

The FARM
Felix Annotated Reference Manual

John Skaller

April 8, 2020

Contents

1	Introduction	15
I	Scripting	16
2	Language Features	18
3	The flavour of Felix	19
3.1	Simple calculations	19
3.2	Yielding Generators	20
3.3	Fibres	20
3.4	Circuits	21
4	Variable Definitions	24
4.1	The <code>var</code> binder	24
4.1.1	The <code>var</code> statement	24
4.1.2	Semantic rules	25
4.1.3	Multiple variables	25
4.1.4	Var expressions	25
4.1.5	Addressing	26
4.1.6	Referencing	27
4.1.7	Object Concept	27
4.1.8	Hoisting and lookup	27
4.1.9	<code>var</code> parameters	28
4.2	The <code>val</code> binder	29
4.2.1	The <code>val</code> statement	29
4.2.2	Multiple values	30
4.2.3	Val expressions	30
4.2.4	Pattern variables	30
4.2.5	<code>let</code> expressions	31
4.2.6	<code>val</code> parameters	31
4.2.7	<code>val</code> semantics	32
4.3	<code>def</code> statement	32
4.3.1	Uniquely typed components	33

5	Assignment Statements	35
5.1	Basic Assignment	35
5.2	Read-modify-write operations	36
5.3	Swap operator	37
5.4	Pre and post increment and decrement	37
6	Control Flow Statements	38
6.1	The <code>goto</code> statement and label <code>prefix</code>	38
6.1.1	Non-local gotos	38
6.1.2	Label values	39
6.1.3	Exchange of control	40
6.2	Conditional Statements	40
6.2.1	<code>if/goto</code>	40
6.2.2	<code>if/return</code>	40
6.2.3	<code>if/call</code>	40
6.2.4	<code>if/perform</code>	41
6.2.5	<code>if/do/elif/else/done</code>	41
6.3	<code>halt</code>	42
6.4	<code>try/catch/endtry</code>	42
6.5	<code>match/endmatch</code>	42
6.6	<code>chainmatch/ormatch</code>	42
6.7	<code>call</code>	43
6.8	procedure return	44
6.9	return from	44
6.10	<code>jump</code>	44
6.11	<code>calcc</code>	44
6.12	function return	45
6.13	<code>yield</code>	45
6.14	<code>call_with_trap</code> and <code>throw_continuation</code>	46
6.15	loops	47
6.15.1	C like loop	47
6.15.2	<code>for/in/upto/downto</code>	47
6.15.3	while loop	48
6.15.4	until loop	48
6.15.5	Control Flow modifiers	48
6.15.6	Parallel for loop	49
6.16	Loops over iterators	49
6.16.1	Recursive for loop	50
6.16.2	<code>match/in</code>	51
6.17	Assertions	51
6.17.1	<code>assert</code>	51
6.17.2	<code>axiom</code>	51
6.17.3	<code>lemma</code>	52
6.17.4	<code>theorem</code>	52
6.17.5	<code>reduce</code>	52
6.18	<code>invariant</code>	53

6.19	code	53
6.20	noreturn code	53
6.21	Service call	53
6.22	with/do/done	54
6.23	do/done	54
6.24	begin/end	55
7	Expressions	56
7.1	Priority levels	56
7.2	Chain forms	57
7.2.1	Pattern let	57
7.2.2	Function let	57
7.2.3	Match chain	58
7.2.4	conditional chain	58
7.3	Alternate conditional chain	58
7.4	Dollar application	58
7.5	Pipe application	59
7.6	Tuple cons constructor	59
7.7	N-ary tuple constructor	59
7.8	Logical implication	59
7.9	Logical disjunction	59
7.10	Logical conjunction	60
7.11	Logical negation	60
7.12	Comparisons	61
7.13	Name temporary	61
7.14	Schannel pipe operators	61
7.15	Right Arrows	61
7.16	Case literals	62
7.17	Bitwise or	63
7.18	Bitwise exclusive or	63
7.19	Bitwise and	63
7.20	Bitwise shifts	63
7.21	Addition	63
7.22	Subtraction	63
7.23	Multiplication	64
7.24	Division operators	64
7.25	Prefix operators	64
7.26	Fortran exponentiation	64
7.27	Felix exponentiation	64
7.28	Function composition	65
7.29	Dereference	65
7.29.1	Operator new	65
7.30	Whitespace application	65
7.30.1	General	65
7.30.2	Caseno operator	66
7.30.3	Likelyhood	66

7.31	Coercion operator	66
7.32	Suffixed name	66
7.33	Factors	66
7.33.1	Subscript	66
7.33.2	Substring	67
7.33.3	Copyfrom	67
7.33.4	Copyto	67
7.33.5	Reverse application	67
7.33.6	Reverse application with deref	67
7.33.7	Reverse application with addressing	67
7.33.8	Unit application	68
7.33.9	Addressing	68
7.33.10	C pointer	68
7.33.11	Macro freezer	68
7.33.12	Pattern variable	68
7.33.13	Parser argument	69
7.34	Qualified name	69
8	Atoms	70
8.1	Record expression	70
8.2	Alternate record expression	70
8.3	Variant type	70
8.4	Wildcard pattern	71
8.5	Ellipsis	71
8.6	Truth constants	71
8.7	callback expression	71
8.8	Lazy expression	71
8.9	Sequencing	71
8.10	Procedure of unit.	72
8.11	Grouping	72
8.12	Object extension	72
8.13	Conditional expression	72
9	Pattern Matching	73
10	Definitions	75
10.1	Functions	75
10.2	Pre- and post-conditions	76
10.3	Higher order functions	77
10.4	Procedures	77
10.5	Generators	78
10.5.1	Yielding Generators	79
10.6	Conversion Operators	80
10.7	Special function <code>apply</code>	80
10.8	Dynanic Objects	81

II	Gross Structure	82
11	Program structure	83
11.1	Parse Unit	83
11.2	AST	83
11.3	Include Directive	84
11.3.1	Syntax	84
11.3.2	Effect	84
11.4	Grammar	84
12	Parsing	86
12.1	Bootstrap Parser Grammar	87
12.1.1	Parser Processing	87
12.1.2	Scheme Actions	88
12.1.3	Spawning Choices	89
12.1.4	No keywords	89
12.1.5	Predefined Variables	90
12.1.6	SCHEME statement	90
12.1.7	STMT statement	90
12.1.8	syntax statement	91
12.1.9	requires statement	91
12.1.10	Priority statement	92
12.1.11	Non-terminal definition	93
12.1.12	Preprocessing productions	93
12.1.13	Grammar Macros	94
12.1.14	regdef statement	95
12.1.15	literal statement	96
12.1.16	open syntax statement	97
12.2	Felix Grammar syntax	97
13	Modules	98
13.1	Special procedure <code>flx_main</code>	98
13.2	Libraries	99
14	Lexicology	100
14.1	Comments	100
14.1.1	C++ comments	100
14.1.2	Nested C comments	100
14.2	Layout	101
14.3	File inclusion	101
14.4	fdoc files	101
14.4.1	Uncomments	101
14.4.2	#line directive	101
14.5	#! directive	102
14.6	Identifiers	102
14.6.1	Plain Identifiers	103

14.6.2 TeX Identifiers	103
14.7 Operator Identifiers	103
14.7.1 Special identifiers	103
14.8 Boolean Literals	104
14.9 Integer Literals	104
14.10 Floating point literals	105
14.11 String like literals	105
14.11.1 Standard string literals	105
14.11.2 Raw strings	105
14.11.3 Null terminated strings	106
14.11.4 Perl interpolation strings	106
14.11.5 C format strings	106
15 Macro processing	111
15.1 Macro val	111
15.2 Macro for	111
15.3 Constant folding and conditional compilation	111
 III Lookup	 113
16 Names	114
16.1 Kinds of Names	114
16.1.1 Simple Names	114
16.1.2 Simple Indexed Names	114
16.1.3 Unqualified Names	114
16.1.4 Qualified Names	115
16.1.5 Special name <code>root</code>	115
16.1.6 Unsuffix Name	115
16.1.7 Suffixed Name	115
16.2 Symbol Tables	115
16.2.1 Symbol definition table	115
16.2.2 Name lookup table	116
16.2.3 Views	116
16.2.4 Name lookup environments	117
16.3 Overload Resolution	117
16.4 Lookup	119
16.4.1 Unqualified name lookup	119
16.4.2 Basic unqualified non-function lookup	119
16.4.3 Basic unqualified function lookup	120
16.4.4 Lookup with signatures	120
16.4.5 Unqualified function lookup with signatures	121
16.4.6 Qualified non-function name lookup	122
16.4.7 Qualified function name lookup	123
16.4.8 Qualified lookup for function with signatures	123

17 Libraries	124
18 Abstract Types	126
18.0.1 Syntax	126
19 Classes	129
19.1 Class Statement	129
19.1.1 Qualified Names	129
19.1.2 Nested Classes	129
19.1.3 Private definitions	130
19.1.4 Single File Rule	130
19.1.5 Completeness Rule	130
19.1.6 Setwise Lookup	130
19.1.7 Top level module alias	130
19.2 Polymorphic Classes	131
19.2.1 Parametric Polymorphism	131
19.2.2 No Variables	131
19.2.3 Indexed Names	131
19.2.4 Deduced indices	131
19.2.5 Elision of indices	131
19.2.6 Sloppy indexing	132
19.2.7 Polymorphic Members	132
19.3 Virtuals and instances	133
19.3.1 Default Definition	134
19.3.2 Matching polymorphic functions	134
19.4 Classes as Categories	134
19.5 Axioms, Lemmas and Theorems	136
19.5.1 Lemmas and Theorems	137
19.5.2 Reductions	137
19.6 Lookup control directives	137
19.6.1 Open directive	137
19.6.2 Inherit directive	138
19.6.3 Rename directive	138
19.6.4 Use directive	139
 IV Type System	 140
20 typedef statement	141
20.1 Type aliases	141
20.1.1 Basic typedef	141
20.1.2 Polymorphic typedef	141
20.2 Type Sets	141
20.2.1 basic type functions (functors)	142
20.2.2 type functions with typematch	142
20.2.3 Higher order functors	143

20.2.4	Functors in classes	143
21	Products	144
21.1	Tuples	144
21.1.1	Unit Tuple	144
21.1.2	No 1-ary Tuple	144
21.1.3	Projections	144
21.1.4	Standalone Tuple projections	145
21.1.5	Alternate Tuple Representation	145
21.1.6	Pattern matching tuples	146
21.2	Records	147
21.2.1	Plain Record	147
21.2.2	Record projections	147
21.2.3	General record	148
21.2.4	Adding fields	148
21.2.5	Removing Fields	148
21.2.6	Adding records	148
21.2.7	Row Polymorphism	149
21.2.8	Interfaces	149
21.3	Structs	150
21.3.1	Alternate Definition Syntax	150
21.3.2	Struct constructor	150
21.3.3	Constructor as function	150
21.3.4	Projections	151
21.3.5	Nested methods	151
21.3.6	Method closures	152
22	Sums	153
22.1	Anonymous Sum Types	153
22.1.1	Sum type	153
22.1.2	Sum values	153
22.1.3	<code>caseno</code> operator	153
22.1.4	Switching on cases	154
22.1.5	Unit arguments	154
22.1.6	Unit sum	155
22.2	Unions	155
22.2.1	<code>enum</code>	156
22.2.2	<code>caseno</code> operator	157
22.3	Variants	157
22.3.1	Variant coercion	157
23	Exponentials	158
23.1	Functions	158
23.1.1	Function	158
23.2	Generators	159
23.3	Yielding Generators	159

23.3.1	Procedure	160
23.3.2	C function	160
23.3.3	C procedure	161
23.3.4	Type indexed functions	161
23.4	Array	162
23.4.1	Multi-arrays	163
24	Type Recursion	165
24.1	The <code>as</code> combinator	165
24.2	Pointers	165
24.3	Maybe NULL pointers	167
24.4	Incrementable Pointers	168
25	Meta-typing	169
25.0.1	<code>typedef fun</code>	169
25.1	<code>typematch</code> expression	169
25.2	Type Sets	170
25.2.1	Union and Intersection	170
25.2.2	Membership	171
25.2.3	Typeset membership constraints	171
26	Polymorphism	173
26.1	Type Constraints	173
26.1.1	Type class constraints	173
26.1.2	Typeset membership constraints	173
26.1.3	Equational Constraints	174
27	Generics	175
27.1	Generic Functions	175
27.1.1	Generic Syntax	175
27.1.2	Using a generic	175
27.1.3	Operational Semantics	176
27.2	The <code>typecase</code> expression	176
27.3	Builtin Generics	177
27.3.1	The <code>_strr</code> function	177
27.3.2	The <code>_eq</code> function	178
27.3.3	The <code>_map</code> function	179
V	Interfacing	182
28	C bindings	183
28.1	Type bindings	183
28.2	Basic <code>type</code> statement	184
28.2.1	Semantics	184
28.3	Adjectives	184

28.3.1	pod types	184
28.3.2	_gc_pointer types	185
28.3.3	incomplete types	185
28.3.4	uncopyable types	185
28.3.5	_gc_type T types	185
28.3.6	finaliser clause	186
28.3.7	scanner clause	186
28.3.8	encoder clause	187
28.3.9	decoder clause	187
28.4	Expression bindings	187
28.5	Function bindings	188
28.6	Conversion bindings	188
28.7	Procedure bindings	189
28.8	cstruct bindings	189
28.9	cenum bindings	190
28.10	cflags bindings	190
28.11	Floating insertions	190
28.12	Property requirements	191
28.13	Package requirements	191
29	Embedding	193
29.1	C Callable libraries	193
29.2	Export directives	193
VI	Library	195
30	Core Primitive Types	196
30.1	Boolean type	196
30.2	Integer types	196
30.2.1	Classification of integers	198
30.3	Floating point types	200
30.4	Complex types	200
30.5	Quaternion type	200
30.6	Char Type	203
30.6.1	ASCII classification functions	203
31	Algebraic Structure of numeric types	205
31.1	Equality	205
31.2	Total Ordering	206
31.3	Addition	206
32	More core types	209
32.1	Slices	209
32.1.1	Slice Definition	209
32.1.2	Syntax	209

32.1.3	Iterators	210
32.1.4	String slices	210
32.1.5	Generalised Slices	211
32.1.6	Slice addition	211
32.1.7	Generalised String slicing	211
32.2	String Type	212
32.2.1	Equality and total ordering	212
32.2.2	Equality of <code>string</code> and <code>char</code>	212
32.2.3	Append to <code>string</code> object	212
32.2.4	Length of <code>string</code>	212
32.2.5	String concatenation	212
32.2.6	Repetition of <code>string</code> or <code>char</code>	213
32.2.7	Application of <code>string</code> or <code>int</code> is concatenation	213
32.2.8	Construct a <code>char</code> from first byte of a <code>string</code>	213
32.2.9	Constructors for <code>string</code>	213
32.2.10	Substrings	213
32.2.11	Map a <code>string char</code> by <code>char</code>	214
32.2.12	STL string functions	214
32.2.13	Construe <code>string</code> as set of <code>char</code>	216
32.2.14	Construe <code>string</code> as stream of <code>char</code>	216
32.2.15	Test if a string has given prefix or suffix	216
32.2.16	Trim off specified prefix or suffix or both	216
32.2.17	Strip characters from left, right, or both end of a string.	217
32.2.18	Justify string contents	217
32.3	Regexps	217
33	Introspection	218
34	Serialisation	219
34.1	Operation	219
34.1.1	<code>encode_varray</code>	219
34.1.2	<code>deccode_varray</code>	219
34.1.3	Usage	219
34.1.4	Process Restrictions	220
34.1.5	Data Restrictions	220
34.1.6	How it works	220
34.1.7	Future Directions	221
35	Fibres and Schannels	222
35.1	<code>spawn_fthread</code> procedure	222
35.2	Schannel types	222
35.3	<code>mk_ioschannel_pair[T]</code> function	223
35.4	<code>ioschannel[T]</code> constructor	223
35.5	<code>read</code> generator and <code>write</code> procedure	223
35.6	Cross thread I/O	223

35.7 broadcast procedure	224
35.7.1 Fibre States	224
35.7.2 Schannel States	224
35.7.3 Abbreviated Type names	225
35.7.4 Deadlock, Livelock, and Suicide	225
35.7.5 Restrictions on use	226
35.8 Nested Scheduling	226
35.9 Synchronous pipelines	227
35.10 Buffers	227
36 Circuits	229
36.1 Chip definition	229
37 Asynchronous Events	232
37.1 Event Service demux	232
37.2 Alarm Clock	232
37.3 Sockets	233
37.4 How Async I/O works	233
38 Preemptive Threading	235
38.1 spawn_pthread procedure	235
38.2 Pchannel types	235
38.3 mk_iopchannel_pair[T] function	235
38.4 iopchannel[T] constructor	236
38.5 read generator and write procedure	236
38.6 concurrently procedure	236
38.7 Atomics	237
38.8 Mutual Exclusion Lock	237
38.9 Condition Variables	237
38.10 Bound Queue	238
38.11 Thread Pool	239
38.11.1 Restrictions	240
38.12 Parallel For Loop	240
38.12.1 Syntax	240
38.12.2 Operation	240
38.12.3 Example	240
38.12.4 Restrictions	241
VII Platform Support	242
39 Platform Modelling	243
40 Processes	245
41 Programs	246
41.1 Basic Program interface	246

<i>CONTENTS</i>	13
41.2 Environment Variables	247
41.3 Shell callouts	247
41.4 Signals	249
42 File System	250
43 Dynamic Linkage	251
43.1 Basic Library	251
43.2 Plugins	251
VIII Data Structures	252
44 Iterators	253
45 Lists	254
45.1 Inductive lists	254
45.1.1 Description	254
45.1.2 Constructors	255
45.1.3 Destructors	256
45.1.4 Maps	257
45.1.5 Useful lists	258
45.1.6 Operators	258
45.1.7 Lists and Strings	259
45.1.8 Searching	259
45.1.9 Sort	261
45.1.10 Streaming list	261
45.1.11 List syntax	261
45.2 Association List	261
45.3 Purely Functional Random Access List.	262
45.3.1 Definition	262
45.3.2 Length	262
45.3.3 Cons	263
45.3.4 Check if empty	263
45.3.5 Split into head and tail	263
45.3.6 Pattern Matching Support	263
45.3.7 Head	263
45.3.8 Tail	263
45.3.9 Random Access	264
45.3.10 Update	264
45.3.11 Iteration	264
45.3.12 Show	264
45.4 Doubly Linked Mutable List	264
45.4.1 Length <code>len</code>	265
45.4.2 Inspection	265
45.4.3 Insertion	265

<i>CONTENTS</i>	14
45.4.4 Deletion	265
45.4.5 Use as a queue	265
45.4.6 Queue iterator	266
45.5 S-expressions	266
45.6 LS-expressions	266
46 Arrays	268
46.1 Native Arrays	268
46.2 Varrays	268
46.3 Darrays	268
46.4 Sarrays	268
46.5 C arrays	268
IX Appendices	269
A Special Names	270
Glossary	272
Index	273

Chapter 1

Introduction

This reference is a guide to the Felix programming language. It is not the usual reference because Felix differs from other systems in that most of the grammar is part of the library, in user space. In principle then, separating the library from the core language is impossible: if anything the core language is defined by the compiler intermediate abstract machine, details of little interest to most programmers.

Furthermore even that characterisation is weak, because considerable functionality is actually embodied in the run time library. For example the compiler knows what a service request is, but it has no idea what an fthread is. It knows what a generator is, and it knows which generators perform yields, but it has no idea what an iterator is, despite the fact these are effectively a core language feature.

Therefore, our presentation cannot be complete, it cannot be precise, and it cannot replace actually reading the library code. Felix is a highly mutable language, major new features can often be introduced without touching the compiler.

For example a complete set of primitive types with their base operations cannot be presented, because, with a couple of exceptions there aren't any such types. Instead, most primitive types are introduced without knowledge of the compiler, by creating bindings to C++ in library code; these bindings defined the type name and some operations on the types in terms of C++.

Part I

Scripting

Felix is primarily a scripting language, meaning, it is intended for writing a program in one file and immediately running it. It's hard to define what this means but the lack of separate interfaces is one of the things that makes writing script simpler.

Although out of logical order, we present the basic statements, expressions, and definitions first, because together with examples that is enough to get most programmers writing script.

Felix code, like most other languages, is compiled before execution. However it differs from many other systems which compile to byte code and interpret it: Felix uses C++ as its bytecode, and compiles it to machine binary and executes that. Compilation times are slower, but the run time performance is better.

Chapter 2

Language Features

- higher order functions, procedures, and coroutines
- parametric polymorphism
- type constraints
- typeclass based polymorphism
- overloading
- generics
- row polymorphism
- tuple, record and struct product types
- sum, variant and union sum types
- rich pattern matching
- regular expressions
- java like object sublanguage
- circuit based coroutines
- platform independent asynchronous socket I/O
- platform independent pre-emptive threading
- direct embedding of C and C++
- can be embedded in framework idle loops
- generates fast binary programs and libraries
- run scripts directly without makefiles

Chapter 3

The flavour of Felix

Although this is a reference manual, it seems wise to show the flavour of the scripting subsystem with examples, and what could be a better starting point than:

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

3.1 Simple calculations

To perform a simple numerical calculation using a function:

```
fun hypot (x:double, y:double) => sqrt (x^2 + y^2);
var x = 4.0;
var y = 5.0;
var d = hypot (x,y);

println$
  "Hypotenuse of triangle base " + x.str +
  ", side " + y.str " is " +
  d.str
;
```

A better looking way:

```
fun hypot (x:double, y:double) =>  $\sqrt{x^2 + y^2}$ ;
```

encoded like this:

```
fun hypot (x:double, y:double) => \(\sqrt \((x^2 + y^2)\));
```

3.2 Yielding Generators

To calculate the number of rabbits using a yielding generator:

```
gen rabbits () = {
  var generation = 1;
  var last_gen = 1;
  var this_gen = 2;
  while true do
    yield (generation, this_gen);
    last_gen <-> this_gen;
    this_gen += last_gen;
  done
}

var fibonacci = rabbits;

for i in 1..10 do
  var bunnies = fibonacci();
  println$ bunnies;
done
```

3.3 Fibres

To parse text into words, using a coroutine pipeline. First a producer of characters:

```
proc char_stream (s:string) (out:%>char) {
  for c in s perform write (out,c);
  write (out,char " ");
}
```

and now a transducer that converts a character stream into a word stream:

```
proc words (inp:%<char, out:%>string) {
  while true do
    var s = "";
    nextch: while true do
      var c = read inp;
      if isalnum c do
```

```

        s+=c;
    else
        if s != "" call write (out,s);
        break nextch;
    done
done // nextch
done
}

```

Finally we need a sink to emit the result:

```

proc emit (inp:%<string) {
    while true do
        var word = read inp;
        println$ "Word = " + word;
    done
}

```

which you can use like this:

```

var text =
    "The quick, brown fox jumped "
    "over the lazy dog"
;

#(char_stream (text) |-> words |-> emit);

```

3.4 Circuits

Circuits generalise the capabilities of pipelines. Two data sources:

```

open syntax chips;

chip w10
    connector pins
    pin outpin: %>int
{
    for i in 0..10 do
        write (pins.outpin,i);
    done
}

chip w100

```

```

connector pins pin outpin: %>int
{
  for i in 100..110 do
    write (pins.outpin,i);
  done
}

```

and a data sink:

```

chip conout
connector pins pin inpin: %<int
{
  while true do
    var x = read pins.inpin;
    println$ "Read " + x.str;
  done
}

```

with a transducer:

```

chip doubler
connector pins
  pin inpin: %<int
  pin outpin: %>int
{
  while true do
    var x = read pins.inpin;
    write (pins.outpin, 2 * x);
  done
}

```

can be combined like this:

```

/*
  w10 ---\
           \---> doubler --> conout
  w100---/
*/

chip fred
connector pins
  pin inch: %<int
{

```

```
circuit
  connect w10.outpin to doubler.inpin
  connect w100.outpin to doubler.inpin
  wire pins.inch to doubler.outpin
endcircuit
}
```

to form a circuit we can use like this:

```
var dummy_channel = mk_schannel[int]();

circuit
  wire dummy_channel to fred.inch
endcircuit

while true do
  var x = read dummy_channel;
  println$ "Read " + x.str;
done
```


Chapter 4

Variable Definitions

Syntax

A *definition* is a statement, clause, expression, phrase or other term which defines a name, but does not cause any observable behavior, or, a class statement, or, a `var` or `val` statement. The latter two exceptions define a name but may also have associated behaviour.

Felix provides two primary kinds of variables:

1. a `var` variable names mutable store, and.
2. a `val` variable names a functional expression

4.1 The `var` binder

4.1.1 The `var` statement

The `var` statement is used to introduce a variable name and potential executable behaviour. It has one of three basic forms:

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

1. The first form specifies the type and an initialising expression which must be of the specified type or a subtype thereof.
2. The second form specifies a variable of the given type without an explicit initialiser, however the variable will be initialised anyhow with the default constructor for the underlying C++ type, although that constructor may be trivial.

3. The third form does not specify the type, it will be deduced from the initialiser.

4.1.2 Semantic rules

1. If the initialiser has observable behaviour it will be observed if at all, when control passes through the variable statement.
2. If the variable introduced by the `var` statement is not used, the variable and its initialiser will be elided and any observable behaviour will be lost.

To be used means to have its address taken in a used expression, to occur in a used expression. A used expression is one which initialises a used variable, or, is an argument to function or generator in a used expression, or an argument to a procedure through which control passes.

In other words, the variable is used if the behaviour of the program appears to depend on its value or its address.

3. The library procedure `C_hack::ignore` ensures the compiler believes a variable is used:

```
var x = expr;
C_hack::ignore x;
```

so that any side effects of `expr` will be seen.

4. In general the argument to any primitive function, generator or procedure will be considered used if its containing entity is also considered used. In general this means there is a possible execution path from a root procedure of the program.

4.1.3 Multiple variables

Multiple variables can be defined at once:

```
var m = 1,2;
var a,b = 1,2;
var c,d = m;
```

With this syntax, no type annotation may be given. The RHS of the statement must be an expression of tuple type, with the number of components of the tuple equal to the number of variables being defined.

4.1.4 Var expressions

Variables can also be defined inside expressions:

```
var x = 1 + (2 as var y) + (var z = y+1);
println$ x,y,z;
```

The `as var` form has low precedence and usually requires parenthesis, in the assignment form parenthesis are mandatory.

Nested `var` definitions are evaluated before their containing expression, and for several expressions directly contained within the same expression, in left to right order. The order is therefore deterministic.

Unnamed one use temporary variables can also be specified in a `var` subexpression:

```
var x = 1 + (var rand());
println$ x;
```

The effect is to enforce eager, single evaluation of the subexpression prior to its containing expression.

4.1.5 Addressing

A variable may have its address taken:

```
var x = 1;
var rwp = &x; // read-write pointer
var rpx = &<x; // read-only pointer
var wpx = &>x; // write-only pointer
```

A variable may be assigned a value directly by an assignment statement:

```
x = 2;
```

or indirectly via a call to the system intrinsic `_storeat` procedure:

```
_storeat(rwp,3);
_storeat(wpx,4);
```

If a variable stored no previous value and assignment sets the first value. Otherwise, if a variable previously stored a value, an assignment replaces it.

The standard grammar also provides the infix operator `<-` which invokes the standard library procedure `storeat` which by default invokes the procedure `_storeat`:

```
storeat(rwpx,3);  
wpx <- 4;
```

The standard library procedure `storeat` can be overloaded, which also changes the behaviour of the infix operator `<-`.

4.1.6 Referencing

If a value is stored in a variable it can be referred to in an expression by using the name of the variable.

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

If a read-write or read only pointer to a variable is obtained, the stored value, if it exists, can be referred to by using the system intrinsic `_deref` function:

```
println$ _deref &x; // prints 1  
println$ _deref &<x; // prints 1
```

The standard grammar also provides the prefix operator `*` which invokes the standard library function `deref` which by default invokes the system intrinsic `_deref`:

```
println$ deref &x; // prints 1  
println$ *(&x); // prints 1
```

The standard library function `deref` can be overloaded, which also changes the behaviour of the prefix operator `*`.

4.1.7 Object Concept

A variable is said to name an object, not a value. This basically means it is associated with the address of a typed storage location.

4.1.8 Hoisting and lookup

A variable can be assigned to, addressed, or otherwise used before the `var` statement or expression which introduces it. This is because Felix does set-wise, not linear, lookup.

An initialising definition such as

```
var x = 1;
```

is split into an uninitialised variable definition and an assignment, and the definition is then hoisted to the top of the scope. For example:

```
x = 1;
var x = 2;
```

is converted to:

```
var x: int;
x = 1;
x = 2;
```

JavaScript behaves the same way, and the phenomena is called *hoisting*.

Care should be taken because it is possible to refer to an uninitialised variable, a variable unexpectedly initialised by the underlying C++ default constructor, or the wrong variable entirely.

For example here:

```
var x = 1;    //1
begin        //2
  var y = x;  //3
  var x = 2;  //4
end          //5
```

the variable `y` is set to an indeterminate value because its initialiser `x` is the one which is subsequently set to 2 on line 4, not the one set to 1 on line 1. This also means this code is legitimate:

```
x = 1;
x = 2;
var x = 3;
```

because `x` is in fact defined as a variable on the third line.

4.1.9 var parameters

A function or procedure may have a parameter marked `var`:

```

fun f(var x: int) => 1; //1
var g = f;           //2
println$ f (rand()); //3
println$ g (rand()); //4

```

An argument bound to a **var** parameter will always be evaluated, if at all, before the main body of the function is entered; that is, it ensure eager evaluation.

A **var** parameter in a direct call or application may or may not be eliminated, unlike a local variable which will always be eliminated if not used.

If the function or procedure is inlined, it is likely but not assured that the parameter and its initialising argument will be eliminated, hence in the example the evaluation of **rand()** on line 3 could be eliminated because the parameter *x* is not used in *f*.

On the other hand it is very unlikely that the argument to *g* will be eliminated, because parameters of a closure cannot be, without changing the closure's type. Therefore the **rand()** on line 4 will almost certainly be evaluated.

4.2 The val binder

4.2.1 The val statement

A **val** statement defines a name for an expression.

```

val x : int = 1;
val z = 1;

```

The value associated with a **val** symbol may be computed at any time between its definition and its use, and may differ between uses, if the initialising expression depends on variable state, such as a variable or call to a generator.

It is not an error to create such a dependence since either the value may, in fact, not change, or the change may not be significant.

Nevertheless the user must be warned to take care with the indeterminate evaluation time and use a **var** when there is any doubt.

Since a **val** simply names an expression, it is associated with a value not an object and cannot be addressed or assigned to.

However this does NOT mean its value cannot change:

```
for var i in 0 upto 9 do
  val x = i;
  println$ x;
done
```

In this example, `x` isn't mutable but it does take on all the values 0 to 9 in succession. This is just a most obvious case: a less obvious one:

```
var i = 0;
val x = i;
println$ x;
++i;
println$ x;
```

which is clearly just an expansion of the the first two iterations of the previously given for loop. However in this case there is no assurance `x` will change after `i` is incremented because the compiler is free to replace any `val` definition with a `var` definition.

4.2.2 Multiple values

Multiple values can be defined at once:

```
val m = 1,2;
val a,b = 1,2;
val c,d = m;
```

With this syntax, no type annotation may be given.

4.2.3 Val expressions

Values can also be defined inside expressions:

```
val x = 1 + (2 as val y) + (val z = y+1);
println$ x,y,z;
```

The `as val` form has low precedence and usually requires parenthesis, in the assignment form parenthesis are mandatory.

4.2.4 Pattern variables

In a pattern match, pattern variables are implicitly `val` variables with their scope limited to the branch for which they're specified: For example the statement:

```
match 1,2 with
| (x,y) => println$ x +y;
endmatch;
```

the variables x and y act as if:

```
begin
  val x = 1;
  val y = 2;
  println$ x + y;
end
```

had been written.

4.2.5 let expressions

A `let` expression can be used to introduce a named value within an expression which has scope limited to the `let` expression:

```
var y = 42;
var x = let y = 1 in y + y;
println$ x,y; // 2,42
```

In fact, in a `let` expression, the phrase after `let` and before

```
let pattern = expr1 in expr2
```

is syntactic sugar for

```
match expr1 with
| pattern => expr2
endmatch
```

4.2.6 val parameters

A function or procedure may have a parameter marked `var`:

```
fun f(val x: int, y: int) => x + y;
```

If a parameter is not explicitly marked `var`, then it is a `val` parameter. The explicit spelling of `val` is allowed but not required.

4.2.7 val semantics

A name introduced by a **val** or **let** binder is assumed to bind to a purely functional expression. This means the expression depends only on explicitly visible terms of the expression, has no side-effects, and, has a value which is temporally invariant or *referentially transparent*.

The compiler may treat the **val** precisely as a **var** and allocate store to hold the value its initialiser designates. Alternatively, the compiler may replace occurrences of the use of the defined name in an expression with the initialising value, or, a combination of both. The compiler may also shift the position of the initialising assignment, in the case it chooses to allocate store, to any point after the definition and before the first use. It may also evaluate the expression earlier, provided it does not impact the evaluation.

The kind of evaluation strategy described is a key feature of Felix and is called *indeterminate evaluation* strategy. It is one of the primary semantic design principles which provide high performance by allowing the compiler to choose whether to evaluate the **val** eagerly or lazily. In particular, **val** parameters of a closure are almost universally eagerly evaluated, which avoids the need to pass the closure a closure over the expression. On the other hand if a function is called directly and is inlined, the compiler is free to elide the parameter and substitute the expression for its occurrences, thus evaluating it lazily and saving a storage location as well.

4.3 def statement

A special statement called a **def** statement can be used to introduce zero or more **var** variables, zero or more **val** variables, and perform simple assignments, simultaneously:

```
var x = 1;
def var a, (val b, x, _) = 2, (3,4,42);
```

The LHS pattern is called an *l-pattern* which consists of a list of *l-terms* each of which may be either a **var** binder introducing a new variable, a **val** binder introducing a new variable, a simple name referring to an existing **var** variable, an underscore, or a nested l-pattern enclosed in parens.

The components other than underscore, are assigned the corresponding values from the term on the RHS which must be of tuple type with the same number of components, recursively for nested l-patterns; underscore is a place holder for a component which is to be ignored.

A component matching an underscore may or may not be evaluated.

4.3.1 Uniquely typed components

Uniqueness typing is provided to provide a way to implement move semantics instead of the usual copy and overwrite semantics of assignment; it allows the programmer to assert acquisition of exclusive ownership of an object, and transfer that ownership along with the objects value.

Felix provides an intrinsic type constructor `_uniq` which maps a type `T` to its unique equivalent.

