# Modern Programming

John Skaller

April 8, 2020

# Contents

# Chapter 1

# Introduction

This book is intended to delve into modern programming techniques. Programming has changed. Nicolas Wirth, introducing Pascal language, coined the equation "Algorithms + Datastructures = Programs".

Whilst research into algorithms continues, many important algorithms such as sorting and searching are well understood. Modern research focuses instead on messaging protocols and type systems.

Today, the core issues in programming do not involve design of algorithms or data structures, since many libraries with heavily tested functions, and subject to competitive performance trials, are available off the shelf.

Rather, the central issues today revolve around *interfacing*, which means making software components work together. And it is not just software which requires interfacing, but the software developers themselves.

From a theoretical perspective, the core problem is restated: the central issue is *composability*. Modern type system research is heavily oriented towards studying how we can make components which can be easily plugged together. Themes include higher order polymorphism, polyadic programming, managing side effects in non-functional code.

In all programming, *correctness* is a vital issue. However from the perspective of a rapidly changing environment, business structure, and client requirements, confidence in code quality must be balanced against rapid development as a method for ensuring the supplier survives financially in the marketplace at all.

Perhaps for this reason most modern programming languages are dynamically typed and simplistic in nature, allowing rapid coding, and easy learning of the tools. Whilst most academic languages are toys primarily intended to provide a proof of practicality for theoretical models, there are a few systems which are production quality languages with strong support bases.

So in this book, I feel there are no real options other than to use a new programming language as the display vehicle.

# Part I

# Felix Overview

# Chapter 2

# Quick Start

We must of course begin with the traditional greeting!

```
    println$ "Hello World";
```

Here `println` is a procedure which outputs a value to standard output. The argument is of course a literal of type `string`. Calls to procedures must be terminated by a semicolon `;`. You will note, we do not require parentheses around the argument to denote a procedure call! We do not like parentheses much! The `$` sign will be explained in more detail later, but for now you should know it is just a low precedence, right associative application or call operator.

You can run this program from your console or terminal, once Felix is installed, by just typing:

```
flx hello.flx
```

assuming the file `hello.flx` contains the sample code and is in the current directory. Behind the scenes Felix does dependency checking, transates the program to C++, compiles the C++ to a machine binary, and runs it.

It works like Python but it performs like C++.

For more information see http://felix-lang.org.

## 2.1 Integers: `int`

Felix has the usual integer type `int` and literals consisting of decimal digits, and the usual **binary operators** `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division, and `%` for remainder. We also have `<<` to multiply by a power of 2, and `>>` to divide by a power of 2.These have the same semantics

as in C, because, these operators are in fact implemented in the Felix standard library by delegating to C.

```
println$ 42;
println$ (42 + 12) * 90 - (36 / 2 + 1) * 127 % 3;
```

Note you must put spaces around the - operator! This is because in Felix - is also a hyphen, allowed in identifier names.

We also have the **unary operator** - for negation, and for symmetry we also have unary + which does nothing.

These operators are just functions so here are some more: the function `str` converts an `int` to a readable string, the function `abs` finds the absolute value of an integer and `sgn` returns $-1$ if its argument is negative, 0 if it is zero, and 1 if it is positive.

We also have the usual **comparisons** on integers represented as infix operators: == for equality, != for inequality, < for less than, > for greater than, and <= and >= for less than or equal and greater than or equal, respectively.

## 2.2 Other integer types

Felix has a lot of other integer types corresponding to those found in C. Each of these types is distinct, none are aliases! See Table 2.2.

#### Lexicology

Integers may be specified with a radix, and an underscore may be used between digits as a separator:

```
var x = 0b1111_0011; // binary
var y = 0o777_321;   // octal
var z = 0d998_444;   // decimal
var a = 0xBADE_DAFE; // hex
```

### 2.2.1 Conversions

All the integer types can be inter-converted with a conversion. This is just the name of the target type, used as a function. For example:

```
var x : long = long 42uz;
var y : int64 = x.int64;
```

The dot notation is just shortcut high precedence reverse application operator much beloved by OO enthusiasts.

Table 2.1: Integer Functions

| name | kind | semantics |
|---|---|---|
| | All Integers | |
| `==` | `T * T -> bool` | equality |
| `!=` | `T * T -> bool` | inequality |
| `<` | `T * T -> bool` | less |
| `<=` | `T * T -> bool` | less or equal |
| `>` | `T * T -> bool` | greater |
| `>=` | `T * T -> bool` | greater or equal |
| `+` | `T * T -> T` | addition |
| `-` | `T * T -> T` | subtraction |
| `*` | `T * T -> T` | multiplication |
| `/` | `T * T -> T` | quotient |
| `%` | `T * T -> T` | remainder |
| `<<` | `T * T -> T` | multiplication by power of 2 |
| `>>` | `T * T -> T` | division by power of 2 |
| `-` | `T -> T` | negation |
| `+` | `T -> T` | no op |
| | Signed Integers | |
| `sgn` | `T -> T` | sign |
| `abs` | `T -> T` | absolute value |
| | Unsigned Integers | |
| `\&` | `T * T -> T` | bitwise and |
| `\|` | `T * T -> T` | bitwise or |
| `\^` | `T * T -> T` | bitwise exclusive or |
| `~` | `T * T -> T` | bitwise complement |

Table 2.2: Felix Integer Types

| Felix | C | Suffix |
|---|---|---|
| Standard signed integers | | |
| `tiny` | `char` | 42t |
| `short` | `short` | 42s |
| `int` | `int` | 42 |
| `long` | `long` | 42l |
| `vlong` | `long long` | 42ll |
| Standard unsigned integers | | |
| `utiny` | `unsigned char` | 42ut |
| `ushort` | `unsigned short` | 42us |
| `uint` | `unsigned int` | 42u |
| `ulong` | `unsigned long` | 42ul |
| `uvlong` | `unsigned long long` | 42ull |
| Exact signed integers | | |
| `int8` | `int8_t` | 42i8 |
| `int16` | `int16_t` | 42i16 |
| `int32` | `int32_t` | 42i32 |
| `int64` | `int64_t` | 42i64 |
| Exact unsigned integers | | |
| `uint8` | `uint8_t` | 42u8 |
| `uint16` | `uint16_t` | 42u16 |
| `uint32` | `uint32_t` | 42u32 |
| `uint64` | `uint64_t` | 42u64 |
| Weird ones | | |
| `size` | `size_t` | 42uz |
| `intptr` | `uintptr_t` | 42p |
| `uintptr` | `uintptr_t` | 42up |
| `ptrdiff` | `ptrdiff_t` | 42d |
| `uptrdiff` | `ptrdiff_t` | 42ud |
| `intmax` | `intmax_t` | 42j |
| `uintmax` | `uintmax_t` | 42uj |
| Addressing | | |
| `address` | `void*` | |
| `byte` | `unsigned char` | |

### 2.2.2 Bitwise operations

For unsigned integers only, bitwise operations are supported. These are `\&` for bitwise and, `\|` for bitwise or, `\^` for bitwise exclusive or, and `~` for bitwise complement.

## 2.3 Booleans: `bool`

The result of a comparison is a new type, `bool` which has two values, `false` and `true`. You can print booleans:

```
println$ 1 < 2;
```

Felix also has a special statement for asserting that a boolean value is true:

```
assert 1 < 2;
```

If the argument of an `assert` is false, then if control flows through it, the program is terminated with an error message. Note that as far as `assert` is neither procedure nor function you do not need to fix operator priorities with `$`.

Booleans support the usual **logical operators**, but in Felix they are spelled out. Conjunction is spelled `and`, whilst disjunction is spelled `or`, implication is spelled `implies`. Of course we also have negation `not`.

### 2.3.1 Conditional Expression

With `bool` and `int` we can demonstrate the conditional expression:

```
println$
  if 1 < 2 then "less"
  elif 1 > 2 then "greater"
  else "equal"
  endif
;
```

Note that bool also forms a total order and can be compared, we have `false` < `true` so that *a*`==`*b* means *a* is equivalent to *b*, we can also say that *a* is true if and only if *b* is true. It turns out inequality *a*`!=`*b* is the same as exclusive or. Be careful though, since if *a* is less than or equal to *b*, written *a*`<=`*b* this actually means that *a* implies *b* logically!

## 2.4 Special symbols

Felix has a very rich set of symbols. This is because, as well as the usual ascii-art symbols, it also supports almost the complete set of TEX, LATEX, and AmSTeX symbols.

TEXsymbols start with an backslash and are followed by a sequence of letters. These symbols display in typeset format when using the `flx_web` web server.

For example the following code:

```
var \alpha = 1;
fun \Gamma (x:int)=> x * x;
println$ \Gamma\alpha;
```

should be typeset as

```
var α = 1;
fun Γ (x:int)=> x * x;
println$ Γα;
```

## 2.5 Variables: `var`

Felix is a procedural programming language, so it has variables! A variable denotes an addressable, mutable, storage location which in Felix, like C, is called an *object*.

```
var x = 1;
var y = 2;
var z = x + 2 * y;
println$ z;
z = 2 * x + y;
println$ z;
```

This code shows variables can be used to factor expressions into a sequence of assignments. We assign variable $x$ the value 1, variable $y$ the value 2, add $x$ to twice $y$ and put the result in variable $z$, then print it.

Then we assign perform a different calculation and assign that value to $z$ and print it.

What appears to be a variable initialisation is actually equivalent to a definition of an uninitialised variable followed by an assignment.

```
var x: int;
x = 1;
```

The first statment reserves uninitialised store of type `int` named $x$ and the second assigns a value to it. Be very careful with variables, initialised or not! Felix has setwise scoping rules, which are similar to C's function scope used for labels. This means in a scope, you can refer to any symbol defined *anywhere* in that scope. We shall see this is useful for recursive functions because it eliminates the need for forward declarations. However the following code has undefined behaviour:

```
println$ x;
var x = 1;
```

There is no syntax error, no type error, and no lookup error in this code. The programmer used an uninitialised variable: even though the variable is assigned a value, it is done too late.

Strangely, this code has deterministic behaviour:

```
println$ x;
var x = "Hello";
```

but it may not do what you expect! It prints nothing! The reason is simple enough: when Felix creates a variable it is first initialised with its C++ default constructor. Since a Felix `int` is literally a C++ `int` the default constructor exists, but it is said to be trivial, meaning, it does nothing. This is to improve performance, in the case the first use of the variable will be to assign a value to it: there's no point putting a value in there and then overwriting it!

On the other hand Felix `string` type is just C++ `::std::basic_string<char>` and its default initialiser sets the string to the empty string `""`. That's what the code above prints!

## 2.6 Floating Point: `double`

Felix also provides a model of C++ type `double` with the usual operators. This is a double precision floating point type which usually follows IEEE standard. You can write a double precision literal in the usual way. Felix follows ISO C-99 for floating point literals.

A set of useful functions is also provided, corresponding to those found in C-99 header file `math.h`.

```
var x = 1.3;
var y = 0.7;
assert sqrt (sqr (sin x) + sqr (cos y)) - 1.0 < 1E-6;
```

Note there is a special caveat with floating point arithmetic. In Felix, `-` has higher precedence than `+`. This means that:

```
var x: double = something;
var y: double; something_else;
assert x + y - y == x;
```

because the subtraction is done first. This can make a difference for integers too, if a calculation overflows, but most floating point operators are not associative: order matters!

Similarly, division has a higher precedence than multiplication!

Floating reals are totally ordered and support exact comparisons. However these operations are not numerically sound, they're based on the underlying finite representation.

Floats also provide checks for `nan` pseudo value, as well as `+inf` and `-inf`.

### 2.6.1 Other floats

As well as `double` Felix provides single precision `float` and extended precision `ldouble` based on `float` and `long double` respectively, with the equivalent operators.

It also supports complex numbers with Cartesian representation based on C++ complex forms. The types are `fcomplex` based on `float`, `dcomplex` based on `double`, and `lcomplex` based on `ldouble`.

In addition Felix provides double precision based `quaternion` type.

