Introduction to Felix

John Skaller

December 2, 2015

Contents

Chapter 1

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`.

1.1 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.

1.2 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 1.1.

1.2.1 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.

1.2.2 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.

Table 1.1: 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`	

1.3 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.

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`.

1.3.1 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!

1.4 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!

1.5 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 poimt 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`.

1.5.1 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.

1.6 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.

1.6.1 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.

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.

1.6.2 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.

1.7 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.

1.7.1 Unit tuple

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

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

It is a unit type.

1.7.2 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.

1.8 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!

1.9 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.

1.10 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;
```

1.11 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.

1.12 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
;
```

1.12.1 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
;
```

1.13 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 xnum / 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
;
```

1.13.1 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.

1.13.2 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.

1.13.3 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, gy;
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, but 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 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 find candidates based on the first argument first, then tries to whittle down the list based on the next argument, and so on.

1.13.4 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

$$\mathtt{U} \to \mathtt{int}$$
$$\mathtt{V} \to \mathtt{int}$$

causes the first function to match and

$$\mathtt{W} \to \mathtt{int}$$

causes the second one to match.

will cause both functions to match the argument type exactly. In this case the most specialised function is chosen if there is one. Here, the first function is clearly more general because the substitution:

$$\mathtt{U} \to \mathtt{W}$$
$$\mathtt{V} \to \mathtt{int}$$

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.

1.14 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!

1.15 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.

1.16 Control Flow

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

1.16.1 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;
```

1.16.2 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
```

1.16.3 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.

```
begin
  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
```

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

```
begin
  var i = 0;
  while i < 10 do
    println$ i;
    ++i;
  done
  until i == 0 do
    println i;
    --i; // decrement
  done
end
```

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 j in 0..<10 do
    println$ j;
  done

  // 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
  for (i=0; i<10; ++i;) do // C style
    println$ i;
  done
end
```

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

1.16.4 Statement Groups

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.

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"l; };
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"l; } ();
```

is equivalent to

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

which is equivalent to

```
begin println$ "Hello"; println$ " World"l; 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.

1.17 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 2

Functional Programming

We shall begin our more serious exploration of Felix with functional programming techniques.

```
fun inner_strlist (x: list[int]) =>
  match x with
  // Emnpty list
  | #Empty => ""

  // One element list
  | Cons (head, #Empty) => str head

  // Two element list
  | Cons (head, Cons (second, #Empty)) => str head ", " + str second

  // More that two elements
  | Cons (head, tail) => str head ", " + inner_strlist tail
  endmatch
;

fun strlist (x:list[int]) => "list (" + inner_strlist x + ")";

println$ strlist (list (1,2,3));
```

Here, the function `inner_strlist` is recursive. It matches the list so it is empty it produces an empty string. If there is only one element, it produces the string representation of that int.

If the list contains two elements, it produces a string containing the string representation of the two integers separated by a comma. Otherwise it produces a string representation of the head, followed by a comma, followed by a string representation of the tail. The tail must contain at least two elements, otherwise the pattern match would not reach that case.

However his function is not tail recursive! Lets rewrite it so it is:

```
fun inner_strlist (x: list[int], result:string) =>
  match x with
  // Emnpty list
  | #Empty => ""

  // One element list
  | Cons (head, #Empty) => str head

  // Two element list
  | Cons (head, Cons (second, #Empty)) => str head ", " + str second

  // More that two elements
  | Cons (head, tail) => inner_strlist (tail, str head ", ")
  endmatch
;
```

A *tail call* is the final call in a function. Tail calls can be optimised from a call followed by a return to just a jump. This avoids pushing the return address, then returning to that address from the called procedure, then immediately returning to the callers return address. Instead, the called procedure just returns to the callers return address directly.

In particular, if the tail call is call to self, then by assigning the call argument to the parameter, we then just jump to the start of the function. In other words, we can optimise tail recursion to a simple loop. This saves more than the space of the return address on the stack, it saves the space required by all the local variables as well. We can reuse these variables, since the old values cannot be accessed.

Here is another, simpler tail recursive function:

```
fun addup (x: list[int]) =
{
  fun aux (rest: list, 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`.

Ok so, what if I wanted to add up the squares of the numbers in the list? Is there a way to avoid that?

The answer is yes:

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

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

```
fun addition (acc:int,y:elt) => x + y;
assert addup x == fold (addition, 0, x);
\end(minted}

There is another way to write the folding expression:

\begin{minted}{felix}
assert addup x == fold (fun (acc:int, y:elt)=>x+y), 0, x);
```

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
\end{felix}

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

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


\begin{minted}{felix}
int * string -> int
\end{felix}

Can we generalise, so we only have one function?

Sure. That's what parametric polymorphism is for.


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

Now we can write:

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

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

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

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

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

What is the type of this function? Well,

```
B:list[ElementType]-> ResultType
```

and since A returns B, its type is:

```
A: ResultType -> (list[ElementType]-> ResultType)
```

and since fold returns A its type is

```
fold: list[ElementType] -> (ResultType -> (list[ElementType]-> ResultType))
```

How would we use this? Well:

```
println$
  (
    (
      (
         fold
           (fun (acc:int, s:string)=>acc + len s))
      )
      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[ElementType] -> ResultType -> list[ElementType]-> ResultType
```

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

```
println$
  fold
    (fun (acc:int, s:string)=>acc + len s))
    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 + len s))
    0
;

var l1 = list ("Hello", "World");
var l2 = list ("Felix", "Rocks");
println$ string_lengths l1, string_lengths l2;
```

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[ResultType, ElementType]
  (binop: ResultType * ElementType -> ResultType)
  (init:ResultType)
  (x:list[ElementType])
=>
  let fun aux (rest: list, acc: int) =>
    match rest with
    | #Empty => rest
    | Cons (head, tail) => aux (tail, binop (acc,head))
    endmatch
  in
  aux x init
;
```