felix

Release 2018.09.20

Aug 01, 2020

Contents

CHAPTER 1

Core System

1.1 Language Reference Manual

This is the Felix Language Reference manual, it is intended primarily to document the common language interface presented to the programmer. It is not complete or precise because the grammar and features the user would normally call a language are actually defined in user space, in the library.

This chapter briefly explains some of the central concepts of Felix.

1.1.1 Simplicity, Performance.

Felix is, first and foremost, dedicated to obtaining run time performance.

Felix motto is *hyperlight* performance which means we aim to run programs *faster than C*.

Let's start with a simple script:

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

do
  val n = 13;
  var v = ack(3,n);
  println$ f"Ack(3,%d): %d" (n, v);
done

fun Tak (x:double, y:double, z:double): double =>
  if (y >= x) then z
  else Tak(Tak(x - 1.0,y,z), Tak(y - 1.0,z,x), Tak(z - 1.0,x,y))
  endif
```

(continues on next page)

(continued from previous page)

```
;

do
  val n = 12.0;
  var v = Tak(n*3.0, n*2.0, n*1.0);
  println$ f"%.2f" v;
done
```

To run it you just say:

```
flx test.flx
```

It's pretty simple. Felix runs programs like Python does, you run the source code directly.

All the generated files are cached in the .felix/cache subdirectory of your $HOME directory. Felix can run script files in read-only directories.

Felix translates the code into C++, compiles the C++, and runs it. Felix programs run *fast*. Felix itself implements high level optimisations beyond the scope of traditional compilers, then passes the generated code to your system C++ compiler which in turn implements low level optimisations.

Here's a silly comparison for Ackermann's function and Takfp, times in seconds:

Compiler	Ack	Takfp
Felix/clang	3.71	6.23
Clang/C++	3.95	6.29
Felix/gcc	2.34	6.60
Gcc/C++	2.25	6.25
Ocaml	2.93	8.41

1.2 Overview

Contents:

1.2.1 Platform Model

Felix define four significant platforms:

- build platform: where Felix itself is built
- host platform: where you edit Felix code and translate to C++
- target platform: where you compile the C++
- run platform: where you run the compiled binaries

Build platform

If you download the Felix system sources and build Felix yourself, your host platform is the build platform. However if you download tarballs of prebuilt binaries, or use a package manager to fetch them, then another computer was used as the build platform.

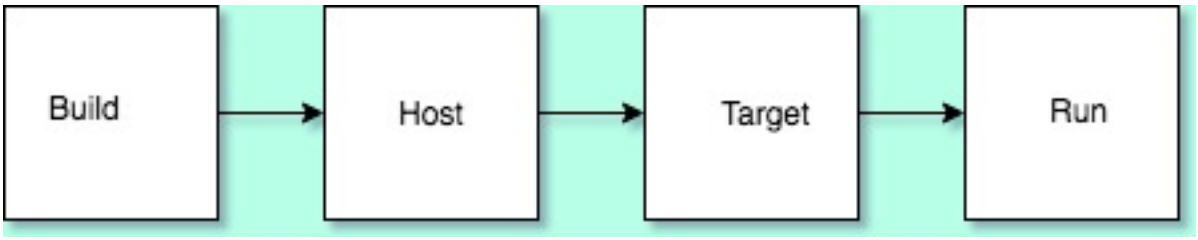

Fig. 1: platform model

Host platform

The system you program on is the host platform. A platform is not just a computer, rather it is a machine together with an operating system and working environment. For example you can actually run Ubuntu on Windows 10 Summer Edition, or you can run Cygwin. On OSX you can run the system clang provided with Xcode, but you can also use *brew* to fetch a more recent clang or even gcc.

Target platform

The target platform is where you run the C++ compiler. If you're on Windows using Cygwin you will probably be running a Posix hosted cross compiler targetting Windows. however you can also invoke Visual Studio's C++ compiler from Cygwin.

Run platform

The run platform is where the binaries get run. You can cross compile from Cygwin to produce binaries designed to run natively on Windows (for C code anyhow!).

The model above is not a complete or accurate picture of reality, however it is a reasonable approximation which is used as a base for achieving your programming goals with good reusability.

1.2.2 Installation Structure

Files in the installation belong to a particular platform.

Build Files

These are shared source code, including build scripts, Ocaml source for the compiler, C++ sources for the run time library, Felix sources for the Felix standard libraries, standard database configuration data, and documentation.

This set of files is version dependent, but platform independent.

Host Files

The host system requires the build files together with the built executables flxg (the compiler), flx (the compiler driver), the toolchain plugins, flx_pkgconfig (the database inspector), and other build tools.

It also includes the configured database, and the control files in $HOME/.felix.

Target Files

The target files include the C++ run time library objects files, static archive libraries, and C++ RTL ++ header files, and all the data requires to compile C++ code to object files and link it to executables or shared libraries.

Run Files

The run files consist of the build executables and shared libraries required to run the user code on the run platform.

Installation Structure

The current installation structure defaults to use of two directories.

Installation Directories

The root install directory is /usr/local/lib/felix. Each version of Felix is a separate subdirectory of the root install directory.

Each install directory contains at least two subdirectories. The share subdirectory contains the build files.

The host subdirectory contains the host files, and the target files for the host platform as well.

Additional target subdirectories can be created. These must contain at least the target files for the selected target platform. At present the build tools also attempt to add host files to the target as well.

Control Directory

There is one control directory per user account, which is the subdirectory .felix contains in the users HOME directory.

This directory has several subdirectories.

The config subdirectory of the control directory contains overrides for the standard configuration which are applied when initially configuring, or later reconfiguring, Felix. They apply to the user host only.

It also contains the control file felix.fpc which is the master control file used to locate other files in non-default locations.

The cache subdirectory has two subdirectories. The text subdirectory containes generates source files, including C++ files. The binary subdirectory contains build object files, libraries, and other binary data for the target and run platforms.

1.2.3 Platform Dependency Model

Felix uses a specific model for handling platform dependencies.

We will show how it works using the system shell as an example.

First, we define a type class parametrised by phantom type variables representing the dependencies:

```
class Shell_class[OS, process_status_t]
{
  // Quote a single argument.
  virtual fun quote_arg:string->string;
```

(continues on next page)

(continued from previous page)

```
  // quote a list of arguments
  fun quote_args (s:list[string]) : string => catmap[string] ' ' quote_arg s;

  ...
}
```

The *Shell_class* is parametrised by two type variables, *OS* and *process_status_t*. The virtual function *quote_arg* must be overriden in an instance. This class can be used with any platform host to quote arguments for any platform target by setting the parameters.

Now, we define a platform specific class for Posix systems:

```
class Bash {

  instance Shell_class[Posix, PosixProcess::process_status_t] {
    fun quote_arg(s:string):string= {
      var r = "";
      for ch in s do
        if ch in "\\\"" do   // leave $ and ` in there, unquoted.
          r += "\\"+ str ch;
        else
          r+= ch;
        done
      done
      return '"'+r+'"';
    }
  }
  fun bash_specific_thing ...

  inherit Shell_class[Posix, PosixProcess::process_status_t];
}
```

To define quoting for posix *bash* shell, we first specify an instance for the abstract shell class *Shell_class* specifying the *OS* parameter as type *Posix*, and the *process_status_t* parameters as *PosixProcess::process_status_t*, defining an override of the virtual *quote_arg*.

We can also add some bash specific functions now.

Then we inherit *Shell_class* into *Bash*, specifying the *OS* parameter as type *Posix*, and the *process_status_t* parameters as *PosixProcess::process_status_t*, which also pulls in *quote_args* function.

This gives us a complete set of operations on *Bash* including platform independent ones and platform dependent ones.

We do the same for Windows:

```
class CmdExe
{
  instance Shell_class[Win32, Win32Process::process_status_t]
  {
    fun quote_arg(s:string):string => '"' + s + '"';
  }
  fun cmdexe_specific_thing ...

  inherit Shell_class[Win32, Win32Process::process_status_t];
}
```

Finally, we use conditional compilation to define

```
class Shell {
  if PLAT_WIN32 do
    inherit CmdExe;
  else
    inherit Bash;
  done
}
```

The symbol *PLAT_WIN32* is a macro, set to true if the host OS is Windows.

What have we achieved?

- You can write code for the current host system using class *Shell* The generated C++ will only work on the current host. If your Felix code *only* uses platform independent features, it will work on other platforms too, but it has to be recompiled by the Felix compiler and will generate different C++.

  If your Felix code uses platform specific features it may fail to compile with Felix compiler because it depends on functions not included by conditional compilation.

- You can write code for Windows on *any* platform using class *CmdExe*. This Felix code is platform independent, it will Felix compile on all platforms. The generated C++ may only compile on Windows and is only useful, whether it compiles or not, on Windows for calling the Windows shell.

- You can write code for Posix on *any* platform using class *Bash*. This Felix code is platform independent, it will Felix compile on all platforms. The generated C++ may only compile on Posix, and it is only useful, whether it compiles or not, on Posix platforms for calling Bash shell.

- You can write platform independent Felix code that is parametrised by the OS and process status types, which extends the base abstraction, without conditional compilation. The extension can then follow the same platform model.

As per the above platform model there are three ways to write code that works on multiple platforms.

Platform Independent Code

First and foremost, you can write code in Felix that generates the same C++ on all platforms, and which works the same way on all platforms. There is an adaption layer which translates the C++ for the platform, usually included in a combination of

- The Felix run time library (RTL),
- the C++ compiler's standard library and
- the platform OS system C library.

This kind of Felix code is said to be *platform independent*.

Platform Adaptive Code

When you write code using the *Shell* class, using only Felix functions in common to all platforms, the code is said to be *platform adaptive*. The interface is the same on both Windows and Posix but the function definitions are not. Here Felix itself uses conditional compilation of Felix code to achieve interoperability. Consequently the generated C++ will vary, depending on the host platform.

Platform Dependent Code

Using platform specific classes such as *Bash* or *CmdExe* you can write platform dependent code for a *specific* target platform independently of your current host platform.

Platform Parametric Code

And finally using the parametrised *Shell_class* you can write code which depends on platform specific features and *defer* deciding how to implement or represent those features or *abstract away* the problem, by using virtual functions. We can call this *platform parametric* code.

Use Cases

The most common way to use Felix is to write code for your own system. In this case, platform specific code is good enough. However if you want to share the code with other Felix programmers you need to write platform adaptive code instead.

However, if you want to write code for non-programmer clients, with a variety of platforms, you need to write platform independent code. Failing that, you need to generate, from platform adaptive code, multiple versions of the C++ code and arrange to compile that code on these platforms, for example by using continuous integration servers such as Travis (for Linux), or Appveyor (for Windows).

Configuration Databse

Felix provides another mechanism to handle platform dependencies.

Linker switches

Even on the same Linux OS, libraries you need to link to can be in various places. System libraries on Debian platforms live in /usr/lib wheres if you build libraries yourself they usually end up in /usr/local/lib. OSX linkers use frameworks whereas Linux does not. Similarly header files can live in various places.

To meet the vagaries of compilation and linkage requirements, Felix provides in-language clauses for type and function bindings to specify libraries required in the abstract, namely a *requires package* clause. The specified package names are mapped to files ending in extension *.fpc* in a configuration database, and those files contain local specification of how to find and link the libraries.

Include Files

C/C++ header files are handled as well.

For this, Felix compiler outputs a single *#include "progname.includes"* directive, and generates a file *progname.resh* which contains a list of the required packages. An external tool, *flx_pkgconfig* queries the configuration database and generates *progname.includes* using the supplied information. The *flx* tool does this automatically, and also uses the *resh* file to organise compiler and linker switches.

Compiler Toolchain Drivers

In addition, the *flx* tool uses plugins to drive your C++ compiler. Each compiler is driven by a distinct plugin module which understands how to translate abstract compilation and linkage requirements into specific command lines for that compiler and OS. Standard toolchains are provided for *gcc* and *clang* on Linux and generic unix platforms and for

OSX, and for Visual Studion 2015's *cl.exe* on Windows. There is also a driver for the iPhone emulator and iPhone for iOS applications.

Compiler toolchain drivers must all provide ways to perform the following abstract tasks:

- compile a C++ translation unit for static linkage to an object file
- compile a C++ translation unit for dynamic linkage to an object file
- combine static link objects into a searchable library
- combine dynamic link objects into a searchable library
- link static link objects and libraries to form a standalone executable
- link dynamic link objects to form a shared dynamic link library
- static link a thunk which can invoke a shared library as a program

In general, linkers can link some code static and other code dynamic in arbitrary combinations. This is too hard to generalise so Felix only supports two models: static and dynamic link. Static link model links everything with a flat namespace into a standalone executable program.

Note that despite this on most platforms some libraries are dynamic linked anyhow: the C library interfacing to the OS is usually dynamically linked at load time on all platforms. Both OSX and Windows usually dynamic link system level APIs. However system dynamic linkage is usually transparent in the sense that the libraries are already provided and do not have to be built, and the mode of linkage is handled automatically by the linker.

Note also that, primarily to support archaic linkage models used on Linux, Felix distinguishes object files desiged for static linkage, and those designed for dynamic linkage: the latter requires -fPIC position independent code, the former is a legacy model which executes slightly faster.

By default Felix adds the suffix *_static* to object file basenames designed for static linkage, and *_dynamic* to object file basenames for dynamic linkage, and similarly for searchable libraries built from them. The reason is that unfortunately archaic linkage technology used on unix platforms can accidentally link the wrong kind of library leading to inexplicable run time crashes.

1.2.4 Compilation Model

The Felix translator takes several sets of files and translates them into a set of C++ files, compiles them, and links the results into a library.

Source File kinds

Grammar Files

The Felix grammar is defined in user space in the library. Certain files, with extension ".fsyn", typically found in the grammar subdirectory of share/lib subdirectory, are processed primarily for grammar definitions: any generated code in these files is simply throw out.

The files used as the base grammar are determined by scanning, environment variables and switches, which can be used to control the syntax of the base system.

Grammar specifications are first class parts of the language, any user file may contain them, however they cannot be exported in a modular fashion at this time so the compilation control machinery is required to share the user grammar, and such sharing is global.

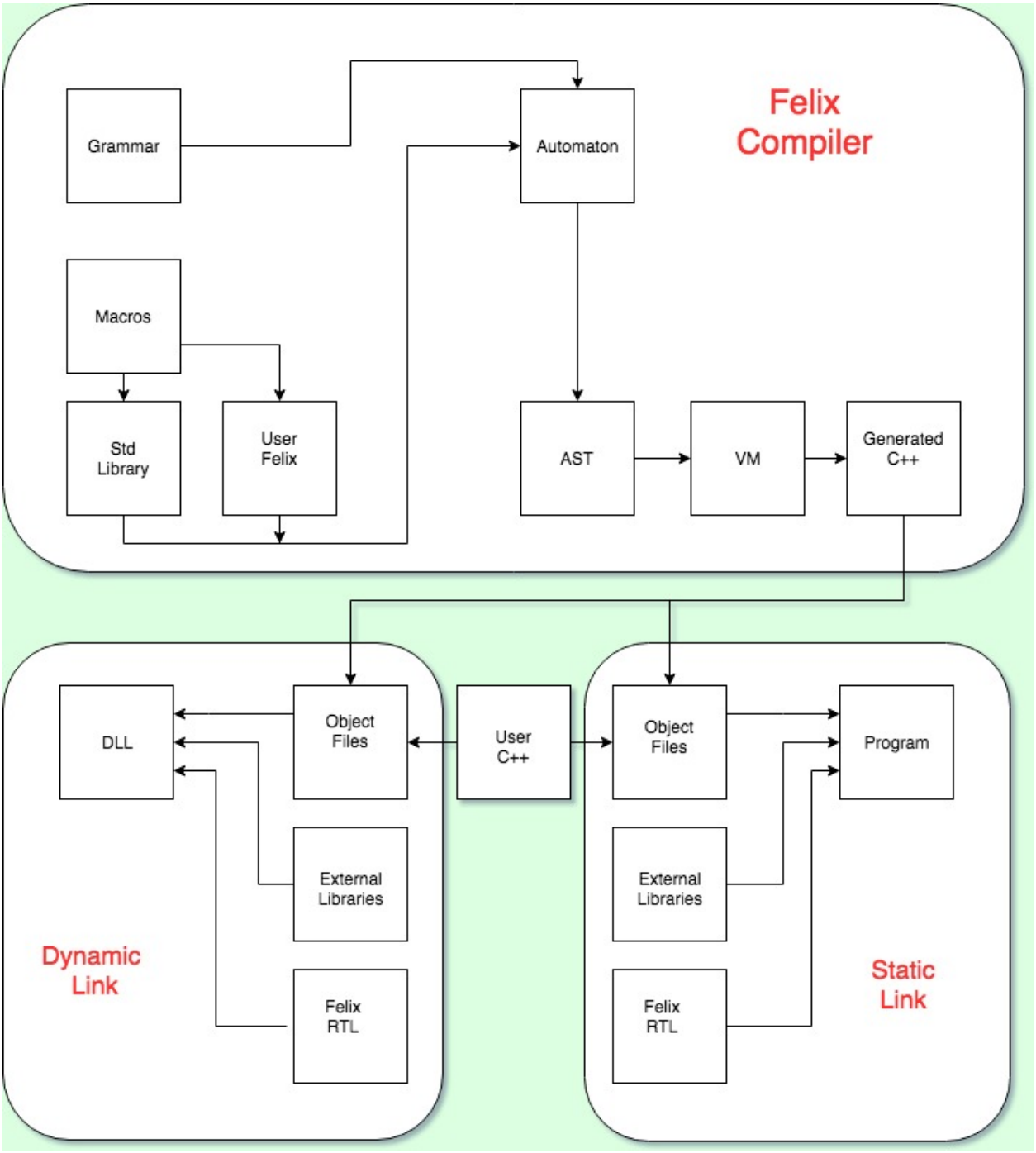

Fig. 2: Compilation Model

Macro Files

A small number of files, usually a single file, is used to define macros which are prefixed to every ordinary file Felix processes. These macros are typically used to specify the host operating systems so as to allow platform dependent conditional compilation. Macro files usually end in extension **.flxh*.

Library Files

These are files provided by the standard Felix install which contain a mix of several kinds of entity. First and foremost, they contain commonly used data types and functions operating on them, such as lists, arrays, strings, regular expressions, etc.

Secondly, the Felix run time library (RTL) is written in C++ and the standard library contains bindings to this library. For example the Felix garbage collector is a C++ class type which is implemented in the RTL and there is a Felix binding to it in the standard library which allows low level access to some of its facilities.

Thirdly, some special operations are encoded with special techniques in the library. For example, the Felix compiler generates, for most heap allocated types, a special Run Time Type Information (RTTI) record. The RTTI includes information locating where in the type pointers are located so that the garbage collector can trace them. The standard library provides the user access to these records to allow some degree of introspection. In particular the RTTI provides encoders and decoders for serialisable types which allows the generic serialisation functions defined in the RTL to operate.

User Files

User code files are written by the programmer and typically consist of either library definitions or program code.

Inclusion model

Unlike C or C++, Felix does not provide a way to include text files in others. Text file inclusion is restricted to macro files which are specified on the command line.

Instead, Felix is designed so each user and library file can be separately parsed, independently of any other such file, and it can therefore cache the results of a parse. The compiler uses dependency checking to decide if a the cache needs to be update when the file is changed. In particular, user and library files depend only on macro files and the base grammar files. Since these rarely change, in general the standard library only needs to be parsed once, and user program files only need to be reparsed if they're modified.

This provide significant performance advantage share by almost all modern languages but lacking in C and C++ where parsing files and all their dependent include files repeatedly for every translation unit is a serious problem in large scale software developments.

See also *Include Directive*.

Include Directive

In Felix and include directive may be given by specifying a Felix file name:

```
include "std/datatype/list";
```

The file name excludes the extension of the file, and must use Unix slash separator, even on Windows. The effect of this directive is as follows: Felix first parses the current file, generating an abstract syntax tree (AST), and gathers the names of the included files at the same time.

Then, it examines the cache to see if it can find the generated AST for the include files. If not, or, if the cached AST is out of data, it parses the included file recursively, updating the cache.

The resulting AST for each include file is then prefixed to the generated AST for the including file. An AST is not included more than once.

Therefore, when compiling a program file the included files end up at the top of the program file. This requires that any executable code in any included file must be able to operate correctly independenly of any other included file.

Order of Initialisation

The only ordering guarantee offered is that the main program file will be executed after any variable initialisations in included files. For example, Felix provides functions to create an alarm clock, and it provides a default alarm clock. The alarm clock will be initialised (if used) before the main program starts executing.

Elision of Unused Variables

Felix guarrantees to elide unused variables other than unused functional parameters. The utility of this assurance can be seen in the following example:

Felix provides a global variable containing a thread pool object. Since this object is in the standard library and will be included directly or indirectly by the mainline program, the thread pool is automatically available if it is used. However construction of thread pools is expensive so, if the global thread pool variable is not used, it is elided. In particular, its initialiser is elided too, and so if the pool is not used, no threads are constructed.

The rule applies to all variables and may sometime lead to surprises. In particular a common mistake is to write a generator with a side-effect and store the result in a variable, assuming the side-effect will occur. If the variable is not used, however, it will be elided and so too the generator application, so the side effect will be lost.

Insertion Model

Felix uses floating insertions to include C++ source dependencies into generated C++ code. There are two kinds of insertion, header insertions which go near the top of the generated header (hpp) file, and body insertions which go near the top of the generated body (cpp) file.

Typically header insertions define type and function interfaces whilst body file provide function definitions.

Literal insertion phrases are illustrated;

```
header '#include "myfile.hpp"'
body 'void f() { cout << "hello"; }'
```

Insertions can be tagged:

```
header cstring_h = "#include <cstring>";
```

Insertions can be used as dependencies of primitive bindings: type bindings, function and procedure bindings, and tagged insertions, expressed by requires clauses:

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

A floating insertion tag definition may use the same tag as another, in this case all the insertion texts will emitted if the tag is required.

Insertion dependencies can be recursive:

```
header one = "void f()"
  requires two
;
header two = "void g()"
  requires one
;
type X = "X" requires one;
```

The code generator finds all type and function bindings used in the final generated code and then finds the transitive closure of the set of required floating insertions. Then it emits the floating insertions in an order compatible with the order of writing. Duplicates are elided based on the actual text of the insertions.

Polymorphic insertions

Tagged floating insertions can be polymorphic. In this case the requirement must suffix the tag name with type arguments:

```
proc rev[T,PLT=&list[T]] : &list[T] = "_rev($1,(?1*)0);" requires _iprev_[T,PLT];

body _iprev_[T,PLT]=
  """
  static void _rev(?2 plt, ?1*) // second arg is a dummy
  { // in place reversal
    //struct node_t { ?1 elt; void *tail; };
    struct node_t { void *tail; ?1 elt; };
    void *nutail = 0;
    void *cur = *plt;
    while(cur)
    {
      void *oldtail = ((node_t*)FLX_VNP(cur))->tail;   // save old tail in temp
      ((node_t*)FLX_VNP(cur))->tail = nutail;          // overwrite current node tail
      nutail = cur;                                    // set new tail to current
      cur = oldtail;                                   // set current to saved old tail
    }
    *plt = nutail;                                     // overwrite
  }
  """
;
```

Resource Database

Felix uses a resource data base to represent some external libraries. The database consists of one or more directories containing resource descriptors which are files ending in extenion *.fpc*. The system is similar to *pkconfig* and there is a separate tool *flx_pkgconfig* which can be used to query it.

The resources are called resource packages, not to be confused with source packages.

Package files consist of a sequence of field definitions:

```
Generated_from: 2403 "/Users/skaller/felix/src/packages/gc.fdoc"
Name: flx_gc
Platform: Unix
Description: Felix default garbage collector (Unix)
```

(continues on next page)

(continued from previous page)

```
provides_dlib: -lflx_gc_dynamic
provides_slib: -lflx_gc_static
includes: '"flx_gc.hpp"'
library: flx_gc
macros: BUILD_FLX_GC
Requires: judy flx_exceptions
srcdir: src/gc
src: .*\.cpp
```

The field *includes* specifies the header requires to compile with the Felix garbage collector.

The *provides_dlib* field specifies the linker switches required to link the shared library version of the collector.

The *provides_slib* field specifies the linker switches require to link the static archive version of the collector.

The *Requires* field specifies packages on which this one depends, in this case *judy* and *flx_exceptions*.

Packages can contain arbitrary fields: in the above package there are fields which are used to control building the run time library.

The configuration database must be created to reflect the location of libraries and header files for each individual system.

In order to access the database the programmer uses a requires package clause:

```
type collector_t = "::flx::gc::generic::collector_t*"
  requires package "flx_gc"
;
```

although note this is only an example and the collector is actually always available.

Here is another package:

```
Generated_from: 3674 "/Users/skaller/felix/src/packages/sdl.fdoc"
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
requires_dlibs: ---framework=OpenGL
requires_slibs: ---framework=OpenGL
```

In this case some special coding is needed to emit the correct linker switches: on OSX the syntax is two words:

```
--framework OpenGL
```

and the extra leading - and internal = have to be removed to emit the correct switches. *flx_pkgconfig* can remove duplicate fields and this could lead to an incorrect isolated framework name if the *–framework* is not duplicated.

The primary effect of the resource packaging system is to abstract away the system dependent details of the location and name of library files, and then allow the programmer to express these dependencies via the abstraction directly in the program.

As a result, Felix can automatically find external headers during C++ compilation, and automatically find libraries during linkage, removing the need for external scripts such as Make files entirely.

Provided you install the required libraries for the Simple Direct Media Layer (SDL) system, for example, and then install suitable *.fpc* files in the configuration database, then Felix can magically run programs which do graphics, and you can write code which works on all platforms supporting SDL.

Note that *flx* tool *also* supports automatic linkage of C and C++ code provided suitable annotations are embedded in the code (however it doesn't support automatic insertion of header files because that would prevent the C++ program from being compiled with conventional tools).

Output Model

By default, Felix generates a shared library which can be run with a fixed loaded program passed the library name as an argument. It is also possible to produce a static link object file, and link the stub loader with the generated library to create a stand alone executable.

Felix does not support mixed mode linkage. You either use all shared libraries or fully statically link everything. The only exception is if the system requires dynamic linkage of certain libraries, for example on OSX the C run time library is always dynamically loaded (even for statically linked executables).

This document does not describe all the capabilities of the *flx* driver tool, please read the tools documentation for that. Suffice it to say the tool can also compile and link in C++ to a Felix program, and, it provides comprehensive caching and dependency checking of all compilation and linkage steps.

By default all outputs go into the cache, even the final executable, and the program is then run, emulating the operation of a scripting language such as Python. It can therefore be regarded as an interpreter which takes a long time to start the first time, but runs code immediately thereafter, and runs it faster than any interpreter could (even one with a Jit).

1.2.5 Execution Model

Felix has a novel and sophisticated execution model. There are two novel features.

Inderminate Evaluation Strategy

Whilst most languages specify eager evaluation, meaning function arguments are evaluated before calling a function, and a few, such as Haskell, specify lazy evaluation, meaning the function is called immediately and the arguments are evaluated inside the function on demand, Felix uses indeterminate evaluation by default: that is; it allows the compiler to choose the evaluation strategy.

Indeterminate evaluation is one reason why Felix is so fast.

When a function is inlined, the parameter can be replaced in each case by the argument expression. This can be faster because further optimisations are possible.

On the other hand if the argument is used many times, it may be better to evaluate it just once. Also, when a closure of the function is formed, it is hard to substitute an as yet unknown argument into the closure, so Felix uses eager evaluation for closure arguments.

Because it is able to chose the fastest strategy, Felix generates extremely fast code. In practice the semantics are the same which every strategy is used for most functions, even if the function is not total/strict, because it is used with arguments for which the behaviour is the same anyhow. For example division is not total because the result is not defined if you divide by zero, but the result is the same for all other values.

Felix provides several ways to modify the default behavious. If a function parameter is marked *var* then even if the function is inlined, it will evaluate the argument and assign it to the parameter *provided the parameter is actually used*; that is; the function will behave as if eagerly evaluated.

On the other hand if you want lazy behaviour you can make your function accept a closure end evaluate it when you need the result.

Fibres

Felix supports shared memory concurrency with conventional pre-emptive threads. However within each thread a collection of fibres may run.

Fibres are logical threads of control which communicate with other fibres using synchronous channels, as well as shared memory. Fibres interleave execution in a pthread under program control: I/O operations on channels cause the current fibre to be suspended until at least a matching opposite operation of another fibre. Reads match writes, so a reader will suspend until a write provides the data it is waiting for, writers suspend until there is a reader to consume the data it provides.

In a fibrated system all events form a total order: fibration is a *sequential programming technique* in which pieces of the program suspend of their own volition, and are resumed by a scheduler at the earliest when their I/O requirement is met.

Fibres communicating with anonymous channels cannot deadlock, deadlock is a legitimate method of suicide. When a fibres starves for input, or blocks for output, which cannot arrive, the fibre is removed (by the garbage collector) and so a deadlock is equivalent to the fibre exiting.

The details are beyond the scope of this brief description and can be found elsewhere.

Asynchronous Event Handling

Felix has an asynchronous event handling system which currently supports two kinds of events: timeout of an alarm clock, and socket readiness notifications.

When utilised a separate system pthread monitors timers and sockets using the best available technology for the current platform: it chooses between select, poll, epoll, kqueue, Windows completion ports and Solaris completion ports.

When a coroutine waits for a clock alarm, or requests a socket read, write, or connection, its fibre is blocked until the alarm triggers, or the requestion socket I/O operation is complete. However *other* coroutines running on the same pthread do not block.

Garbage Collection and Threads

Felix runs with a garbage collector. Use of the collector is "optional" in a certain sense. The collector is a naive world-stop mark/sweep variety, with a parallel mark algorithm.

The collector can be called manually, or is triggered by an attempt to allocate memory when a threshold is reached. Felix uses an allocator which itself calls the C library *malloc*. The collector is precise with object allocated on the Felix heap, tolerates but cannot scan objects allocated on the C heap, and conservative on the machine stack.

When a collection is triggered it must wait until all threads stop. Threads will stop only when they attempt an allocation or explicitly check for the GC pending flag. Because of this Felix provides its own versions of synchronisation primitives such as mutex and semaphores. Raw system primitives can only be used with knowledge of the memory and execution model. For example a raw mutex is fine provided the scope of the lock does no allocations and completes in bounded time. The Felix mutex is actually a spin lock that checks the GC flag whilst spinning (and also inserts an small OS based sleep in the spin loop to encourage the holder of the lock to run and complete.

Closures

It is important to note variables *including* ‘val‘s are part of a stack frame object which may be allocated on the heap and the capture is via a pointer to the whole frame. This means when a closure is executed, the value of the captured variable is the value current *at the time the closure executes* and not the value at the time of capture.

For example:

```
var x = 1;
var g: (1->0)^3;
noinline proc f (y:int) () { println$ y; };
for i in 1..3 do
  &g.(i - 1) <- f x; // capture value x in y
  ++x;
done
for i in 1..3 do
  g.(i - 1) (); // prints 1,2,3
done
```

Without the *noinline* Felix is too smart and the variable y is inlined to the mainline, so there is only one copy, when you run the script, it prints 4,4,4. If you just change the parameter to *var y:int* to force eager evaluation, it prints 3,3,3.

Capture by address is not a design fault, it is in fact the only option. Just consider:

```
var x = 1;
fun getx() => x;
++x;
println$ getx();
```

You would be surprised if this printed 1! You expect the function to report the current value of x.

1.2.6 Parsing

Overview

Unlike most programming languages, Felix has tiny bootstrap grammar hard coded into the compiler. That grammar does not specify the commonly used part of the language, instead, it specifies an how to parse an EBNF like syntax which specifies the actual grammar.

The parser starts by reading these EBNF specifications which are part of the standard library in files ending in the extension *.fsyn*. It translates these specifications into an internal data structure. The specifications look like this:

```
syntax blocks
{
  stmt = block;
  block := "do" stmt* "done" =># '`(ast_seq ,_sr ,_2)';
  block := "begin" stmt* "end" =># '(block _2)';
  block := "perform" stmt =># '_2';
}
```

The basic form of a grammar rule consists of:

- a nonterminal on the left
- an *:=* symbol,
- a production, which is a regular expression like specification involving

** quoted strings, which are treated as terminals, ** identifiers, which are nonterminals, ** and the symbols * for the Kleene closure of 0 or more repetitions, ** + for 1 or more repetitions, ** *?* for 0 or 1 repetition, ** and parenthesis *(* and *)* for grouping. * followed by a *=>#* symbol, then * a string in R5RS Scheme call the action * and finally a terminating semicolon *;*.

The *syntax* group is called a DSSL or domain specific sublanguage.

Then the parser hits a statement like this:

```
open syntax blocks;
```

and this is where the fun starts. The parser takes the DSSL specified and *augments* the current parser with the specified syntax. After all the extensions so specified a special command causes the parser to save the extended automaton to disk as a cache: dependency checking allows us to skip the above process if the grammar is unmodified and load :ethe automaton immediately.

The parser is now ready to process actual Felix files. When it parses a string according to a grammar rule it sets the Scheme variables *_1*, *_2*, *_3* etc to the abstracted syntax trees previously generated and then executes the Scheme in the action code. This results in a Scheme S-expression being returned. Note that the Scheme code is compiled and the compiled code saved, so that the code is not interpreted each time.

After the whole file is parsed, the resuling Scheme S-expression is translated to a simpler kind of S-expression which is easier to pattern match against in Ocaml, and then that S-expression is translated into the first stage syntax tree using Ocaml variants which are required for the compiler. Subsequent processing stages macro process and desugar the syntax tree to another form.

The result of parsing each file is saved in the cache so if the file and the grammar is not changed, the AST can be loaded from the cache. Therefore Felix usually only parse the files you're actively working on. Felix is specifically designed so that files can be parsed independently of eaach other to minnimise parsing time. The cached files AST files end in extension *.par*.

The parser used is Dypgen, and the Scheme used is OCS Scheme. Dypgen is a GLR+ parser. The automaton has an LALR1 kernel and it resolves shift/reduce and reduce/reduce conflicts by following both alternatives simultaneously; each path is called a parser thread. Later, a parser thread may fail reducing the number of threads being executed, and the parser can also merge threads if they produced the same result a different way. The parser also has a number of other features including the ability of the action code to kill a thread by throwing an exception.

Syntax specifications are part of the Felix language. They can be used anywhere in Felix files, and they even respect scopes. However *only* files processed in the initial grammar processing phase can be shared at this time.

1.3 Lexical Structure

1.3.1 Overall Lexical Structure

Felix supports 3 kinds of source files.

flx files

These files have **.flx* extension. They are the primary kind of code only file.

Felix is a free form language meaning end of lines are generally not significant. White space can be freely inserted between significant lexemes, and sometimes it is required to separate them correctly.

There are two forms of lexical comments. Felix allows the usual C++ style *// to eol* comments and C style */* to */* comments, except that C style comments can be nested.

Continuation lines are not supported.

Felix does not have any keywords. Instead in certain contexts, certain identifiers are treated as keywords. The same words are ordinary identifiers in other contexts.

The Felix grammar is specified in the library, so it can be easily extended by the programmer. When adding new grammar, avoid misusing symbols such as *;* or *,*, as this may either fail or make the code unreadable.

Identifiers, integer literals, floating literals, and string literals, are also defined in the library in the grammar.

fdoc files

A second form of file uses the **.fdoc* extension. These files are used for test cases.

Fdoc files start with commentary, in *prose* mode. You can write any text, as if in a comment. Special lines are allowed to support typesetting and interpretation, switching to *code* mode, and specifying the output in *expect* mode:

```
@title Name of the Test
@h1 A heading
@h2 A subheading
Now we will check some things.
@felix
fun f() => 1;
println$ f();
@
@doc
This should print just 1.
@expect
1
@
```

The *@felix* line switches to code mode, and any line starting with @ ends code mode. The *@expect* line allows the expected output to be provided.

This form of code can be run exactly like ordinary **.flx* files.

However, the *flx* tool can also run and check a whole test suite from a single command line.

fdoc packages

There is an advanced form of **.fdoc* files which provide a set of code files. The files have to be extracted with a special tool, *flx_iscr*. Most of the Felix system is stored on GitHub as packages. Packages allow collections or related files to be put together along with documentation.

1.3.2 Identifiers

Syntax

```
syntax felix_ident_lexer {
  /* identifiers */
  regdef ucn =
      "\u" hexdigit hexdigit hexdigit hexdigit
    | "\U" hexdigit hexdigit hexdigit hexdigit hexdigit hexdigit hexdigit hexdigit;

  regdef prime = "'";
  regdef dash = '-';
  regdef idletter = letter | underscore | hichar | ucn;
  regdef alphnum = idletter | digit;
  regdef innerglyph = idletter | digit | dash;
  regdef flx_ident = idletter (innerglyph ? (alphnum | prime) +)* prime* ;
  regdef tex_ident = slosh letter+;
  regdef sym_ident =
    "+" | "-" | "*" | "/" | "%" | "^" | "~" |
    "\&" | "\|" | "\^" |
    /* mutator */
```

(continues on next page)

(continued from previous page)

```
    "&=" | "|=" | "+=" | "-=" | "*=" | "/=" | "%=" | "^=" | "<<=" | ">>=" |
    /* comparison */
    "<" | ">" | "==" | "!=" | "<=" | ">=" | "<<" | ">>" | "<>"
  ;

  /* NOTE: upgrade to support n"wird + name" strings */
  literal flx_ident =># "(utf8->ucn _1)";
  literal tex_ident =># "_1";
  literal sym_ident =># "_1";

  sname := flx_ident =># "_1" | tex_ident =># "_1" | sym_ident =># "_1";

}
```

Description

C like identifiers

Felix has several forms of identifier. The C like identifier starts with a letter or underscore, and is followed by a sequence of letters, underscores digits, a hyphen, or a single quote mark. The letters can be any unicode code point allowed in ISO C++. Only ASCII digits are allowed.

Names starting with one or more underscores are reserved for the system. A unicode escape *uFFFF* or *uFFFFFFFF* may also be used and is translated to the UTF-8 byte sequence.

Tex Identifiers

Felix also allows any TeX identifier, a slosh ‘‘ followed by a sequence of ASCII letters.

N strings

The string form consisting of an *n* or *N* followed by a short string literal specifies the contents of the short string literal should be considered as an identifier. This is useful in two places: first, where the identifier is required to have a certain spelling so it can bind to C or C++ code. and secondly when an identifier is required but the parser would recognise it as a keyword in that context. Note that Felix has no keywords, identifiers are simply recognised as keyword like tokens in certain contexts, and not others. For example:

```
var var = 1; // the first var is a keyword, the second an identifier
n"var" = 2;  // var = 2 would be a syntax error
```

Special Symbols

Certain operators are recognised as identifiers to allow defining them as ordinary functions; from the specification:

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

1.3.3 Integer Literals

Syntax

```
SCHEME """
(define (findradix s)  ; find the radix of integer lexeme
  (let*
    (
      (n (string-length s))
      (result
        (cond
          ((prefix? "0b" s)`(,(substring s 2 n) 2))
          ((prefix? "0o" s)`(,(substring s 2 n) 8))
          ((prefix? "0d" s)`(,(substring s 2 n) 10))
          ((prefix? "0x" s)`(,(substring s 2 n) 16))
          (else `(,s 10))
        )
      )
    )
    result
  )
)
""";

SCHEME """
(define (findtype s) ;; find type of integer lexeme
  (let*
    (
      (n (string-length s))
      (result
        (cond
          ((suffix? "ut" s)`(,(substring s 0 (- n 2)) "utiny"))
          ((suffix? "tu" s)`(,(substring s 0 (- n 2)) "utiny"))
          ((suffix? "t" s)`(,(substring s 0 (- n 1)) "tiny"))

          ((suffix? "us" s)`(,(substring s 0 (- n 2)) "ushort"))
          ((suffix? "su" s)`(,(substring s 0 (- n 2)) "ushort"))
          ((suffix? "s" s)`(,(substring s 0 (- n 1)) "short"))

          ((suffix? "ui" s)`(,(substring s 0 (- n 2)) "uint"))
          ((suffix? "iu" s)`(,(substring s 0 (- n 2)) "uint"))
          ((suffix? "i" s)`(,(substring s 0 (- n 1)) "int"))

          ((suffix? "uz" s)`(,(substring s 0 (- n 2)) "size"))
          ((suffix? "zu" s)`(,(substring s 0 (- n 2)) "size"))
          ((suffix? "z" s)`(,(substring s 0 (- n 1)) "ssize"))

          ((suffix? "uj" s)`(,(substring s 0 (- n 2)) "uintmax"))
          ((suffix? "ju" s)`(,(substring s 0 (- n 2)) "uintmax"))
          ((suffix? "j" s)`(,(substring s 0 (- n 1)) "intmax"))

          ((suffix? "up" s)`(,(substring s 0 (- n 2)) "uintptr"))
          ((suffix? "pu" s)`(,(substring s 0 (- n 2)) "uintptr"))
          ((suffix? "p" s)`(,(substring s 0 (- n 1)) "intptr"))

          ((suffix? "ud" s)`(,(substring s 0 (- n 2)) "uptrdiff"))
          ((suffix? "du" s)`(,(substring s 0 (- n 2)) "uptrdiff"))
```

(continues on next page)

(continued from previous page)

```
          ((suffix? "d" s)`(,(substring s 0 (- n 1)) "ptrdiff"))

          ;; must come first!
          ((suffix? "uvl" s)`(,(substring s 0 (- n 3)) "uvlong"))
          ((suffix? "vlu" s)`(,(substring s 0 (- n 3)) "uvlong"))
          ((suffix? "ulv" s)`(,(substring s 0 (- n 3)) "uvlong"))
          ((suffix? "lvu" s)`(,(substring s 0 (- n 3)) "uvlong"))
          ((suffix? "llu" s)`(,(substring s 0 (- n 3)) "uvlong"))
          ((suffix? "ull" s)`(,(substring s 0 (- n 3)) "uvlong"))

          ((suffix? "uv" s)`(,(substring s 0 (- n 2)) "uvlong"))
          ((suffix? "vu" s)`(,(substring s 0 (- n 2)) "uvlong"))

          ((suffix? "lv" s)`(,(substring s 0 (- n 2)) "vlong"))
          ((suffix? "vl" s)`(,(substring s 0 (- n 2)) "vlong"))
          ((suffix? "ll" s)`(,(substring s 0 (- n 2)) "vlong"))

          ;; comes next
          ((suffix? "ul" s)`(,(substring s 0 (- n 2)) "ulong"))
          ((suffix? "lu" s)`(,(substring s 0 (- n 2)) "ulong"))

          ;; last
          ((suffix? "v" s)`(,(substring s 0 (- n 1)) "vlong"))
          ((suffix? "u" s)`(,(substring s 0 (- n 1)) "uint"))
          ((suffix? "l" s)`(,(substring s 0 (- n 1)) "long"))

          ;; exact
          ((suffix? "u8" s)`(,(substring s 0 (- n 2)) "uint8"))
          ((suffix? "u16" s)`(,(substring s 0 (- n 3)) "uint16"))
          ((suffix? "u32" s)`(,(substring s 0 (- n 3)) "uint32"))
          ((suffix? "u64" s)`(,(substring s 0 (- n 3)) "uint64"))
          ((suffix? "i8" s)`(,(substring s 0 (- n 2)) "int8"))
          ((suffix? "i16" s)`(,(substring s 0 (- n 3)) "int16"))
          ((suffix? "i32" s)`(,(substring s 0 (- n 3)) "int32"))
          ((suffix? "i64" s)`(,(substring s 0 (- n 3)) "int64"))
          (else `(,s "int"))
        )
      )
    )
    result
  )
)
""";

SCHEME """
(define (parse-int s)
  (let*
    (
      (s (tolower-string s))
      (x (findradix s))
      (radix (second x))
      (x (first x))
      (x (findtype x))
      (type (second x))
      (digits (first x))
      (value (string->number digits radix))
    )
```

(continues on next page)

(continued from previous page)

```
    (if (equal? value #f)
       (begin
         (newline)
         (display "Invalid integer literal ") (display s)
         (newline)
         (display "Radix ")(display radix)
         (newline)
         (display "Type ")(display type)
         (newline)
         (display "Digits ")(display digits)
         (newline)
        error
       )
       `(,type ,value)
    )
  )
)
""";

//$ Integer literals.
//$
//$ Felix integer literals consist of an optional radix specifer,
//$ a sequence of digits of the radix type, possibly separated
//$ by an underscore (_) or quote ('_ character, and a trailing type specifier.
//$
//$ The radix can be:
//$ 0b, 0B - binary
//$ 0o, 0O - octal
//$ 0d, 0D - decimal
//$ 0x, 0X - hex
//$
//$ The default is decimal.
//$ NOTE: unlike C a leading 0 in does NOT denote octal.
//$
//$ Underscores are allowed between digits or the radix
//$ and the first digit, or between the digits and type specifier.
//$
//$ The adaptable signed type specifiers are:
//$
//$ t         -- tiny   (char as int)
//$ s         -- short
//$ i         -- int
//$ l         -- long
//$ v,ll      -- vlong (long long in C)
//$ z         -- ssize (ssize_t in C, a signed variant of size_t)
//$ j         -- intmax
//$ p         -- intptr
//$ d         -- ptrdiff
//$
//$ These may be upper of lower case.
//$ A "u" or "U" before or after such specifier indicates
//$ the correspondin unsigned type.
//$
//$ The follingw exact type specifiers can be given:
//$
//$      "i8" | "i16" | "i32" | "i64"
//$    | "u8" | "u16" | "u32" | "u64"
```

(continues on next page)

(continued from previous page)

```
//$    | "I8" | "I16" | "I32" | "I64"
//$    | "U8" | "U16" | "U32" | "U64";
//$
//$ The default type is "int".
//$

syntax felix_int_lexer {
  /* integers */
  regdef bin_lit  = '0' ('b' | 'B') (dsep ? bindigit) +;
  regdef oct_lit  = '0' ('o' | 'O') (dsep ? octdigit) +;
  regdef dec_lit  = '0' ('d' | 'D') (dsep ? digit) +;
  regdef dflt_dec_lit  =  digit (dsep ? digit) *;
  regdef hex_lit  = '0' ('x' | 'X') (dsep ? hexdigit)  +;
  regdef int_prefix = bin_lit | oct_lit | dec_lit | dflt_dec_lit | hex_lit;

  regdef fastint_type_suffix =
    't'|'T'|'s'|'S'|'i'|'I'|'l'|'L'|'v'|'V'|"ll"|"LL"|"z"|"Z"|"j"|"J"|"p"|"P"|"d"|"D";
  regdef exactint_type_suffix =
      "i8" | "i16" | "i32" | "i64"
    | "u8" | "u16" | "u32" | "u64"
    | "I8" | "I16" | "I32" | "I64"
    | "U8" | "U16" | "U32" | "U64";

  regdef signind = 'u' | 'U';

  regdef int_type_suffix =
      '_'? exactint_type_suffix
    | ('_'? fastint_type_suffix)? ('_'? signind)?
    | ('_'? signind)? ('_'? fastint_type_suffix)?;

  regdef int_lit = int_prefix int_type_suffix;

  // Untyped integer literals.
  literal int_prefix =># """
  (let*
    (
      (val (stripus _1))
      (x (parse-int val))
      (type (first x))
      (value (second x))
    )
    value
  )
  """;
  sinteger := int_prefix =># "_1";

  // Typed integer literal.
  literal int_lit =># """
  (let*
    (
      (val (stripus _1))
      (x (parse-int val))
      (type (first x))
      (value (second x))
      (fvalue (number->string value))
      (cvalue fvalue)       ;; FIXME!!
    )
```

(continues on next page)

(continued from previous page)

```
    `(,type ,fvalue ,cvalue)
  )
  """;
  sliteral := int_lit =># "`(ast_literal ,_sr ,@_1)";

  // Typed signed integer constant.
  sintegral := int_lit =># "_1";
  sintegral := "-" int_lit =># """
  (let*
    (
      (type (first _2))
      (val (second _2))
      (val (* -1 val))
    )
    `(,type ,val)
  )
  """;

  strint := sintegral =># "(second _1)";
}
```

1.3.4 Floating Numbers

Syntax

```
//$ Floating point literals.
//$
//$ Follows ISO C89, except that we allow underscores;
//$ AND we require both leading and trailing digits so that
//$ x.0 works for tuple projections and 0.f is a function
//$ application
syntax felix_float_lexer {
  regdef decimal_string = digit (underscore ? digit) *;
  regdef hexadecimal_string = hexdigit (underscore ? hexdigit) *;

  regdef decimal_fractional_constant =
    decimal_string '.' decimal_string;

  regdef hexadecimal_fractional_constant =
    ("0x" |"0X")
    hexadecimal_string '.' hexadecimal_string;

  regdef decimal_exponent = ('E'|'e') ('+'|'-')? decimal_string;
  regdef binary_exponent = ('P'|'p') ('+'|'-')? decimal_string;

  regdef floating_suffix = 'L' | 'l' | 'F' | 'f' | 'D' | 'd';
  regdef floating_literal =
    (
      decimal_fractional_constant decimal_exponent ? |
      hexadecimal_fractional_constant binary_exponent ?
    )
    floating_suffix ?;

 // Floating constant.
```

(continues on next page)

(continued from previous page)

```
  regdef sfloat = floating_literal;
  literal sfloat =># """
  (let*
     (
       (val (stripus _1))
       (val (tolower-string val))
       (n (string-length val))
       (n-1 (- n 1))
       (ch (substring val n-1 n))
       (rest (substring val 0 n-1))
       (result
         (if (equal? ch "l") `("ldouble" ,val ,val)
           (if (equal? ch "f") `("float" ,val ,val) `("double" ,val ,val))
         )
       )
     )
     result
   )
   """;

  strfloat := sfloat =># "(second _1)";

  // Floating literal.
  sliteral := sfloat =># "`(ast_literal ,_sr ,@_1)";

}
```

1.3.5 String Forms

Syntax

```
SCHEME """
(define (decode-string s)
  (begin
    (adjust-linecount s)
    (let*
      (
        (n (string-length s))
        (result
          (cond
            ((prefix? "w'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "W'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "c'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "C'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "u'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "U'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "f'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "F'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "q'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "Q'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "n'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "N'''" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "r'''" s)(substring s 4 (- n 3)))
            ((prefix? "R'''" s)(substring s 4 (- n 3)))
```

(continues on next page)

(continued from previous page)

```
            ((prefix? "'''" s)(unescape (substring s 3 (- n 3))))

            ((prefix? "w\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "W\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "c\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "C\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "u\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "U\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "f\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "F\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "q\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "Q\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "n\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "N\"\"\"" s)(unescape (substring s 4 (- n 3))))
            ((prefix? "r\"\"\"" s)(substring s 4 (- n 3)))
            ((prefix? "R\"\"\"" s)(substring s 4 (- n 3)))
            ((prefix? "\"\"\"" s)(unescape (substring s 3 (- n 3))))

            ((prefix? "w'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "W'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "c'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "C'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "u'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "U'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "f'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "F'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "q'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "Q'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "n'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "N'" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "r'" s)(substring s 2 (- n 1)))
            ((prefix? "R'" s)(substring s 2 (- n 1)))
            ((prefix? "'" s)(unescape (substring s 1 (- n 1))))

            ((prefix? "w\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "W\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "c\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "C\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "u\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "U\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "f\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "F\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "q\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "Q\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "n\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "N\"" s)(unescape (substring s 2 (- n 1))))
            ((prefix? "r\"" s)(substring s 2 (- n 1)))
            ((prefix? "R\"" s)(substring s 2 (- n 1)))
            ((prefix? "\"" s)(unescape (substring s 1 (- n 1))))

            (else error)
          )
        )
      )
      ;;(begin
      ;;   (newline)(display "string=")(display s)
      ;;   (newline)(display "text=")(display result)
```

(continues on next page)

(continued from previous page)

```
          result
      ;;)
    )
  )
)
""";

// Scheme string to Felix string literal
SCHEME """
(define (strlit s)
    `(ast_literal ,_sr "string" ,s ,(string-append "::std::string(" (c-quote-string␣
↪s) ")"))
)
""";

//$ String literals.
//$
//$ Generaly we follow Python here.
//$ Felix allows strings to be delimited by;
//$
//$ single quotes '
//$ double quotes "
//$ triped single quotes '''
//$ 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.
//$
//$ These forms all allows embedded escape codes.
//$ These are:
//$
//$  \a  -  7 : bell
//$  \b  -  8 : backspace
//$  \t  -  9 : horizontal tab
//$  \n  - 10 : linefeed, newline
//$  \r  - 13 : carriage return
//$  \v  - 11 : vertical tab
//$  \f  - 12 :form feed
//$  \e  - 27 : escape
//$  \\  - \  : slosh
//$  \"  - "  : double quote
//$  \'  - '  : single quote
//$  \   - 32 : space
//$
//$  \xFF - hexadecimal character code
//$  \o7 \o77 \o777 -- octal character code (stops on count of 3 or non-octal␣
↪character)
//$  \d9 \d99 \d999 -- decimal character code (stops on count of 3 or non-decimal␣
↪character)
//$  \uFFFF - utf8 encoding of specified hex value
//$  \UFFFFFFFF - utf8 encoding of specified hex value
//$
//$ 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!
```

(continues on next page)

(continued from previous page)

```
//$
//$ A prefix "w" or "W" specifies a wide character string,
//$ of character type wchar. DEPRECATED.
//$
//$ A prefix of "u" or "U" specifes a string of uint32.
//$ This is a full Unicode string.
//$ THIS FEATURE WILL BE DEPRECATED.
//$ IT WILL BE REPLACED BY C++11 Unicode compliant 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.
//$
//$ 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.
//$
//$ A literal prefixed by a "f" or "F" is a C format string.
//$ Such strings are actually functions.
//$ The string contains code such as "%d" or other supported
//$ C format string. Variable field width specifiers "*" are
//$ not permitted. The additional format specification %S
//$ is supported and requires a C++ string argument.
//$ Such functions accept a tuple of values like this:
//$
//$ f"%d-%S" (42, "Hello")
//$
//$ 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.
//$
//$ The special literal with a "n" or "N" prefix is a way to encode
//$ an arbitrary sequence of characters as an identifer 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];
//$
syntax felix_string_lexer {
  /* Python strings */
  regdef qqq = quote quote quote;
  regdef ddd = dquote dquote dquote;

  regdef escape = slosh _;

  regdef dddnormal = ordinary | hash | quote | escape | white | newline;
  regdef dddspecial = dddnormal | dquote dddnormal | dquote dquote dddnormal;

  regdef qqqnormal = ordinary | hash | dquote | escape | white | newline;
  regdef qqqspecial = qqqnormal | quote qqqnormal | quote quote qqqnormal;
```

(continues on next page)

(continued from previous page)

```
  regdef qstring_tail = (ordinary | hash | dquote | escape | white) * quote;
  regdef dstring_tail = (ordinary | hash | quote | escape | white) * dquote;
  regdef qqqstring_tail = qqqspecial * qqq;
  regdef dddstring_tail = dddspecial * ddd;

  regdef qstring = quote qstring_tail;
  regdef dstring = dquote dstring_tail;
  regdef qqqstring = qqq qqqstring_tail;
  regdef dddstring = ddd dddstring_tail;


  regdef raw_dddnormal = ordinary | hash | quote | slosh | white | newline;
  regdef raw_dddspecial = raw_dddnormal | dquote raw_dddnormal | dquote dquote raw_
↪dddnormal;

  regdef raw_qqqnormal = ordinary | hash | dquote | slosh | space | newline;
  regdef raw_qqqspecial = raw_qqqnormal | quote raw_qqqnormal | quote quote raw_
↪qqqnormal;

  regdef raw = 'r' | 'R';

  regdef raw_dstring_tail =  (ordinary | hash | quote | escape | white) * dquote;
  regdef raw_qqqstring_tail = raw_qqqspecial * qqq;
  regdef raw_dddstring_tail = raw_dddspecial * ddd;

  regdef raw_dstring = raw dquote dstring_tail;
  regdef raw_qqqstring = raw qqq qqqstring_tail;
  regdef raw_dddstring = raw ddd dddstring_tail;

  regdef plain_string_literal = dstring | qqqstring | dddstring;
  regdef raw_string_literal = raw_dstring | raw_qqqstring | raw_dddstring;

  regdef string_literal = plain_string_literal | qstring | raw_string_literal;

  regdef wstring_literal = ('w' | 'W') plain_string_literal;
  regdef ustring_literal = ('u' | 'U') plain_string_literal;
  regdef cstring_literal = ('c' | 'C') plain_string_literal;
  regdef qstring_literal = ('q' | 'Q') plain_string_literal;
  regdef fstring_literal = ('f' | 'F') plain_string_literal;
  regdef nstring_literal = ('n' | 'N') plain_string_literal;

   // String as name.
  literal nstring_literal =># "(decode-string _1)";
  sname := nstring_literal =># "_1";

  // String for pattern or code template.
  regdef sstring = string_literal;
  literal sstring =># "(decode-string _1)";

  // Cstring for code.
  regdef scstring = cstring_literal;
  literal scstring =># "(decode-string _1)";

  // String for string parser.
  regdef strstring = string_literal;
  literal strstring =># "(c-quote-string (decode-string _1))";
```

(continues on next page)

(continued from previous page)

```
// String like literals.
regdef String = string_literal;
literal String =># """
  (let*
    (
      (ftype "string")
      (iv (decode-string _1))
      (cv (c-quote-string iv))
      (cv (string-append "::std::string(" cv ")"))
    )
    `(ast_literal ,_sr ,ftype ,iv ,cv)
  )
""";
sliteral := String =># "_1";

regdef Wstring = wstring_literal;
literal Wstring =># """
  (let*
    (
      (ftype "wstring")
      (iv (decode-string _1))
      (cv (c-quote-string iv))
      (cv (string-append "wstring(" cv ")"))
    )
    `(ast_literal ,_sr ,ftype ,iv ,cv)
  )
""";
sliteral := Wstring =># "_1";

regdef Ustring = ustring_literal;
literal Ustring =># """
  (let*
    (
      (ftype "ustring")
      (iv (decode-string _1))
      (cv (c-quote-string iv))
      (cv (string-append "ustring(" cv ")"))
    )
    `(ast_literal ,_sr ,ftype ,iv ,cv)
  )
""";
sliteral := Ustring =># "_1";

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";
```

(continues on next page)

(continued from previous page)

```
  regdef Qstring = qstring_literal;
  literal Qstring =># "`(ast_interpolate ,_sr ,(decode-string _1))";
  sliteral := Qstring =># "_1";

  regdef Fstring = fstring_literal;
  literal Fstring =># "`(ast_vsprintf ,_sr ,(decode-string _1))";
  sliteral := Fstring =># "_1";

}
```

Description

Felix provides string like literals with several roles:

- strings
- C strings
- arbitrary identifiers
- formatting functions
- interpolation strings

Short Literal

A basic string literal consists of a quote ‘, some text excluding a quote, and a terminating quote, all on one line, or, a double quote “, some text excluding a double quote, and a terminating double quote, all on one line. The text consists of UTF-8 encoded Unicode and should not contain any code points below space (0-0x1F). The system does not check the validity of the UTF-8 encoding or code points represented.

String literals have type *string* in Felix and represented by C++ *::basic_string<char>*.

Long Literal

A long literal consist of three quotes ‘’‘, some text which may include the end of a line, does not contain three quotes, and is terminated by three quotes, or, three double quotes “”“, some text excluding three double quotes, which may span multiple lines, terminated by three double quotes. The system does not check the validity of the UTF-8 encoding or code points represented. Long literals are sometimes called triple quoted strings.

Escape Codes

Short and long literals may include escape codes. Although most of the literal text is processed as written, escape codes are translated to other sequences of bytes. An escape code consists of a slosh ‘‘ character and some following characters.

C Escapes

Numeric Escapes

Numeric escapes start with *d*, *o*, or *x* followed by digits in decimal, octal, or hex radices respectively. Hex letters can be upper or lower case. The escape is terminated by either a non-radix character, or the maximum number allowed digits: 3,3 and 2 respectively.

Unicode Escapes

A unicode escape is *u* and exactly 4 hex digits or *U* and exactly 8 hex digits. The hex encoding is translated to an integer and then the escape is replaced by the UTF-8 representation of that code point. Felix uses a full UTF-8 encoding so up to 2^32 values of 1 to 5 bytes may be generated.

Raw Strings

A short or long string literal using double quote delimiters may be prefixed by *r* or *R* indicating a raw string, in which escape codes are not recognised. Note the prefix cannot be used for single quoted strings because single quotes are allowed in identifiers.

Raw strings are mainly used for regular expression strings because regular expression encodings contain a lot of special characters, particularly sloshes: without raw strings each slosh would have to be encoded as two sloshes. They're also useful on Windows where slosh is a path separator.

C strings

A string of type *cstring* which is represented by an array of characters in C (that is, a pointer to a char) can be created by prefixing a string literal with *c*.

Identifiers

An identifier can be written as a string prefixed by *n*. This is useful if an identifier would not be recognised in a certain context. For example:

```
var var = 1;
n"var" = 2;
```

Format Functions

A string literal prefixed by *f* or '*F* is a format function. It uses C *printf* like codes and is implemented using *::std::vsnprintf*. It is first called with a NULL string and 0 length for the buffer to calculate the required buffer size, then the buffer is allocated and the function actually run.

Felix also supports a *%S* specifier for C++ strings. It is converted to *%s* and the internal char array of the C++ string used as an argument.

The * format specifier is not supported.

The compiler scans the string to calculate the type of the arguments. The arguments must be presented as a tuple.

```
println$ f'Hello %S on %d' ("Felix", 42);
```

Code	Type
hhd	tiny
hhi	tiny
hho	utiny
hhx	utiny
hhX	utiny
hd	short
hi	short
hu	ushort
ho	ushort
hx	ushort
hX	ushort
d	int
i	int
u	uint
o	uint
x	uint
X	uint
ld	long
li	long
lu	ulong
lo	ulong
lx	ulong
lX	ulong
lld	vlong
lli	vlong
llu	uvlong
llo	uvlong
llx	uvlong
llX	uvlong
zd	ssize
zi	ssize
zu	size
zo	size
zx	size
zX	size
jd	intmax
ji	intmax
ju	uintmax
jo	uintmax
jx	uintmax
jX	uintmax
td	ptrdiff
ti	ptrdiff
tu	uptrdiff
to	uptrdiff
tx	uptrdiff
tX	uptrdiff
e	double
E	double
f	double

Continued on next page

Table 1 – continued from previous page

Code	Type
F	double
g	double
G	double
a	double
A	double
Le	ldouble
LE	ldouble
Lf	ldouble
LF	ldouble
Lg	ldouble
LG	ldouble
La	ldouble
LA	ldouble
c	int
S	string
s	&char
p	address
P	address

Interpolation Strings

A string prefixed by a *q* or *Q* is an interpolation string. Such a string may contain *$(varname)* where varname is a visible variable name. The code is replaced by *%S*, and a tuple whose components are the application of the *str* function to the variable is formed and the string, considered now as a format function literal, is applied to it.

```
var x = 1; var y = 2;
println$ q"$(x) + $(y)";
```

Constant Folding

Felix compiler concatenates adjacent string literals. So for example you can do this:

```
var x =
  "To be\n"
  "Or not to be\n"
  "That is the question\n"
;
```

1.4 Macros

Felix only provides a rudimentary macro system. Macros are scoped like other symbols. The primary purpose of macros is to support conditional compilation. Macros are usually gathered into a single file which is prepended using a compiler switch to all other files, after parsing.

For advanced synthetic code generation, Felix uses Scheme programming language in user action codes of the grammar.

1.4.1 Syntax

```
stmt := "macro" "val" snames "=" sexpr ";"

stmt := "forall" sname "in" sexpr "do" stmt* "done"

x[sthename_pri] := "noexpand" squalified_name
```

1.4.2 Semantics

The *macro val* statement is used to associate a macro name with an expression.

The *forall* statement is used to generate a sequence of statements repeatedly replacing occurences of the given name with each of the given expression in turn, and is useful for table generation.

The noexpand prefix is used to prevent expansion of a name which might have been a macro val. It is stripped out by the macro processor.

1.4.3 Constant Folding

The Felix macro processor performs constant folding for certain types and operations.

Expressions involving only boolean constants *true* and *false* and the logical operations *and*, *or* and *not* are evaluated and simplified.

Adjacent string literals are concatenated.

1.4.4 Conditional Compilation

Felix does not provide any specific directives for conditional compilation. Instead, conditional expressions and statements with compile time constant conditions are simplified by the macro processor, providing conditional compilation using the same syntax as run time conditionals.

For example:

```
macro val POSIX = true;
if POSIX do
   typedef file = int;
else do
   typedef file = HANDLE;
done
```

is reduced to:

```
typedef file = int;
```

by the macro processor, so that even if *HANDLE* is not defined, there will be no type error because the code using it is eliminated.

The macro processor also reduces pattern matches if possible, in fact, conditionals are represented by pattern matches!

1.5 Directives

Directives control the operation of the Felix compiler.

1.5.1 Directive Statements

include directive

The include directive causes the compiler to ensure the specified file is parsed, and ensure the abstract syntax tree generated by parsing is prepended to the current AST being generated by the parser.

See *Include Directive* for details.

open directive

The open directive injects the set of public names of the specified class or library specialised to the given types, in a shadow lookup scope just underneath the primary scope in which the open directive is written.

Symbols in shadow scopes are hidden by definitions in the primary scope and are not exported as as public members of the current primary class.

```
class X { fun f(x:int)=> x * x; }
class Y { open X; fun g(y:int) => f x; }
class Z {
  open Y;
  fun g(y:int) => y + 1; // hides Y::g of int
  fun h(k:int) => X::f k; // X::f not visible without qualification
}
open X, Y, Z;
println$ f 1; // X::f
println$ Y::g 1; // resolve X::g, Y::g ambiguity
```

An example with specialisations:

```
class P[T] { fun h(x:T) => x,x; }
open[U] P[U];
println$ f (f (f 1));
```

inherit directive

Syntax

```
namespace_stmt := "inherit" stvarlist squalified_name ";"
```

Description

The inherit directive injects the set of public names of the specified class or library specialised to the given types, into the current primary scope, as if they were defined there.

Injected symbols may clash with each other and definitions in the primary scope. Clashes with non-function symbols lead to a fatal compiler error because the compiler cannot construct a suitable entry in the symbol table. Clashes with function symbols lead to ambiguity errors only when the function name is used and overload resolution performed. Such clashes can be resolved by using a qualified name.