The programmer announces a value is acquired using the `box` intrinsic, which when applied to an object of type `T` returns an object of type `_uniq T`. The programmer can then pass the object to a function which requires a uniquely typed parameter. The function can then pass the object on, or use the `kill` intrinsic to forget it, otherwise, to use the boxed object the `unbox` intrinsic is applied to return the boxed type `T`.

```
begin
  fun toupper (var s: uniq varray[char]) {
    var x = unbox s;
    for i in 0 ..< x.len.int perform
      set(x,i,toupper (get(x,i)));
    return box x;
  }

  var u = box (varray "hello");
  u = toupper u;
  println$ (unbox u);
end
```

Because a function acquiring a uniquely typed object is the exclusive owner, it is free to modify the object without compromising referential transparency, because by specification there can be no other observer.

In other words box/unbox together perform a move from the caller of a function to the function's implementation, using the uniqueness typing of the interface to facilitate enforcement of the contract.

Variables which have a type which is `_uniq`, or a product type with one or more component `_uniq` are treated specially. Control flow analysis attempts to assign a consistent state to such variables or components, either *dead* or *live*.

The library provides a functor `uniq: TYPE -> TYPE` which maps a type `T` to `_uniq T`.

1. Assignment to a live variable or component is not permitted and an error reported if detected.

2. Use of a dead variable or component in an expression is not permitted and an error reported if detected.
3. Using a live variable or component changes the state of the component from live to dead.
4. Assigning to a dead component changes the state from dead to live.
5. If the variable or component is a parameter or part of a parameter it is initially considered live, otherwise if it is a local variable it is considered dead.
6. A variable must be dead when control exits.
7. If a variable is used in one branch of a pattern match or conditional it must be used in all of them.
8. A uniquely typed component of an variable of a product type is treated as if it were a whole variable.

This rule is subtle! For example:

```
begin
  var x = 1, uniq 2;
  var y = x; // kills x.1
  println$ y.0;
end // ERROR: y.1 unused!
```

9. `_uniq T` is a subtype of `T`. Therefore, a variable of unique type can be passed as an argument to a function with the corresponding non-unique type, the argument will be implicitly unboxed.

Top level global variables are not analysed and should not be uniquely typed because they're not considered by the flow analyser.

Taking the address of a uniquely type variable is ignored by the flow analyser. For therefore the correctness of the analysis depends on the programmer not abusing pointers to unique variables or components.

The flow analyser will report an error if a consistent state cannot be assigned to a variable or component.

The flow analyser does first order static analysis only. It cannot cope with either indirect or dynamic access.

Chapter 5

Assignment Statements

Syntax

5.1 Basic Assignment

In Felix, the core assignment operation is an intrinsic procedure

```
_storeat: &>T * T -> 0
```

which is wrapped by a library procedure:

```
proc storeat[T] (p:&>T,v:T) => _storeat(p,v);
```

so it can be overloaded. The parser, in turn, maps the statement

```
p <- v;
```

to call the library procedure. The type `&>T` denotes a write-only pointer to `T`, however the type `&T` of a read-write pointer to `T` is a subtype, so a pointer of that type can also be used for assignment:

```
var x = 1;  
&x <- 2;
```

Felix provides a conventional assignment operator:

```
var x = 1;
x = 2;           // means &x <- 2;
```

The left hand side of such an assignment must be a variable, or a dereference of a pointer expression:

```
var px = new 1; // on the heap
*px = 2;        // means px <- 2;
```

Initialisation of a variable in a `var` statement, or `var` l-term of a `def` statement is also an assignment.

5.2 Read-modify-write operations

The following read/modify/write operations are similar to those in C. Each of these dispatches to a *procedure* of the same name, taking two arguments, the first being a read-write pointer and the second a value. The parser automatically takes the address of the LHS of the assignment when calling the procedure. Therefore, the LHS must be a simple variable.

```
var x = 1u; var y=1u;

// from class Integer
// all integers
x += y; // means +=(&x,y)
x -= y; // means -=(&x,y)
x *= y; // means *=(&x,y)
x /= y; // means /=(&x,y)
x %= y; // means %=(&x,y)
x <<= y; // means <<=(&x,y)
x >>= y; // means >>=(&x,y)

// from class Bits
// unsigned integers only
x ^= y; // means ^=(&b,y)
x |= y; // means |=(&b,y)
x &= y; // means &=(&b,y)
```

Felix does not guarantee these operations are atomic.

5.3 Swap operator

Felix provides the `_swap` intrinsic procedure to swap the contents of two variables.

```
var x,y = 1,2;  
_swap (&x,&y); // now x=2, y=1
```

The variables must be of the same type. The parser maps the infix operator `<->` to the procedure `swap` which is defined in the library to call `_swap` by default.

```
var x = 1;  
var y = 2;  
x <-> y; // swap so x = 2, y = 1
```

5.4 Pre and post increment and decrement

Felix provides pre and post increment and decrement. These are procedures taking a pointer argument, not expressions. The procedures have special names as indicated:

```
var x = 1;  
++x; // means pre_incr (&x);  
--x; // means pre_decr (&x);  
x++; // means post_incr (&x);  
x--; // means post_decr (&x);
```

Chapter 6

Control Flow Statements

6.1 The goto statement and label prefix

Felix statements may be prefixed by a label to which control may be transferred by a `goto` statement:

```
alabel:>  
  dosomething;  
  goto alabel;
```

The label must be visible from the `goto` statement.

There are three kinds of `gotos`.

1. A *local goto* is a direct jump to a label in the same scope as the `goto` statement.
2. A *non-local goto* is a direct jump to any other label, visible in an ancestor scope.
3. An *indirect goto* is a `goto` with a non-constant expression of `LABEL` type as its operand.

6.1.1 Non-local gotos

Non-local transfers of control may cross procedure boundaries. They may not cross function or generator boundaries.

The procedure or function containing the label must be active at the time of the control transfer.

A non-local goto may be wrapped in a procedure closure and passed to a procedure from which the goto target is not visible.

```
proc doit (err: 1 -> 0) { err; }

proc outer () {
  proc handler () { goto error; }
  doit (handler);
  return;

  error:> println$ error;
}
```

This is a valid way to handle errors. the code is correct because `outer` is active at the time that `handler` performs the control transfer.

6.1.2 Label values

Labels are first class values of the kind `LABEL`. They can be stored in variables, and targetted by a goto statement.

```
proc demo (selector:int) {
  var pos : LABEL =
    if selector == 1
    then lab1
    else lab2
    endif
  ;
  goto pos;
lab1:>
  println$ "Lab1"; return;
lab2:>
  println$ "Lab2"; return;
}
```

Used of saved label values may not be sound unless the procedure frame of the procedure containing the label at the time it is saved remains live at the time it is jumped to.

Label values are not merely code addresses, they are closures which capture the state of the containing procedure frame. This prevents the frame from being garbage collected, but it does not prevent control being returned from the frame. The effect of executing an already returned frame will cause the program to refer to local variables which have the values

they had when last modified. This is well defined!

In the current implementation, returning from a procedure resets the return address to NULL which means a return statement will actually return to the invoking scheduler terminating the fibre. Decoupling an exhausted frame from its caller is not done to make returning to a dead frame safe. This is a side effect of an optimisation which is intended to prevent a data pointer into the frame making the call chain reachable.

6.1.3 Exchange of control

Built on top of label addressing and indirect gotos, the **branch-and-link** instruction is conceptually the most fundamental control instruction. The library implementation is

```
inline proc branch-and-link (target:LABEL, save:LABEL)
{
    save <- next;
    goto target;
    next:>
}
```

A good example is [here](#), which shows an example of coroutines.

6.2 Conditional Statements

6.2.1 if/goto

If the condition is true, jump to the goto target, otherwise continue without jumping.

```
if c goto lab;
```

6.2.2 if/return

If the condition is true, return from the current procedure, otherwise continue without returning.

```
if c return;
```

6.2.3 if/call

If the condition is true, call the specified procedure, otherwise continue without calling it.

```
if c call f x;
```

6.2.4 if/perform

If the condition is true, execute the specified statement, otherwise continue without executing it.

```
if c perform x = 1; // equivalent to
```

6.2.5 if/do/elif/else/done

The procedural conditional branch is used to select a control path based on a boolean expression.

The `else` and `elif` clauses are optional.

```
if c1 do
  stmt1;
  stmt2;
elif c2 do
  stmt3;
  stmt4;
else
  stmt5;
  stmt6;
done
```

The `elif` clause saves writing a nested conditional. The above is equivalent to:

```
if c1 do
  stmt1;
  stmt2;
else
  if c2 do
    stmt3;
    stmt4;
  else
    stmt5;
    stmt6;
  done
done
```

One or more statements may be given in the selected control path. The bodies of the subclasses do not form a scope. Variables declared therein are visible in the containing context. Labels therein can be targetted by `gotos`.

6.3 halt

Stops the program with a diagnostic.

```
halt "Program complete";
```

6.4 try/catch/endtry

The try/catch construction may only be used to wrap calls to C++ primitives, so as to catch exceptions.

```
proc mythrow 1 = "throw 0;";
try
  mythrow;
catch (x:int) =>
  println$ "Caught integer " + x.str;
endtry
```

6.5 match/endmatch

The form:

```
match expr with
| pattern1 => stmts1
| pattern2 => stmts2
...
endmatch;
```

is an extension of the C switch statement. See [chapter 9 Pattern Matching](#).

The match argument expression is compared in turn with each pattern. If a match is found, the pattern variables in the pattern are bound to the corresponding sub-expressions of the argument expression. Then the statements of that branch are executed with those variables in scope. The variables are `val` bound and so immutable. The branch statements form a scope.

6.6 chainmatch/ormatch

The form:

```
chainmatch expr1 with
| pattern1 => stmts1
| pattern2 => stmts2
```

```

ormatch expr2 with
| pattern3 => stmts3
| pattern4 => stmts4
ormatch expr3 with
| pattern5 => stmts5
| pattern6 => stmts6
| _ => stmts7
endmatch;

```

is sugar for:

```

match expr1 with
| pattern1 => stmts1
| pattern2 => stmts2
| _ =>
  match expr2 with
  | pattern3 => stmts3
  | pattern4 => stmts4
  | _ =>
    match expr3 with
    | pattern5 => stmts5
    | pattern6 => stmts6
    | _ => stmts7
    endmatch;
  endmatch;
endmatch;

```

6.7 call

The `call` statement is used to invoke a procedure.

```

proc p(x:int) { println$ x; }
call p 1;

```

The word `call` may be elided in a simple call:

```

p 1;

```

If the argument is of unit type; that is, it is the empty tuple, then the tuple may also be elided in a simple call:

```
proc f() { println$ "Hi"; }  
call f (); // is equivalent to  
f(); // is equivalent to  
f;
```

6.8 procedure return

The procedural return is used to return control from a procedure to its caller.

A return is not required at the end of a procedure where control would otherwise appear to drop through, a return is assumed:

```
proc f() { println$ 1; }  
// equivalent to  
proc f() { println$ 1; return; }
```

6.9 return from

The return from statement allows control to be returned from an enclosing procedure, provided that procedure is active.

```
proc outer () {  
  proc inner () {  
    println$ "Inner";  
    return from outer;  
  }  
  inner;  
  println$ "Never executed";  
}
```

6.10 jump

The procedural jump is an abbreviation for the more verbose sequence:

```
jump procedure arg; // is equivalent to  
call procedure arg;  
return;
```

6.11 calcc

The calcc statement

```
callcc f a;
```

calls the function f passing the location of the next statement as its argument. The returned value is then called with a as its argument.

The location is represented as a value of LABEL type. If the procedure wishes to return to this point it should goto the parameter which received the argument.

```
proc f(cc:LABEL) (x:int) {  
  println$ "F got " + x.str;  
  goto cc;  
}  
callcc f 42;  
println$ "Back";
```

The callcc statement above is equivalent to:

```
var tmo = f next;  
call tmp 42;  
next:>
```

6.12 function return

The functional return statement returns a value from a function.

```
fun f () : int = {  
  return 1;  
}
```

Control may not fall through the end of a function.

6.13 yield

The yield statement returns a value from a generator whilst retaining the current location so that execution may be resumed at the point after the yield.

For this to work a closure of the generator must be stored in a variable which is subsequently applied.

```
gen counter () = {  
  var x = 0;  
  next_integer:>
```

```

    yield x;
    ++x;
    goto next_integer;
}

var counter1 = counter;
var zero = counter1 ();
var one = counter1 ();
println$ zero, one;

```

6.14 call_with_trap and throw_continuation

The `call_with_trap` statement calls a procedure, catching any uncaught thrown continuation, if there is one.

If control returns normally, it continues normally.

A procedure of type `1->0` may be thrown with a `throw_continuation` statement. It will be caught by the closest dynamically enclosing `call_with_trap` statement or the top level scheduler.

Such a trapped continuation then replaces the currently running procedure and begins execution.

```

fun divide (error:1->0) (a:int, b:int) = {
    if b == 0 perform throw_continuation error;
    return a/b;
}

proc demo() {
    proc divide_by_zero () {
        println$ "Divide by zero";
        goto bad;
    }
    var q = divide divide_by_zero (1,0);
    println$ "Quotient " + q.str;
    return;
bad:>
    println$ "Error Handled";
}
call_with_trap demo;
println$ "Done";

```

6.15 loops

Library Reference

Felix has two kinds of loops, low level loops and high level loops.

Felix loops consist of a control section and a body. The body must be a group statement. There are two kinds of group statements:

1. An *basic group* is a **perform** statement, a **do/done** group, an **if/do/done** statement, or a loop whose body is a basic group.

Basic groups do not constitute a scope.

2. A *block* A **begin/end** statement or a statement **match** statement is a block.

Block define an anonymous scope. Variables defined in a block do not escape, and you cannot jump to a label in a block.

6.15.1 C like loop

This loop is similar to a C for loop except that the increment operation is a statement, not an expression, and so may need a semi-colon as a terminator. It is the fastest and most flexible loop, but it is hardest to optimise.

```
for(var i=0; i<20; ++i;)
    perform println$ i;
println$ "Final i " + i.str;
```

Note the control variable is defined in the containing scope and is not local to the loop.

6.15.2 for/in/upto/downto

A basic loop with an inclusive range.

```
// up
for var ti:int in 0 upto 9 do println$ ti; done
for var i in 0 upto 9 do println$ i; done
for i in 0 upto 9 do println$ i; done

// down
for var tj:int in 9 downto 0 do println$ j; done
for var j in 9 downto 0 do println$ j; done
for j in 0 upto 9 do println$ j; done
```

The start and end expressions must be of the same type.

If the control variable is defined in the loop with a type annotation, that type must agree with the control variable.

The type must support comparison with the equality operator `==` the less than or equals operator `<=` and increment with the pre increment procedure `pre_incr`

The loop logic takes care to ensure the control variable is not incremented (resp. decremented) past the end (resp.start) value.

Loops over unsigned integer types cannot handle the empty case. Loops over signed integer types cannot span the whole range of the type.

6.15.3 while loop

The while loop executes the body repeatedly whilst the control condition is true at the start of the loop body.

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

6.15.4 until loop

The until loop executes the loop body repeatedly until the control condition is false at the start of the loop, it is equivalent o a while loop with a negated condition.

```
var i = 0;
until i == 9 do println$ i; ++i; done
```

6.15.5 Control Flow modifiers

Low level loops may be labelled with a loop label which is used to allow break, continue, and redo statements to modify the usual control flow.

```
outer:for var i in 0 upto 9 do
  inner: for var j in 0 upto 9 do
    println$ i,j;
    if i == j do break inner; done
    if i * j > 60 do break outer; done
  done
done
```

redo

The redo statement causes control to jump to the start of the specified loop without incrementing the control variable.

break

The break statement causes control to jump past the end of the specified loop, terminating iteration.

continue

The continue statement causes the control variable to be incremented and tests and the next iteration commenced or the loop terminated.

6.15.6 Parallel for loop

The pfor loop is a high level loop which uses a thread pool to execute loops over subranges of the specified range in parallel.

```
proc copy(inp: varray[uintptr], out: varray[uintptr]) {  
  pfor i in 0 upto 99 do  
    set(out, i, f (get(inp,j)));  
  done  
}
```

If the system thread pool is not created, it is constructed. The range of the control variable is divided into n subranges, where n is the number of threads. Then n procedures are created which run the loop over each subrange and are submitted to the thread pool. The statement after the pfor statement is executed after all the threads have completed.

Note the thread pool is not destroyed, even if it was created, the programmer must do this or the program will not terminate. [This is a design fault!]

The pfor statement requires that iterations of the loop be either uncoupled, or coupled appropriately, for example, by using atomic variables.

If pfor statements are nested, the result is undefined. It is always bad for performance and is likely to lead to a deadlock. A nested pfor is executed from a job running in the thread pool and will submit further jobs to the thread pool and then wait for them to complete. Unfortunately it is likely the won't start because the submitting job is still active, even though it is waiting.

6.16 Loops over iterators

The iterator based for loop takes sets a variable to each value stored in any data type with an iterator:

```
for i in (1,2,3) perform println$ i;  
for i in 0..<10 perform println$ i;
```

An iterator is a generator taking the data type as an argument and returning an option type. The iterator over the data type constructs an iterator and then repeatedly calls it until it returns `None`, storing the argument v of the `Some v` otherwise returned in i and then executing the loop body.

The iterator is obtained generically, that is, by attempting to apply a function named `iterator` to the container.

The argument can also be any generator returning an option type, including any iterator.

This loop is wrapped in a block so that the control variable is *not* available at the end of the loop, definitions in the body do not leak into the surrounding context, and jumps into the loop body are not possible.

6.16.1 Recursive for loop

The `rfor` loop is similar to the iterator based for loop, however it constructs a new data frame for each iteration. It does this by converting the loop into a procedure which self-tail calls itself to implement the iteration.

The compiler may optimise an `rfor` loop using self-tail-call optimisation, so it uses a single frame, as for a `for` loop. However, the optimisation is not performed unless it is safe, and in particular if a pointer to a local variable is exported from the body, it cannot be performed.

An `rfor` loop should be used when creating a closure that binds to the loop body, if you wish it to bind to the current values of that iteration. If a `for` loop is used instead the closure will bind to the sole data frame and containing the values of the last iteration, not the values current at the time the closure is formed. For example:

```

begin
  println$ "for loop -----";
  var v = varray[1->int] 4uz;
  for i in (0,1,2,3) perform push_back(v,{i});
  for j in 0..3 perform println$ #(v.j);
  // 3,3,3,3
end
begin
  println$ "rfor loop -----";
  var v = varray[1->int] 4uz;
  rfor i in (0,1,2,3) perform push_back (v,{i});
  for j in 0..3 perform println$ #(v.j);
  // 0,1,2,3
end

```

6.16.2 match/in

This is an advanced iterator based loop which allows a single arbitrary pattern match to be applied to the values supplied by the iterator and present only those which match to the loop body.

```

match x,y when x < y in ((1,2),(4,3))
  perform println$ x,y;

```

6.17 Assertions

Library Reference

6.17.1 assert

Ad hoc assertion throws an assertion exception if its argument is false.

```

assert x > 0;

```

6.17.2 axiom

An axiom is a relationship between functions, typically polymorphic, which is required to hold.

```

axiom squares (x:double) => x * x >= 0;
class addition[T]
{
  virtual add : T * T -> T;
}

```

```

virtual == : T * T -> bool;

axiom assoc (x:T, y:T, z:T) :
  add (add (x,y),z) == add (x, add (y,z))
;
}

```

In a class, an axiom is a specification constraining implementations of virtual function in instances.

Axioms are restricted to first order logic, that is, they may be polymorphic, but the universal quantification implied is always at the head.

Existential quantification can be provided in a constructive logic by actually constructing the requisite variable.

Second order logic, with quantifiers internal to the logic term, are not supported.

6.17.3 lemma

A lemma is similar to an axiom, except that it is easily derivable from axioms; in particular, a reasonable automatic theorem prover should be able to derive it.

6.17.4 theorem

A theorem is similar to a lemma, except that it is too hard to expect an automatic theorem prover to be able to derive it without hints or assistance.

There is currently no standard way to prove such hints.

6.17.5 reduce

A reduce statement specifies a term reduction and is logically equivalent to an axiom, lemma, or theorem, however it acts as an instruction to the compiler to attempt to actually apply the axiom.

The compiler may apply the axiom, but it may miss opportunities for application.

The set of reductions must be coherent and terminal, that is, after a finite number of reductions the final term must be unique and irreducible.

Application of reduction is extremely expensive and they should be used lightly.

```

reduce revrev[T] (x: list[T]) : rev (rev x) => x;

```

6.18 invariant

An invariant is an assertion which must hold on the state variables of an object, at the point after construction of the state is completed by the constructor function and just before the record of method closures is returned, and, at the start and end of every method invocation.

The invariant need not hold during execution of a method.

Felix inserts the a check on the invariant into the constructor function and into the post conditions of every procedure or generator method.

```
object f(var x:int, var y:int) =
{
  invariant y >= 0;
  method proc set_y (newy: int) => y = newy;
}
```

6.19 code

The code statement inserts C++ code literally into the current Felix code.

The code must be one or more C++ statements.

```
code 'cout << "hello";';
```

6.20 noreturn code

Similar to code, however noreturn code never returns.

```
noreturn code "throw 1;";
```

6.21 Service call

The service call statement calls the Felix system kernel to perform a specified operation.

It is equivalent to an OS kernel call.

The available operations include:

```
union svc_req_t =
/*0*/ | svc_yield
/*1*/ | svc_get_fthread    of &fthread    // CHANGED LAYOUT
/*2*/ | svc_read          of address
```

```

/*3*/ | svc_general      of &address  // CHANGED LAYOUT
/*4*/ | svc_reserved1
/*5*/ | svc_spawn_pthread of fthread
/*6*/ | svc_spawn_detached of fthread
/*7*/ | svc_sread        of _schannel * &gcaddress
/*8*/ | svc_swrite       of _schannel * &gcaddress
/*9*/ | svc_kill         of fthread
/*10*/ | svc_reserved2
/*11*/ | svc_multi_swrite of _schannel * &gcaddress
/*12*/ | svc_schedule_detached of fthread
;

```

These operations are typically related to coroutine or thread scheduling. However `svc_general` is an unspecified operation, which is typically used to invoke the asynchronous I/O subsystem.

Service calls can only be issued from flat code, that is, from procedures, since they call the system by returning control, the system must reside exactly one return address up the machine stack at the point a service call is executed.

6.22 with/do/done

The `with/do/done` statement is used to define temporary variables which are accessible only in the `do/done` body of the statement.

It is the statement equivalent of the `let` expression.

```

var x = 1;
with var x = 2; do println$ x; done
assert x == 1;

```

6.23 do/done

The `do/done` statement has no semantics and merely acts as a way to make a sequence of statements appear as a single statement to the parser.

Jumps into `do/done` groups are therefore allowed, and any labels defined in a `do/done` group are visible in the enclosing context.

Any variables, functions, or other symbols defined in a `do/done` group are visible in the enclosing context.

```
do something; done
```

6.24 begin/end

The begin/end statement creates an anonymous procedure and then calls it. It therefore appears as a single statement to the parser, but it simulates a block as would be used in C. It is exactly equivalent to a brace enclosed procedure called by a terminating semi-colon.

```
begin
  var x = 1;
end
// equivalent to
{
  var x = 1;
};
```


Chapter 7

Expressions

Syntax

Expressions are listed in approximate order of precedence, starting with the weakest binding.

We will often exhibit expressions in the form

```
var x = expr;
```

so as to present a complete statement. The `x` is of no significance.

7.1 Priority levels

```
priority
  let_pri <
  slambda_pri <
  spipe_apply_pri <
  sdollar_apply_pri <
  stuple_cons_pri <
  stuple_pri <
  simplies_condition_pri <
  sor_condition_pri <
  sand_condition_pri <
  snot_condition_pri <
  stex_implies_condition_pri <
  stex_or_condition_pri <
  stex_and_condition_pri <
  stex_not_condition_pri <
```

```

scomparison_pri <
sas_expr_pri <
ssetunion_pri <
ssetintersection_pri <
sarrow_pri <
scase_literal_pri <
sbor_pri <
sboxor_pri <
sband_pri <
sshift_pri <
ssum_pri <
ssubtraction_pri <
sproduct_pri <
s_term_pri <
sprefixed_pri <
spower_pri <
ssuperscript_pri <
srefr_pri <
sapplication_pri <
scoercion_pri <
sfactor_pri <
srcompose_pri <
sthename_pri <
satomic_pri
;

```

7.2 Chain forms

7.2.1 Pattern let

The traditional let binding of ML. The syntax is

```

var x = let pattern = expr1 in expr2; // equivalent to
var x = match expr1 with pattern => expr2 endmatch;

```

```

var x = let a = 1 in a + 1; // equivalent to
var x = match 1 with a => a + 1 endmatch

```

7.2.2 Function let

A let form which makes a function available in the expression

```
var x =  
  let fun f(y:int)=> y + 1 in  
  f 42  
;
```

7.2.3 Match chain

A variant on the terminated match which allows a second match to be chained onto the last branch without any endmatch.

```
var y = list (1,2);
var x =
  match y with
  | #Empty => "Empty"
  | _ =>
    match y with
    | h ! Empty => h.str
    | _ =>
      match y with
      | h1 ! h2 ! Empty => h1.str + "," + h2.str
      ;

println$ x;
```

7.2.4 conditional chain

A variant on the terminated if/then/elif/else allowing chaining.

```
var x =  
    if c1 then r1  
    elif c2 then r2 else  
    if c3 then r3 else  
    r4  
;
```

7.3 Alternate conditional chain

```
var x = n / d unless d == 0 then 0;
```

7.4 Dollar application

A right associative low precedence forward apply operator taken from Haskell.

```
var x = str$ rev$ list$ 1,2,3;
```

7.5 Pipe application

A left associative low precedence reverse apply operator taken from C#.

```
var x = 1,2,3 |> list |> rev |> str;
```

7.6 Tuple cons constructor

A right associative cons operator for tuples. Allows concatenating an element to the head/front/left end of a tuple. Can also be used in a pattern match to recursively decode a tuple like a list.

```
var x = 3,4;
var y = 1 ,, 2 ,, x; // 1,2,3,4
```

7.7 N-ary tuple constructor

There is a non-associative n-ary tuple constructor consists of a sequence of expressions separated by commas.

```
var x = 1,2,3,4;
```

7.8 Logical implication

An operator for function `implies`.

```
var x = false implies true;
```

7.9 Logical disjunction

A chaining operator for function `lor`.

```
var x = true or false;
```

A shortcut enabled disjunction is available too:

```
fun f(x:int, y:int) =>
  y == 0 or else x / y > 1
;
println$ f(1,0), f(10,1), f(1,10);
```

This operation is very expensive. The second argument is translated to a closure and the system `orelse` function is called:

```
noinline fun orelse (x: bool, y:1->bool) : bool =>
  if x then true else #y
;
```

The closure and conditional ensure the second argument is only evaluated if the first is false.

7.10 Logical conjunction

A chaining operator for function `land`.

```
var x = true and false;
```

A shortcut enable conjunction is available too:

```
fun g(x:int, y:int) =>
  y != 0 and then x / y <= 1
;
println$ g(1,0), g(10,1), g(1,10);
```

This operation is very expensive. The second argument is translated to a closure and the system `andthen` function is called:

```
noinline fun andthen (x: bool, y:1->bool) : bool =>
  if x then #y else false
;
```

The closure and conditional ensure the second argument is only evaluated if the first is true.

7.11 Logical negation

A bool operator for function `lnot`.

7.12 Comparisons

Non-associative.

```
var x =
  a < b or
  a > b or
  a <= b or
  a >= b or
  a == b or
  a != b or
  1 in list(1,2,3)
  1 \in list (1,2,3)
;
```

7.13 Name temporary

Allows a subexpression to be named as a `val` by default or a `var`.

```
var x = a + (f y as z) + z; // equivalent to
val z = f y; var x = a + z + z;

var x = a + (f y as var k) + k; // equivalent to
var k = f y; var x = a + k + k;
```

Note that the `var` for ensures the subexpression is eagerly evaluated, before the containing expression.

7.14 Schannel pipe operators

Used to flow data through schannels from the source on the left to the sink on the right via processing units in between.

```
spawn_fthread$ source |-> filter |-> enhancer |-> sink;
```

This variant uses an iterator to stream data out of a data structure:

```
spawn_fthread$ list (1,2,3) >-> sink;
```

7.15 Right Arrows

Right associative arrow operators.

List cons operator.

```
var x = 1 ! 2 ! list (3,4);
```

Function types (type language only):

```
D -> C // Felix function
D --> C // C function pointer
```

7.16 Case literals

The case tag is only used in pattern matches. The sum or union type isn't required because it can be deduced from the match argument.

```
var a = match a with Some v => v | #None => 0;
```

The case constructor with integer caseno has two uses.

It creates a value of a sum type with no arguments:

```
var x = case 1 of 2; // aka true
```

or it is a function for a sum type variant with an argument:

```
var x = (case 1 of int + double) 4.2;
```

A case literal with a name instead of an integer constructs a variant instead:

```
typedef maybe = union { No; Yes of int; };
var x = (case Yes of maybe) 42;
```

The tuple projection function names a tuple projection:

```
typedef triple = int * long * string;
var snd = proj 1 of triple;
var y: int = snd (1, 2L, "3");
```

7.17 Bitwise or

Left associative.

```
var x = q | b;
```

7.18 Bitwise exclusive or

Left associative.

```
var x = q ^ b;
```

7.19 Bitwise and

Left associative.

```
var x = q & b;
```

7.20 Bitwise shifts

Left associative.

```
var x = a << b; // left shift  
var x = a >> b; // right shift
```

7.21 Addition

Chain operator. Non-associative for types. Left associative for expressions.

```
var x = a + b + c;
```

7.22 Subtraction

Left associative.

```
var x = a - b;
```


7.23 Multiplication

Chain operator. Non-associative for types. Left associative for expressions.

```
var x = a * b;
```

7.24 Division operators

Left associative. Not carefully: higher precedence than multiplication, unlike C!!

```
var x = a * b / c * d; // means
var x = a * (b / c) * d;

var x = a * b % c * d; // means
var x = a * (b % c) * d;
```

7.25 Prefix operators

```
var x = !a;
var x = -a; // negation
var x = ~a; // bitwise complement
```

7.26 Fortran exponentiation

Infix `**` is special syntax for function `@pow`. The left operand binds more tightly than `**` but the right operand binds as for prefixed operators or more tightly. Observe that:

```
var x = -a**-b; // means
var x = -(a**(-b));
```

preserving the usual mathematical syntax.

7.27 Felix exponentiation

Left associative. The right operand binds as deref operator or more tightly. Used for array notation in the type language.

```
var x = a ^ ix;
```

7.28 Function composition

Standard math notation. Left associative. Same precedence as exponentiation. Spelled `\circ`.

```
var x = f \circ g;
```

7.29 Dereference

For function `deref`.

```
var x = *p;
```

For builtin dereference operator:

```
var x = _deref p;
```

Note these usually have the same meaning however the function `deref` can be overloaded. If the overloaded definition is not to be circular it may use `_deref` when dereferencing pointers.

7.29.1 Operator `new`

Copies a value onto the heap and returns a pointer.

```
var px = new 42;
```

7.30 Whitespace application

Operator `whitespace` is used for applications.

7.30.1 General

```
var x = sin y;
```

7.30.2 Caseno operator

Returns the integer tag value of the value of an anonymous sum, union, or variant type.

```
var x = caseno true; // 1
var x = caseno (Some 43); // 1
```

7.30.3 Likelyhood

Indicates if a bool valued expression is likely or unlikely to be true. Used to generate the corresponding gcc optimisation hints, if available.

```
if likely (c) goto restart;
if unlikely (d) goto loopend;
```

7.31 Coercion operator

left associative. The right operand is a type.

```
var x = 1L :>> int; // cast
```

7.32 Suffixed name

The most general form of a name:

```
var x = qualified::name of int;
```

Used to name functions, with the right operand specifying the function argument type.

7.33 Factors

7.33.1 Subscript

Used for array and string subscripting. Calls function `subscript`. For strings, returns a character. If the subscript is out of range after adjustment of negative index, returns `char 0` and thus cannot fail.

```
var x = a . [ i ]; // i'th element
```

7.33.2 Substring

Calls function `substring`. Negative indices may be used to offset from end, i.e. -1 is the index of the last element. Out of range indices (past the end or before the start, after adjustment of negative indices) are clipped back to the end or start respectively.

```
var x = a . [first to past];
      // past is one past the last element referred to
```

7.33.3 Copyfrom

Calls function `copyfrom`. Copies from designated index to end. Supports negative indices and range clipping for strings.

```
var x = a . [to past]; // from the first
```

7.33.4 Copyto

Calls function `copyto`. Supports negative indices and range clipping for strings.

```
var x = a . [to past]; // from the first
```

7.33.5 Reverse application

Left associative.

```
var x = y .f; // means
var x = f y;
```

7.33.6 Reverse application with deref

Left associative

```
var x = p *. k; // means
var x = (*p) . k;
```

7.33.7 Reverse application with addressing

Left associative

```
var x = v &. k; // means
var x = (&v) . k;
```

7.33.8 Unit application

Prefix operator applies argument to empty tuple.

```
var x = #f; // means
var x = f ();
```

7.33.9 Addressing

Finds the pointer address of a variable. Means pointer to in type language.

```
var x : int = 1;
var px : &int = &x;
// address of x
// type pointer to int
```

7.33.10 C pointer

Used in type language for pointer to type or NULL.

```
var px : @char = malloc (42);
```

Note that this symbol is also used in fdoc as a markup indicator. Please keep out of column 1, do not follow with a left brace.

7.33.11 Macro freezer

Used to disable macro expansion of a symbol.

```
macro val fred = joe;
var x = fred + noexpand fred; // means
var x = joe + fred;
```

7.33.12 Pattern variable

Notation `v` Used in patterns to designate a `val` variable to be bound in the pattern matching.

```
var x =
  match y with
  | Some v => "Some " + v.str
  | #None => "None";
;
```

7.33.13 Parser argument

Notation `n` for some integer `@{n}`. In user defined syntax designates the `n`'th term of a syntax production.

7.34 Qualified name

A name in Felix has the form:

```
class1 :: nested1 :: ... :: identifier [ type1, type2, ... ]
```

where the qualifiers and/or type list may be elided. This is the same as C++ except we use `[]` instead of `@{<>}` for template argument types.

