# The FARM
# Felix Annotated Reference Manual

John Skaller

May 9, 2016

# Contents

# Chapter 1

# Introduction

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

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

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

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

# Part I

# Gross Structure

# Chapter 2

# Program structure

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

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

## 2.1 Parse Unit

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

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

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

## 2.2 AST

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

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

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

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

## 2.3 Include Directive

### 2.3.1 Syntax

An include directive has the syntax:

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

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

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

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

### 2.3.2 Effect

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

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

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

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

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

## 2.4 Grammar

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

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

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

The parser is a scannerless GLR parser with significant extensions.

# Chapter 3

# Parsing

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

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

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

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

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

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

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

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

## 3.1 Bootstrap Parser Grammar

Compiler Reference

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

### 3.1.1 Parser Processing

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

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

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

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

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

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

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

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

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

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

### 3.1.2 Spawning Choices

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

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

### 3.1.3 No keywords

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

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

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

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

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

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

### 3.1.4 Predefined Variables

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

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

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

### 3.1.5 `SCHEME` statement

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

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

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

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

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

the possible parses of that statement may occur, leading to multiple observations of the side-effects.

### 3.1.6 `STMT` statement

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

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

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

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

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

### 3.1.7 `syntax` statement

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

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

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

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

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

### 3.1.8 `requires` statement

Within a dssl definition, the requires statement consists of the word `requires` followed by a comma separated list of identifiers, and terminated by a semicolon.

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

### 3.1.9 Priority statement

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

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

### 3.1.10 Non-terminal definition

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

by a semi-colon `;`.

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

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

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

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

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

### 3.1.11 Preprocessing productions

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

- Expressions contained in parentheses are replaced by a generated non-terminal name and the non-terminal is defined by the enclosed sequence, its action is to return a list of the ASTs of each symbol.
- A non-terminal postfixed by `*` or `+` is replaced by a generated non-terminal defined to recursively generate a list of the non-terminal, with either 0 or more or 1 or more entries in the list, respectively.
- A non-terminal postfixed by `?` returns the a list containing just the s-expression returned by the non-terminal if it is matched, otherwise an empty list.
- Macro calls are expanded. This done by replacing each occurence of a macro parameter with the correponding argument of the call.

### 3.1.12 Grammar Macros

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

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

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

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

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

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

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

Open recursion allows a recursive structure to be extended so that the extensions *backport* to the core, in other words, the extensions are covariant. This solves a significant problem in grammar extension. Normally in a grammar extension is easily done by simply adding extra productions for a non-terminal.

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

The problem with clashing is only a symptom of the deeper problem: the mech-

anism does not obey Meyer's Open/Closed principle.

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

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

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

### 3.1.13 `regdef` statement

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

- An atom, being
  - an alias for a regular expression,
  - a string, representing that string,
  - a string in square brackets `[ ]` representing one of the characters in the string.
  - an underscore `_` representing any character
  - a dot `.` representing any character except newline
  - a decimal integer representing the character with that ordinal
  - a range consisting of pair of decimal integers separated by a dash and enclosed in square brackets `[` and `]` representing one of the characters in the range, inclusive
  - a regular expression in parentheses `( )`
- a repetition consisting of an atom optionally followed by
  - a `*` for zero or more occurences
  - a `+` for one or more occurences
  - a `?` for zero or one
- a sequence of repetitions
- A list of alternatives separated by a vertical bar `|` each being a sequence

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

### 3.1.14 `literal` statement

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

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

For example, from the library:

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

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

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

### 3.1.15 `open syntax` statement

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

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

## 3.2 Felix Grammar syntax

Library Package Grammar directory

# Chapter 4

# Modules

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

## 4.1 Special procedure `flx_main`

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

A simple example:

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

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

println$ "Init done";
```

produces output:

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

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

## 4.2 Libraries

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

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

# Chapter 5

# Lexicology

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

Felix uses a scannerless parser, there are no keywords.

## 5.1 Comments

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

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

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

### 5.1.1 C++ comments

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

### 5.1.2 Nested C comments

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

## 5.2 Layout

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

## 5.3 File inclusion

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

## 5.4 fdoc files

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

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

### 5.4.1 Uncomments

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

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

### 5.4.2 #line directive

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

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

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

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

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

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

## 5.5 #! directive

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

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

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

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

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

## 5.6 Identifiers

Library Reference

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

### 5.6.1 Plain Identifiers

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

```
Ab_cd1  a’ b-x
```

Identifies starting with underscore are reserved for the implementation.

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

### 5.6.2 TeX Identifiers

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

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

## 5.7 Operator Identifiers

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

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

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

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

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

### 5.7.1 Special identifiers

The special string literal with a "n" or "N" prefix is a way to encode an arbitrary sequence of characters as an 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];
```

## 5.8 Boolean Literals

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

## 5.9 Integer Literals

Library Reference

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

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

An underscore is permitted after the prefix.