## 2.7 Strings: `string`

Felix uses C++ strings for its own strings for compatibility. String literals have 6 forms following Python. Strings not spanning multiples lines can be enclosed in either single or double quotes. Strings spanning multiple lines may be enclosed in tripled single or double quotes.

```
var ss1 = 'Short String';
var ss2 = "Short String";
var ls3 = """
A poem may contain
```

```
many lines of prose
""";
var ls4 = '''
Especially if it is written
by T.S. Elliot
''';
```

Note that the triple quoted strings contain everything between the triple quotes, including leading and trailing newlines if present.

Strings can be concatenated by writing them one after the other separated by whitespace.

```
var rose = "Rose";
var ss5 =
  "A " rose ", "
  "by another "
  "name."
;
```

Note that concatenation works for string expressions in general, not just literals.

### 2.7.1 Escape Codes

Special escapes may be included in strings. The simple escapes are for newline, `\n`, tab `\t`, form feed `\f`, vertical tab `\v`, the escape character `\e`, `\a` alert or bell, `\b` backspace, `\'` single quote, `\"` double quote, `\r` carriage return, `\\` backslash (slosh).

These can be used in any simple string form. Note carefully each is replaced by a single character. This includes `\n`, even on Windows.

In addition Felix provides `\xXX` where each X is one of the hex digits 0123456789, $ABCDEF$, or, $abcdef$. The hex escape is at most two characters after the `x`, if the second character is not a hex digit, the escape is only one character long, the sequence is replaced by the char with ordinal value given by the hex code.

Felix also provides decimal and octal escapes using `\dDDD` and `\oOOO` respectively, with a 3 character limit on the decoder. Note carefully Felix does NOT provide C's octal escape using a 0 character. Octal is totally archaic.

Felix also provides two unicode escapes. These are `\uXXXX` and `\UXXXXXXXX` which consist of up to 4 and up to 8 hex digits exactly. The corresponding value is translated to UTF-8 and that sequence of characters replaces the escape. The value must be in the range supported by UTF-8.

Table 2.3: String Escapes

| Escape | Name | Decimal Code |
|---|---|---|
| | Basic | |
| \a | ASCII bell | 7 |
| \b | ASCII backspace | 8 |
| \f | ASCII Form Feed | 12 |
| \n | ASCII New Line | 10 |
| \t | ASCII Tab | 9 |
| \r | ASCII Carriage Return | 13 |
| \v | ASCII Vertical Tab | 11 |
| \' | ASCII Single Quote | 39 |
| \" | ASCII Double Quote | 34 |
| \\ | ASCII Backslash | 92 |
| | Numeric | |
| \d999 | Decimal encoding | |
| \o777 | Octal encoding | |
| \xFF | Hex encoding | |
| \uFFFF | UTF-8 encoding | |
| \UFFFFFFFF | UTF-8 encoding | |

Felix also provides raw strings, in which escapes are not recognised. This consists of the letter `r` or `R` followed by a single or triple quoted string with double quote delimiter. You cannot use the raw prefix with single quoted strings because the single quote following a letter is allowed in identifiers.

### 2.7.2 String Functions

Felix has a rich set of string functions. The most important is `char`, which returns a value of type `char`. If the string argument has zero length, the character with ordinal value 0 is returned, otherwise the first character of the string is returned. Again, following Python, Felix does not provide any character literals!

#### Comparisons

We provide the usual comparison operators.

#### Length

Felix also provides the most important function `len` which returns the length of a string. The return type is actually `size` which is a special unsigned integer type corresponding to ISO C's `size_t`.

#### Substring

Felix can fetch a substring of a string using Python like convenions.

```
var x = "Hello World";
var copied = x.[to];   // substring
var hello = x.[to 5];  // copyto
var world= x.[6 to];   // copyfrom
var ello = x.[1 to 5]; // substring
var last3 = x.[-3 to]; // substring
```

The first index is inclusive, the second exclusive. The default first position is 0, the default last position is the length of the string. If the range specified goes off either end of the string it is clipped back to the string. If the indices are out of order an empty string is returned.

A negative index is translated to by adding the string length.

The substring function is defined so it cannot fail. The name of the actual library function called by this notation is shown in the corresponding comment.

#### Index

To fetch a single character use:

```
var x = "Hello world";
var y : char = x.[1]; // subscript
```

If the index is out of range, a character with ordinal 0 is returned. Negative indices are translated by adding the string length. The index function cannot fail.

### 2.7.3 Regexps and Regdefs

Felix provides Google's Google RE2 engine for regular expressions. The basic syntax and capabilities are a subset of Perl's PCRE, only RE2 actually works correctly and performs well. RE2 does not support backreferences.

#### Basic Matching

A regexp can be compiled with the `RE2` function. Matching is done with the `Match` function. `Match` only supports a complete match. There's no searching or partial matching. Instead, just use repeated wildcards as shown.

```
var r = RE2(" *([A-Za-z_][A-Za-z0-9]*).*");
var line = "Hello World";
var maybe_subgroups = Match (r, line);
match maybe_subgroups with
| #None => println$ "No match";
```

```
  | Some a =>
    println$ "Matched " + a.1;
endmatch;
```

### Streamable matching

You may want to match more than one instance of a pattern in a string. For example, you may want to capture each word in a line of text. This can be done by iterating over a regex like the following

```
var r2 = RE2("\w+"); // try to match a word
var sentence = "Hello World";
for x in (r2, sentence) do
    println$ x.0;
done
```

### Regular definitions

Regular expressions are quoting hell. Luckily Felix provides a solution: regular definitions.

```
regdef lower = charset "abcdefghijklmnopqrstuvwxyz";
regdef upper = charset "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
regdef digit = charset "0123456789";
regdef alpha = upper | lower;
regdef cid0 = alpha | "_";
regdef cid1 = cid0 | digit;
regdef cid = cid0 cid1 *;
regdef space = " ";
regdef white = space +;
regdef integer = digit+;
```

These are some basic definitions. Note that `regdef` introduces a new syntax corresponding with the notation usually used for regular expressions.

This is called a DSSL or Domain Specific Sub-Language. Its not a DSL, because that's a complete new language, rather the *sub* suggests its an extension of normal Felix. The extension is written entirely in user space. Now to use these definitions:

```
// match an assignment statement
regdef sassign =
  white? "var" white?
  group (cid) white? "=" white?
```

```
    (group (cid) | group (integer))
    white? ";" white?
  ;

  var rstr : string = sassign.Regdef::render;
  var ra = RE2 rstr;
  var result = Match (ra, " var a = b; ");
  println$
    match result with
    | #None => "No match?"

    | Some groups =>
      "Assigned " + groups.1 +
      if groups.2 != "" then
          " from variable " + groups.2
      else
          " from integer" + groups.3
      endif
    endmatch
  ;
```

Note that the regdef kind of variable must be converted to a Perl regexp in a string form using the `render` function.

# Chapter 3

# Data types

## 3.1 Pointers &

Since variables are mutable addressable objects we can take the address of a variable:

```
var x = 1;
var px : &int = &x;
var y = *px;
px <- 42;
println$ x,y;
```

Here, the type of a pointer to `int` is written `&int`, the address of a variable `x` is written `&x`, the dereference operation applied to the pointer `px` is written `*px`.

The store operation `px <- 42` is a core operation. In particular assignment is just a shorthand for storing at the address of a variable:

```
x = 42; // means
&x <- 42;
```

Note that in principle the `&` symbol is *NOT* an operator. A variable is in fact a constant, its not variable at all. It's just that the constant is the address of a storage object. What's stored *at* the address, or *in* a variable may change, the variable itself does not.

## 3.2 Tuples

Felix has an strucurally typed product where components are accessed by position, commonly called a *tuple*. Tuples can be constructed using the non-associative n-ary comma operator and accessed by using a plain decimal integer as the projection function:

```
var x = 1,"Hello", 42.7; // type int * string * double
var i = x . 0;
var h = x . 1;
var d = x  . 2;
```

Tuple is just another name for Cartesian product. They allow you to pack several values together into a single value in such a way that you can get the components you put in out again.

We shall see tuples are vital for functions, since functions can only take a single argument. To work around this fact, we can pack multiple values together using a tuple.

### 3.2.1 Unit tuple

There is a special tuple with nothing in it called the *unit* tuple, written:

```
var u2 : 1 = ();
var u : unit = ();
```

where the alias `unit` is defined in the library by:

```
typedef unit = 1;
```

The 1 signifies that the unit type has only one value. It is a unit type.

### 3.2.2 Arrays

If all the elements of a tuple are the same type, its called an *array*. In this case the accessor index, or projection, can be an expression of `int` type.

```
var x : int ^4 = 1,2,3,4;
for var i in 0 upto x.len.int - 1 do
  println$ x . i;
done
```

You will also see the use of a low level `for` loop here, and the use of the `len` function to get the array length. The `len` function returns a value of type `size` but the loop variable `i` is an `int`, so we have to convert it.

## 3.3 Unit Sums

The type of the array in the last section is given by

```
int ^ 4 = int * int * int * int
```

This is the usual meaning of the exponential operator `^`: raising to some power $n$ means multiplication of $n$ copies.

However you may be surprised to learn that in Felix, `4` is a type, and

```
4 = 1 + 1 + 1 + 1
```

The type `1` here is also called `unit` and is just the type of the empty tuple `()`. Now, you have seen that the cartesian product type is denoted using the n-ary non-associative operator `*`, so it is not so surprising that the type denoted by the n-ary non-associative operator `+` is called a sum. So, the type `4` is a called a *unitsum* because it is the sum of units.

Values of a unit sum just represent cases. The notation is ugly:

```
var x : int ^4 = 1,2,3,4;
var third = case 2 of 4;
println$ x.third; // the *true* projection
```

especially as it's zero origin. When you use an integer array index, it has to be bounds checked at run time. However if you use a unit sum index, the check is done by the type system at compile time.

There is a very commonly used unitsum. You already know it: `bool` is nothing more than another name for type `2` and the special words `false` and `true` are just aliases for `case 0 of 2` and `case 1 of 2` respectively!

## 3.4 Records

A record is a tuple with named fields.

```
var r : (a:int, b:string) = (a=42, b="hitch-hiker);
println$ r.a, r.b;
```

Records support advanced features including row polymorphism with scoped labels.

## 3.5 Structs

A struct is a nominally typed product with named fields.

```
struct X {
  a:int;
  b:string;
  fun show() => self.b + " " + self.a.str;
};
var x = X(42, "World");
x.show;
```

Functions and procedures can be included in a record but are actually defined outside it with a curried argument named `self`. It has the type of the record for a function, and a pointer to the record for a procedure. So the above is equivalent to:

```
struct X {
  a:int;
  b:string;
};
fun show(self:X) => self.b + " " + self.a.str;
```

## 3.6 Lists

A list is a variable length sequence of values of the same type. An empty list of `int` is denoted `Empty[int]`. Given a list you can create a new one with a new element on the front using the constructor `Cons` as follows:

```
var x = Empty[int];
x = Cons (1,x);
x = Cons (2,x);
x = Cons (3,x);
var y = Cons (3, Cons (2, Cons (1, Empty[int])));
```

Of course this is messy! Here is a better way:

```
var z = list (3,2,1);
```

This method converts a tuple to a list. You can add two lists together, prepend a value, or add a value to the end of a list with the infix + operator:

```
var x = list (1,2,3);
x = 1 + x + x + 42;
```

Take care that + associates to the left and you don't accidentally add two integers together! There is a second operator you can use as well which is right associative and prepends an element to a list:

```
var x = 3 ! 2 ! 1 ! Empty[int];
x = 42 ! x;
```

You can use the `len` function to find the length of a list, and test if an element is in a list using the `in` operator:

```
var x = list (1,2,3);
assert len x in x; // 3 is in the list!
```

Lists in Felix are purely functional data structures: you cannot modify a list. All the nodes in a list are immutable, which means when you prepend an element $A$ to a list $L$, and then prepends an element $P$ to the same list $L$, the tail of the list is shared. Lists can be passed around efficiently without copying.

When some prefix of a list is no longer accessible because the function prepending the prefix returns without saving the list, the prefix elements will be removed automatically by the Felix garbage collector.