Chapter 8

Atoms

8.1 Record expression

```
var x = (name="Hello", age=42);
```

8.2 Alternate record expression

```
var x =  
  struct {  
    var name = "Hello";  
    var age = 42;  
  }  
;
```

8.3 Variant type

Denotes a variant type.

```
var x :  
  union {  
    Cart of double * double;  
    Polar of double * double;  
  }  
;
```

8.4 Wildcard pattern

Used in a pattern match, matches anything.

```
var x = match a with _ => "anything";
```

8.5 Ellipsis

Used only in C bindings to denote varags.

```
fun f: int * ... -> int;
```

8.6 Truth constants

```
false // alias for case 0 of 2
true  // alias for case 1 of 2
```

8.7 callback expression

??

```
callback [ x ]
```

8.8 Lazy expression

Function of unit.

```
var f = { expr };
var x = f ();
```

8.9 Sequencing

Function dependent on final expression.

```
var x = ( var y = 1; var z = y + y; z + 1 ); // equivalent to
var x = #{ var y = 1; var z = y + y; return z + 1; };
```


8.10 Procedure of unit.

```
var p = { println$ "Hello"; } // procedure
p ();

var f = { var y = 1; return y + y; }; // function
var x = f ();
```

8.11 Grouping

Parentheses are used for grouping.

```
var x = (1 + 2) * 3;
```

8.12 Object extension

```
var x = extend a,b with c end;
```

8.13 Conditional expression

```
var x =
  if c1 then a elif c2 then b else c endif
;
```

Chapter 9

Pattern Matching

The patterns are composed of these forms:

```
pat | pat | pat ...      // alternate matches
(v1, v2, ... )           // tuple match
h!t                       // list match
h,,t                      // tuple cons
Ctor                      // const union or variant match
Ctor v                    // nonconst union or variant match
(fld1=f1, fld2=f2, ...)  // record match
pat as v                  // assign variable to matched subexpression
pat when expr             // guarded match
pat1 | pat2               // match either pattern
999                       // integer match
"str"                     // string match
lit1 .. lit2              // range match
-                          // wildcard match
```

The guarded match only matches the pattern if the guard expression is true.

```
match x with
| (x,y) when y != 0 => ...
endmatch
```

The tuple as list cons match is a form of row polymorphism where the first element of a tuple and the remaining elements considered as a tuple are matched.

A good example of this is found in the library [here](#) which allows printing a tuple of arbitrary number of components, indeed, this facility was implemented precisely to allow this definition in the library.

Record matches succeed with any record containing a superset of the specified fields.

As well as integer and string matches, a literal of any type with an equality and inequality operator can be matched. In addition, if there is a less than or equal operator \leq an inclusive range match can be specified.

Chapter 10

Definitions

Syntax

10.1 Functions

A felix function definition takes one of three basic forms:

```
fun f (x:int) = { var y = x + x; return y + 1; }  
fun g (x:int) => x + x + 1;  
fun h : int -> int = | x => x + x + 1;
```

The first form is the most general, the body of the function contains executable statements and the result is returned by a return statement.

The second form is equivalent to a function in the first form whose body returns the RHS expression.

The third form specifies the function type then the body of a pattern match. It is equivalent to

```
fun h (a:int) = { return match a with | x => x + x + 1 endmatch; }
```

The first two forms also allow the return type to be specified:

```
fun f (x:int) : int = { var y = x + x; return y + 1; }  
fun g (x:int) :int => x + x + 1;
```

Functions may not have side effects.

All these function have a type:

```
D -> C
```

where D is the domain and C is the codomain: both would be `int` in the examples.

A function can be applied by the normal forward notation using juxtaposition or what is whimsically known as operator whitespace, or in reverse notation using operator dot:

```
f x
x.f
```

Such applications are equivalent. Both operators are left associative. Operator dot binds more tightly than whitespace so that

```
f x.g    // means
f (g x)
```

A special notation is used for application to the unit tuple:

```
#zero // means
zero ()
```

The intention is intended to suggest a constant since a pure function with unit argument must always return the same value.

This hash operator binds more tightly than operator dot so

```
#a.b // means
(#a).b
```

10.2 Pre- and post-conditions

A function using one of the first two forms may have pre-conditions, post-conditions, or both:

```
fun f1 (x:int when x > 0) => x + x + 1;
fun f2 (x:int) expect result > 1 => x + x + 1;
fun f3 (x:int when x > 0) expect result > 1 => x + x + 1;
fun f4 (x:int when x > 0) : int expect result > 1 => x + x + 1;
```

Pre- and pos-conditions are usually treated as boolean assertions which are checked at run time. The compiler may occasionally be able to prove a pre- or post-condition must hold and elide it.

The special identifier `result` is used to indicate the return value of the function.

10.3 Higher order functions

A function may be written like

```
fun hof (x:int) (y:int) : int = { return x + y; }
fun hof (x:int) (y:int) => x + y;
```

These are called higher order functions of arity 2. They have the type

```
int -> int -> int    // or equivalently
int -> (int -> int)  //since -> is right associative.
```

They are equivalent to

```
fun hof (x:int) : int -> int =
{
  fun inner (y:int) : int => x + y;
  return inner;
}
```

that is, a function which returns another function.

Such a function can be applied like

```
hof 1 2 // or equivalently
(hof 1) 2
```

since whitespace application is left associative.

10.4 Procedures

A function which returns control but no value is called a procedure. Procedures may have side effects.

```
fun show (x:int) : 0 = { println x; }
proc show (x:int) { println x; }
proc show (x:int) => println x;
```

The second form is a more convenient notation. The type 0 is also called `void` and denotes a type with no values.

A procedure may return with a simple return statement:

```
proc show (x:int) { println x; return; }
```

however one is assumed at the end of the procedure body.

Procedures can also have pre- and post-conditions.

A procedure may be called like an application, however it must be a whole statement since expressions of type `void` may not occur interior to an expression.

```
show 1;
1.show;
```

If a procedure accepts the unit argument, it may be elided:

```
proc f () => show 1;
f; // equivalent to
f ();
```

10.5 Generators

A generator is a special kind of function which is allowed to have side effects. It is defined similarly to a function, but using the binder `gen` instead of `fun`:

```
var seqno = 1;
gen seq () { var result = seqno; ++seqno; return result; }
```

When a generator is directly applied in an expression, the application is replaced by a fresh variable and the generator application is lifted out and assigned to the variable. For example:

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

will always print an even number because it is equivalent to

```
var tmp = #seq;
println$ twice tmp;
```

Therefore even if `twice` is inlined we end up with the argument to `println` being `@{tmp+tmp}` and not `@{#se` which would print an odd number.

10.5.1 Yielding Generators

A generator may contain a `yield` statement:

```
gen fresh() {
  var x = 1;
  while x < 10 do
    yield x;
    ++x;
  done
  return x;
}
```

In order to use such a yielding generator, a closure of the generator must be stored in a variable. Then the generator may be called repeatedly.

```
var g = fresh;
for i in 1 upto 20 do
  println$ i,#g;
done
```

This will print pairs (1,1), (2,2) up to (10,10) then print (11,10), (12,10) up to (20,10).

The `yield` statement returns a value such that a subsequent call to a closure of the generator will resume execution after the `yield` statement. Therefore `yield` is a kind of cross between a `return` and a subroutine call.

If a generator executes a `return` statement, that is equivalent to yielding a value and setting the resume point back to the `return` statement, in other words `return expr;` is equivalent to

```
while true do yield expr; done
```

Yielding generators should not be called directly because they will always start at the beginning with a fresh copy of any local variables used to maintain state.

Function closures differ from generator closures in that the closures is cloned before every application to ensure that the initial state is fresh.

Yielding generators are primarily intended to implement iterators, that is, to provide lazy lists or streams.

10.6 Conversion Operators

Felix provides a special notation which allows an identifier naming a type to return a value of that type:

```
typedef cart = dcomplex;
typedef polar = dcomplex;
ctor cart (x:double, y:double) => dcomplex (x,y);
ctor polar (r: double, theta: double) =>
    dcomplex (r * sin theta, r * cos theta)
;
var z = cart (20.0,15.0) + polar (25.8, 0.7 * pi);
```

Such conversion operators are sometimes (incorrectly) called constructors. The constructions are equivalent to

```
fun _ctor_cart (x:double, y:double) : cart => dcomplex (x,y);
fun _ctor_polar (r: double, theta: double): polar =>
    dcomplex (r * sin theta, r * cos theta)
;
```

When a type with a simple name is applied to a value, Felix tries to find a function with that name prefixed by `_ctor_` instead.

Note that Felix generates a constructor for struct and cstruct types automatically with argument type the product of the types of the structure fields.

10.7 Special function apply

When Felix finds and application

```
f a
```

where `f` is a value of type `F` which is not a function (or `C` function) type, Felix looks instead for a function named `apply` with argument of type:

```
F * A
```

where `A` is the type of `a`. For example

```
fun apply (x:string, y:string) => x + y; // concat
var x = "hello " "world"; // apply a string to a string
```

10.8 Dynamic Objects

Felix provides an object system with syntax based on Java, and technology based on CLOS.

An object is a record of function closures, closed over the local scope of a constructor function that returns the record.

```
interface person_t {
  get_name: 1 -> string;
  set_age: int -> 0;
  set_job : string -> 0;
  get_job : 1 -> string;
}

object person (name:string, var age:int) implements person_t =
{
  var job = "unknown";
  method fun get_name () => name;
  method proc set_age (x:int) { age = x; }
  method fun get_job () => job;
  method proc set_job (x:string) { job = x; }
}

var john = person ("John", 42);
println$ #(john.name) + " is " + #(john.age).str;
```

The entity `person` is a function which when called with its argument of name and age returns a record of type `person_t` consisting of closures of the functions and procedures marked as method in its definition.

Since functions hide their local variables the object state is hidden and can only be accessed using the methods.

The `implements` clause is optional.

Objects provide an excellent way for a dynamically loaded shared library to export a set of functions, only the object function needs to be exported so it has a C name which can be linked to with `dlopen`.

Part II

Gross Structure

Chapter 11

Program structure

A Felix program consists of a nominated root parse unit and the transitive closure of units with respect to inclusion.

The behaviour of this system consists of the action of the initialisation code in each unit, performed in sequence within a given unit, with the order of action between units unspecified.

11.1 Parse Unit

A parse unit is a file augmented by prefixing specified import files to the front. These consist of a suite of grammar files defining the syntax and other files defining macros.

By convention syntax files have the extension `.fsyn`, and other import files have the extension `.flxh`.

With this augmentation all parse units in a program are independently parsed to produce an list of statements represented as abstract syntax, denoted an AST (even though it is a list of trees, not a single tree).

11.2 AST

The program consists of the concatenation of the ASTs of each parse unit, resulting in a single AST, which is then translated to a C++ translation unit by the compiler.

The ASTs to be concatenated are those parsed from the parse units specified by the transitive closure of include directives.

The order of concatenation is unspecified. Note that include directives do *not* cause an AST to be inserted at the point where the directive is written.

Parse units can be and are parsed independently. The contents of a parse unit cannot influence the parsing of another.

11.3 Include Directive

11.3.1 Syntax

An include directive has the syntax:

```
include "filename";  
include "dirname/filename";  
include "../filename";
```

where the filename is a Unix relative filename, may not have an extension, and may not begin with or contain `..` (two dots).

If the filename begins with `./` then the balance of the name is relative, a sibling of the including file, otherwise the name is searched for on an include path.

In either case, a search succeeds when it finds a file with the appropriate base path in the search directory with extension `.flx` or `.fdoc`. If both files exist the most recently changed one is used. If the time stamps are the same the choice is unspecified.

11.3.2 Effect

An include directive is included in the AST generated by the parser.

An extraction process extracts the set of directives and locates the files specified and adds them to a set of files which must be parsed.

Unparsed files are upgraded to parse units and parsed, and then the extraction process again applied.

When all the files have been processed the resulting ASTs are concatenated.

Felix uses a caching scheme to avoid parsing all the files every time.

11.4 Grammar

The Felix grammar is part of the library. It is notionally prefixed to each file to be processed prior to any import files to specify the syntax with which the file is to be parsed and translated to an AST.

The grammar uses an augmented BNF like syntax with parse actions specified in R5RS Scheme.

The resulting S-expressions are translated to an intermediate form and then into an internal AST structure.

The parser is a scannerless GLR parser with significant extensions.

Chapter 12

Parsing

Felix has the most advanced parser of any production programming language. The parser is extensible, and Felix makes good use of this. The core, hardcoded grammar, called the *bootstrap grammar* does not provide the syntax for the Felix programming language the average production programmer is interested in.

Therefore, we recommended this chapter be lightly scanned or even skipped, until such time as a deeper understanding is required. Nevertheless we feel obligated to present some details now, since the ability to define Domain Specific Sub-Languages is one of the key advantages of the Felix system.

In Felix, the usual programming language is defined by a grammar in the library in user space. The compiler interprets the grammar to produce a parser for the scope in which the grammar is active. Grammar extension activations are properly scoped.

As with any modern programming language it is sometimes difficult to distinguish features present in the core language and those defined in the standard library. In particular the semantics of some features in the library could not be defined in the core language, but their syntactic representation can be. This is most evident in the C programming language where, for example, Posix defines semantics for multi-threading which are represented by library functions which could not possibly be defined in C.

However in Felix the situation is even more extreme, since the very language itself is defined in the library. What is more, most of the definitions are real Felix code: we are not just leveraging the library for syntactic modelling of semantics we could not otherwise define, but are actually presenting the definitions.

Naturally this is not possible without a core built-in parsing system and semantics, this called the bootstrap parser and the first stage abstract machine.

It reads the library syntax specifications to extend itself to the full blown Felix

language described in this document and maps the input syntax onto terms of the abstract machine.

12.1 Bootstrap Parser Grammar

Compiler Reference

The bootstrap parser is a hard coded parser which translates Felix EBNF like syntax specification into a parser automaton, which in turn translates Felix programs.

12.1.1 Parser Processing

Felix use the Dypgen parsing system, which is a run time extensible GLR parsing engine with extensions.

When the parsing of the production of a non-terminal is complete, the associated action code is executed.

The action code is a string containing a procedure written in R5RS Scheme. The OCS Scheme language processor is used for the processing.

Using predefined variables representing the s-expressions associated with the evaluation of the actions associated with the symbols of the production, the client action code constructs a new s-expression representing the intended AST for parsing that production.

Example

Here is a complete, self contained, working example:

```
// define the DSSL Ceiling
syntax Ceiling {
  satom := "\lc" sexpr "\rc" =>#
    "`(ast_apply ,_sr (,(noi 'ceiling) (,_2)))"
  ;
}

// define the ceiling operation
fun ceiling (x:double) = "ceil($1)"
  requires '#include <math.h>'
  ;

// make the DSSL available for use
open syntax Ceiling;

// Use it
println$ \lc 42.3 \rc;
```

In the above example, a production for the `satom` non-terminal, recognises a the TeX like expression `\lc e \rc`, where the e in the middle is any expression. The Scheme code returns an s-expression consisting of the `ast_apply` symbol, followed by the value in the current source reference variable `_sr`, then a pair consisting of the function name `ceiling` to be executed and its argument `_2` which is the s-expression returned from the second position in the production.

The Scheme function `noi` converts a symbol into an s-expression which is the AST representation of a Felix name. The `ast_apply` s-expression is the AST representation of a Felix application.

We then define the function `ceiling` in Felix by lifting it from the ISO C math library.

Finally we have to open the DSSL so the new grammar becomes active, and then when we write an expression like `\lc 42.3 \rc` the parser translates it into a function application.

The Felix standard library actually defines `\lceil` and `\rceil` and maps $[e]$ to the function `ceil` so we couldn't use them in the example.

12.1.2 Scheme Actions

Provided the returned AST is acceptable to subsequent processing steps, this mechanism is safe provided the executed Scheme code is purely functional.

If the Scheme actions have side effects, care must be taken. Dypgen parsers may parse the same text more than once. At any point in the parsing process the next symbol is used to determine a possible set of parses, and all these parses are then explored simultaneously.

Most of these parses will fail at some point in the future, and the AST constructed at that point is discarded. Except in special circumstances, if more than one parse succeeds, the Felix translator will fail with a parsing ambiguity.

The point, however, is that side effects from exploratory parses cannot be discarded, nor can the order in which the parses is performed be predicted. Although the parses are, in principle, simultaneous, Dypgen actually processes a single token for each parse in sequence, suspending the parsing of one thread until all parses have processed the token, then resuming with the next token, having discarded failed parses, but also possibly spawning a new set of parses. In other words, the parsing is performed using coroutines which ensure that the input is only consumed in a linear fashion without any backtracking.

If side effects are necessary, then they should usually be idempotent. For example setting a variable to a fixed value is an idempotent operation. Incrementing a variable is not.

There is, however, a special exception to this rule. Top level statements are guaranteed to be parsed exactly once. Although the start symbol is defined recursively, the recursion is optimised to a plain loop. This is not necessarily the case for nested statements!

12.1.3 Spawning Choices

When presented with a non-terminal to parse, a GLR parser notionally processes all alternatives simultaneously. Many of these branches will fail, indeed, if the parse is to be ambiguous all but one should eventually fail. This method could be called purely speculative parsing.

However, Dypgen does not actually try all the alternatives. Some alternatives must start with a particular token or one of a set of tokens. Instead of starting the parse and failing after examining the next token, Dypgen uses an LALR1 predictive parser to narrow down the set of alternatives. The LALR1 automaton is a finite state machine which given the next token supplies a set of possible parse which might continue. Although LALR1 is not accurate, it is conservative, and only rejects alternatives that could not possibly work.

12.1.4 No keywords

In most language, there is a severe lack of usable symbols. To compensate for this, certain identifiers called keywords are specified as symbols instead. The tokeniser checks if an identifier is a keyword and returns a suitable token if it is, otherwise it returns an identifier.

Unfortunately, because keywords encroach on user space, many languages such as C++ try to minimise the set used, and in the case of a language extension, introducing a new keyword may break existing code. So there is tendency to reuse the keyword in a different context, leading to the nightmare that is evident in C++.

Felix also has keywords, but they are not global. The Felix parser is scannerless which means it does not require a separate tokeniser. Instead tokenisation is local to each parser state, and in particular the identifiers used as keywords are only recognised in this state. Felix also provides a consistent way to unkeyword with a special form of literal to force the parser to recognise an identifier instead of a keyword.

```
// var = 1;
// Syntax error, variable init expected

var var = 1;
// OK, declares identifier "var" as a variable
```

```
n"var" = 1;
// allows assigning to identifier named "var".
```

12.1.5 Predefined Variables

When scheme code is executed certain global variables are automatically predefined.

- The variable `_sr` is set to contain the current location in the source file. It consists of a scheme list of the current original source filename as a string, then the starting line, starting column, ending line, and ending column of the text being parsed by the current production.
- The variables `_1`, `_2` etc refer to the scheme s-expressions associated with the parse of the `n`'th symbol in a production.

Together these variables allow the scheme action code of a production to construct an scheme s-expression representing the AST desired for parsing the associated production.

12.1.6 SCHEME statement

The keyword `SCHEME` followed by a string literal and semi-colon causes the embedded R5RS scheme interpreter to execute the text of the string literal as scheme code during parsing.

It is primarily intended to introduce global procedure definitions as helper functions to be used in the parser action codes.

However it may contain arbitrary Scheme code, including, for example diagnostic output, an interpreter, or anything else. The return value of the execution is discarded.

```
SCHEME """
  (display "Hello parsing world\n")
""";
```

The `SCHEME` statement may be used anywhere a statement is allowed, it results in a empty statement in the AST. At the top level, the side-effects will only be seen once. However if nested in a grouping statement, multiple explorations of the possible parses of that statement may occur, leading to multiple observations of the side-effects.

12.1.7 STMT statement

The keyword `STMT` followed by a string literal and semi-colon causes the embedded R5RS scheme interpreter to execute the text of the string literal as scheme

code during parsing.

The result of the execution is returned and appended to the current list of statements being parsed.

It is primarily intended to provide an mechanism whereby arbitrary Felix statements can be generated under program control, in other words it is a very sophisticated and properly structured macro processor. The client must have intimated knowledge of the which s-expressions are accepted by the subsequent compiler processing phases and what their semantics are.

Care must be taken if the result of the execution is not invariant. For example if the time, a file from the file system, or other such variable entity is used in the computation, the Felix dependency checking system will not recognise the external dependencies and the cached version of the current file's parse may be used.

```
for i in 0..10 do
  println "Stopping now";
  STMT `(ast_halt ,_sr "STOPPED")';
done
```

12.1.8 syntax statement

The **syntax** statement consists of the word **syntax** followed by an identifier and a list of syntax definition statements enclosed in curly braces { and }. The entity defined is called a *Domain Specific Sub-Language* or just *dssl*.

The given definitions are recorded as a set under the given name but are not activated. For example here is the DSSL for the Felix macros:

```
syntax macros {
  requires expressions, statements, list;

  stmt := "macro" "val" snames "=" sexpr ";" =>#
    `(ast_macro_val ,_sr ,_3 ,_5)";

  stmt := "forall" sname "in" sexpr "do" stmt* "done" =>#
    `(ast_macro_forall ,_sr ( ,_2) ,_4 ,_6)"
;
}
```

12.1.9 requires statement

Within a dssl definition, the **requires** statement consists of the word **requires** followed by a comma separated list of identifiers, and terminated by a semi-colon.

The names in the list identify other dssls on which this dssl depends, so that subsequently activating the dssl will also cause the dependent dssls, recursively, to be activated.

12.1.10 Priority statement

The `priority` statement introduces a set of priority names which can be used for indexed non-terminals, and specifies a total ordering between them, sometimes called a precedence. Here are the priorities used for non-terminal `x` representing expressions in the Felix grammar, lower priority means weaker binding:

```
priority
  let_pri <
  slambda_pri <
  spipe_apply_pri <
  sdollar_apply_pri <
  stuple_cons_pri <
  stuple_pri <
  simplies_condition_pri <
  sor_condition_pri <
  sand_condition_pri <
  snot_condition_pri <
  stex_implies_condition_pri <
  stex_or_condition_pri <
  stex_and_condition_pri <
  stex_not_condition_pri <
  scomparison_pri <
  sas_expr_pri <
  ssetunion_pri <
  ssetintersection_pri <
  sarrow_pri <
  scase_literal_pri <
  sbor_pri <
  sbxor_pri <
  sband_pri <
  sshift_pri <
  ssum_pri <
  ssubtraction_pri <
  sproduct_pri <
  s_term_pri <
  sprefixed_pri <
  spower_pri <
  ssuperscript_pri <
  srefr_pri <
  sapplication_pri <
  scoercion_pri <
```

```

sfactor_pri <
srcompose_pri <
sthenname_pri <
satomic_pri
;

```

12.1.11 Non-terminal definition

A nonterminal definition, in its most basic form, consists of the non-terminal name, followed by `:=` followed by a list of alternatives separated by a `|` followed by a semi-colon `;`.

An alternative is a sequence of grammar symbols followed by `=>#` followed by a string. The sequence of symbols is called the rhs (right hand side) of the production, and the string is called the action. The string must encode an R5RS Scheme procedure which returns an s-expression representing the AST subtree desired to be inserted in the syntax tree.

The non-terminal name on the left is either an identifier, or, an identifier followed by `[`, an priority identifier, followed by `]`. A priority identifier is an identifier appearing in a priority specification.

An grammar symbol is either a predefined atom, a string which is required to match the input stream, a non-terminal expression, or a grammar expression enclosed in parentheses (`(` and `)`), a macro invocation or a grammar symbol followed by one of `*` representing 0 or more occurrences, `+` representing one or more occurrences, or `?` representing 0 or 1 occurrence.

A non-terminal expression is either a name, or a name followed by a priority indicator, or a macro invocation. A priority indicator is a priority name enclosed in either `[` and `]` which indicates a fixed non terminal, or `[>` and `]` which indicates all the non-terminals with a greater priority than the specified one.

A macro invocation is a name optionally followed by `::` and another name, followed by a sequence of macro arguments. A macro argument is a symbol enclosed in `<` and `>`.

12.1.12 Preprocessing productions

Felix pre-processes productions to reduce them to a sequence of basic symbols: either a terminal or non-terminal symbol.

- Expressions contained in parentheses are replaced by a generated non-terminal name and the non-terminal is defined by the enclosed sequence, its action is to return a list of the ASTs of each symbol.
- A non-terminal postfixed by `*` or `+` is replaced by a generated non-terminal defined to recursively generate a list of the non-terminal, with either 0 or

more or 1 or more entries in the list, respectively.

- A non-terminal postfixed by ? returns the a list containing just the s-expression returned by the non-terminal if it is matched, otherwise an empty list.
- Macro calls are expanded. This done by replacing each occurrence of a macro parameter with the corresponding argument of the call.

12.1.13 Grammar Macros

A grammar macro is defined by the same syntax as a non-terminal, except that a sequence of parameter names is added on the lhs after the first name. The first name may not be indexed.

On expansion due to an invocation of a macro, each parameter name in the production is replaced by the non-terminal argument.

As an example, suppose we wish to define a comma separated list of strings, a comma separated list of integers, and a comma separated list of names. By using a macro, the common structure of the productions can be captured: a macro is basically a grammar function. The following is actually found in the Felix library:

```
syntax list
{
  seplist1 sep a := a (sep a)* =># '(cons _1 (map second _2))';
  seplist0 sep a = seplist1<sep><a>;
  seplist0 sep a := sepsilon =># '()';
  commalist1 a = seplist1<","><a>;
  commalist0 a = seplist0<","><a>;

  snames = commalist1<sname>;
  sdeclnames = commalist1<sdeclname>;
}
```

Although grammar macros were primarily intended as basic boilerplate templates, macros can also be passed as arguments to another macro, thereby allowing higher order macros. Furthermore, since arguments are presented in a curried form, closures can also be created, that is, macros with some but not all parameters fixed.

In particular, a macro can actually be passed to itself! Although this seems weird it is in fact a very high power technique used in functional programming and type systems, known as *open recursion*.

Open recursion allows a recursive structure to be extended so that the extensions *backport* to the core, in other words, the extensions are covariant. This solves a

significant problem in grammar extension. Normally in a grammar extension is easily done by simply adding extra productions for a non-terminal.

The problem is that the nonterminal is then effectively a global variable and extensions from different sources can clash. However the use of a global automatically assures we have covariant extensions, since existing productions using the non-terminal will automatically parse the extensions since they're now alternatives of the non-terminal.

The problem with clashing is only a symptom of the deeper problem: the mechanism does not obey Meyer's Open/Closed principle.

Open recursion solves this problem. The recursions are replaced by parameters, flattening the term structure and allowing extensions in a tree like manner. The programmer then closes the loop at any desired point with a new definition formed by passing the flat term to itself.

More generally, a term structure with more details such as a language with statements, value expressions, type expressions, and patterns, allows any combination of the extensions to be used with nested (recursive) structures equal to the top level or constrained to some lower level as required.

This method obeys the Open/Closed principle because each closure is specific and cannot be further extended, yet the open part of the system remains open for extension.

12.1.14 `regdef` statement

The `regdef` statement defines an alias for a regular expression. A regular expression may consist of:

- An atom, being
 - an alias for a regular expression,
 - a string, representing that string,
 - a string in square brackets [] representing one of the characters in the string.
 - an underscore _ representing any character
 - a dot . representing any character except newline
 - a decimal integer representing the character with that ordinal
 - a range consisting of pair of decimal integers separated by a dash and enclosed in square brackets [and] representing one of the characters in the range, inclusive
 - a regular expression in parentheses ()
- a repetition consisting of an atom optionally followed by

- a * for zero or more occurrences
- a + for one or more occurrences
- a ? for zero or one
- a sequence of repetitions
- A list of alternatives separated by a vertical bar | each being a sequence

```
regdef octdigit = ['01234567'];
regdef linefeed = 10;
regdef hichar = [128-255];
regdef white = space | tab;
regdef bin_lit = '0' ('b' | 'B') (underscore? bindigit) +;
```

12.1.15 literal statement

The `literal` statement associates a non-terminal with a regular expression.

The regular expression is formed by replacing all aliases with their regular definitions, recursively, until no aliases are left.

For example, from the library:

```
regdef Cstring = cstring_literal;
literal Cstring =>#
"""
    (let*
      (
        (ftype "cstring")
        (iv (decode-string _1))
        (cv (c-quote-string iv))
      )
      `(ast_literal ,_sr ,ftype ,iv ,cv)
    )
  """;
sliteral := Cstring =># "_1";
```

Assuming `cstring_literal` is an alias for the regular definition for a C style string, we first create a final alias with the name of the desired non-terminal `Cstring`.

Then we define `Cstring` as a non-terminal with the `literal` statement, giving the required action code. The layout shown is the standard form for literals in the AST. the first field is the source reference `_sr`, the second field `ftype` is the Felix type of the literal. The third field is the internal representation of the value. The last field is a string representing the value in C++.

12.1.16 open syntax statement

The **open syntax** statement consists of the words **open syntax** followed by a comma separated list of names.

It simultaneously activates the transitive closure with respect to **requires** statements, of dssls previously recorded by **syntax** statements identified by the list of names.

12.2 Felix Grammar syntax

[Library Package Grammar directory](#)

Chapter 13

Modules

Every Felix program is encapsulated in a module with the name being a mangle of the basename of the root unit. The mangling replaces characters in the filename with other characters so that the module name is a valid ISO C identifier.

13.1 Special procedure `flx_main`

A program module may contain at most one top level procedure with no arguments, exported as `flx_main`. After initialisation code suspends or terminates, this procedure is invoked if it exists. It is the analogue of `main` in C++ however it is rarely used: side-effects of the root unit initialisation code are typically used instead.

A simple example:

```
println "Init";
var i,o = mk_ioschannel_pair[int]();
write (o,42);

export proc flx_main()
{
  println$ "main " + (read i).str;
  println$ "done ..";
}

println$ "Init done";
```

produces output:

```
Init  
main 42  
done ..  
Init done
```

Note that `flx_main` must be exported to ensure that an `extern "C"` symbol is created by the linker.

13.2 Libraries

In Felix a library is a root unit together with its transitive closure with respect to inclusion, which does not contain a top level exported `flx_main`.

A program unit can be augmented by a set of libraries which are then considered as if included, but without an include directive being present.

Chapter 14

Lexicology

All Felix files are considered to be UTF-8 encoded Unicode.

Felix uses a scannerless parser, there are no keywords.

14.1 Comments

There are lexical commenting methods for *.flx files. Comments are treated as white space separators. For example

```
println$ f/**/x; // parsed as f x not fx
```

The two forms of lexical comments are exclusive, once the parser is scanning one kind of comment the other is not recognised.

14.1.1 C++ comments

C++ style comments consist of // followed by all the characters up to and including the next newline character.

14.1.2 Nested C comments

C style comments consist of the lead in sequence /* followed by all the characters up to and including the balancing exit sequence */. These comments can span multiple lines and can be nested. When scanning comments lead in and exit sequences are recognised as such even in strings.

14.2 Layout

Felix treats code points 0 through 32 (space) as whitespace which may be used freely between symbols. Whitespace is significant in strings, however, and new-line is a terminator for C++ style comments.

14.3 File inclusion

There is (deliberately) no support in the Felix language for lexical (physical) file inclusion. Inclusions are processed at the AST level instead, allowing files to be parsed independently of other files. However command line switches can be used to prepend files or sets of files to the command input file, in particular the grammar and some standard macros are notionally inserted.

14.4 fdoc files

As well as *.flx files, the Felix language processor can directly process *.fdoc files using a limited subset of available fdoc commands.

fdoc files are processed slightly differently to *.flx files. The translator begins treating the file in comment mode, so all text is ignored up to a Felix leadin code.

14.4.1 Uncomments

A felix uncomment switches to processing lines as Felix program code. It consists of the line @felix, and is terminated by any line starting with @.

```
@title This is an fdoc.
@h1 Fdocs are documents.
They can contain code:
@felix
var x = 1;
@
Which defines a variable and
@felix
println$ x;
@
which prints it.
```

14.4.2 #line directive

Felix provides support for programs that generate Felix code by allowing C style #line directives. Such a directive consists of the characters #line at the start of a line, followed by whitespace, a decimal number indicating the line number in

the original source, and optionally whitespace followed by a filename in double quotes.

If the filename is present the parser original source filename is set to it. The line number sets the line number, 1 origin, so that the next source line will be taken to be obtained from the original source file at that line number.

Felix provides two standard programs which make use of this facility: `flx_tangle` and `flx_iscr` both of which are literate programming tools which extract source code from mixed code and comments.

In the event of a compilation error, Felix will specify that the error occurred at a location in the original source file, as indicated by `#line` directives. An example:

```
#line 42 "anerror.fdoc"
var x = error;
```

If this program is processed by Felix the error on the second line will be reported as an error on line 42 of the file `anerror.fdoc`.

14.5 `#!` directive

If the first line of an `*.flx` file starts with `#!` then the line is ignored. This allows a file on a Unix system marked executable to specify its natural translator so that the file may be run directly as a program. On Linux you should use:

```
#!env /usr/local/lib/felix/felix-latest/host/bin/flx
```

assuming you have a standard install and have linked `felix-latest` to a directory containing an installed version of Felix such as

```
%/usr/local/lib/felix/felix-2016.05.25%.
```

This is the standard way to refer to the most recent version of Felix installed on Unix systems. Note the path name to the translator on Unix systems must be an absolute path for security reasons.

14.6 Identifiers

Library Reference

Felix has three kinds of basic identifiers, plain identifiers, which are an enhanced variant of standard C identifiers, TeX identifiers, which are encodings of mathematical symbols in the style of \TeX , and some ascii-art character sequences normally use for punctuation or operators which are also recognised as names.

14.6.1 Plain Identifiers

A plain identifier starts with a letter or underscore, then consists of a sequence of letters, digits, dash (-), apostrophe ('), has no more than one apostrophe or dash in a row, except at the end no dash is allowed, and any number of apostrophies.

```
Ab_cd1  a' b-x
```

Identifiers starting with underscore are reserved for the implementation.

A letter may be any Unicode character designated for use in an identifier by the ISO C++ standard. In practice, all high bit set octets are allowed. Identifiers are uniquely identified by their sequence of ISO-10646 (Unicode) code points, alternate encodings of the same glyph are distinct.

14.6.2 TeX Identifiers

A TeX identifier starts with a slash and is followed by a sequence of letters.

Here is a partial table of [TeX Symbols](#) recognised by the grammar as identifiers with undefined semantics but pre-assigned kind and precedence.

14.7 Operator Identifiers

Felix allows some operators to be used as an identifier. For example you can write:

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

to define addition on int in C. Symbols recognised by the parser such as + are usually mapped to functions with the same name as the operator.

These operators are recognised as identifiers by the parser in positions where an identifier is expected:

```
+  -  *  /  %  ^
\& \| \^ ~
=  += -= *= /= %= ^= &= |= <=>=
<  > == != <= >= << >>
```

14.7.1 Special identifiers

The special string literal with a "n" or "N" prefix is a way to encode an arbitrary sequence of characters as an identifier in a context where the parser might interpret it otherwise. It can be used, for example, to define special characters as functions. For example:


```
typedef fun n"@ (T:TYPE) : TYPE => cptr[T];
```

14.8 Boolean Literals

There are two literals of type `bool`, namely `true` and `false`.

14.9 Integer Literals

Library Reference

An plain integer literal consists of a sequence of digits, optionally separated by underscores. Each separating underscore must be between digits.