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

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

Note there are no negative integer literals.

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

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

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

There is a table of the types Table 20.1 Felix Integer Types.

Note the suffices do not entirely agree with C.

## 5.10 Floating point literals

Library Reference

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

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

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

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

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

There is a table of the operators Table 20.5 Floating Point Operators.

## 5.11 String like literals

Library Reference

### 5.11.1 Standard string literals

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

The single quote forms must be on a single line.

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

The complete list of special escapes is shown in table Table 5.1 String Escapes.

Table 5.1: String Escapes

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

### 5.11.2 Raw strings

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

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

### 5.11.3 Null terminated strings

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

### 5.11.4 Perl interpolation strings

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

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

### 5.11.5 C format strings

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

Such strings are actually functions.

The string contains code such as "C format specifiers.

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

Variable field width specifiers "*" are not permitted.

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

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

The argument types and code types are fully checked for type safety. There are some tables of accepted codes: Table 5.2 C format codes: integer, ?? ??, Table 5.4 C format codes: floating, Table 5.5 C format codes: other.

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

Table 5.2: C format codes: integer

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

Table 5.3: C format codes: special integer

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

Table 5.4: C format codes: floating

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

Table 5.5: C format codes: other

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

# Chapter 6

# Macro processing

Library Syntax Reference

Compiler Semantics Reference

## 6.1 Macro val

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

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

## 6.2 Macro for

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

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

## 6.3 Constant folding and conditional compilation

Compiler Semantics Reference

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

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

Strings are concatenated.

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

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

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

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

# Part II

# Lookup

# Chapter 7

# Names

## 7.1 Kinds of Names

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

### 7.1.1 Simple Names

A *simple name* is just an identifier.

```
x
joe
```

### 7.1.2 Simple Indexed Names

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

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

### 7.1.3 Unqualified Names

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

### 7.1.4 Qualified Names

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

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

### 7.1.5 Special name `root`

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

### 7.1.6 Unsuffixed Name

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

### 7.1.7 Suffixed Name

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

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

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

## 7.2 Symbol Tables

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

### 7.2.1 Symbol definition table

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

### 7.2.2 Name lookup table

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

Each scope is associated with two name lookup tables.

#### Private name lookup table

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

#### Public name lookup table

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

### 7.2.3 Views

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

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

For example:

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

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

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

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

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

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

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

This is effected as follows by overload resolution.

### 7.2.4 Name lookup environments

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

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

#### Primary name lookup table

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

#### Shadow name lookup table

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

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

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

## 7.3 Overload Resolution

Overload resolution proceeeds as follows. We start with the result of a function lookup which returns a set of views called candidates. Our aim is to select a

unique function, together with assignments to the type variables of the definition, which cause the signature of the call to equal the signature of the function, and any constraints specified for the function to be satisfied.

- During overload resolution, a subset of the candidates views is selected which match the supplied name and signatures.
- Constraints are now applied to eliminate candidates additional candidates. See section 15.1 Type Constraints for more details.
- The subset is then reduced by eliminating views which are strictly less specialised than another.
- If the result of this reduction consists of a set of more than one view of the same function, then they must map to the same specialisation of the function, and overload resolution has succeeded.

The algorithm is non trivial. Here is another example:

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

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

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

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

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

Our conclusion is that the programmer is free to open overlapping views of the same function, and in particular duplicated opens do not cause a problem (the same consideration applies to non-functions). *This is* not *the case when specifying type class instances, where overlaps are associated with distinct definitions,*

*so that a single most specialised definition is required. This issue, however, does not occur in ordinary (phase 1) lookup of names. It occurs only during instantiation by monomorphisation when a type class virtual must select from available instances. In this case, there is no overloading to consider since the particular function in each instance is already selected: we only need consider the specialisation lattice.*

## 7.4 Lookup

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

Note that unlike C++, Felix has no automatic type conversions. Matching of signatures in Felix must be exact. (However see subsection 12.3.5 Row Polymorphism).

### 7.4.1 Unqualified name lookup

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

### 7.4.2 Basic unqualified non-function lookup

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

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

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

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

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

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

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

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

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

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

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

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

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

### 7.4.3 Basic unqualified function lookup

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

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

### 7.4.4 Lookup with signatures

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

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

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

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

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

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

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

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

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

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

### 7.4.5 Unqualified function lookup with signatures

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

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

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

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

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

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

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

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

### 7.4.6 Qualified non-funtion name lookup

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

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

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

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

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

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

### 7.4.7 Qualified function name lookup

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

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

### 7.4.8 Qualified lookup for function with signatures

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

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

# Chapter 8

# Classes

Syntax

## 8.1 Class Statement

The top level Felix module can contain submodules which are specified by a non-polymorphic class statement:

```
class classname { ... }
```

### 8.1.1 Qualified Names

The effect is to produce a qualified name to be used outside the class:

```
class classname { proc f () {} }
classname::f ();
```

### 8.1.2 Nested Classes

Classes may be nested.

```
class A {
  class B {
    proc f () {}
    f();
  }
  B::f();
}
A::B::f ();
```

### 8.1.3 Private definitions

A class may contain private definitions, if a symbol is marked private it is only visible in the scope class in which it is defined, including any nested classes:

```
class A {
  private proc f () {}
  class B {
    f(); // visible inside A
  }
  f(); // visible inside A
}
// A::f () fails, f is not visible outside A
```

### 8.1.4 Single File Rule

A class must be specified within a single file.