```
class X { fun f(x:int) => x * x; }
class Y { inherit X; fun g(x:int) => f x; }
println$ Y::f 1; // Y::f aka X::f
```

rename directive

Syntax

```
namespace_stmt := "rename" sdeclname "=" squalified_name ";"
namespace_stmt := "rename" "fun" sdeclname "=" squalified_name ";"
```

Decription

The rename directive can be is used to inject a single name into the current scope defining it by another name, either from the current scope, or some other scope. The name can either be a non-function name, a function name, or a class name. The name can be polymorphic and the defining expression can be specialised.

```
class X {
  fun f(x:int) => x * x;
  fun f(x:double) => x * x;
}
class Y {
  rename fun g = X::f;
  fun h(x:int) => g x;
}
println$ Y::g 1; // X::f of int
```

use directive

Syntax

```
namespace_stmt := "use" sname "=" squalified_name ";"
namespace_stmt := "use" squalified_name ";"
```

Description

The use directive injects a single symbol or set of function signatures into the current scope, as if it were defined there. It is a special shortcut version of the *rename* directive used when the injected name is the same as the source name.

library directive

Syntax

```
namespace_stmt := "library" sname "=" ? scompound
namespace_stmt := "open" "library" sname "=" ? scompound
```

Description

The library directive constructs an part of an extensible scope. Multiple library directives can be given for the same name. Libraries can therefore be defined in multiple files, whereas classes must be specified in a single file.

Libraries cannot be polymorphic and serve only to provide a qualified name prefix for names.

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

class directive

Syntax

```
namespace_stmt := "class" sdeclname ";"
```

Description

The class directive specifies the rest of the current file should be considered as a class definition. The directive is syntactic sugar for the standard class definition, the entire purpose is to allow easier indentation of the text.

Qualified Names

Names can be qualified by the class of library in which to lookup the name. This can be used to resolve ambiguities, or, to find a symbol if the class or library containing the name is not open. Opening classes or libraries causes namespace pollution, which is especially problematic if the open is in the top level (global or root) scope and is generally reserved for core algebras.

Export directive

Syntax

```
stmt := "export" "requires" srequirements ";"

cbind_stmt := "export" "fun" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "cfun" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "proc" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "cproc" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "struct" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "union" ssuffixed_name "as" sstring ";"

cbind_stmt := "export" "type" "(" sexpr ")" "as" sstring ";"

stmt := "export" "python" "fun" ssuffixed_name "as" sstring ";" =>#
```

The export directive tell the compiler to export a symbol with a special name. The *export* directive can also be used as an adjective.

See also *export adjective*

export python directive

The *export python* directive tells the compiler the function is part of a Python module. It has no effect on the function itself, however it causes the compiler to generate a Python module table containing the function in the output. Felix generates module tables for Python 3. To work correctly the function must have arguments and return types compliant with Python C API.

1.5.2 Adjectival directives

A function, generator, procedure or type definition may be prefixed with an adjectival directive that provides instructions for its use or properties.

inline adjective

A function or procedure definition can be qualified by the adjective *inline* to tell the compiler to inline direct applications or calls.

Recursive functions or procedures cannot be inlined, it is an error to specify inline for them. [Currently ignored].

An inline function will not be inlined if it is invoked via a closure.

noinline adjective

The *noinline* adjective on a function or procedure definition tells the compiler not to inline it.

The *inline* and *noinline* directives are not optimisation hints, they are mandatory requirements with semantic impact. This is because Felix has indeterminate evaluation strategy and may choose to eagerly or lazily evaluate arguments. Indirect calls or direct calls to recursive function cannot be inlined and necessarily use eager evaluation. Inlined calls generally use lazy evaluation.

pure adjective

The *pure* adjective tells the compiler the programmer thinks the function is pure; that is, it is dependent only on its parameters. Dependence on invariant symbols outside the definition is permitted, however invoking an impure function or procedure is not.

The compiler will examine the function to try to determine if it is pure. If the compiler can prove it is not pure, the compilation will be aborted with an error message, otherwise the function will be taken as pure. [Purity violation is not currently implemented]

impure adjective

The *impure* adjective tells the compiler to treat the function as impure, whether it is actually impure or not.

Purity helps enable certain optimisations. For functions, purity ensures referential transparency.

total adjective

The *total* adjective tells the compiler the function or procedure will work correctly with all arguments of the correct type, that is, that there are no pre-conditions.

Felix provides a way to specify pre-conditions, but not all pre-conditions can or should be specified, and pre-conditions can and usually are omitted.

partial adjective

The *partial* adjective tells the compiler the function may fail with some correctly typed arguments, that is, that the function may have pre-conditions.

If pre-conditions are given along with the partial adjective it should indicate the pre-conditions are not complete.

strict adjective

The strict adjective tells the compiler that if an argument expression is evaluated lazily and fails, then the function would have failed anyhow.

Some functions require lazy evaluation. For example consider:

```
fun myif(c:bool, t:int, f:int) =>
  if c then t else f
 ;
 var y = 0;
 var x = myif(y==0, 1, 1/y);
```

This code will crash if the third argument to *myif* is evaluated before the function is called, even though the final result does not depend on it it. However if the application is inlined the resulting expression:

```
if y==0 then 1 else 1/y endif
```

will not crash becuase the else branch is not taken. Indeed in the example the compiler may optimise the code to just *1* because it knows *y==0* must be true and the nasty division by zero is not only not executed, it isn't even present in the code.

The *strict* adjective tells the compiler it is safe to eagerly evaluate the function application: if the evaluation of the argument would fail, then the function would fail even with lazy evaluation, for example because the argument is always are required.

Felix assumes functions are strict. Even if this is not the case, the function may still work correctly on the arguments for which it applied.

nonstrict adjective

This tells the programmer the function is not strict in one or more arguments. It has no effect on the compiler, which continues to assume the function is, in fact strict. Rather, it tells the programmer to be careful to call the function with arguments for which eager and lazy evaluation would produce the same result.

If this is not possible the programmer must change the argument type to accept a closure and evaluate the argument on demand, thereby enforcing lazy evaluation.

method adjective

The *method* adjective may only be used in an *object* and tells the compiler a closure of the function over the objects internal state must be included in the record value returned as the value of a field named after the function name.

virtual adjective

The virtual adjective can only be used in a class and tells the compiler the function, procedure, or type may be overriden in an instance. A virtual function must be defined in an instance if, and only if, it is actually used, and, it is not defined in the class.

export adjective

The export adjective is equivalent to an export directive specifying the function or type, providing the C name the same as the Felix nae.

See also *Export directive*

private adjective

The *private* adjective tells the compiler that the symbol being defined is private to the current class and should not be exported. Each class has a symbol table with two indices: the public index and the private index.

The private index maps names to definitions of all symbols defined in the class, whereas the public index omits symbols marked private.

Helper functions should be marked private as they are not intended to be used by the client of the class. Types intended only for internal as implementation details should also be marked private.

Note only the public access to the name of a private type is hidden: the type itself is still visible. For example a public function can return a value of a private type. The client can still name the type. For example:

```
class X {
  private typedef t = int;
  fun f () : t => 1;
}
var x = X::f();
typedef u = typeof x;
var z = x + x;
```

The client now has a name *u* for the type, even though they do not know it is an *int*. Also the calculation of *z* is legitimate, even though it depends on the type of *x* being an *int*. Therefore, *private* hides only the name of a definition.

Similarly and more obviously, a client cannot directly call a private function but they can call a closure of it a public function returns.

pod adjective

The *pod* adjective tells the compiler a type lifted from C is a plain old datatype. This means that it has a trivial destructor, it tells the compiler to omit the pointer to the destructor in a generated RTTI object, and this ensure the garbage collector will not waste time invoking the destructor when it doesn't do anything anyhow.

Note that in Felix all data types must be first class which means they must by copyable, movable, and assignable (unless marked incomplete).

incomplete adjective

The *incomplete* adjective tells the compiler a type is not first class, and that expressions of the type may not be used. However pointers to such types may be used. Such pointers, however, cannot be dereferenced so the type acts as a phantom to separate these pointers from each other by type.

The *incomplete* adjective only makes sense on a C type binding.

uncopyable adjective

The *uncopyable* adjective tells the compiler a value of the type cannot be copied or assigned. They can, however, be constructed and destroyed.

This adjective has no semantics at the moment but is intended to tell the garbage collector the type may not be used in a copyable arena. Copying collectors work by having two arenas, and copy, compactly, objects from one arena into a fresh arena, then delete the old arena, in order to perform their function (that is, they do not collect and dispose of garbage but collect and retain reachable objects instead).

When allocating an object, the copyability attribute is passed as a boolean flag to the C++ operator new, so it can choose to place the object in an uncopyable space, whilst other objects are placed in a copyable space. The intention is to allow copyable objects to be compacted by moving them together in an arena, improving performance and freeing up larger blocks of free space. However the current Felix gc does not do compaction and the flag is ignored.

_gc_pointer adjective

The *_gc_pointer* adjective tells the compiler a type lifted from C is actually a pointer to an object managed gy the garbage collector. The type must be a pointer. The effect is to include storage locations of this type in the table of offsets of pointers in the RTTI object for any type containing an *_gc_pointer*, so that the garbage collector can trace it.

_gc_type T adjective

The *_gc_type T* adjective must be used in conjunction with the *_gc_pointer* adjective and tells the compiler the actual type pointed at is *T*.

The effect is that during code generation a C binding specification which requires a pointer to an RTTI object will provide one pointing to a *T* rather than the expected type. Here is an example:

```
private incomplete type RE2_ = "::re2::RE2";
_gc_pointer _gc_type RE2_ type RE2 = "::re2::RE2*";
gen _ctor_RE2 : string -> RE2 = "new (*PTF gcp, @0, false) RE2($1)";
```

We bind the private type *RE2_* to the C type *RE2*. This is the type of a Google RE2 regular expression object.

It's private so the public cannot allocate it. Instead we use the type RE2 which is a pointer, and thus copyable. Because it is a pointer we have to specify *_gc_pointer*.

Now, the constructor *_ctor_RE2* takes a string and returns a Felix *RE2* (C type *RE2**) which is a pointer to a heap allocated object of type *_RE2* (C type *RE2*).

The constructor does the allocation, so it must provde the shape of the *RE2_* object, and this is what the specification *_gc_type RE2_* does. This allows the notation *@0* to refer to the shape of *RE2_* instead of RE2 which it would normally.

1.6 Type System

Felix is a statically typed scripting language which has a rich and powerful type system. It supports type deduction and both ad-hoc and type class based overloading, but not type inference. It provides standard parametric polymorphism with ad-hoc weak constraints.

Contents:

1.6.1 Kinding System

Felix has a basic kinding system. There is one builtin kind, namely *TYPE*, the kind of all types.

There are two kind constructors. The infix binary function constructor -> can be used to denote the kind of a type function. The chained non-associative n-ary product constructor * can be used to denote the kind of a type tuple.

Kinds are required in some circumstances, for example Monads are parametrised by a type function:

```
class Monad [M: TYPE->TYPE] {
  virtual fun ret[a]: a -> M a;
  virtual fun bind[a,b]: M a * (a -> M b) -> M b;
  fun join[a] (n: M (M a)): M a => bind (n , (fun (x:M a):M a=>x));
}
```

A suitable function is:

```
typedef fun opt_f (T: TYPE): TYPE => opt[T];
```

which has kind:

```
TYPE -> TYPE
```

A kind annotation can be used with a type variable in a polymorphic specification, if omitted it defaults to *TYPE*. Types also are implicitly kinded as *TYPE*.

When a list of type variables is given separated by commas each type is implicitly kinded as TYPE and the whole list is implicitly kinded as a type tuple. For example in

```
fun f(T,U] (x:T,y:U) => x,y;
```

the type variable list *T,U* has kind

```
TYPE * TYPE
```

Two more kinds will be introduced in the future:

```
BOOL
INT
```

These are the kinds of compile time booleans and integers, respectively.

1.6.2 Type Aliases

Simple Alias

Felix allows you to define an alias for a type:

```
typedef myint = int;
```

Type Schema

An alias can be polymorphic, in which case it specifies a type schema:

```
typedef pair[T,U] = T * U;
var x: pair[int, double] = 42, 7.2;
```

1.6.3 Type Functions

A type function can be defined: this is a function which takes one or more types and returns a type:

```
typedef fun pair_f(T:TYPE,U:TYPE):TYPE  => T * U;
var x : pair_f (int, long) = 42, 7.2;
```

Note there is a technical difference between a type schema, which is an index family of types, and a type function, which is a higher kind. The entity pair_f has kind:

```
TYPE * TYPE -> TYPE
```

Since the domain is a pair of types, a type tuple must be given as an argument. A type tuple is of a different kind to the type *of* a tuple, which is merely of kind TYPE.

Type function applications must be resolved during phase 1 lookup.

Functors

If the type function is structurally parametric, it is also called a functor. Not all type functions are structure preserving, which is the requirement for a functor.

Functors can be used as arguments to type classes. For example the library function Monad depends on a functor:

```
class Monad [M: TYPE->TYPE] {
  virtual fun ret[a]: a -> M a;
  virtual fun bind[a,b]: M a * (a -> M b) -> M b;
  fun join[a] (n: M (M a)): M a => bind (n , (fun (x:M a):M a=>x));
}
```

1.6.4 Type Matches

Syntax

Felix has a special type expression known as a type match, it is the type system analogue of a match:

```
typedef fun dom (T:TYPE):TYPE =>
  typematch T with
  | ?D -> _ => D
  | _ => 0
  endmatch
;
```

The type function dom above finds the domain type or a function type, if the argument is a function type, otherwise it returns the void type 0. Note that dom does not preserve structure and so is not a functor.

An identifier in a type pattern is assumed to be a type name unless it is prefixed by *?* which indicates a pattern variable.

Semantics

The rule for reduction of a type match is non-trivial!

First, the algorithm compares the match argument type with the pattern of the first branch. If it matches, it returns the RHS of the branch, replacing any occurences of the pattern variables with the parts of the argument that they have bound to.

If the argument *can never match* then the first branch is discarded and we proceed to the next branch. If there are no branches left the typematch fails.

If the argument *can possibly match in the future* but doesn't at present, the whole type match is returned (minus any branches which have been discarded).

Match is done by unification. First, we look to see if the argument is a specialisation of the pattern, this includes equivalence. If so we have a match. If not we determine if the pattern unifies with the argument. If not, the argument can never be a specialisation of the pattern, even after type variables are replaced in the argument. Otherwise, it may become a specialisation in the future.

For example:

```
typedef X[T] =
  typematch T with
  | ?D -> ?C => 1
  | _ -> 0
  endmatch
;
```

As written, the argument T is not a specialisation of the first pattern *D -> C* but it could become one if, for example, we set T to *int->long*. In that case the most general unifier (MGU) would be the pair *D<-int*,'C<-long'. Therefore, reduction is deferred.

Overload resolution cannot proceed with unreduced type matches. Therefore, it attempts to remove all type matches and fails if at least one cannot be reduced away.

1.6.5 Type Sets

Felix allows you define a set of types:

```
typedef sints = typesetof (short,int,long);
typedef uints = typesetof (ushort,uint,ulong);
```

The list of elements of a type set can be either types or typesets:

```
typedef ints = typesetof (sints,uints, byte);
```

You can also calculate the union of two typesets with the TeX operator *cup*, the symbol ..math:*cup*, it also accepts types as an abuse of notation.

```
typeset ints = sints \cup uints \cup byte;
```

Membership of a typeset can be tested with the *in* operator:

```
long in ints
```

The result is a type, either void (0) for false and unit (1) for true. [This is a design fault, the result should of kind BOOL]

Typesets are used for constrained polymorphism, primarily with primitive bindings:

```
fun add[T where T in ints]) : T * T -> T = "$1+$2";
fun add[T in ints]) : T * T -> T = "$1+$2";
fun add[T:ints] : T * T -> T = "$1+$2";
```

The second and third forms are a shortcut versions of the first.

When a query is made of a typeset, it is expanded to a typematch:

```
// T in sints
typematch T with
| short => 1
| int => 1
| long => 1
| _ => 0
endmatch
```

This means the elements of a typeset can be parameterised by one or more type variables provided they're in scope. For example:

```
typedef iorspair[S,U] = typesetof (S * S, U * U);
fun add[T,U,W where T in sints, U in uints, W in iorspair[S,U]) ...
```

where the last condition expands to

```
typematch W with
| S * S => 1
| U * U => 1
| _ => 0
endmatch
```

1.6.6 Typecase

A typecase expression computes a value based on a type match:

```
fun f[T:GENERIC] (x:T) =>
  typecase T with
  | int => "int"
  | double => 4.2
  | _ => "notnum"
  endmatch
;

println$ f 1;
println$ f 3.4;
println$ f "bad";
```

Type case expressions are only useful in generic functions.

1.6.7 Typeof operator

Just as typecase maps from types to values (more precisely type expressions to value expressions), the typeof operator does the opposite: it maps from expressions to types. For example:

```
fun f(x) = {
  var y : typeof x;
  y = x;
  return x,y;
}
println$ f "hello";
println$ f 42;
```

1.6.8 Primitive Types

Felix provides almost no primitive types. Instead, it provides two orthogonal facilities.

First, it provides a way to lift, or bind, types from C or C++. Second, it provides type constructors to manufacture new types from other types.

Type Bindings

Any first class type defined in C++ can be lifted into Felix with a type binding statement:

```
type int = "int";

type string = "::std::basic_string<char>"
   requires Cxx_headers::string
;

type vector[T] = "::std::vector<?1>"
  requires Cxx_headers::vector
;

type fred = "fred"
  requires header """
    struct fred { int x; };
  """
;
```

Templates with type parameters can be lifted by using the special encoding of a question mark followed by a single digit which refers to the n'th type parameter in the Felix type: the name given in the Felix type is a dummy and not used.

A type binding may provide a requires clause which specifies a floaing insertion which ensures the C++ code generated by the compiler inserts the required definition of the C++ type. Floating insertions are described in detail elsewhere.

Floating insertions which inject #includes for all C and C++ standard headers, and all Posix headers are provided in the library, split according to the ISO standard version.

The code of an insertion is emitted at most once, depending on whether the type is used or not.

Distinct Types

In the Felix standard library, most named typed from C, C++, and Posix, are distinct types, even if they're aliases in C/C++. This can be particularly annoying however the choice ensures proper separation of overloads based on these types, independently of platform dependent aliasing in C.

1.6.9 Abstract Types

Felix provides a class based mechanism for defining an abstract type represented by another type. Here is an except from the definition of *Darray*:

```
class Darray[T] {

  private struct darray_ctl[T]
  {
    a: varray[T];
```

(continues on next page)

(continued from previous page)

```
    resize: size * size --> size;
  }

  type darray[T] = new &darray_ctl[T];

  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));
  }


  ctor[T] darray[T] () =>
    _make_darray[T]$ new darray_ctl[T](varray[T] 20uz , dflt_resize);

  ...
}
```

An abstract type is defined in a class by its representation as a new type based on an old one by using the *type LHS = new RHS;* construction. The LHS type is the abstract type and the RHS is its representation. In this case the RHS is *darray_ctl* which is marked *private* to stop it escaping from the class.

The special name *_make_LHS* is used to define a constructor for the new type.

The special name *_repr_* is used to map the LHS type to the RHS type so it can be operated on to implement the functions which the user want to export.

The representation of the LHS cannot be accessed outside the class. In particular the *_make_LHS* and *_repr_* functions are private and can only be used inside the class. Thus, the implementation of LHS by RHS is hidden outside the class, in fact it is hidden inside the class as well, except by using the *_make_LHS* and *_repr_* functions.

1.6.10 Type Classes

Felix has Haskell style type classes which are used to provide a consistent notational system.

Class definition

A class is defined like this:

```
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;

  fun eq(x:t, y:t)=> x == y;
  fun ne(x:t, y:t)=> x != y;
  fun \ne(x:t, y:t)=> x != y;
  fun \neq(x:t, y:t)=> x != y;
}
```

This class defines the notion of an equivalence relation for some as yet unspecified type t. It does this by defining two virtual functions, == and != which can be overridden in instance specifications for particular types.

The == function is pure virtual and must be defined in an instance for the other functions to work. On the other hand the != function may be overridden, but if it is not the specified default will be used.

The non-virtual functions `eq, ne, \ne, \neq` cannot be overridden and are defined in terms of the virtuals.

This class also specifies axioms which are used to partly define the required semantics.

Instance definition

An instance of our class can be specified like this:

```
instance Eq[int] {
  fun == : int * int -> bool = "$1==$2";
}
```

Here, we have defined equality for type `int` and this also effects the definition of the functions `!=, eq, ne, \ne, \neq` as well.

Usage

The functions in a class may be refered to by qualified name lookup:

```
println$ Eq::== (1,2);
println$ Eq::eq (1,2);
```

Alternatively, you can pass the class dictionary in to a function or procedure like this:

```
fun eq3[T with Eq[T]] (a:T, b:T, c:T) => a == b and b == c;
println$ eq3 (42, 42, 42);
```

The print statement will work because the functions in the Eq class are in scope inside the eq3 function, and there is an instance for type int.

Another way to make the class methods available is to open the class:

```
open Eq[int];
println$ 42 == 42 and 42 == 42;
```

Here the class was opened only for type int.

Resolution

Use of class methods is handled in two phases. In the first phase the methods are chosen by the normal overload resolution process based on access to the class, and completely ignoring whether or not appropriate instances are available.

The second phase occurs during monomorphisation, when references to virtual functions are replaced by their overloads from the most specialised instance. An error diagnostic will be printed and the compilation terminated if no such instance exists.

Virtual types

A class may also specify a virtual type. This is an existential type which is dependent on the class type parameters in way which cannot be specified by a formula. Instead, the type is specified in each instance. Type parameters are deduced from applications to select a class virtual function, a virtual type is used in the result:

```
class Add[T1, T2] {
   virtual type U;
   virtual fun promote: T1 * T1-> U;
}

instance Add[int, long] {
   typedef U = long;
   fun add(x:int,y:long):long = "$1+$2";
}
```

Since a virtual type is only known at monomorphisation type a value of that type can only be used in the argument of a function which is parametrically polymorphic or a virtual class method.

1.6.11 Overloading System

Felix provides two forms of overloading: ad-hoc overloading is similar to C++, whereas type class based overloading is a two stage process where functions are first bound as if parametrically polymorphic, and then the best instance is selected by a similar algorithm.

Unification

The primary stage of overload resolution works by examining the set of functions with the required name which are visible at the point of call. Felix uses a technical variant of the unification algorithm which first calculates if the argument type is a specialisation of the parameter, if so, retaining the most general unifier (MGU).

Specialisation is the same as unification except that the type variables of the parameter must be disjoint from those in the argument, which is ensured by alpha-conversion is necessary. The parameter's type variables form the dependent set, and those of the argument the independent set. The MGU must contain assignments to each of the variables of the dependent set. A specialisation of the function is then calculated by replacing the dependent type variables with the value assigned to them.

If, from the set of available function signatures, there is more than one match, then the algorithm tries to find the most specialised. It does this by removing from the set any signature less specialised than some other.

If there are two or more signatures left at the end of this phase, two further steps are followed to resolve the ambiguity.

First, if the function is a higher order function (HOF) called with several arguments, the next argument is considered: functions which do not have a second argument are discarded and those for which there is a non-matching second argument are also rejected, of those that remain, we repeat the primary unification step. In practice, this is done simultaneously.

Constraints

Next, of the remaining functions, constraints are applied to reduce the set of candidates further. At this point if two or more functions remain the call is deemed ambiguous. If there is exactly one remaining function that is the selected function.

Outer scopes

Finally if there are now available functions, the overload resolution fails, however this is not the end of the story. Felix has several ways to continue. The primary recovery mechanism is to consider functions in the next outermost scope. The overload resolution process is repeated until we either find as match, find an ambiguity, or run out of scopes by

reaching the top level scope. Note that this is *different* to C++ in that functions are visible from outer scope without the need to inject them.

If we run out of scopes, the lookup machinery has a number of specialised ways to try again which are beyond the scope of this description to elaborate, however one is to prefix *_ctor_* to the name of the function and try again. This means if you apply a type to a value, the primary overload fails (because a type with that name is found, instead of a function so the set of candidates is empty). The effect of this rule is to search again for conversion functions named as *ctor* in the user code.

Another method for a retry is to consider the parameter names, if the function is called with a record. Felix also supports default arguments.

Type Class virtual instantiation

Type class virtual functions are instantiated purely with unification, the tricks, constraints, and multi-level scoping considerations do not apply here: it is a purely a matter of specialisation.

Subtyping

When the unification engine allows an argument with a type which is a subtype of the parameter, the mismatch is detected by the lookup machinery and a coercion is inserted for *each* of the functions in the overload resolution candidate set. Because this is done first, an exact match and a match obtained by subtyping are considered equal matches and this results in an ambiguity. Unlike C++, at this time Felix has no way to weight matches based on "how much subtyping" is done although the resolution would be precisely the same as the primary method for selecting the "most specialised" for structural types, there is no clear way to do this for nominal types, however.

1.6.12 Uniqueness Types

Felix provides a special type constructor *_uniq* which can be used to indicate ownership of a value, typically a pointer. Here is a synopsis of the user interface:

```
open class Unique
{
  // box up a value as a unique thing
  fun box[T] : T -> _uniq T = "($t)";

  // unsafely unpack the unique box
  fun unbox[T] : _uniq T -> T = "($t)";

  // kill a live unique value
  proc kill[T] : uniq T = ";";

  // functor for typing
  typedef fun uniq (T:TYPE):TYPE => _uniq T;

  // peek inside the box without changing livenes state
  fun peek[T] : &<(uniq T) -> T = "*($t)";
}
```

The intention is as follows: a function, say *f*, can specify a uniquely typed argument so that it can claim exclusive ownership of the value. Since all others are now excluded from access to the value, the function can then modify the value without the mutation being considered a side-effect, thus preserving referential transparency.

The function can then pass the modified value on "as if" it had made a local copy which it would necessarily have unique access to, and modified that value. The copy would be required so accesses external to the value continued to see the original value, however since such accesses are disallowed, the original can safely be modified.

The client of the function cannot pass a plain value to the function because that would be a type error: instead the client has to box the value to make it acceptable to the receiver function. By doing this the programmer is forced by the type system to become aware that the client must relinquish access to the value after calling the function.

Felix does not enforce safety in that the client may box a copy of a value and then access the original.

The client on the other hand cannot use the value until it is removed from its box, so the client programmer again is forced to take a specific action to take responsibility for correct handling.

In other words, the uniqueness typing system enforces a contract in which both sender and receiver must take positive action to comply with the transfer.

Felix provides a limited first order guarrantee that transfer of ownership of values stored in variables respects move semantics. First order means the assurance does not extend to indirections, either via pointers, or via closures. The limitation is that the checks may not work correctly in abnormal circumstances. For example, a uniq value must normally be either killed or ownership transfered, however in case of a program exit Felix may not detect a failure to kill a unique value.

The check Felix does is based on static flow control analysis. The general rule is that if a variable has a unique type, then if the variable is dead it can be written which livens it, if it is live then it must be read exactly once, which kills it.

Felix ensures that if a live variable is moved to one branch of a conditional or pattern match, it is moved to all of them (except in case of a match failure which will abort the program).

Dually, Felix ensures that if a variable is assigned to a component of a variable of product type, that component is treated "as if" it were a whole variable.

A more detailed discussion with examples can be found here: https://modern-computing.readthedocs.io/en/latest/unique.html.

1.7 Core Algebras

Contents:

1.7.1 Product Types

Contents:

Product Specification

See also Product_wiki.

Tuples

Constructors

```
// unit tuple constructor
satom := "(" ")"
```

(continues on next page)

(continued from previous page)

```
//$ Tuple formation non-associative.
x[stuple_pri] := x[>stuple_pri] ( "," x[>stuple_pri])+

//$ Tuple formation by prepend: right associative
x[stuple_cons_pri] := x[>stuple_cons_pri] ",," x[stuple_cons_pri]

//$ Tuple formation by append: left associative
x[stuple_cons_pri] := x[stuple_cons_pri] "<,,>" x[>stuple_cons_pri]
```

There are four basic constructors.

Unit Tuple

The syntax () specifies a canonical unit tuple, a product with no components.

```
()
```

Tuple of one component

A tuple of one component is identical to that component. The projection for that tuple is the identity function.

Chained comma

A list of expressions separated by commas forms a tuple. The comma operator is said to be a chain operator. The operation is non-associative.

Parentheses may be used to indicate grouping.

```
1,"hello",42.1
(1,"hello"), 42.1
1,("hello", 42.1)
```

All three of these tuples are disinct.

Prepend operator

The *cons* form of a tuple uses the right associative binary operator ,, to prepend a value to an existing tuple of at least two components.

```
var x = "a",,1,,"hello",42,1;
var y = "joy",,x;;
```

Note that the first case must use a , because the right hand term of the ,, operator must be a tuple. Although *()* is a tuple, *1,()* is equal to *1* and is not.

Append operator

Finally the left associative <,,> operator appends a value to an existing tuple. It is provided for symmetry with ,, but the syntax is arcane.

```
var x = 1,"hello"<,,>42.4;
var y = x<,,>"bye";
```

The left term of the append operator must be a tuple with at least two components.

Addition operator

The left associative infix binary addition operator + can also be used to append a value to a tuple. It has the same semantics as the <„> operator but a different precedence. Its use can be confusing sometimes, as it can be mistaken of integer addition.

```
var x = "Hello",1;

println$ x + x; // ("Hello", 1, ("Hello", 1))
println$ x + 1; // ("Hello", 1, 1)
println$ 1 + x; // (1, ("Hello", 1))
```

Extend operation

The *extend..with..end* operator packs a list of tuples or values into a tuple. The initial values are separated by commas, then a *with* keyword is used, then the remaining values are given, terminated by the *end* keyword:

```
var y = extend (1,2), "hello", (42.2,("bye",99)) with (55,"hh") end;
// (1, 2, hello, 42.2, (bye, 99), 55, hh)

var z = "hello",22;
println$ extend z with 34 end;
// ("hello", 22, 34)
```

Note that extend is expanded before monomorphisation, so a type variable will be treated as a single value, even if it is later replaced by a pair: had a pair been given both values would be in the result, instead of a single pair.

Types

```
//$ Tuple type, non associative
x[sproduct_pri] := x[>sproduct_pri] ("*" x[>sproduct_pri])+

//$ Tuple type, right associative
x[spower_pri] := x[ssuperscript_pri] "**" x[sprefixed_pri]

//$ Tuple type, left associative
x[spower_pri] := x[ssuperscript_pri] "<**>" x[sprefixed_pri]

//$ Array type
x[ssuperscript_pri] := x[ssuperscript_pri] "^" x[srefr_pri]
```

The first three types are alternate ways to express a tuple type. The different forms are significant with polymorphism.

For example consider the following code which performs a lexicographic equality test on any tuple:

```
class Eq[T] { virtual fun == : T * T -> bool }
```

(continues on next page)

(continued from previous page)

```
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 code uses polymorphic recursion via type class virtual function overloads to analyse a tuple like a list by using the tuple Cons operator **.

There are two ground cases given, the first one checks for a pair to terminate the recursion, the second is more specialised and checks for a pair of the same type, that is, an array of two elements.

Value Projections

```
x[scase_literal_pri] := "proj" sinteger "of" x[ssum_pri]
```

Projection functions for a given tuple type can be written. Projections are first class functions, like any other. The projection index must be a literal decimal integer between 0 and n-1, inclusive, where n is the number of components of the tuple.

```
var x = 1,"hello",42;
var p = proj 1 of (int * string * int);
println$ p x; // "hello"
println$ x.p; // "hello"
```

Pointer Projections

For every value projection, there is a corresponding pointer projection, represented as an overloaded function. That is, you can use the same syntax for projections on pointers as values.

```
var x = 1,"hello",42;
var p = proj 1 of &(int * string * int);
println$ *(p &x); // "hello"
println$ *(&x.p); // "hello"
println$ *(x&.p); // "hello"
var wop = proj 1 of &>(int * string * int);
&>x . wop <- "bye";
var rop = proj 1 of &<(int * string * int);
println$ *(&<x . rop);
```

Note the special sugar *x&.p* which is equivalent to *&x.p*.

It's important to note that the application of projections to pointers as well as values *solves a major problem in C++* by eliminating entirely any need for the concept of lvalues and reference types. Pointers are first class values and the calculus illustrated above forms a coherent algebra which cleanly distinguishes purely functional values, but, via pointers, provides the same algebraic model for imperative code.

The concept of a pointer cleanly distinguishes a value from a mutable object. In particular

- all values are immutable, but
- all products are mutable and their components separately mutable, if you can obtain a pointer to the value type.

There are three basic ways to do this:

- store the value in a variable and takes its address, or,
- copy the value onto the heap with operator *new* which returns a pointer.
- Library functions can also provide pointers.

