This can most easily be implemented using a local variable defined with **`let mutable state = ....`** The **`state`** identifier must not be initialised whenever **`environment`** is called:

```
    let environment =
        let mutable state = ...
        ....

```

or it would not remember anything. Instead you must define a function e.g. **`makeEnvironment`** which is called once, contains the **`let mutable`** definition, and returns the **`environment`** function which can then be called multiple times.

Reflection

The toy code shown in this worksheet will be used (in an expanded form) as a real example in Worksheet 5.

- Pattern matching, and active patters, are a key linguistic feature of functional languages that works very well. Having used them you start to wonder how anyone could ever write programs using **`if then else`**. It is an interesting question why (generally) this has not been taken up in mainstream languages, in spite of requests that it should be so. (Some of the innovative features in F# have been pushed into C# a few years later).
- Another key question is the merit of enforcing *no forward references*. Should programmers be forced (as in F#) to avoid cycles in module dependency graphs? This is a big restriction. There is a technical reason for keeping it in F# not present in languages that do not do type inference. This could be got round with more complex tooling. Otherwise, related questions are:
    - Why are forward references (and hence cyclic dependencies) undesirable in programs?
    - Do languages with mutable state *need* cyclic references more than functional languages?
    - Does the *passing a forward reference as a parameter* trick always resolve a forward reference? Is it somehow better (in terms of reducing program complexity) than having a normal forward reference?
    - Should *small-scale* local forward references be restricted less than large-scale (intra-module) ones? F# 4.1 specification has an interesting language proposal along these lines, see RFC FS-1009
    - Does F# ever allow dependency cycles? Are there cases where allowing cycles (on small or large scale) leads to better code than restricting them?

In F# module functions are all (by default) available externally as part of the module interface. Other languages, for example ML, F#'s highly influential ancestor, have explicit interface statements defining what is exported. Should these be added to F#? In F# a lot of the language detail is driven by a pragmatic wish to allow near perfect "as you type" type inference. How valuable is this feature? Should it drive language design?