### 8.1.5 Completeness Rule

Classes are not extensible, a definition of a class with the same name in the same scope is not permitted.

### 8.1.6 Setwise Lookup

The body of a class forms a nested scope. Within a class all symbols defined in the class are visible, along with all those visible in the enclosing context.

```
var x = "Hello";
class A {
   proc f() { g(); } // g visible before defined
   proc g() { x = x + 1; }
   var y = f(); // calls g which uses x uninitialised!
   var x = 1;
}
```

Beware using uninitialised variables! The procedure `g` above uses `A::x` before it is initialised.

### 8.1.7 Top level module alias

The reserved name `root` may be used as a prefix for the top level module:

```
var x = 1;
class A { var x = root::x; }
```

## 8.2 Polymorphic Classes

### 8.2.1 Parametric Polymorphism

A class may be specified with one or more type variables. Such a class is said to be a polymorphic class.

```
class MakePair[T,U] {
  fun fwd_pair (x:T) (y:U) => x,y;
  fun rev_pair (x:T) (y:U) => x,y;
}
```

### 8.2.2 No Variables

Polymorphic classes may not directly contain variables, all members must be function or type definitions.

### 8.2.3 Indexed Names

Entities from a class can be specified from outside using a qualified name in which the class name is followed a square bracketted list of types instantiating the corresponding type variables, this is an *indexed name*:

```
var f = MakePair[int,string]::fwd_pair 42 "Hello";
var r = MakePair[int,string]::rev_pair 42 "Hello";
```

### 8.2.4 Deduced indices

However one or more or even all the types can be omitted from the right end of the list if they can be deduced by overload resolution.

### 8.2.5 Elision of indices

The square brackets are optional if they enclose an empty list.

```
f = MakePair[int,string]::fwd_pair 42 "Hello";
f = MakePair[int]::fwd_pair 42 "Hello";
f = MakePair[]::fwd_pair 42 "Hello";
f = MakePair::fwd_pair 42 "Hello";
```

Note that all symbols in Felix are considered to be indexed. So for example this is correct:

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

### 8.2.6 Sloppy indexing

It is also permitted to move the instance types across qualification boundaries:

```
f = MakePair[int]::fwd_pair[string] 42 "Hello";
```

although this practice is not recommended.

From the outside, class type variables are viewed as univeral quantification. From inside however, the variable names are fixed types which happen to be unknown, that is they're to be considered as existentials.

This means they cannot be specified explicitly or implicitly inside the class with unqualified access:

```
class A[T] {
  fun f(x:T) => x,x;
   fun g(y:T) => f y; // correct
   fun h(y:T) => A[T]::f y; // correct
   // fun h(y:T) => f[T] y; // wrong
}
```

From inside the class the function `f` above is monomorphic, not polymorphic. Although `T` is not known, it is a fixed, invariant type.

### 8.2.7 Polymorphic Members

Functional members of a class may be polymorphic. This is independent of the class type parameters.

```
class A[T] {
  fun f[U] (x:T, y:U) => x,y;
  fun g(a:T) => f[T * T] (a,a);
```

In this case the function `f` has its own personal type variable `U` which is set in the call in `g` to `T * T` because its argument is a pair of `a` which has that type.

## 8.3 Virtuals and instances

A polymorphic class may contain functions or procedures marked `virtual`. In this case a definition is optional.

```
class A[T] {
  // non virtual function
  fun f(x:T) => x,x;

  // virtual function with definition
  virtual fun g (x:T) => f x;

  // virtual function without definition
  virtual fun h : T -> T * T;
}
```

If a virtual function without a definition from a class is used, it must be defined for the types for which it is used. This is done in an instance statement:

```
var a = 1; // int
var x = A[int]::h a; // h used for T=int

// instance defining h for int
instance A[int] {
  fun h(a:int) => a + 1, a - 1;
}
```

An instance can be defined anywhere in a non-functional scope provided the class being instantiated is visible.

In this case symbols in the instance definition are bound the context of the instance scope and its enclosing scope in the usual way.

It is not possible to directly access the definitions in an instance from outside the instance. Instead all access is diverted through the top level class being instantiated. Symbols are bound to the top level class member first to fix the type variables, then a matching instance is sought. This second lookup is called *phase 2 lookup*, it occurs during monormophisation.