A prefixed integer literal is a plain integer literal or a plain integer literal prefixed by a radix specifier. The radix specifier is a zero followed by one of the letters `bB` for binary radix, `oO` for octal radix, optionally one may use `dD` for decimal radix, although this is the default, and `xX` for hexadecimal radix.

An underscore is permitted after the prefix.

The radix is the one specified by the prefix or decimal by default.

The digits of an integer consist of those permitted by the radix: `01` for binary, `01234567` for octal, `0123456789` for decimal, `0123456789abcdefABCDEF` for hex.

Note there are no negative integer literals.

A type suffix may be added to the end of a prefixed integer to designate a literal of a particular integer type, it has the form of an upper or lower case letter or pair of letters usually combined with a prefix or suffix `u` or `U` to designate an unsigned variant of the type. The allowed lower case suffices are:

```
t s l ll
ut us u ul ull
tu su lu llu
i8 i16 i32 i64
u8 u16 u32 u64
p d j
zu pu du ju
uz up ud uj
```

In addition, one or more letters may be upper case, except that `lL` and `Ll` are not permitted.

There is a table of the types [Table 30.1 Felix Integer Types](#).

Note the suffices do not entirely agree with C.

14.10 Floating point literals

Library Reference

Floating point literals follow ISO C89, except that underscores are allowed between digits, and a digit is required both before and after the decimal point if it is present.

The mantissa may be decimal, or hex, a hex mantissa uses a leading 0x or 0X prefix optionally followed by an underscore.

The exponent may designate a power of 10 using E or e, or a power of 2, using P or p.

A suffix may be F,f,D,d, L or l, designating floating type, double precision floating type, or long double precision floating type.

```
123.4
123_456.78
12.6E-5L
0xAf.bE6f
12.7p35
```

There is a table of the operators [Table 30.5 Floating Point Operators](#).

14.11 String like literals

Library Reference

14.11.1 Standard string literals

Generally we follow Python here. Felix allows strings to be delimited by; single quotes ', double quotes ", triped single quotes ''' or tripled double quotes """".

The single quote forms must be on a single line.

The triple quoted forms may span lines, and include embedded newline characters.

The complete list of special escapes is shown in table [Table 14.1 String Escapes](#).

14.11.2 Raw strings

A prefix "r" or "R" on a double quoted string or triple double quoted string suppresses escape processing, This is called a raw string literal.

NOTE: single quoted string cannot be used, because this would clash with the use of single quotes/apostrophies in identifiers.

Table 14.1: String Escapes

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

14.11.3 Null terminated strings

A prefix of `"c"` or `"C"` specifies a C NTBS (Nul terminated byte string) be generated instead of a C++ string. Such a string has type `+char` rather than `string`.

14.11.4 Perl interpolation strings

A literal prefixed by `"q"` or `"Q"` is a Perl interpolation string. Such strings are actually functions. Each occurrence of `$(varname)` in the string is replaced at run time by the value `"str varname"`. The type of the variable must provide an overload of `"str"` which returns a C++ string for this to work.

```
var x = 1;
var y = 3.2;
println$ q"x=$(x), y=$(y)";
```

14.11.5 C format strings

A literal prefixed by a `"F"` or `"F"` is a C format string.

Such strings are actually functions.

The string contains code such as `"C"` format specifiers.

```
var x = 1;
var y = 3.2;
println$ f"x=%03d, y=%4.1f, s=%S" (x,y,"Hello");
```

Variable field width specifiers "*" are not permitted.

The additional format specification is supported and requires a Felix string argument.

If `vsnprintf` is available on the local platform it is used to provide an implementation which cannot overrun. If it is not, `vsprintf` is used instead with a 1000 character buffer.

The argument types and code types are fully checked for type safety. There are some tables of accepted codes: [Table 14.2 C format codes: integer](#), [Table 14.3 C format codes: special integer](#), [Table 14.4 C format codes: floating](#), [Table 14.5 C format codes: other](#).

Please see a suitable reference to learn how to use C format codes.

Table 14.2: C format codes: integer

Code	Type	Radix
hhd	tiny	decimal
hhi	tiny	decimal
hho	utiny	octal
hhx	utiny	hex
hhX	utiny	HEX
hd	short	decimal
hi	short	decimal
hu	ushort	decimal
ho	ushort	octal
hx	ushort	hex
hX	ushort	HEX
d	int	decimal
i	int	decimal
u	uint	decimal
o	uint	octal
x	uint	hex
X	uint	HEX
ld	long	decimal
li	long	decimal
lu	ulong	decimal
lo	ulong	octal
lx	ulong	hex
lX	ulong	HEX
lld	vlong	decimal
lli	vlong	decimal
llu	uvlong	decimal
llo	uvlong	octal
llx	uvlong	hex
llX	uvlong	HEX

Table 14.3: C format codes: special integer

Code	Type	Radix
zd	ssize	decimal
zi	ssize	decimal
zu	size	decimal
zo	size	octal
zx	size	hex
zX	size	HEX
jd	intmax	decimal
ji	intmax	decimal
ju	uintmax	decimal
jo	uintmax	octal
jx	uintmax	hex
jX	uintmax	HEX
td	ptrdiff	decimal
ti	ptrdiff	decimal
tu	uptrdiff	decimal
to	uptrdiff	octal
tx	uptrdiff	hex
tX	uptrdiff	HEX

Table 14.4: C format codes: floating

Code	Type	format
e	double	scientific
E	double	SCIENTIFIC
f	double	fixed
F	double	FIXED
g	double	general
G	double	GENERAL
a	double	hex
A	double	HEX
Le	ldouble	scientific
LE	ldouble	SCIENTIFIC
Lf	ldouble	fixed
LF	ldouble	FIXED
Lg	ldouble	general
LG	ldouble	GENERAL
La	ldouble	hex
LA	ldouble	HEX

Table 14.5: C format codes: other

Code	Type	
c	int (prints char)	
S	string	
s	&char	
p	address	hex
P	address	HEX

Chapter 15

Macro processing

[Library Syntax Reference](#)

[Compiler Semantics Reference](#)

15.1 Macro val

The macro val statement is used to specify an identifier should be replaced by the defining expression wherever it occurs in an expression, type expression, or pattern.

```
macro val WIN32 = true;
macro val hitchhiker;
macro val a,b,c = 1,2,3;
```

15.2 Macro for

This statement allows a list of statements to be repeated with a sequence of replacements.

```
forall name in 1,2,3 do
  println$ name;
done
```

15.3 Constant folding and conditional compilation

[Compiler Semantics Reference](#)

Felix provides two core kinds of constant folding: folding of arithmetic, boolean, and string values, and deletion of code, either statements or expressions, which would become unreachable due to certain value of conditionals.

Basic operations on integer literals, namely addition, subtraction, negation, multiplication, division, and remainder are folded.

Strings are concatenated.

Boolean and, or, exclusive or, and negation, are evaluated.

False branches of if/then/else/endif expression and match expressions are eliminated.

False branches of if/do/elif/else/done are also eliminated.

By this mechanism of constant folding and elimination, Felix provides conditional compilation without the need for special constructions.

Part III

Lookup

Chapter 16

Names

16.1 Kinds of Names

Felix has several kinds of names which can find two kinds of entities.

16.1.1 Simple Names

A *simple name* is just an identifier.

```
x  
joe
```

16.1.2 Simple Indexed Names

An *simple indexed name* is an identifier followed by an open square bracket `[`, a possibly empty comma separated list of type expressions, followed by a close square bracket `]`. If the list is empty, the name is equivalent to a simple name.

```
x[] // equivalent to just x  
joe[int]  
fred[int,string]
```

16.1.3 Unqualified Names

An *unqualified name* is a simple name or simple indexed name. Because indexed names with an empty index list are equivalent to simple unindexed names, all unqualified names are notionally simple indexed names.

16.1.4 Qualified Names

A *qualified name* is sequence of at least two unqualified names separated by two colons ::.

```
A::B::f
A[int]::f[long]
root::B
```

16.1.5 Special name root

The special name `root` is an alias for the top level module name and may be used to force qualified name lookup starting with the top level module.

16.1.6 Unsuffix Name

An *unsuffixed name* is a qualified or unqualified name.

16.1.7 Suffixed Name

A *suffixed name* is an unsuffixed name followed by a suffix consisting of the word `of` followed either by a simple name, or a possibly empty comma separated list of type expressions enclosed in parentheses (and).

```
fun f (x:int) => x;
fun f (x:double, s:string) => x.str + s;
var closure = f of (double * string); // suffixed name
```

Suffixed names are used to disambiguate references to functions from an overloaded set.

16.2 Symbol Tables

Felix represents scopes using symbol tables. There are two kinds of symbol table.

16.2.1 Symbol definition table

Every symbol defined is allocated a unique integer representative and the definition is stored in a hash table keyed by this integer, this is the *symbol definition table*. The definition includes a reference to the parent if any, and, a name lookup table if the defined entity contains a scope.

16.2.2 Name lookup table

For each scope, another kind of hash table called a *name lookup table* is used which maps a string name to a single entry which is one of two kinds: either the entry is a non-function entry or a function entry. Non-function entries contain a single view of a symbol, whilst function entries contain a list of views.

Each scope is associated with two name lookup tables.

Private name lookup table

This table maps all the names defined in the scope to all the definitions in the scope. It is used for unqualified lookup within the scope or any children of the scope.

Public name lookup table

This table contains mappings of the names of all definitions in the scope which are not marked private to those definitions. It is used for qualified lookup into the scope from outside the scope.

16.2.3 Views

A *view* of a symbol consists of the index of the definition in the symbol definition table together with a list of type variables and a list of type expressions using these type variables, this list of type expression must agree in number with the type variables in the entry in the symbol definition table.

The interpretation of a definition referred to by a view is as a polymorphic definition indexed by the view type variables, with each use of the definition's type variables in the body of the definition replaced by the corresponding type in the view's list of type expressions.

For example:

```
class A[T] { fun f:T * T -> T; }
open[U, V] A[U * V];
var x = f ( (1,2.0), (3, 4.0) );
```

This matches because the view of `f` available in the open has type

```
(U * V) * (U * V)
```

via the assignment `T <- U * V`, and the assignments `U <- int` and `V <- long` mean the function call is a specialisation of the view, which in turn is a specialisation of the original function. The view provided by the open may be described by a triple:

```
(index=A::f; vs=U,V; ts=U*V)
```

where `index` is the integer index of `f` in the definition table.

Multiple views of a function, even if overlapping, never create an ambiguity: views are logically merged with non-discriminated setwise union.

This is effected as follows by overload resolution.

16.2.4 Name lookup environments

A name lookup environment consists of a stack of pairs of name lookup tables. The top of the stack represents the current, innermost scope. This is where searching for an unqualified name or first component of a qualified name begins. Each layer represents an including or parent scope, up to the bottom of the stack, which represents the outermost or global scope, which has no parent.

The two name lookup tables at each level are the primary name lookup table and the shadow name lookup table.

Primary name lookup table

The primary name lookup table contains mappings for all the symbols defined in the scope or injected into the scope with a `inherit` or `rename` directives.

Shadow name lookup table

The shadow name lookup table contains mappings for all the symbols made available by an `open` directive or by the `with` clause of the type variable specification of a definition.

`open` directives make the public names defined in, or inherited into, a class available via a view.

The name lookup algorithms always search the primary lookup table first, if that fails they try the shadow table. This ensures any definitions in a scope can be used to hide names provided by an `open`, which ensures it is possible to resolve ambiguities or modify semantics for a particular name, whilst retaining the other names.

16.3 Overload Resolution

Overload resolution proceeds as follows. We start with the result of a function lookup which returns a set of views called candidates. Our aim is to select a unique function, together with assignments to the type variables of the definition, which cause the signature of the call to equal the signature of the function, and any constraints specified for the function to be satisfied.

- During overload resolution, a subset of the candidates views is selected which match the supplied name and signatures.
- Constraints are now applied to eliminate candidates additional candidates. See [section 26.1 Type Constraints](#) for more details.
- The subset is then reduced by eliminating views which are strictly less specialised than another.
- If the result of this reduction consists of a set of more than one view of the same function, then they must map to the same specialisation of the function, and overload resolution has succeeded.

The algorithm is non trivial. Here is another example:

```
class A[T] { fun f (x:T) => x; }
open[U] A[U * int];
open[V] A[int * V];
println$ f (1,1);
```

Here, there are two views of `f`, *both* of which have `T <- int` as a specialisation. The first view is accepted because `U <- int` shows the call signature `int * int` is a specialisation of signature `U * int`, and the second because `V <- int` shows the call signature is a specialisation of the signature `int * V`.

Now, the first view provides the substitution `T <- U * int` and the second `T <- int * V`, and substituting the respective specialisations of the views by the call results in the same specialisation of the original function `T <- int * int`. Therefore this specialisation, being unique, allows the function `f` to be called directly with signature `int * int`.

Not all calls match all views. For example a call on signature `long * int` only matches the first view, and with `int * long` only matches the second: the point is that if more than one view of the same function matches, it is never ambiguous because the views always lead to the same specialisation of the function.

This is not true if the views are views of distinct functions. Something like this can and does happen when the inverse of view lookup is performed when instantiating class virtual functions with distinct instances. In this case, if there are two instances which match, neither of which is most specialised, then Felix will report an ambiguity, even though semantically the behaviour is required to be the same: the semantic requirement is too sloppy to ensure this is meaningful.

Our conclusion is that the programmer is free to open overlapping views of the same function, and in particular duplicated opens do not cause a problem (the same consideration applies to non-functions). *This is not the case when specifying type class instances, where overlaps are associated with distinct definitions, so that a single most specialised definition is required. This issue, however, does not occur in ordinary (phase 1) lookup of names. It occurs only during*

instantiation by monomorphisation when a type class virtual must select from available instances. In this case, there is no overloading to consider since the particular function in each instance is already selected: we only need consider the specialisation lattice.

16.4 Lookup

Names refer either to a function set or single non-function entity. Finding the entity a particular use of a name refers to is called *name lookup* or *name binding*.

Note that unlike C++, Felix has no automatic type conversions. Matching of signatures in Felix must be exact. (However see [subsection 21.2.7 Row Polymorphism](#)).

16.4.1 Unqualified name lookup

Unqualified name lookup is performed using one of two base general algorithms, depending on the kind of name required.

16.4.2 Basic unqualified non-function lookup

When seeking a non-function, Felix first looks for the simple (non-indexed) part of the unqualified name in the current scope.

If the name is found but it is a function symbol the compiler aborts with a diagnostic message.

If the name is not found, it then looks in the special shadow scope of the current context which contains symbols inserted by `open` or `rename` directives.

If an `open` or `rename` directive introduces the same non-function name into the same shadow scope and it refers to the same non-function, with the equivalent indices, one entry is discarded to prevent an ambiguous name error, when in fact both insertions are equivalent and refer to the same entity.

The equivalent indices check works by testing for type equality after alpha conversion. For example

```
class Unique[T] { typedef A = B; }
open[U] Unique[U * U];
open[V] Unique[V * V];
var x: A[int]; // OK
```

If there are two occurrences of the name in the shadow scope, it aborts the compile with a diagnostic message. Note: this error occurs whilst creating the shadow scope, even if the symbol is not used.

If the name is still not found, Felix repeats the process in the parent scope, if any, or aborts the compile with a diagnostic if there is no parent.

```
var x = 1;
class A {
  var x = "hello";
  class C {
    fun f() => x; // A::x = hello
  }
}

begin
  fun f() => x; // root::x = 1
end

open A;
fun f() => x; // root::x = 1

begin
  open A;
  fun f () => x; // A::x = hello
end
```

If the name is indexed, it then checks that the number and kind of indices matches the definition of the symbol discovered, if not it aborts the compile with a diagnostic.

16.4.3 Basic unqualified function lookup

This is the same as non-function lookup, except that the result is expected to be a set of functions with exactly one member.

```
fun f() => 1;
var closure = f; // only one f, OK
```

16.4.4 Lookup with signatures

Felix can find a list of signatures for a function lookup in two contexts.

If the name is a suffixed name, the suffix supplies the sole signature:

```
fun f (x:int) => x;
fun f (x:double) => x;
var closure = f of (double);
```

In an application or call the argument or arguments supply the signatures:

```
fun f (x:int) => x;
fun f (x:double) => x;
var xi = f 1; // f of (double);
var xd = f 1.0; // f of (double);

fun g(x:int) (y:int) => x + y;
fun g(x:int) (y:double) => x.double + y;

var yi = g 1 2; // signatures int,int
var ydi = g 1 2.0; // signatures int,double
// var closure = g 1; // ambiguous, error
```

Care must be taken designing functions to reduce possible ambiguities. Using curried arguments has advantages but one disadvantage is that eta-expansion may be required to select the right overload. A famous example from the Felix library:

```
class List {
  fun map[T,U] (f:T->U) (ls: list[T]) : list[U];
}
class Varray {
  fun map[T,U] (f:T->U) (ls: varray[T]) : varray[U];
}

open List;
open Array;
fun itod (x:int): double => x.double;
// var closure = map itod; // Error, which map?
```

16.4.5 Unqualified function lookup with signatures

When seeking a function with signatures, felix follows similar rules to non-functions, except as follows.

When a function symbol is found, it consists of a set of functions, not a single function. Felix then performs overload resolution to determine if any of the candidates matches.

If there is no match at all, it proceeds with the next step. If there is more than one match, and no match is most specialised, it aborts with a diagnostic message.

Lookup in the shadow scope is performed similarly. Function names introduced from multiple opens are merged prior to the shadow scope lookup. This merger

ensures that if the same name is present in two opens, refers to the same original symbol, and has the same type indices, one is discarded. Note: this merge of equivalent names is not done when constructing the shadow scope, but during overload resolution. Therefore it is not an error to introduce the same name with the same signature for distinct functions, it is only an error if there is an attempt to use such a name. This is different from non-function lookup.

```

fun f(x:int) => x; // signature int
class A {
  fun f(x:double) => x; // signature double
  fun g() {
    return f 1; // signature int: finds root::f of (int)
  }
}
begin
  open A;
  fun f(x:string) => x; // signature string;
  var x = f 1.1; // signature int: finds A::f of (double)
end

```

For the first lookup, there is no `f` in `g`. There is no shadow scope. The `f` in `A` has signature `double` which doesn't match. The `f` in `root` has signature `int` which matches.

For the second lookup we want an `f` with signature `double`. The `f` in the `begin/end` scope does not match. The `f` in the `begin/end` shadow scope introduced by the `open` does match so `A::f of (double)` is found.

Note that this algorithm is not the same as C++. Felix has fixed the design fault.

16.4.6 Qualified non-function name lookup

Qualified name lookup for a non-function proceeds by first performing an unqualified name lookup on the first component of the qualified name, ignoring indices.

If the name is not found the compile aborts and a diagnostic is issued. If the name is not a class name, the compiler aborts and a diagnostic is issued.

Now, the next component is searched for in the specified class scope. If the name is not found, the compile aborts and a diagnostic is issued. The search ignores shadow scopes, and it does not proceed into the context of the classes parent. However it does take inherited names into account. All names except the final one must resolve to class names.

The search a non-ultimate name requires a class to be found in the previously found class. Qualified name lookup, therefore, drills down into a class hierarchy

from the root.

The final name is sought similarly and must be of the kind sought.

Finally the complete sequence of type index parameters is checked for length against the entity's definition.

16.4.7 Qualified function name lookup

Searching for a qualified name of a function is the same as for a non-function, except that a set of overloaded functions may be returned.

If a unique symbol is required, then the set must contain only one function name.

16.4.8 Qualified lookup for function with signatures

A standard qualified name lookup is performed so that the last component returns a function set.

Overload resolution is then performed on the function set. The result is final.

Chapter 17

Libraries

The `library` statement may be used to form extensible groups of named definitions.

```
library A {  
  proc f(x:int) => println$ "A:f " + x.str;  
}  
library B {  
  proc f(x:int) => println$ "B:f " + x.str;  
}  
library A {  
  proc g(x:int) => println$ "A:g ";  
}  
  
open A;  
f 1;  
B::f 1;
```

Unlike classes and modules, libraries are extensible and so may span multiple files (however the library statement may not!).

Libraries cannot be polymorphic, and may not contain virtual functions or procedures directly.

Libraries must be at the top level or nested in other libraries.

Statements within a library statement are executed in order. Statements in multiple library statements for the same library in the same file are executed in order.

Statements in a library statement nested in a library statement are executed before any statements in the enclosing library statement.

Top level code not enclosed in a library statement is treated as if enclosed in an anonymous library.

Otherwise the order of execution is unspecified.

Chapter 18

Abstract Types

An abstract type is a type for which a specific implementation is given, but which can only be accessed by specified methods.

18.0.1 Syntax

An abstract type is created like this, taken from the standard library. First define the type in terms of another type:

```
open class Darray {  
  private struct darray_ctl[T]  
  {  
    a: varray[T];  
    resize: size * size --> size;  
  }  
  type darray[T] = new &darray_ctl[T];  
}
```

Now, we provide a constructor for the abstract type:

```
//£ Make an empty darray, give it 20 slots  
//£ for no particular reason.  
ctor[T] darray[T] () =>  
  _make_darray[T]$ new darray_ctl[T] (  
    varray[T] 20uz , dflt_resize  
  );
```

We use the special name `_make_darray` for the constructor. It constructs a value of the abstraction implementation secretly and casts it to the abstract type.

Now we need some methods defined on the abstract type:

```
// resize
proc do_resize[T] (pd: darray[T], new_size: size)
{
  var old = (_repr_ pd)*.a;
  (_repr_ pd).a <-
    varray[T] (new_size, (len old), (fun(i:size)=>old.i));
}

// pop head
proc pop[t](a:darray[t]) {
  pop (_repr_ a)*.a;
  newsize := (_repr_ a)*.resize
    (maxlen (_repr_ a)*.a, len (_repr_ a)*.a)
  ;
  if newsize != maxlen (_repr_ a)*.a
    call do_resize (a,newsize)
  ;
}
} // end Darray class
```

These methods use the special name `_repr_` to cast the abstract type to its implementation.

So, the statement

```
type darray[T] = new &darray_ctl[T];
```

defines a new, abstract, polymorphic type, `darray[T]` implemented as a pointer to a `struct` containing an bounded array `varray` and a `resize` function.

In order to construct a `darray`, we first create pointer to a `darray_ctl` objects, and then cast it to the abstract type using special name `_make_darray`. The cast from the representation to the abstract type is the name of the abstract type constructor prefixed by `_make_`.

To access the implementation, which is needed to define methods on the abstract type, we use the cast `_repr_`, which converts the abstract type back to its implementation.

Both these casts are private to the containing class `Darray`, so they can only be used in the class defining the abstract type, because private names of a class are only accessible within the class (and its descendants of course).

Therefore, the mechanism provides a way to rigorously protect the implementation of the abstract type. The privacy of the representation type in the example

is not necessary, but is used to prevent namespace pollution.

Chapter 19

Classes

Syntax

19.1 Class Statement

The top level Felix module can contain submodules which are specified by a non-polymorphic class statement:

```
class classname { ... }
```

19.1.1 Qualified Names

The effect is to produce a qualified name to be used outside the class:

```
class classname { proc f () {} }  
classname::f ();
```

19.1.2 Nested Classes

Classes may be nested.

```
class A {  
  class B {  
    proc f () {}  
    f();  
  }  
  B::f();  
}  
A::B::f ();
```

19.1.3 Private definitions

A class may contain private definitions, if a symbol is marked private it is only visible in the scope class in which it is defined, including any nested classes:

```
class A {
  private proc f () {}
  class B {
    f(); // visible inside A
  }
  f(); // visible inside A
}
// A::f () fails, f is not visible outside A
```

19.1.4 Single File Rule

A class must be specified within a single file.

19.1.5 Completeness Rule

Classes are not extensible, a definition of a class with the same name in the same scope is not permitted.

19.1.6 Setwise Lookup

The body of a class forms a nested scope. Within a class all symbols defined in the class are visible, along with all those visible in the enclosing context. This is called setwise lookup.

```
var x = "Hello";
class A {
  proc f() { g(); } // g visible before defined
  proc g() { x = x + 1; }
  var y = f(); // calls g which uses x uninitialised!
  var x = 1;
}
```

Beware using uninitialised variables! The procedure `g` above uses `A::x` before it is initialised.

19.1.7 Top level module alias

The reserved name `root` may be used as a prefix for the top level module:

```
var x = 1;
class A { var x = root::x; }
```

19.2 Polymorphic Classes

19.2.1 Parametric Polymorphism

A class may be specified with one or more type variables. Such a class is said to be a polymorphic class.

```
class MakePair[T,U] {
  fun fwd_pair (x:T) (y:U) => x,y;
  fun rev_pair (x:T) (y:U) => x,y;
}
```

19.2.2 No Variables

Polymorphic classes may not directly contain variables, all members must be function or type definitions.

19.2.3 Indexed Names

Entities from a class can be specified from outside using a qualified name in which the class name is followed a square bracketed list of types instantiating the corresponding type variables, this is an *indexed name*:

```
var f = MakePair[int,string]::fwd_pair 42 "Hello";
var r = MakePair[int,string]::rev_pair 42 "Hello";
```

19.2.4 Deduced indices

However one or more or even all the types can be omitted from the right end of the list if they can be deduced by overload resolution.

19.2.5 Elision of indices

The square brackets are optional if they enclose an empty list.

```
f = MakePair[int,string]::fwd_pair 42 "Hello";
f = MakePair[int]::fwd_pair 42 "Hello";
f = MakePair[]::fwd_pair 42 "Hello";
f = MakePair::fwd_pair 42 "Hello";
```

Note that all symbols in Felix are considered to be indexed. So for example this is correct:

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

19.2.6 Sloppy indexing

It is also permitted to move the instance types across qualification boundaries:

```
f = MakePair[int]::fwd_pair[string] 42 "Hello";
```

although this practice is not recommended.

From the outside, class type variables are viewed as [universal quantification](#). From inside however, the variable names are fixed types which happen to be unknown, that is they're to be considered as [existentials](#).

This means they cannot be specified explicitly or implicitly inside the class with unqualified access:

```
class A[T] {
  fun f(x:T) => x,x;
  fun g(y:T) => f y; // correct
  fun h(y:T) => A[T]::f y; // correct
  // fun h(y:T) => f[T] y; // wrong
}
```

From inside the class the function `f` above is monomorphic, not polymorphic. Although `T` is not known, it is a fixed, invariant type.

19.2.7 Polymorphic Members

Functional members of a class may be polymorphic. This is independent of the class type parameters.

```
class A[T] {
  fun f[U] (x:T, y:U) => x,y;
  fun g(a:T) => f[T * T] (a,a);
```

In this case the function `f` has its own personal type variable `U` which is set in the call in `g` to `T * T` because its argument is a pair of `a` which has that type.

19.3 Virtuals and instances

A polymorphic class may contain functions or procedures marked `virtual`. In this case a definition is optional.

```
class A[T] {
  // non virtual function
  fun f(x:T) => x,x;

  // virtual function with definition
  virtual fun g (x:T) => f x;

  // virtual function without definition
  virtual fun h : T -> T * T;
}
```

If a virtual function without a definition from a class is used, it must be defined for the types for which it is used. This is done in an instance statement:

```
var a = 1; // int
var x = A[int]::h a; // h used for T=int

// instance defining h for int
instance A[int] {
  fun h(a:int) => a + 1, a - 1;
}
```

An instance can be defined anywhere in a non-functional scope provided the class being instantiated is visible.

In this case symbols in the instance definition are bound the context of the instance scope and its enclosing scope in the usual way.

It is not possible to directly access the definitions in an instance from outside the instance. Instead all access is diverted through the top level class being instantiated. Symbols are bound to the top level class member first to fix the type variables, then a matching instance is sought. This second lookup is called *phase 2 lookup*, it occurs during monomorphisation.

If there is no matching instance, the program fails with a diagnostic.

If there is more than than one matching instance, the most specialised instance is used. If there is no most specialised instance, the use of the class member is ambiguous and the program fails with a diagnostic.

An instance must be not less specialised than the class it instantiates (that is, equally specialised or more specialised).

Instance specialisation is judged by the polymorphic subtyping rule which is implemented using a variant of the unification algorithm in which dependent and independent types are distinguished.

A polymorphic type $S[T]$ parametrised by a type variable T is more specialised than a polymorphic type $A[U]$ if U may be replaced by a type expression containing T the result of which equals $S[T]$.

If S has more than one type variable, substitutions for each of them must be found. If A has more than one type variable, the substitutions involve all of these variables.

A specialisation $S1$ of A is strictly more specialised than a specialisation $S2$ of A if $S1$ is also strictly a specialisation of $S2$.

19.3.1 Default Definition

If a virtual function has a definition, it is called a *default definition*. On a use of a function with a default definition, if no matching instance is found, the default definition is specialised and used instead. Note that default definitions do not resolve ambiguity from overlapping instances.

19.3.2 Matching polymorphic functions

If a virtual function is polymorphic, the personal type variables of the function must not be specialised in an instance. The class type variables are resolved independently of the function type variables.

You may think of this as if the class type variables are resolved first by finding a uniquely matching instance, and then the function type variables are resolved in a second step (although in fact there is no coupling between the resolutions).

19.4 Classes as Categories

A class may be thought of in a category theoretic sense.

A base category consists of objects which consists of a finite set of base types and all combinations thereof formed by the available set of type combinators such as tuples, records, function types, etc.

The arrows of this category consist of all the functions explicitly defined on these types, and all compositions thereof. This includes the identity functions implicitly defined by copying.

This is the category generated by the graph consisting of the defined functions, where paths in the graph are (reverse) function compositions. This is called the free category generated by the graph.

A class in the context of a base category generates an extension to that category by throwing in some additional types and arrows. The type variables are added to the set of base types, so that the set of combinations is extended. The functions are added to the set of digraph edges of the base generating graph, to add additional arrows.

This view of the role of a class is vital to understand semantic rules which correspond to the notion of a category generated by a graph with relations. The relations form a constraint. In category theory, the relations are generally equations relating the arrows which specify that certain function compositions are equal to others.

The freely generated category is sometimes called an initial algebra. When constraints are added a new category is formed by collapsing some sequences of arrows to a single arrow. There is a mapping called an epi-morphism which takes each arrow from the freely generated category to the target so that two distinct composites may map to the same arrow in the target.

Conversely, there is a reverse mapping from the arrows of the target category to sets of arrows of the freely generated one, forming another category called a quotient category. These two mappings are functions and together form an adjunction.

The purpose of this explanation is not merely for your entertainment. In most programming classes are intended to be constrained, that is, the instances are intended to have a certain behaviour.

Consider for example:

```
class Addable [T] {  
  virtual fun add: T * T -> T;  
}
```

Now consider the following instances:

```
instance Addable[uint] {  
  fun add (x:uint, y:uint) => x + y;  
}  
  
instance Addable[int] {  
  fun add (x:int, y:int) => x + y;  
}  
  
instance Addable[double] {  
  fun add (x:double, y:double) => x + y;  
}
```


These instances are good according to the specification, however the addition of `int` and `double` is unspecified because we may have overflow. Furthermore one may get a big surprise adding floating point representations of reals: if you add a big float to a small one the small one may be so small that the result is equal to the big one.

Basic laws of arithmetic are not obeyed by floating representations, in particular addition of floats is not associative!

It is usual to specify requirements in comments and the programmer must carefully check that instances obey the constraints specified for the class in these comments.

Felix, however, can do a bit better.

19.5 Axioms, Lemmas and Theorems

Felix allows some semantic constraints on class instances to be specified with explicit axioms. For example:

```
class Addable [T] {
  virtual fun add: T * T -> T;
  axiom associative (x:T, y:T, z:T):
    add (add (x,y), z) == add (x, add (y, z))
  ;
}
```

The axiom documents in a formal language the requirement that the addition operator defined in an instance must be associative. An axiom is similar to an ordinary boolean function which tests whether some condition is met or not. By applying an axiom to particular values we can check for non-conformance of instances.

However, in a categorical sense, an equational axiom plays precisely the role of a relation in the formation of a category with generators and relations. It is a vital part of practical specification of semantics.

We should note that not all constraints can be expressed as axioms in Felix. In particular since Felix only provides first order polymorphism, only equational predicates in Horn form can be used, that is, all the quantifiers must be (implicitly or explicitly) on the left of the formula. Interior quantifiers require second order polymorphism.

Note also that like most programming languages we use constructive mathematics which constrains the role of existentials to calculations. That is, it is not acceptable to assert that something exists because if it did not a contradiction would be derived. Indirect proof is not allowed in constructive mathematics.

Instead, one must prove something exists by actually constructing one! So for example given $a < b$ we know there exists a number x such that $a < x < b$. A proof of this by asserting that if no such x existed, then we would have a contradiction, is not allowed in constructive mathematics. Instead the proof is simply given by $x = (a + b)/2$. And of course .. the assertion is false for many types such as integers!

19.5.1 Lemmas and Theorems

Lemmas and theorems play the same role as axioms. The idea, however, is that lemmas and theorems can be derived from the axioms. A lemma is a simple rule which a good automatic theorem prover could derive, and which seems obvious to a human.

A theorem is a more difficult rule which would require a proof assistant with hints to derive a proof.

19.5.2 Reductions

A reduction is also an axiom, but it has an additional property. Instead of an equation, reductions use a directed equality:

```
reduce revrev[T] (x:list[T]):
    rev (rev x) => x
;
```

This reduction asserts that if you reverse a list twice you end up back where you started. But it does more: it give the compiler a hint that it is allowed to replace an expression which reverse a list twice with the list, removing the two reversing operations.

This is a particularly interesting example of the general rule that all theorems are nothing more than optimisations, in this case literally allowing the compiler to optimise a program. Note that it is only a hint!

19.6 Lookup control directives

19.6.1 Open directive

The simple `open` directive may be used to make the symbols defined in a class visible in the scope containing the `open` directive.

```
class X { var x = 1; }
open X;
println$ x;
```

Names made visible by an open directive live in a weak scope under the current scope. Names in the weak scope may be hidden by definitions in the current scope without error.

```
class X { var x = 1; }
open X;
var x = 2;
println$ x; // prints 2
```

The open directive is not transitive. The names it makes visible are only visible in the scope in which the open directive is written.

19.6.2 Inherit directive

The inherit directive allows all of the public symbols of a class to be included in another scope as if they were defined in that scope. This means such names inherited into a class can be accessed by qualification with the inheriting class name, and will be visible if that class is opened.

Inheriting is transitive.

If a name is inherited it will clash with a local definition.

```
class A { var a = 1; }
class B { inherit A; }
println$ B::a;
```

19.6.3 Rename directive

This directive is can be used to inherit a single symbol into a scope, possibly with a new name, and also to add an alias for a name in the current scope.