## 3.7 Pattern Matching and Union

A list is actually defined like this:

```
union list[T] =
  | Empty
  | Cons of T * list[T]
;
```

The `union` construction defines a nominally typed sum. The words `Empty` and `Cons` are called type constructors. This union is also polymorphic! This means you can make lists of any type. The first case is just an empty list. The second case, `Cons` joins together an element of type `T` and another list. In other words it creates a new list by adding a new element onto the front. A type like this is called an inductive type.

A list can be taken apart with a pattern match:

```
var x = list (3,2,1);
println$
  match x with
  | #Empty =>  "Empty"
  | Cons(v, tail) => "first element " + v.str
  endmatch
;
```

Note you have to write `#` in front of `Empty`. This is to tell Felix that you mean a constructor taking no arguments. Otherwise it will be taken as a pattern variable!

### 3.7.1 Option Type

Another commonly used union type is the option type `opt`:

```
union opt[T] =
  | None
  | Some of T
;

fun maybe_divide (num:int, denom:int) =>
  if denom == 0 then None[int]
  else Some (num/denom)
;

println$
  match maybe_divide (10,0) with
  | #None =>  "Divide by Zero"
  | Some j => "Quotient " + j.str
  endmatch
;
```

### 3.7.2 Enumerations `enum`

If all the constructors of a union have no arguments an alternate form can be used:

```
enum colour = red, green, blue; // equivalent to
union colour = red | green | blue;
```

# Chapter 4

# Algorithms

## 4.1 Functions

We have enough preliminaries now to finally introduce functions. Without further ado, here are some basic functions:

```
fun twice (x:int) : int => x + x;
fun thrice (x:int) :int  => twice x + x;
```

Functions also have a more expanded form:

```
fun trickdiv (num:int, denom:int) :int =
{
   var y = if denom == 0 then 1 else denom endif;
   return num / y;
}
```

There is a rule for functions:

**functions defined with the `fun` binder may not have observable side effects**

The rule is relaxed; that is, not enforced, to permit debugging, profiling, coverage checking, etc.

### Pattern Syntax for Functions

Functions can also be defined with an abbreviated form which merges the definition with a pattern match:

```
fun len[T] : list[T] -> int =
  | #Empty => 0
  | Cons (_, tail) => 1 + len tail
;
```

This is just shorthand for

```
fun len[T] (x:list[T]): int =>
  match x with
  | #Empty => 0
  | Cons (_, tail) => 1 + len tail
  endmatch
;
```

### 4.1.1 Calling functions

In Felix you can call a function using operator whitespace:

```
fun twice (x:int) : int => x + x;
println$ twice 42;
```

This is the usual prefix mathematical notation. In Felix, parentheses are not required around arguments to make a function call, they are just used for grouping. Some programmers also like postfix notation and you can use that too:

```
fun twice (x:int) : int => x + x;
fun thrice (x:int) :int  => twice x + x;
println$ thrice 42.twice;
```

Here, `twice` is called first, because operator `.` has a higher precedence that operator whitepsace. Both operators are left associative! Operator dot is called *reverse application* because the argument is written first, then the function to apply.

Felix also has operator dollar `$` which is a very low precedence right associative operator:

```
println$ k $ h $ g 42.f;
println ( k ( h ( g ( f 42))));
```

and there is also a left associative low precedence reverse order operator too:

```
h $ g 42.f |> k |> println;
println ( k ( h ( g ( f 42))));
```

This is called *reverse pipe application* and has an even lower precedence than forward dollar application.

There is one more application operator!

```
fun hhgttg() => 42;
h $ g #hhgttg.f |> k |> println;
```

Operator hash `#` just applies a function to the unit tuple `()`, it is often used for constant functions. It is a very high precedence operator.

It's no wonder they wanted to spacedoze the Earth to build a freeway.

### 4.1.2 Overloading

You can define two functions with the same name, this is never an error. However to apply a function with the same name as another, it must be possible to distinguish between them.

*Overloading* provides one method for doing this:

```
fun twice (x:int) => x + x;
fun twice (x:double) => x + x;
println$ twice 42, twice 42.1;
```

In the above example, the first application in the `println` argument tuple calls the first function, because the argument is of type `int`, and the second application calls the second function because the argument is of type `double`.

Selecting functions based on matching the argument and parameter types like this is called overload resolution. Note that unlike C++ there are no automatic conversions in Felix. The argument and parameter type must match exactly.

Overloading first looks in the scope of the application. If matching fails to find any candidate functions, it proceeds to the next outer level, until there are no more levels left.

#### Named Arguments

In Felix you can use named arguments. This helps to resolve overload clashes when the candidate functions have the same type. It is also useful if you forget the order of the arguments.

```
fun f(x:int, y:int) => x + y;
fun f(a:int, b:int) => a - b;
println$ f (y=1, x=2), f (a=1, b=2);
```

In fact, a function using named parameters will also accept a value of record type:

```
fun f(x:int, y:int) => x + y;
fun f(a:int, b:int) => a - b;
var xy = (y=1, x=2);
var ab = (a=1, b=2);
println$ f xy, f ab;
```

#### Default Arguments

A function can be specified with default arguments:

```
fun f(x:int=5, y:int) => x + y;
println$ f (y=1);
```

If you wish to use default arguments you must use the named parameters feature directly or provide a record argument with missing fields. Default arguments do not work with tuple arguments.

### 4.1.3 Higher Order Functions

In Felix, a function can become a first class value. This value is called a closure. Functions that accept functions as arguments, or return functions, are called higher order functions or *HOF*'s.

```
fun twice(x:int) => x + x;
fun both (x:int, y:int, g:int -> int) => g x, g y;
println$ both (3,7,twice);
```

The function `both` applies its argument `g` to both x and y, returning the pair of results. On the other hand here:

```
fun increment (x:int) : int * int -> int =
{
   fun add (y:int) => x + y;
   return add;
}
println$ increment 3 42; // 45
```

```
var add3 = increment 3;
println$ add3 42; // 45
```

the increment function returns another function which increments its argument y, when applied, by the argument x first passed to it. The variable `add3` binds the first argument x, and is another function which will add that x to its argument when applied.

The function `increment` is said to have *arity* 2 because it appears to accept two arguments. In fact, there is no such thing as a function accepting two arguments.

The function of higher arity is sometimes said to be *curried* named for theorist Howard Curry, although some prefer Indian food. If a function returns a function which is not immediately applied to an argument it is said, surprisingly, to be *partially applied*. When Felix does overload resolution it can take into account as many arguments as are provided for a curried function application. It finds candidates based on the first argument first, then tries to whittle down the list based on the next argument, and so on.

### 4.1.4 Polymorphic Functions

In Felix, you can write functions that work with a family of types.

```
fun diag[T] (x:T) => x, x;
```

This is the famous diagonal function, which makes two copies of its argument. It works for any type, so it is a polymorphic function. In general such functions can *only* do structural manipulations so they are of limited utility.

Unless, that is, they're HOFs! For example:

```
fun add3[T] (x:T, y:T, z:T, add:T * T -> T) =>
  add (x, add (y,z))
;
```

#### Overloading Polymorphic Functions

Polymorphic functions can be overloaded too. However unlike ordinary functions, more than one function might match. For example:

```
fun f[U,V] (x: U, y: V) => y,x;
fun f[W] (x:W, y:int) => y+1,x;
println$ f (2,4); // which f?
```

A function matches if the parameter type can be specialised to the argument type by substituting some type expression for each of the function's type variables. In the above case

$$\texttt{U} \rightarrow \texttt{int}$$
$$\texttt{V} \rightarrow \texttt{int}$$

causes the first function to match and

$$\texttt{W} \rightarrow \texttt{int}$$

causes the second one to match.

In this case the most specialised function is chosen if there is one. Here, the first function is clearly more general because the substitution:

$$\texttt{U} \rightarrow \texttt{W}$$
$$\texttt{V} \rightarrow \texttt{int}$$

into the type of the first function's parameter yields the second function's parameter type. Since there is no substitution in the other direction, the second function is strictly more specialised and so it is selected.

The process of finding a substiution which makes types with type variables equal is called unification. The substitution effecting equality is called a *unifier*, and the two terms rendered equal are said to be *unified*. There can be more than one unifier, a most general unifier is one which can produce all the other unifiers by some substitutions. If there is a most general unifier it is unique up to changing variable names.

### 4.1.5 Conversion Operators

Felix has a special shortcut for defining conversions:

```
typedef cart_complex = (x:double, y:double);
typedef polar_complex = (r:double, theta:double);

ctor cart_complex (z: polar_complex) =>
  (x=z.r * cos z.theta, z.r * sin z.theta)
;

ctor polar_complex (z: cart_complex) =>
  (r = sqrt (z.x.sqr + z.y.sqr), theta = tan (y / z))
;
```

```
ctor cart_complex (x:double) => (x=x,y=0.0);
ctor polar_complex (x:double) => (r=x, theta=0.0);

var z1 = cart_complex 0.5;
var z2 = polar_complex z1;
println$ z2;
```

The binder `ctor` must be followed by the name of a type. A `typedef` alias will do. The definition is expanded:

```
ctor T (x:arg) => expr;

//equivalent to:
fun _ctor_T (x:arg) : T => expr;
```

When Felix finds an application `f`$a$ where `f` is a simple name, it first tries to find a function named `f`. If a type name is found instead of a function, Felix looks again for a function named `_ctor_`$f$ where $f$ is the type name.

### 4.1.6 Apply operator

When Felix finds an application `f`$a$ where `f` is a simple name, it first tries to find a function named `f`. If a value of some type $F$ is found instead, Felix looks again for a function named `apply` with a tuple argument of type `F * A` where $A$ is the type of the argument. For example:

```
fun apply (x:string, y:string) => x + y;
println$ "Hello" " " "World";

fun apply (x:double, y:double) => x * y;
fun apply (x:int, y:double) => x * y;
var x = 2.3;
println$ 2 x + 4 x x;
```

This allows objects of any type other than functions to also act as functions in a user defined way.

## 4.2 Procedures

A *procedure* is like a function which returns no value and is allowed to have side effects.

```
proc printint(x:int)
{
   println$ "An integer " + x.str;
}
```

This is in fact syntactic sugar for

```
fun printint(x:int) : void =
{
   println$ "An integer " + x.str;
}

printit 42;
```

The pseudo type `void` is the type you use when there is no value.

Procedures with unit argument type have a special call notation, the unit argument () can be dropped:

```
proc print42()
{
   println$ "The world is coming to an end";
}

print42; // no () required
```

You're right, I hate excess parens!

## 4.3 Generators

There is one further special kind of function, called a *generator*. A generator returns a value and may have a side effect.

```
var counter = 0;
gen fresh() : int =
{
  ++counter;
  return counter;
}
println$ #fresh, #fresh; // 1,2
```

The term generator comes from the exemplar generator, namely the random number generators.

## 4.4 Control Flow

Felix has a rich control flow architecture. Here are some control flow constucts.

### 4.4.1 Conditional

Here is the expanded procedural conditional branch.

```
begin
  if cond do
    something;
  elif cond2 do
    something_more;
  else
    last; resort;
  done
end
```

There are also several short forms.

```
  if cond goto lab;
  if cond return;
```

### 4.4.2 Jumps

The usual `goto` and target label.

```
begin
  var i = 1;
  start:> // label
    call println i; // procedure call
    ++i; // increment i
    if i < 10 goto start; // conditional jump
end
```

### 4.4.3 Loops

Low level inclusive loops are flat code. Jumping in and out with `goto` is permitted. The control variable can optionally be declared in a loop, and exists in the whole containing scope.

#### Counting Loops

In this forms, the control variable may be declared by the loop if necessary. Note the control variable is accessible outside the loop!

```
for var i:int in 0 upto 10 do // inclusive
  println$ i; // procedure call
done

for i in 10 downto 0 do // inclusive
  println$ i; // procedure call
done
```

### While Loops

Here is the usual while loop and its negation the until loop.

```
// While loop
var i = 0;
while i < 10 do
  println$ i;
  ++i;
done

// until loop
until i == 0 do
  println i;
  --i; // decrement
done
```

### Index Iterators

And here are loops using iterators, note that the control variable is auto declared:

```
// inclusive subrange of int iterator
for j in 0..10 do
  println$ j;
done

// exclusive subrange of int iterator
for k in 0..<10 do
  println$ k;
done
```

### Visitors

Most data structures provide iterators for visitor loops:

```
// array iterator
for j in (1,2,3,4,5,6,7,8,9,10) do
  println$ j; // array iterator
done

// list iterator
for j in list (1,2,3,4,5,6,7,8,9,19) do
  println$ i; // list iterator
done
```

### C style

This is the fastest and most general loop:

```
for (i=0; i<10; ++i;) do // C style with addition of trailing semicolon
  println$ i;
done
```

Iterator based loops are implemented using yielding generators as explained in the next section.