If there is no matching instance, the program fails with a diagnostic.

If there is more than than one matching instance, the most specialised instance is used. If there is no most specialised instance, the use of the class member is ambiguous and the program fails with a diagnostic.

An instance must be not less specialised than the class it instantiates (that is, equally specialised or more specialised).

Instance specialisation is judged by the polymorphic subtyping rule which is implemented using a variant of the unification algorithm in which dependent and independent types are distinguished.

A polymorphic type S[T] parametrised by a type variable T is more specialised than a polymorphic type A[U] if U may be replaced by a type expression containing T the result of which equals S[T].

If S has more than one type variable, substitutions for each of them must be found. If A has more than one type variable, the substitutions involve all of these variables.

A specialisation S1 of A is strictly more specialised than a specialisation S2 of A if S1 is also strictly a specialisation of S2.

### 8.3.1 Default Definition

If a virtual function has a definition, it is called a *default definition*. On a use of a function with a default definition, if no matching instance is found, the default definition is specialised and used instead. Note that default definitions do not resolve ambiguity from overlapping instances.

### 8.3.2 Matching polymorphic functions

If a virtual function is polymorphic, the personal type variables of the function must not be specialised in an instance. The class type variables are resolved independently of the function type variables.

You may think of this as if the class type variables are resolved first by finding a uniquely matching instance, and then the function type variables are resolved in a second step (although in fact there is no coupling between the resolutions).

## 8.4 Classes as Categories

A class may thought of in a category theoretic sense.

A base category constists of objects which consists of a finite set of base types and all combinations thereof formed by the available set of type combinators such as tuples, records, function types, etc.

The arrows of this category consist of all the functions explicity defined on these types, and all compositions thereof. This includes the identity functions implicitly defined by copying.

This is the category generated by the graph consisting of the defined functions, where paths in the graph are (reverse) function compositions. This is called the free category generated by the graph.

A class in the context of a base category generates an extension to that category by throwing in some additional types and arrows. The type variables are added to the set of base types, so that the set of combinations is extended. The functions are added to the set of digraph edges of the base generating graph, to add additional arrows.

This view of the role of a class is vital to understand semantic rules which correspond to the notion of a category generated by a graph with relations. The relations form a constraint. In category theory, the relations are generally equations relating the arrows which specify that certain function compositions are equal to others.

The freely generated category is sometimes called an initial algebra. When constraints are added a new category is formed by collapsing some sequences of arrows to a single arrow. There is a mapping called an epi-morphism which takes each arrow from the freely generated category to the target so that two distinct composites may map to the same arrow in the target.

Conversely, there is a reverse mapping from the arrows of the target category to sets of arrows of the freely generated one, forming another category called a quotient category. These two mappings are functions and together form an adjunction.

The purpose of this explanation is not merely for your entertainment. In most programming classes are intended to be constrained, that is, the instances are intended to have a certain behaviour.

Consider for example:

```
class Addable [T] {
  virtual fun add: T * T -> T;
}
```

Now consider the following instances:

```
instance Addable[uint] {
  fun add (x:uint, y:uint) => x + y;
}

instance Addable[int] {
  fun add (x:int, y:int) => x + y;
}

instance Addable[double] {
  fun add (x:double, y:double) => x + y;
}
```

These instances are good according to the specification, however the addition of `int` and `double` is unspecifed because we may have overflow. Furthermore one may get a big surprise adding floating point representations of reals: if you add a big float to a small one the small one may be so small that the result is equal to the big one.

Basic laws of arithmetic are not obeyed by floating representations, in particular addition of floats is not associative!

It is usual to specify requirements in comments and the programmer must carefully check that instances obey the constraints specified for the class in these comments.

Felix, however, can do a bit better.

## 8.5 Axioms, Lemmas and Theorems

Felix allows some semantic constraints on class instances to be specified with explicit axioms. For example:

```
class Addable [T] {
  virtual fun add: T * T -> T;
  axiom associative (x:T, y:T, z:T):
    add (add (x,y), z) == add (x, add (y, z))
  ;
}
```

The axiom documents in a formal language the requirement that the addition operator defined in an instance must be associative. An axiom is similar to an ordinary boolean function which tests whether some condition is met or not. By applying an axiom to particular values we can check for non-conformance of instances.

However, in a categorical sense, an equational axiom plays precisely the role of a relation in the formation of a category with generators and relations. It is a vital part of practical specification of semantics.

We should note that not all constraints can be expressed as axioms in Felix. In particular since Felix only provides first order polymorphism, only equational predicates in Horn form can be used, that is, all the quantifiers must be (implicitly or explicitly) on the left of the formula. Interior quantifiers require second order polymorphism.