When applied to a function name all functions with that name are renamed.

```
class A {
  var a = 1;
  proc f() {}
  proc f(x:int) {}
}

class B {
  rename a = A::a;
  rename fun f = A::f;
}
```

The new name injected by a rename may be polymorphic:

```
class A { proc f[T] () {} }  
class B { rename g[T] = A::f[T]; }
```

19.6.4 Use directive

This is a short form of the rename directive:

```
class A { var a = 1; }  
class B { use A::a; use b = A::a; }
```

It cannot be applied to functions. The first form is equivalent to

```
use a = A::a;
```

Unlike the rename directive the new name cannot be polymorphic and is limited to a simple identifier.

Part IV

Type System

Chapter 20

typedef statement

Syntax

20.1 Type aliases

The plain `typedef` statement is used to define an alias for a type. It does not create a new type.

20.1.1 Basic typedef

```
typedef Int = int;
```

20.1.2 Polymorphic typedef

Typedef can also be polymorphic:

```
typedef Cplx[T] = {x:T,y:T};
```

20.2 Type Sets

A type set is a component of a type constraint expression which restricts a type variable one of a specified list of entities. Each entity may be either a type, a type predicate, or another type set. Syntax like:

```
typesetof (short, int, long, vlong)
```

is used for a typeset expression.

The typedef statement may be used to define an alias for a typeset:

```
typedef Integers = typesetof (short, int, long, vlong);
```

A typeset can be used with a membership test or kinding specification to form a constraint. For example:

```
fun f[T where T in Integers] (x:T) => x;
```

See also [Table 30.3 Integer Typesets](#), [subsection 26.1.2 Typeset membership constraints](#).

20.2.1 basic type functions (functors)

Typedef can also be used to create a type functions, or *functors*. A basic definition is like this:

```
typedef fun pair0 (T:TYPE, U:TYPE) : TYPE => T * U;

var x0 : pair0 (int, long) = 1,2L;
println$ x0;
```

The kind TYPE is the meta-type or kind of all types. The return kind must be specified for functors.

20.2.2 type functions with typematch

A more sophisticated definition allows a typematch:

```
typedef fun pair1: TYPE * TYPE -> TYPE =
| int,_ => int
| _,int => int
| ?T, ?U => T * U
;

var x1a : pair1 (int,double) = 1;
var x1b : pair1 (long, double) = 1L,2.0;
println$ x1a,x1b;
```

See also [section 25.1 typematch expression](#).

20.2.3 Higher order functors

You can also have curried type functions:

```
typedef fun pair2 (T:TYPE) (U:TYPE) : TYPE => T * U;

var x1 : pair2 int long = 1,2L;
println$ x1;

typedef pair3[T] = pair2 int T;
var x3 : pair3 [double] = 1,2.0;

println$ x3;
```

Type functions can be overloaded just like ordinary functions.

20.2.4 Functors in classes

Note carefully there is a difference between an *indexed type* and a *type function*. In the above example, `pair3` is not a functor, its an indexed type. Functors can be passed to type classes, for example:

```
class Monad [M: TYPE->TYPE] {
  virtual fun bind[a,b]: M a * (a -> M b) -> M b;
  virtual fun ret[a]: a -> M a;
}

typedef fun Maybe (t:TYPE):TYPE=>opt[t];

instance Monad [Maybe] {
  fun bind[a,b] (x:Maybe a, f:a -> Maybe b) =>
    match x with
    | #None[a] => None[b]
    | Some x => f x
    endmatch
  ;

  fun ret[a] (x:a):Maybe a => Some x;
}
```

The `M` above accepts a functor, `Maybe` in the instance. We can't use `opt` directly because it is an indexed type, not a functor.

Chapter 21

Products

21.1 Tuples

Tuple types are well known: a tuple is just a Cartesian Product with components identified by position, starting at 0. The n-ary type combinator is infix `*` and the n-ary value constructor is infix `,:`

```
val tup : int * string * double = 1, "Hello", 4.2;
```

21.1.1 Unit Tuple

The 0-ary tuple type is denoted `1` or `unit` with sole value `()`:

```
val u : unit = ();
```

21.1.2 No 1-ary Tuple

The 1-ary tuple of type `T` component value `v` is identified with the type `T` and has value `v`.

21.1.3 Projections

The individual components of a tuple may be accessed by a projection function. Felix uses an integer literal to denote this function.

```
var x = 1, "Hello";  
assert 0 x == 1; assert x.0 == 1;  
assert 1 x == "Hello"; assert x.1 == "Hello";
```

A pointer to a tuple is also in itself a tuple, namely the tuple of pointers to the individual components. This means if a tuple is addressable, so are the components.

```
var x = 1, "Hello";
val px = &x;
val pi = px.0; pi <-42;
val ps = px.1; ps <-"World";
assert x.0 == 42;
assert x.1 == "World";
```

In particular note:

```
var x = 1, "Hello";
&x.0 <- 42;
```

because the precedences make the grouping `(&x).0`.

You cannot take the address of a tuple component because a projection of a value is a value.

Assignment to components of tuples stored in variables is supported but only to one level, for general access you must take a pointer and use the store-at-address operator `<-`.

21.1.4 Standalone Tuple projections

The projections of a tuple can also be written in an expanded form so that they may stand alone as functions:

```
var first = proj 0 of (int * string);
var a = 1, "Hello";
var one = a . first;
var two = a . proj 1 of (int * string);
```

21.1.5 Alternate Tuple Representation

Felix provides a second representation of tuples. Given a tuple of at least two components, another value may be prefixed to the head of the tuple with the `,,` operator:

```
var tail = 1,2L; // binary tuple
var triple = "head",,tail; // ternary tuple
assert triple == "head",1,2L;
```

Similarly, given a tuple type of at least two components, another type may be prefixed to the head of the type with the ****** combinator:

```
typedef pair = int * long;
typedef triple = string ** pair;
// same type as string * int * long
```

21.1.6 Pattern matching tuples

The real value of the alternate notation lies in pattern matching. For example:

```
match "head",1,2L with
| head,,tail => println$ tail;
endmatch;
```

The real power comes with classes:

```
instance [T,U with Eq[T], Eq[U]] Eq[T ** U] {
  fun == : (T ** U) * (T ** U) -> bool =
    | (ah ,, at) , (bh ,, bt) => ah == bh and at == bt;
  ;
}

instance [t,u with Eq[t],Eq[u]] Eq[t*u] {
  fun == : (t * u) * (t * u) -> bool =
    | (x1,y1),(x2,y2) => x1==x2 and y1 == y2
  ;
}

instance [t with Eq[t]] Eq[t*t] {
  fun == : (t * t) * (t * t) -> bool =
    | (x1,y1),(x2,y2) => x1==x2 and y1 == y2
  ;
}
```

This mechanism allows recursive processing of heterogenous lists, that is, tuples. In the example we compute equality. The first instance provides heterogenous list recursion, the second case terminates the recursion. The third case is also a terminal which allows the recursion to work for arrays as well. Without this the library array recursion would be more specialised.

21.2 Records

A record is similar to a tuple except the components are named and considered unordered up to duplication.

21.2.1 Plain Record

A plain record is one without duplicate fields. A plain record type is one without duplicate fields or a row variable. A record is constructed using a parenthesis enclosed list of comma separated field assignments. An empty record is equivalent to an empty tuple.

```
typedef xy = (x:int, y:int);
var r : xy = (x=1,y=2);
```

21.2.2 Record projections

A component of a record may be accessed with a function called a record value projection, it is denoted by the name of the field.

```
var r (x=1,y=2);
println$ x r, r.y;
```

Record value projections can also be used as stand-alone functions. For example:

```
var rl = list ((x=1,y=11),(x=2,y=22));
var xs = map (x of (x:int, y:int)) rl;
println$ xs; // list (1,2)
```

Records also have pointer projections, overloaded with the value projections: if the name of a field is applied to a pointer to a record, a pointer to the named component field is returned. This allows assignment and other mutators to be applied to record components.

```
var r =(x=1,y=2);
var px = &r.x; // means (&r).x
px <- 42;
r&.y <- 23;
println$ r.x, r.y; // (42,23)
```

Record pointer projects can also be used as stand-alone functions:

```
var prjx = x of (xy);
var prjpx = x of (&xy);
```

21.2.3 General record

Records may have duplicate fields. In this case, reading from left to right in a record literal, a duplicate field is hidden by a previous field of the same name, in a push down stack like fashion.

```
var r = (x=1,y=2,x="Hello");
println$ r._strr, r.x;
// ((x=1,x='Hello',y=2), 1)
```

Note that the generic function `_strr` displays the whole of the record including duplicate fields. However projections only find the left-most field.

21.2.4 Adding fields

Fields can be added to an existing record to construct a new record:

```
var r = (x=1,y=2);
var r2 = (a="one",b="two",x="newx" | r);
println$ r2._strr,r2,x;
// ((a='one',b='two',x='newx',x=1,y=2), newx)
```

Again, leftmost fields hide rightmost ones.

21.2.5 Removing Fields

The `without` operator can be used to return a record with some fields removed. It works on values of record and polyrecord type. Note that because Felix uses scoped fields once a field is removed, the previous value of that field is exposed if it exists, and can also be removed. This means it is correct and sometimes necessary to list a field more than once when using the `without` operator.

```
var x = (a=1,b=2,c=3,c=4);
var y = (x without b c); // (a=1,c=4)
```

21.2.6 Adding records

You can also add two records with infix `+`:

```
var r = (a=1) + (b=2) + (a="hello");
println$ r._strr, r.a.str; // ((a=1,a='hello',b=2), 1)
```

The leftmost field with a given name dominates. Record addition by `+` is only applied if a user defined addition is not found for the argument types.

Currently, addition of fields of two records of the same type is not supported: it is likely the user intended to add corresponding field values rather than hide the fields in the right argument with those on the left.

21.2.7 Row Polymorphism

Felix provides a special record type called a *polyrecord* which supports row polymorphism with scoped labels in the style of Daan Leijen. The article is [here](#).

This allows a generic function to be written which accepts an argument which is or contains a value of a record type with more fields than required. Unlike subtyping, the extra fields, whilst inaccessible, are not lost and can be returned by the function. For example:

```
val circle = (x=0.0,y=0.0,r=1.0);
val square = (x=0.0,y=0.0,w=1.0,h=1.0);

fun move[T]
  (dx:double, dy:double)
  (shape: (x:double, y:double | T))
=>
  (x=shape.x+dx, y=shape.y+dy | (shape without x y))
;

var inc = 1.0,1.0;
println$ (move inc circle)._strr, (move inc square)._strr;
```

21.2.8 Interfaces

An interface is a special notation for a record type all of whose fields are functions or procedures.

```
interface fred {
  f: int -> int;
  g: int -> 0; // procedure
}

// equivalent to
typedef fred = (f: int -> int, g: int -> 0);
```

The primary use is for specifying the type of a Java like objects. See [section 10.8 Dynamic Objects](#)

21.3 Structs

A struct is a a nominally typed record, that is, it must be defined, and each definition specifies a distinct type.

```
struct S { x:int; y:int; };
var s1 : S = S (1,2);
var s2 : S = S (y=42,x=32);
println$ s1.x,s1.y,s2.x,s2.y;
```

21.3.1 Alternate Definition Syntax

As a special case, a `struct` can be defined with the field type before the field name:

```
struct S { int x; int y; };
```

Furthermore there is a special syntax:

```
typedef struct {
  int x;
  int y;
} S;
```

for defining structs. The purpose of this syntax is to allow a C struct to be copied directly into Felix code with minimal editing. Note the above form does *not* define an alias, it defines a struct called `S`.

21.3.2 Struct constructor

A struct value can be constructed using the structure name as a function and passing a tuple of values corresponding by position to the fields of the struct, as shown above. A record with matching fields can also be used. Of course in both cases the types must also agree.

21.3.3 Constructor as function

A struct constructor can be used a first class function. In this case the argument type is a tuple, not a record.

```

struct S { x:int; y:int; };

proc p (factory: int * int -> S) {
  var a = factory (1,2);
  println$ a.x,a.y;
}
p S;

```

21.3.4 Projections

The field names are projection functions and can be applied to a struct value to extract the nominated component.

They can also be applied to a pointer to a struct to find a pointer to the nominated component.

```

struct S {x:int, y:int; };
var a = S (1,2);
var px = &a.x;
px <- 11;
&a.y <- 22;

```

Projections can be used as first class functions with the notation:

```

var prjx = x of (S);
var prjpx = x of (&S);

```

can be used to refer to a projection in isolation, and a pointer projection in isolation, that is, as unapplied first class functions.

21.3.5 Nested methods

A struct may also contain function and procedure definitions:

```

struct A {
  x:int;
  y:int;
  fun get2x => 2 * self.x;
  fun get2y () => 2 * self.y;
  proc diag (d:int) { self.x <- d; self.y <-d; }
};

```

These functions are precisely equivalent to:


```
fun get2x (self:A) => 2 * self.x;  
proc diag (self: &A) (d:int) { self.x <-d; self.y <-d; }
```

Note that for a function `self` is a value, for a procedure `self` is a pointer.

21.3.6 Method closures

Because of these definitions, we can form object closures over a struct:

```
var a = A(1,2);  
var g2y = a . get2y;  
var di = a . diag;
```

Note we can't form a closure for `get2x` without an explicit wrapper, i.e. eta-expansion.

Chapter 22

Sums

22.1 Anonymous Sum Types

Sum types are the dual of tuples. They represent a sequence of possible cases, potentially with arguments. For example the sum type:

22.1.1 Sum type

```
typedef num = int + long + double;  
// dual to int * long * double
```

represents three cases and is dual to the tuple type shown.

22.1.2 Sum values

The value of a sum represents one of the cases of the type. Case indices are 0 origin as with tuples.

```
typedef num = int + long + double;  
var c0 = (case 0 of num) 42;  
var c1 = (case 1 of num) 53L;  
var c2 = (case 2 of num) 66.0;
```

The parenthesised expressions above are called *type constructors*.

22.1.3 caseno operator

The case number of a sum value can be extracted as an `int`:

```
println$ caseno c0, caseno c1, caseno c2;
// 0,1,2
```

22.1.4 Switching on cases

Sum variables are decoded with a match which may also extract an argument value:

```
typedef num = int + long + double;
var x = (case 1 of num) 53L;
println$
  match x with
  | case 0 (i) => "int " + i.str
  | case 1 (l) => "long" + l.str
  | case 2 (d) => "double " + d.str
  endmatch
;
```

The match tags do not include the type of the sum, since it is known from the argument.

22.1.5 Unit arguments

A sum component may have a unit argument.

```
typedef maybe_int = 1 + int;
var x = case 0 of maybe_int;
match x with
| case 0 () => println$ "Nothing";
| case 1 i => println$ "Integer " + i.str;
endmatch;
```

For the first case the () may be omitted:

```
typedef maybe_int = 1 + int;
var x = case 0 of maybe_int;
match x with
| case 0 => println$ "Nothing";
| case 1 i => println$ "Integer " + i.str;
endmatch;
```

or the unit application syntax can be used:

```

typedef maybe_int = 1 + int;
var x = case 0 of maybe_int;
match x with
| #case 0 => println$ "Nothing";
| case 1 i => println$ "Integer " + i.str;
endmatch;

```

If the argument is a unit, the constructor is sometimes called a *constant constructor*.

22.1.6 Unit sum

There is a family of special sum types equivalent to:

```

2 = 1 + 1
3 = 1 + 1 + 1
4 = 1 + 1 + 1

```

Recall type 1, or `unit`, is the type of the empty tuple.

The type 2 is also known as `bool`, and represents two cases where `false` is an alias for `case 0 of 2` and `true` is an alias for `case 1 of 2`.

The type 0 or `void`, is the type of no values.

These types are called *unit sums* because they're a sum of a certain number of units.

Note carefully that:

$$x + (y + z), (x + y) + z, x + y + z$$

are three distinct types because operator `+` is not associative.

22.2 Unions

A union is the dual of a struct. It is a nominally typed version of a sum. Here for example is a list of integers:

```

union intlist {
  iEmpty ;
  iCons of int * intlist;
};

```

This alternative syntax is more commonly used and comes from ML family:

```
union intlist =
  | iEmpty
  | iCons of int * intlist
;
```

The fields of a union type are injections or type constructors. In effect they cast their argument to the type of the union, thus unifying heterogeneous types into a single type.

Pattern matches are used to decode unions.

```
var x = iEmpty;
x = iCons (1, x);
x = iCons (2, x); // list of two integers

fun istr (x:intlist) =>
  match x with
  | #iEmpty => "end"
  | iCons (i, tail) => i.str + "," + istr tail
  endmatch
;
```

The first variant represents an empty list. The second variant says that a pair consisting of an int and a list can be considered as a list by applying the type constructor iCons to it.

Unlike product types, a sum may directly contain itself. This is because sum types are represented by pointers.

22.2.1 enum

A restricted kind of union, being a nominally typed version of a unit sum.

```
enum colour { red, green, blue }; // same as
enum colour = red, green, blue; // same as
union colour = red | green | blue;
```

The tag value of an enum can be set:

```
enum wsize = w8=8, w16=16, w32=32, w64=64;
```

22.2.2 caseno operator

The caseno operator can find the tag value of any sum type, the anonymous sum, union, enum or variant as an integer.

```
assert caseno w16 == 16;  
assert caseno (case 1 of 2) == 1;
```

22.3 Variants

Variants are the sum type which are the dual of records. They used named injections like unions but are structurally typed.

22.3.1 Variant coercion

Variant values always have a singleton type, so to be used with a set of variants, a coercion must be used.

```
typedef vars = union { Int of int ; Float of float; };  
var x = Int 32; // type union {Int of int}  
var y = x :>> vars; // coerce to type vars
```

The type being coerced must be super type of the coercion type, this means it must have subset of the cases of the coercion type.

Chapter 23

Exponentials

23.1 Functions

Felix provides four core function types.

23.1.1 Function

Functions use infix operator `->` to denote a function, where the LHS is called the domain, and the RHS the codomain:

```
var f : int -> long = fun (x:int)=>x.long;
```

The operator is right associative so that

```
var f : int -> long -> string =  
  fun (x:int) =>  
    fun (y:long)=>  
      (x.long + y).str  
;  
// equivalent to  
var f : int -> (long -> string) =  
  fun (x:int) (y:long) =>  
    (x.long + y).str  
;
```

If a function returns a function, it is a higher order functions. The number of `->` symbols in the type is known as the *arity* of the function.

Function defined with `fun` binder may not have side effects.

Functions are slaves which accept data and control, perform a calculation, and return a result and control to the caller, which is the master. The master/slave relationship may also be expressed as a client/server relationship.

Felix uses the machine stack to save and retrieve the master return address for a function.

At run time a function closure is represented by a single machine address which is nominally a pointer to a heap allocated C++ object which retains the function local variables and provides an `apply()` method.

Functions which depend only on their arguments and any constants in their context are said to be *pure*, those which depend on external variables are *impure*. The kind can be noted with an adjectival descriptor on the definition:

```
pure fun sin: double -> double = "sin($1)";
var x = 1;
impure fun get_x() => x;
```

Impure functions are sometimes called observers or accessors.

23.2 Generators

A generator is a function which is permitted to have side effects. It is defined with the `gen` binder instead of `fun`.

Generators have the same type as functions.

The exemplar generator is of course `rand()` which returns a different value each call, depending on some external state.

23.3 Yielding Generators

Generators may contain a `yield` statement which returns a value and control, but also records the current location in the local state, to allow the generator to be resumed where it left off.

To use a yielding generator a closure must be stored in a variable.

Yielding generators are a special case of coroutines, implemented in a special way.

```
gen integers () = {
  var i = 0;
next:>
  yield i;
  ++i;
```



```

    goto next;
}

var counter = integers;
var zero = counter();
var one = counter();

```

23.3.1 Procedure

A procedure is given the type of a function with the codomain set to 0 (aka void). Procedures are not functions, they return control but not a value. Technically a procedure is a delimited continuation.

Although procedures look like functions, in Felix they are utterly different because they do not use the machine stack to store the return address. Instead, procedures use a heap allocated data frame, with pointers to ancestral context and the caller continuation return address, and save the current location in a special variable in the data frame.

Thus procedures may cooperatively swap themselves out in favour of another procedure.

A procedure with its ancestral context, call chain, and current location is called a fibre. It represents an exchangeable locus of control.

23.3.2 C function

Felix can deal with C functions too. The infix operator `-->` is used to denote a C function type. C functions can be defined directly in Felix like:

```
cfun f(x:int)=>x + x;
```

Felix will represent this function like:

```
int f(int x) { return x + x; }
```

which has a C function type. The Felix C function type is a C function closure, in C a pointer to the C function.

If a `cfun` uses the thread frame or garbage collector, it will fail to compile at C++ compile time because Felix always passes a pointer to the thread frame explicitly.

We provide `cfun` primarily to allow construction of C data types used by drivers as callback functions.

However they can also be used to define an interface to a C callable library written in Felix.

Note that Felix knows that C functions accept multiple arguments rather than tuples and automatically unpacks a tuple passed to a C function.

23.3.3 C procedure

Felix can deal with C procedures too, the type is the same as a C function with the codomain set to void.

```
var y = 1;
cproc f(x:int) {y = x}
```

Felix will represent this like:

```
void f(int x) { y = x; }
```

which has a C function type. The Felix C function type is a C function closure, in C a pointer to the C function.

If a `cfun` uses the thread frame or garbage collector, it will fail to compile at C++ compile time because Felix always passes a pointer to the thread frame explicitly.

23.3.4 Type indexed functions

Felix has an extension which allows function types to be indexed by types. For example:

```
struct IO {};
fun f(x:int) : [IO] int = {
  println$ x; return x;
}
var clos : int -> [IO] int = f;
```

The indexing type is known as the *effects* type of the function. For functions without effects annotations, the effects type defaults to `unit`.

By combining effects types, row polymorphism, and typematches, Felix provides tools to manage various effects. For example:

```
fun map[T,U] (f:T->[G] U) (list[T]) : [G] list[U] => ..
```

says that the higher order function `map`, given a function `f` with effect `G`, will itself have effect `G`.

This is an experimental feature that is not yet utilised by the library.

23.4 Array

Felix has various kinds of array. The term is abused and sometimes refers to the abstract concept, and sometimes the statically typed fixed length array described here.

An array is nothing but a tuple all of whose elements have the same type. It is convenient to use an exponential operator with a unit sum index to provide a compact notation:

```
int ^ 3 // array of 3 integers equivalent to
int * int * int
```

Therefore the value:

```
var a3 = 1,2,3;
```

is, in fact, an array. As for tuples an integer literal applied to an array value returns a component, however for arrays, an expression may be used as well:

```
var i = 1;
var y = a3 . i;
var z = a3 . proj i of (int^3);
```

The last form is equivalent to

```
var z = a .
  match i with
  | 0 => proj 0 of (int^3)
  | 1 => proj 1 of (int^3)
  | 2 => proj 2 of (int^3)
  | _ => throw error
endmatch
;
```

in other words there is a run time array bounds check equivalent to a match failure. Note that of course the actual generated code is optimised!

A run time check can be avoided by using the correct type of index:

```
var i = case 1 of 3;
var z = a . i; // no run time check
```

23.4.1 Multi-arrays

Whilst we introduced the exponential notation

$$B \wedge J$$

as a mere shorthand, where J is a unit sum, in fact Felix allows the index to be any compact linear type.

A compact linear type is any combination of sums, products, and exponentials of unit sums. The type

$$3 * 4 * 5$$

for example is compact linear, and therefore Felix allows the array type

```
typedef matrix = double ^ (3 * 4 * 5)
```

Although this looks like the type

```
typedef array3 = double ^ 3 ^ 4 ^ 5
```

as suggested by the usual index laws, the latter is an array size 5 of arrays size 4 of arrays size 3 which can be used like:

```
var a : array3;
var z = a . case 1 of 5 . case 1 of 4 . case 1 of 3;
```

where you will note that the projections are applied in the reverse order to the indices. On the other hand to use the first form we have instead:

```
var m : matrix;
var z = a . (case 1 of 3, case 1 of 4, case 1 of 5);
```

The exponent here is a value of a compact linear type. It is a single tuple value! It is called a multi-index when applied to an array.

The advantage of this type is that there is an obvious encoding of the values shown in this psuedo code:

```
i * 3 * 4 + j * 3 + k
```

which is nothing more than a positional number notation where the base varies with position. That encoding clearly associates with the compact linear value as integer in the range 0 to 59, or, alternatively, an value of type 60. In other words the type is linear and compact.

Since clearly, given an integer in range 0 through 59 and this type we can decode the integer into a tuple, being the positional representation of the integer in this weird coding scheme, the type is clearly isomorphic to the subrange of integer.

Therefore Felix allows you to coerce an integer to a compact linear type with a run time check, and convert a compact linear type to an integer or a unitsum:

```
var i : int = ((case 1 of 3, case 1 of 4, case 1 of 5) :>> int);
var j : 60 = ((case 1 of 3, case 1 of 4, case 1 of 5) :>> 60);
var clt : 3 * 4 * 5 = (16 :>> 3 * 4 * 5);
```

Because we can do this we can now write a loop over a matrix with a single iterator:

```
for i in 60 do
  println$ m . (i :>> 3 * 4 * 5);
done
```

This is an advanced topic which will require an extensive explanation beyond the scope of this summary. However we will not that this facility provides a very high level feature known as polyadic array programming. In short this means that one may write routines which work on matrices of arbitrary dimension. You can of course do this in C by doing your own index calculations at run time and using casts, however Felix does these calculations for you based on the type so they're always correct.

Chapter 24

Type Recursion

Felix provides two mechanisms to allow type recursion: implicit recursion by self reference, and explicit recursion using pointers.

24.1 The `as` combinator

We can express the type of a list without a definition using the `as` combinator:

```
typedef intlist = 1 + int * X as X;
```

The `as` combinator has low precedence and introduces a binding variable similar to a μ binder in type theory, however the variable is visible outside the scope it controls as well. The inner use of the variable is known as a *fixpoint* and is a place where whole term controlled by the binder may be substituted, resulting, if repeated, in an infinite recursion. The use of the binder therefore allows infinite data types to be represented finitely.

In the list case above, a list is either a unit, or a pair consisting of an `int` and another list.

Type recursion in Felix may be written anywhere but will not work correctly unless a box intervenes. A box is either any sum type or a pointer.

24.2 Pointers

Pointers provide another mechanism for implementing type recursion explicitly. Pointers are more powerful than fixpoints, which is not always an advantage.

The notation for a pointer to an object of type `T` is either

```
&T
ptr[T] // alias for &T
```

Values of pointer type can originate in two primary ways: from the address of a variable:

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

or by constructing an object on the heap. The `new` operator copies a value to a heap object and returns its address as a pointer:

```
var px : &int = new 1;
```

The basic Felix pointer type cannot be NULL, cannot be incremented, and always points to a single object. It is therefore always safe to dereference to fetch the value stored at the pointer with the prefix dereference operator `*`:

```
var px : &int = new 1;
var x = *px;
```

Objects located by pointers are intrinsically mutable. A new value can be stored with the store at operator:

```
var px : &int = new 1;
px <- 2;
```

Pointers to products have special properties. More precisely, for each value projection p which extracts a the value of a product component, there is a corresponding pointer projection which accepts a pointer to a product object and returns a pointer to the component;

```
var pair = 1, "Hello";
var s = pair . 1; // value projection
var ps = &pair . 1; // pointer projection
ps <- "Bye"; // pair is now 1, "Bye"
```

This machinery eliminates the need for the concept of addressable lvalues. Thus you cannot take the address of a product component:

```
var pair = 1, "Hello";
// var ps = &(pair.1); // ERROR
```

because `pair.1` is a value, and values do not have public addresses. Instead Felix provides some sugar:

```
var pair = 1, "Hello";
var ps = pair&.1; // means (&pair).1
var &p = &pair;
var s = p*.1; // means (*p).1
```

24.3 Maybe NULL pointers

Felix also has a pointer type that could perhaps be NULL. It is defined in the library by:

```
union cptr[T] = | nullptr | Ptr of &T;
```

and the alias

```
@T
```

may also be used. Due to implementation magic, this type is compatible with C/C++ pointers, that is, Felix `nullptr` is C `NULL`. In particular, one should write, for example:

```
gen malloc: size -> @T = "malloc($1)";
```

because `malloc` could return `NULL`. Strict adherence to correctly typing C functions is necessary to ensure the guarantee dereferencing Felix pointers is safe.

Note that dereferencing C pointers, even if not `NULL`, need not be safe because the pointed at object may not be live. Felix guarantees to keep heap allocated objects to which Felix pointers exist live.

Note that a `cptr[T]` obtained from a `ptr[T]` has this guarantee as well.

24.4 Incrementable Pointers

Felix also has C style incrementable pointers given the type `carray[T]` or alias `+T`. Pointers of this type are not intrinsically safe to use. They can be `NULL`.

The `carray` is provided as a model of C arrays so that they can be manipulated in Felix, even if such operations are risky.

Chapter 25

Meta-typing

Felix provides some facilities for meta-typing.

25.0.1 typedef fun

The notation

```
typedef fun diag (T:TYPE):TYPE=> T * T;  
var x: diag int = 1,2;
```

defines a type function (or functor). Given a type T, this function returns the type for a pair of T's. The identifier TYPE denotes the kind which is a category of all types.

Applications of type functions must be resolved during binding, since the result may influence overloading.

25.1 typematch expression

Felix has a facility to inspect and decode types at compile time.

```
typedef T = int * long;  
var x:  
  typematch T with  
  | A * B => A  
  | _ => int  
endmatch  
= 1  
;
```

As with type functions, type matches must be resolved during binding. If a type match fails, an error is issued a compilation halted. The wildcard type pattern `_` matches any type.

The real power of the type match comes when combined with a type function:

```
typedef fun promote (T:TYPE): TYPE =>
  typematch T with
  | #tiny => int
  | #short => int
  | #int => int
  | #long => long
  | #vlong => vlong
  endmatch
;
```

This functor does integral promotions of signed integer types corresponding to ISO C rules.

25.2 Type Sets

A type set is a component of a type constraint expression which restricts a type variable one of a specified list of entities. Each entity may be either a type, a type predicate, or another type set. Syntax like:

```
typesetof (short, int, long, vlong)
```

is used for a typeset expression.

The typedef statement may be used to define an alias for a typeset:

```
typedef Integers = typesetof (short, int, long, vlong);
```

25.2.1 Union and Intersection

Two operators are provided for typesets, union and intersection which are spelled `\cup` and `\cap` respectively and are printed as set theoretic union and intersection symbols \cup and \cap .

```
typedef sints = typesetof (int,long);
typedef uints = typesetof (uint,ulong);
typedef ints = sints  $\cup$  uints;
typedef shorts = typesetof (short,int,long);
typedef ssints = ints  $\cap$  shorts;
```

25.2.2 Membership

The membership operator tests if a type is a member of a typeset, it is spelled `in` or `\in`, the latter being typeset as the set theoretic symbol \in . It serves to define the semantics of a typeset as a `typematch`.

```
typedef ints = typesetof(short,int,long);
typedef result = int ∈ ints;
```

The membership test above is precisely equivalent to

```
typedef result = typematch int with
| short => 1
| int => 1
| long => 1
| _ => 0
endmatch;
```

25.2.3 Typeset membership constraints

Typesets were introduced to Felix to simplify bindings to C functions. For example:

```
typedef someints = typesetof (int, long);
typedef someuints = typesetof (uint, ulong);
fun someadd[T in someints] : T * T -> T = "$1+$2";
fun someadd[T in someuints] : T * T -> T = "$1+$2";
println$ someadd (1,2); // OK, first function
println$ someadd (1u,2u); // OK, second function
// println$ someadd (1,2u); // fails
```

In this example, a call to `someadd` will only work if the arguments types are the same and both either `int` or `long`. Overload resolution first treats the functions as if there were no constraints. Then, any candidate which fails the constraint is thrown out of the set.

This method saves writing out all the cases individually, which may be exponential in the number of individual types involved, whilst at the same time preventing invalid cases which an unconstrained binding would allow.

If two candidates exist with the same signatures, and one has a constraint and the other does not (or has the constant constraint `true`), the constrained candidate is considered more specialised than the unconstrained one.

However in general, constraints cannot be compared to see which is most specialised. In particular if for all values of type variables constraint A implies

constraint B, A is more specialised, however there is no operational method for measuring this in general (halting problem!).

Chapter 26

Polymorphism

26.1 Type Constraints

Felix provides 4 kinds of type constraints.

26.1.1 Type class constraints

Class constraints have no impact on phase 1 overload resolution when selecting a binding.

However, when binding the definition of a symbol, specified class views are introduced in a shadow scope as if the class were opened.

During monomorphisation, any virtuals must be mapped to instance functions and this is done by a process similar to that used for overload resolution.

26.1.2 Typeset membership constraints

Typesets were introduced to Felix to simplify bindings to C functions. For example:

```
typedef someints = typesetof (int, long);
typedef someuints = typesetof (uint, ulong);
fun someadd[T in someints] : T * T -> T = "$1+$2";
fun someadd[T in someuints] : T * T -> T = "$1+$2";
println$ someadd (1,2); // OK, first function
println$ someadd (1u,2u); // OK, second function
// println$ someadd (1,2u); // fails
```

In this example, a call to `someadd` will only work if the arguments types are the same and both either `int` or `long`. Overload resolution first treats the functions

as if there were no constraints. Then, any candidate which fails the constraint is thrown out of the set.

This method saves writing out all the cases individually, which may be exponential in the number of individual types involved, whilst at the same time preventing invalid cases which an unconstrained binding would allow.

If two candidates exist with the same signatures, and one has a constraint and the other does not (or has the constant constraint true), the constrained candidate is considered more specialised than the unconstrained one.

However in general, constraints cannot be compared to see which is most specialised. In particular if for all values of type variables constraint A implies constraint B, A is more specialised, however there is no operational method for measuring this in general (halting problem!).

26.1.3 Equational Constraints

Felix allows a constraint to be For example:

```
fun f[T,U where T * int == int * U] (x:T * U) => x;
println$ f (1,2);
```

This example uses an equational constraint.

Here is another example using a typematch:

```
typedef fun ispair (x:TYPE) : TYPE =>
  typematch x with
  | _ * _ => 1
  | _ => 0
  endmatch
;

fun f[T where ispair T] (x:T)=>x;

println$ f (42,33L); // OK, signature ispair (int * long) true
```

Chapter 27

Generics

27.1 Generic Functions

Generics provide a way to do *cut and paste* polymorphism, similar to what C++ does. It is useful for one-off scripts, or times when parametric polymorphism and type classes are not enough.

27.1.1 Generic Syntax

The syntax is to just put the kind of a type variable as `GENERIC`:

```
fun accumulate [T:GENERIC] (container:T) = {  
  var sum = 0;  
  for elt in container do  
    sum = sum + elt;  
  done  
  return sum;  
}
```

27.1.2 Using a generic

With the above example you can write:

```
println$ accumulate (1,2,3,4);  
println$ accumulate (1,2,3,4).list;  
println$ accumulate (1,2,3,4).varray;
```


27.1.3 Operational Semantics

Felix handles generics as follows. The generic function is treated first, as if it were polymorphic, that is, the `GENERIC` kind replaced by the default `TYPE`. After overload resolution is complete, if a generic function is selected, Felix clones the function and all its descendants, creating a new function. In these clones, however, the generic type variable is removed and in the definition of the function and its descendants it is replaced by the actual type deduced by the overload resolution algorithm. The system then replaces the overload result with a reference to this new function.