The fundamental calculus of projections is just ordinary functional calculus. This is the point! In particular composition of pointer projections is equivalent to adding the offsets of components in a nested product. For example:

```
var x = (1,(2,3));
var p1o = proj 1 of (&(int * int^2));
var p1i = proj 1 of (&(int^2));
var p = p1o \odot p1i; // reverse composition
println$ *(&x.p); // 3
```

The address calculations are purely functional and referentially transparent.

Projection Applications

There is a short cut syntax for applying a projection to a tuple, you can just apply an integer literal directly:

```
var x = 1,"hello",42;
println$ 0 x, x.1;
```

Note that since operator dot . just means reverse application, then *x.1* is the same as *1 x*.

Slice Value Projections

A slice with integer literal arguments can be applied to a tuple to construct a new tuple consisting of the components in range of the slice. The components selected are in the intersection of the given slice and a slice from 0 to the length of the tuple minus 1. No error is possible.

```
var x = 1,4.2,"hello",42u;
println$ x. Slice_all;       // (1, 4.2, hello, 42)
println$ x. (..);            // (1, 4.2, hello, 42)
println$ x. (1..);           //  (4.2, hello, 42)
println$ x. (..3);           // (1, 4.2, hello, 42)
println$ x. (1..3);          // (4.2, hello, 42)
println$ x. (1..<3);         // (4.2, hello)
println$ x. (1.+2);          // (4.2, hello)
println$ x. (3..0);          // ()
println$ x. Slice_none;      // ()
println$ x. (Slice_one 2);   // "hello"
```

Ties and Slice Pointer Projections

Felix has a generic operator *_tie* which takes a pointer to a structurally typed product and produces an isomorphic product type, where the type of each component becomes a pointer to the component type. This is called a tie:

```
var x = 1,4.2,"hello",42u;
var px = _tie &x; // a tuple of pointers
println$ *(px . 2); // hello
```

The same effect can be obtained for tuples, with a pointer slice:

```
var x = 1,4.2,"hello",42u;
var px = &x . (..); // a tuple of pointers
println$ *(px . 2); // hello
```

Of course, any slice can be used, and, it also works for read only and write only pointers.

[And should if the product is a compact linear type or array, neither of which is implemented as at 2018/08/25]

Reversed Tuple

For any tuple the generic operator *_rev* forms a tuple with the components in reversed order.

```
var x = 1,4.2,"hello",42u;
println$ _rev x; // (42, hello, 4.2, 1)
```

Tuple Patterns

```
//$ Tuple pattern match right associative
stuple_pattern := scoercive_pattern ("," scoercive_pattern )*

//$ Tuple pattern match non-associative
stuple_cons_pattern := stuple_pattern ",," stuple_cons_pattern

//$ Tuple pattern match left associative
stuple_cons_pattern := stuple_pattern "<,,>" stuple_cons_pattern

//$ Tuple projection function.
x[scase_literal_pri] := "proj" sinteger "of" x[ssum_pri]
```

Tuple patterns are an advanced kind of tuple accessor.

```
match 0,1,(2,3,(4,5,6),7,8) with
|  _,x1,(x2,_,(x4,,x56),,x78 =>
   // x1=1, x2=2, x4=4, x56=(5,6), x78=(7,8)
   ...
endmatch
```

Tuple patterns are *irrefutable*, that is, they cannot fail to match if they type check, provided subcomponent matches are also irrefutable. For this reason they are often used in *let* form matches which only admit one branch syntactically:

```
let _,x1,(x2,_,(x4,,x56),,x78 =
  0,1,(2,3,(4,5,6),7,8)
in
```

(continues on next page)

(continued from previous page)

```
  // x1=1, x2=2, x4=4, x56=(5,6), x78=(7,8)
  ...
```

If all the components of a tuple have the same type, then the tuple is called an array. Perhaps more precisely, a fixed length array where the length is fixed at compile time. The jargon *farray* is sometimes used to be specific about this kind of array.

An alternate more compact type annotation is available for arrays:

```
var x : int ^ 4 = 1,2,3,4;
```

In addition, arrays allow an expression for the shortcut form of projections applications, as well as decimal integer literals. Two types may be used for an array index:

```
var x : int ^ 4 = 1,2,3,4;
for i in 0..<4 perform println$ x.i;
println$ x.(`1:4);
```

The index of an array, in this case *4* is not an integer, it is a sum of 4 units, representing 4 cases. Therefore the correct projection should be of type 4, however Felix allows an integral type, which is coerced to type 4.

See the section on *sum types* for more information on unit sums.

Array Projections

Array projections are similar to non-array projections except that the projection index can be an expression. The keyword *aproj* must be used for an array projection, and the indexing type must be precisely the type of the array exponent.

```
var x = 1,2,3,4;
var n = `2:4; // index
var px = aproj n of (int ^ 4);
println$ x.px;
```

A direct application may also use a one of two shortcut forms:

```
var x = 1,2,3,4;
var n = `2:4;   // precise index
println$ x . n; // precise shortcut projection
var m = 2;      // integer index
println$ x . m; // checked shortcut
```

The integer form requires a run time bounds check.

Generalised Arrays

By virtue of the existence of compact linear types and coercions representing isomorphisms on them, Felix supports a notion of generalised arrays. In particular, the structure of an array does not have to be linear.

For example:

```
var x : (int ^ 3) ^ 2 = ((1,2,3),(4,5,6));
var y : int ^ (2 * 3) = x :>> (int ^ (2 * 3));
var z = x :>> (int ^ 6);
```

(continues on next page)

(continued from previous page)

```
for i in 0..<2
  for j in 0..<3 do
    println$ x.i.j;
    println$ y.(i:>>2,j:>>3);
    println z.(i * 3 + j);
  done
```

Note the unfortunate requirement to coerce the integer indices to the precisely correct type. [To be fixed]

In this example, *x* is an array of arrays, however *y* is a *matrix*: the index of the matrix is not a single linear value but rather, the index is a tuple.

The coercion used to convert type *x* to *y* is an isomorphism. Underneath in *both* cases we have a linear array of 6 elements, *z*.

The coercions on the arrays above are called *reshaping* operations. They are casts which *reconsider*, or *reinterpret* the underlying linear array as a different type.

Note: in the matrix form, Felix has chosen the indexing tuple to be of type *2 * 3* so that a reverse application can be thought of as first selecting one of the two subarrays, then selecting one of the three elements. In other words, you write the *i* and *j* indices in the array of arrays form and matrix form in the same order when using reverse application. However the order differs if you use forward application:

```
for i in 0..<2
  for j in 0..<3 do
    println$ j (i x);
    println$ (i:>>2,j:>>3) y;
  done
```

The choice of ordering is arbitrary and confused by the fact that numbers are written in *big-endian* form which tuple indices are written in *little-endian* form. The use of big-endian numbers is unnatural in western culture where script is written from left to right, we should be using little-endian. However our number system is derived from tha Arabic, which is written right to left, so in that script, numbers put the least significant digits first.

Consequently there is a natural ordering conflict, since our numbers are backwards from our ordering of array elements. Take care with, for example, square matrices where the type system cannot detect an incorrect ordering. Take evem more care with coercions, since they override the type system!

Polyadic Array Handling

Because of the reshaping isomorphisms, it is possible to write a single *rank independent* routine which performs some action on a linear array which can be applied to an array of any shape. All you need to do is coerce the generalised array argument to a suitable isomorphic linear form, apply the routine, and cast the resulting linear array back.

For more details please see *Compact Linear Type*.

Records

Syntax

```
//$ Record type.
satom := "(" srecord_mem_decl ("," srecord_mem_decl2)*  ")"
  srecord_mem_decl := sname ":" stypeexpr
  srecord_mem_decl := ":" stypeexpr
```

(continues on next page)

(continued from previous page)

```
  srecord_mem_decl2 := sname ":" stypeexpr
  srecord_mem_decl2 := ":" stypeexpr
  srecord_mem_decl2 := stypeexpr

//$ record value (comma separated).
satom := "(" rassign ("," rassign2 )* ")"
  rassign := sname "=" x[sor_condition_pri]
  rassign := "=" x[sor_condition_pri]
  rassign2 := sname "=" x[sor_condition_pri]
  rassign2 := "=" x[sor_condition_pri]
  rassign2 := x[sor_condition_pri]

satom := "(" sexpr "without" sname+ ")"

satom := "(" sexpr "with" rassign ("," rassign2 )* ")"
```

Felix allows you to create record values on the fly like this:

```
var x = (a=1, b="hello", c=4.2);
println$ x._strr;
```

The builtin function *_strr* can be used to translate a record value to a string.

The record value *x* we specified above has the record type:

```
(a:int, b:string, c:double)
```

Fields

The fields of a record can be accessed by name:

```
var x = (a=1, b="hello", c=4.2);
println$ x.a;
println$ a x;
```

The field name is used as a projection function, it is a first class function. If two records have the same field name, the function is selected by the usual overloading process:

```
var x = (a=1, b="hello", c=4.2);
var y = (b=42, d=99);
println$ x.b; // selects the string
println$ b y; // selects the integer
var prj = b of (b:int, d:int);
println$ prj y;
```

Duplicate Fields

Records may have duplicate fields:

```
var x = (a=1, a="hello",a=42);
println$
```

In this case the first or leftmost field is always selected when the field name is used as a projection.

Duplicate fields arise naturally as a result of some record operations.

Blank Field Names

The name of a field can be blank. There are three ways to indicate a blank name in a record value:

```
var x = (n""=1, ="hello", 42, y=1);
```

The first method uses an explicit identifier literal to construct an identifer from an empty string. The second form simply omits the identifier but keeps the = sign. The third form omits the identifier and the = sign. All three methods are equivalent, however the third form is not permitted at the begining of a record value. A similar syntax may be used for record types:

```
typedef x_t = (n"":int, string, :int, y:int);
```

Accessing blank fields

Felix allows blank record fields to be access by number.

```
var x = (n""=1, ="hello", 42, y=1);
println$ x.1; // "hello"
```

Record with all blank fields

If all the fields of a record have blank names, then the record is a tuple. Therefore tuples are just a special case of records, and since arrays are a special case of tuples, arrays are also a special case of records.

Record Addition

Two records can be concatenated by using the infix + operator:

```
var x = (a=1,b=2) + (a=3, d=3);
```

As usual if there is a duplicate field, the left field hides any fields to its right with the same name.

Functional Update

A record can be updated using functional update syntax:

```
var x = (a=1, b=2, c=3);
var u = (x with b=42, c=99);
```

The result is a new record with the values of the specified fields replaced. Only the first field of a duplicate set can be updated. The field must exist, and must have the same type.

Dropping Fields

A record can also be updated by removing fields:

```
var x = (a=1, b=2, c=3, c=99);
var u = (x without c c b);
```

More than one field can be removed by listing the field names without a separating comma. If a field is duplicated only the leftmost field is removed, the next field can be removed by giving the same name again.

Adding Fields

Fields can be added on the left with *polyrecord* syntax:

```
var x = (a=1,b=2);
var y = (c=1,a=66 | x);
```

Polyrecords are a separate advanced topic discussed under the topic *row polymorphism*.

PolyRecords

Polyrecords are an extension of a record type designed to support row polymorphism.

Syntax

```
//$ polyRecord type.
satom := "(" srecord_mem_decl ("," srecord_mem_decl2)*  "|" stypeexpr ")" =>#
 "`(ast_polyrecord_type ,(cons _2 (map second _3)) ,_5)";

//$ polyrecord value
//$ record value (comma separated).
satom := "(" rassign ("," rassign2 )* "|" sexpr ")"
```

Description

A polyrecord is an extension of the record concept which allows a product type to be extended by adding new fields on the left:

```
typedef D2 = (x:double, y:double);
typedef D3 = (z:double | D2);

var a = (x=1.1, y=2.2);
var b = (z=5.1 | b);
```

The expression after the vertical bar in a polyrecord expression must have tuple or record type. The compiler actually allows any type during type checking, so in principle any value can be tagged with an attribute. However the code generation has no support for representations other than tuples or records.

The primary purpose of polyrecords is to support row polymorphism.

Structures

Struct Definitions

Syntax

```
stmt := "struct" sdeclname "=" ? "{" sstruct_mem_decl * "}"
stmt := "export" "struct" sdeclname "=" ? "{" sstruct_mem_decl * "}"
  sstruct_mem_decl := sname ":" stypeexpr ";"
  sstruct_mem_decl := stypeexpr sname ";" // C hack
  sstruct_mem_decl := sfunction
```

Description

Structs are nominally typed records:

```
struct X {
  a: int;
  b: int;
}
var x = X (1,2);
```

Every struct is equipped with a constructor which is named after the struct, and which accepts a tuple consisting of arguments of the types defined in the struct, in order of writing: these are used to initialise the corresponding fields.

Structs are products like records, so the field names can be used as both value projections and pointer projections:

```
var a = x.a; // value
var pb = &x.b;
```

A special syntax is allowed for structs to make it a bit easier to import specifications from C:

```
struct X {
  int a;
  &long b;
}
```

In this form the type goes first, as in C. However note that the type specification is a Felix type, so we have to use *&long* for a pointer to *long* rather than *long** as in C.

Structs may contain function and procedure definitions:

```
struct X {
  int x;
  fun get => self.x;
  fun getter () self.x;
  proc set (a:int) { self.x <- a; }
}
```

Such definitions are *not* methods but ordinary functions with an extra *mgic* argument so the above is equivalent to:

```
struct X {
  int x;
}
fun get (self: X) => self.x;
fun getter (self: X) () => self.x;
proc set (self: &X) (a:int) { self.x <- a; }
```

Note that the argument to a function is a value of the structure type named *self*. For a procedure, the magic argument is a *pointer* instead. The *magic* argument is added automatically so in the original struct definition the function *get* appears to have no argument.

Because contained functions and procedure have the hidden *magic* argument, closures can be formed if there are other arguments:

```
var a = X(1);
var asetter = a.set;
asetter 42;
var agetter = a.getter;
println$ agetter();
```

Note in the example we cannot make a closure for *get* since it has only one argument.

Cstruct Definitions

Syntax

```
stmt := "cstruct" sdeclname "=" ? "{" sstruct_mem_decl * "}" srequires_clause ";"

//$ A hack to help with cut and paste from C headers into Felix
stmt := "typedef" "struct" "{" sstruct_mem_decl * "}" sdeclname srequires_clause ";"
;

//$ A hack to help with cut and paste from C headers into Felix
stmt := "typedef" "struct" sdeclname "{" sstruct_mem_decl * "}" sdeclname srequires_
↪clause ";"
```

A cstruct definition is similar to a struct definition, except no C++ code is emitted for the type definition. Instead, a C struct or union of the same name must be defined, typically in a C include file.

1.7.2 Coproduct Types

Contents:

Coproduct Specification

https://en.wikipedia.org/wiki/Coproduct

Anonymous Sum Types

Sum types are a positionally accessed structural type dual to tuples. They're used like this:

```
typedef integ = short + int + long;
fun show (x:integ) =>
  match x with
  | `0 s => "short="+ s.str
  | `1 i => "int="+ i.str
  | `2 l => "long="+ l.str
  endmatch
;
var x = `1:integ 42;
println$ "Value " + show x;
```

With sums, the cases are numbered from 0 up. To specify a value, the constructor index, sum type, and any argument must be given.

Unitsums

A special case of sum types is a sum of units. Recall the type of an empty tuple is designated as 1 or unit. Then a unit sum has a type like this:

```
typedef three = unit + unit + int;
typedef three = 1 + 1 + 1;
typedef three = 3;
```

All these forms are equivalent. An integer given as a type is takens to be a sum of that many units.

Apart from 1, which is a unit sum of 1 unit, there are two other important unit sums named in the library:

```
typedef void = 0;
typedef bool = 2;
```

The type void is a type with no values. The type bool is a type with two values. The names false and true are synonyms for ‘0:2 and ‘1:2.

Unit sums are important because they’re used as array indices. For example an array of 4 ints has the type

```
int ^ 4
```

The 4 there is not an integer, but the type 4.

Sum types are not used very often because remembering cases by number is hard.

Polymorphic Variants

Polymorphic variants are a kind of open union type. A value is formed by using a globally unique name preceded by a backquote character and followed by a value:

```
`Some 42;
```

This variant has the type

```
`Some of int
```

A polymorphic variant type is given by a set of unique constructor names and types:

```
typedef option[T] = (
  | `Some of int
  | `None
);
```

and such types obey both width and depth subtyping rules.

Width subtyping says that a type A with a subset of the constructors of a type P is a subtype of P, depth subtyping extends the rule to also allow the arguments of A’s constructors to be subtypes of those of P. In other words, subtyping is covariant.

Felix applies subtyping rules automatically when applying a function or procedure to a value:

```
proc show[T with Str[T]] (a: option[T]) {
   match a with
   | `Some v => println$ "Some " + v.str;
   | `None => println$ "None"
```

(continues on next page)

(continued from previous page)

```
   endmatch;
}
show (`Some 42);
```

However a coercion is required for assigments, including initialisations:

```
var x = `Some 42 :>> option[int];
```

Since pattern variables are also set by initialisation, a special form of pattern is required to specify the variable type: because you can consider a pattern an "inside-out" form of code, the pattern is written as a backwards coercion, with the type first. This is illustrated in the next example.

Polymorphic variants are weaker than unions in that they offer less safety guarantees because the set of constructors and their arguments are open. However this allows extremely powerful but reasonably well constrainted extension models, demonstrated in the example below which uses the technique known as open recursion:

```
typedef addable' [T] = (
 | `Val of int
 | `Add of T * T
 )
;

fun show'[T] (show: T->string) (x: addable'[T]) =>
 match x with
 | `Val q => "Val " + q._strr
 | `Add (a,b) => show a + " + " + show b
;

typedef addable = addable'[addable];
fun show(x:addable): string => show' show x;

var x = `Add (`Val 1, `Val 2);
println$ show x;

typedef subable' [T] = (
| addable'[T]
| `Sub of T * T
);

fun show2'[T] (show2: T->string) (x:subable'[T]) =>
  match x with
  | `Sub (a,b) => show2 a + " - " + show2 b
  | (addable'[T] :>> y) => show'[T] show2 y
;

typedef subable = subable'[subable];
fun show2 (x:subable): string => show2' show2 x;

var y = `Add (`Sub (`Val 1, `Val 2), `Val 3);
println$ show2 x; // <============
println$ show2 y;
```

In this example we define first a type of expression which allows just values and addition. We provide a function to show such expressions as a string. We do this by first providing a type with an unspecified parameter, and then use a fixpoint operation (self-reference) to bind the parameter to the type. The pattern for the function is similar: we first provide an open function which takes an argument function to show the value of the type of the parameter, and then fix the function to just addable types by applying the function to itself.

Then, to extend the system to work with subtraction as well, we define a new type by adding a subtraction term to the open form of the addition term, and then again fixate the type. Similarly, the open form of the show function handles the new subtraction term itself but delegates to the open form of the function handling addition. Then we fixate that function by applying it to itself to obtain a closed function.

The power of this method is seen in the line indication with a commented arrow: the closed show2 function which works with expressions containing subtractions also works with the older, legacy expressions containing only addition.

It works because of covariant subtyping: the closed terms with addition are subtypes of the closed terms that also include subtraction.

The vital importance of this technique cannot be overstated. Unlike object orientation, which requires methods to have contravariant argument types, open recursion is covariant. It therefore supports Meyer's Open/Closed principle whilst, despite his intentions, object orientation does not.

Unions

Basic Unions

Unions are a nominal type which is used to specified alternatives.

```
union Maybe[T] {
| Nothing
| Just of T
}

fun show[T] (x:Maybe[T]) =>
  match x with
  | Nothing => "Nothing"
  | Just v => v.str
  endmatch
;

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

The fields of a union are usually called constructors. Constructors may be either constant constructors like Nothing above, or non-constant constructors like Just, which take an argument.

Pattern matching is used as shown to decode a value of union type. Matches consist of an argument and a list of branches. Each branch contains a pattern and a handler.

In the Some branch the pattern variable v is set to 1 in the example, and converted to a string in the handler expression.

Pattern matching selects the first branch with a pattern that matches and evaluates the corresponding handler.

Generalised Algebraic Data Types

Felix also provides generalised algebraic data types, or GADTS: A GADT is a polymorphc union with a per constructor existential constraint on the type variable.

```
union pair[T] =
| PUnit of unit => pair[unit]
| Pair[T,U] of U * pair[T] => pair[U * pair[T]]
;
```

(continues on next page)

(continued from previous page)

```
var x1 = #PUnit[unit];
var x2 = Pair (22,x1);
var x3 = Pair (99.76,x2);

fun f[T:GENERIC] (x:T) = {
  match x with
  | Pair (a,b) => return a.str + ","+b.f;
  | PUnit => return "UNIT";
  endmatch;
}

println$ f x3;
```

With a GADT, some components may have a RHS after the => symbol which must be the union type subscripted with a constraint on the type variables: in the example the *PUnit* constructor returns a *pair* with parameter T constrained to type unit, whereas the *Pair* constructor introduces and existential parameter T, and returns a *pair* with type argument *pair[U * pair[T]]*.

This particular construction can be used to recursively define a heterogenous list similar to a system tuple type, but amenable to recursive analysis by a generic function such as f above. The analysis requires polymorphic recursion which Felix does not support directly but in this case can be emulated by a generic function which is expanded by the compiler to a nest of specialised functions.

1.7.3 Compact Linear Types

Variadic Positional Number Systems

Let a be any non-negative number and let d be any positive number, then there are unique values q and r such that

$$a = r + qd + \quad \text{such that } q \geq 0 \text{ and } r < d$$

We define integer division:

$$a \operatorname{div} d = q$$

and remainder

$$a \operatorname{rmd} d = r$$

Note that in C, `a/b` where the dividend and quotient are integral is integer division and `a%b` is the remainder.

Let

$$c_0, c_1, ..., c_{n-1}$$

be finite sequence of positive integers called *radices*, and let

$$v_0, v_1, ..., v_{n-1}$$

be a sequence of non-negative integers called *coefficients* such that

$$v_i < c_i \quad \text{for } i \in 0..n-1$$

Let

$$z_0 = 1$$
$$z_i = c_i z_{i-1} = \sum_{j=0}^{i-1} c_j \text{ for } i \in 1..n-1$$

so that z_i is the product of all the c_j for $j < i$; these quantites are called *weights*.

Let

$$a = \prod_{i=0}^{n-1} v_i z_i$$

Then the total number of values a could take is z_{n-1} and the maximum is therefore $z_{n-1} - 1$.

This is known as a *variadic positional number system*. For any number in the range 0 to the maximum, the coefficients v_i are uniquely determined and there is a bijection between the integer values and the sequence of coefficients.

Given a value a we would like to be able to calculate the value v_i for a given i. Let us first rewrite the formula for a like this:

$$\begin{aligned} a &= \sum_{j=0}^{i-1} v_j z_j + v_i z_i + \sum_{k=i+1}^{n-1} v_k z_k \\ &= \underbrace{(\sum_{j=0}^{i-1} v_j z_j)}_{r} + \underbrace{(v_i + \sum_{k=i+1}^{n-1} v_k (z_k / z_i))}_{q} \underbrace{z_i}_{d} \end{aligned}$$

We note that this is of the required quotient and remainder form since the left term is clearly less than z_i, and, since z_i divides z_k exactly for $k \geq i$, so we can find

$$q = a \operatorname{div} z_i = v_i + \sum_{k=i+1}^{n-1} v_k (z_k / z_i)$$

But now we can rewrite that term as well:

$$\begin{aligned} q &= v_i + (\sum_{k=i+1}^{n-1} v_k (z_k / (z_i c_i))) c_i \\ &= v_i + (\sum_{k=i+1}^{n-1} v_k (z_k / z_{i+1})) c_i \end{aligned}$$

Again it is true by specification that $v_i < c_i$ and z_{i+1} divides z_k exactly for $k \geq i+1$ which is the lowest index of the sum, therefore since the equation has the required quotient and remainder form:

$$v_i = (a \operatorname{div} z_i) \operatorname{rmd} c_i$$

or in C:

```
vi = a / zi % ci
```

We remark that for powers of radicies which are powers of 2, the above formula reduces to a right shift followed by mask.

We are not finished though. Let us assume that v_i itself is suitably encoding a positional form using radices

$$c'_0, c'_1, ..., c'_{m-1}$$

with coefficents

$$v'_0, v'_1, ..., v_{m-1'}$$

so that again with

$$z'_h = \prod_{q=0}^{h-1} c'_q$$

we have

$$v_j = \sum_{h=0}^{m-1} v'_h z'_h$$

and we want to find the v'_g. Obviously we can just do this:

```
vg' = (a / zi % ci) / zg' % cg'
```

by using the same formula recursively. However that formula is not good because it uses 4 constants. Can we do it with just two, calculated from the four?

The method is simple: we just expand the series substituting the vj' in, the using the distributive and associative laws multiply the inner terms by z_g and we can see we just have a new positional number system. The working out is left as an exercise. The result is we can use this formula:

```
vg' = (a / zi * zg') % cg'
```

Compact Linear Types

You may wonder why we did the above calculations! In Felix, we define a compact linear type inductively as:

- unit
- any product of compact linear types
- any sum of compact linear types

Felix has special notation for sums of units. Unit can also be written as type 1. A sum of n units can be written as n:

```
unit = 1
2 = 1 + 1 // aka bool
3 = 1 + 1 + 1
...
```

These types are called *unitsums* because they're sums of units. Using the decimal representation is more convenient that the 1-ary representation. The type 2 is well known, it is called *bool*.

Values of unitsums are written with a zero origin case number and the type:

```
`0:1 // ()
`0:2 // false
`1:2 // true
`3:5 // case 3 of 5
...
```

Note again the unfortunate fact we use zero-origin case numbers which reads badly in natural language!

We can form products of unit sums:

```
var x : 3 * 4 * 5 = `1:3,`2:4,`3:5
```

for example. Now, with some luck, you might see this:

$$c0 = 3, c1 = 4, c2 = 5$$
$$v0 = 1, v1 = 2, v2 = 5$$

and immediately recognize nothing more difficult than a variadic positional number system! In fact this is precisely how Felix represents a compact linear type: as a single machine word holding an integer.

Value Projections

Projections for components of compact linear products use the same syntax as for non-compact products.

```
typedef p345_t = 3 * 4 * 5;
var x : p345_t = `1:3,`2:4,`3:5;
println$ x.1; // `2:4

var p = proj 1 of (p345_t);
println$ x.p;
```

You will now understand the C++ representation:

```
// compact linear type
typedef ::std::uint64_t cl_t;

// projection
struct RTL_EXTERN clprj_t
{
  cl_t divisor;
  cl_t modulus;
  clprj_t () : divisor(1), modulus(-1) {}
  clprj_t (cl_t d, cl_t m) : divisor (d), modulus (m) {}
};

// apply projection to value
inline cl_t apply (clprj_t prj, cl_t v) {
  return v / prj.divisor % prj.modulus;
}
```

The most important bit, however is this:

```
// reverse compose projections left \odot right
inline clprj_t rcompose (clprj_t left, clprj_t right) {
  return clprj_t (left.divisor * right.divisor, right.modulus);
}
```

Composing projections is how we get at components of nested tuples. Its most important that the composite of two projections is a projection, and the representation above satisfies that condition.

Pointers

As you know by now, by combining pointers with projection functions, we obtain a purely functional, referentially transparent mechanism for address calculations.

So you may wonder how we can get a pointer into a compact linear product since the value hidden is inside an integer and is not addressable.

The answer is seen by the C++ representation again:

```
struct RTL_EXTERN clptr_t
{
  cl_t *p;
  cl_t divisor;
  cl_t modulus;
  clptr_t () : p(0), divisor(1),modulus(-1) {}
  clptr_t (cl_t *_p, cl_t d, cl_t m) : p(_p), divisor(d),modulus(m) {}

  // upgrade from ordinary pointer
  clptr_t (cl_t *_p, cl_t siz) : p (_p), divisor(1), modulus(siz) {}
};
```

As you can see, a compact linear pointer uses three machine words. The first word *p* is just a pointer to the whole containing location, which is a machine word. But we also store a divisor and modulus value, which identifies how to find the component.

Here's how we get a value using the pointer:

```
// dereference
inline cl_t deref(clptr_t q) { return *q.p / q.divisor % q.modulus; }
```

To apply a projection to a pointer:

```
// apply projection to pointer
inline clptr_t applyprj (clptr_t cp, clprj_t d)  {
  return  clptr_t (cp.p, d.divisor * cp.divisor, d.modulus);
}
```

And more complicated to store a value in a component:

```
// storeat
inline void storeat (clptr_t q, cl_t v) {
    *q.p = *q.p - (*q.p / q.divisor % q.modulus) * q.divisor + v * q.divisor;
    //*q.p -= ((*q.p / q.divisor % q.modulus) - v) * q.divisor; //???
}
```

Here's an example in Felix, which translates to code using the C++ above (which is part of the Felix RTL):

```
var x = true,false,true;
var px = &x;      // ordinary pointer
var p1 = px . 1; // compact linear pointer
p1 <- true;       // store 1 bit
println$ x;       // true, true, true
println$ *p1;     // true

var prj = proj 1 of (&(2^3));
p1 = &x. prj;
p1 <- false;
println$ x;       // true, false, true
println$ *p1;     // false
```

Compact linear pointers have read-only and write-only variants too, which are supertypes of the read-write pointer, the same as for ordinary pointers.

Sum Types

The representation of sums is simple. Consider a sum

```
typedef s567 = 5 + 6 + 7;
```

We will be dealing with values of this type of the form:

```
(case 1 of s567) `3:6
```

The left term is an injection function which selects the second case (remember case numbers are zero origin) and encodes into the sum type a value of the second case type, which is 6.

Whatever the encoding is, we have to be able to extract the case number and the value, the value is called the argument because it is the argument of the injection function.

We will calculate how to do the encoding based on how we are going to do the decoding. What we will do is assign successive integer ranges to the cases:

```
case 0: 0..4    // size = 4 + 1 - 0 = 5
case 1: 5..10   // size = 10 + 1 - 5 = 6
case 2: 11..17  // size = 17 + 1 - 11 = 7
```

To encode a value, we just add the injection argument to the minium for that case.

The decode must reverse this process:

```
if v < 5 then case 0, value v
elif v  < 11 then case 1 value v - 5
elif v < 18 then case 2, value v - 11
```

The minimum value of a case is the sum of all the previous cases, or 0 if there aren't any. The maximum is just the mimimum plus the case size minus 1. The comparator in the chain above is the minimum of the next case, the last test is not required since it is the only remaining altenative.

The fastest way to perform the encoding is to take the sequence of types, considered as integers, and calculate an array of sums of all the previous values. The sum of previous values is called the *offset* of the case. The encoding then is just the sum of the value plus the offset of the case.

The decoding, unforunately, requires a loop, however we can reduce the number of calculations by repeated subtractions of successive bases. We start with the value to decode, setting the case number to 0. If the value is less than the base, we return the case number and use the value as the argument. Otherwise we increment the case number and subtract the size of the current base which is found from a lookup table, then repeat.

It is instructive to consider that the representation of a unit sum is simply the case number. There is only one possible argument, which is always 0.

In Felix, there are two closely related terms. The general term is an injection function. However if the injection takes a unit argument, it is a constant function and the result of applying the injection is known, so there is a second term which represents the value of constant injections, so we can elide their application.

The syntax using *inj n of type* is always an injection function. However the syntax *case n of type* is identical to the injection term of the injection is not a constant functions, otherwise, it is the value the injection would produce instead.

In particular the notation *case 1 of 2* or *'1:2* is the value *true* and not an injection function. It has type *2* whereas *inj 1 of 2* has type *1->2*.

Pointer type syntax

```
satom := "_pclt<" stypeexpr "," stypeexpr ">"
satom := "_rpclt<" stypeexpr "," stypeexpr ">"
satom := "_wpclt<" stypeexpr "," stypeexpr ">"
```

A pointer to a compact linear type *_pclt<D,C>* specifies a pointer to a component type *C* embedded in a complete compact linear type *D*, which occupies a machine word. This type is a subtype of the read-only pointer type *_rpclt<D,C>* and write only pointer type *_wpclt<D,C>*.

1.7.4 Pointer Types

Machine Pointers

A machine pointed is a typed machine address. Felix has three types of machine level pointers.

Type	Deref	Storeat	Semantics
&<T	Y	N	Read only pointer to T
&>T	N	Y	Write only pointer to T
&T	Y	Y	Read or write pointer to T

Read only pointers are semantically equivalent to C++ const pointers.

Operations

Dereference

The system builtin dereference operation is named `_deref` and is an operator equivalent to a function

```
fun _deref: &<T -> T;
```

The library defines an overloadable function of the same type:

```
fun deref[T] (p:&<T): T => _deref p;
```

which can also be invoked by the syntax:

```
*p
```

The dereference operation fetches the value of a storage location at the address of the pointer.

Storeat

The system buildin storage operation is named `_storeat` and is equivalent to a procedure

```
proc _storeat: &>T * T;
```

The library defines an overloadable procedure of the same type:

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

which can also be invoked by the syntax

```
p <- v;
```

The storeat operation stores its value argument into the storage location at the address of the pointer.

Subtyping Rules

The read-write pointer type is an invariant subtype of the read-only pointer type, and an invariant subtype of the write only pointer type.

In theory read is covariant and write contravariant but this is not implemented at the present time.

This means a read-write pointer may be used wherever either a read-only or write-only pointer is required.

Constructors

All three types of pointers can be constructed by addressing a variable.