Note also that like most programming languages we use constructive mathematics which constrains the role of existentials to calculations. That is, it is not acceptable to assert that something exists because if it did not a contradiction would be derived. Indirect proof is not allowed in constructive mathematics.

Instead, one must prove something exists by actually constructing one! So for example given $a < b$ we know there exists a number $x$ such that $a < x < b$. A proof of this by asserting that if no such $x$ existed, then we would have a contradiction, is not allowed in constructive mathematics. Instead the proof is simply given by $x = (a + b)/2$. And of course .. the assertion is false for many types such as integers!

### 8.5.1 Lemmas and Theorems

Lemmas and theorems play the same role as axioms. The idea, however, is that lemmas and theorems can be derived from the axioms. A lemma is a simple rule which a good automatic theorem prover could derive, and which seems obvious to a human.

A theorem is a more difficult rule which would require a proof assistant with hints to derive a proof.

### 8.5.2 Reductions

A reduction is also an axiom, but it has an additional property. Instead of an equation, reductions use a directed equality:

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

This reduction asserts that if you reverse a list twice you end up back where you started. But it does more: it give the compiler a hint that it is allowed to replace an expression which reverse a list twice with the list, removing the two reversing operations.

This is a particularly interesting example of the general rule that all theorems are nothing more than optimisations, in this case literally allowing the compiler to optimise a program. Note that it is only a hint!

# Chapter 9

# General lookup

By default Felix looks up symbols in nested scopes, starting with all symbols in the current scope and proceeding through its containing scope outwards until the outermost scope is reached.

Symbols are visible in the whole of a scope, both before and after their introduction.

A symbol lookup may properly find either a single non-function symbol, which is final, or a set of function symbols.

If the kind of symbol being sought is a function symbol, overload resolution is performed on the set of function signatures found in a scope. If a best match is found, that is final. If no match is found the search continues in the next outermost scope.

All other cases are in error.

# Chapter 10

# Overload Resolution

Blah.

# Chapter 11

# Lookup control directives

## 11.1 Open directive

The simple `open` directive may be used to make the symbols defined in a class visible in the scope containing the `open` directive.

```
class X { var x = 1; }
open X;
println$ x;
```

Names made visible by an open directive live in a weak scope under the current scope. Names in the weak scope may be hidden by definitions in the current scope without error.

```
class X { var x = 1; }
open X;
var x = 2;
println$ x; // prints 2
```

The open directive is not transitive. The names it makes visible are only visible in the scope in which the open directive is written.

## 11.2 Inherit directive

The inherit directive allows all of the public symbols of a class to be included in another scope as if they were defined in that scope. This means such names inherited into a class can be accessed by qualification with the inheriting class name, and will be visible if that class is opened.

Inheriting is transtitive.

If a name is inherited it will clash with a local definition.

```
class A { var a = 1; }
class B { inherit A; }
println$ B::a;
```

## 11.3 Rename directive

This directive is can be used to inherit a single symbol into a scope, possibly with a new name, and also to add an alias for a name in the current scope.

When applied to a function name all functions with that name are renamed.

```
class A {
  var a = 1;
  proc f() {}
  proc f(x:int) {}
}

class B {
  rename a = A::a;
  rename fun f = A::f;
}
```

The new name injected by a rename may be polymorphic:

```
class A { proc f[T] () {} }
class B { rename g[T] = A::f[T]; }
```

## 11.4 Use directive

This is a short form of the rename directive:

```
class A { var a = 1; }
class B { use A::a; use b = A::a; }
```

It cannot be applied to functions. The first form is equivalent to

```
use a = A::a;
```

Unlike the rename directive the new name cannot be polymorphic and is limited to a simple identifier.

## 11.5 Export directives

The `export` directives make the exported symbol a root of the symbol graph.

The functional export and forces it to be place in the generated code as an `extern "C"` symbol with the given name:

```
export fun f of (int) as "myf";
export cfun f of (int) as "myf";
export proc f of (int) as "myf";
export cproc f of (int) as "myf";
```

Functions are exported by generating a wrapper around the Felix function. If the function is exported as `fun` or `proc` the C function generated requires a pointer to the thread frame as the first argument, if the `cfun` or `cproc` forms are used, the wrapper will not require the thread frame.

In the latter case, the Felix function must not require the thread frame.

A type may also be exported:

```
export type ( mystruct ) as "MyStruct";
```

This causes a C typedef to be emitted making the name `MyStruct` an alias to the Felix type. This is useful because Felix types can have unpredictable mangled names.

The word `export` optionally followed by a string may also be used as a prefix for any Felix function, generator, or procedure definition. If the string is omitted is taken as the symbol name. The effect is the same as if an export statement has been written.