Therefore, the call is dispatched to a non-generic function, specialised for the arguments.

This kind of operation is also done by the type class specialiser during monomorphisation, and inlining routines during optimisation, however with generics the specialisation is done *before* binding.

Unlike C++, the specialisation is done in the context of the definition. This is more reliable, but it means functions in the context of use cannot participate in overload resolution.

```
class X {
  proc f(x:int) => println$ "X::f " + x;
  proc g[T:GENERIC] (x:T) => f x;
  g 25;
}

class Y {
  proc f(x:int) => println$ "Y::f " + x;
  X::g 33;
}

X::g 99;
```

The output is:

```
X::f 25
X::f 33
X::f 99
```

27.2 The typecase expression

The `typecase` expression allows you to write expressions dependent on a type. Typecase expressions must be resolved before binding is complete.

```

fun f[T:GENERIC] (x:T) =>
  typecase T with
  | int => x + 1
  | double => x + 1.0
  endmatch
;

println$ f 1, f 4.2;

```

Although primarily of use in generic functions, typecases can also be useful for conditional compilation. For example:

```

if PLAT_WIN32
  type file = "HANDLE";
else
  type file = "int";
endif

fun read (f:file) =>
  typecase typeof f with
  | int => unix::read f
  | HANDLE => win32::read f
  endmatch
;

```

27.3 Builtin Generics

Felix has a number of builtin generic functions.

27.3.1 The `_strr` function

The `_strr` function converts any value into a string, with a syntax roughly approximating the syntax used in the program.

It was primarily intended to simplify output of records, the final `r` from to record.

```

println$ (a=1,b="hello", c=(1,2,3), a=(1,2,3).list);
struct A {
  a:int;
  b:int;
}
println$ (A(1,2))._strr;

```

The output is:

```
(a=1,a=Snoc (Snoc (Snoc (Empty,3),2),1),b='hello',c=(1, 2, 3))
A {a=1,b=2}
```

The `Snoc(tail,head)` terms there are the native list constructors used by Felix, which are reversed `Cons(head,tail)` terms. Felix uses these because they generate a pointer followed by data in C++, which means list scanning is independent of the data type.

Operational Semantics

The converter works by calling the `str` function for primitives such as `int`, `string`, or `float`, so these must be in scope. The type class based `str` is injected into the global shadow scope so this will always be bound, however if there is no instance for a type an error will be generated during monomorphisation.

For structured types, `_strr` exhibits the structure in some nice way, wrapping the child members of the structures, which are converted to strings recursively.

There is a limit on the recursion depth, and Felix will print `...` when it hits that depth.

```
println$ (1,2,3,4,5,6,7,8,9).list._strr;
```

produces

```
Snoc (Snoc (Snoc (Snoc (... ,4),3),2),1)
```

27.3.2 The `_eq` function

Felix provides a builtin generic function `_eq` which can be used to compare small acyclic values of arbitrary types:

```
var x = (1,(a=2,b=3,c=(1,2).list));
var y = x;
println$ _eq (x,y);
```

Operational Semantics

The comparator requires arguments of the same type. It operates by a recursive descent into the structure of the types. If the type of a subexpression is primitive, the function `==` is used. This function is defined in the typeclass `Eq`, and may also be defined in the context of the comparison.

Do not use this generic on a cyclic value, it will loop forever at run time.

The current implementation only expands tuples and records. It delegates to `==` for all other types. Therefore, lists will be handled correctly since `==` is already defined for lists in the library.

Technically `_eq` is polyadic, not generic. However the implementation is generic.

27.3.3 The `_map` function

The `_map` operator maps an overloaded function set over a tuple. For example:

```
fun f(x:int)=> x + 1;
fun f(x:double)=> (x+42.0).str+ "!!";
fun f[T] (x:T)=>x;
println$ _map f (1,2.2,"Hello");
```

produces:

```
(2, 44.2!!, Hello)
```

Although many generic operations can better be defined using type classes, it is not possible to do this kind of operation other than with a generic.

All folds, for example, can be defined with type classes because folding mandates the return type be invariant, and we can use polymorphism on the argument type. However this is not possible with mapping because the return type is not dependent on the argument type in a predictable manner.

Technically, Felix can indeed express the return type of the individual functions. Consider:

```
typedef fun fret(argtype:TYPE):TYPE =>
  typematch argtype with
  | int => int
  | double => string
  | ?other => other
endmatch
;

class U[T] {
  virtual fun ff: T -> fret T;
}

instance U[int] {
  fun ff(x:int)=> x + 1;
```

```

}

instance U[double] {
  fun ff(x:double)=> (x+42.0).str+ "!!";
}

instance[T] U[T] {
  fun ff[T] (x:T)=>x;
}

open[T] U[T];

println$ ff 22;
println$ ff 3.0;
println$ ff "Hello";

```

The first two prints actually work, but the last one fails:

```

CLIENT ERROR
[Cannotmatch] Cannot instantiate virtual ff<271188>[string]
In /Users/skaller/felix/b.flx: line 57, cols 1 to 20
56: println$ ff 3.0;
57: println$ ff "Hello";
    *****
58:

```

The reason is, that the `ff[T]` in instance `U[T]` does not have the correct type to specialise the virtual. The virtual has type

```

T ->
  typematch argtype with
  | int => int
  | double => string
  | ?other => other
endmatch

```

whereas the instance has type

```

T -> T

```

however matching `T` against `int` or because whether the match succeeds depends on the not yet performed replacement of `T` with a specific type.

Note that in theory if the `double` branch were removed, the `typematch` would always return `T`, since if it is `int` it returns `int`, and otherwise returns `that`. However the compiler is not able to reduce the `typematch` like that because it looks only at the top branch to begin, and continues only on definite failure.

Part V

Interfacing

Chapter 28

C bindings

Felix is specifically designed to provide almost seamless integration with C and C++.

In particular, Felix and C++ can share types and functions, typically without executable glue.

However Felix has a stronger and stricter type system than C++ and a much better syntax, so binding specifications which lift C++ entities into Felix typically require some static glue.

28.1 Type bindings

In general, Felix requires all primitive types to be first class, that is, they must be default initialisable, copy constructible, assignable, and destructible. Assignment to a default initialised variable must have the same semantics as copy construction.

It is recommended C++ objects provide move constructors as Felix generated code uses pass by value extensively.

The Felix type system does not support C++ references in general, you should use pointers instead.

However, there is a special lvalue annotation for C++ functions returning lvalues that allows them to appear on the LHS of an assignment. Only primitives can be marked lvalue.

The Felix type system does not support either const or volatile. This has no impact when passing arguments to C++ functions. However it may be necessary to cast a pointer returned from a primitive function in order for the generated code to type check.

28.2 Basic type statement

The type statement is used to lift a C++ type into Felix.

The simplest example:

```
type metres = "double";
type Z = "long";
```

creates a Felix type `metres` based on C++ type `double`.

Polymorphic types can be created too:

```
type pair[T,U] = "::std::pair<?1,?2>";
```

The names of the type variables T and U are not used in the binding, but serve as placeholders. Instead the encoding `?1` and `?2` is used to refer to the first and second type variables. The Felix type `pair[metres,Z]` will become C++ type `::std::pair<double,long>` in the generated code.

Please be sure to always fully qualify C++ names including leading `::` since you do not know the namespace context in which the generated code will reside.

28.2.1 Semantics

Unless otherwise noted all types lifted into Felix with the type binding statement must be first class. This means that they must have a publically accessible default initialiser, copy constructor, copy assignment operator, and destructor.

To lift a type which is immobile, lift a pointer to the type instead. Please see below.

28.3 Adjectives

Felix needs to know some things about types to handle them properly, it uses a system of adjectives to provide some basic information.

28.3.1 pod types

A pod type is a *plain old data type* of the kind common in C. In Felix the annotation is used for optimisation: calls to the destructor are not made by the Felix garbage collector when an object of this type is finalised. This will have no impact provided the type has a trivial destuctor in C++.

Do not specify this adjective unless you are sure the destructor does no useful work.

```
pod type metres = "double";
```

28.3.2 `_gc_pointer` types

The `_gc_pointer` adjective tells Felix that the type is a pointer which could be a pointer to an heap allocated object managed by the GC.

When a value of this type is included in a larger objects, the run time type description will include the offset of this subobject in the array of offsets which might contain managed pointers.

It is always safe to use this annotation if the C++ types is a pointer.

28.3.3 `incomplete` types

The adjective `incomplete` may be used to specify the type denoted may not be constructed. If the C++ type being bound is incomplete, for example `void`, or an incomplete C struct, this annotation will cause a type error in Felix if an attempt is made to construct values of the type.

The primary reason for introducing incomplete types is to permit a Felix pointer to be formed to it, in Felix. For example

```
incomplete type MyVoid = "void";
typedef MyVoidPointer = &MyVoid;
```

28.3.4 `uncopyable` types

This annotation may be used if a value of the type can be constructed but not copied. Since Felix always initialised variables with assignments, uncopyable type are useful only for heap construction yielding a pointer.

28.3.5 `_gc_type T` types

This adjective is for gurus only! In conjunction with `_gc_pointer` it is used to permit contruction of an immobile C++ type whilst at the same time ensuring that the type garbage collected.

The method is fraught with peril and use of C++ smart pointers is recommended instead.

```
private uncopyable type bQ_ =
    "::flx::pthread::bound_queue_t"
;
_gc_pointer _gc_type bQ_ type ts_bound_queue_t[T] =
    "::flx::pthread::bound_queue_t*"
```

```

requires
package "flx_bound_queue",
scanner "::flx::pthread::bound_queue_scanner"
;
ctor[[T] ts_bound_queue_t[T]: !ints =
    """
        new (*PTF gcp, @0, false) ::flx::pthread::bound_queue_t(
            PTF gcp->collector->get_thread_control(), (size_t)$1)
        """ requires property "needs_ptf";

```

In this example from the library, we define the thread safe bound queue as type `bq_` and mark it private and uncopyable to protect it. The C++ type is defined in the Felix run time library.

The type the user gets is a pointer, which is defined by a binding to C++ again, declared to be a pointer, and declared to point to `bq_`.

What this actually does is seen in the constructor which, in C++, constructs the object on the heap, passing the magic encoding `@0` to the second argument of the new expression.

This encoding *normally* denotes the pointer to the shape (RTTI) for the return type of the function. However that type, `ts_bound_queue` has the attribute `_gc_type bq_` which tells Felix to use the shape object for that type instead. We need to do this because the heap value is represented by a pointer to the type, but we want the garbage collector to be able to map such a pointer to the type of the heap object at run time.

28.3.6 finaliser clause

A primitive type may have a **finaliser** clause with string argument which must specify a C++ function with type

```
typedef void finaliser_t (collector_t*, void*);
```

which is used to destroy objects of the type. If not specified the C++ destructor is used, unless the type is a **pod** in which case no finaliser is deployed. The finaliser is called by the collector after an object is determined to be unreachable, before the underlying storage is reclaimed.

28.3.7 scanner clause

A primitive type may have a **scanner** clause with string argument which must specify the name of a C++ function with type `::flx::gc::generic::scanner_t` given by the typedef

```
typedef void *scanner_t(
    collector_t*, gc_shape_t *, void *, size_t, int
);
```

This is a custom function responsible for finding all the pointers in a data type.

28.3.8 encoder clause

A primitive type may have an **encoder** clause with string argument which must specify the name of a C++ function with type `::flx::gc::generic::encoder_t` given by the typedef

```
typedef ::std::string encoder_t (void *);
```

This is a custom function responsible for encoding the type in a string for serialisation.

28.3.9 decoder clause

A primitive type may have an **decoder** clause with string argument which must specify the name of a C++ function with type `::flx::gc::generic::decoder_t` given by the typedef

```
typedef ::std::size_t decoder_t(void *, char *, ::std::size_t);
```

This is a custom function responsible for decoding a string encoded by the corresponding encoder and building an object of the specified type on the heap.

28.4 Expression bindings

Felix has a simple method for introducing constants and expressions from C++ using the `const` statement:

```
type metres = "double";
const zero_metres: metres = "0.0";
const one_metre: metres = "1.0";
```

Const statements cannot be polymorphic.

28.5 Function bindings

In order to use types lifted from C++, Felix has a special form of function definition:

```
type metres = "double";
const zero_metres: metres = "0.0";
const one_metre: metres = "1.0";
fun +: metres * metres -> metres = "$1+$2";
fun *: metres * double -> metres = "$1*$2";
fun *: double * metres -> metres = "$1*$2";
fun todouble: metres -> double = "$1";
var len = one_metre * 3.7;
var len2 = len + len;
println$ "length " + len2.todouble.str;
```

Here the encodings \$1 and \$2 refer to the first and second components of the argument tuple type.

Felix instantiates such functions by substituting the C++ encoding of the argument component into the text string of the function at the position nominated location, possibly adding parentheses to ensure precedence rules do not interfere with the intended semantics.

There is a special set of strings for which Felix knows the precedence. You must use exactly these strings to trigger the rules. It is also possible to manually specify the precedence. If Felix does not understand the precedence then parentheses will be inserted. The purpose of the precedence controls is entirely to reduce the need for the generated C++ to be littered with parentheses.

[Put table of special strings here] [Put list of precedence names here]

It is sometimes necessary to refer to other entities than the argument components.

[Put table of binding encodings here]

28.6 Conversion bindings

A special form is allowed for conversion function:

```
type metres = "double";
ctor metres: double = "$1";
ctor double: metres = "$1";
```

This is exactly the same as:

```
fun _ctor_metres: double -> metres = "$1";
fun _ctor_double: metres -> double = "$1";
```

so that the syntax makes it easier to see a conversion is being defined and what is being converted.

Conversion statements can be polymorphic, but the type variables must be introduced like this:

```
type vect[T] = "::std::vector<?1>"
  requires header '#include <vector>';
fun len[T]: vect[T]->size = "$1.size()";
ctor[T] vect[T]: size = "::std::vector<?1>($1)";
var x = vect[int] 42uz;
println$ x.len;
```

When Felix is seeking a function, if it finds a type name instead, it retries the lookup with `_ctor_` prefixed to the type name. This only works with actual names! However the name may be the defined name of a type or a typedef alias.

Despite the spelling, a `ctor` is a conversion operator and not a type constructor.

28.7 Procedure bindings

Procedure bindings work the same as function bindings, by emitting the C++ string specified with substitutions.

The primary difference is that procedure bindings must generate one or more C++ statements.

28.8 cstruct bindings

A `cstruct` definition has the same syntax as a `struct` and provides the same facilities, except that the compiler does not actually emit a C++ structure definition.

Instead the statement is used to lift an existing C type into Felix. It can be used for structs or unions.

The compiler provides projection and pointer projections and a memberwise constructor.

Be warned against using the memberwise constructor if the definition is not complete, for example for certain standard C and Posix types which are incompletely specified.

[It is not currently possible to turn this off] [It currently is only possible to bind to names in the global namespace]

28.9 cenum bindings

Blah.

28.10 cflags bindings

Blah.

28.11 Floating insertions

Felix has a sophisticated system of *floating insertions* which are primarily designed to ensure that if a C++ binding is used, the appropriate C++ header files are included so the binding code is well defined. For example:

```
type vect[T] = "::std::vector<?1>"
  requires header '#include <vector>';
```

Floating insertions can be directly introduced as indicated for inclusion in either the C++ **header** file or **body** file Felix generates.

Floating insertions *float* to near the top of the relevant file. Insertions specified in a single file appear in order of writing, if they appear at all, and at most once. The actual insertion strings are checked to prevent duplication.

Insertions can be named:

```
header vector_h = '#include <vector>';
type vect[T] = "::std::vector<?1>"
  requires vector_h
;
```

so that a `requires` clause may specify the name of an insertion. Duplication of names is permitted:

```
header vector_h = '#include <vector>';
header vector_h = '#include <stdlib.h>';
type vect[T] = "::std::vector<?1>"
  requires vector_h
;
```

A floating insertion is included if, and only if, a C binding is actually used.

Floating insertions themselves may have dependencies expressed with requires clause:

```
header algo_h = '#include <algorithm>';
header vector_h = '#include <vector>'
    requires algo_h
;
```

Insertions can also be polymorphic.

```
body sho[T] = 'show<?1>()';
```

A module or non-polymorphic class may contain a naked requires statement:

```
class Vect {
    requires algo_h;
    type vect[T] = 'vector<?1>';
```

The effect is to apply the requirement to all C++ bindings in the class or module. Note: the specified insertion is only emitted if a C++ binding is used!

28.12 Property requirements

A requires clause may also specify properties:

```
fun f: int -> int = "f($1)"
    requires property "needs_ptf"
;
```

This provides a flexible tagging system in addition to the use of adjectives.

[List of properties here]

28.13 Package requirements

A requires clause may specify a package requirement:

```
fun f: int -> int = "f($1)"
    requires package "mylib"
;
```


Packages are a collection of platform dependent resources which are configured so that the abstract requirement is used to locate the resource descriptors in a database managed by the `flx_pkgconfig` system, mapping the abstract platform independent name to platform dependent names.

Packages can be used to add header files to compiler C++ output, and to find and link support libraries. In some cases, they can also help build the package.

The package system requires the user to supply a search path consisting of a list of directories which are searched to find the file with the package name and extension `.fpc`.

A typical file looks like this:

```
Name: SDL2
Description: Simple Direct Media Layer 2.0
cflags: -I/usr/local/include/SDL2
includes: "SDL.h"
provides_dlib: -L/usr/local/lib -lSDL2
provides_slib: -L/usr/local/lib -lSDL2
```

This description allows Felix to compile and link programs using the SDL2 library without any special switches on the `flx` command line.

The standard location for an installed Felix configuration is in the subdirectory `host/config` on Unix or `host\config` on Windows.

Chapter 29

Embedding

29.1 C Callable libraries

29.2 Export directives

The `export` directives make the exported symbol a root of the symbol graph.

The functional export and forces it to be place in the generated code as an `extern "C"` symbol with the given name:

```
export fun f of (int) as "myf";
export cfun f of (int) as "myf";
export proc f of (int) as "myf";
export cproc f of (int) as "myf";
```

Functions are exported by generating a wrapper around the Felix function. If the function is exported as `fun` or `proc` the C function generated requires a pointer to the thread frame as the first argument, if the `cfun` or `cproc` forms are used, the wrapper will not require the thread frame.

In the latter case, the Felix function must not require the thread frame.

A type may also be exported:

```
export type ( mystruct ) as "MyStruct";
```

This causes a C typedef to be emitted making the name `MyStruct` an alias to the Felix type. This is useful because Felix types can have unpredictable mangled names.

The word **export** optionally followed by a string may also be used as a prefix for any Felix function, generator, or procedure definition. If the string is omitted is taken as the symbol name. The effect is the same as if an export statement has been written.

Part VI

Library

Chapter 30

Core Primitive Types

30.1 Boolean type

The type `bool` which is also called 2 provides the usual boolean logic values `false` and `true`. The name `bool` is actually an alias for type `unit` sum of two cases. The name `false` is an alternate name of the value `case 0 of 2` and the name `true` is an alternate name for the value `case 1 of 2`.

See `?? ??` for more information.

30.2 Integer types

[Library Reference](#)

There is a table of the types [Table 30.1 Felix Integer Types](#).

Note that all these types are distinct unlike C and C++. The types designated are not the complete set of available integer like types since not all have literal representations.

Signed integers are expected to be two's complement with one more negative value than positive value. Bitwise and, or, exclusive or, and complement operations do not apply with signed types.

The effect of overflow on signed types is unspecified.

Unsigned types use the standard representation. Bitwise operations may be applied to unsigned types. Basic arithmetic operations on unsigned types are all well defined as the result of the operation mathematically modulo the maximum value of the type plus one.

The maximum value of an unsigned type is one less than two raised to the power of the number of bits in the type. The number of bits is 8, 16, 32, or 64 or 128

Table 30.1: Felix Integer Types

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

for all unsigned types.

There is a table of the operators [Table 30.2 Integer Operators](#).

30.2.1 Classification of integers

Library Reference

Integer types can be grouped into sets. In Felix this can be done with a `typeset` construction. The defined typesets are show in [Table 30.3 Integer Typesets](#).

Fast integers correspond to the usual C distinct integer types.

Exact integers are aliases to fast integers of specified sizes, in Felix these are distinct types.

The weird integers are special purposes aliases to fast integers which have special uses, again in Felix these are distinct types.

- The type `ptrdiff` is a signed type intended to represent the difference between two pointers.
- The `size` type is for measuring the sizes of objects and number of elements in an array.
- The `intmax` and `uintmax` types are the largest available signed and unsigned integer types, respectively.
- Finally the `intptr` and `uintptr` types are the same size as a pointer and can be used for low level numerical operations on pointers either by a conversion or reinterpret cast. They're typically used for to pack extra data into the low bits of a pointer to a type with an alignment which ensures the low bits of the pointer are zero.

Typesets can be used to simplify specification of C function bindings by leveraging constrained polymorphism.

```
fun add[T in ints]: T * T -> T = "$1+$2";

// or equivalently
fun add[T : ints]: T * T -> T = "$1+$2";
```

A special form which implies a quantifier for every use also exists, by specifying a `!` followed by a typeset name. For example,

```
fun intsum: !ints * !ints -> long = "(long)($1+$2)";

// or equivalently
fun intsum[T:ints, U:ints] : T * U -> long = "(long)($1+$2)";
```

Table 30.2: Integer Operators

symbol	kind	type	semantics
All Integers			
<code>==</code>	infix-nassoc	$T * T \rightarrow \text{bool}$	equality
<code>!=</code>	infix-nassoc	$T * T \rightarrow \text{bool}$	inequality
<code><</code>	infix-nassoc	$T * T \rightarrow \text{bool}$	less
<code><=</code>	infix-nassoc	$T * T \rightarrow \text{bool}$	less or equal
<code>></code>	infix-nassoc	$T * T \rightarrow \text{bool}$	greater
<code>>=</code>	infix-nassoc	$T * T \rightarrow \text{bool}$	greater or equal
<code>+</code>	infix-lassoc	$T * T \rightarrow T$	addition
<code>-</code>	infix-lassoc	$T * T \rightarrow T$	subtraction
<code>*</code>	infix-lassoc	$T * T \rightarrow T$	multiplication
<code>/</code>	infix-lassoc	$T * T \rightarrow T$	quotient
<code>%</code>	infix-lassoc	$T * T \rightarrow T$	remainder
<code><<</code>	infix-lassoc	$T * T \rightarrow T$	multiplication by power of 2
<code>>></code>	infix-lassoc	$T * T \rightarrow T$	division by power of 2
<code>-</code>	prefix	$T \rightarrow T$	negation
<code>+</code>	prefix	$T \rightarrow T$	no op
<code>succ</code>	func	$T \rightarrow T$	successor
<code>pred</code>	func	$T \rightarrow T$	predecessor
Signed Integers			
<code>sgn</code>	func	$T \rightarrow T$	sign
<code>abs</code>	func	$T \rightarrow T$	absolute value
Unsigned Integers			
<code>\&</code>	infix-lassoc	$T * T \rightarrow T$	bitwise and
<code>\ </code>	infix-lassoc	$T * T \rightarrow T$	bitwise or
<code>\^</code>	infix-lassoc	$T * T \rightarrow T$	bitwise exclusive or
<code>~</code>	prefix	$T * T \rightarrow T$	bitwise complement
nassoc: non-associative			
lassoc: left associative			
func: function			
note prefix - maps to function <code>neg</code>			

Table 30.3: Integer Typesets

<code>fast_sints</code>	(tiny, short, int, long, vlong)
<code>exact_sints</code>	(int8,int16,int32,int64)
<code>fast_uints</code>	(utiny, ushort, uint, ulong,ulong)
<code>exact_uints</code>	(uint8,uint16,uint32,uint64)
<code>weird_sints</code>	(ptrdiff, ssize, intmax, intptr)
<code>weird_uints</code>	(size, uintmax, uintptr)
<code>sints</code>	<code>fast_sints</code> \cup <code>exact_sints</code> \cup <code>weird_sints</code>
<code>uints</code>	<code>fast_uints</code> \cup <code>exact_uints</code> \cup <code>weird_uints</code>
<code>fast_ints</code>	<code>fast_sints</code> \cup <code>fast_uints</code>
<code>exact_ints</code>	<code>exact_sints</code> \cup <code>exact_uints</code>
<code>ints</code>	<code>sints</code> \cup <code>uints</code>

Note `intsum` can add any different kinds of integers, returning a `long`.

Polymorphism with typeset constraints is *only* useful for C bindings because the constraints are not propagated. It simply saves writing out all the finite combinations when genericity in the C or C++ target code means the C code would be the same in each case.

30.3 Floating point types

Library Reference

There is a table of the operators [Table 30.5 Floating Point Operators](#), we also have [Table 30.4 Trig functions](#).

Felix also provides `FINFINITY`, `DINFINITY` and `LINFINITY` for positive infinity representation for `float`, `double` and `ldouble` types, respectively.

30.4 Complex types

There are three complex types, `fcomplex`, `dcomplex` and `lcomplex` corresponding to cartesian representation using a pair of `float`, `double` and `ldouble` values, respectively.

There is a table of the operators [Table 30.6 Complex Operators](#), we also have [Table 30.4 Trig functions](#).

30.5 Quaternion type

There is a table of the operators [Table 30.7 Quaternion Operators](#).

Table 30.4: Trig functions

symbol	type	semantics
circular		
<code>sin</code>	<code>T -> T</code>	sine
<code>cos</code>	<code>T -> T</code>	cosine
<code>tan</code>	<code>T -> T</code>	tangent
<code>asin</code>	<code>T -> T</code>	arc (inverse) sine
<code>acos</code>	<code>T -> T</code>	arc (inverse) cosine
<code>atan</code>	<code>T -> T</code>	arc (inverse) tangent
hyperbolic		
<code>sinh</code>	<code>T -> T</code>	hyperbolic sine
<code>cosh</code>	<code>T -> T</code>	hyperbolic cosine
<code>tanh</code>	<code>T -> T</code>	hyperbolic tangent
<code>asinh</code>	<code>T -> T</code>	arc (inverse) hyperbolic sine
<code>acosh</code>	<code>T -> T</code>	arc (inverse) hyperbolic cosine
<code>atanh</code>	<code>T -> T</code>	arc (inverse) hyperbolic tangent
Note: Inverses return primary branch		

Table 30.5: Floating Point Operators

symbol	kind	type	semantics
<code>==</code>	infix-nassoc	<code>T * T -> bool</code>	equality
<code>!=</code>	infix-nassoc	<code>T * T -> bool</code>	inequality
<code><</code>	infix-nassoc	<code>T * T -> bool</code>	less
<code><=</code>	infix-nassoc	<code>T * T -> bool</code>	less or equal
<code>></code>	infix-nassoc	<code>T * T -> bool</code>	greater
<code>>=</code>	infix-nassoc	<code>T * T -> bool</code>	greater or equal
<code>+</code>	infix-nassoc	<code>T * T -> T</code>	addition
<code>-</code>	infix-nassoc	<code>T * T -> T</code>	subtraction
<code>*</code>	infix-nassoc	<code>T * T -> T</code>	multiplication
<code>/</code>	infix-nassoc	<code>T * T -> T</code>	quotient
<code>-</code>	prefix	<code>T -> T</code>	negation
<code>abs</code>	func	<code>T->T</code>	absolute value
<code>log10</code>	func	<code>T->T</code>	base 10 logarithm
<code>sqrt</code>	func	<code>T->T</code>	square root
<code>ceil</code>	func	<code>T->T</code>	ceiling
<code>floor</code>	func	<code>T->T</code>	floor
<code>trunc</code>	func	<code>T->T</code>	truncate
<code>embed</code>	func	<code>int->T</code>	embedding
nassoc: non-associative			
lassoc: left associative			
func: function			
note prefix <code>-</code> maps to function <code>neg</code>			

Table 30.6: Complex Operators

symbol	kind	type	semantics
<code>fcomplex</code>	ctor	<code>float * float -> fcomplex</code>	cartesian
<code>fcomplex</code>	ctor	<code>float -> fcomplex</code>	from real
<code>dcomplex</code>	ctor	<code>double * double -> dcomplex</code>	cartesian
<code>dcomplex</code>	ctor	<code>double -> dcomplex</code>	from real
<code>lcomplex</code>	ctor	<code>ldouble * ldouble -> lcomplex</code>	cartesian
<code>lcomplex</code>	ctor	<code>ldouble -> lcomplex</code>	from real
<code>==</code>	infix-nassoc	<code>T * T -> bool</code>	equality
<code>!=</code>	infix-nassoc	<code>T * T -> bool</code>	inequality
<code>+</code>	infix-lassoc	<code>T * T -> T</code>	addition
<code>-</code>	infix-lassoc	<code>T * T -> T</code>	subtraction
<code>*</code>	infix-lassoc	<code>T * T -> T</code>	multiplication
<code>/</code>	infix-lassoc	<code>T * T -> T</code>	quotient
<code>-</code>	prefix	<code>T -> T</code>	negation
<code>real</code>	func	<code>T->R</code>	real part
<code>imag</code>	func	<code>T->R</code>	imaginary part
<code>abs</code>	func	<code>T->R</code>	norm
<code>arg</code>	func	<code>T->R</code>	argument

nassoc: non-associative
 lassoc: left associative
 func: function
 ctor: constructor
 note prefix `-` maps to function `neg`

Table 30.7: Quaternion Operators

symbol	kind	type	semantics
<code>==</code>	infix-nassoc	<code>T * T -> bool</code>	equality
<code>!=</code>	infix-nassoc	<code>T * T -> bool</code>	inequality
<code>+</code>	infix-lassoc	<code>T * T -> T</code>	addition
<code>-</code>	infix-lassoc	<code>T * T -> T</code>	subtraction
<code>*</code>	infix-lassoc	<code>T * T -> T</code>	multiplication
<code>/</code>	infix-lassoc	<code>T * T -> T</code>	quotient
<code>-</code>	prefix	<code>T -> T</code>	negation

nassoc: non-associative
 lassoc: left associative
 func: function
 note prefix `-` maps to function `neg`

Table 30.8: ASCII charsets

symbol	definition
upper	ABCDEFGHIJKLMNOPQRSTUVWXYZ
lower	abcdefghijklmnopqrstuvwxyz
letters	upper + lower
digits	0123456789
alphanum	letters + digits
cidcont	alphanum+"_"
flxidcont	alphanum+"_'"
camlidcont	alphanum+"_'"
numeric	digits + ".eEdD_"

30.6 Char Type

Felix has a type `char` corresponding to C's `char`. There are no literals of type `char`, instead there is a function, also called `char` to construct a `char` from a string literal. If the string at least length 1, the first character is returned, if the string is empty the NUL char `\0x00` is returned.

An overload of the `char` function also accepts an `int` argument and constructs a `char` with the designated C code value which must be in the range 0 through 255.

The function `ord` extracts the C code value of a `char` as an `int` in the range 0 through 255.

We note that C code values were originally used for the ASCII character set. Strings of `char` may be used to represent ISO-10646 (Unicode) code points via the UTF-8 encoding method. ASCII characters in the range 0 through 127 agree with Unicode interpretation.

30.6.1 ASCII classification functions

There is a set of [Table 30.9 ASCII classification functions](#) used to test the ASCII class of `char`.

The library also provides character sets represents as strings, show in [Table 30.8 ASCII charsets](#)

Table 30.9: ASCII classification functions

symbol	type	semantics
ISO C functions		
<code>isupper</code>	<code>char -> bool</code>	A-Z
<code>islower</code>	<code>char -> bool</code>	a-z
<code>isalnum</code>	<code>char -> bool</code>	A-Za-z0-9
<code>isalpha</code>	<code>char -> bool</code>	A-Za-z
<code>isdigit</code>	<code>char -> bool</code>	0-9
<code>isxdigit</code>	<code>char -> bool</code>	0-9A-Za-z
<code>iscntrl</code>	<code>char -> bool</code>	0x0-0x1F
<code>isspace</code>	<code>char -> bool</code>	0x20
<code>isblank</code>	<code>char -> bool</code>	0x20,0x09
<code>isprint</code>	<code>char -> bool</code>	0x20-0x7e
<code>ispunct</code>	<code>char -> bool</code>	punctuation
Lexing functions		
<code>isidstart</code>	<code>char -> bool</code>	First char identifier
<code>iscamlidcont</code>	<code>char -> bool</code>	Ocaml, subsequent chars
<code>iscidcont</code>	<code>char -> bool</code>	C, subsequent chars
<code>isflxidcont</code>	<code>char -> bool</code>	Felix, subsequent chars
<code>isnumeric</code>	<code>char -> bool</code>	0-9+-.Ee
<code>isalphanum</code>	<code>char -> bool</code>	A-Za-z0-9
<code>isletter</code>	<code>char -> bool</code>	A-Za-z
<code>issq</code>	<code>char -> bool</code>	'
<code>isdq</code>	<code>char -> bool</code>	"
<code>isslosh</code>	<code>char -> bool</code>	\
<code>isnull</code>	<code>char -> bool</code>	0x0
<code>iseol</code>	<code>char -> bool</code>	\n LF: Unix, CR: Windows

Chapter 31

Algebraic Structure of numeric types

Basic Algebra Package Number Package

The functionality identified in the previous chapter is not provided on a type by type basis, but instead leverages the mathematic structure of the types. This is modelled using Felix type classes.

In this section we display a subset of the definitions from the library. The real code also defines some other classes and also a rich set of alternate names for operators utilising the presence of LaTeX symbols: this allows for pleasant typesetting but usually doesn't provide essential semantic operations.

31.1 Equality

We start with class `Eq` which models equality based on the notion it is an equivalent relation:

```
class Eq[t] {  
  virtual fun == : t * t -> bool;  
  virtual fun != (x:t,y:t):bool => not (x == y);  
  
  axiom reflex(x:t): x == x;  
  axiom sym(x:t, y:t): (x == y) == (y == x);  
  axiom trans(x:t, y:t, z:t): x == y and y == z implies x == z;  
}
```

By providing an instance with `==`, inequality `!=` is automatically provided by the default definition. An instance can also provide the inequality, as this is

sometimes more efficient.

Conformance to the axioms in instances is required, these axioms define the required semantics.

