Introduction to Felix

John Skaller

November 27, 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 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. 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 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`.

1.2.1 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!

1.3 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!

1.4 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 == 0.0;
```

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

Similarly, division has a higher precedence than multiplication!

1.5 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.

1.5.1 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 0123456789ABCDEFabcdef. 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.

1.5.2 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 s 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.

1.6 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 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.

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
;
```

1.8 Functions

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

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

1.9 Procedures

Stuff.

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
  // Emmpty 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
;
```