# Part III

# Type System

# Chapter 12

# Type constructors

Syntax

## 12.1 typedef

The typedef statement is used to define an alias for a type. It does not create a new type.

```
typedef Int = int;
```

## 12.2 Tuples

Tuple types are well known: a tuple is just a Cartesian Product with components identified by position, starting at 0. The n-ary type combinator is infix `*` and the n-ary value constructor is infix `,`:

```
val tup : int * string * double = 1, "Hello", 4.2;
```

The 0-ary tuple type is denoted `1` or `unit` with sole value `()`:

```
val u : unit = ();
```

The 1-ary tuple of type `T` component value `v` is identified with the type `T` and has value `v`.

The individual components of a tuple may be accessed by a projection function. Felix uses an integer literal to denote this function.

```
var x = 1,"Hello";
assert 0 x == 1; assert x.0 == 1;
assert 1 x == "Hello"; assert x.1 == "Hello";
```

[There should be a way to name this function without application to a tuple!]

A pointer to a tuple is also in itself a tuple, namely the tuple of pointers to the individual components. This means if a tuple is addressable, so are the components.

```
var x = 1, "Hello";
val px = &x;
val pi = px.0; pi <-42;
val ps = px.1; ps <-"World";
assert x.0 == 42;
assert x.1 == "World";
```

In particular note:

```
var x = 1, "Hello";
&x.0 <- 42;
```

because the precedences make the grouping `(&x).0`.

You cannot take the address of a tuple component because a projection of a value is a value.

Assignment to components of tuples stored in variables is supported but only to one level, for general access you must take a pointer and use the store-at-address operator `<-`.

### 12.2.1 Tuple projections

The projections of a tuple can also be written in an expanded form so that they may stand alone as functions:

```
var first = proj 0 of (int * string);
var a = 1, "Hello";
var one = a . first;
var two = a . proj 1 of (int * string);
```

## 12.3 Records

A record is similar to a tuple except the components are named and considered unordered up to duplication.

### 12.3.1 Plain Record

A plain record is one without duplicate fields. A plain record type is one without duplicate fields or a row variable. A record is constructed using a parenthesis en-

closed list of comma separated field assignments. An empty record is equivalent to an empty tuple.

```
typedef xy = (x:int, y:int);
var r : xy = (x=1,y=2);
```

### 12.3.2 Record projections

A component of a record may be accessed with a function called a record value projection, it is denoted by the name of the field.

```
var r (x=1,y=2);
println$ x r, r.y;
```

Record value projections can also be used as stand-alone functions. For example:

```
var rl = list ((x=1,y=11),(x=2,y=22));
var xs =  map (x of (x:int, y:int)) rl;
println$ xs; // list (1,2)
```

Records also have pointer projections, overloaded with the value projections: if the name of a field is applied to a pointer to a record, a pointer to the named component field is returned. This allows assignment and other mutators to be applied to record components.

```
var r =(x=1,y=2);
var px = &r.x; // means (&r).x
px <- 42;
r&.y <- 23;
println$ r.x, r.y; // (42,23)
```

Record pointer projects can also be used as stand-alone functions:

```
var prjx = x of (xy);
var prjpx = x of (&xy);
```

### 12.3.3 General record

Records may have duplicate fields. In this case, reading from left to right in a record literal, a duplicate field is hidden by a previous field of the same name, in a push down stack like fashion.

```
var r = (x=1,y=2,x="Hello");
println$ r._strr, r.x;
  // ((x=1,x='Hello',y=2), 1)
```

Note that the generic function `_strr` displays the whole of the record including duplicate fields. However projections only find the left-most field.

### 12.3.4 Adding fields

Fields can be added to an existing record to construct a new record:

```
var r = (x=1,y=2);
var r2 = (a="one",b="two",x="newx" | r);
println$ r2._strr,r2,x;
  // ((a='one',b='two',x='newx',x=1,y=2), newx)
```

Again, leftmost fields hide rightmost ones. You can also add two records with infix +:

```
var r = (a=1) + (b=2) + (a="hello");
println$ r._strr, r.a.str; // ((a=1,a='hello',b=2), 1)
```

The leftmost field with a given name dominates. Record addition by + is only applied if a user defined addition is not found for the argument types.

Currently, addition of fields of two records of the same type is not supported: it is likely the user intended to add corresponding field values rather than hide the fields in the right argument with those on the left.

### 12.3.5 Row Polymorphism

Felix provides a special record type called a *polyrecord* which supports row polymorphism with scoped labels in the style of Daan Leijen. The article is here.

This allows a generic function to be written which accepts an argument which is or contains a value of a record type with more fields than required. Unlike subtyping, the extra fields, whilst inaccessible, are not lost and can be returned by the function. For example:

```
val circle = (x=0.0,y=0.0,r=1.0);
val square = (x=0.0,y=0.0,w=1.0,h=1.0);

fun move[T] (dx:double, dy:double) (shape: (x:double, y:double | T)) =>
  (x=shape.x+dx, y=shape.y+dx | (shape without x y))
;

var inc = 1.0,1.0;
println$ (move inc circle)._strr, (move inc square)._strr;
```

#### `without` operator

The `without` operator can be used to return a record with some fields removed. It works on values of record and polyrecord type. Note that because Felix uses scoped fields once a field is removed, the previous value of that field is exposed

if it exists, and can also be removed. This means it is correct and sometimes necessary to list a field more than once when using the `without` operator.