### Breaks

Felix allows modification of loop control flow as follows:

```
var sum = 0;
iloop: for i in 0 upto 10 do
  var prod = 0;
  jioop: for j in 0 upto 10 do
    if j % 2 == 0 continue jloop;
    prod = prod * j;
    if prod > 5000 break jloop;
  done
  if sum < 20000 redo iloop;
done
```

The `break`, `continue` and `redo` statements cause early exit, early continuation, or complete restarting of the `for`, `forall`, `while`, or `until` loop specified by the loop name. Loops can be named with an identifier followed by a colon `:` immediately preceeding the loop.

### 4.4.4 Statement Groups

#### Do group

Statements can be grouped using the `do .. done` constuction. The body of a loop is a single statement, the do group is used to present the parser many statements considered as one. Jumps into and out of a do group are allowed, they do not represent a scope.

#### Block function

There is another group called a *block* which does represent a scope: the `begin .. end` construction. You can jump out of a block, but you cannot jump in, and you cannot return.

There is a reason for this:

```
var doit = { println$ "Hello"; println$ " World"; };
call doit ();
```

The expression in curly braces `{}` is an anonymous procedure of type $1 \to 0$. To call this procedure, we must apply it to a value of type unit, and there is only one such value, namely ().

The `begin..end` construction is just an anonymous procedure which is then applied to unit immediately, so that

```
{ println$ "Hello"; println$ " World"; } ();
```

is equivalent to

```
{ println$ "Hello"; println$ " World"; };
```

which is equivalent to

```
begin println$ "Hello"; println$ " World"; end
```

only we don't need the trailing semi-colon `;`.

So because this is a procedure, a `return` statement would just return from the anonymous procedure, and not the containing procedure. To work around this you can name the procedure from which you wish to return using a `return from` statement:

```
proc outer (x:int)
{
   {
      println$ "Inner";
      if x > 0 do return from outer; done
   };
   println$ "Negative";
}
```

Do not try to return from a procedure which is not active, all hell will break loose.

## 4.5 Yielding Generator

Here is a way to write and use what is called an *yielding generator*. This is a generator utilising a `yield` statement.

```
begin
  gen down (var start:int) () =
  {
    for i in start downto 0 do
      yield i;
    done
    return i;
  }

  var it = down 10;
  var x = #it;
  while x >= 0 do
    println x;
    x = #it;
  done
end
```

A yielding generator must be a function of type $1 \to T$ for some type T. In this case `down 10` has that type. A closure over the generator must be assigned to a variable to hold the state, named `it` in the example.

The closured stored in the variable is then called by applying it to the unit value (), written like `#it` here because I hate parens. This causes the generator to run until it yields a value or returns.

The next call to the generator causes it to proceed from where it left off after a yield, or to do the terminating return again if necessary. So yielding generators can always be called infinitely producing a stream of values.

The caller and the generator are coroutine which swap control back and forth as required separately by each, that is, they are both masters. The generator is a so-called *push master* and the client is a *pull* master.

We will see shortly that Felix has different techniques to implement fully general coroutines.

# Chapter 5

# Concepts: type classes

Felix provides a way of systematically organising functions and operators known as *type classes*, modelled on the construction of the same name in Haskell.

## 5.1 Monomorphic Classes

A nonpolymorphic `class` is just a closed namespace:

```
class X
{
   fun f(x:int) => x + 1;
   fun g(x:int) => x * x;
}

println$ X::f 42, X::g 42
```

As well as using explicit qualification, you can open a class:

```
open X;
fun g(x:int) => "Hello " + x.str;
println$ f 42, g 42; // X::f, root::g
```

When a class is opened it occupies a shadow lookup space just behind the space into which it is opened. Therefore the local `g` above hides `X::g`. The global namespace is called `root` in case you need explicit qualification.

When you open a class inside another, the symbols of the former are available for definition of the latter, but they are not available to clients of the latter. Thus opening is not transititive.

```
class A { fun f(x:int) => x; }
class B { open A; fun g (x:int) => f x; }
type-error println$ B::f 42; // no f in B
```

If you want to include the symbols of one class in another, use `inherit` instead:

```
class A { fun f(x:int) => x; }
class B { inherit A; fun g (x:int) => f x; }
println$ B::f 42; // OK, f is in B
```

## 5.2 Polymorphic Classes

When a class is polymorphic things get more interesteding. Polymorphic classes are basically collections of template declarations and definitions.

```
class AddGroup[T]
{
   virtual fun == : T * T -> bool;
   fun != (x:T, y:T) => not (x == t);

   virtual fun + : T * T -> T;
   virtual fun zero : 1 -> T;
   virtual fun neg : T -> T;
   fun - (x:T, Y:T) : T =>  x + neg y;

   axiom negation (x:T): x + neg x == zero();

   private fun mulby (x:T, count: int) : T =
   {
     var res = zero();
     for i in 0 ..< count do res = res + x; done
     return res;
   }
   fun * (x:T, count: int) : T => mulby (x,count);
}
```

The above class specifies an additive group. The specification is entirely abstract because it is given entirely in terms of functions. The four `virtual` functions are undefined here and form the *basis* of the abstraction.

The inequality, subtraction, and multiplication functions shown are defined in terms of the abstract basis. The `mulby` function is marked as `private` and is only a local helper function, it will not be exported as part of the class interface.

The `axiom` is a semantic constraint on the behaviour of the functions.

As you see the class above has "holes" in it, namely the undefined virtual functions. Here is how to provide definitions:

```
struct MyInt { v:int; } // to avoid conflict with std lib

instance AddGroup[MyInt]
{
  fun == (x:MyInt, y:MyInt) => x.v == y.v;
  fun zero () => MyInt (0);
  fun + (x:MyInt, y:MyInt) => MyInt (x.v + y.v);
  fun - (x:MyInt) => MyInt (-x.v);
}
```

To use this, it is convenient to open our class for our type:

```
open AddGroup[MyInt];

var x = MyInt(1);
var y = MyInt(42);
var z = x + -y * 3 - MyInt(74) != AddGroup[MyInt]::zero();
```

Note that we used explicit qualification for the `zero` function although it is not necessary in this case. In general, functions with the same name taking a unit argument have to be explicitly qualified to distinguish them, there is no type distinction in the argument for overload resolution to use.

### 5.2.1 Two Phase Binding

Felix uses a lookup model which provides two phases of binding. In the first phase, all binding is polymorphic. Bindings to virtual functions may occur in this phase.

In the second phase, the whole program is monomorphised, eliminating all type variables. This phase is sometimes called *instantiation*. When an application or call of a virtual function is seen, Felix searches for an instantce matching the monomorphic type parameters and replaces the virtual call with a concrete one, or bugs out if no instance is found.

### 5.2.2 Polymorphic Recursion

You should note that instance functions may themselves contain virtual calls, and this permits the instantiator to implement polymorphic recursion. That is, a limited form of *second order polymorphism* is available in Felix, but only via type classes. Polymorphic recursions must either bottom out on monomorphic types,

or reduce to monomorphic recursion after a finite number of steps. The current compiler does not check this in advance, and instead just uses an expansion limit.

# Chapter 6

# Dynamic Objects

Felix provides a dynamic kind of object and interface with syntax similar to Java.

```
interface Employee_t {
  name: 1 -> string;
  service: 1 -> double;
  base_pay: 1-> double;
  pay : double  -> double;
  set_base_pay : double -> 0;
}

object Employee (
  ename: string,
  eservice: double,
  abase_pay: double
)
implements Employee_t =
{
  var ebase_pay = abase_pay;
  method fun name () => ename;
  method fun service () => eservice;
  method fun base_pay () => ebase_pay;
  method proc set_base_pay (v:double) => ebase_pay = v;
  method fun pay (extra: double) =>
    extra + service * ebase_pay / 52.0;
}
```

To use objects:

```
proc show (p: Employee_t) {
  println$ "Employee " + #(p.name) +
    " earns " + p.pay 0.0 + " per week"
  ;
}

var joe  = Employee ("joe", 1.5, 90_000.0);
show joe;
```

Dynamic objects do not introduce any new functionality to the Felix language. An `interface` is nothing more than an alias for a record type:

```
typedef Employee_t = (
  name: 1 -> string,
  service: 1 -> double,
  base_pay: 1 -> double,
  pay: double -> double,
  set_base_pay: double -> 0,
);
```

An object is precisely a function which returns a value of the specified interface type by returning a record of closures of all the specified methods:

```
fun Employee (
  ename: string,
  eservice: double,
  abase_pay: double
) : Employee_t =
{
  var ebase_pay = abase_pay;

  fun name () => ename;
  fun service () => eservice;
  fun base_pay () => ebase_pay;
  proc set_base_pay (v:double) => ebase_pay = v;
  fun pay (extra: double) =>
    extra + service * ebase_pay / 52.0;

  return (
    name = name,
    service = service,
    base_pay = base_pay,
    pay = pay,
    set_base_pay = set_base_pay
```

```
    );
}
```

Felix dynamic objects provide encapsulation based on functional abstraction. In particular to use a value of an object type the client only requires knowledge of the interface.

The `interface` clause of the object specification can be left out in the same way as the return type of a function can be left out. Similarly, objects can be anonymous, in the same way functions can be anonymous:

```
var Cplx = object (x:double, y:double) =
  {
    method fun Real () => x;
    method fun Imag () => y;
    method fun Arg () => atan2 (y,x);
    method fun Mod () => sqrt (x * x + y * y);
  }
;
var z = Cplx (1.0,1.0);
println$
  "Complex " + #(z.Real) + #(z.Imag)+"i = " +
  #(z.Mod)+"e^" + #(z.Arg)+"i"
;
```

Note carefully that `Cplx` is a closure over the object constructor, that is, it is a function returning a record of methods, it has to be applied to an argument to actually produce the object.

Because object types are just record types, we can use dynamic techniques are row polymorphism to provide very strong statically typed object management. The system is extremely powerful and much more expressive than C++ or Java objects; of course the downside of such power is the usual ability to shoot yourself in the foot.

## 6.1 Plugins

Plugins are a special kind of object for which the implementation is dynamically loaded at run time from a shared library. Plugins can also be preloaded so the plugin interface can be used for a statically linked program without modification.