31.2 Total Ordering

Now we can define a total order:

```
class Tord[t]{
  inherit Eq[t];
  virtual fun < : t * t -> bool;
  fun >(x:t,y:t):bool => y < x;
  fun <= (x:t,y:t):bool => not (y < x);
  fun >= (x:t,y:t):bool => not (x < y);

  fun max(x:t,y:t):t=> if x < y then y else x endif;
  fun min(x:t,y:t):t => if x < y then x else y endif;

  axiom trans(x:t, y:t, z:t): x < y and y < z implies x < z;
  axiom antisym(x:t, y:t): x < y or y < x or x == y;
  axiom reflex(x:t, y:t): x < y and y <= x implies x == y;
  axiom totality(x:t, y:t): x <= y or y <= x;
}
```

To have a total order, we first have to have equality, this is effected by use of the `inherit` statement.

Next we define the less than function as virtual, instances must define this. The other functions are provided automatically.

The semantics are specified by the axioms which define a total ordering: reflexivity, anti-symmetry, transitivity, and finally totality.

31.3 Addition

Now we define addition in two steps. First we define approximate addition

```
class FloatAddgrp[t] {
  inherit Eq[t];
  virtual fun zero: unit -> t;
  virtual fun + : t * t -> t;
  virtual fun neg : t -> t;
  virtual fun prefix_plus : t -> t = "$1";
  virtual fun - (x:t,y:t):t => x + -y;
```

```

virtual proc += (px:&t,y:t) { px <- *px + y; }
virtual proc -= (px:&t,y:t) { px <- *px - y; }

reduce id (x:t): x+zero() => x;
reduce id (x:t): zero()+x => x;
reduce inv(x:t): x - x => zero();
reduce inv(x:t): - (-x) => x;
axiom sym (x:t,y:t): x+y == y+x;

fun add(x:t,y:t)=> x + y;
fun plus(x:t)=> +x;
fun sub(x:t,y:t)=> x - y;
proc pluseq(px:&t, y:t) { += (px,y); }
proc minuseq(px:&t, y:t) { -= (px,y); }
}

```

Most the functions here are virtual without defaults. For the procedures we provide either virtual defaulted definitions or just the definitions.

The key requirement here is simply that some number added to zero returns the original number, as does negating a number twice. We also require subtracting a number from itself yields zero.

Normally addition has stronger requirements such as associativity, but this class is designed to support floating point numbers, and floating addition is not associative.

However since integer addition is, we provide another class:

```

class Addgrp[t] {
  inherit FloatAddgrp[t];
  axiom assoc (x:t,y:t,z:t): (x + y) + z == x + (y + z);
  reduce inv(x:t,y:t): x + y - y => x;
}

```

for an additive group, which just throws in the associative law. The semantic axiom requires cancellation.

Although technically signed integers in Felix do not obey these semantics due to overflow, the real axiom required cannot be modelled in Felix. The true axiom for computer signed integers is that the axioms are obeyed locally, that is, in a compact set close to zero.

This axiom would say, for example, that there exists some d such that if $|a| < d$ and $|b| < d$ and $|c| < d$ then

$$(a + b) + c = a + (b + c)$$

where the restriction is there to ensure there is no overflow. However constructive mathematics cannot model existentials, and instead must provide an actual formula for calculating d . In this case, given the range of the integers, we can actually do that, however in general, even if it can be done, it is all too hard to bother.

Instead we just rely on the programmer understanding that the laws only apply locally and using calculations which they believe will not overflow. In practice many calculations with integers do not overflow, however there are a few very nasty cases.

Chapter 32

More core types

32.1 Slices

A basic slice is just a compact subset of the integers.

32.1.1 Slice Definition

```
union slice[T] =  
  | Slice_all  
  | Slice_from of T  
  | Slice_from_counted of T * T /* second arg is count */  
  | Slice_to_incl of T  
  | Slice_to_excl of T  
  | Slice_range_incl of T * T  
  | Slice_range_excl of T * T  
  | Slice_one of T  
;
```

The `_incl` suffix implies the second argument is included, whilst the `_excl` suffix implies it is excluded.

All slices have a lower bound, if the initial value is not specified, the slice starts at 0.

32.1.2 Syntax

We provide a special syntax for slices as illustrated.

```

var i7up = 7..;           // from 7 up
var i7to11 = 7..11;       // 7 upto 11 inclusive
var i7lt12 = 7..<12;      // 7..11
var iall = ..;            // from 0 up
var ito11 = ..11;         // 0..11
var ilt12 = ..<12;        // 0..11

```

The syntax with `<99` was stolen from Apple's Swift language.

32.1.3 Iterators

The slice type has an iterator method which allows you to enumerate the slice from its lower bound in increasing order. The iterator returns an option type and can be implicitly used in a for loop. The following program:

```

var it = iterator$ 7..8;
for i in 0..3 do
    println$ i,it ();

```

outputs:

```

(0, Some 7)
(1, Some 8)
(2, None)
(3, None)

```

32.1.4 String slices

Slices can be used with strings to extract substrings.

```

var x = "Hello World";
println$ x.[2..4];

```

Before the slice is applied it is clipped to the string dimensions by a special translation. First, negative values are considered to be relative to one past the end of the string going backwards, so the value `-1` for example is the last character in the string, and `-2` is the second last character.

After translation by adding the string length to negative values, the resulting start and end locations are clipped to the range `0..n` where *n* is the string length: the lower bound is the maximum of the calculated value and 0, the upper bound, exclusive, is the minimum of the calculated value and the string length. If the lower bound is greater than or equal to the upper bound exclusive

the result is an empty string, otherwise the substring from the lower bound to the upper bound exclusive is returned.

This sounds complicated but it is what Python does. Note that the substring slicing operation cannot fail.

[TODO: slices can be applied to arrays as well but this is not yet implemented]

32.1.5 Generalised Slices

A generalised slice is an arbitrary sequence of integers built by adding slices together. They also support mapping of slices.

```
union gslice[T] =
  | GSlice of slice[T]
  | GSSList of list[gslice[T]]
  | GSIList of list[T]
  | GSIter of 1 -> opt[T]
  | GSMap of (T -> T) * gslice[T]
;
```

A generalised slice can be built from an ordinary slice using `GSlice`.

It can also be built from a list of integers using `GSIList`, or a list of generalised slices using `GSSList`.

When an iterator of a `gslice` is used, it simply iterates over the underlying slices in order. The resulting stream of integers need not be either compact or ascending, and values can be repeated.

The form `GSIter` is a high power generalisation which allows any iterator over integers to form a `gslice`, without calling the iterator yet. So for example using a `GSSList` of two `GSIter` forms, when iterated over, will use the first iterator until it runs out of values, then move on to the second one.

The `GSMap` provides a way to take any sequence of integers from a `gslice` and translate them using a function.

Generalised slices are not supported by any high level mathematical concept, they're just a grab bag of convenient tools for generating sequences of integers.

32.1.6 Slice addition

Slices and generalised slices can be added to return a generalised slice.

32.1.7 Generalised String slicing

Strings can be mapped by generalised slices:

```
var yank = "11/23/16";
var oz = yank.[(3..5)+(0..2)+(6..7)];
var nip = yank.[(6..7)+(2..2)+(0..4)];
println$ yank,oz,nip;
```

[TODO: negative indices don't appear to work in generalised slices]

32.2 String Type

32.2.1 Equality and total ordering

```
fun == (s:string, c:string): bool
fun < (s:string, c:string): bool
fun <= (s:string, c:string): bool
fun > (s:string, c:string): bool
fun >= (s:string, c:string): bool
```

32.2.2 Equality of string and char

```
fun == (s:string, c:char): bool
fun == (c:char, s:string): bool
fun != (s:string, c:char): bool
fun != (c:char, s:string): bool
```

32.2.3 Append to string object

```
proc += : &string * string
proc += : &string * +char
proc += : &string * char
```

32.2.4 Length of string

```
fun len: string -> size
```

32.2.5 String concatenation

```

fun + : string * string -> string
fun + : string * carray[char] -> string
fun + : string * char -> string
fun + : char * string -> string
fun + ( x:string, y: int):string

```

32.2.6 Repetition of string or char

```

fun * : string * int -> string
fun * : char * int -> string

```

32.2.7 Application of string to string or int is concatenation

```

fun apply (x:string, y:string):string
fun apply (x:string, y:int):string

```

32.2.8 Construct a char from first byte of a string

Returns nul char (code 0) if the string is empty.

```

ctor char (x:string)

```

32.2.9 Constructors for string

```

ctor string (c:char)
ctor string: +char
ctor string: +char * !ints
fun utf8: int -> string

```

32.2.10 Substrings

```

fun subscript: string * !ints -> char
fun copyfrom: string * !ints -> string
fun copyto: string * !ints -> string
fun substring: string * !ints * !ints -> string
fun subscript (x:string, gs:gslice[int]):string
proc store: &string * !ints * char

```

32.2.11 Map a string char by char

```
fun map (f:char->char) (var x:string): string = {
```

32.2.12 STL string functions

These come in two flavours: the standard C++ operations which return `stl_npos` on failure, and a more Felix like variant which uses an `@option` type.

```
const stl_npos: size

fun stl_find: string * string -> size
fun stl_find: string * string * size -> size
fun stl_find: string * +char -> size
fun stl_find: string * +char * size -> size
fun stl_find: string * char -> size
fun stl_find: string * char * size -> size

fun find (s:string, e:string) : opt[size]
fun find (s:string, e:string, i:size) : opt[size]
fun find (s:string, e:+char) : opt[size]
fun find (s:string, e:+char, i:size) : opt[size]
fun find (s:string, e:char) : opt[size]
fun find (s:string, e:char, i:size) : opt[size]

fun stl_rfind: string * string -> size
fun stl_rfind: string * string * size -> size
fun stl_rfind: string * +char-> size
fun stl_rfind: string * +char * size
fun stl_rfind: string * char -> size
fun stl_rfind: string * char * size -> size

fun rfind (s:string, e:string) : opt[size]
fun rfind (s:string, e:string, i:size) : opt[size]
fun rfind (s:string, e:+char) : opt[size]
fun rfind (s:string, e:+char, i:size) : opt[size]
fun rfind (s:string, e:char) : opt[size]
fun rfind (s:string, e:char, i:size) : opt[size]

fun stl_find_first_of: string * string -> size
fun stl_find_first_of: string * string * size -> size
fun stl_find_first_of: string * +char -> size
fun stl_find_first_of: string * +char * size -> size
fun stl_find_first_of: string * char -> size
fun stl_find_first_of: string * char * size -> size
```

```

fun find_first_of (s:string, e:string) : opt[size]
fun find_first_of (s:string, e:string, i:size) : opt[size]
fun find_first_of (s:string, e:+char) : opt[size]
fun find_first_of (s:string, e:+char, i:size) : opt[size]
fun find_first_of (s:string, e:char) : opt[size]
fun find_first_of (s:string, e:char, i:size) : opt[size]

fun stl_find_first_not_of: string * string -> size
fun stl_find_first_not_of: string * string * size -> size
fun stl_find_first_not_of: string * +char -> size
fun stl_find_first_not_of: string * +char * size -> size
fun stl_find_first_not_of: string * char -> size
fun stl_find_first_not_of: string * char * size -> size

fun find_first_not_of (s:string, e:string) : opt[size]
fun find_first_not_of (s:string, e:string, i:size) : opt[size]
fun find_first_not_of (s:string, e:+char) : opt[size]
fun find_first_not_of (s:string, e:+char, i:size) : opt[size]
fun find_first_not_of (s:string, e:char) : opt[size]
fun find_first_not_of (s:string, e:char, i:size) : opt[size]

fun stl_find_last_of: string * string -> size
fun stl_find_last_of: string * string * size -> size
fun stl_find_last_of: string * +char -> size
fun stl_find_last_of: string * +char * size -> size
fun stl_find_last_of: string * char -> size
fun stl_find_last_of: string * char * size -> size

fun find_last_of (s:string, e:string) : opt[size]
fun find_last_of (s:string, e:string, i:size) : opt[size]
fun find_last_of (s:string, e:+char) : opt[size]
fun find_last_of (s:string, e:+char, i:size) : opt[size]
fun find_last_of (s:string, e:char) : opt[size]
fun find_last_of (s:string, e:char, i:size) : opt[size]

fun stl_find_last_not_of: string * string -> size
fun stl_find_last_not_of: string * string * size -> size
fun stl_find_last_not_of: string * +char -> size
fun stl_find_last_not_of: string * +char * size -> size
fun stl_find_last_not_of: string * char -> size
fun stl_find_last_not_of: string * char * size -> size

fun find_last_not_of (s:string, e:string) : opt[size]
fun find_last_not_of (s:string, e:string, i:size) : opt[size]
fun find_last_not_of (s:string, e:+char) : opt[size]

```



```

fun find_last_not_of (s:string, e:+char, i:size) : opt[size]
fun find_last_not_of (s:string, e:char) : opt[size]
fun find_last_not_of (s:string, e:char, i:size) : opt[size]

```

32.2.13 Construe string as set of char

```

instance Set[string,char] {
  fun \in (c:char, s:string) => stl_find (s,c) != stl_npos;
}

```

32.2.14 Construe string as stream of char

```

instance Iterable[string, char] {
  gen iterator(var x:string) () = {
    for var i in 0 upto x.len.int - 1 do yield Some (x[i]); done
    return None[char];
  }
}
inherit Streamable[string,char];

```

32.2.15 Test if a string has given prefix or suffix

```

fun prefix(arg:string,key:string)=
fun suffix (arg:string,key:string)
fun startswith (x:string) (e:string) : bool

// as above: slices are faster
fun endswith (x:string) (e:string) : bool

fun startswith (x:string) (e:char) : bool
fun endswith (x:string) (e:char) : bool

```

32.2.16 h2 Trim off specified prefix or suffix or both

```

fun ltrim (x:string) (e:string) : string
fun rtrim (x:string) (e:string) : string
fun trim (x:string) (e:string) : string

```

32.2.17 Strip characters from left, right, or both end of a string.

```
fun lstrip (x:string, e:string) : string
fun rstrip (x:string, e:string) : string
fun strip (x:string, e:string) : string
fun lstrip (x:string) : string
fun rstrip (x:string) : string
fun strip (x:string) : string
```

32.2.18 Justify string contents

```
fun ljust(x:string, width:int) : string
fun rjust(x:string, width:int) : string
```

32.3 Regexps

[Syntax](#)

[Combinators](#)

[Google Re2 Binding](#)

Chapter 33

Introspection

Chapter 34

Serialisation

Felix allows certain data structures to be serialised into a string, so that the data structure can be recovered from the string.

34.1 Operation

34.1.1 `encode_varray`

This function accepts the address of a serialisable object, and returns a string containing the serialised contents of the object.

34.1.2 `deccode_varray`

This accepts a string previously serialised by `encode_varray`, rebuilds a copy of the data structure it encoded, and returns the address of the object.

34.1.3 Usage

The data structure to be serialised must reside entirely on the heap and satisfy other requirements. A pointer to the object is cast to type `address`. When deserialising, the address returned should be cast to a pointer to the same data type used in serialisation.

```
include "std/felix/serialise";
var px: &list[int] = (1,2,3,4).list.new;
var serialised_x : string = px.addressss.Serialise::encode_varray;
var px_copy = serialised_x.Serialise::decode_varray.&list[int];
println$ *px == *px_copy;
```

34.1.4 Process Restrictions

Serialisation functions depend on the machine address of the Felix run time type information (RTTI) objects which also control garbage collection.

Therefore serialisation is only guaranteed to work within a single process. On some architectures with some linkage strategies it may be possible to save the string to disk, and reloaded it in a different process running the same program. In particular the core requirement must be met by the operating system: to load the process at the same virtual address. If you wish to try this, it is best to statically link your program since different orders of dynamic linkage may result in different addresses for shared library objects (DLLs).

If serialisation works across different processes, it should also work if the serialised data is transmitted to an remote machine running the same architecture, same operating system, and binary identical Felix program.

34.1.5 Data Restrictions

All data to be serialised must either be a C data type bound to Felix with supporting serialisation and deserialisation functions provided in the binding, or, any Felix combination of such primitive data types effected by a tuple, array, struct, union, sum or unit sum type, or pointer type. Recursive data types are supported. Circular data structures do not present a problem.

Most Felix primitive types provided in the library are supported, except for data types derived from Judy arrays.

You cannot serialise function or procedure closures because these refer to the global thread frame object, which is the root for most of the rest of the program data: serialisation would then attempt to capture the whole program. Even if the thread frame were serialisable, deserialisation would create a copy of it which is not supported.

Pointers to C functions, however, are serialisable since they're just addresses. Pointers to an foreign objects remain invariant. Most simple data structures are serialised by simply using a blit operation, namely `memcpy`.

C++ strings are serialisable. Since serialisation produces a C++ string, data structures containing serialised data are also serialisable.

34.1.6 How it works

Serialisation works by copying serialising primitives with user supplied serialisation functions. These currently must be written in C++. Non-primitives are serialised by copying the object and finding subobjects to also serialised using the same pointer location data the garbage collector uses from the types RTTI object.

Serialised data is tagged with the address of the RTTI object so that the deserialisation function and location of interior pointers can be found.

34.1.7 Future Directions

A more portable method of locating RTTI than using the machine address would remove many of the restrictions, in particular the restriction that the exact same program must be used for deserialisation as for serialisation, as well as the restriction that deserialisation should be in the same processes.

In fact the primary difficulty here is finding a unique universal representation of a data type. Were such a kind available a search could be used to map universal type tokens to the RTTI address.

More portable representations of the data may also remove the restriction that the same processor must be run the processes with the same underlying binary ABI (so that, for example, an `int` has the same binary representation on both architectures).

Chapter 35

Fibres and Schannels

[Library Package](#)

Fibres (or f-threads) are coroutines which exchange control with other fibres by reading and writing synchronous channels (schannels).

35.1 `spawn_fthread` procedure

The `spawn_fthread` procedure spawns a new fibre (fthread). The procedure being spawned must have type `unit->void` also written `1->0`.

The new fibre begins execution immediately and continues until suspended by either an unmatched I/O request on an schannel, or a read operation matched by a write.

Fibres are similar to threads, however control exchange is synchronous, not pre-emptive. A communicating set of fibres runs in a pre-emptive single thread.

Fibres can also be suspended waiting for asynchronous events such as socket I/O or a timer.

35.2 Schannel types

There is only one kind of schannel, however the type system provides three views:

ischannel[**T**] an input schannel for reading T values,

oschannel[**T**] an output schannel for writing T values,

ioschannel[**T**] an input/output schannel for either reading or writing T values.

Each of these types is also a constructor for the corresponding type of schannel.

35.3 `mk_ioschannel_pair[T]` function

This function creates a single schannel, returning a read end and a write end in a tuple:

```
var ins, outs = mk_ioschannel_pair[int]();
```

35.4 `ioschannel[T]` constructor

A bidirectional channel can also be created:

```
var ios = ioschannel[int]();
```

35.5 read generator and write procedure

The read and write operators on schannels are used for communication between fibres.

- When a read matches a write, the reader continues with the data, and the writer is placed by on the scheduler active list. If there is more than one writer, the writer chosen is unspecified.
- If a read has no matching write, the reading fibre joins a set of fibres waiting on the schannel for a write, and the scheduler begins execution of one fibre from its active list.
- A write will transfer control and data to one reader. The reader chosen is unspecified. The writer is then placed on the scheduler active list.
- If a write has no matching read, the writing fibres joins a set of fibres waiting for a read, and the scheduler begins execution of one fibre from its active list.

Thus, an schannel is, in effect, a localised scheduler, queueing either writers or readers until matching operations are performed.

35.6 Cross thread I/O

Schannel read and write operations are not thread safe. Schannels should usually not be shared between pthreads to ensure I/O cannot be performed between pthreads using schannel I/O. Pthreads can use `pchannels` to communicate instead. On the other hand fibres must not use `pchannels` to communicate within the same pthread since this would be an automatic deadlock.

35.7 broadcast procedure

This procedure writes to all fibres waiting on an schannel, then returns. Unlike `write` it does not block.

`broadcast` can be used directly from C++. It is intended to permit custom C++ driver code to communicate with a suspended Felix system. The driver code must ensure the data sent with a broadcast remains live whilst target fthreads are processing it.

Felix `broadcast` is invoked by the member function `external_multi_write` of the object of type `async_sched` with type shown below, passing the schannel to broadcast over, and a pointer to the data to broadcast.

```
void async_sched::external_multi_swrite(
    ::flx::rtl::schannel_t *chan,
    void *data
)
```

Felix can be embedded in C++ code, in particular it can run in framework loop idle function as a callback. It runs until there are no active fibres. The `external_multi_write` is used to schedule more active fibres so the next resumption via the idle loop callback has work to do.

35.7.1 Fibre States

Fibres may be in one of five states:

Running Only one fibre can be running at a time in a given pthread.

Waiting to Run A fibre is active if it is on the scheduler active list but not running.

Waiting on Schannel A fibre may be waiting for either input or to output.

Waiting for Asynchrons Event Fibres may also wait for asynchronous events.

A fibre which is running or waiting to run is said to be *active*. A fibre which is waiting on an schannel or asynchronous event is said to be suspended.

35.7.2 Schannel States

An schannel may be in one of three states:

Empty The schannel has no fibres on its wait list.

Waiting for a reader The fibre has one or more writers on its wait list.

Waiting for a writer The fibre has one or more readers on its wait list.

35.7.3 Abbreviated Type names

Felix provides short hand notation for input and output schannels as follows:

alias	full name	semantics
%<T	ischannel[T]	input schannel
%>T	oschannel[T]	output schannel
%<>T	ioschannel[T]	bidirectional schannel

35.7.4 Deadlock, Livelock, and Suicide

Both fibres and schannels are garbage collected. Thus they either be reachable or unreachable.

If a fibre is active it is a root, and is always reachable. If a fibre is waiting for an asynchronous event it is also always reachable.

An schannel may be also be reachable if it is stored in a variable in the global thread frame. This is the global data object which is shared between all pre-emptive threads.

Alternatively it may be reachable from a procedure reachable from an active fibre.

A fibre may terminate by returning. In this case it is unreachable and will be collected. This is known as normal termination.

A fibre which can only be reached from an unreachable schannel will be garbage collected. This is known as suicide and ensures fibres cannot deadlock: deadlocked fibres suicide and so aren't deadlocked because they no longer exist. For example consider:

```
begin
  var i1,o1 = ml_ioschannel_pair[int]();
  var i2,o2 = ml_ioschannel_pair[int]();
  spawn_fthread { write (o1,1); println$ read i2; };
  spawn_fthread { write (o2,2); println$ read i1; };
end
```

The first fthread suspends on channel 1, waiting to write, the second fthread suspends on channel 2, waiting to write. The fthreads are suspended and deadlocked. However the main fthread is active and now runs, and the begin/end block returns so its data frame, containg the two schannels, is no longer reachable. This means the two suspended fthreads are not reachable either. So the two fibres have suicided, they are no longer deadlocked because they no longer exist.

Suicide is not necessarily an error, it is, in fact, a common way to terminate fthreads. The problem above terminates correctly.

Now, consider the program above without the `begin/end` block. In this case, the `fthreads` will not be collected because they're reachable from global store (the so-called thread frame). So that program will not terminate, and is said to be *live-locked*.

It is a live-lock because the main `fthread` can read the channels and thus the suspended fibres and *could* unsuspend them with suitable I/O operations.

It is the responsibility of the holder of an schannel to ensure other holders I/O requests are satisfied. If a fibre does not intend to perform I/O on an schannel it should forget it. In the example the main thread forgets the schannels by forgetting the `begin/end` blocks data frame when it returns.

35.7.5 Restrictions on use

Fibres can only be spawned with a basic `spawn_fthread` by top level procedures or procedural descendants, or, from functions or generator which are inlined (eliminating the function).

This is because procedures spawned with a basic `spawn_fthread` with a service call which requires the machine stack be empty, since the service calls are effected by *returning* control to the scheduler.

Similarly, the same restrictions apply to schannel I/O.

However these restrictions can be partly overcome with nested schedulers.

35.8 Nested Scheduling

A nested scheduler of type `fibre_scheduler` may be created a constructor accepting a unit argument. It must be deleted when no longer required by the procedure `delete_fibre_scheduler`.

Fibres may be added to the scheduler using the advanced `spawn_fthread` procedure, this procedure takes a `fibre_scheduler` argument and returns a closure accepting procedure of type `1->0`. Fibres starting with the procedure are added to the nested scheduler list, but are not yet run.

The `run` procedure takes a populated scheduler and runs the fibres on it.

Fibres on nested schedulers may communicate with each other using schannel operations.

The `run` procedure will not return until all the fibres on the nominated scheduler have completed. The fibres may not do asynchronous operations, or make service calls other than schannel I/O operations, however the scheduler handles schannel I/O properly.

The result of attempting to communicate with fibres on another scheduler such as the master scheduler are undefined.

```
fun foo () =
{
  var chin, chout = mk_ioschannel_pair[int]();
  var sched = fibre_scheduler();
  var res = 1;
  spawn_fthread sched { write (chout,42); };
  spawn_fthread sched { res = read chin; };
  run sched;
  delete_fibre_scheduler sched;
  return res;
}
println$ foo();
```

35.9 Synchronous pipelines

[Library Reference](#) lah.

A synchronous pipeline is a special case of fibres and channels in which data flow from one fibre to a series of others in sequence using a one schannel for each adjacent pair to connect them.

Special constructors are provided to make pipelines which ensure that only the fibres in the pipeline know the connecting schannels.

This ensures that the pipeline collapses when there is no more data to processes.

The starting point of a pipeline is known as a source, the ending point a sink, and the pieces in the middle are called transducers.

A transducer is a control inverted function. This means that instead of being called with data, it reads it, and instead of returning with a result it writes it.

Unlike functions, which form client/server relations, tranduces are all peers.

35.10 Buffers

Synchronous I/O is unbuffered. This means the sequencing of reads and writes is critical. If a fibre writes channel A then B, and another reads B then A, the system is deadlocked and the fibres may suicide.

In order to read a set of inputs in an arbitrary order, buffers can be added to each input.

A buffer is simply an fibre running an infinite loop which reads from one channel and writes on another:

```
var Ain,Aout = mk_ioschannel_pair[int]();
var Bin,Bout = mk_ioschannel_pair[int]();
var Cin,Cout = mk_ioschannel_pair[int]();
var Din,Dout = mk_ioschannel_pair[int]();

proc buffer (r:%<int, w:%>int) () {
  while true perform write (w,read r);
}

spawn_fthread { write (Aout, 1); write (Bout,2); };
spawn_fthread { var d = read Din; c = read Cin; println$ c,d; };
spawn_fthread$ buffer (Ain,Cout);
spawn_fthread$ buffer (Bin,Dout);
```

Now, as you can see, the reader procedure reads the data from A in the reverse order to that in which it is written, but this works because of the buffers. Note that the first fthread can now proceed as soon as the buffer has read its output, before the second fthread gets it. In fact the second fthread gets what it wrote first.

Buffers are mandatory for a feedback.

Chapter 36

Circuits

Felix provides a high level general purpose *chips and wires* sublanguage which is designed to modularise use and deployment of safer interacting fibre based processing systems.

36.1 Chip definition

A chip is designated like:

```
open syntax chips;

chip filereader (filename:string)
  connector file
  pin outch: %>char
{
  var data = load filename;
  for ch in data do
    write (file.outch,ch);
  done
}
```

A chip is an ordinary procedure, which may have parameters such as `filename` in the example. The parser creates another argument for you: designated the `connector`, and named `file` in the example, has the type of a record whose fields are the connectors `pins`, in this case there is only one pin, `outch`, which has type `%>char` which is an abbreviation for `oschannel[char]` and is therefore a synchronous output channel of `char`.

Here is an output device which emits the characters of a string instead:

```

chip stringreader (s:string)
  connector file
  pin outch: %>char
{
  for ch in s do
    write (file.outch,ch);
  done
}

```

Such output devices are called *sources* or *suppliers*.

Now, we want to parse the data stream into words, so we will make a transducer chip.

```

chip parser
  connector pins
  pin inchar: %<char
  pin outword: %>string
{
  var ch = read inchar;
  while true do
    while isspace ch perform
      ch = read inchar
    ;
    var s = ""+ch;
    while not (isspace ch) do
      s += ch;
      ch = read inchar;
    done
    write (outword,s);
  done
}

```

Notice that if the input is exhausted in the middle of a word, the final word is not emitted. This transducer will process and character streams sent to it and then block waiting for data that is not coming. This is the correct way to design circuits. The absence of an explicit termination is the mark of a good design. Circuits should simply collapse when there is no work to do.

To complete the job, we will provide a simple sink or consumer chip:

```

chip printer
  connector pins
  pin inword: %<string
{

```

```
while true do
  var s = read inword;
  println s;
end
}
```

Now, we have all the devices we need so lets wire them together:

```
device src = string_reader "Hello World\n";
circuit
  connect src.outch,parser.inchar
  connect parser.outch, printer.inword
endcircuit
```

Thats it!

[document wire clause and making a chip out of chips]

Chapter 37

Asynchronous Events

Felix uses asynchronous I/O with fibres to provide access to asynchronous events. Currently two kinds of events are supported: alarm clock timeout events and socket I/O buffer and connection events.

Waiting for an event blocks the current fibre, but it does not block other fibres in the same pthread.

37.1 Event Service demux

The event delivery service is called **demux** and runs in a separate preemptive thread. It interfaces with the OS native event delivery services as shown in [Table 37.1 Event Delivery Service Providers](#).

Felix automatically selects the best available service.

37.2 Alarm Clock

The alarm clock service allows a fibre to block for a selected amount of time. A clock must be constructed first.

Table 37.1: Event Delivery Service Providers

Service	Availability	Notes
<code>select</code>	all	fallback
<code>poll</code>	Posix	better than select
<code>epoll</code>	Linux	better than poll
<code>kqueue</code>	BSD Unix, OSX	better than poll
<code>Solaris iocompletion ports</code>	Solaris	better than select
<code>Windows iocompletion ports</code>	Win32	better than select

Table 37.2: Asynchronous Socket Operations

Operation	Used By	Provider class
Platform Specific		
connect	Client	Faio_posix
accept	Server	Faio_posix
async_rw	Client and Server	Faio_posix
async_read	Client and Server	Faio_posix
async_write	Client and Server	Faio_posix
Connect	Client	Faio_win32
Accept	Server	Faio_win32
WSARecv	Client and Server	Faio_win32
WSASend	Client and Server	Faio_win32
TransmitFile	Client and Server	Faio_win32
Platform Independent		
connect	Client	Socket
accept	Server	Socket
read	Server	Socket
write	Server	Socket

```
fun mk_alarm_clock: 1 -> alarm_clock;

// delay in seconds
proc sleep(clock: alarm_clock, delta: double);
```

For example to delay 5 seconds use:

```
var clock = mk_alarm_clock();
sleep (clock, 5.0);
```

Note again this only causes the current active fibre to sleep. Clocks are reusable and may delays can be queued by any number of fibres within the same pthread.

37.3 Sockets

Socket operations are covered more completely in ?? ??. Socket operations are synchronous unless shown in [Table 37.2 Asynchronous Socket Operations](#).

37.4 How Async I/O works

Asynchronous I/O is performed as follows. First an fthread

Unfortunately shutdown and close requests are synchronous due to a design fault in sockets and TCP/IP protocol. Although Posix supports asynchronous shutdown, this may cause underlying socket buffers to be prematurely flushed. The only way to asynchronously shutdown a socket would be to perform a synchronous shutdown in a dedicated pthread, defeating the purpose of using fibres.

Chapter 38

Preemptive Threading

[Library Package](#) Felix provides pre-emptive threading based on OS native threads called *pthreads*.

Felix pthreads are detached, there is no support for OS joinable threads. The primary synchronisation primitive is the *pchannel*.

38.1 `spawn_pthread` procedure

The `spawn_pthread` procedure spawns a new pthread. The procedure being spawned must have type `unit->void` also written `1->0`.

38.2 Pchannel types

There is only one kind of pchannel, however the type system provides three views:

`ipchannel[T]` an input pchannel for reading T values,

`opchannel[T]` an output pchannel for writing T values,

`iopchannel[T]` an input/output pchannel for either reading or writing T values.

Each of these types is also a constructor for the corresponding type of pchannel.

38.3 `mk_iopchannel_pair[T]` function

This function creates a single pchannel, returning a read end and a write end in a tuple:

```
var inp, outp = mk_iopchannel_pair[int]();
```

38.4 iopchannel[T] constructor

A bidirectional channel can also be created:

```
var ios = iopchannel[int]();
```

38.5 read generator and write procedure

The read and write operators on pchannels are used for communication between pthreads.

Pchannels are technically *monitors*. There is no buffering. Unmatched reads block until a matching write occurs on the same underlying pchannel. Unmatched writes block until a matching read occurs. Neither reader nor writer may proceed until data is exchanged.

Unlike fibres and schannels, pthread can be deadlock by out of order I/O.

38.6 concurrently procedure

```
open class ForkJoin {
  proc concurrently_by_iterator (var it:1 -> opt[1->0])
  proc concurrently[T with Streamable[T,1->0]] (d:T)
}
```

Felix provides the procedure `concurrently_by_iterator` which uses an iterator returning procedures of type `1->0` to launch pthreads.

A more convenient procedure, `concurrently`, accepts any data structure with an iterator.

`concurrently` automatically joins the launched threads.

```
// using an array
concurrently (
  { println$ "Thread1"; },
  { println$ "Thread2"; },
  { println$ "Thread3"; }
);
```

38.7 Atomics

Felix atomics utilise the C++11 standard library atomics. Atomic variables are not managed by the GC.

```
open class Atomic
{
  type atomic[T];
  ctor[T] atomic[T]: T;
  proc delete[T] : atomic[T];

  gen load[T] : atomic[T];
  proc store[T] : atomic[T];
  proc store[T] (a:atomic[T]) (v:T);

  // integer types only
  proc pre_incr[T] : &atomic[T];
  proc pre_decr[T] : &atomic[T];
}
```

38.8 Mutual Exclusion Lock

```
open class Mutex
{
  type mutex;
  ctor mutex: unit;
  proc lock: mutex;
  proc unlock: mutex;
  proc destroy: mutex;
}
```

Felix provides a garbage collector aware mutual exclusion lock `mutex`. It is a timed mutex which periodically checks for a GC world stop request.

Condition variables are not garbage collected.

38.9 Condition Variables

Felix provides a GC aware condition variable based on C++11 condition variables. It uses a timed variant to periodically check for a GC world stop request.

```

open class Condition_Variable
{
    type condition_variable;
    ctor condition_variable: unit;
    proc destroy: condition_variable;

    //ℓ lock/unlock associated mutex
    proc lock : condition_variable;
    proc unlock : condition_variable;

    proc wait: condition_variable;
    proc signal: condition_variable;

    proc broadcast: condition_variable;

    gen timedwait: condition_variable * double -> int;
}

```

Note that unlike Posix condition variables, the associated mutex is built-in to the condition variable.