### 12.3.6 Interfaces

An interface is a special notation for a record type all of whose fields are functions or procedures.

```
interface fred {
  f: int -> int;
  g: int -> 0; // procedure
}

// equivalent to
typedef fred = (f: int -> int, g: int -> 0);
```

The primary use is for specifying the type of a Java like object.

## 12.4 Structs

A struct is a a nominally typed record, that is, it must be defined, and each definition specifies a distinct type.

```
struct S { x:int; y:int; };
var s : S = S (1,2);
println$ s.x,s.y;
```

A struct value can be constructued using the structure name as a function and passing a tuple of values corresponding by position to the fields of the struct.

A struct constructor can be used a first class function.

The field names are projection functions and can be applied to a struct value to extract the nominated component, or applied to a pointer to a struct to find a pointer to the nominated component.

The notation (may be changed soon)

```
var prjx = x of (S);
var prjpx = x of (&S);
```

can be used to refer to a projection in isolation, and a pointer projection in isolation, that is, as unapplied first class functions.

A struct may also contain function and procedure definitions:

```
struct A {
  x:int;
  y:int;
  fun get2x => 2 * self.x;
  fun get2y () => 2 * self.y;
  proc diag (d:int) { self.x <- d; self.y <-d; }
};
```

These functions are precisely equivalent to:

```
fun get2x (self:A) => 2 * self.x;
proc diag (self: &A) (d:int) { self.x <-d; self.y <-d; }
```

Note that for a function `self} is a value, for a procedure @{self` is a pointer.

Because of these definitions, we can form object closures over a struct:

```
var a = A(1,2);
var g2y = a. get2y;
var di = a . diag;
```

Note we can't form a closure for `get2x` without an explicit wrapper, i.e. eta-expansion.

## 12.5 Sums

Sum types are the dual of tuples. They represent a sequence of possible cases, potentially with arguments. Case indicies are 0 origin. Sum variables are decode with a match which may also extract an argument value:

```
typedef num = int + long + double;
var x = (case 1 of num) 53L;
println$
  match x with
  | case 0 (i) => "int " + i.str
  | case 1 (l) => "long" + l.str
  | case 2 (d) => "double " + d.str
  endmatch
;
```

### 12.5.1 Unit sum

There is a family of special sum types equivalent to:

```
2 = 1 + 1
3 = 1 + 1 + 1
4 = 1 + 1 + 1
```

Recall type 1, or unit, is the type of the empty tuple. The type 2 is also known as bool, and represents two cases where `false} is an alias for @{case 0 of 2} and @{true` is an alias for `case 1 of 2`.

The type 0 or void, is the type of no values.

These types are called unit sums because they're a sum of a certain number of units.

Note carefully that:

```
x + (y + z), (x + y) + z, x + y + z
```

are three distinct types because operator + is not associative.

## 12.6 union

A union is the dual of a struct. It is a nominally typed version of a sum. Here for example is a list of integers:

```
union intlist {
  iEmpty ;
  iCons of int * intlist;
};
```

This alternative syntax is more commonly used and comes from ML family:

```
union intlist =
  | iEmpty
  | iCons of int * intlist
;
```

The fields of a union type are injections or type constructors. In effect they cast their argument to the type of the union, thus unifying heterogenous types into a single type.

Pattern matches are used to decode unions.

```
var x = iEmpty;
x = iCons (1, x);
x = iCons (2, x); // list of two integers

fun istr (x:intlist) =>
  match x with
  | #iEmpty => "end"
  | iCons (i, tail) => i.str + "," + istr tail
  endmatch
;
```

The first variant represents an empty list. The second variant says that a pair consisting of an int and a list can be considered as a list by applying the type constructor iCons to it.

Unlike product types, a sum may directly contain itself. This is because sum types are represented by pointers.

### 12.6.1 enum

A restricted kind of union, being a nominally typed version of a unit sum.

```
enum colour { red, green, blue }; // same as
enum colour = red, green, blue; // same as
union colour = red | green | blue;
```

The tag value of an enum can be set:

```
enum wsize = w8=8, w16=16, w32=32, w64=64;
```

### 12.6.2 caseno operator

The caseno operator can find the tag value of any sum type, the anonymous sum, union, enum or variant as an integer.

```
assert caseno w16 == 16;
assert caseno (case 1 of 2) == 1;
```

## 12.7 variant

Wariants the sum type which are the dual of records. They used named injections like unions but are structurally typed.

```
typedef vars = union { Int of int ; Float of float; };
```

## 12.8 Array

Felix has various kinds of array. The term is abused and sometimes refers to the abstract concept, and sometimes the statically typed fixed length array described here.