Table 6.1: Summary of Types

| | | |
|---|---|---|
| **Primitive** | | |
| integers | | |
| floats | | |
| strings | | |
| regexps | | |
| **Nominal** | | |
| unions | | |
| structs | | |
| cstructs | | |
| **Structural Products** | | |
| unit | 1 | |
| tuples | $T_0$*$T_2$*...*$T_{n-1}$ | |
| records | $(f_0 : T_0, f_1 : T_1, ..., f_{n-1} : T_{n-1})$ | |
| **Structural Sums** | | |
| void | 0 | |
| bool | 2 | |
| unitsums | 99 | |
| sums | $T_0$+$T_2$+...+$T_{n-1}$ | |
| variants | | |
| **Structural Exponentials** | | |
| arrays | $T$^$N$ | |
| functions | $D \to C$ | |
| cfunctions | $D$-->$C$ | |
| pointers | &$T$ | |
| **Library** | | |
| option | `opt[T]=Some of T \|None` | option type |
| list | `list[T]=Empty\|Cons of (T*list[T])` | functional list |
| varray | varray[T] | bounded stable variable length array |
| darray | darray[T] | unbounded reallocable variable length array |
| sarray | sparse array | |
| bsarray | bounded sparse array | |
| strdict | string dictionary | |
| judy array | Judy1, JudyL arrays | |
| schannel | synchronous channels | |
| pchannel | asynchronous channels | |

Table 6.2: Summary of Binders

| Binders | |
|---|---|
| var | declare and initialise a variable |
| val | name a value |
| def | compound assignment, variable binder, and value binder |
| fun | define a function |
| cfun | define a function with C function type |
| gen | define a generator |
| ctor | define a conversion function |
| proc | define a procedure or fibre |
| cproc | define a procedure with C function type |
| struct | define a struct |
| cstruct | model an existing C struct |
| union | define a discriminated union |
| enum | define an enumeration |
| class | define a type class |
| interface | define an object interface |
| object | define an object constructor |
| typedef | define an alias for a type |
| syntax | define new syntax |
| type..new | define an abstract type |
| Directives | |
| include | include a library file |
| inherit | inherit symbols into a class |
| open | open a class |
| requires | specify requirements of C bindings |
| open syntax | augment parser with new grammar |
| comment | write a comment |
| todo | specify future code |
| private | do not export symbol from class |
| rename | rename or import a symbol |
| macro val | define a macro |
| forall | generate a table |

Table 6.3: Summary of Statements

| Control | |
|---|---|
| goto | unconditional control transfer |
| goto-indirect | goto address stored in LABEL variable |
| return | return from procedure or function |
| return from | return from specified procedure |
| call | call a procedure |
| jump | jump to procedure with arguments |
| yield | suspend generator emitting value |
| halt | stop program |
| if..do..elif..else..done | standard conditional |
| match..with | pattern matching |
| with | execute statements with temporary binders |
| trace | trace execution on UDP port |
| type-error | document compilation error |
| `_svc` | Felix scheduler service call |
| `spawn_fthread` | schedule fibre |
| `spawn_pthread` | schedule pre-emptive thread |
| `read` | read channel |
| `write` | write channel |
| `branch-and-link` | low level control exchange |
| loops | |
| while | execute whilst condition true |
| until | execute until condition true |
| for | execute repeatedly with bounded control variable |
| break | exit specified loop |
| continue | start next iteration of specified loop |
| redo | restart current iterator of specified loop |
| Groups | |
| do..done | statement group |
| begin..end | scoped statement group |
| perform | single statement |
| Assignment | |
| `p<-a;` | |
| `v=a;` | |
| `v1<->v2;` | |
| `v+=a;` | |
| `v-=a;` | |
| `v*=a;` | |
| `v/=a;` | |
| `v%=a;` | |
| `v&=a;` | |
| `v\|=a;` | |
| `v^=a;` | |
| `v<<=a;` | |
| `v>>=a;` | |
| `++v;` | |
| `--v;` | |
| Plugins | |
| static-link-symbol | |
| static-link-plugin | |
| export | |
| export python | |

# Chapter 7

# Interfacing

## 7.1 Embedding C++

Felix was specifically designed to allow almost seamless lifting of C++ code. Indeed most of the standard library types are simply types lifted from C++. The provides ABI compatibility with C++ and hence C, although syntactic compatibility is sacrificed in favour of a superior type system.

### 7.1.1 Lifting types

The `type` statement is used to lift types from C++.

```
type myint = "int";
var x:myint;

type metres = "double";
var m: metres;

type mystring="::std::basic_string<char>"
  requires header '#include <string>'
;
var s: mystring;
```

Template classes can be lifted to polymorphic types too, the notation `?1`, `?2`, etc is used to associate the type variables.

```
type vector[T] = "vector<?1>"
  requires header '#include <vector>'
;
var v : vector[int]; // vector<int>
```

Note that if the type is defined in a C or C++ header you must ensure the header is included in the code Felix generates as shown. The `requires` clause will be explained shortly.

If you need to lift a lot of non polymorphic types at once there is a special shortcut using `ctypes` binder:

```
ctypes int,long,short,double;
```

### 7.1.2 Lifting Values

Of course, types are useless without values, so Felix provides a way to lift expressions with the `const` binder:

```
const one : int = "1";
const pi : double = "22.0/7";
var z = pi; // rough approx
```

The Felix type of the lifted value must be given. Note the value can be an arbitrary expression.

Note: it is also possible to lift literals by extending the Felix grammar. The Felix grammar is actually defined in the standard library in user space, so extending it can be done in Felix. However it is beyond the scope of this quick introduction to explain the somewhat arcane details.

### 7.1.3 Lifting Functions

Lifting types is all very well, but they're not very useful unless they can be manipulated. C++ functions can be lifted to functions and procedures like this:

```
fun addup3 : int * int * int -> int = "$1+$2+$3";
fun sin : double -> double = "sin($1)"
  requires header '#include <math.h>'
;
proc coutit[T] : T = "cout << $1;"
  requires header "#include <iostream>"
;
```

Here is a more sophisticated example:

```
type vector[T] = "::std::vector<?1>"
  requires header "#include <vector>"
;
```

```
proc push_back[T] : vector[T] * T = "$1.push_back($2);";
fun len[T]: vector[T] -> int = "$1.size()";

type viter[T] = "::std::vector<?1>::const_iterator"
  requires header "#include <vector>"
;
fun deref[T] : viter[T] -> T = "*$1";
fun next[T] : viter[T] -> viter[T] = "$1+1";
fun !=[T] : viter[T] * viter[T] -> bool = "$1!=$2";

fun begin[T] : vector[T] -> viter[T] = "$1.begin()";
fun end[T] : vector[T] -> viter[T] = "$1.end()";

var v : vector[int]; // C++ default init to empty
begin
  var l = list (1,2,3,4); // Felix list
  for elt in l do push_back(v, elt); done
end

println$ "Length = " + v.len.str;

// print STL vector
var it = v.begin;
while it != v.end do
  println$ deref it;
  it = next it;
done
```

There are some things to note here!

First, you may note the functions `begin` and `end` used here. This is bad practice, the names `stl_begin` and `stl_end` would be better. But the question is: why does it work at all? Aren't `begin` and `end` keywords?

The answer is simple: no, they can't be keywords because Felix has no keywords! You may have thought that there were far too many keywords in Felix but now you know the truth. There are none. Zero. Zilch. Felix uses a sophisticated extended generalised scannerless LR parser and the grammar only recognises certain identifiers as special in contexts where they might occur.

The second important thing to note here is that we have cheated and some care is needed to avoid a serious issue. The Felix `vector` we have defined is polymorphic and can be used with any type where the generated C++ type would work, but it must not be used with a Felix type which uses the garbage collector! Unions and nonunit sum types and Felix pointers in particular usually require garbage collection, and that includes Felix `list` type.

If you put such a type into a C++ vector, the Felix garbage collector will not know it is there, and objects pointed at may appear to be unreachable and be deleted when they're stored in the vector. It is safe to put these types into a vector or other C++ data structure if the pointers can be regarded as weak; that is, if they're also stored somewhere else which ensures they remain reachable whilst the vector is holding them.

The third thing to note is that we have broken a very important protocol in this example: we have modified a value, namely the vector `v`. Although C++ allows modifying values, Felix does not. Felix only allows modification of objects. In particular although `v` in the example is an addressable store and therefore an object, all modifications to objects must be done via the address of the object, that is, by using a pointer. Felix has no concept of lvalue like C++, and no reference types: we use pointers instead.

The problem is that we have used the abstraction provided by the Felix lifting constructs to hide the underlying data type. The compiler cannot know that the type you lifted was not in fact a pointer to an object and the mutators lifted operating through the pointer to modify the object it points to. That would be allowed!

Now let us fix it:

```
proc push_back[T] : &vector[T] * T = "$1.push_back($2);";
fun begin[T] : &vector[T] -> viter[T] = "$1.begin()";
fun end[T] : &vector[T] -> viter[T] = "$1.end()";
```

There! We have to adjust the corresponding usage of course. Note carefully the iterator itself is indeed an abstract pointer. So operations on the vector via the iterators obtained do NOT require any adjustment! Although none are present in this example.

### 7.1.4 Fixed Insertions

You can put C++ code directly into Felix. Statements can be added with `cstmt` statement. An argument may optionally be added. This is basically an anonymous Felix procedure written in C++.

Arbitrary expressions can be inserted with with `cexpr..endcexpr`, also optionally taking an argument.

The short form `cvar` lifts a variable.

```
cstmt """
#include <iostream>
static int x = 1;
""";
```

```
type INT = "int";
fun + :INT * INT -> INT = "$1 + $2";

fun two (): INT =>
  cvar [INT] x + cexpr[INT] 'x' endcexpr
;
cstmt "::std::cout <<$1<<endl;" (two());
```

### 7.1.5 Floating Insertions

Floating insertions allow you to conditionally include C++ code based on usage. For example:

```
type vector[T] = "::std::vector[?1]"
  requires header "#include <vector>"
;

proc push_back[T] (&vector[T] :pv, v:T) =>
  "$1->push_back($2);"
;

var x : vector[int];
push_back (&x,1);
push_back (&x,2);
```

The `requires` clause creates a dependency, the `header` specifier says it depends on a floating insertion, which if required will end up near the top of the generated C++ header file for your program. This ensures C++ `::std::vector` is in scope.

The requirement is conditional on the type being used, as it is in the `push_back` procedure. On the other hand, since it is used in that procedure, it isn't necessary to also place a requirement on the use of the procedure. Calling `push_back` requires the `push_back` procedure which in turn requires the `vector` type, which in turn requires the floating insertion.

Floating insertions can be named, and can also have their own requirements. All the standard C, C++, and posix headers are named in `std/c/c_headers` and `std/c/cxx_headers`.

```
header iostream = "#include <iostream>";
type vector[T] = "::std::vector[?1]" requires iostream;
```

## 7.2 Foreign Libraries

Here is a quick guide to embedding a foreign C library `mylib` with functions `mycal1` and `mycal2` on `double`. The first thing of course is to make a binding file:

```
// file mylib.flx
class MyLib
{
  requires package "mylib";
  fun mycal1: double -> double = "mycal1($1)";
  fun mycal2: double -> double = "mycal2($1)";
}
```

Requirements specified in a `class` are inherited by all C bindings in the class. Here is a new requirement on a `package`, and there is no requirement on a header file!

Now you must make another file, the package descriptor, this one is for Linux:

```
Name: mylib
includes: '"mylib.h"'
cflags: -Imydir
provides_dlib: -lmylib
provides_slib: -lmylib
```

The file must be named `mylib.fpc`, and placed in a configuration database directory the `flx` processor searches. The specified inclusion (`#include "mylib.h"`) will be put in the generated C++ code automatically. Furthermore the include file search flag `-Imydir` will be added to the C++ compiler compilation command line, and the linker flag `-lmylib` added to the compiler linkage command line. In this case the `provides_dlib` implies the library file `libmylib.so` and `provides_slib` implies `libmylib.a`.

This mechanism works on OSX and Windows, and with gcc, clang, cl, and other compilers. The objective is to ensure the Felix code is platform independent, and shove all the platform dependent stuff into a single place. Here's a windows package:

```
Name: mylib
includes: '"mylib.h"'
cflags: /Imydir
provides_dlib: /DEFAULTLIB:mylib
provides_slib: /DEFAULTLIB:mylib
```

Felix uses a tool called `flx_pkgconfig` to query and manage packages. The tool itself is platform independent and the databases are also language neutral.

# Part II

# Functional Programming

Functional programming is the art of encoding a system of types and functions.

# Chapter 8

# Theory

In set-theoretic mathematics a function is triple consisting of a specified set $D$ called the *domain* a set $C$ call the *codomain* and a method $f$ of assigning an element $c \in C$ of the codomain to each element $d \in D$ of the domain, written:

$$f : D \to C$$

where

$$d \mapsto c$$

which we also write in equational form as

$$f(d) = c$$

In programming, the challenge is to find a suitable representation, or representations, of the sets as a type, and to encode the functions so as to faithfully model the mathematics. In particular if type `D` represents set $D$, type `C` represents set $C$, then for all elements $d$ of $D$ if value `d` represents $d$ and `f` is an encoding of function $f$ then if $c = f(d)$ then `f(d)` represents $c$. Pictorially, where $r$ is the representation mapping, we require that the following diagram commutes

$$\begin{array}{ccc} D & \xrightarrow{f} & C \\ \downarrow{\scriptstyle r} & & \downarrow{\scriptstyle r} \\ \texttt{D} & \xrightarrow{\texttt{f}} & \texttt{C} \end{array}$$

meaning

$$r \cdot \texttt{f} = f \cdot r$$

*Category Theory* provides a number of tools to assist in the constructions of function encodings, provide a suitable programming language is available, and

given a fixed choice of data types. The most important of these tools is *functional decomposition*. Given two function $f : A \to B$ and $g : B \to C$ we can construct the composiite $h = f \cdot g$.