```
var x = 1; // type int
var ropx : &<int = &<x;
var wopx : &>int = &>x;
var rwpx : &int = &x;
```

In addition, a read-write pointer is returned by the system intrinice operator `new` which copies a value onto the heap and returns a pointer to it:

```
var px = new 42; // &int
```

Other operations returning pointers are defined in the library, typically by binding to C or C++ functions such as `malloc`.

Pointer Projections

Projection operators applying to arrays, tuples, records, and structs, are all overloaded to work on pointers to these types. For example, to store a value in a structure component:

```
struct X { int a; int b; };
var x = X (1,2);
&x . a <- 42; // sets x.a to 42
```

Object Concept

The use of pointer projections equips Felix with a radical model of values and objects. Instead of references, lvalues and rvalues as in C++, Felix has a powerful alternative model.

In Felix all values of product type are immutable and first class, including arrays. However if such a value is stored in an addressable variable, it has an address. If a value is constructured on the heap a pointer is returned.

Thus, the components of a value can be fetched but not modified, whilst, via pointer projections, the components of a value stored in an addressable location can be fetched and modified.

However, pointers are themselves first class values. Therefore, Felix supports mutation entirely with a value semantics.

1.7.5 Function Types

Specification

https://en.wikipedia.org/wiki/Exponential_object

Felix Exponentials

Felix has 4 primary classes of executable components.

The general type of a function, generator, or procedure is given by

```
D -> [E] C
```

where D is the domain, C is the codomain, and E is a type representing effects. Generally the effects value is not written and defaults to unit. If C is void (0), the type denotes a procedure, otherwise it is a function or generator.

Values of these types are pointers to a procedure or function object.

C function type

The type:

```
D --> C
```

is the type of a C/C++ function pointer. It can be used where a Felix function is required. If C is void, the C function is returning C void. If D is unit or 1, the C function has no arguments. Multiple function parameters are encoded with a tuple type.

Do not confuse C functions with function primitive bindings, closures or which are Felix function type.

1.7.6 Subtyping

Felix supports a specific set of subtyping coercions. Subtyping coercions can be applied manually, and, in specific circumstances automatically.

Passing an argument to a function or procedure is subject to automatic subtyping. Assigning a variable or initialising a value inline does *not* admit automatic subtyping.

Tuples

Tuples and arrays support depth subtyping but not width subtyping. Width subtyping would allow you to pass a tuple with an overlong tail which would be silently chopped off.

Records

Records subtyping is covariant and supports both width and depth subtyping. A record argument may have more fields than the parameter is matches. The field component values may also by subtypes of the corresponding parameter fields.

Polymorphic Variants

Polymorphic variant subtyping is covariant and supports both with and depth subtyping. A variant argument may have less fields than the parameter it matches. Constructor argument values may also be subtypes of the corresponding parameter constructors.

Pointers

Read only and write only pointers are both subtypes of read-write pointers. The pointed at types must be equal.

Functions

Functions are covariant in their domain and contravariant in their codomain.

Nominal Types

Nominal types including primitives, structs and unions do not admit subtying relations by default. However the programmer can write a single step coercion. Currently coercions do not chain. The coercions are done before overloading and so can result in an ambiguity even of one function parameter matches exactly and the other requires a coercion.

```
supertype: long (x:int) => x.long;
```

Polymorphic Specialisation

Polymorphic specialisation is a subtyping relation, but we call it subsumption. The argument of a function must be more specialised than the parameter. Unlike other subtyping steps, overloads select the most function or procedure with the most specialised matching parameter.

[This rule should be applied for subtyping as well as subsumption but isn't.]

1.8 Definitions

1.8.1 Functions

Felix function types in a Felix are written:

```
D -> C
```

where D is the domain type, and C the codomain type, which may not be 0.

Functional Definition

A Felix function can be written with several forms. The simplest form is to use an expression to define the calculation:

```
fun square (x:int): int => x * x;
```

which has type

```
int -> int
```

Imperative Definition

An expanded form of a function definition uses imperative code:

```
fun pythag(x:double, y:double) = {
   var x2 = x * x;
   var y2 = y * y;
   var h = sqrt (x2 + y2);
   return h;
}
```

Pattern Match Definition

A definition like:

```
fun f: opt[int] -> int =
| Some x => x + 1
| None => 0
;
```

is a shortcut form for:

```
fun f (v: opt[int]): int =>
  match v with
  | Some x => x + 1
  | None => 0
;
```

Parameter Forms

A function can only have one parameter, however several can be given if the parameter type is a tuple.

```
fun pythag(x:double, y:double) => ...
```

This is roughly an irrefutable pattern match. The tupled parameter form can nest:

```
fun f(x:double, (y:int, z:long)) => ...
```

Var parameters

By default, a parameter component is treated as a *val* meaning the evaluation strategy for the component is determined by the the compiler and the component is immutable.

If a parameter component is marked *var*, however, it is eagerly evaluated, and is also addressable (and thus mutable).

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

Record Argument form

Given the two functions and application:

```
fun f(x:int, y:double) : int => ..
fun f(a:int, b:double) : int => ..
```

A function can be called with named parameters, that is, with a record:

```
println$ f(x=1,b=2.3);
```

which resolves the ambiguity.

Default Arguments

Default arguments are also allowed on trailing components:

```
fun f(x:int, y:double=4.2) : int => ..
println$ f(x=1);
```

To use the default value, In this case the function must be called with an argument of record type.

Purity

In Felix functions may depend on variables in a containing scope, or, store located via a pointer, therefore functions need not be pure. An adjective can be used to specify a function is pure:

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

The compiler checks functions to determine if they're pure. If it finds they are, it adds the pure attribute itself. It a function is found not to be pure but a pure adjective is specified, it is a fatal error. If the compiler is unable to decide if a function is pure, it is assumed to be pure if and only if the pure adjective is specified.

Inline Functions

Functions can have the *inline* and *noinline* adjective:

```
inline fun add(x:int, y:int) => x + y;
noinline fun sub (x:int, y:int) => x - y;
```

The inline keyword is not a hint, it forces the function to inlined on a direct application unless the function is recursive

Closure are usually not inlined.

Inlining impacts semantics because inline functions usually result in non-var parameters being lazily evaluated. Also, if a parameter isn't used, its initialisation may be elided, whereas for a closure only the type is known and the argument has to be evaluated.

A function marked *noinline* will never be inlined.

Side Effects

Functions in Felix are not allowed to have side effects. The compiler does not enforce this rule. However the compiler optimises code assuming there are no side effects in functions, these optimisations are extensive and pervasive.

It is acceptable to add imperative debugging instrumentation to functions, because the behaviour in the face of optimisations is precisely what the debugging instrumentation is designed to report.

Elision of Constant Functions

If a function return value is invariant, the compiler may delete the function and replace applications of it with the constant returned value. The compiler may or may not be able to determine the invariance and return value in general but Felix guarrantees this property in one very important case: a function with a unit codomain type.

C bindings

Felix can lift, or bind, a C or C++ function as a primitive:

```
 fun sin: double -> double = "sin($1)"
   requires C_headers::math_h
 ;

fun arctan2 : double * double -> double =
  "atan2($1,$2)" requires C_headers::math_g
;
```

The special encoding $1, $2 etc refer to components of the domain tuple. The encoding $a is a quick way to unpack all the arguments.

[More codes]

C function type

Felix also has a type for C function values (pointers):

```
D --> C
```

Do not confuse C function values with Felix functions specified by C bndings: the latter are first class Felix functions.

Closures

A felix function can be converted to a closure, which is a first class value including both the function and its context.

For example:

```
fun add(x:int) = {
  fun g(y:int) => x + y;
  return g;
}
var h = add 1;
var r = h 2; // r set to 3
```

In the example, f has type:

```
int -> int -> int
```

The function arrow is right associative so this means f accepts an int (x), and returns a function which accepts another int (y) and returns an int, which is their sum. The variable h contains a closure of g bound to its context which contains the variable x, which has been set to 1.

A closure is represent at run time by a pointer to an object so passing closures around is cheap. Closures are usually allocated on the heap, which is has a cost. The context of a closure is a list of the most recent activation records of the lexically enclosing function frames (ancestors) called a display. All functions, by default, also include the global data frame, called the thread-frame (because it is shared by all threads).

CLosures exist at run time and cannot be polymorphic.

Higher Order Functions

Functions which accept function arguments, or arguments, or return function values, are called higher order functions.

A special notation exist for defining a function which returns another function:

```
fun add (x:int) (y:int) => y;
println$ f 1 2;
```

Here add is a higher order function with arity 2, it has type

```
int -> int -> int
```

and is equivalent to the previous version of add.

Polymorphic functions

Functions support parametric polymorphism:

```
fun swap[T,U] (x:T, y:T) => y,x;
```

You can also use type class constraints:

```
fun showeol[T with Str[T]] (x:T) => x.str + "\n";
```

The effect of a type class constraint is to inject the methods of the class, specialised to the given arguments, into the scope of the function body. In the example str is a method of Str which translates a value of type T into a human readable string.

Constructor functions

A type name can be used as a function name like this:

```
typedef polar = complex;
ctor complex: double * double = "::std::complex($1,$2)";
ctor polar: double * double = "::std::polar($1,$2)";
var z = polar (1.0, 0.0);
```

The code *ctor* is actually a misnomer: these functions are actually conversions, not type constructors .. but the *ctor* name has stuck.

Constructor function can be polymorphic, in this case the type variables have to be added after the ctor word:

```
ctor[T] vector: 1 = "::std::vector<?1>()";
```

Subtyping Conversion functions

A subtyping conversion can be provided for nominal types:

```
supertype: long (x:int) => x.long;
```

This says int is a subtype of long, so that a function accepting a long will also accept and int. It is recommended not to use this feature unless emulating inheritance based subtyping of structure values.

Projection Functions

Projections of tuple types can be used as functions with the special name *proj* followed by a literal int, the domain type must then be given with an *of* suffix:

```
proj 1 of (int * double)
```

Recall the integer literal is zero origin!

Projections for records, structs, and cstructs use the field name, with a type suffix if necessary to resolve overloads.

Injection Functions

Injections of anonymous sums can be used as functions with the special notation:

```
`1: (int + double)
case 1 of (int + double)
```

Recall the integer literal is zero origin! The more verbose *case* form is considered deprecated.

Injections for unions use the constructor name, possibly with an *of* suffix to resolve overloads.

Pre and post conditions

Functions can have pre-conditions:

```
fun checked_sqrt
  (x:double where x >= 0.0)
  : double expect result >= 0.0
  => sqrt x
;
```

Pre and post conditions are checked dynamically at run time. They are not part of the function type.

Row Polymorphism

1.8.2 Procedures

Blah.

1.8.3 Generators

A generator is a special function like construction which is permitted to have an *internal* effect. The prototypical generator is the rand() function.

An internal effect is one which changes some state on each call and which my modify the result returned by a subsequent call, the state, may not be observed except in the return value of the function.

A generator is defined by using the binder "gen" instead of "fun".

Note that the type of a generator is the same as function type.

Iterators

Felix has a special role for generators named iterator: they're used to traverse a data structure such as a list, array, tree or other type, visiting certain values in some order, and returning each such value in sequence on each call. In C++ terminology an iterator is an input iterator.

Yielding Generators

Most generators maintain internal mutable state in local variables. In order to preserve modification between applications, as well as the current location of the program counter, they execute a yield statement to return a value. Subsequent calls to the generator continue after the yield statement.

To use a yielding generator, a closure must be assigned to a variable to hold the state. For example:

```
gen iterator(xs:T^N) () : opt[T] =
{
  if xs.len > 0uz do
    for var j in 0uz upto xs.len - 1uz do
      yield Some (xs,j).unsafe_get;
    done
  done
  return None[T];
}

proc check() {
  var a = 1,2,3,4;
  var it = iterator a;
next:>
  var v = it ();
  match v with
  | Some x =>
    println$ x;
    goto next;
  | None => ;
  endmatch;
```

defines and uses an array iterator.

1.8.4 Objects

Syntax

```
satom := sadjectives "object" stvarlist slambda_fun_args fun_return_type "=" scompound
sfunction := "object" sdeclname sfun_arg* "implements" object_return_type "="␣
↪scompound
sfunction := "object" sdeclname sfun_arg*  "=" scompound
sfunction :=
  "object" sdeclname sfun_arg* "extends" stypeexpr_comma_list
  "implements" object_return_type "=" scompound

sfunction := "object" sdeclname sfun_arg*  "extends" stypeexpr_comma_list "="␣
↪scompound
sadjective := "method" =># "'Method";
stmt := "interface" sdeclname stype_extension "{" srecord_type "}" =>#
  srecord_type := srecord_mem_decl (";" srecord_mem_decl)* ";"
  stypelist := stypeexpr ("," stypeexpr)*
  stype_extension := "extends" stypelist
  stype_extension := sepsilon
```

Basic Objects

Felix has a special kind of function which is used to construct a Java like object:

```
interface person_t {
  get_name: 1 -> string;
  get_age: 1 -> double;
}

object person (name:string, age: double)
  implements person_t =
{
  method fun get_name => name;
  method fun get_age => age;
}

var joe = person ("joe", 42);
println$ joe.get_name () + " is " + (joe.get_age ()) .str
```

An interface is precisely a record type with an alternate syntax. An object is precisely a function returning a record consisting of closures of all the functions and procedures marked as methods. The optional implements clause specifies the return type.

Java like objects were implemented as a joke to show how powerful Felix is .. however they turned out to be quite useful and are used heavily in the representation of plugins.

The implements clause is optional.

1.8.5 Coroutines

Blah.

1.9 Expression Syntax

1.9.1 Expressions

Expressions in generally used to perform calculations and construct values. Because Felix has a user defined grammar, there are many expression forms which reduce to function applications. In turn, since functions are not permitted side-effects, with some caveats expression forms can be regarded as referentially transparent.

The two main caveats are generators and impurity.

When an expression contains a direct generator application, it is lifted out of the expression: the application is replaced by a variable which is initialised before the expression is evaluated. After the lift and replacement, the remaining expression may be free of effects. However, generators have the same type as functions, so if the application is indirect, for example the application of a closure, Felix doesn't know if it is a generator or function and may or may not lift it out.

Some functions may depend on variables and indeed, and expression can contain variables. Since the evaluation is side-effect free the variable cannot change during the evaluation of the expression. But it can change in a loop so that a subsequent evaluation returns a different result. Of course the most trivial case is when the expression is nothing more than a variable, such as a loop control variable, in which case we'd be surprised if the value didn't change!

1.9.2 Precedence Indices

The grammar provides a standard sequence of precedence indices. Grammar rules for expressions generally look like this:

```
x[s_term_pri] := x[s_term_pri] "/" x[>s_term_pri] =># '(Infix)';
```

The name *x* is used for expressions, the precedence index is a special modifier on the LHS of the production. In the middle part, *x[s_term_pri]* means any expression with the same precedence or higher, whilst *x[>s_term_pri]* means any expression with a strictly higher precedence. Thus, this means the division operator / is left associative.

On the other hand:

```
x[sdollar_apply_pri] := x[>sdollar_apply_pri] "$" x[sdollar_apply_pri] =>#
  "`(ast_apply ,_sr (,_1 ,_3))"
;
```

which means the Haskell *$* application operator is right associative.

Here is the precedence specification from the grammar:

```
priority
  let_pri <
  slambda_pri <

  // low precedence applications
  spipe_apply_pri <
  sdollar_apply_pri <

  // tuples
  stuple_cons_pri <
  stuple_pri <

  // basic logic
  simplies_condition_pri <
```

(continues on next page)

(continued from previous page)

```
  sor_condition_pri <
  sand_condition_pri <
  snot_condition_pri <


  // TeX symbol logic
  stex_implies_condition_pri <
  stex_or_condition_pri <
  stex_and_condition_pri <
  stex_not_condition_pri <

  scomparison_pri <
  sas_expr_pri <

  // set operators
  ssetunion_pri <
  ssetintersection_pri <

  // arrow types
  sarrow_pri <

  // constructor forms
  scase_literal_pri <

  // bitwise operators
  sbor_pri <
  sbxor_pri <
  sband_pri <
  sshift_pri <

  // numeric operators
  ssum_pri <
  ssubtraction_pri <
  sproduct_pri <
  s_term_pri <
  sprefixed_pri <
  spower_pri <
  ssuperscript_pri <

  // addression operations
  srefr_pri <
  scoercion_pri <

  // high precedence applications
  sapplication_pri <

  sfactor_pri <
  srcompose_pri <
  sthename_pri <

  // atomic forms
  satomic_pri
;
```

1.9.3 Let Forms

Syntax

```
x[let_pri] := "let" spattern "=" x[let_pri] "in" x[let_pri]
x[let_pri] := "let" spattern "=" x[let_pri] "in" x[let_pri]
x[let_pri] := "let" "fun" sdeclname fun_return_type "=" smatching+ "in" x[let_pri]
x[let_pri] := "let" pattern_match
x[let_pri] := pattern_match

//$ Named temporary value.
x[sas_expr_pri] := x[sas_expr_pri] "as" sname


//$ Named variable.
x[sas_expr_pri] := x[sas_expr_pri] "as" "var" sname
```

Description

A let form allows an expression to be factored:

```
let p = expr1 in expr2
```

for example:

```
let x2 = x * x in
let y2 = y * y in
  sqrt (x2 + y2)
```

Another let form defines a local function:

```
let fun sq(x:int) = x * x in
  sqrt (sq x + sq y)
```

The *as* expressions allow a *val* or *var* to be defined inside an expression. The definition is lifted out of the expression and replaced by the named variable. This program:

```
var y = (1 as x) + 10;
var z = y + 100;
println$ x + y + z;
```

is equivalent to:

```
var x = 1;
var y = x + 10;
var z = y + 100;

println$ x + y + z;
```

1.9.4 Tuple Expressions

Syntax

```
//$ Tuple formation by cons: right associative.
x[stuple_cons_pri] := x[>stuple_cons_pri] ",," x[stuple_cons_pri]

//$ Tuple formation by append: left associative
x[stuple_cons_pri] := x[stuple_cons_pri] "<,,>" x[>stuple_cons_pri]

//$ Tuple formation non-associative.
x[stuple_pri] := x[>stuple_pri] ( "," x[>stuple_pri])+
```

1.9.5 Logical Operations

Syntax

```
//$ Boolean false.
satom := "false" =># "'(ast_typed_case  0 2)";

//$ Boolean true.
satom := "true" =># "'(ast_typed_case  1 2)";

//$ Logical implication.
x[simplies_condition_pri] := x[>simplies_condition_pri] "implies" x[>simplies_
↪condition_pri]

//$ Logical disjunction (or).
x[sor_condition_pri] := x[>sor_condition_pri] ( "or" x[>sor_condition_pri])+

//$ Logical conjunction (and).
x[sand_condition_pri] := x[>sand_condition_pri] ( "and" x[>sand_condition_pri])+

//$ Logical negation (not).
x[snot_condition_pri] := "not" x[snot_condition_pri]


// tex logic operators
x[stex_implies_condition_pri] := x[>stex_implies_condition_pri]  "\implies" x[>stex_
↪implies_condition_pri]

x[stex_or_condition_pri] := x[>stex_or_condition_pri] ( "\lor" x[>stex_or_condition_
↪pri])+

x[stex_and_condition_pri] := x[>stex_and_condition_pri] ( "\land" x[>stex_and_
↪condition_pri])+

x[stex_not_condition_pri] := "\lnot" x[stex_not_condition_pri]


bin := "\iff" =># '(nos _1)'; // NOT IMPLEMENTED FIXME
bin := "\impliedby" =># '(nos _1)'; // NOT IMPLEMENTED FIXME

//$ Conditional expression.
satom := sconditional "endif" =># "_1";

//$ Conditional expression (prefix).
sconditional := "if" sexpr "then" sexpr selse_part
```

(continues on next page)

(continued from previous page)

```
    selif := "elif" sexpr "then" sexpr

    selifs := selif
    selifs := selifs selif

    selse_part:= "else" sexpr
    selse_part:= selifs "else" sexpr
```

}

Semantics

```
//$ Bitwise operators.
class Bits[t] {
  virtual fun \^ : t * t -> t = "(?1)($1^$2)";
  virtual fun \| : t * t -> t = "$1|$2";
  virtual fun \& : t * t -> t = "$1&$2";
  virtual fun ~: t -> t = "(?1)(~$1)";
  virtual proc ^= : &t * t = "*$1^=$2;";
  virtual proc |= : &t * t = "*$1|=$2;";
  virtual proc &= : &t * t = "*$1&=$2;";

  fun bxor(x:t,y:t)=> x \^ y;
  fun bor(x:t,y:t)=> x \| y;
  fun band(x:t,y:t)=> x \& y;
  fun bnot(x:t)=> ~ x;

}
```

Description

Felix provides two boolean types: `bool` and `cbool` with constants `true` and `false` and `ctrue` and `cfalse` respectively and the following basic operations:

Operator	Function	Semantics
or	lor	disjunction
orelse	orelse	lazy disjunction
	nor	negated disjunction
and	land	conjunction
andthen	andthen	lazy conjunction
	nand	negated conjunction
not	not	negation
implies	implies	implication

The lazy forms require a function of type `1->bool` and `1->cbool` respectively for their second argument, which is evaluated only if the first argument is does not determine the final value.

The type bool is an alias for the sum type 2, which is a compact linear type and will cost 64bits of store.

The type cbool is a binding to C++ bool, and is typically one byte. Functionally these types are equivalent however pointers to cbool are sometimes required for C++ compatibility.

1.9.6 Comparisons

General

Syntax

```
x[scomparison_pri]:= x[>scomparison_pri] cmp x[>scomparison_pri]
x[scomparison_pri]:= x[>scomparison_pri] "not" cmp x[>scomparison_pri]
x[scomparison_pri]:= x[>scomparison_pri] "\not" cmp x[>scomparison_pri]
```

Description

Felix provides a very large set of comparison operators. A collection of TeX operators are used for defined comparisons along with the usual ascii art symbols.

In addition there is a large collection of TeX operators which are not currently used and are available for overloading by the user.

All comparisons can be negated by prefixing the comparison operator with *not*.

```
a < b
a not < b
```

Equivalences

Syntax

```
cmp := "=="
cmp := "!="
cmp := "\ne"
cmp := "\neq"
```

Semantics

```
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;

  fun eq(x:t, y:t)=> x == y;
  fun ne(x:t, y:t)=> x != y;
  fun \ne(x:t, y:t)=> x != y;
  fun \neq(x:t, y:t)=> x != y;
}
```

Description

An equivalence relation is a reflexive, symmetric, transitive comparison. The prototypical equivalence relation is equality. Felix uses C's == operator for equality and supplies the negated forms *!=*, *ne* and *neq* as as well.

Generic Equality

Felix provides a generic function *_eq* which generates an equality relation for non-array tuples and records, and delegates to == for arrays, sums, pointers, polymorphic variants and nominal types.

It is based on the bound value type and generates and unbound expanded comparison which is subsequently bound, therefore it depends on the names *_eq* and ==.

The generated function is given the bound name *__eq*. The compiler checks for this function and uses it if already defined, thereby avoiding duplicate definitions.

[NOTE: It should work for unions and polymorphic variants too, but is not implemented]

```
println$ _eq ((a=1, b=2) , (a=1, b=2));
```

Partial Orders

Syntax

```
cmp := "\subset"       // \(\subset \)
cmp := "\supset"       // \(\supset \)
cmp := "\subseteq"     // \(\subseteq \)
cmp := "\subseteqq"    // \(\subseteqq \)
cmp := "\supseteq"     // \(\supseteq \)
cmp := "\supseteqq"    // \(\supseteqq \)

cmp := "\nsubseteq"    // \(\nsubseteq \)
cmp := "\nsubseteqq"   // \(\nsubseteqq \)
cmp := "\nsupseteq"    // \(\nsupseteq \)
cmp := "\nsupseteqq"   // \(\nsupseteqq \)

cmp := "\subsetneq"    // \(\subsetneq \)
cmp := "\subsetneqq"   // \(\subsetneqq \)
cmp := "\supsetneq"    // \(\supsetneq \)
cmp := "\supsetneqq"   // \(\supsetneqq \)
```

Semantics

```
class Pord[t]{
  inherit Eq[t];
  virtual fun \subset: t * t -> bool;
  virtual fun \supset(x:t,y:t):bool =>y \subset x;
  virtual fun \subseteq(x:t,y:t):bool => x \subset y or x == y;
  virtual fun \supseteq(x:t,y:t):bool => x \supset y or x == y;

  fun \subseteqq(x:t,y:t):bool => x \subseteq y;
  fun \supseteqq(x:t,y:t):bool => x \supseteq y;
```

(continues on next page)

(continued from previous page)

```
  fun \nsubseteq(x:t,y:t):bool => not (x \subseteq y);
  fun \nsupseteq(x:t,y:t):bool => not (x \supseteq y);
  fun \nsubseteqq(x:t,y:t):bool => not (x \subseteq y);
  fun \nsupseteqq(x:t,y:t):bool => not (x \supseteq y);

  fun \supsetneq(x:t,y:t):bool => x \supset y;
  fun \supsetneqq(x:t,y:t):bool => x \supset y;
  fun \supsetneq(x:t,y:t):bool => x \supset y;
  fun \supsetneqq(x:t,y:t):bool => x \supset y;

  axiom trans(x:t, y:t, z:t): \subset(x,y) and \subset(y,z) implies \subset(x,z);
  axiom antisym(x:t, y:t): \subset(x,y) or \subset(y,x) or x == y;
  axiom reflex(x:t, y:t): \subseteq(x,y) and \subseteq(y,x) implies x == y;
}
```

Description

An improper (non-strict) partial order is a reflexive, transitive, anti-symmetric comparison. Proper (strict) partial orders are irreflexive. The prototypical partial order is the subset relation. In type theory, subtype relations are also partial orders.

Reference: https://en.wikipedia.org/wiki/Partially_ordered_set

Total Orders

Syntax

```
cmp := "<"          // \(< \)

cmp := "\lt"        // \(\lt \)
cmp := "\lneq"      // \(\lneq \)
cmp := "\lneqq"     // \(\lneqq \)

cmp := "<="         // \(<= \)
cmp := "\le"        // \(\le \)
cmp := "\leq"       // \(\leq \)
cmp := "\leqq"      // \(\leqq \)

cmp := ">"          // \(> \)
cmp := "\gt"        // \(\gt \)
cmp := "\gneq"      // \(\gneq \)
cmp := "\gneqq"     // \(\gneqq \)

cmp := ">="         // \(>= \)
cmp := "\ge"        // \(\ge \)
cmp := "\geq"       // \(\geq \)
cmp := "\geqq"      // \(\geqq \)

cmp := "\nless"     // \(\nless \)
cmp := "\nleq"      // \(\nleq \)
cmp := "\nleqq"     // \(\nleqq \)
cmp := "\ngtr"      // \(\ngtr \)
```

(continues on next page)

(continued from previous page)

```
cmp := "\ngeq"      // \(\ngeq \)
cmp := "\ngeqq"     // \(\ngeqq \)
```

Semantics

```
class Tord[t]{
  inherit Eq[t];
  // defined in terms of <, argument order swap, and boolean negation

  // less
  virtual fun < : t * t -> bool;
  fun lt (x:t,y:t): bool=> x < y;
  fun \lt (x:t,y:t): bool=> x < y;
  fun \lneq (x:t,y:t): bool=> x < y;
  fun \lneqq (x:t,y:t): bool=> x < y;


  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;


  // greater
  fun >(x:t,y:t):bool => y < x;
  fun gt(x:t,y:t):bool => y < x;
  fun \gt(x:t,y:t):bool => y < x;
  fun \gneq(x:t,y:t):bool => y < x;
  fun \gneqq(x:t,y:t):bool => y < x;

  // less equal
  fun <= (x:t,y:t):bool => not (y < x);
  fun le (x:t,y:t):bool => not (y < x);
  fun \le (x:t,y:t):bool => not (y < x);
  fun \leq (x:t,y:t):bool => not (y < x);
  fun \leqq (x:t,y:t):bool => not (y < x);
  fun \leqslant (x:t,y:t):bool => not (y < x);


  // greater equal
  fun >= (x:t,y:t):bool => not (x < y);
  fun ge (x:t,y:t):bool => not (x < y);
  fun \ge (x:t,y:t):bool => not (x < y);
  fun \geq (x:t,y:t):bool => not (x < y);
  fun \geqq (x:t,y:t):bool => not (x < y);
  fun \geqslant (x:t,y:t):bool => not (x < y);

  // negated, strike-through
  fun \ngtr (x:t,y:t):bool => not (x < y);
  fun \nless (x:t,y:t):bool => not (x < y);

  fun \ngeq (x:t,y:t):bool => x < y;
  fun \ngeqq (x:t,y:t):bool => x < y;
  fun \ngeqslant (x:t,y:t):bool => x < y;
```

(continues on next page)

(continued from previous page)

```
  fun \nleq (x:t,y:t):bool => not (x <= y);
  fun \nleqq (x:t,y:t):bool => not (x <= y);
  fun \nleqslant (x:t,y:t):bool => not (x <= y);


  // maxima and minima
  fun max(x:t,y:t):t=> if x < y then y else x endif;
  fun \vee(x:t,y:t) => max (x,y);

  fun min(x:t,y:t):t => if x < y then x else y endif;
  fun \wedge(x:t,y:t):t => min (x,y);

}
```

Description

An improper (non-strict) total, or linear order, is an anti-symmtric, transitive relation with the connex property.

Reference: https://en.wikipedia.org/wiki/Total_order

operator	numeric semantics
==, \eq	equality
!=, \ne	inequality
<, \lt	less than
<=, \le	less or equal
>, \gt	greater than
>=, \ge	greater or equal

1.9.7 Tex Sytmbols

```
    // A

  bin := "\amalg"                 // \(\amalg \)
  cmp := "\approx"                // \(\approx \)
  cmp := "\approxeq"              // \(\approxeq \)
  cmp := "\Arrowvert"             // \(\Arrowvert \)
  cmp := "\arrowvert"             // \(\arrowvert \)
  cmp := "\asymp"                 // \(\asymp \)

    // B

  cmp := "\backsim"               // \(\backsim \)
  cmp := "\backsimeq"             // \(\backsimeq \)
  cmp := "\bar"                   // \(\bar \)
  cmp := "\barwedge"              // \(\barwedge \)
  cmp := "\between"               // \(\between \)
  bin := "\bigcap"                // \(\bigcap \)
  bin := "\bigcirc"               // \(\bigcirc \)
  bin := "\bigcup"                // \(\bigcup \)
  bin := "\bigodot"               // \(\bigodot \)
  bin := "\bigoplus"              // \(\bigoplus \)
```

(continues on next page)

(continued from previous page)

```
  bin := "\bigotimes"             // \(\bigotimes \)
  bin := "\bigsqcup"              // \(\bigsqcup \)
  bin := "\bigtriangledown"       // \(\bigtriangledown \)
  bin := "\bigtriangleup"         // \(\bigtriangleup \)
  bin := "\biguplus"              // \(\biguplus \)
  bin := "\bigvee"                // \(\bigvee \)
  bin := "\bigwedge"              // \(\bigwedge \)
  bin := "\bowtie"                // \(\bowtie \)
  bin := "\Box"                   // \(\Box \)
  bin := "\boxdot"                // \(\boxdot \)
  bin := "\boxminus"              // \(\boxminus \)
  bin := "\boxplus"               // \(\boxplus \)
  bin := "\boxtimes"              // \(\boxtimes \)
  cmp := "\Bumpeq"                // \(\Bumpeq \)
  cmp := "\bumpeq"                // \(\bumpeq \)

      // C

  bin := "\Cap"                   // \(\Cap \)
  bin := "\cdot"                  // \(\cdot \)
  bin := "\cdotp"                 // \(\cdotp \)
  cmp := "\circeq"                // \(\circeq \)
  bin := "\circledast"            // \(\circledast \)
  bin := "\circledcirc"           // \(\circledcirc \)
  bin := "\circleddash"           // \(\circleddash \)
  cmp := "\cong"                  // \(\cong \)
  bin := "\coprod"                // \(\coprod \)
  bin := "\Cup"                   // \(\Cup \)
  cmp := "\curlyeqprec"           // \(\curlyeqprec \)
  cmp := "\curlyeqsucc"           // \(\curlyeqsucc \)
  bin := "\curlyvee"              // \(\curlyvee \)
  bin := "\curlywedge"            // \(\curlywedge \)

      // D

  arr := "\dashleftarrow"         // \(\dashleftarrow \)
  arr := "\dashrightarrow"        // \(\dashrightarrow \)
  bin := "\divideontimes"         // \(\divideontimes \)
  cmp := "\doteq"                 // \(\doteq \)
  cmp := "\Doteq"                 // \(\Doteq \)
  cmp := "\doteqdot"              // \(\doteqdot \)
  bin := "\dotplus"               // \(\dotplus \)
  bin := "\doublebarwedge"        // \(\doublebarwedge \)
  bin := "\doublecap"             // \(\doublecap \)
  bin := "\doublecup"             // \(\doublecup \)
  bin := "\Downarrow"             // \(\Downarrow \)
  bin := "\downarrow"             // \(\downarrow \)
  bin := "\downdownarrows"        // \(\downdownarrows \)
  bin := "\downharpoonleft"       // \(\downharpoonleft \)
  bin := "\downharpoonright"      // \(\downharpoonright \)