38.10 Bound Queue

Felix provides thread safe, GC aware, resizeble bound queue. The queue itself is garbage collected. The queue stores pointers to heap allocated copies of the queued objects, which are scanned by the GC.

```

open class TS_Bound_Queue
{
    ctor[T] ts_bound_queue_t[T]: !ints;
    proc enqueue[T] (Q:ts_bound_queue_t[T]) (elt:T);
    gen dequeue[T] (Q:ts_bound_queue_t[T]): T;
    proc wait[T]: ts_bound_queue_t[T]; // until Q empty
    proc resize[T]: ts_bound_queue_t[T] * !ints;
}

```

The constructor creates a queue with the specified maximum capacity or *bound*.

The **enqueue** operation may wait until there is space in the queue before depositing its payload and returning.

The **dequeue** operation may wait whilst the queue is empty until it can acquire a payload before returning.

The `wait` procedure blocks until the queue is empty.

The `resize` procedure changes the queue bound.

Note: When threads are enqueueing and dequeuing from the queue, there is no defined method for closing down the use of the queue. Neither `enqueue` nor `dequeue` can fail, nor can they be cancelled.

38.11 Thread Pool

```
class ThreadPool
{
  typedef job_t = 1 -> 0;
  fun get_nthreads () => nthreads;

  proc barrier();
  proc start ();
  proc start (n:int);
  proc queue_job (job:job_t);
  proc stop ();
  proc post_barrier();
  proc notify (chan:opchannel[int]) ();
  proc join ();

  proc pfor_segment (first:int, last:int) (lbody: int * int -> 1 -> 0)
  inline proc forloop (lbody: int -> 0) (first:int, last:int) ()
}
```

Felix provides a global thread pool which consists of a thread safe job queue and a set of running Felix pthreads. The number of threads is determined from the architecture if possible, otherwise set to a reasonable value such as 8.

The job queue is bound to 1024 jobs.

The `queue_job` procedure accepts a unit procedure of type `1->0` and adds it to the job queue. It may block whilst the job queue is full. Queuing a job automatically starts up the thread pool.

Threads in the pool dequeue jobs from the queue and run them. When finished, a thread fetches another job if there is one, and runs it. If there are no jobs, the thread waits until a job is available.

The `start` procedure can be used to manually start the thread pool.

The `post_barrier` procedure posts `n` barrier jobs, where `n` is the number of threads in the pool. A barrier job causes a thread to stop acquiring jobs and wait until all threads are waiting on a barrier.

The `join` procedure posts n barriers, where n is the number of threads in the pool. This causes each thread to suspend on a barrier until all threads have suspended. Then all threads may continue processing jobs.

The `stop` procedure attempts to stop the pool by posting a n special `ThreadStop` jobs, where n is the number of threads in the pool. A `ThreadStop` job causes the thread executing it to suicide.

38.11.1 Restrictions

Jobs submitted to the thread pool should be either be entirely independent or transiently coupled. In general, jobs should not communicate with each other or other threads using condition variables or pchannels. It is counter productive for jobs to spawn new threads although it is not disallowed, because the thread pool is designed to saturate multi-core CPUs (that is, keep all the cores busy).

Transient coupling, however, is permitted. For example if jobs are used to calculate some value which is then added to a single variable, a mutex can be used to serialise access to the variable, because the lock will only be held transiently.

38.12 Parallel For Loop

38.12.1 Syntax

```
loop_stmt :=
  "pfor" sname "in" sexpr "upto" sexpr block
;
```

38.12.2 Operation

The parallel for loop constructs n subjobs which are submitted to the thread pool, where n is approximately the number of threads in the thread pool. Each job runs a loop over a subrange of the loop range so that all the indices in the range are covered.

The thread pool is automatically started if it wasn't running. Note: it is not automatically stopped. You must manually call `ThreadPool::stop()` if you use a `pfor` statement otherwise Felix will not terminate.

38.12.3 Example

The `pfor` loop was specifically designed for matrix multiplication. The following code multiplies a by b^T where the latter is the transpose of b , and puts the result in r . The matrices are square of dimension N . The inner product is captured in a function and returns a single value.

```
pfor i in 0 upto (N - 1) do
  for (var j=0; j<N; ++j;)
    perform &r . i . j <- inner_product (&a.i, &b.j);
  done
```

38.12.4 Restrictions

It is essential the body of the `pfor` loop behaves independently for each value of the control index, or uses lock free coupling such as adding to a shared atomic variable.

`pfor` loops must not be nested. It would be counter productive, but it will also most likely cause a deadlock.

Part VII

Platform Support

Chapter 39

Platform Modelling

The Felix language processor is a translator which outputs C++. Because of this, the translator can produce code intended to run on one platform on an unrelated platform. For example, Windows native programs can be generated on Linux, although there may not be a suitable compiler and library available to produce a binary, or an emulator to run it.

So for platform related libraries, Felix provides three kind of code:

- platform specific code
- platform independent code
- platform adaptive code

Platform specific code consists of functions which are specific to the platform.

Platform independent code uses a type class to represent a common subset of functions with the same interface. The type parameter is used to represent the platform variable.

Class instances then bind the virtuals of the class for each supported platform.

By convention the class name ends in `_class`.

Platform adaptive code is the simplest to use, and simply provides the platform dependent code for the host platform. The host is specified by macro file which is included with parse units, and conditional compilation is used for the selection.

Note that *in addition* to the above Felix level modelling, similar modelling methods can be used in C++ libraries to which Felix binds. Therefore a call to a function like `Shell::system()` can be mapped in Felix in a platform independent way to the underlying call in the ISO C library.

The client may write platform independent code, accepting the restrictions of common functionality, but the code must reside in a class with OS control

parameter which can be passed on to the platform mapping virtual functions.

Note that Felix with the C++ backend is a cross-cross compiler, not just a cross compiler! There do exist C++ compilers that run on Linux but produce Windows binaries. Felix front end such a compiler, and produce Windows C++ for it.

Chapter 40

Processes

Blah.

Chapter 41

Programs

41.1 Basic Program interface

```
class System
{
    const argc:int;
    fun argv:int -> string;
    fun argv_dflt (x:int) (y:string) : string;
    fun args () : list[string];
    gen system (cmd:string) : int => Shell::system(cmd);
    gen exit: int -> any;
    gen abort: 1 -> any;
    const ptf:ptf_t;

    //£ pexit examines the return code from a system call.
    //£ If the code is 0 it exists with 0.
    //£ On Windows:
    //£     if the code is -1, it exits with errno.
    //£     otherwise code 3
    //£ On Unix:
    //£     if the code is non-zero then
    //£         if the callout aborted, return its abort code.
    //£         if the callout died due to a signal, exit with code 2
    //£         otherwise exit with code 3
    //£ In both these cases a non-zero return causes a message
    //£ to be printed on stderr.

    proc pexit(e:int)
    gen get_stdout(x:string) : int * string => Shell::get_stdout x;
```

```
}

```

41.2 Environment Variables

```
class Env_class[os]
{
  //£ Separator for filename lists
  virtual fun pathsep: 1 -> string;

  //£ Get the value of a given variable.
  //£ Returns empty string if the variable doesn't exist.
  fun getenv:string -> string =
    "::-flx::rtl::strutil::atostr(std::getenv($1.c_str()))"
    requires package "flx_strutil", Cxx_headers::cstdlib;

  //£ Get the value of a given variable.
  //£ Returns specified default if the variable doesn't exist.
  fun getenv(name:string,dflt:string):string=>let result = getenv(name) in
    if String::len result != 0uz then result else dflt endif
  ;

  fun getenvlist (name:string) : list[string] =>
    split (getenv name, #pathsep)
  ;
}

```

41.3 Shell callouts

```
class Shell_class[OS, process_status_t]
{
  // Quote a single argument.
  // Note: kills Bash wildcard replacement.
  virtual fun quote_arg:string->string;
  fun quote_args (s:list[string]) : string => catmap[string] ' ' quote_arg s;

  // Mainly for Windows we need a way to quote command line strings too.
  virtual fun quote_line_for_system: string->string;

  virtual fun parse: string -> list[string];

  //-----

```



```

// system() function

//£ System command is ISO C and C++ standard.
gen raw_system: string -> int = "::std::system($1.c_str())"
    requires Cxx_headers::cstdlib
;

//£ basic command with line quoting.
gen basic_system (cmd: string) :int =>
    cmd.quote_line_for_system.raw_system
;

// string argument
// responds to FLX_SHELL_ECHO environment variable
gen system (cmd:string) = {
    if Env::getenv "FLX_SHELL_ECHO" != "" do
        eprintln$ "[system] " + cmd;
    done
    return basic_system cmd;
}

// list of string argument
gen system (args:list[string]) : int =>
    args.quote_args.system
;

gen system[T with Iterable[T,string]] (args:T) : int =
{
    var lst = Empty[string];
    for arg in args do
        lst = lst + arg;
    done
    return system lst;
}

virtual fun quote_line_for_popen: string -> string;

//£ get_stdout is a synchronous version of popen_in/pclose pair.
virtual gen raw_get_stdout : string -> int * string;

gen basic_get_stdout (cmd: string) : int * string =>
    cmd.quote_line_for_popen.raw_get_stdout
;

gen get_stdout (cmd:string) : int * string = {
    if Env::getenv "FLX_SHELL_ECHO" != "" do

```

```
        eprintln$ "[get_stdout] " + cmd;
    done
    return basic_get_stdout cmd;
}

// arbitrary Streamable argument
gen get_stdout (args:list[string]) : int * string =>
    args.quote_args.get_stdout
;

gen get_stdout[T with Iterable[T,string]] (args:T) : int * string;
}
```

41.4 Signals

Chapter 42

File System

Chapter 43

Dynamic Linkage

43.1 Basic Library

43.2 Plugins

Part VIII

Data Structures

Chapter 44

Iterators

Chapter 45

Lists

45.1 Inductive lists

45.1.1 Description

Lists are the core data structure for most functional programming languages. A list is an inductive data type: it is defined by a base term, `Empty[T]`, and an induction which says if x is a list of T , and v is a value of T , then `Cons (v,x)` is a list.

The induction model of construction admits to a recursive model of destruction, for example:

```
fun str[T with Str[T]] (x:list[T]) : string =>
  match
  | Empty => "Empty"
  | Cons (head, tail) => head.str + ", " + str tail
  endmatch
;
```

In fact, Felix defines Snoc lists rather than Cons lists, using a `Snoc(list,value)` constructor based on a C++ implementation, instead of a `Cons (value, list)` constructor and then provides a translation with a user defined type constructor. The idea is to get the tail pointer first in the underlying data type, so that polymorphic visitors can be written without the offsets depending on the value type T .

```
open class List
{
  union list[T] = | Empty | Snoc of list[T] * T;
```

```

fun _match_ctor_Cons[T] : list[T] -> bool = "!!$1";
inline fun _ctor_arg_Cons[T]: list[T] -> T * list[T] =
  "flx::list::snoc2cons<?1>($1)"
  requires snoc2cons_h
;
inline fun Cons[T] (h:T, t:list[T]) => Snoc (t,h);

```

45.1.2 Constructors

copy function

Make an entirely new copy of a list. Primarily a helper.

```

fun copy[T] (x:list[T]):list[T];

```

Copy and return last copy_last

Yet another helper procedure.

```

proc copy_last[T] (inp:list[T], out:&list[T], last:&list[T]) {
  out <- rev inp;
  rev_last (out, last);
}

```

Named constructor for empty list

```

ctor[T] list[T] () => Empty[T];

```

Construct a singleton list

Does not work if the argument is an array or option iterator.

```

//£ Make a list with one element.
//£ NOTE: list (1,2) is a list of 2 ints.
//£ To get a list of one pair use list[int*int] (1,2) instead!
ctor[T] list[T] (x:T) => Snoc(Empty[T],x);

```

Construct a list from an array

```

//£ Make a list from an array.
ctor[T,N] list[T] (x:array[T, N]) : list[T];

```


List comprehension.

Make a list from an option stream. Named variant.

```
//£ List comprehension:
//£ Make a list from a stream.
fun list_comprehension[T] (f: (1->opt[T])) = {
```

Constructor variant.

```
//£ List comprehension:
//£ Make a list from a stream.
ctor[T] list[T] (f: (1->opt[T])) => list_comprehension f;
\end{minted}

\subsection{Construe a list as an array value}
\begin{minted}
//£ Construe a list as an array value
instance[T] ArrayValue[list[T],T] {
  //£ Return number of elements in a list.
  pure fun len (x:list[T]) = {

    //£ get n'th element
    pure fun unsafe_get: list[T] * size -> T =

    //£ Apply a procedure to each element of a list.
    proc iter (_f:T->void) (x:list[T]) {

    //£ Traditional left fold over list (tail rec).
    fun fold_left[U] (_f:U->T->U) (init:U) (x:list[T]):U =

    //£ Right fold over list (not tail rec!).
    fun fold_right[U] (_f:T->U->U) (x:list[T]) (init:U):U =

  }
}
```

45.1.3 Destructors

Test for empty list is_empty

```
//£ Test if a list is empty.
pure fun is_empty[T] : list[T] -> 2 =
```

Tail of a list tail

```
//£ Tail of a list, abort with match failure if list is empty.
pure fun tail[T] (x:list[T]) : list[T] = {
```

Head of a list head

```
//£ Head of a list, abort with match failure if list is empty.
pure fun head[T] (x:list[T]) : T = {
```

45.1.4 Maps**Reverse map a list rev_map**

Tail recursive.

```
//£ map a list, return mapped list in reverse order (tail rec).
fun rev_map[T,U] (_f:T->U) (x:list[T]): list[U] = {
```

Map a list map

Tail recursive. Uses `rev_map` and then inplace reversal. This is safe because we enforce linearity by abstraction.

```
//£ map a list (tail-rec).
// tail rec due to in-place reversal of result.
fun map[T,U] (_f:T->U) (x:list[T]): list[U] =
```

Reverse a list rev.

Tail recursive.

```
//£ reverse a list (tail rec).
pure fun rev[T] (x:list[T]):list[T]= {
```

Zip a pair of lists to a list of pairs zip2

Returns a list the length of the shortest argument.

```
//£ Zip two lists into a list of pairs.
//£ Zips to length of shortest list.
fun zip2[T1,T2] (l1: list[T1]) (l2: list[T2]) : list[T1 * T2] =
```

45.1.5 Useful lists

A list of integers range.

From low to @high exclusive with given @step.

```
//£ Generate an ordered list of ints between low and high with given step.
//£ Low included, high not included.
fun range (low:int, high:int, step:int) =
```

Consecutive integers range

```
//£ Range with step 1.
fun range (low:int, high:int) => range(low, high, 1);
```

Non-negative integers to limit range

num integers 0 to num-1.

```
//£ Range from 0 to num (excluded).
fun range (num:int) => range(0, num, 1);
```

45.1.6 Operators

Concatenate two lists join.

```
//£ Concatenate two lists.
fun join[T] (x:list[T]) (y:list[T]):list[T] =

//£ Concatenate two lists.
pure fun + [T] (x:list[T], y: list[T]):list[T] => join x y;
```

Cons an element onto a list.

```
//£ Prepend element to head of list.
pure fun + [T] (x:T, y:list[T]):list[T] => Snoc(y,x);
```

Append an element onto a list.

O(N) slow.

```

//£ Append element to tail of list (slow!).
noinline fun + [T] (x:list[T], y:T):list[T] => rev$ Snoc (rev x,y);

//£ Append element to tail of list (slow!).
proc += [T] (x:&list[T], y:T) { x <- *x + y; }

```

Concatenate a list of lists cat

```

//£ Concatenate all the lists in a list of lists.
noinline fun cat[T] (x:list[list[T]]):list[T] =

```

45.1.7 Lists and Strings

Pack list of strings into a string with separator cat

```

//£ Concatenate all the strings in a list with given separator.
pure fun cat (sep:string) (x:list[string]):string =

```

Map a list to a list of strings and cat with separator catmap

```

fun catmap[T] (sep:string) (f:T -> string) (ls: list[T]) =>

fun strcat[T with Str[T]] (sep: string) (ls: list[T]) =>

fun strcat[T with Str[T]] (ls: list[T]) =>

```

45.1.8 Searching

Value membership

```

//£ Return true if one value in a list satisfies the predicate.
fun mem[T] (eq:T -> bool) (xs:list[T]) : bool =>

//£ Return true if one value in the list satisfies the relation
//£ in the left slot with
//£ the given element on the right slot.
fun mem[T, U] (eq:T * U -> bool) (xs:list[T]) (e:U) : bool =>

//£ Construe a list as a set, imbuing it with a membership
//£ test, provided the element type has an equality operator.

```

```
instance [T with Eq[T]] Set [list[T], T] {
  fun \in (x:T, a:list[T]) => mem[T,T] eq of (T * T) a x;
}
```

Value Find by relation find

Returns option.

```
//£ return option of the first element in a list satisfying the predicate.
fun find[T] (eq:T -> bool) (xs:list[T]) : opt[T] =>

//£ Return option the first value in the list satisfies the relation
//£ in the left slot with
//£ the given element on the right slot.
fun find[T, U] (eq:T * U -> bool) (xs:list[T]) (e:U) : opt[T] =>

//£ Return a sub list with elements satisfying the given predicate.
noinline fun filter[T] (P:T -> bool) (x:list[T]) : list[T] =

//£ Push element onto front of list if there isn't one in the
//£ list already satisfying the relation.
fun prepend_unique[T] (eq: T * T -> bool) (x:list[T]) (e:T) : list[T] =>

//£ Attach element to tail of list if there isn't one in the
//£ list already satisfying the relation.
fun insert_unique[T] (eq: T * T -> bool) (x:list[T]) (e:T) : list[T] =>

//£ Remove all elements from a list satisfying relation.
fun remove[T] (eq: T * T -> bool) (x:list[T]) (e:T) : list[T] =>

//£ Attach element to tail of list if there isn't one in the
//£ list already satisfying the relation (tail-rec).
noinline fun append_unique[T] (eq: T * T -> bool) (x:list[T]) (e:T) : list[T] = {

//£ Take the first k elements from a list.
fun take[T] (k:int) (lst:list[T]) : list[T] =>

//£ Drop the first k elements from a list.
fun drop[T] (k:int) (lst:list[T]) : list[T] =>

fun list_eq[T with Eq[T]] (a:list[T], b:list[T]): bool =>
instance [T with Eq[T]] Eq [list[T]] {
  fun ==(a:list[T], b:list[T])=> list_eq(a,b);
}
```

45.1.9 Sort

```
//£ Sort a list with given less than operator, which must be
//£ total order. Uses varray sort (which uses STL sort).
fun sort[T] (lt:T*T->bool) (x:list[T])=

//£ Sort a list with default total order.
//£ Uses varray sort (which uses STL sort).
fun sort[T with Tord[T]](x:list[T])=> sort lt x;
```

45.1.10 Streaming list

```
instance[T] Iterable[list[T],T] {
  //£ Convert a list to a stream.
  gen iterator (var xs:list[T]) ()
```

45.1.11 List syntax

```
syntax listexpr
{
  //£ List cons, right associative.
  x[sarrow_pri] := x[>sarrow_pri] "!" x[sarrow_pri];
}
```

45.2 Association List

A list of pairs

```
open class Assoc_list
{
  typedef assoc_list[A,B] = List::list[A*B];

  // check is the key (left element) of a pair
  // satisfies the predicate
  fun mem[A,B] (eq:A -> bool) (xs:assoc_list[A,B]) : bool =>

  // check is the key (left element) of a pair
  // satisfies the relation to given element
  fun mem[A,B,T] (eq:A * T -> bool) (xs:assoc_list[A,B]) (e:T) : bool =>

  instance[A,B] Set[assoc_list[A,B], A] {
```

```

    fun mem[A,B with Eq[A]] (xs:assoc_list[A,B]) (e:A) : bool =>
    }

    // find optionally the first value whose associate key satisfies
    // the given predicate
    fun find[A,B] (eq:A -> bool) (xs:assoc_list[A,B]) : opt[B] =>

    // find optionally the first value whose associate key (left slot)
    // satisfies the given relation to the given element (right slot)
    fun find[A,B,T] (eq:A * T -> bool) (xs:assoc_list[A,B]) (e:T) : opt[B] =>

    fun find[A,B with Eq[A]] (xs:assoc_list[A,B]) (e:A) : opt[B] =>
    }

```

45.3 Purely Functional Random Access List.

45.3.1 Definition

```

//£ Purely functional Random Access List.
//£ Based on design from Okasaki, Purely Functional Datastructures.
//£ Transcribed from Hongwei Xi's encoding for ATS2 library.
//£
//£ An ralist provides  $O(\log N)$  indexed access and amortised
//£  $O(1)$  consing. This is roughly the closest thing to
//£ purely functional array available.

class Ralist
{

    //£ Auxilliary data structure.
    union pt[a] = | N1 of a | N2 of pt[a] * pt[a];

    //£ Type of an ralist.
    union ralist[a] =
        | RAnil
        | RAevn of ralist[a]
        | RAodd of pt[a] * ralist[a]
    ;
}

```

45.3.2 Length

```
//£ Length of an ralist.
fun ralist_length[a] : ralist[a] -> int =
```

45.3.3 Cons

```
//£ Cons: new list with extra value at the head.
fun ralist_cons[a] (x:a, xs: ralist[a]) =>
```

45.3.4 Check if empty

```
//£ Check for an empty list.
fun ralist_empty[a]: ralist[a] -> bool =
```

45.3.5 Split into head and tail

```
//£ Procedure to split a non-empty ralist
//£ into a head element and a tail.
proc ralist_uncons[a] (xs: ralist[a], phd: &a, ptl: &ralist[a])
```

45.3.6 Pattern Matching Support

```
//£ User define pattern matching support
fun _match_ctor_Cons[T] (x:ralist[T]) =>not ( ralist_empty x);
fun _match_ctor_Empty[T] (x:ralist[T]) => ralist_empty x;
fun _ctor_arg_Cons[T] (x:ralist[T]) : T * ralist[T] =
```

45.3.7 Head

```
//£ Head element of a non-empty ralist.
fun ralist_head[a] (xs: ralist[a]) : a =
```

45.3.8 Tail

```
//£ Tail list of a non-empty ralist.
fun ralist_tail[a] (xs: ralist[a]) : ralist[a] =
```


45.3.9 Random Access

```
//£ Random access to an ralist. Unchecked.
fun ralist_lookup[a] (xs:ralist[a],i:int)=>
```

45.3.10 Update

```
//£ Return a list with the i'th element replaced by x0.
//£ Index is unchecked.
fun ralist_update[a] (xs:ralist[a], i:int, x0:a) =>
```

45.3.11 Iteration

```
//£ Callback based iteration.
//£ Apply procedure to each element of the ralist.
proc ralist_foreach[a]
(
  xs: ralist[a],
  f: a -> void
)
)
```

45.3.12 Show

```
//£ Convert ralist to a string.
instance[a with Str[a]] Str[ralist[a]]
{
  fun str (xx: ralist[a]):string = {
  }
```

45.4 Doubly Linked Mutable List

A `dlist_t` is a doubly linked mutable list. It is suitable for use as non-thread-safe queue.

```
class DList[T]
{
  typedef dnode_t=
  (
    data: T,
```

```

    next: cptr[dnode_t], // possibly NULL
    prev: cptr[dnode_t]  // possibly NULL
);
typedef dlist_t = (first:cptr[dnode_t], last:cptr[dnode_t]);
    // invariant: if first is null, so is last!

ctor dlist_t () => (first=nullptr[dnode_t],last=nullptr[dnode_t]);

```

45.4.1 Length len

```

fun len (x:dlist_t) = {

```

45.4.2 Inspection

```

fun peek_front (dl:dlist_t) : opt[T] =>
fun peek_back (dl:dlist_t) : opt[T] =>

```

45.4.3 Insertion

```

proc push_front (dl:&dlist_t, v:T) {
proc push_back (dl:&dlist_t, v:T) {

```

45.4.4 Deletion

```

gen pop_front (dl:&dlist_t): opt[T] = {
gen pop_back (dl:&dlist_t): opt[T] = {

```

45.4.5 Use as a queue

We can implement enqueue and dequeue at either end, we'll make enqueue `push_front` and dequeue `pop_back` for no particular reason.

```

typedef queue_t = dlist_t;
proc enqueue (q:&queue_t) (v:T) => push_front (q,v);
gen dequeue (q:&queue_t) :opt[T] => pop_back q;
ctor queue_t () => dlist_t ();

```

45.4.6 Queue iterator

Fetch everything from a queue.

```
gen iterator (q:&queue_t) () => dequeue q;
```

45.5 S-expressions

A scheme like data structure.

```
class S_expr
{
  union sexpr[T] = Leaf of T | Tree of list[sexpr[T]];

  fun fold_left[T,U] (_f:U->T->U) (init:U) (x:sexpr[T]):U =>
  proc iter[T] (_f:T->void) (x:sexpr[T]) {

  fun map[T,U] (_f:T->U) (x:sexpr[T]):sexpr[U] =>

  instance[T with Eq[T]] Set[sexpr[T],T] {
    fun \in (elt:T, x:sexpr[T]) =>
  }
  instance[T with Str[T]] Str[sexpr[T]] {
    noinline fun str(x:sexpr[T])=>
  }
}
```

45.6 LS-expressions

A scheme like data structure, similar to sexpr, only in this variant the tree nodes also have labels.

```
class LS_expr
{
  union lsexpr[T,L] = | Leaf of T | Tree of L * list[lsexpr[T,L]];

  fun fold_left[T,L,U] (_f:U->T->U) (_g:U->L->U) (init:U) (x:lsexpr[T,L]):U =>

  proc iter[T,L] (_f:T->void) (_g:L->void) (x:lsexpr[T,L]) {
  fun map[T,L,U,V] (_f:T->U) (_g:L->V) (x:lsexpr[T,L]):lsexpr[U,V] =>
  instance[T,L with Str[T], Str[L]] Str[lsexpr[T,L]] {
    noinline fun str(x:lsexpr[T,L])=>
  }
}
```

```
}
```

Chapter 46

Arrays

46.1 Native Arrays

46.2 Varrays

46.3 Darrays

46.4 Sarrays

46.5 C arrays

Part IX

Appendices

Appendix A

Special Names

- `_make_type_name` Cast the representation of an abstract type to the abstract type.
- `_repr_` Cast an abstract type to its representation.
- `_map` Generic map.
- `_eq` Generic equality.
- `_rev` Tuple and tuple type reversal. Applied after monomorphisation. Works on arrays too using library function.
- `_map` Tuple map and tuple type map. Applied after monomorphisation. Works on arrays too.
- `_rev_map` Reversing tuple map and tuple type map. Applied after monomorphisation. Works on arrays too using library function.
- `_strr` Generic conversion to string. Supports records and tuples. Delegates primitives and arrays to library function `str`.
- `_tie` Converts a pointer to a tuple, to a tuple of pointers to the components of the original tuple.
- \prod Product of functions. Prefix quantifier over tuple.
- \sum Sum of functions. Prefix quantifier over tuple.
- $\langle \cdot \rangle$ Mediating morphism of product of functions.
- $[\cdot]$ Mediating morphism of sum of functions.
- `apply` User defined application.
- `typeof` Denotes type of an expression. The expression is not evaluated.
- `_match_ctor_name` User defined pattern match constructor check.

`_ctor_arg_name` User defined pattern match extractor.

`_ctor_arg_type_name` User defined constructor/conversion function.

`TYPE` The kind of all types.

`GENERIC` A psuedo kind used to qualify a type variable in a definition which causes the compiler to generate a fresh definition for each instance type actually used on demand.

`_felix_type_name` Trait reporting bound type name as string.

```
_felix_type_name[int * long] // "int * long"
```

`_scheme` An application of the `_scheme` psuedo function to a string causes the macro processor to execute the string as embedded Scheme code, resulting in an S-expression. The S-expression is then converted into Felix expression, in the same way the S-expressions generated by the parser are converted into expressions.

It can also be used as a procedure name in statement.

`_str` An application of `_str` to any Felix expression causes the macro processor to translate the expression into a string representing the AST terms for the expression as would be presented in a compiler diagnostic. Useful to examine exactly how the parser translates a particular Felix expression.

`range_check` Applied to a tuple of three expressions, this psuedo function is translated by the macro processor into a system range check expression which checks that the middle term is greater than or equal to the first term and less than the third term. If the conditions are not met, an exception will be raised at run time.

Glossary

coroutine a programming unit which exchanges control with others on a neutral basis. 18

polymorphism a language feature which allows a function to process arguments of more than one type, and to behave differently depending on the argument type. 18

setwise lookup a method of lookup where a symbol may be used before or after it is defined. 130

Index

- #line
 - directive, 101
- , 65
- action
 - scheme, 88
- addition
 - numeric, 206
- address
 - variable, 68
- algebra
 - numeric types, 205
- algorithm
 - lookup, 117
- append
 - string, 212
- application
 - reverse, 67
 - reverse with address, 67
 - reverse with deref, 67
 - unit, 68
- apply
 - non-function, 80
- argument
 - parser, 69
- arity
 - function, 77
- ascii
 - classification, 203
- assertion
 - axiom, 51
- asynchronous
 - event, 232
 - IO, 233
- atomic
 - type, 237
- axiom, 51
- binding
 - C, 183
 - type, 183
- body
 - floating insertion, 190
- bool, 71
 - literal, 104
 - type, 196
- break, 49
- C comments
 - grammar, 100
- C++ comments
 - grammar, 100
- call_with_trap, 46
- casenp, 66
- cflags
 - binding, 190
- char
 - type, 203
- class
 - nested, 129
 - statement, 129
- coercion, 66
- comments
 - grammar, 100
 - nested, 100
- complex
 - type, 200
- composition, 65
- conditional, 72
- const
 - statement, 187
- constraints
 - type, 173
- constructor, 80
- continuation, 46

- continue, 49
- conversion, 80
 - automatic, 119
 - binding, 188
 - operator, 188
- dcomplex, 200
- decoder
 - clause, 187
- definition
 - private, 130
- definitions, 75
- dereference, 65
- ellipsis, 71
- encoder
 - clause, 187
- enum
 - binding, 190
- environment
 - lookup, 117
- equality
 - numeric, 205
- event
 - asynchronous, 232
 - demux, 232
- expect
 - function, 76
- export
 - directive, 193
- false, 71
- fcomplex, 200
- fdoc, 101
- finaliser
 - clause, 186
- floating insertion
 - named, 190
 - polymorphic, 191
 - textbf, 190
- floating point
 - type, 200
- flx_main, 98
- for loop, 47
- function
 - arity, 77
 - binding, 188
 - definition, 75
 - higher order, 77
 - postcondition, 76
 - precondition, 76
 - result, 76
 - type, 76
 - when, 76
- generator, 20
 - yielding, 79
- goto, 38
- grouping
 - parentheses, 72
- header
 - floating insertion, 190
- identifier
 - arbitrary, 103
 - grammar, 102
 - operator, 103
 - TeX, 103
- include
 - directive, 84
 - grammar, 101
- insertion
 - floating, 190
- integer
 - literal, 104
 - operator, 199
 - type, 196
 - typeset, 200
- iterator
 - slice, 210
- label, 38
- layout
 - grammar, 101
- lcomplex, 200
- length
 - string, 212
- let fun, 57
- library
 - statement, 124
- likely, 66

- literal
 - bool, [104](#)
 - grammar, [96](#)
 - integer, [104](#)
 - string, [105](#)
- lookup
 - name, [119](#)
 - qualified
 - function, [123](#)
 - non-function, [122](#)
 - setwise, [130](#)
 - shadow, [121](#)
 - signature, [120](#), [121](#)
 - suffixed name, [120](#)
 - unqualified
 - function, [121](#)
 - unqualified name, [119](#)
- macros
 - grammar, [94](#)
- matching
 - exact, [119](#)
- name
 - indexed, [114](#)
 - qualified, [69](#), [115](#), [129](#)
 - simple, [114](#)
 - suffixed, [66](#), [115](#)
 - unqualified, [114](#)
 - unsuffixed, [115](#)
- non-terminal
 - grammar, [93](#)
- object
 - dynamic, [81](#)
 - variable as, [27](#)
- open
 - directive, [117](#), [119](#)
- open syntax
 - grammar, [97](#)
- ordering
 - total, [206](#)
- overload
 - candidate, [117](#)
 - resolution, [117](#)
 - signature, [118](#)
 - specialisation, [118](#)
 - type constraint, [118](#)
 - view, [118](#)
- package
 - requirement, [191](#)
- parentheses
 - grouping, [72](#)
- pattern
 - matching, [73](#)
 - variable, [68](#)
 - wildcard, [71](#)
- pattern matching, [42](#)
- pointer
 - C, [68](#)
- pointers, [165](#)
- polymorphism
 - parametric, [131](#)
 - row, [73](#)
- postcondition
 - function, [76](#)
- precondition
 - function, [76](#)
- priority
 - grammar, [92](#)
- procedure
 - binding, [189](#)
 - definition, [77](#)
 - unit, [72](#)
- procedure call, [43](#)
- quaternion
 - type, [200](#)
- record
 - expression, [70](#)
 - matching, [74](#)
- redo, [48](#)
- regdef
 - grammar, [95](#)
- rename
 - directive, [119](#)
- requirement
 - package, [191](#)
 - property
 - textbf, [191](#)

- requires
 - clause, [190](#)
 - grammar, [91](#)
- root, [115](#)
- scanner
 - clause, [186](#)
- scheme
 - action, [88](#)
- sequencing, [71](#)
- serialisation, [219](#)
 - decode, [219](#)
 - decoder, [187](#)
 - encode, [219](#)
 - encoder, [187](#)
- shadow
 - symbol table, [117](#)
- side-effect
 - function, [75](#)
- slice
 - addition, [211](#)
 - generalised, [211](#)
 - iterator, [210](#)
 - string, [210](#), [211](#)
 - type, [209](#)
- string
 - append, [212](#)
 - equality, [212](#)
 - escape, [106](#)
 - length, [212](#)
 - literal, [105](#)
 - map, [214](#)
 - slice, [210](#)
 - stl, [214](#)
 - type, [212](#)
- struct
 - binding, [189](#)
- subscript, [66](#)
- substring, [67](#)
- symbol
 - view, [116](#)
- symbol table
 - lookup, [116](#)
 - primary, [117](#)
 - private, [116](#)
 - public, [116](#)
 - shadow, [117](#)
- table
 - symbol definition, [115](#)
- throw, [46](#)
- trap, [46](#)
- true, [71](#)
- type
 - _gc_pointer, [185](#)
 - _gc_type, [185](#)
 - abstract, [126](#)
 - atomic, [237](#)
 - function, [76](#)
 - incomplete, [185](#)
 - pod, [184](#)
 - statement, [184](#)
 - uncopyable, [185](#)
- typedef, [141](#)
- uncomment, [101](#)
- Unicode, [100](#)
- until loop, [48](#)
- UTF-8, [100](#)
- utf8, [100](#)
- varargs, [71](#)
- variable, [24](#)
- view
 - symbol, [116](#)
- while loop, [48](#)
- with
 - clause, [117](#)
- yield, [20](#), [45](#), [79](#)