Note we use the midposition dot operator for reverse composition, meaning the first function written is applied to an argument first, then the second one. Do not confuse this with the lower dot which represents reverse application:

$$(f \cdot g)a = (a.f).g = a.(f \cdot g) = (g \circ f)a$$

Now, given some function $h : A \to C$ it is natural to ask if it can be broken up into two functions $f : A \to B$ and $g : B \to C$ for some $B$ such $h = f \cdot g$. If so, this is called a *serial decomposition* of the function.

In Felix code:

```
fun f: A -> B;
fun g: B -> C;
fun h: A -> C => f \cdot g;
```

Another tool is parallel composition:

```
fun f: A -> X;
fun g: B -> Y;
fun fg = A * B -> X * Y = f * g;
```

More of these tools are discussed in the section on Categorical Programming.

# Chapter 9

# Functional Basics

We will begin by considering functional programming in the *applicative* style. In this style at the basic level we have two kinds of entities: *values* and *functions*. We say that applying a function to a value returns a result value to which another function may be applied. A composite of function applications is called an *expression*.

The following program consists of both procedural and functional code:

```
var x = 1.0;
var y = 2.0;
var hyp = sqrt (x^2 + y^2);
println$ hyp;
```

## 9.1 Higher Order Functions (HOF)

Ok so, what if I wanted to add up the squares of the numbers in the list? Is there a way to avoid rewriting the whole thing? The answer is yes.

Let's start with simple tail recursive function:

```
fun addup (x: list[int]) =
{
  fun aux (rest: list[int], acc: int) =>
    match rest with
    | #Empty => rest
    | Cons (head, tail) => aux (tail, acc + head)
    endmatch
  ;
```

```
  return aux (x, 0);
}
```

The non-recursive function `addup` contains a nested tail recursive function `aux`.

Then a real HOF:

```
fun fold
(
  binop: int * int -> int,
  init: int,
  lst: list[int]
) =
{
  fun aux (rest: list[int], acc: int) =>
    match rest with
    | #Empty => rest
    | Cons (head, tail) => aux (tail, binop (acc, head))
    endmatch
  ;
  return aux (lst, init);
}
```

Fold works with any binary operator on `int`. For example:

```
lst = list(1,2,3);
fun addition (x:int, y:int) => x + y;
assert addup lst == fold (addition, 0, lst);
```

There is another way to write the folding expression:

```
assert addup lst == fold ((fun (x:int, y:int)=>x+y), 0, lst);
```

using an unnamed, or anonymous function, sometimes called a lambda.

So what happens if we want to make the sum of the square roots of a list of `int`?

It should work, right, if we change the signature of the binary operator to

```
double * int -> double
```

and pass an appropriate function. The first component of the argument is intended to hold the accumulated sum.

What if the list were of `string` and we wanted to get the complete length? Then we'd use

```
int * string -> int
```

## 9.2 Polymorphism

Can we generalise, so we only have one function?

Sure. That's what parametric polymorphism is for. Using $R$ for the result type and $V$ for the list element type:

```
fun fold[V,R]
(
  binop: R * V -> R,
  init: R,
  lst: list[V]
) =
{
  fun aux (rest: list[V], acc: R) =>
    match rest with
    | #Empty => rest
    | Cons (head, tail) => aux (tail, binop (acc, head))
    endmatch
  ;
  return aux (lst, init);
}
```

Now we can write:

```
println$
  fold [string,int]
  (
    (fun (acc:int, s:string)=>acc + s.len.int),
    0,
    (list ("Hello", "World"))
  )
;
```

This is all very nice! Even better we can write:

```
println$
  fold
  (
```

```
      (fun (acc:int, s:string)=>acc + s.len.int),
      0,
      (list ("Hello", "World"))
    )
  ;
```

because the instances of the type variables required can be deduced from the arguments.

However our implementation has a slight ugliness: we first define a function, then return an application of it, which consists of two statements, and requires the expanded form of function definition.

## 9.3 Let/in construction

Can we do it as a one liner, using only the simplified form? The answer is yes, using a `let-in` expression:

```
fun fold[V,R]
(
  binop: R * V -> R,
  init:R,
  lst:list[V]
) =>
  let fun aux (rest: lst, acc: R) =>
    match rest with
    | #Empty => rest
    | Cons (head, tail) => aux (tail, binop (acc, head))
    endmatch
  in
  aux (lst, init)
;
```

The fold above utilises what is called a *type schema*. This is a limited kind of polymorphism also called first order polymorphism, or simply *templates*. Type schemes are limited to having type variables universally quantifed on the outside left, and can be instantiated by replacing the type variable with an actual type. At run time, there are no type variables about. You cannot have a polymorphic function at run time. So

```
// NO polymorphic function closures in Felix!
type-error var x = fold[int, string];
```

There is another way to write this, using a method called currying, in a form called curry form. This is the more conventional form in some languages.

In functional programming languages, functions are first class, meaning you can pass them into another function, and return them from a function. Consider this:

```
fun fold[V,R] (binop: R * V -> R) =
{
    fun A(init:R) =
    {
      fun B(lst:list[V]) =>
        let fun aux (rest: list[V], acc: R) =>
          match rest with
          | #Empty => acc
          | Cons (head, tail) => aux (tail, binop (acc, head))
          endmatch
        in
        aux (lst, init);
      return B;
   }
   return A;
}
```

What is the type of this function? Well,

```
B: list[V]-> R
```

and since A returns B, its type is:

```
A: R -> (list[V]-> R)
```

and since fold returns A its type is

```
fold: list[V] -> (R -> (list[V]-> R))
```

How would we use this? Well:

```
println$
  (
    (
      (
         fold
           (fun (acc:int, s:string)=>acc + s.len.int))
```

```
      )
      0
    )
    (list ("Hello", "World"))
  )
;
```

Note that the -> function operator is right associative we don't need the parens so we can write this as:

```
fold: list[V] -> R -> list[V]-> R
```

getting rid of parens, and, since application (the whitespace operator!) is left associative, we can write:

```
println$
  fold
    (fun (acc: int, s: string) => acc + s.len.int)
    0
    (list ("Hello", "World"))
;
```

again getting rid of parens. The form:

```
  fold fn init alist
```

is the curried form of the call we wrote in tuple form like:

```
  fold (fn, init, alist)
```

Curried form has an advantage:

```
fun string_lengths (x:list[string]) =>
   fold
    (fun (acc:int, s:string)=>acc + s.len.int)
    0
    x
;

println ("Hello", "World").list.string_lengths; // 10
```

In the function, we simply apply the fold to the first three arguments, which returns a function of one argument, a list of strings.

Curried form is so useful, there is syntactic sugar for writing functions in this form:

```
fun fold[R, V]
  (binop: R * V -> R)
  (init: R)
  (lst: list[V])
=>
  let fun aux (rest: list[V], acc: R) =>
    match rest with
    | #Empty => acc
    | Cons (head, tail) => aux (tail, binop (acc, head))
    endmatch
  in
  aux (lst, init)
;
```

# Chapter 10

# Recursion

We shall begin our more serious exploration of Felix with functional programming techniques. Perhaps the most important is recursion. A recursive function is one which may call itself, directly or indirectly. Recursive functions should usually be reentrant to ensure each fresh invocation of the function has its own state variables.

We recognise two kinds of recursion, *static recursion* and *dynamic recursion*.

## 10.1 Static Recursion

Static recursion occurs when there is a loop in the static call graph.

It may be *direct*, in which a function explicitly calls itself, which is also known as *self-recursion*.

```
// self recursion
fun factorial (n:int) =>
  if n <= 1 then 1
  else n * factorial (n - 1)
;
```

Next we show *indirect* recursion using two functions:

```
// indirect recursion
fun factorial1 (n:int) =>
  if n <= 1 then 1
  else n * factorial2 (n - 1)
;
fun factorial2 (n:int) =>
```

```
    if n <= 1 then 1
    else n * factorial1 (n - 1)
;
```

The example is contrived of course. Indirect recursion between two siblings can always be eliminted by inlining:

```
// indirect recursion reduced to direct
// recursion by inlining
fun factorial1 (n:int) =>
  if n <= 1 then 1
  else n *
    (
      let m = n - 1 in
      if m <= 1 then 1
      else
        m * factorial1 (m - 1)
    )
;
```

If the recursion is between a parent child pair, it can be eliminated by inlining the child.

Using these two steps, a succession of inlining operations can reduce any indirect recursion to a direct recursion. The reason is due to the visibility constraint on calls: any function may call its children, itself or any sibling, its parent, or any function its parent may call.

Although ancestoral calls cannot be directly eliminated by inlining, there must be a call from the candidate functions parent to it, or one of its siblings, leading to the inlining of the candidate and moving it towards the root.

## 10.2 List Examples

We shall begin with some basic routines on lists. We will first write the "obvious" routines to use as specifications. Later we will have to look at their performance.

### 10.2.1 Reversing a list

To reverse a list, we run down the list constructing a new one. At the end, we return the result.

Our routine takes two arguments, a list `inp` and a list `out` onto which we must prepend the reversed copy of `inp`.

```
fun prepend_reversed[T] (inp:list[T]) (out: list[T]) =>
  match inp with
  | #Empty => out
  | Cons (head, tail) =>
    prepend_reversed tail (Cons (head, out))
  endmatch
;
```

We can use this routine to reverse a list.

```
fun rev[T] (a: list[T]) => prepend_reversed a Empty[T];
```

### 10.2.2 Join Lists

Now suppose we wish to prepend a list $a$ onto a list $b$. How would we do that? Well it's easy:

```
fun join[T] (a: list[T], b: list[T]) =>
  prepend_reversed (rev a) b
;
```

### 10.2.3 Map a List

This routine creates a new list with each element a function of the old value.

```
fun rev_map[U,T] (f:T->U) (inp:list[T]) : list[U]=>
  match inp with
  | #Empty => Empty[U]
  | Cons (head, tail) => Cons (f head, rev_map f tail)
  endmatch
;

fun map[U,T] (f:T->U) (inp:list[T]) : list[U] =>
  rev$ rev_map f inp
;
```

### 10.2.4 Left fold a List

Folds fold the elements of a list of $T$ into a single value of type $U$ by applying a function of type $U \times T \to U$. Lists have two folds, one starting at the top or left, and one starting at the bottom.

```
fun fold_left[U,T] (f:U -> T -> U) (acc:U) (inp:list[T]) : U =>
  match inp with
  | #Empty => acc
  | Cons (head, tail) => fold_left f (f acc head) tail
  endmatch
;
```

Notice the order of the arguments, and see how in the second branch the head element is folded into the accummulator to form an argument for the recursive call.

### 10.2.5 Right fold a List

Here is the right fold:

```
fun fold_right[U,T] (f:T -> U -> U) (inp:list[T]) (acc:U): U =>
  match inp with
  | #Empty => acc
  | Cons (head, tail) => f (fold_right f acc tail) head
  endmatch
;
```

Notice again the order of the arguments and see how in the second branch the recursion produces an argument for the function f.

### 10.2.6 Concatenate List of Lists

Using our fold we can concatenate a list of lists into a single list.

```
fun cat[T] (x:list[list[T]]):list[T] =>
  match x with
  | #Empty => Empty[T]
  | Cons(h,t) => fold_left join of (list[T]) h t
  endmatch
;
```

### 10.2.7 Filter

Filter selects all the elements of a list satisfying a predicate.

```
fun filter[T] (P:T -> bool) (x:list[T]) : list[T] =>
  match inp with
  | #Empty => Empty[T]
```

```
    | Cons(h,t) =>
      if P(h) then Cons(h, filter P t)
      else filter P t
    endmatch
;
```

### 10.2.8 Prefix and Suffix

For a list $l$ of length $n$, `take` $k$ $l$ returns the head sublist of length $min(n, k)$.

```
fun take[T] (k:int) (lst:list[T]) : list[T] =>
  if k <= 0 then Empty[T]
  else
    match lst with
      | #Empty => Empty[T]
      | Cons(x,xs) => Cons(x, take[T] (k - 1) xs)
    endmatch
  endif
;
```

For a list $l$ of length $n$, `drop` $k$ $l$ returns the list with first $min(n, k)$ elements removed.

```
  fun drop[T] (k:int) (lst:list[T]) : list[T] =>
    if k <= 0 then lst
    else
      match lst with
        | #Empty => list[T] ()
        | Cons(x,xs) => drop (k - 1) xs
    endif
  ;
```

## 10.3 Tail Recursion

### 10.3.1 Tail Calls

A final value returned by a function is said to be in *tail position*.