      // E

  cmp := "\eqcirc"                // \(\eqcirc \)
  cmp := "\eqsim"                 // \(\eqsim \)
  cmp := "\eqslantgtr"            // \(\eqslantgtr \)
  cmp := "\eqslantless"           // \(\eqslantless \)
```

(continues on next page)

(continued from previous page)

```
  cmp := "\equiv"                 // \(\equiv \)

      // F

  bin := "\fallingdotseq"         // \(\fallingdotseq \)

      // G

  cmp := "\geqslant"              // \(\geqslant \)
  arr := "\gets"                  // \(\gets \)
  cmp := "\gg"                    // \(\gg \)
  cmp := "\ggg"                   // \(\ggg \)
  cmp := "\gggtr"                 // \(\gggtr \)
  cmp := "\gnapprox"              // \(\gnapprox \)
  cmp := "\gnsim"                 // \(\gnsim \)
  cmp := "\gtrapprox"             // \(\gtrapprox \)
  cmp := "\gtrdot"                // \(\gtrdot \)
  cmp := "\gtreqless"             // \(\gtreqless \)
  cmp := "\gtreqqless"            // \(\gtreqqless \)
  cmp := "\gtrless"               // \(\gtrless \)
  cmp := "\gtrsim"                // \(\gtrsim \)
  cmp := "\gvertneqq"             // \(\gvertneqq \)

      // H

  arr := "\hookleftarrow"         // \(\hookleftarrow \)
  arr := "\hookrightarrow"        // \(\hookrightarrow \)

      // I

      // J

  bin := "\Join"                  // \(\Join \)

      // K

      // L

  arr := "\leadsto"               // \(\leadsto \)
  arr := "\Leftarrow"             // \(\Leftarrow \)
  arr := "\leftarrow"             // \(\leftarrow \)
  arr := "\leftarrowtail"         // \(\leftarrowtail \)
  arr := "\leftharpoondown"       // \(\leftharpoondown \)
  arr := "\leftharpoonup"         // \(\leftharpoonup \)
  arr := "\leftleftarrows"        // \(\leftleftarrows \)
  arr := "\Leftrightarrow"        // \(\Leftrightarrow \)
  arr := "\leftrightarrow"        // \(\leftrightarrow \)
  cmp := "\leftrightarrows"       // \(\leftrightarrows \)
  cmp := "\leftrightharpoons"     // \(\leftrightharpoons \)
  arr := "\leftrightsquigarrow"   // \(\leftrightsquigarrow \)
  cmp := "\leqslant"              // \(\leqslant \)
  cmp := "\lessapprox"            // \(\lessapprox \)
  cmp := "\lessdot"               // \(\lessdot \)
  cmp := "\lesseqgtr"             // \(\lesseqgtr \)
  cmp := "\lesseqqgtr"            // \(\lesseqqgtr \)
  cmp := "\lessgtr"               // \(\lessgtr \)
  cmp := "\lesssim"               // \(\lesssim \)
```

(continues on next page)

(continued from previous page)

```
  arr := "\Lleftarrow"            // \(\Lleftarrow \)
  cmp := "\lll"                   // \(\lll \)
  cmp := "\llless"                // \(\llless \)
  cmp := "\lnapprox"              // \(\lnapprox \)
  cmp := "\lnot"                  // \(\lnot \)
  cmp := "\lnsim"                 // \(\lnsim \)
  arr := "\Longleftarrow"         // \(\Longleftarrow \)
  arr := "\longleftarrow"         // \(\longleftarrow \)
  arr := "\Longleftrightarrow"    // \(\Longleftrightarrow \)
  arr := "\longleftrightarrow"    // \(\longleftrightarrow \)
  arr := "\longmapsto"            // \(\longmapsto \)
  arr := "\Longrightarrow"        // \(\Longrightarrow \)
  arr := "\longrightarrow"        // \(\longrightarrow \)
  cmp := "\ltimes"                // \(\ltimes \)
  cmp := "\lvertneqq"             // \(\lvertneqq \)

// M

  arr := "\mapsto"                // \(\mapsto \)

// N

  cmp := "\ncong"                 // \(\ncong \)
  cmp := "\ngeqslant"             // \(\ngeqslant \)
  cmp := "\ni"                    // \(\ni \)
  cmp := "\nleqslant"             // \(\nleqslant \)
  cmp := "\nparallel"             // \(\nparallel \)
  cmp := "\nprec"                 // \(\nprec \)
  cmp := "\npreceq"               // \(\npreceq \)
  cmp := "\nsim"                  // \(\nsim \)
  cmp := "\nsucc"                 // \(\nsucc \)
  cmp := "\nsucceq"               // \(\nsucceq \)
  cmp := "\ntriangleleft"         // \(\ntriangleleft \)
  cmp := "\ntrianglelefteq"       // \(\ntrianglelefteq \)
  cmp := "\ntriangleright"        // \(\ntriangleright \)
  cmp := "\ntrianglerighteq"      // \(\ntrianglerighteq \)

// O

  bin := "\odot"                  // \(\odot \)
  bin := "\ominus"                // \(\ominus \)
  bin := "\oplus"                 // \(\oplus \)
  bin := "\oslash"                // \(\oslash \)
  //bin := "\otimes"              // \(\otimes \)

// P

  cmp := "\perp"                  // \(\perp \)
  bin := "\pm"                    // \(\pm \)
  cmp := "\prec"                  // \(\prec \)
  cmp := "\precapprox"            // \(\precapprox \)
  cmp := "\preccurlyeq"           // \(\preccurlyeq \)
  cmp := "\preceq"                // \(\preceq \)
  cmp := "\precnapprox"           // \(\precnapprox \)
  cmp := "\precneqq"              // \(\precneqq \)
  cmp := "\precnsim"              // \(\precnsim \)
  cmp := "\precsim"               // \(\precsim \)
```

(continues on next page)

(continued from previous page)

```
  bin := "\prod"                  // \(\prod \)
  cmp := "\propto"                // \(\propto \)

// Q

// R

  cmp := "\rhd"                   // \(\rhd \)
  arr := "\Rightarrow"            // \(\Rightarrow \)
  arr := "\rightarrow"            // \(\rightarrow \)
  arr := "\rightarrowtail"        // \(\rightarrowtail \)
  arr := "\rightharpoondown"      // \(\rightharpoondown \)
  arr := "\rightharpoonup"        // \(\rightharpoonup \)
  arr := "\rightleftarrows"       // \(\rightleftarrows \)
  arr := "\rightleftharpoons"     // \(\rightleftharpoons \)
  arr := "\rightleftharpoons"     // \(\rightleftharpoons \)
  arr := "\rightrightarrows"      // \(\rightrightarrows \)
  arr := "\rightsquigarrow"       // \(\rightsquigarrow \)
  arr := "\Rrightarrow"           // \(\Rrightarrow \)
  cmp := "\rtimes"                // \(\rtimes \)

// S

  bin := "\setminus"              // \(\setminus \)
  cmp := "\sim"                   // \(\sim \)
  cmp := "\simeq"                 // \(\simeq \)
  cmp := "\smallsetminus"         // \(\smallsetminus \)
  bin := "\sqcap"                 // \(\sqcap \)
  bin := "\sqcup"                 // \(\sqcup \)
  cmp := "\sqsubset"              // \(\sqsubset \)
  cmp := "\sqsubseteq"            // \(\sqsubseteq \)
  cmp := "\sqsupset"              // \(\sqsupset \)
  cmp := "\sqsupseteq"            // \(\sqsupseteq \)
  bin := "\square"                // \(\square \)
  cmp := "\Subset"                // \(\Subset \)
  cmp := "\succ"                  // \(\succ \)
  cmp := "\succapprox"            // \(\succapprox \)
  cmp := "\succcurlyeq"           // \(\succcurlyeq \)
  cmp := "\succeq"                // \(\succeq \)
  cmp := "\succnapprox"           // \(\succnapprox \)
  cmp := "\succneqq"              // \(\succneqq \)
  cmp := "\succnsim"              // \(\succnsim \)
  cmp := "\succsim"               // \(\succsim \)
  cmp := "\Supset"                // \(\Supset \)

// T

  cmp := "\thickapprox"           // \(\thickapprox \)
  cmp := "\thicksim"              // \(\thicksim \)
  bin := "\times"                 // \(\times \)
  arr := "\to"                    // \(\to \)
  bin := "\triangle"              // \(\triangle \)
  bin := "\triangledown"          // \(\triangledown \)
  cmp := "\triangleleft"          // \(\triangleleft \)
  cmp := "\trianglelefteq"        // \(\trianglelefteq \)
  cmp := "\triangleq"             // \(\triangleq \)
  cmp := "\triangleright"         // \(\triangleright \)
```

(continues on next page)

(continued from previous page)

```
  cmp := "\trianglerighteq"       // \(\trianglerighteq \)
  arr := "\twoheadleftarrow"      // \(\twoheadleftarrow \)
  arr := "\twoheadrightarrow"     // \(\twoheadrightarrow \)

// U

  cmp := "\unlhd"                // \(\unlhd \)
  cmp := "\unrhd"                // \(\unrhd \)
  bin := "\Uparrow"              // \(\Uparrow \)
  bin := "\uparrow"              // \(\uparrow \)
  bin := "\Updownarrow"          // \(\Updownarrow \)
  bin := "\updownarrow"          // \(\updownarrow \)
  bin := "\upharpoonleft"        // \(\upharpoonleft \)
  bin := "\upharpoonright"       // \(\upharpoonright \)
  bin := "\uplus"                // \(\uplus \)
  bin := "\upuparrows"           // \(\upuparrows \)

// V

  cmp := "\varsubsetneq"         // \(\varsubsetneq \)
  cmp := "\varsubsetneqq"        // \(\varsubsetneqq \)
  cmp := "\varsupsetneq"         // \(\varsupsetneq \)
  cmp := "\varsupsetneqq"        // \(\varsupsetneqq \)
  cmp := "\veebar"               // \(\veebar \)

// W


// X

  arr := "\xleftarrow"           // \(\xleftarrow \)
  arr := "\xrightarrow"          // \(\xrightarrow \)

// Y


// Z



// The precedences here are a hack: so many operators.
// The general effect is: except for keyword logic connectives,
// these operations are all done AFTER any ASCII art ops
// and, only one is allowed per sub-expression: you must use parens
// if you use more than one. We'll fix this for some key operations later,
// particularly the setwise and logic connectors. However, the comparisons
// are at the right precedence.
// (fact is, I don't know what half the operators are for anyhow .. )

  x[stuple_pri] := x[>stuple_pri] "\brace" x[>stuple_pri] =># "(Infix)";
  x[stuple_pri] := x[>stuple_pri] "\brack" x[>stuple_pri] =># "(Infix)";


  x[scomparison_pri]:= x[>scomparison_pri] bin x[>scomparison_pri]
  // set ops (note: no setminus, its a standard binop at the moment ;)
  // note: no \Cap or other variants .. would interfere with chain
  // there's no reason at all to chain these anyhow, they're standard left assoc␣
↪operators
```

(continues on next page)

(continued from previous page)

```
  // All arrows are right associative .. hmm ..
  x[sarrow_pri] := x[scase_literal_pri] arr x[sarrow_pri]
```

1.9.8 Slice Expressions

Syntax

```
x[sarrow_pri] := x[>sarrow_pri] ".." x[>sarrow_pri]
x[sarrow_pri] := x[>sarrow_pri] "..<" x[>sarrow_pri]
x[sarrow_pri] := "..<" x[>sarrow_pri]
x[sarrow_pri] := ".." x[>sarrow_pri]
x[sarrow_pri] := x[>sarrow_pri] ".."
x[sarrow_pri] := ".."
x[sarrow_pri] := x[>sarrow_pri] ".+" x[>sarrow_pri]
```

1.9.9 Variant literals

```
//$ Case value, sum types
x[scase_literal_pri] := "case" sinteger "of" x[ssum_pri]
x[scase_literal_pri] := "`" sinteger "of" x[ssum_pri]
x[scase_literal_pri] := "`" sinteger ":" x[ssum_pri]

//$ Variant value, polymorphic variant type
x[sthename_pri] := "#" "case" sname
x[sthename_pri] := "#" "`" sname
x[sapplication_pri] := "case" sname x[>sapplication_pri]
x[sapplication_pri] := "`" sname x[>sapplication_pri]

//$ Variant decode.
x[sapplication_pri] := "caseno" x[>sapplication_pri]
x[sapplication_pri] := "casearg" x[>sapplication_pri]


//$ Tuple projection function.
x[scase_literal_pri] := "proj" sinteger "of" x[ssum_pri]

// coarray injection
// (ainj (r:>>4) of (4 *+ int)) 42
x[scase_literal_pri] := "ainj"  stypeexpr "of" x[ssum_pri]
```

Description

Blah.

1.9.10 Applications

Syntax

```
x[sdollar_apply_pri] := x[stuple_pri] "$" x[sdollar_apply_pri]
x[spipe_apply_pri] := x[spipe_apply_pri] "|>" x[stuple_pri]
x[stuple_pri]  := x[stuple_pri] "`(" sexpr ")" sexpr

x[sapplication_pri] := x[sapplication_pri] x[>sapplication_pri]
x[sapplication_pri] := "likely" x[>sapplication_pri]
x[sapplication_pri] := "unlikely" x[>sapplication_pri]

x[sfactor_pri] := x[sfactor_pri] "." x[>sfactor_pri]
x[sfactor_pri] := x[sfactor_pri] "*." x[>sfactor_pri]
x[sfactor_pri] := x[sfactor_pri] "&." x[>sfactor_pri]

x[sthename_pri] := "#" x[sthename_pri]
```

Description

Felix has a large number of application operators, from highest precedence to lowest:

```
f$a    // Haskell operator dollar, right associative
a|>f   // operator pipe apply: reverse application, left associative
f a    // operator whitespace, left associative
a.f    // operator dot: reverse application, left associative
a*.f   // means (*a).f
a&.f   // means (&a).f
#f     // constant evaluator: means f ()
```

Another application for binary operator is

```
a `(f) b // means f (a,b)
```

Overloading

There are two kinds of application: a *direct* application and an *indirect* application. A direct application is the most common kind and involves applying a function by name to an argument expression:

```
fun f(x:int) => x;
fun f(y:double) => y;
println$ f 1;
```

In this case, there are two visible functions named *f*, the first one is selected because its domain has the same type as the argument, namely *int*.

The algorithm which selects the function to use is called overload resolution.

Indirect Application

When an expression other than a function name is applied, there are two cases: *normal* indirect application or *special* application.

If the expression has function type, the expression represents a function closure rather than a function. The argument it is applied to must match the domain of the function type:

```
fun f(x:int) => x;
var g = f;
println$ g 1;
```

Special Apply

If the expression being applied has type T which is not a function type, then Felix instead looks for a function named *apply* which takes a tuple of type *T * A* where *A* is the type of the argument. For example:

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

Here is a string is applied to a string. Since a string isn't a function, Felix looks for and finds a function named *apply* with domain *string * string*.

Likelyhood

The *likely* and *unlikely* pseudo functions are optimisation hints applied to expressions of boolean type which indicate that the value is likely (or unlikely, respectively) to be true. The hint is passed on to C++ compilers which have an intrinsic to support it, the hint allows the C++ compiler to reorganise code so that the most likely flow continues on and the least likely uses a branch, the idea being to keep the instruction pipeline full and perhaps influence speculative execution choices.

In particular, Felix adds *likely* to branches in loops which cause the loop to repeat and *unlikely* to those which terminate the loop.

1.9.11 Addressing

Syntax

```
//$ C dereference.
x[srefr_pri] := "*" x[srefr_pri]

//$ Deref primitive.
//x[srefr_pri] := "_deref" x[srefr_pri]

//$ Operator new.
x[srefr_pri] := "new" x[srefr_pri]

//$ Felix pointer type and address of operator.
x[sthename_pri] := "&" x[sthename_pri]

//$ Felix pointer type and address of operator.
x[sthename_pri] := "_uniq" x[sthename_pri]
x[sthename_pri] := "_rref" x[sthename_pri]
x[sthename_pri] := "&<" x[sthename_pri]
x[sthename_pri] := "_wref" x[sthename_pri]
x[sthename_pri] := "&>" x[sthename_pri]


//$ Felix address of operator.
x[sthename_pri] := "label_address" sname
```

(continues on next page)

(continued from previous page)

```
//$ C pointer type.
x[sthename_pri] :=  "@" x[sthename_pri]
```

1.9.12 Functional Expressions

In Felix, a function is modelled in principle by a C++ class. A function value, on the other hand, is not a function, rather it is a called a closure because it captures its environment and may have internal state. Closures are represented as pointers to objects of some C++ class.

The distinction is important. In principle in an application *f a* the *f* is a function value of some suitable type. The type given for a function in a definition is the type of its closure. However if the function expression is just a name, overload resolution is performed to find a suitable function to apply, and the application is direct so that the function generating the closure being applied is known, and the application is optimised, for example, by inlining the application.

Function names

Syntax

```
ssuffixed_name := squalified_name "of" x[sthename_pri]
```

Description

The full name of a function defined by the user can be given with a suffixed name. For example:

```
class X { fun f(x:int) => x; }
var g = X::f of int;
```

The *of* suffix is used in lieu of an argument to perform overload resolution and select a specific function.

This syntax is also used to name union constructors.

Tuple projections

Syntax

```
x[scase_literal_pri] := "proj" sinteger "of" x[ssum_pri]
```

Description

You can name a specific projection of a tuple type by:

```
typedef t = int * long * string;
var g : t -> string = proj 2 of t;
```

Array projections

```
typedef t = int^5;
var g : t -> 5 -> int = aproj of t;
```

[NOTE: THIS ISNT IMPLEMENTED BUT SHOULD BE]

Record and struct projections

Records and structs use the field name as the name of the projection, so the usual suffixed form can be used to specify a projection.

```
typedef t = (a:int, b:long, c:string);
var g : t -> string = a of t;
struct X { a:int; b:long; c:string);
var h : X -> string = a of X;
```

Sum Injections

Syntax

```
x[scase_literal_pri] := "case" sinteger "of" x[ssum_pri]
x[scase_literal_pri] := "`" sinteger "of" x[ssum_pri]
x[scase_literal_pri] := "`" sinteger ":" x[ssum_pri]
```

Coarray Injection

Syntax

```
// coarray injection
// (ainj (r:>>4) of (4 *+ int)) 42
x[scase_literal_pri] := "ainj"  stypeexpr "of" x[ssum_pri] =># "`(ast_ainj ,_sr ,_2 ,_
↪4)";
```

Compositions

Forward and reverse serial, parallel, mediating morphisms.

```
//$ Reverse composition
x[srcompose_pri] := x[srcompose_pri] "\odot" x[>srcompose_pri]

//$ Forward composition
x[ssuperscript_pri] := x[ssuperscript_pri] "\circ" x[>ssuperscript_pri]

// ????
x[ssuperscript_pri] := x[ssuperscript_pri] "\cdot" x[>ssuperscript_pri]
```

Categorical Constructions

```
// mediating morphism of a product <f,g>
satom := "\langle" sexpr "\rangle" =># "`(ast_apply ,_sr (,(noi 'lrangle) (,_2)))";
satom := "\left" "\langle" sexpr "\right" "\rangle" =># "`(ast_apply ,_sr (,(noi
↪'lrangle) (,_3)))";

// mediating morphism of a sum [f,g]
satom := "\lbrack" sexpr "\rbrack" =># "`(ast_apply ,_sr (,(noi 'lrbrack) (,_2)))";
satom := "\left" "\lbrack" sexpr "\right" "\rbrack" =># "`(ast_apply ,_sr (,(noi
↪'lrbrack) (,_3)))";
```

```
fun f(x:int) => x.str;
fun g(x:int) => x.double+42.1;

// mediating morphism of product
println$ \langle f , g\rangle 1; // ("1", 43.1)

// parallel composition
println$ \prod (f , g) (1,2); // ("1", 44.1)
```

Composition Sumary

There are two composition operators for functions, both are left associative:

operator	semantics
\circ	forward composition
\odot	reverse composition

Lambda Forms

A unit function or procedure can be written inline, anonymously:

```
// functions
{ 42 }                          // 1->int
{ var x = 1; x * x }            // 1->int
{ var x = 1; return x * x; }    // 1->int

// procedure
{ var x = 1; println$ x; }      // 1->0
```

A useful construction:

```
{
  var x = 1;
  println$ "Hello";
};
```

looks like a block in C except for the terminating *;*. Actually it is a call to an anonymous procedure since the *call* can be elided, and the argument *()* can also be elided. You can jump out of an anonymous procedure but not into it, since it creates a scope. You cannot jump out of functions, and thus not anonymous functions either.

Functions or procedures with arguments can be written too:

```
(fun (x:int)=>x * x)
(proc (x:int){println$ x;})
```

The enclosing parens are not part of the syntax but are often required to get the precedence right.

1.9.13 Coercions

```
//$ Suffixed coercion.
x[scoercion_pri] := x[scoercion_pri] ":>>" x[>scoercion_pri]
```

1.9.14 Qualified Names

Syntax

```
//$ Qualified name.
sreally_qualified_name := squalified_name "::" ssimple_name_parts

squalified_name := sreally_qualified_name

squalified_name := ssimple_name_parts


ssimple_name_parts := sname
ssimple_name_parts := sname "[" "]"
ssimple_name_parts := sname "[" sexpr "]"

//$ Suffixed name (to name functions).
ssuffixed_name := squalified_name "of" x[sthename_pri]
```

1.9.15 Bracket Forms

```
//$ Array expression (deprecated).
satom := "[|" sexpr "|]"

//$ Short form anonymous function closure.
satom := "{" sexpr "}"

//$ Grouping.
satom := "(" sexpr ")"
satom := "\(" sexpr "\)"
satom := "\[" sexpr "\]"
satom := "\{" sexpr "\}"

//$ floor and ceiling
satom := "\lceil" sexpr "\rceil"
satom := "\lfloor" sexpr "\rfloor"

//$ absolute value
satom := "\lvert" sexpr "\rvert"
satom := "\left" "|" sexpr "\right" "|"
satom := "\left" "\vert" sexpr "\right" "\vert"
```

(continues on next page)

(continued from previous page)

```
//$ norm or length
satom := "\lVert" sexpr "\rVert"
satom := "\left" "\Vert" sexpr "\right" "\Vert"

// mediating morphism of a product <f,g>
satom := "\langle" sexpr "\rangle"
satom := "\left" "\langle" sexpr "\right" "\rangle"

// mediating morphism of a sum [f,g]
satom := "\lbrack" sexpr "\rbrack"
satom := "\left" "\lbrack" sexpr "\right" "\rbrack"
```

1.10 Pattern Matching

Felix provides an advanced pattern matching system which includes generic patterns and user defined patterns.

Pattern matching provides a way to decode a data structure by providing an image of the type with "holes" in it which are indicated by variables. Provided pattern matches the value the variables take on the value of the part of the type which is missing.

For products, pattern variables are projections of the value, possibly chained together in reverse order.

For sum types including variants, the argument of the constructor which created the value is extracted, after checking the value was indeed made by the corresponding injection function. If not, the next pattern is tried.

A pattern match over a product type is said to be irrefutable because it cannot fail after static type checking; however a pattern it is included in may fail, and a pattern matching a component may also fail.

1.10.1 Matches

Syntax

```
syntax patterns {

  block = match_stmt;

  smatch_head := "chainmatch" sexpr "with" stmt_matching+ =># "`(,_2 ,_4)";
  smatch_link := "ormatch" sexpr "with" stmt_matching+ =># "`(,_2 ,_4)";
  smatch_chain := smatch_chain smatch_link =># "(cons _2 _1)"; // revsersed
  smatch_chain := smatch_link =># "`(,_1)";

  match_stmt := smatch_head smatch_chain "endmatch" ";" =>#
    "`(ast_stmt_chainmatch ,_sr ,(cons _1 (reverse _2)))"
  ;

  match_stmt := smatch_head "endmatch" ";" =>#
    "`(ast_stmt_match (,_sr ,_1))"
  ;

  //$ Pattern match statement.
  //$ At least one branch must match or the program aborts with a match failure.
  match_stmt:= "match" sexpr "with" stmt_matching+ "endmatch" ";" =>#
```

(continues on next page)

(continued from previous page)

```
    "`(ast_stmt_match (,_sr ,_2 ,_4))";

  match_stmt:= "match" sexpr "do" stmt_matching+ "done" =>#
    "`(ast_stmt_match (,_sr ,_2 ,_4))";

  //$ A single branch of a pattern match statement.
  //$ The match argument expression is compared to the pattern.
  //$ If it matches any contained pattern variables are assigned
  //$ the values in the corresponding possition of the expression,
  //$ and the statements are executed.
  private stmt_matching := "|" spattern "=>" stmt+ =># "`(,_2 ,_4)";

  //$ Pattern match expression with terminator.
  satom := pattern_match "endmatch" =># "_1";

  //$ Pattern match expression without terminator.
  //$ Match the expression against each of the branches in the matchings.
  //$ At least one branch must match or the program aborts with a match failure.
  pattern_match := "match" sexpr "with" smatching+ =>#
    "`(ast_match ,_sr (,_2 ,_4))";

  //$ The match argument expression is compared to the pattern.
  //$ If it matches any contained pattern variables are assigned
  //$ the values in the corresponding possition of the expression,
  //$ and expression is evaluated and becomes the return value
  //$ of the whole match.
  smatching := "|" spattern "=>" x[let_pri] =># "`(,_2 ,_4)";

  //$ Match nothing.
  smatching := "|" "=>" sexpr =># "`((pat_none ,_sr) ,_3)";
```

1.10.2 Patterns

Syntax

```
spattern := sguard_pattern ("|" sguard_pattern)* =># "(chain 'pat_alt _1 _2)";
sguard_pattern := swith_pattern "when" sexpr =># "`(pat_when ,_sr ,_1 ,_3)";
sguard_pattern := swith_pattern =># "_1";
```

With pattern

Syntax

```
swith_pattern := sas_pattern "with" spat_avars =># "`(pat_with ,_sr ,_1 ,_3)";
  spat_avar := sname "=" stypeexpr =># "`(,_1 ,_3)";
  spat_avars := list::commalist1<spat_avar> =># "_1";
swith_pattern := sas_pattern =># "_1";
```

As pattern

Syntax

```
sas_pattern := scons_pattern "as" sname =># "`(pat_as ,_sr ,_1 ,_3)";
sas_pattern := scons_pattern =># "_1";
```

Cons pattern

Syntax

```
scons_pattern := stuple_cons_pattern "!" scons_pattern
scons_pattern := stuple_cons_pattern
scons_pattern :="[" slist_pattern "]"
scons_pattern :="[" "]"
```

List pattern

Syntax

```
slist_pattern := scoercive_pattern "," slist_pattern
slist_pattern := scoercive_pattern
slist_pattern := scoercive_pattern ",," scoercive_pattern
```

Tuple Cons Pattern

Syntax

```
stuple_cons_pattern := stuple_pattern ",," stuple_cons_pattern =>#
stuple_cons_pattern := stuple_pattern "<,,>" stuple_cons_pattern =>#
stuple_cons_pattern := stuple_pattern =># "_1"
```

Tuple Pattern

Syntax

```
stuple_pattern := scoercive_pattern ("," scoercive_pattern )* =>#
```

Coercive Pattern

Syntax

```
scoercive_pattern := sapplicative_pattern "|>" x[sarrow_pri]
scoercive_pattern := sapplicative_pattern ":" x[sarrow_pri]
scoercive_pattern := sapplicative_pattern
scoercive_pattern := stypeexpr ":>>" sname
```

Applicative Pattern

Syntax

```
sapplicative_pattern := sctor_name sargument_pattern

private sapplicative_pattern := sctor_name stypeexpr+ sargument_pattern

  //$ The sum type constructor can either be a qualified name...
  private sctor_name := sname

  //$ or it can be a case literal.
  sctor_name := "case" sinteger
  sctor_name := "`" sinteger


sapplicative_pattern := "case" sname sargument_pattern
sapplicative_pattern := "`" sname sargument_pattern
  sargument_pattern := satomic_pattern
sapplicative_pattern := satomic_pattern
```

Atomic Pattern

Syntax

```
  satomic_pattern := sname
  satomic_pattern := "?" sname
  satomic_pattern := "val" sname
  satomic_pattern := "#" sctor_name
  satomic_pattern := "#" "case" sname
  satomic_pattern := "`" sname
  satomic_pattern := "case" sinteger
  satomic_pattern := "`" sinteger

  satomic_pattern := "true"
  satomic_pattern := "false"

  satomic_pattern := "_"
  satomic_pattern := "(" spattern ")"
  satomic_pattern := "(" ")"

  satomic_pattern :=  "(" spat_assign ("," spat_assign )* ")"
    spat_assign := sname "=" spattern =># "`(,_1 ,_3)";

  satomic_pattern :=  "(" spat_assign ("," spat_assign )* "|" sname ")"

  satomic_pattern := "$" "(" sexpr ")"
  satomic_pattern := sliteral
  satomic_pattern := sliteral ".." sliteral

}
```

1.11 Statements

1.11.1 Assignments

Felix primary method of setting store is the intrinsic *_storeat*:

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

The library procedure take a pointer or write-only point to T and a value V of type T, and calls the system intrinsic _storeat. The parser in turn maps

```
p <- v;
```

to the procedure *storeat*. For simple variables only you can write:

```
x = v;
```

which is notionally sugar for

```
&x <- v;
```

In addition each of the following infix operators calls a two argument procedure with the same name as the operator:

operator	usual meaning for uints
+=	increment
-=	decrement
/=	quotient
*=	product
%=	remainer
<<=	mul 2^N
>>=	div 2^N
^=	bitwise exclusive or
&=	bitwise and
|=	bitwise or

1.11.2 Conditionals

The simplest form of a conditional construction is:

```
if cond do
   stmts
 elif cond do
   stmts
 ...
 else
   stmts
 done
```

The *elif* and *else* clauses are optional. The final *done* does not require a trailing semicolon. The construction is sugar for a collection of labels and gotos, so that it is ok to put labels in the controlled statements and jump into the middle of a conditional with a goto.

A simplified form is drives a single statement:

```
if cond perform stuff;
```

A more advanced statement is:

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

The final endmatch and semicolon is mandatory to distinguish the construction from a match expression. If none of the pattern match the program aborts with a match failure exception.

1.11.3 Loops

While loops

Syntax:

```
loop_stmt := optlabel "while" sexpr block
loop_stmt := optlabel "repeat" block
loop_stmt := optlabel "until" sexpr block
```

Here is a while loop:

```
while x>1 do
  println$ x;
  --x;
done
```

The loop body executes repeatedly until the condition is not satisfied. If the condition is initially unsatisfied the body is not executed.

The *repeat* loops is an infinte loop equivalent to *while true*.

The *until* loop is a *while* loop with a negated condition.

C style for loop

Syntax:

```
loop_stmt := optlabel "for" "(" stmt sexpr ";" stmt ")" stmt
loop_stmt := optlabel "for" stmt "while" sexpr ";" "next" stmt block
loop_stmt := optlabel "for" stmt "until" sexpr ";" "next" stmt block
```

The first two forms execute a statement once which generally assigns a value to a control variable. The expression must be of type *bool* and is checked to see if the loop should execute. The C form and the *while* form check the condition whilst the *until* form checks the negated condition. The *next* statement is used to increment the control variable.

Integer For Loops

Syntax:

```
loop_stmt := optlabel "for" sname "in" sexpr "upto" sexpr block
loop_stmt := optlabel "for" "var" sname ":" sexpr "in" sexpr "upto" sexpr block
loop_stmt := optlabel "for" "var" sname "in" sexpr "upto" sexpr block
loop_stmt := optlabel "for" sname "in" sexpr "downto" sexpr block
loop_stmt := optlabel "for" "var" sname ":" sexpr "in" sexpr "downto" sexpr block
loop_stmt := optlabel "for" "var" sname "in" sexpr "downto" sexpr block
```

These are low level for loops which operate over inclusive ranges. These loops require an integral control variable. The forms with *var* create the control variable, the other forms require it already exist. The control variable is available after these loops execute.

The block of these forms do not constitue a scope, the loops are implemented with gotos. Therefore you can put labels inside the blocks and goto them, and you can return from the current procedure or function inside the block.

Parallel Loop

Syntax:

```
loop_stmt := "pfor" sname "in" sexpr "upto" sexpr block
```

The *pfor* loop requires the body of the loop to behave independently of other iterations. the range is split into N parts and N pthreads are executed, each one handling a subrange of the loop. The threads run in the system thread pool. N is chosen by the system depending on the thread pool size and/or number of available cores.

pfor loops *must not be nested*. The reason is that the *pfor* loop uses the system thread pool. It is safe in general for jobs in the thread pool to enqueue jobs to the thread pool. It is also inefficient because the thread pool already executes N threads concurrently, where N is roughly equal to the number of processor core available.

It is not necessary to initialise the thread pool to use a *pfor* loops, it will be done automatically. However since the thread pool *is* used, it *must* be destroyed to terminate the program.

Generic Loops

Syntax:

```
loop_stmt := optlabel "for" sname "in" sexpr block
loop_stmt := optlabel "rfor" sname "in" sexpr block
loop_stmt := optlabel "match" spattern "in" sexpr block =>#
```

The generic *for* requires an function named *iterator*. You can provide it directly, or, you can provide any data structure which has an iterator method (that is, a function named iterator which accepts the data structure as an argument). The iterator will usually be a yielding generator and it must return an option type *opt[T]*.

The loops process the *Some x* values yielded until *None* is found.

The control variables goes out of scope at the end of the loop.

The *for* variant uses a goto to loop around.

The *rfor* variant uses recursion instead. The recursion will be flattened to a goto loop if it is safe, otherwise *rfor* will create a frame for every iteration.

The *match* variant sets more than one variable by decoding the argument of the *Some* constructor.

Labelled loops

Most loops allow an optional label which is written with just a *:* suffix. You cannot goto such a label. The label is a name for the loop.

Labelled loops support labeled *break*, *continue* and *redo* statements.