If an expression in tail position is an application, it is known as a *tail call*. In expanded form, in this function for example:

```
fun f(x:int) = {
  var a = x + y;
```

```
  return g ( h (a + 1));
}
```

The tail call is to $g$, the functional part of the return expression's application. Not all returns have a tail call, in this example:

```
fun f(x:int) = {
  var a = x + 1;
  var b = g ( h (a + 1));
  return b;
}
```

a variable is returned. This function can be rewritten without changing the semantics to the previous one, showing that whether a call is in tail position is a syntactic property. It is also language dependent. For example in Felix:

```
fun f (c:bool) = {
  return
     if c then f 1 else g 1 endif
  ;
}
```

the calls to $f$ and $g$ are in tail position by specification. Similarly, in

```
fun f (c:bool)  = {
  return
     match c with
     | false => f 1
     | true => g 1
     endmatch
  ;
}
```

the calls to $f$ and $g$ are in tail position. However consider this function:

```
fun ifthenelse (c:bool, a:int, b:int) = {
  return
     match c with
     | false => a
     | true => b
     endmatch
  ;
}
```

then in this function:

```
fun f (c:bool) = {
  return
     ifthenelse (c, f 1, g 1)
  ;
}
```

it is the call to `ifthenelse` which is in tail position.

### 10.3.2 Tail Call Optimisation

Tail calls are important because they lead to an optimisation. If we save the current function's return addres on the stack and jump to the tail function, on completion that function will jump to the return address, popping it off the stack, and then the current function will jump to its return address popping it off the stack. It is more efficient to just jump to the tail function without saving the return address, so that when it returns by popping the return address off the stack and jumping to it, it is returning from itself and the caller.

### 10.3.3 Tail Recursion

If the tail call is to the function itself, the function is said, somewhat ambiguously, to be *tail recursive*. The ambiguity is seen here in Ackerman's famous function:

```
fun ack(x:int,y:int):int =>
  if x == 0 then y + 1
  elif y == 0 then ack(x - 1, 1)
  else ack(x - 1, ack(x, y - 1))
  endif
;
```

Here there are recursive calls, two in tail position and one not.

Tail recursive calls may lead to another improvement. Instead of pushing the argument onto the stack, we can just assign the argument to our parameter. This is safe because the parameter is not accessed after the call returns, since it is a tail call. This means the recursion can be performed as a loop using a mutable variable as the parameter. If a function calls itself, and all calls are in tail position, the function is said to be *tail recursive*.

For example this function is tail recursive:

```
fun f (x:int, n:int) = {
  if n <= 0 return x;
  return f (x * n, n - 1);
}
```

and can be optimised to:

```
fun f (var x:int, var n:int) = {
again:
  if n <= 0 return x;
  n, x = x * n, n - 1;
  goto again;
}
```

eliminating the recursion entirely. Note carefully the parallel assignment to the parameter components, it is equivalent to

```
  var tmp1 = x * n;
  var tmp2 = n - 1;
  n = tmp1;
  x = tmp2;
```

### 10.3.4 Refactoring for tail recursion

Let us observe again our routine `prepend_reversed`:

```
fun prepend_reversed[T] (inp:list[T]) (out: list[T]) =>
  match inp with
  | #Empty => out
  | Cons (head, tail) =>
    prepend_reversed tail (Cons (head, out))
  endmatch
;
```

This routine is tail recursive, since the recursive call is in tail position. However consider our `rev_map` routine:

```
fun rev_map[U,T] (f:T->U) (inp:list[T]) : list[U]=>
  match inp with
  | #Empty => Empty[U]
  | Cons (head, tail) => Cons (f head, rev_map f tail)
  endmatch
;
```

Clearly this is not tail recursive because the recursive call to `rev_map` is not in tail position. Can we recode this routine to make it tail recursive? The answer is yes:

```
fun trrmap[U,T]
  (f:T->U)
  (inp:list[T])
  (out:list[U])
: list[U] =>
  match inp with
  | #Empty => out
  | Cons (head, tail) => trmap tail (Cons ((f head ), out))
  endmatch
;

fun rev_map[U,T] (f:T->U) (inp:list[T]) =>
  trrmap f inp Empty[U]
;
```

Clearly now, the recursive call to `trrmap` is in tail position. The trick used here is very common: we add an extra parameter to the routine to hold the temporary result.

Examine again the `fold_left` routine, the recursion occurs on the branch:

```
  | Cons(h,t) => fold_left join of (list[T]) h t
```

and is clearly in tail position. However, this is not the case for the `fold_right` routine:

```
  | Cons (head, tail) => f (fold_right f acc tail) head
```

Can we fix it? The answer is yes, technically. Consider

```
fun fold_right[U,T]
  (f:T -> U -> U)
  (inp:list[T])
  (acc:U)
: U  =>
  let fun f' (acc:U) (elt:T) => f elt acc in
  fold_left f' acc (rev list)
;
```

Although this routine is not even recursive, it is calling a two tail recursive routines: `fold_left` and `rev`.

The question we must ask here is whether this recoding is worthwhile. Although only a constant amount of machine stack is required for saving return addresses and passing parameters, we have had to make a temporary reversed list on the heap, costing an amount of time and space proportional to the list length.

On the other hand the non-tail recursive routine only requires a single stack of pairs consisting of a return address and pointer to a list node: we also need to save the accumlator of course, however it can be a single variable. Whether or not the underlying compilation engine actually uses a single variable for the result is probably architecture and type dependent: clearly if the fold result is an integer a machine register might be used.

In any case it simply isn't clear whether the tail recursive formulation is faster or slower, and even a naive measurement may be difficult since in a garbage collected language the freeing of the list may be delayed, making a measurement problematic. In Felix for example, a microtest may fail because by default Felix does not clean up memory on program termination.

Indeed, we must emphasize now that this doubt about the performance of programs encoded largely using functional programming techniques is one of the major negative aspects of those techniques. In general with any modern compiler and modern processor performance is not only hard to predict, it is also quite hard to measure, but with functional programming the situation is even more difficult.

Functional programming does offer an advantage for certain classes of problems where the functional style make it easier to reason about the correctness of the encoding. This is primarily due to a property known as *referential transparency*, however that property is only possessed by encodings which use recursion and immutable values in place of mutable variables. The price is a loss of ability to reason about performance.

I personally think it is also arguable whether functional code is easier to reason about in practice, rather than academic papers. Indeed, the applicative functional model based on lambda calculus has serious issues in respect of evaluation strategy. We will say more about this later.

In this sense Felix excels because it also permits procedural programming. Although fraught with difficulties mixing functional and imperatives styles can provide a more balanced program in the sense that the difficulty of reasoning about correctness of imperative code can be balanced with greater confidence in its good performance, where it matters.

## 10.4 Generalised Folds: Catamorphisms

Unsurprisingly, folds can be mechanically written for many data types. A more precise formulation in category theory can be given, and is explained by the notion of a catamorphism. We will come back to this later.

Let us develop a tree fold. Our tree is binary and carries data at each node: both leaf and branch nodes:

```
union tree[T] =
  | Leaf of T
  | Branch of T * tree[T] * tree[T]
;

fun fun fold[R,V]
  (binop: R * V -> R)
  (init: R)
  (tr: tree[V])
=>
  match tr with
  | Leaf v => binop (init, v)
  | Branch (v, l, r) =>
    let left = fold binop init l in
    let right = fold binop left r in
    binop (right, v)
;
```

Note this formulation isn't tail recursive: it's not possible to visit tree nodes without remembering where you're up to.

## 10.5 Maps

Actually maps are easier than folds! A map function simply copies the structure of its argument, applying a unary operator to each element. Here is a list map:

```
fun map[R,V] (f: V -> R) (lst: list[V]) : list[R] =>
  match lst with
  | #Empty => Empty[R]
  | Cons (head, tail) => Cons (f head, map f tail)
  endmatch
;
```

You will note this implementation is not tail recursive! We will leave it as an exercise for the reader to develop a tail recursive formulation.

Now we will do a tree map:

```
fun map[R,V] (f: V -> R) (tr: tree[V]) : tree[R] =>
  match tr with
  | Leaf v => Leaf (f v)
  | Branch (v, l, r) => Branch (f v, map f l, map f r)
  endmatch
;
```

As you can guess, writing maps is rather boring. This is because a map produces an exact copy of the original data structure with only the data type changed. An operation which *preserves structure* has a technical name: it is called a functor.

We will provide a way to encode maps in a type class now, but before we can do this we need to observe a technical detail!

Although our list and tree are considered in the abstract as data functors, in fact they are type schema, or indexed types. They're not actually functors. So to make them functors we will have to do this:

```
typedef list_f = fun (T:TYPE): TYPE => list[T];
typedef tree_f = fun (T:TYPE): TYPE => tree[T];
```

This is our first example of *higher kinded programming* which is sometimes loosely called meta-programming. The functions we have shown takes a type and produces another type. For example:

```
typedef intlist = list_f int;
```

Note that `list_f` is not yet a true functor. It is a map from types to types which is the object part of a functor definition. The collection of types forms a category which is named `TYPE` in Felix.

Now we can rewrite our maps like this:

```
class Functor[F:TYPE->TYPE]
{
  virtual fun fmap[V,R]: (V->R) -> (F V -> F R);
}

instance Functor[list_f]
{
  fun fmap[V,R] (f:V->T) (lst: list_f V) : list_f R=>
    match lst with
    | #Empty => Empty[R]
```

```
      | Cons (head, tail) => Cons (f head, fmap f tail)
      endmatch
  ;
}

instance Functor[tree_f]
{
  fun fmap[V,R] (f: V -> R) (tr: tree[V]) : tree[R] =>
    match lst with
    | Leaf v => Leaf (f v)
    | Branch (v, l, r) => Branch (f v, fmap f l, fmap f r)
    endmatch
  ;
}
```

The pair consisting of the type mapping `list_f` and an associate function mapping `Functor[list_f]::fmap` are a functor, because the two constraints are satisfied. First, its obvious that for any type $T$, an identity map:

```
fun id[T](x:T) => x;
```

is taken to the function

```
Functor[list_f]::fmap id
```

This just makes a copy of the list. Secondly, given any two function

```
fun f[A,B]: A -> B;
fun g[B,C]: B -> C;
```

then:

```
(Functor[list_f]::fmap g) \circ (Functor[list_f]::fmap f)
```

has the same semantics as:

```
Functor[list_f]::fmap (g \circ f)
```

In other words, mapping a list using $f$ and then mapping it again using $g$ produces the same final result as mapping it by the composite function $g \circ f$ in one

go. This requirement is more than just a rule, however. Its a crucial optimisation called deforestation or *fusion*. Although not always the case, applying the composite function once to each element to build a new list is probably faster than building a temporary list, building another from it, and the deleteing the temporary list.

Such semantic rules for functors are important enough to encode them in the type class:

```
class Functor[F:TYPE->TYPE]
{
  virtual fun fmap[V,R]: (V->R) -> (F V -> F R);

  axiom identity_preserving[A, B with Eq[A]] (f:A->B) (id:A->A) (x:A):
    id x == x implies fmap id (f x) == fmap f x
  ;

  reduce structure_preserving[A,B,C] (f:A->B, g:B->C):
    (fmap g) \circ (fmap f) =>
    fmap (g \circ f)
  ;
}
```

The identity rules is a bit tricky, it says that for every function $f : A \to B$ and function $id : A \to A$ for which for all $x \in A$ we have $fx = x$, then the fmap of the $id$ function applied to the value $f$ maps $x$ to is the same as the value that the $fmap$ of function $f$ gives. We need this long winded exposition because we cannot directly assert that two functions are equal.