```
 doit: for var i in 1 upto 10 do
   if i == 5 continue doit;
   if i == 7 do
      ++i;
      redo doit;
   done
   if i == 9 break doit;
done
```

The *continue* statement jumps to the start of the selected loop, adjusting the control variable as usual before checking.

The *break* statement exits the selected loop immediately.

The *redo* statement restarts the body of the selected loop without adjusting the control variable and without checking it.

1.11.4 Assertions

Assert

```
assertion_stmt := "assert" sexpr ";"
```

The usual assert statement. Abort the program if the argument expression evaluates to false when control flows through the assert statement. Cannot be switched off!

Axiom

```
assertion_stmt  := "axiom" sdeclname sfun_arg ":" sexpr ";"
```

Define an axiom with a general predicate. An axiom is a function which is true for all arguments. Axioms are core assertions about invariants which can be used to specify semantics and form the basis of reasoning about semantics which goes beyond structure.

```
assertion_stmt  := "axiom" sdeclname sfun_arg ":" sexpr "=" sexpr ";"
```

A variant of an axiom which expresses the semantic equality of two expressions. Do not confuse this with an expresion containing run time equality (==). Semantic equality means that one expression could be replaced by the other without any observable difference in behaviour in any program, this can be asserted even if the type does not provide an equality operator (==).

Lemma

```
assertion_stmt  := "lemma" sdeclname sfun_arg ":" sexpr ";"
assertion_stmt  := "lemma" sdeclname sfun_arg ":" sexpr "=" sexpr ";"
```

A lemma is a proposition which it is expected could be proved by a good automatic theorem prover, given the axioms.

Theorem

```
assertion_stmt  := "theorem" sdeclname sfun_arg ":" sexpr proof? ";"
assertion_stmt  := "theorem" sdeclname sfun_arg ":" sexpr "=" sexpr proof? ";"
```

A theorem is a proposition which it is expected could NOT be proved by a good automatic theorem prover, given the axioms. In the future, we might like to provide a "proof sketch" which a suitable tool could fill in. For the present, you can give a proof as a plain text in a string as a hint to the reader.

Reduction

```
assertion_stmt  := "reduce" sname "|"? sreductions ";"
  private sreduce_args := "(" stypeparameter_comma_list ")"
  private sreduction := stvarlist sreduce_args ":" sexpr "=>" sexpr
  private sreductions := sreduction
  private sreductions := sreduction "|" sreductions
```

A reduction is a special kind of proposition of equational form which also directs the compiler to actually replace the LHS expression with the RHS expression when found.

Reductions allow powerful high level optimisations, such as eliminating two successive list reversals.

The client must take great care that reductions don't lead to infinite loops. Confluence isn't required but is probably desirable.

Reductions should be used sparingly because searching for patterns to reduce is applied to every sub-expression of every expression in the whole program, repeatedly after any reduction is applied, and this whole process is done at several different places in the program, to try to effect the reductions. Particularly both before and after inlining, since that can destroy or create candidate patterns.

1.11.5 Goto and Labels

Code may be marked with labels and jumps to any visible label can be written. Gotos, however, may not jump though a function. A goto can cross procedure boundaries, or it can be a local goto within a function.

```
  var x = 10;
again:>
  println$ x;
  if x == 0 goto finish;
  --x;
  goto again;
finish:>
  println$ "Done";
```

If the goto is to a label in the current context, it is called a local goto.

```
proc f () {
  println$ "This is f";
  goto finish;
}

  f ();
finish:>
  println$ "Done";
```

If the goto is to a label in a surrounding context, it is called a non-local goto. Non-local gotos may convert to local gotos as a result of compiler optimisations such as inlining. Local gotos can never become non-local.

A procedure containing a non-local goto may be passed as an argument to another procedure:

```
  proc f () {
    println$ "This is f";
    goto finish;
  }
  proc g (h: 1-> 0) {
    h ();
  }
  g();
finish:>
  println$ "Done";
```

This can be used to provide error handling or an abnormal exit. Be sure that the context of the target label is active or the result may be unpredictable.

Labels are first class values of type *LABEL* and can be stored in variables:

```
  var lab : LABEL =
    if c then tr else fa endif
  ;
  goto lab;
tr:>
  println$ "True";
  goto finish;
fa:>
  println$ "False";
finish:>
  println$ "Done";
```

As with non-local gotos, the programmer must ensure the context of the target is live at the time a goto is done.

Label values encapsulate both the target code address and its context at the time they're created. Note that contexts are identified by frame address and frames are mutable. In particular the return address of a frame can be zeroed out by the system if the frame returns.

The library contains the following low level operation:

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

which can be used to implement coroutines. Branch and link works by jumping to the label stored in the selected *target*, whilst saving the current location in the store pointed at by *save*. The target routine can then call for a branch to the saved value, providing a store to save its own current location. For example this allows two routines to regularly exchange control.

```
  var l1: LABEL;
  var l2: LABEL = p1;
  println$ "Start";
  branch-and-link (&l2, &l1);
  println$ "p2";
  branch-and-link (&l2, &l1);
```

(continues on next page)

(continued from previous page)

```
  // not reached

p1:>
  println$ "p1";
  branch-and-link (&l1, &l2);
  println$ "Finish";
```

The value stored in a label is converted to a continuation by setting the contination frames current program counter to the code address of the label, overwriting the previous program counter. The goto then make the modified continuation the current continuation of the current fibre and resumes it.

Local direct gotos are optimised by eliding the continuation, since by definition the context of the goto and the context of the target are the same.

The current continuation of an executing procedure can be obtained with the unit function *current_continuation*, it returns the current procedure frame which has type *cont*. It is just the C++ *this* pointer of the procedures activation record:

```
fun current_continuation: unit -> cont = "this";
```

A continuation can be invoked by throwing it:

```
proc _throw: cont
```

The current position within the continuation is of type LABEL and is a function of a continuation value:

```
fun current_position : cont -> LABEL;
```

The implicit entry point of a continuation or procedure closure can be found with the *entry_label* function:

```
fun entry_label : cont -> LABEL;
fun entry_label[T] (p:T->0):LABEL;
```

1.11.6 Subroutine Calls

Call

The *call* statement invokes a procedure:

```
call f x;
call g ();
g ();
g;
#g;
call h.1 x;
```

The word *call* can be elided. If the procedure has a unit argument, it can be elided.

Return

A plain *return* returns from a procedure. An implicit return is added to the end of a procedure.

Return from

A *return from* can be used to exit an outer procedure.

```
proc f () {
  proc g() {
    if c do
      return;
    else
      return from f;
    done
  }
  f();
}
```

Jump

A *jump* is a tail call. It is equivalent to a call followed by a return.

```
proc w() { println$ " World"; }
proc hw () { println$ "Hello"; jump w(); }
```

Yield

The *yield* statements returns a value whilst saving the current program counter in a hidden local variable so that a subsequent invocation of a generator restarts just after the yield. For this to work a closure of a generator must be stored and used for applications, for example in a variable.

If a generator contains a *yield* statement it is called a *yielding generator*. Iterators are usually yielding generators.

```
gen iterator[T] (var tail: list[T]) () = {
again :>
  match tail with
  | Cons (hd, tl) =>
    yield Some head;
    tail = tl;
    goto again;
  | Empty => return None[T];
  endmatch;
}

var lst = ([ 1,2,3 ]);
var it = iterator lst;
var elt = it ();
again:>
  match elt with
  | Some =>
    println$ v;
    goto again;
  | None => ;
  endmatch;
```

1.11.7 Traps

The *call_with_trap* operation is a special variant of a call in which an exception handling trap is established and then the procedure called on the given argument.

Inside the procedure an error handling procedure is defined and passed to client code.

The client code can then use *throw_continuation* to throw the error handler. The error handler is then called in the context of the *call_with_trap* which should be the context of its definition.

```
call_with_trap {
  proc ehandler() {
    eprintln("BATCH MODE ERROR HANDLER");
    result = 1;
    goto err;
   }
   result = runit(ehandler);
 err:>
};
proc runit (ehandler: 1->0) {
  throw_continuation ehandler;
}
```

In this case the error handler does a non-local goto to exit, and jumps to a label at the end of the anonymous procedure which was called with a trap, then that procedure exits normally.

Continuations can be thrown inside functions, and are implemented with a C++ throw which unwinds the machine stack. However procedures use a spaghetti stack consisting of heap allocated stack frames. The top level scheduler guards invocations of procedural continuations with a C++ catch clause, however compiler generated procedure calls may elide the guard for performance reasons.

The *call_with_trap* operation ensures the system scheduler handles the call of the procedure, instead of optimised generated code.

When the scheduler guard catches a continuation, it discards the currently running continuation of the current fibre, and replaces it with the continuation which it caught.

Be sure to use both throws and long jumps with care as neither are intrinsically safe in the following sense: it is possible to throw or jump to code in a continuation which has already exited. A non-local goto resets the continuations program counter to the selected target and executes the exhausted frame until it returns. However the return has already been taken. The system may choose to zero out the return address of a frame when it returns, in which case a second return will terminate the fibre .. but not before it reaches the return instruction.

1.11.8 Try and Catch

The *try/catch* construction is used to provide a *catchable block* with the ability to catch and process C++ exceptions.

Syntax

```
block := "try" stmt+ catches "endtry" =>#
catch := "catch" sname ":" sexpr  "=>" stmt+
catches := catch+
```

Description

Felix does not generate user catchable C++ exceptions. When Felix does throw an exception it is a fatal error and will be caught by the exception handling library, a diagnostic printed, and the process terminated.

C++ primitives, however, can throw exceptions which need to be caught.

For example:

```
body bad_def = "struct bad{};";

body hello_def = """
  void hello() {
    throw bad();
  }
""" requires bad_def;

type bad = "bad" requires bad_def;
proc hello: 1 = "hello();" requires hello_def;

try
  hello();
catch badval: bad =>
  println$ "bad";
endtry
println$ "Done";
```

The constraint on the body of the catch is this: Felix most general method of calling a procedure is to save the current continuation, construct the procedure object on the heap, and return a pointer to the procedure object to the system scheduler.

However, *try/catch* is implemented with a C++ *try/catch* which means a standard procedure call will lead to a switch case label inside the *try* body, which is not allowed in C++. Furthermore, if an except were thrown by the called procedure it would unwind the machine stack and end up unwinding the scheduler subroutine call as well, finally being caught by the exception abort trap wrapping the scheduler.

The problem is, procedures use the Felix spaghetti stack of linked heap objects, not the machine stack.

Actually some procedure calls uses a compiler generated micro-scheduler repeatedly calling the target *resume()* method until it returns NULL, and then continues with the current procedure. This, in effect, does use the machine stack, but the micro-scheduler doesn't catch general C++ exceptions. The micro-scheduler is only used if the compiler is sure the procedure does not do any service calls.

Additionally, some procedures reduce to ordinary C procedures which are then called directly. These use the machine stack, and *try/catch* will work on them because they're equivalent to calling a primitive, which of course is also a C procedure.

The long and short of all this is that the ony *reliable* use of *try/catch* is around a list of calls to primitives or inlined Felix procedures with this property, recursively.

1.11.9 Debugging

Type Checking

```
stmt := "type-error" stmt
stmt := "type-assert" stmt
```

These two statements are a programming aid which allows the programmer to write a dummy statement which is expected to contain a type error or not, respectively.

The type-error form causes a compile time abort of the statement does not contain a type error. Its primary use is pedagogical, to show type errors in syntax coloured code.

The type-assert form causes a compile time error if the statement does not type check. If the statement does type check, it is removed from the program, generating no actual code. Its primary use is to validate presence of required overloads. Note that in Felix type checking and overload resolution are the same process.

Tracing

```
stmt := "trace" sname sstring
```

The trace statement emits a debug message. Trace code generation is disabled by default. Consider this program:

```
proc checker() {
  trace checkname "Trace checker";
  println$ "Checker run";
}

for i in 1..3 perform checker;
```

Here is a run:

```
~/felix>flx --force  x
Checker run
Checker run
Checker run
```

To enable trace code generation the *FLX_ENABLE_TRACE* switch must be set for C++ compilation:

```
~/felix>flx --force --cflags=-DFLX_ENABLE_TRACE x
1 : TRACE: Trace checker
Felix location: /Users/skaller/felix/x.flx 3[3]-3[34]
C++ location  : /Users/skaller/.felix/cache/text/Users/skaller/felix/x.cpp 72
Checker run
2 : TRACE: Trace checker
Felix location: /Users/skaller/felix/x.flx 3[3]-3[34]
C++ location  : /Users/skaller/.felix/cache/text/Users/skaller/felix/x.cpp 72
Checker run
3 : TRACE: Trace checker
Felix location: /Users/skaller/felix/x.flx 3[3]-3[34]
C++ location  : /Users/skaller/.felix/cache/text/Users/skaller/felix/x.cpp 72
Checker run
```

UDP Tracing

Tracing via UDP is available on Unix systems, it is not currently implemented for Windows.

```
Debug::enable_local_udp_trace;

proc checker() {
  Debug::send_udp_trace_message "HIHI";
  println$ "Checker run";
```

(continues on next page)

(continued from previous page)

```
}

for i in 1..3 perform checker;
```

UDP tracing must be enabled. This creates a UPD socket on port 1153 of IP address *127.0.0.1*.

The output from the UDP socket can be monitored by the C++ program *flx_udp_trace_monitor*. This is a stand-alone C++ program found in the *src/tools* directory. You have to compile this by hand.

```
~/felix>clang++ src/tools/flx_udp_trace_monitor.cxx
~/felix>./a.out
UDP Trace Monitor Listening on port 1153
```

Make sure to start the monitor in a terminal first. Now you can run the Felix program:

```
~/felix>flx --force  x
Bound Trace Output Socket OK!
First UDP Trace message sent OK! 4 bytes = 'HIHI'
Checker run
Checker run
Checker run
```

The monitor will show:

```
Received = 4
Buffer = HIHI
Received = 4
Buffer = HIHI
Received = 4
Buffer = HIHI
```

1.12 Library Algebras

Contents:

1.12.1 Sets

Sets provide an alternate representation of logical operators.

Syntax

```
syntax setexpr
{
  cmp := "in"
  cmp := "\in"
  cmp := "\notin"
  cmp := "\owns" ;

  x[ssetunion_pri] := x[ssetunion_pri] "\cup" x[>ssetunion_pri]
  x[ssetintersection_pri] := x[ssetintersection_pri] "\cap" x[>ssetintersection_pri]
}
```

Semantics

```
class Set[c,t] {
  fun mem (elt:t, container:c):bool => elt \in container;
  virtual fun \in : t * c-> bool;
  fun \owns (container:c, elt:t) => elt \in container;
  fun \ni (container:c, elt:t) => elt \in container;
  fun \notin (elt:t, container:c) => not (elt \in container);

  fun \cup[c2 with Set[c2,t]]
    (x:c, y:c2) =>
    { e : t | e \in x or e \in y }
  ;

  fun \cap[c2 with Set[c2,t]]
    (x:c, y:c2) =>
    { e : t | e \in x and e \in y }
  ;

  fun \setminus[c2 with Set[c2,t]]
    (x:c, y:c2) =>
    { e : t | e \in x and e \notin y }
  ;
}
```

A *set* is any type with a membership predicate ∈ spelled *\in*. You can also use function *mem*. The parser also maps *in* to operator *\in*.

We also provide a reversed form :math::*owns* spelled *\owns*, and negated forms ∋ spelled *ni* or *notin*.

Three combinators are provided as well, ∩ spelled *\cap* provides intersection, ∪ spelled *\cup* provides the usual set union, and \ spelled *setminus* the asymmetic set difference or subtraction.

Note that sets are not necessarily finite.

1.12.2 Set Forms

Syntax

```
setbar := "|" =># "_1";
setbar := "\|" =># "_1";
setbar := "\mid" =># "_1";

setform := spattern ":" stypeexpr setbar sexpr

satom := "{" setform  "}" =># "_2";
satom := "\{" setform  "\}" =># "_2";
```

Definition

```
interface set_form[T] { has_elt: T -> bool; }

instance[T] Set[set_form[T], T] {
  fun \in (elt:T, s:set_form[T]) => s.has_elt elt;
```

(continues on next page)

(continued from previous page)

```
}

open[T] Set[set_form[T],T];
```

Description

A *set_form* is a record type with a single member *has_elt* which returns true if it's argument is intended as a member of some particular set.

We construe a set_form as a Set by providing an instance.

A set_form is basically just the membership predicate remodelled as a noun by encapsulating the predicate in a closure and thereby abstracting it.

We provide an inverse image:

```
fun invimg[t,c2,t2 with Set[c2,t2]]
  (f:t->t2, x:c2) : set_form[t] =>
  { e : t | (f e) \in x }
;
```

Cartesian Product of set_forms.

This uses some advanced instantiation technology to allow you to define the cartesian product of a sequence of sets using the infix TeX operator $\otimes$ which is spelled *otimes*. There's also a left associative binary operator $\times$ spelled *times*.

Operators

```
fun \times[U,V] (x:set_form[U],y:set_form[V]) =>
  { u,v : U * V | u \in x and v \in y }
;

fun \otimes[U,V] (x:set_form[U],y:set_form[V]) =>
  { u,v : U * V | u \in x and v \in y }
;

fun \otimes[U,V,W] (head:set_form[U], tail:set_form[V*W]) =>
  { u,v,w : U * V * W | u \in head and (v,w) \in tail }
;

fun \otimes[NH,OH,OT] (head:set_form[NH], tail:set_form[OH**OT]) =>
  { h,,(oh,,ot) : NH ** (OH ** OT) | h \in head and (oh,,ot) \in tail }
;
```

Example:

```
var p = \{ x,y: int * int | x == y \};
println$ (1,1) in p;
```

1.12.3 Containers

Roughly, a Container is a finite Set. It is a derived type specified in the library with a type class:

```
class Container [c,v]
{
  inherit Set[c,v];
  virtual fun len: c -> size;
  fun \Vert (x:c) => len x;
  virtual fun empty(x: c): bool => len x == size(0);
}
```

The || operator, spelled *Vert* is an alternative name for *len*.

1.12.4 Address Types

Several special types are provided for raw memory manipulation: *caddress*, *address* and *byte*.

An address is a machine address which can be treated as an unsigned integer, and also supports the dereference operator. All standard pointers can be converted to type *address* and *caddress*.

The dereference function *deref* which can also be written with a prefix * as in C, returns a value of type *byte*. A byte is an 8 bit unsigned integer similar to *uint8*.

1.12.5 Equivalence Relation

Syntax

```
syntax cmpexpr
{
  x[scomparison_pri]:= x[>scomparison_pri] cmp x[>scomparison_pri] =># "`(ast_apply ,_
↪sr (,_2 (,_1 ,_3)))";
  x[scomparison_pri]:= x[>scomparison_pri] "not" cmp x[>scomparison_pri] =># "`(ast_
↪not ,_sr (ast_apply ,_sr (,_3 (,_1 ,_4))))";
  cmp := "==" =># "(nos _1)";
  cmp := "!=" =># "(nos _1)";
  cmp := "\ne" =># '(nos _1)';
  cmp := "\neq" =># '(nos _1)';
}
```

Semantics

```
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;

  fun eq(x:t, y:t)=> x == y;
  fun ne(x:t, y:t)=> x != y;
  fun \ne(x:t, y:t)=> x != y;
  fun \neq(x:t, y:t)=> x != y;
}
```

1.12.6 Partial Order

Syntax

```
syntax pordcmpexpr
{
  cmp := "\subset" =># '(nos _1)';
  cmp := "\supset" =># '(nos _1)';
  cmp := "\subseteq" =># '(nos _1)';
  cmp := "\subseteqq" =># '(nos _1)';
  cmp := "\supseteq" =># '(nos _1)';
  cmp := "\supseteqq" =># '(nos _1)';

  cmp := "\nsubseteq" =># '(nos _1)';
  cmp := "\nsubseteqq" =># '(nos _1)';
  cmp := "\nsupseteq" =># '(nos _1)';
  cmp := "\nsupseteqq" =># '(nos _1)';

  cmp := "\subsetneq" =># '(nos _1)';
  cmp := "\subsetneqq" =># '(nos _1)';
  cmp := "\supsetneq" =># '(nos _1)';
  cmp := "\supsetneqq" =># '(nos _1)';
}
```

Semantics

```
class Pord[t]{
  inherit Eq[t];
  virtual fun \subset: t * t -> bool;
  virtual fun \supset(x:t,y:t):bool =>y \subset x;
  virtual fun \subseteq(x:t,y:t):bool => x \subset y or x == y;
  virtual fun \supseteq(x:t,y:t):bool => x \supset y or x == y;

  fun \subseteqq(x:t,y:t):bool => x \subseteq y;
  fun \supseteqq(x:t,y:t):bool => x \supseteq y;

  fun \nsubseteq(x:t,y:t):bool => not (x \subseteq y);
  fun \nsupseteq(x:t,y:t):bool => not (x \supseteq y);
  fun \nsubseteqq(x:t,y:t):bool => not (x \subseteq y);
  fun \nsupseteqq(x:t,y:t):bool => not (x \supseteq y);

  fun \supsetneq(x:t,y:t):bool => x \supset y;
  fun \supsetneqq(x:t,y:t):bool => x \supset y;
  fun \supsetneq(x:t,y:t):bool => x \supset y;
  fun \supsetneqq(x:t,y:t):bool => x \supset y;

  axiom trans(x:t, y:t, z:t): \subset(x,y) and \subset(y,z) implies \subset(x,z);
  axiom antisym(x:t, y:t): \subset(x,y) or \subset(y,x) or x == y;
  axiom reflex(x:t, y:t): \subseteq(x,y) and \subseteq(y,x) implies x == y;
}
```

1.12.7 Total Order

Semantics

```
class Tord[t]{
  inherit Eq[t];
  // defined in terms of <, argument order swap, and boolean negation

  // less
  virtual fun < : t * t -> bool;
  fun lt (x:t,y:t): bool=> x < y;
  fun \lt (x:t,y:t): bool=> x < y;
  fun \lneq (x:t,y:t): bool=> x < y;
  fun \lneqq (x:t,y:t): bool=> x < y;


  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;


  // greater
  fun >(x:t,y:t):bool => y < x;
  fun gt(x:t,y:t):bool => y < x;
  fun \gt(x:t,y:t):bool => y < x;
  fun \gneq(x:t,y:t):bool => y < x;
  fun \gneqq(x:t,y:t):bool => y < x;

  // less equal
  fun <= (x:t,y:t):bool => not (y < x);
  fun le (x:t,y:t):bool => not (y < x);
  fun \le (x:t,y:t):bool => not (y < x);
  fun \leq (x:t,y:t):bool => not (y < x);
  fun \leqq (x:t,y:t):bool => not (y < x);
  fun \leqslant (x:t,y:t):bool => not (y < x);


  // greater equal
  fun >= (x:t,y:t):bool => not (x < y);
  fun ge (x:t,y:t):bool => not (x < y);
  fun \ge (x:t,y:t):bool => not (x < y);
  fun \geq (x:t,y:t):bool => not (x < y);
  fun \geqq (x:t,y:t):bool => not (x < y);
  fun \geqslant (x:t,y:t):bool => not (x < y);

  // negated, strike-through
  fun \ngtr (x:t,y:t):bool => not (x < y);
  fun \nless (x:t,y:t):bool => not (x < y);

  fun \ngeq (x:t,y:t):bool => x < y;
  fun \ngeqq (x:t,y:t):bool => x < y;
  fun \ngeqslant (x:t,y:t):bool => x < y;

  fun \nleq (x:t,y:t):bool => not (x <= y);
  fun \nleqq (x:t,y:t):bool => not (x <= y);
  fun \nleqslant (x:t,y:t):bool => not (x <= y);

  // maxima and minima
```

(continues on next page)

(continued from previous page)

```
  fun max(x:t,y:t):t=> if x < y then y else x endif;
  fun \vee(x:t,y:t) => max (x,y);

  fun min(x:t,y:t):t => if x < y then x else y endif;
  fun \wedge(x:t,y:t):t => min (x,y);
}
```

Syntax

```
syntax tordcmpexpr
{
  cmp := "<" =># "(nos _1)";

  cmp := "\lt" =># '(nos _1)';
  cmp := "\lneq" =># '(nos _1)';
  cmp := "\lneqq" =># '(nos _1)';

  cmp := "<=" =># "(nos _1)";
  cmp := "\le" =># '(nos _1)';
  cmp := "\leq" =># '(nos _1)';
  cmp := "\leqq" =># '(nos _1)';

  cmp := ">" =># "(nos _1)";
  cmp := "\gt" =># '(nos _1)';
  cmp := "\gneq" =># '(nos _1)';
  cmp := "\gneqq" =># '(nos _1)';

  cmp := ">=" =># "(nos _1)";
  cmp := "\ge" =># '(nos _1)';
  cmp := "\geq" =># '(nos _1)';
  cmp := "\geqq" =># '(nos _1)';

  cmp := "\nless" =># '(nos _1)';
  cmp := "\nleq" =># '(nos _1)';
  cmp := "\nleqq" =># '(nos _1)';
  cmp := "\ngtr" =># '(nos _1)';
  cmp := "\ngeq" =># '(nos _1)';
  cmp := "\ngeqq" =># '(nos _1)';

  bin := "\vee" =># '(nos _1)';
  bin := "\wedge" =># '(nos _1)';
}
```

1.12.8 Additive Groups

Syntax

```
syntax addexpr
{
  //$ Addition: left non-associative.
  x[ssum_pri] := x[>ssum_pri] ("+" x[>ssum_pri])+ =># "(chain 'ast_sum _1 _2)" note
↪"add";
```

(continues on next page)

(continued from previous page)

```
  //$ Subtraction: left associative.
  x[ssubtraction_pri] := x[ssubtraction_pri] "-" x[sproduct_pri] =># "(Infix)";
}
```

Semantics

```
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); }
}

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;
}
```

1.12.9 Multiplicative SemiGroup with Unit

Syntax

```
syntax mulexpr
{
  //$ multiplication: non-associative.
  x[sproduct_pri] := x[>sproduct_pri] ("*" x[>sproduct_pri])+ =>#
    "(chain 'ast_product _1 _2)" note "mul";
}
```

Semantics

```
class FloatMultSemi1[t] {
  inherit Eq[t];
  proc muleq(px:&t, y:t) { *= (px,y); }
  fun mul(x:t, y:t) => x * y;
  fun sqr(x:t) => x * x;
  fun cube(x:t) => x * x * x;
  virtual fun one: unit -> t;
  virtual fun * : t * t -> t;
  virtual proc *= (px:&t, y:t) { px <- *px * y; }
  //reduce id (x:t): x*one() => x;
  //reduce id (x:t): one()*x => x;
}

class MultSemi1[t] {
  inherit FloatMultSemi1[t];
  axiom assoc (x:t,y:t,z:t): (x * y) * z == x * (y * z);
  //reduce cancel (x:t,y:t,z:t): x * z ==  y * z => x == y;
}
```

Description

A multiplicative semigroup with unit is an approximate multiplicative semigroup with unit and associativity and satisfies the cancellation law.

1.12.10 Rings

Syntax

```
syntax divexpr
{
  //$ division: right associative low precedence fraction form
  x[stuple_pri] := x[>stuple_pri] "\over" x[>stuple_pri]

  //$ division: left associative.
  x[s_term_pri] := x[s_term_pri] "/" x[>s_term_pri]

  //$ remainder: left associative.
  x[s_term_pri] := x[s_term_pri] "%" x[>s_term_pri]

  //$ remainder: left associative.
  x[s_term_pri] := x[s_term_pri] "\bmod" x[>s_term_pri]
}
```

Semantics

```
// Approximate Ring with Unit
class FloatRing[t] {
  inherit FloatAddgrp[t];
  inherit FloatMultSemi1[t];
}
```

(continues on next page)

(continued from previous page)

```
// Ring with Unit
class Ring[t] {
  inherit Addgrp[t];
  inherit MultSemi1[t];
  axiom distrib (x:t,y:t,z:t): x * ( y + z) == x * y + x * z;
}

// Approximate Division Ring
class FloatDring[t] {
  inherit FloatRing[t];
  virtual fun / : t * t -> t; // pre t != 0
  fun \over (x:t,y:t) => x / y;

  virtual proc /= : &t * t;
  virtual fun % : t * t -> t;
  virtual proc %= : &t * t;

  fun div(x:t, y:t) => x / y;
  fun mod(x:t, y:t) => x % y;
  fun \bmod(x:t, y:t) => x % y;
  fun recip (x:t) => #one / x;

  proc diveq(px:&t, y:t) { /= (px,y); }
  proc modeq(px:&t, y:t) { %= (px,y); }
}

// Division Ring
class Dring[t] {
  inherit Ring[t];
  inherit FloatDring[t];
}
```

Description

Approximate Unit Ring. An approximate ring is a set which has addition and multiplication satisfying the rules for approximate additive group and multiplicative semigroup with unit respectively.

Ring. A ring is a type which is a both an additive group and multiplicative semigroup with unit, and which in addition satisfies the distributive law.

Approximate Division Ring. An approximate division ring is an approximate ring with unit with a division operator.

Division Ring. An associative approximate division ring.

1.12.11 Integral Algebra

Contents:

Integer Algebra

Semantics

```
//$ Integers.
class Integer[t] {
  inherit Tord[t];
  inherit Dring[t];
  inherit Bidirectional[t];
  virtual fun << : t * t -> t = "$1<<$2";
  virtual fun >> : t * t -> t = "$1>>$2";

  fun shl(x:t,y:t)=> x << y;
  fun shr(x:t,y:t)=> x >> y;

  virtual fun maxval: 1 -> t = "::std::numeric_limits<?1>::max()";
  virtual fun minval: 1 -> t = "::std::numeric_limits<?1>::min()";

}

//$ Signed Integers.
class Signed_integer[t] {
  inherit Integer[t];
  virtual fun sgn: t -> int;
  virtual fun abs: t -> t;
}

//$ Unsigned Integers.
class Unsigned_integer[t] {
  inherit Integer[t];
  inherit Bits[t];
}
```

Integer Types

Felix supports several families of integer types.

Integers Types

Platform variant integer types are based on C integer types whose sizes vary from platform to platform. In the tables below, the suffix is appended to a plain integer literal as a type modifier. The suffix letters can be in any order and can be upper or lower case.

ISO C Signed Integer Types

The size in bytes on Windows 32 and 64 bit platforms and Unix 32 and 64 bit platfoms is indicated.

Felix	C	Suffix	W32	W64	U32	U64
tiny	signed char	t	1	1	1	1
short	short	s	2	2	2	2
int	int		2	2	4	4
long	long	l	4	4	4	8
vlong	long long	vl,lv	8	8	8	8

Int type may also use *i* suffix.

ISO C Unsigned Integer Types

All unsigned integer types provide a range of values from 0 to $256^n - 1$ for some n. The size in bytes is the same as the corresponding signed type.

Felix	C	Suffix
utiny	usigned char	ut
ushort	unsigned short	us
uint	unsigned int	u
ulong	unsigned long	ul
uvlong	unsigned long long	uvl,ulv

ISO C and Posix Aliases

Felix	C	Suffix
intmax	intmax_t	j
uintmax	uintmax_t	uj
ssize	ssize_t	z
size	size_t	uz
intptr	intptr_t	p
uintptr	uintptr_t	up
ptrdiff	ptrdiff_t	d
uptrdiff	uptrdiff_t	ud

The intmax and uintmax types are 8 bytes on Windows and Unix platforms. The other types are all the same as the machine word size, either 4 or 8 bytes.

Exact Signed Integer Types

Felix	C	Suffix
int8	int8_t	i8
int16	int16_t	i16
int32	int32_t	i32
int64	int64_t	i64

Exact Unigned Integer Types

Felix	C	Suffix
uint8	uint8_t	u8
uint16	uint16_t	u16
uint32	uint32_t	u32
uint64	uint64_t	u64

Integer Literals

An integer literal consists of an optional radix indicator, a string of digits with possible embedded spacers, and an optional suffix.

Radices

Felix supports 4 radices.

Radix	Prefix	Digits
Hex	0x	0123456789abcdefABCDEF
Decimal	0d	0123456789
Octal	0o	01234567
Binary	0b	01

If omitted decimal radix if used. Radix letter may be upper or lower case. Note, in Felix a leading zero digit does not imply octal radix as in C.

Spacers

Integers allow an underscore between digits, after the radix specifier if one is given, or before the suffix specifier, if one is given.

Operations

All Integers

All integer types support the following operations.

Operation	Operator
Addition	infix +
Subtraction	infix -
Multiplication	infix *
Division	infix /
Remainder	infix %
Left Shift	infix <<
Right Shift	infix >>

Left and right shifts are defined as multiplication by positive or negative powers of 2, respectively.

Signed Integers Only

Operation	Operator
Negation	prefix -, neg
Sign	sgn
Absolute Value	abs

The *sgn* operator returns -1 for negative, 0 for zero, and 1 for positive.

Unsigned Integer Only

These operations are bitwise logic operations. They are not available for signed integers.

Operation	Operator
ones complement	~
bitand	infix \&
bitor	infix |
bitxor	infix \^

1.12.12 Slices

Felix provides two slice types, these are derived types defined in the library.

Basic Type

A basic slice type:

```
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
  | Slice_none
;
```

Slice Expression

Slice values can be written like:

```
..                    // all
first ..              // first to maxval
first .+ count        // first to first+count exclusive
first .. last         // inclusive range
first ..< past        // range exclusing end
Slice_one val         // single value
Slice_none            // empty slice
```

The base type of a slice must be of class Integral, both signed and unsigned types can be used. If there are two parameters they must be the same type.

Slice Membership

A value can be tested to see if it is in a slice:

```
42 in 1..100
```

Slice iterator

Slices are equipped with iterators. This means you can write a loop like:

```
for v in first .. last
  perform println$ v
;
```

If the slice is given with literal integer values the loop is optimised to bypass the iterator closure and use inline range checks.

1.12.13 Floating Numbers

Approximate Reals

Felix has three floating types:

Felix	C	Suffix	Positive Infinity
float	float	f	FINFINITY
double	double float	d	DINFINITY
ldouble	long double float	l	LINFINITY

where the d suffix can be omitted. The lexical rules are

```
regdef decimal_string = digit (underscore ? digit) *;
regdef hexadecimal_string = hexdigit (underscore ? hexdigit) *;

regdef decimal_fractional_constant =
  decimal_string '.' decimal_string;

regdef hexadecimal_fractional_constant =
  ("0x" |"0X")
  hexadecimal_string '.' hexadecimal_string;

regdef decimal_exponent = ('E'|'e') ('+'|'-')? decimal_string;
regdef binary_exponent = ('P'|'p') ('+'|'-')? decimal_string;

regdef floating_suffix = 'L' | 'l' | 'F' | 'f' | 'D' | 'd';
regdef floating_literal =
  (
    decimal_fractional_constant decimal_exponent ? |
    hexadecimal_fractional_constant binary_exponent ?
  )
  floating_suffix ?;
```

This is consistent with ISO C, except that underscores may be used to separate digits, and a decimal point is required and must be surrounded by digits. Therefore *0.* and *.0* are not permissible floating literals.

IEEE NaN (Not a Number) values can be checked for with the function *isnan*. IEEE Infinity values can be checked for with the function *isinf*.

Approximate Complex

Felix has three complex types based on C++ complex<T> where T is one of the three floating types.

Felix	C++
fcomplex	::std::complex<float>
dcomplex	::std::complex<double>
lcomplex	::std::complex<long double>

Constructors

Complex numbers can be constructed using the typedef name *complex* and two arguments of the same floating real type, or the typedef name *polar* and two arguments of the same floating real type. These construct a complex number in Cartesian and Polar coordinates, respectively.

Approximate Quaternion

Felix provides a single type *quaternion* which is based on float double. It is defined in the library module *Quaternion* which should be opened to use it.

1.12.14 Strings

Felix provides a basic string type, `string` based on C++ `::std::basic_string<char>`. The operations are defined in the library.

String literals can use several lexical forms.

Single quoted strings are delimited by either single or double quote marks, and may not span lines.

Triple quoted strings are delimited by either three single or three double quote marks, and may span multiple lines.

1.13 Data Types

Contents:

1.13.1 Arrays

Contents:

Carray

Blah.

Varray

Blah.

Darray

Blah.

Sarray

Blah

1.13.2 Lists

```
syntax listexpr
{
  //$ List cons, right associative.
  x[sarrow_pri] := x[>sarrow_pri] "!" x[sarrow_pri]

  satom := "(" "[" stypeexpr_comma_list "]" ")"
}
```

Semantics

```
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]

inline fun Cons[T] (h:T, t:list[T]) => Snoc (t,h);
```

Lists in Felix are defined as Snoc lists, in which the order list tail occurs *before* the node data value. This allows some operations to be written in C++ which work for lists, independently of the data type of the stored value T.

However lists are usually used with a Cons operator for which the head value goes first, then the tail. This is arranged by providing a Cons function to construct lists, and by providing user defined pattern matches for the Cons form.

Lists in Felix are purely functional and immutable.

1.13.3 Regular Expressions

Blah.

1.14 Fibres

1.14.1 General Description

The Felix fibration system is implemented by service calls into the run time library.

These calls are basically:

- mk_ioschannel_pair[T]
- read channel

- write (channel, value)
- spawn_fthread
- suicide
- run

1.14.2 Channel construction

A channel is a single object however typically we might say:

```
var inp, out = mk_ioschannel_pair[int]();
```

to create a channel object on the heap, and then cast a pointer to to the object to *ischannel[int]* and a copy to *oschannel[int]*. These types can be abbreviated *%<int* and *%>int* respectively.

1.14.3 Spawning a Fibre

To create a fibre, we just call:

```
spawn_fthread p;
```

where p is any unit procedure to create a fibre of control which starts at the entry point of *p*.

1.14.4 Writing on a channel

We can write data onto a channel with the procedure call:

```
write (out, value);
```

1.14.5 Reading from a channel

We can read data with the psuedo function

```
read inp
```

1.14.6 Termination

A fibre terminates when its initial coroutine returns, it calls *suicide()*, starves or blocks.

1.14.7 Starvation

A fibre starves if it is suspended on a channel which is unable to be written. What this means is that no other active procedure owns the pointer to the channel, in other words, the channel is unreachable. In this case, the fibre evaporates automatically because only active fibres are known to the scheduler, and fibres are otherwise anonymous. Thus the continuation and thus all the stack frames of the fibre are unreachable and can be reaped by the garbage collector.

Thus, fibres cannot deadlock, because if they do they no longer exist. Starvation is equivalent to suicide.

1.14.8 Blockage

A fibre blocks if it is suspended on channel which is unable to be read. A with starvation, blocking is equivalent to suicide.

Note if a procedure stack contains a channel, or data structures which make the channel reachable, then the channel is considered accessible, even if the procedure has no control path which will lead to an I/O attempt on it. For this reason channels should be forgotten except by those using them.

Contrarily, when a channel is reachable and a fibres is suspended on it, if the procedure which can reach it never does so, that is called a *livelock*.

1.14.9 Constructing a new Scheduler

Felix top level mainline code forms a coroutine which the system spawns automatically. However you can also create a sub-scheduler with the procedure call

```
run p;
```

run is a subroutine, it creates a new scheduler object, spawns *p* on that scheduler, and runs the scheduler until there are no active fibres left on that scheduler.

Note that if *p* itself spawns new fibres they will become active on the same scheduler as *p*, however, *fibres can migrate between schedulers*.

1.14.10 Example

Here is a simple example.

```
proc example () {
  var inp,out = mk_ioschannel_pair[int]();
  spawn_fthread {
     for i in 0..9 perform write (out,i);
  };
  spawn_fthread {
    repeat perform println$ read inp;
  };
}
example();
println$ "Done";
```

In the example, we create a channel with a read and write endpoint, and then spawn two fibres. The first one writes 10 numbers and suicides by returning. The seccond one reads 10 numbers and prints them, then suicides by starvation.

It is important to note that the abstract logic does not specify when the *Done* is printed. After a spawn, both the spawner and spawnee are active. After a read and write match up, both the reader and writer are active. The implementation is free to choose which of all the active fibres to run next. However Felix runs the spawnee before the spawner, and it runs the reader before the writer, so the *Done* will actually print last.

The other very important thing to note is that the *example* procedure knows the channels being used, however the channel endpoints are stored in its stack frame, which will become unreachable when *example* returns. Thus, only the reader and writer will have access to the channel at that time, and once the writer has terminated that leaves the reader blocked: it is trying to read from a channel which no active fibre can write on. This, the reader becomes unreachable, and so when the mainline terminates the program is finished.

1.14.11 Binding Channels With HOFs.

A better way to write the code above is to use Higher Order Functions (HOFs).

```
fun make() = {
  typedef r_t = (inp: %<int);
  typedef w_t = (out: %>int);

  proc writer (x: w_t) () {
     for i in 0..9 perform write (x.out,i);
  };
  proc reader (y: r_t) () {
    repeat perform println$ read y.inp;
  };

  var i,o = mk_ioschannel_pair[int]();
  return reader (inp=i), writer (out=o);
}
proc example () {
  var r,w = make();
  spawn_fthread r;
  spawn_fthread w;
}
example();
println$ "Done";
```

Here the reader and writer are functions which take a record argument whose fields are the required channels and return a unit procedure.

1.14.12 Syntactic Supoport

The protocol above is supported by special syntax:

```
chip writer
  connector x
    pin out: %>int
 {
   for i in 0..9 perform write (x.out,i);
 }

 chip reader
   connector y
     pin inp: %<int
 {
    repeat perform println$ read y.inp;
 }

 circuit
   connect writer.out, reader.inp
 endcircuit

 println$ "Done";
```

The *chip* constructions above are exactly the same as the procedures in the previous example. The connectors are the record parameters, the pins are the fields of the record.

The *circuit* statement constructs the channels required to connect the pins automatically, binds them to the parameters, and then spawns the resulting unit procedures as fibres.

1.14.13 Sources, Sinks, and Transducers

What is important to note here is that connectors can have any number of pins. Coroutines are not restricted to using one communication channel.

The writer above, with a single output pin, is called a *source*. The reader above, with a single input pin, is called a *sink*. And the following shows a *transducer*:

```
chip squareit
  connector x
    pin inp: %<int
    pin out: %>int
 {
   repeat do
     var i = read x.inp;
     write (x.out, i*i);
   done
 }

 circuit
   connect writer.out, squareit.inp
   connect squareit.out, reader.inp
 endcircuit
```

1.14.14 Pipelines

When you run a set of coroutines starting with a source, followed by a sequence of transducers, and terminated by a sink, the construction is called a *closed pipeline* and is precisely a unit procedure.

There are special operators to simplify pipeline construction:

```
var pipeline = writer |-> squareit |-> reader;
pipeline ();
```

Pipelines can also be open, if there is no source at the beginning and no sink at the end, or half open, where there is a source at the start but no sink at the end, or no source at the start but a sink at the end.

In fact the pipeline operator is associative:

LHS	RHS	Result
Source	Sink	Closed Pipline
Source	Transducer	Source
Transducer	Transducer	Transducer
Transducer	Sink	Sink

in particular for any legitimate combination:

```
a |-> b |-> c
(a |-> b) |-> c
a |-> (b |-> c)
```

are equivalent.

1.14.15 Library Chips

We can simplify our code again by using standard library chips. Here is the whole program again:

```
proc readit (y:int) { println$ y; }

gen  writeit () : opt[int] = {
  for i in 0..9 perform yield Some i;
  return None[int];
}
fun squareit (x:int) => x * x;

var pipeline = iterate writeit |-> function squareit |-> procedure readit;
pipeline ();
println$ "Done";
```

The *iterate* chip is an adaptor that accepts an iterator and produces a source.

The *function* chip is an adaptor that accepts a function and proceduces a transducer.

The *procedure* chip is an adaptor that accepts a procedure with one argument and produces a sink.

Here's another example:

```
run (
  iterate (1,2,3).iterator |->
  function (fun (x:int) =>  x * x) |->
  procedure (proc (x:int) { println$ x; })
);
```

which prints the squares of the values of an array 1,2,3 in a single line by using anonymous functions and the standard iterator method for arrays.

1.14.16 More Library Chips

writeblock

Starves connected reader.

```
chip writeblock[T]
  connector io
    pin inp : %<T
{
}
```

readblock

Blocks connected writer.

```
chip readblock[T]
  connector io
    pin inp: %>T
{
}
```

sink

Universal sink. Reads input forever.

```
chip sink[T]
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    C_hack::ignore (x);
  done
}
```

source

Writes fixed value forever.

```
chip source[T] (a:T)
  connector io
    pin out: %>T
{
  while true do
    write (io.out, a);
  done
}
```

value

One shot source

```
chip value[T] (a:T)
  connector io
    pin out: %>T
{
  write (io.out, a);
}
```

generator

Writes values acquired from a generator.

```
chip generator[T] (g: 1->T)
  connector io
    pin out: %>T
  {
    repeat perform write (io.out, g());
  }
```

iterate

Writes values acquired from an iterator, terminates when and if iterator becomes exhausted.

```
chip iterate[T] (g: 1->opt[T])
  connector io
    pin out: %>T
  {
    again:>
      var x = g();
      match x with
      | Some v =>
        write (io.out, v);
        goto again;
      | None => ;
      endmatch;
  }
```

source_from_list

A specialised source which writes the values of a list. Terminates at the end of the list.

```
chip source_from_list[T] (a:list[T])
  connector io
    pin out: %>T
{
  for y in a perform write (io.out,y);
}
```

bound_source_from_list

Writes Some x, for each x in the list, then writes an infinite tail of None.

```
chip bound_source_from_list[T] (a:list[T])
  connector io
    pin out: %>opt[T]
{
  for y in a perform write (io.out,Some y);
  while true perform write (io.out,None[T]);
}
```

function

Function adaptor. Converts a function to transducer. Repeatedly reads input, writes result of applying function to it.

```
chip function[D,C] (f:D->C)
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    var y = f x;
    write (io.out, y);
  done
}
```

procedure

Procedure adaptor. Converts a procedure taking one argument to a sink.

```
chip procedure[D] (p:D->0)
  connector io
    pin inp: %<D
{
  while true do
    var x = read io.inp;
    p x;
  done
}
```

filter

Convert a predicate and function to a transducer. Reads value from input, applies function to it, and writes result if it satisfies the predicate. Note the predicate applies to the output of the function, not the input to it.

```
chip filter[D,C] (c:D->bool) (f:D->C)
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    if c x do
       write (io.out, f x);
    done
  done
}
```

filter

A variant of the two argument filter which reads a value, applies the function to it, and checks the resulting option type. If Some v is returned, writes v, if None is returned does not write anything.

```
chip filter[D,C] (f:D->opt[C])
  connector io
    pin inp: %<D
    pin out: %>C
{
  while true do
    var x = read io.inp;
    match f x with
    | Some y => write (io.out, y);
    | None => ;
    endmatch;
  done
}
```

sink_to_list

This chip accepts a pointer to a variable containing a list. Each value read is prepended to the list.

```
chip sink_to_list[T] (p: &list[T])
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    p <- Cons (x,*p);
  done
}
```

sink_to_unique_list

A variant of *sink_to_list* for which the value is prepended to the list if, and only if, it is not already in the list.

```
chip sink_to_unique_list[T with Eq[T]] (p: &list[T])
  connector io
    pin inp : %<T
{
  while true do
    var x = read (io.inp);
    if not (x in *p) perform
      p <- Cons (x,*p)
    ;
  done
}
```

buffer

A single value buffer, equivalent to a function adaptor passed the identity function.

```
chip buffer [T]
  connector io
    pin inp: %<T
    pin out: %>T
{
  while true do
    var x = read io.inp;
    write (io.out, x);
  done
}
```

dup

Copies input to two outputs.

```
chip dup [T]
  connector io
    pin inp: %<T
    pin out1: %>T
    pin out2: %>T
{
  while true do
    var x = read io.inp;
```

(continues on next page)

(continued from previous page)

```
    write (io.out1, x);
    write (io.out2, x);
  done
}
```

debug_buffer

A variant of a buffer which also prints diagnostics before reading, after reading and before writing, and after writing.

```
chip debug_buffer [T with Str[T]] (tag:string)
  connector io
    pin inp: %<T
    pin out: %>T
{
  while true do
    println$ "Debug buffer [" + tag + "] READ";
    var x = read io.inp;
    println$ "Debug buffer [" + tag + "] read " + x.str;
    write (io.out, x);
    println$ "Debug buffer [" + tag + "] written " + x.str;
  done
}
```

oneshot

A one shot buffer. Reads one value and writes it, then terminates.

```
chip oneshot [T]
  connector io
    pin inp: %<T
    pin out: %>T
{
  var x = read io.inp;
  write (io.out, x);
}
```

store

Repeatedly stores read values into a variable.

```
chip store[T] (p:&T)
  connector io
    pin inp: %<T
{
  while true do
    var x = read io.inp;
    p <- x;
  done
}
```

fetch

Repeatedly writes the ccurrent value of a variable.

```
chip fetch[T] (p:&T)
  connector io
    pin out: %>T
{
  while true do
    write (io.out, *p);
  done
}
```

debug_sink

Writes input to standard output.

```
chip debug_sink [T with Str[T]] (s:string)
  connector io
    pin inp: %<T
{
  while true do
    var x = read io.inp;
    println$ "Debug sink ["+s+"] "+x.str;
  done
}
```

latch

Satisfies all reads on its output channel with the last value read on the input channel. Blocks readers until at least one value is read from its input channel.

```
chip latch[T]
  connector io
    pin inp: %<T
    pin out: %>T
{
   var x = read io.inp;
   device w = fetch &x;
   device r = store &x;
   circuit
     wire io.inp to r.inp
     wire io.out to w.out
   endcircuit
}
```

1.14.17 Duplex Channels

A duplex channel can be used to first send data of type D from one coroutine to another, and then have the second coroutine send data of type C back along the same channel.

This protocol emulates a standard function call where D is the domain of the function and C the codomain. It can be done with two monotyped half-duplex channels as well: using a duplex channel saves one heap allocation and enforces the subroutine call protocol.

1.14.18 Session Typed Channels

Underneath, channels are untyped, and I/O operations transfer a single machine address. Therefore, with casts, you can read and write a pointer to any data type safely provided the read and write agree on the type.

Type systems have been developed, called *session types* which can be used to statically enforce agreement on the type of data being communicated, where the type varies over time, however Felix currently does not support any session types other than duplex channels.

1.15 Pre-emptive Threads

Felix supports contruction of pre-emptive threads.

1.15.1 Spawning

A library procedure *spawn_pthread* accepts a unit procedure argument and spawns it is a detached anonymous pre-emptive thread.

```
spawn_pthread { println$ "hello"; };
```

This means you cannot join threads, and you do not get an identifier or handle for a thread. All Felix threads have the same status, including the initial thread. A process does not end until all threads terminate.

1.15.2 Pchannels

The primary vehicle for synchronisation is the *pchannel*. Technically a pchannel is a monitor. When a thread writes to a pchannel it is blocked until another thread reads its data. Similarly if a thread reads from a channel it is blocked until another thread writes it data.

```
var inp, out = mk_iopchannel_pair[int]();
spawn_pthread {
  println$ "R1";
  var i = read inp;
  println$ "R2-" + i.str;
};
spawn_pthread {
  println$ "W1";
  write (out, 42);
  println$ "W2";
};
println$ "Spawning done";
```

In this program R1 and W1 print first in an indeterminate order. Then R2 and W2 print, in an indeterminate order. The spawning done message can print at any time. The I/O access to the pchannel therefore acts a barrier.

Any number of threads can attempt to read or write on a pchannel. If a write is already writing, another writer will block until the operation is completed by a reader reading, similarly only one reader can wait for data at a time, another reader will block until the reader has acquired its data.

Pchannels can be used to join threads.

1.15.3 Mutual Exclusion

Felix provides two kinds of mutex locks. The primary lock is a spin loop with a delay. We use a loop rather than a system mutex because the loop also checks to see if another thread has requested a garbage collection.

Felix also provides a raw OS mutex. It is not safe unless correctly used. The critical region protected by a raw lock *must not perform a Felix heap allocation* because that can trigger a garbage collection. The problem is the collector is a world stop collector which must wait until all threads suspend, and a thread trapped waiting for an OS mutex cannot check if a garbage collection is requested, resulting in a deadlock.

1.15.4 Condition Variables

GC aware condition variable with builtin mutex.

1.15.5 Atomic Operations

Uses C++11 atomics.

1.15.6 Thread Pool

1.15.7 Pfor

1.15.8 Concurrently Procedure

The *concurrently_by_iterator* procedure implements a fork/join protocol. It accepts an iterator which yields unit procedures, calls the iterator and spawning a Felix pthread for each procedure yielded. When the iterator is exhausted, it waits for all the spawned pthread to complete before continuing.

A variant, *concurrently* accepts any data structure with an iterator method:

```
concurrently$
  { println$ "T1"; },
  { println$ "T2"; },
  { println$ "T3"; }
;
```

The concurrently procedure does not use the thread pool.

1.16 Representations

Representation of Felix entities in C++.

1.16.1 Function Representation

Functional code uses the machine stack for function return addresses.

A function type object is an abstract class with a pure virtual method called apply which returns a representation of the codomain and accepts a representation of the domain.

A function is derived from its type and implements the apply method.

Function closures in Felix are pointers to function type objects, therefore all functions of the same type are represented by a pointer to the same C++ class. The actual function is called by virtual dispatch.

The function class constructor is used to store a pointer to the thread frame object and the display, which is the list of the most recent activation records of the ancestors of the function at the time the closure was created. The function can use the display to access the ancestor local variables.

The objects pointer to by the display members can be either function or procedure frames. Here is an example.

Felix code:

```
noinline fun k(z:int) = {
  fun f(x:int) = {
    var y = x;
    return  y + z;
  }
  return f;
}
```

Function types

```
//TYPE 52224: int -> int
struct _ft52224 {
  typedef int rettype;
  typedef int argtype;
  virtual int apply(int const &)=0;
  virtual _ft52224 *clone()=0;
  virtual ~_ft52224(){};
};
```

Function class:

```
struct thread_frame_t;

//FUNCTION <50810>: k int -> (int -> int)
//    parent = None
struct k {
  thread_frame_t *ptf;

  int z;
  k(thread_frame_t *);
  k* clone();
  _ft52224* apply(int const &);
};

//FUNCTION <50812>: k::f int -> int
//    parent = k<50810>
struct f: _ft52224 {
  thread_frame_t *ptf;
  k *ptrk;

  int x;
  int y;
  f  (thread_frame_t *, k*);
  f* clone();
  int apply(int const &);
};
```

Function apply methods:

```
//FUNCTION <50812>: k::f: Apply method
int f::apply(int const &_arg ){
  x = _arg;
  y  = x; //init
  return y + ptrk->z ;
}

//FUNCTION <50810>: k: Apply method
_ft52224* k::apply(int const &_arg ){
  z = _arg;
  return (new(ptf->gcp, f_ptr_map) f(ptf, this));
}
```

Clone methods:

```
//FUNCTION <51331>: k: Clone method
  k* k::clone(){
  return new(*PTF gcp,k_ptr_map,true) k(*this);
}

//FUNCTION <51333>: k::f: Clone method
  f* f::clone(){
  return new(*PTF gcp,f_ptr_map,true) f(*this);
}
```

Constructors:

```
//FUNCTION <51331>: k: Constructor
k::k(thread_frame_t *ptf_) ptf(ptf_) {}


//FUNCTION <51333>: k::f: Constructor
f::f
  (
    thread_frame_t *ptf_
    k *pptrk
  )
  ptf(ptf_), ptrk(pptrk) {}
```

The symbol *gcp* is a pointer to the garbage collector profile object. The symbol *f_ptr_map* is a pointer to the static run time type information for *f* which is associated with the store allocated for the closure of f created to the collector can trace it. This is necessary because the closure of *f* contains a pointer to a closure of *k*, as well as the thread frame object.

The type of *k* is elided because Felix knows the function not formed into a closure, this is an optimisation.

The clone method (not show) invokes the copy constructor, it is used when calling the function to ensure the initial state is invariant. This may be necessary if the function is called twice through the closure, particularly if it is recursive.

1.16.2 Non-Yielding Generators

A non-yielding generator has the same representation as a function, except that the clone method returns *this* instead of a pointer to a heap allocated copy of the class object.

Whilst function values stored in variables are cloned to ensure they have an invariant initial state, generators aren't, to ensure internal state is preserved between calls.

1.16.3 Yielding Generators

A yielding generator is a generator with a *yield* statement. Yield returns a value and saves the current program counter.

The *apply* function body is called then the function jumps to the saved program counter. Note that the parameter is set to the argument of each invocation.

```
gen f (var x:int) = {
  var i = 10;
  while i > 0 do
    yield x;
    --x; --i;
  done
  return 0;
}

var k = f;

var v = k 4;
while v > 0 do
  println$ v;
  v = k 2;
done
```

This program prints 4 once then 1, nine times.

The usual way to write generators is to use a higher order function:

```
gen f (var x:int) () = {
  while x > 0 do
    yield x;
    --x;
  done
  return 0;
}

var k = f 4;

var v = #k;
while v > 0 do
  println$ v;
  v = #k;
done
```

The header looks like this:

```
//FUNCTION <51331>: f int -> (unit -> int)
//    parent = None
struct f {
  thread_frame_t *ptf;

  int x;
  f(thread_frame_t *);
  f* clone();
  _ft52601* apply(int const &);
};

//FUNCTION <51333>: f::f'2 unit -> int
```

(continues on next page)

(continued from previous page)

```
//    parent = f<51331>
struct _fI51333_f__apos_2: _ft52601 {
  thread_frame_t *ptf;
  int pc;
  f *ptrf;

  _fI51333_f__apos_2  (FLX_FPAR_DECL f*);
  _fI51333_f__apos_2* clone();
  int apply();
};
```

The apply method looks like this:

```
//FUNCTION <51331>: f: Apply method
_ft52601* f::apply(int const &_arg ){
  x = _arg;
  return
    new(*ptf->gcp,_fI51333_f__apos_2_ptr)
      _fI51333_f__apos_2 (ptf, this)
  ;
}

//FUNCTION <51333>: f::f'2: Apply method
int _fI51333_f__apos_2::apply(){
  switch (pc) {
    case 0:
    continue__ll_x_102_L51335:;
      if(!(0 < ptrf->x))
        goto break__ll_x_102_L51336;
      pc = 52614
      return ptrf->x;//yield
    case 52614:
      {
      int* _tmp52615 = (int*)&ptrf->x;
      --*((_tmp52615));
      }
      goto continue__ll_x_102_L51335;
    break__ll_x_102_L51336:;
      return 0;
  }
}
```

With gcc compiler, a computed goto is used instead of a switch.

1.16.4 Compact Linear Type Representation

A complete value of a compact linear type is stored in a 64 bit machine word.

```
typedef ::std::uint64_t cl_t;

// ****************************************************
/// COMPACT LINEAR PROJECTIONS
// ****************************************************

struct RTL_EXTERN clprj_t
```

(continues on next page)

(continued from previous page)

```
{
  cl_t divisor;
  cl_t modulus;
  clprj_t () : divisor(1), modulus(-1) {}
  clprj_t (cl_t d, cl_t m) : divisor (d), modulus (m) {}

};

// reverse compose projections left \odot right
inline clprj_t rcompose (clprj_t left, clprj_t right) {
  return clprj_t (left.divisor * right.divisor, right.modulus);
}

// apply projection to value
inline cl_t apply (clprj_t prj, cl_t v) {
  return v / prj.divisor % prj.modulus;
}

// ********************************************************
/// COMPACT LINEAR POINTERS
// ********************************************************

struct RTL_EXTERN clptr_t
{
  cl_t *p;
  cl_t divisor;
  cl_t modulus;
  clptr_t () : p(0), divisor(1),modulus(-1) {}
  clptr_t (cl_t *_p, cl_t d, cl_t m) : p(_p), divisor(d),modulus(m) {}

  // upgrade from ordinary pointer
  clptr_t (cl_t *_p, cl_t siz) : p (_p), divisor(1), modulus(siz) {}
};

// apply projection to pointer
inline clptr_t applyprj (clptr_t cp, clprj_t d)  {
  return  clptr_t (cp.p, d.divisor * cp.divisor, d.modulus);
}

// dereference
inline cl_t deref(clptr_t q) { return *q.p / q.divisor % q.modulus; }

// storeat
inline void storeat (clptr_t q, cl_t v) {
    *q.p = *q.p - (*q.p / q.divisor % q.modulus) * q.divisor + v * q.divisor;
    //*q.p -= ((*q.p / q.divisor % q.modulus) - v) * q.divisor; //???
}
```

1.16.5 General Variant Representation

```
// ********************************************************
/// VARIANTS. Felix union type
/// note: non-polymorphic, so ctor can be inline
// ********************************************************
```

(continues on next page)

(continued from previous page)

```
struct RTL_EXTERN _uctor_
{
  int variant;  ///< Variant code
  void *data;   ///< Heap variant constructor data
  _uctor_() : variant(-1), data(0) {}
  _uctor_(int i, void *d) : variant(i), data(d) {}
  _uctor_(int *a, _uctor_ x) : variant(a[x.variant]), data(x.data) {}
};
```

1.16.6 Jump Address Representation

```
struct RTL_EXTERN jump_address_t
{
  con_t *target_frame;
  FLX_LOCAL_LABEL_VARIABLE_TYPE local_pc;

  jump_address_t (con_t *tf, FLX_LOCAL_LABEL_VARIABLE_TYPE lpc) :
    target_frame (tf), local_pc (lpc)
  {}
  jump_address_t () : target_frame (0), local_pc(0) {}
  jump_address_t (con_t *tf) : target_frame(tf), local_pc(0) {}
  // default copy constructor and assignment
};
```

1.16.7 Continuation Base

```
struct FLX_EXCEPTIONS_EXTERN con_t ///< abstract base for mutable continuations
{
  FLX_PC_DECL               ///< interior program counter
  struct _uctor_ *p_svc;          ///< pointer to service request

  con_t();                  ///< initialise pc, p_svc to 0
  virtual con_t *resume()=0;///< method to perform a computational step
  virtual ~con_t();
  con_t * _caller;          ///< callers continuation (return address)
};
```

Abstract Representation of Procedural Continuations

Service Address

Address of a service request, usually NULL.

Callers Continuation

Pointer to the calling procedure's continuation, or NULL if there isn't one.

Program Counter

A location in the code set when the continuation is suspended to allow resumption from the suspension point.

Display

An array of pointers to continuations consisting of the activation records of the parent, grandparent, great grandparent, etc, through to the outermost procedure at the time this continuation is created.

Thread Frame

A pointer to the thread frame, which is a global record shared by all threads of the current process. It contains at least a pointer to the system garbage collector, the program arguments, and pointers to the standard input, output and error streams and possibly some other technical data. The rest of the frame contains the global level variables.

Local Variables

The local variables of the procedure.

Notes

The return address of a procedure consists of a pointer to the calling continuation and the program counter stored in *that* continuation (not in the current one).

Optimisation

Function and procedure objects are generally allocated on the heap. However if it is safe, Felix can allocate them on the machines stack.

Furthermore, it may also replace them with actual C++ functions.

Finally, it can also inline functions so they may not exist at all as discrete objects. Within certain bounds direct calls and applications are inlined.

1.16.8 Slist Representation

```
// ********************************************************
/// SLIST. singly linked lists: SHARABLE and COPYABLE
/// SLIST manages pointers to memory managed by the collector
// ********************************************************

struct RTL_EXTERN slist_node_t {
  slist_node_t *next;
  void *data;
  slist_node_t(slist_node_t *n, void *d) : next(n), data(d) {}
};


struct RTL_EXTERN slist_t {
```

(continues on next page)

(continued from previous page)

```
  slist_t(){} // hack
  gc::generic::gc_profile_t *gcp;
  struct slist_node_t *head;

  slist_t (gc::generic::gc_profile_t*); ///< create empty list

  void push(void *data);                ///< push a gc pointer
  void *pop();                          ///< pop a gc pointer
  bool isempty()const;
};
```

1.16.9 Synchronous Channel Representation

```
// ******************************************************
/// SCHANNEL. Synchronous channels
// ******************************************************

struct RTL_EXTERN schannel_t
{
  slist_t *waiting_to_read;             ///< fthreads waiting for a writer
  slist_t *waiting_to_write;            ///< fthreads waiting for a reader
  schannel_t(gc::generic::gc_profile_t*);
  void push_reader(fthread_t *);        ///< add a reader
  fthread_t *pop_reader();              ///< pop a reader, NULL if none
  void push_writer(fthread_t *);        ///< add a writer
  fthread_t *pop_writer();              ///< pop a writer, NULL if none
private: // uncopyable
  schannel_t(schannel_t const&) = delete;
  void operator= (schannel_t const&) = delete;
};
```

1.16.10 Fibre Representation

```
struct RTL_EXTERN fthread_t // fthread abstraction
{
  con_t *cc;                    ///< current continuation

  fthread_t();                  ///< dead thread, suitable for assignment
  fthread_t(con_t*);            ///< make thread from a continuation
  _uctor_ *run();               ///< run until dead or driver service request
  void kill();                  ///< kill by detaching the continuation
  _uctor_ *get_svc()const;      ///< get current service request of waiting thread
private: // uncopyable
  fthread_t(fthread_t const&) = delete;
  void operator=(fthread_t const&) = delete;
};
```

CHAPTER 2

Standard Library

Please see https://felix-library-packages.readthedocs.io/en/latest/index.html.

CHAPTER 3

Indices and tables

- genindex

Index