The current version of Felix can check the axioms against test data, however it cannot mechanically implement the reduction rule to effect an optimisation.

# Part III

# Procedural Programming

# Chapter 11

# Procedural Basics

In Felix, procedures are subroutines which return control but no value, they should have side effects, directly or indirectly.

```
proc hello()
{
   println "Hello";
}

proc sayit (s:string)
{
  println$ "Say " + s;
}
hello;
sayit "Hello";
```

The type of a procedure is written `T -> 0` where $T$ is the type of the argument. The zero indicates no value is returned, it is an alias for `void`.

## 11.1 Variables and Pointers

The core concept of procedural programming revolves not around observable behaviour, but the use of non-observable storage locations called variables which can be modified to contain time dependent values.

There is therefore an intrinsic relation between control flow and the state of a program's variables, and this relation is the heart of procedural programming.

In Felix, functional programming involves values, whilst procedural programming involves variables as well. The relation between the two is explicit: values

are immutable, so functional code ignoring variables is pure. On the other hand, a variable is a symbolic name for the address of an object, which is a contiguous region of store.

In principle, in Felix, assignment is effected by a store operator:

```
var x = 1; // a variable
var px = &x; // the variable address
px <- 42; // the store-at mutator
```

Variables are addressable, which means a pointer to the variable can be obtained. There is no assignment, rather a procedure is used to store a value at an address.

At first glance the philosophical model has no semantic impact, however we will show that the model is very powerful. In particular, in a functional setting, a value of a product type is characterised by projections.

Consider now:

```
val x = (a=1, b=2); // record value
val bv = x . b; // b is the projection
var y = x; // put value in variable!
val px = &y; // address of object
val pa = px . a; // address of a subobject
pa <- 42;
&y . a <- 42; // equivalent by substitution
y . a = 42; // equivalent by sugar
```

What is important here is that all product projections from product type $P$ to component type $C$, having type $P->C$, are overloaded so that there is also a projection with the same name of type $\&P->\&C$.

In paricular note that you cannot "take the address" of a subcomponent of a value stored in a variable! In fact, the `&` is not an operator at all. Rather the spelling `&y` is the true name of the variable, and the plain usage like `y` is really sugar for `*&y`, that is, it is really an auto-dereference of the pointer associated with the variables type and storage address.

Felix allows construction of objects on the heap and locally: heap allocations return a pointer directly. Local objects, represented by a variable, can be silently replaced by heap objects and vice versa.

The most noteworthy example of the utility of the concept is the way Felix handles arrays.

```
// a tuple of elements, all the same type, is an array
var x = 1,2,3,4;
```

```
for i in 0..<4 do
  &x . i <- x . i + 1;
done
```

Here the projection is denoted by an expression, not merely a constant. Felix array values are immutable, however you can modify an array object as shown.

# Chapter 12

# Iterators

Stuff.

# Chapter 13

# Dynamic Objects

Stuff.

# Chapter 14

# Reactive Programming

Stuff.

# Part IV

# Active Programming

# Chapter 15

# Fibres

Felix provides synchronous (cooperative) multi-tasking by way of *fibres* or *f-threads*. The procedure to launch a fibre is

```
spawn_fthread: 1 -> 0
```

For example you can do this:

```
spawn_fthread { println$ "Hello World"; };
println$ "Fibre spawned";
```

However the following may not do what you first expect:

```
var fts = varray[1->0] 4uz;
for var i in 0 upto 3 do
  push_back (fts, { println$ i; });
done
for j in 0 upto 3 do
  spawn_fthread fts.j;
done
```

This is because Felix captures variables by address not value. So it will print "3" 4 times, because that is the value of variable $i$ at the time the fibre runs.

To actually capture the current value do this instead:

```
var fts = varray[1->0] ();
noinline proc pi (var k:int) () { println$ k; }
for var i in 0 upto 3 do
```

```
    push_back (fts, pi i);
  done
  for var j in 0 upto 3 do
    spawn_fthread fts.j;
  done
```

The `noinline` is essential. Remember the notation is a short hand for

```
  fun pi (var k:int) = {
    proc aux () { println$ k; }
    return aux;
  }
```

The function `pi` will return the procedure `aux` however if it is inlined so that parameter $k$ is replaced by $i$, then aux will print $i$ in every instance: this is lazy evaluation. Using `noinline` ensures a closure is formed in the loop with k assigned to the current value of $i$. Then `aux` refers to the variable $k$ which is different for every instance of the closure.

Another solution is to use recursion:

```
  var fts = varray[1->0] 4uz;
  proc spawnem (var i:int) {
    push_back (fts, {println$ i; } );
    if i < 3 call spawnem (i+1);
  }
  spawnem 0;
  for var j in 0 upto 3 do
    spawn_fthread fts.j;
  done
```

Even though this is clearly a case of tail recursion, Felix does *not* perform the tail recursion optimisation. The compiler knows there is a reference to the procedure frame which escapes the call's lifetime so the frame must be preserved and not reused.

# Chapter 16

# Synchronous Channels

Fthreads are designed to cooperate by using synchronous channels or *schannels*.

# Chapter 17

# Pipelines

Stuff.

# Chapter 18

# Coroutines

Stuff.

# Chapter 19

# Continuations

Stuff.

# Chapter 20

# Symmetric Lambda Calculus

Stuff.

# Part V

# Concurrency

# Chapter 21

# Asynchronous Events

Stuff.

## 21.1 Timers

Stuff.

## 21.2 Socket I/O

Stuff.

# Chapter 22

# Pre-emption

Stuff.

## 22.1 Pthreads

Stuff.

### 22.1.1 Designing a Thread Pool

Let us suppose we wish to divide the task of multiplying a matrix up into 4 pieces, so one thread calculates a quarter of the result. One important property of this division is that the parts are entirely independent.

A pool of threads is useful to solve problems which can be split into such pieces.

```
interface thread_pool_t
{
  stop : 1 -> 0;
  schedule: 1 -> 0;
}

object thread_pool (maxthreads: int)
  implements thread_pool_t =
{
  union Action = | Run of 1->0 | Stop;
  var ictrl, octrl = #mk_pchannel_pair[Action];
  var maxthreads = 4;

  proc pool_thread ()
  {
```

```
      next:>
        var cmd : Action = read ctrl_chan;
        match cmd with
        | Run p -> p(); goto next;
        | Stop -> ;
        endmatch;
      }

      proc start() =>
        for i in 0..<maxthreads
          call spawn_pthread pool_thread;

      method proc stop() =>
        for i in 0..<maxthreads
          call write(octrl, Stop);

      method proc schedule (job: 1 -> 0) =>
         write (octrl, Run job);

      start;
    }
    var pool = thread_pool 4;
```

It is important to know that Felix pthreads are detached, they have no identity and can't be joined. This is the correct way to do threads! The single pchannel used here is a perfect mechanism for synchronisation. All the pthreads try to fetch the next job. The first one will block waiting for a job to be sent on the channel. The second one will block waiting for the first one. On the other hand the master thread sheduling the jobs will block until a job being scheduled is dispatched to some pool thread.

The pool is assigned to a global variable and starts automatically if it is used. If it is not use the Felix compiler will optimise away the variable and the pool will not start.

The stop method must be called exactly once to bring down the threads otherwise the Felix program cannot terminate.

You may wonder, how can we find out if scheduled jobs are completed? Here is a correct answer:

```
    interface job_t (p:1->0)
    {
       wait: 1 -> 0;
    }
```

```
object job (p:1->0) implements job_t =
{
  union job_status_t = Finished;
  var istatus, ostatus = #mk_iopchannel_pair[job_status_t];

  proc job (p: 1-> 0) =>
    { p; spawn_pthread { write(ostatus, Finished); }; };

  proc start(p:1->0) =>
     pool.schedule (job p);

  method proc wait () =>
    match read istatus with
    | #Finished => ;
    endmatch;

  start p;
}
```

Note carefully that the completion notification is issued by a spawned secondary thread! If the pool job were to issue the notification, the pool would block up until the completion status was acknowledged. There must be a better way! And there is: atomics, discussed next. However we must first note carefully the core of the issue: pchannels are synchronisation vehicles. A data transmission guarrantees one reader and writer will block until the exchange is complete.

It is crucial to understand deeply that every thread runs its own private clock, and there is by default **no correlation at all between the private clocks of threads**. In particular this means it is nonsense to even speak of an event on one thread occuring before or after an event on another unless the clocks are synchronised. The only assurance one has is that each thread has a monotonic increasing clock.

So perhaps you are temped to say, "Well if E1 occurs in thread P1, and then E2 occurs, then if X1 occurs in thread P2, then if X1 is before E1, it is also before E2".

This is utter nonsense: you are assuming some central global clock. There is no such clock. There is no fault in the deduction made in the above statement, the statement is not well formed in the first place because it tries to make the conclusion based on the assumption an event in one thread must occur before or after an event in another. This is nonsense. The only shared concept of time is one based on synchronisation events. Such an event locks the clocks together, that is, it creates an association between two events which then allows one to use the private monotonicity to make an ordering argument about subsequent and previous events.

For example in P1 if A occurs before synchronisation event Y, and if in P2, B occurs after synchronisation event Y, then one may deduce A occurs before B.

Note that spawning a thread does NOT create a synchronisation event! The spawned thread may never start, meaning it may not start until the spawner has terminated. Or it may start and complete, before the spawn command itself completes.

Pchannel I/O operations are heavyweight synchronisation events: they guarantee clocks are in agreement for both parties of the exchange.

### 22.1.2 Atomics

Atomic variables provide lightweight synchronisation events. This means they only guaranetee synchronisation of one of the two participating parties clocks.

Let us supposed an atomic flag is clear, and thread P1 sets it. Thread P2 will do some work and then wait until the flag is set before proceeding.

What is known? Thread P2 knows that all the events of thread P1 that had to be complete before the flag is set are complete, so it knows all of these events are before any events of its own that follow the retrieval of a set flag. However thread P1 knows nothing about thread P1.

### 22.1.3 Locks

A *mutual exclusion lock* or *mutex* for short, is a mechanism by which a thread may perform a sequence of memory reads and writes such that any other thread using the same lock and respecting the locking protocol, is excluded from access to the store being accessed, so that the other thread will either see none, or all, of the writes performed.

The protocol usually consists of a thread setting the lock, performing some memory operations which should be regarded as atomic, and then releasing the lock.

Technically, there are no ordering constraints on other regions of memory than those protected by the lock. Unfortunately, most mutex do not permit specification of the region to be protected, therefore, the whole of memory is protected. This means after the lock is released all other threads will observe all writes previously performed by the locking thread.

```
    var share1 = 0;
    var share2 = 0;
    var m = #mutex;

    var clock = #Faio::mk_alarm_clock;

    proc writer () =>
```

```
    for i in 0..<10 do
      m.lock;
      ++share1;
      ++share2;
      m.release;
      sleep(clock, 1.0);
    done
  ;

  proc reader() =>
    while true do
      m.lock;
      var r1 = share1;
      var r = share1 + 256 * share2;
      m.release;
      println$ r;
      if r1 >= 10 break;
    done
  ;

  spawn_pthread writer;
  spawn_pthread writer;
  spawn_pthread reader;
```

In this simple example it is implicit that `share1` and `share2` are protected by the mutex `m`. The lock-access/modify-release protocol use by all threads ensures access and modification is serialised so that the reader thread cannot see `share1` and `share2` except when they're equal. The value of these variables is indeterminate outside of the dynamic scope of the lock-release events.

### 22.1.4 Barriers

Stuff.

### 22.1.5 Concurrently Statement

# Chapter 23

# Parallelism

Stuff.

## 23.1 Thread Pool

Stuff.

## 23.2 Parallel For Loop

Stuff.

# Part VI

# Categorical Programming

# Chapter 24

# Category Theory

Stuff.

# Chapter 25

# Symmetric Categorical Language

Stuff.

# Chapter 26

# Meta-programming

Stuff.

# Chapter 27

# Polyadicity: The Holy Grail

Stuff.