An array is nothing but a tuple all of whose elements have the same type. It is convenient to use an exponential operator with a unit sum index to provide a compact notation:

```
int ^ 3 // array of 3 integers equivalent to
int * int * int
```

Therefore the value:

```
var a3 = 1,2,3;
```

is, in fact, an array. As for tuples an integer literal applied to an array value returns a component, however for arrays, an expression may be used as well:

```
var i = 1;
var y = a3 . i;
var z = a3 . proj i of (int^3);
```

The last form is equivalent to

```
var z = a .
  match i with
  | 0 => proj 0 of (int^3)
  | 1 => proj 1 of (int^3)
  | 2 => proj 2 of (int^3)
  | _ => throw error
  endmatch
;
```

in other words there is a run time array bounds check equivalent to a match failure. Note that of course the actual generated code is optimised!

A run time check can be avoided by using the correct type of index:

```
var i = case 1 of 3;
var z = a . i; // no run time check
```

### 12.8.1 Multi-arrays

Whilst we introduced the exponential notation

```
  B ^ J
```

as a mere shorthand, where J is a unit sum, in fact Felix allows the index to be any compact linear type.

A compact linear type is any combination of sums, products, and exponentials of unit sums. The type

```
3 * 4 * 5
```

for example is compact linear, and therefore Felix allows the array type

```
typedef matrix = double ^ (3 * 4 * 5)
```

Although this looks like the type

```
typedef array3 = double  ^ 3 ^ 4 ^ 5
```

as suggested by the usual index laws, the latter is an array size 5 of arrays size 4 of arrays size 3 which can be used like:

```
var a : array3;
var z = a . case 1 of 5 . case 1 of 4 . case 1 of 3;
```

where you will note that the projections are applied in the reverse order to the indices. On the other hand to use the first form we have instead:

```
var m : matrix;
var z = a . (case 1 of 3, case 1 of 4, case 1 of 5);
```

The exponent here is a value of a compact linear type. It is a single tuple value! It is called a multi-index when applied to an array.

The advantage of this type is that there is an obvious encoding of the values shown in this psuedo code:

```
i * 3 * 4 + j * 3 + k
```

which is nothing more than a positional number notation where the base varies with position. That encoding clearly associates with the compact linear value as integer in the range 0 to 59, or, alternatively, an value of type 60. In other words the type is linear and compact.

Since clearly, given an integer in range 0 through 59 and this type we can decode the integer into a tuple, being the positional representation of the integer in this weird coding scheme, the type is clearly isomorphic to the subrange of integer.

Therefore Felix allows you to coerce an integer to a compact linear type with a run time check, and convert a compact linear type to an integer or a unitsum:

```
var i : int = ((case 1 of 3, case 1 of 4, case 1 of 5) :>> int);
var j : 60 = ((case 1 of 3, case 1 of 4, case 1 of 5) :>> 60);
var clt : 3 * 4 * 5 = (16 :>> 3 * 4 * 5);
```

Because we can do this we can now write a loop over a matrix with a single iterator:

```
for i in 60 do
  println$ m . (i :>> 3 * 4 * 5);
done
```

This is an advanced topic which will require an extensive explanation beyond the scope of this summary. However we will not that this facility provides a very high level feature known as polyadic array programming. In short this means that one may write routines which work on matrices of arbitrary dimension. You can of course do this in C by doing your own index calculations at run time and using casts, however Felix does these calculations for you based on the type so they're always correct.

# Chapter 13

# Meta-typing

Felix provides some facilities for meta-typing.

### 13.0.1 typedef fun

The notation

```
typedef fun diag (T:TYPE):TYPE=> T * T;
var x: diag int = 1,2;
```

defines a type function (or functor). Given a type T, this function returns the type for a pair of T's. The identifier `TYPE` denotes the kind which is a category of all types.

Applications of type functions must be resolved during binding, since the result may influence overloading.

## 13.1 typematch

Felix has a facility to inspect and decode types at compile time.

```
typedef T = int * long;
var x:
  typematch T with
  | A * B => A
  | _ => int
  endmatch
  = 1
;
```

As with type functions, type matches must be resolved during binding. If a type match fails, an error is issued a compilation halted. The wildcard type pattern _ matches any type.

The real power of the type match comes when combined with a type function:

```
typedef fun promote (T:TYPE): TYPE =>
  typematch T with
  | #tiny => int
  | #short => int
  | #int => int
  | #long => long
  | #vlong => vlong
  endmatch
;
```

This functor does integral promotions of signed integer types corresponding to ISO C rules.

## 13.2 type sets

TBD

# Chapter 14

# Abstract types

Felix provides abstract types as demonstrated in this example.

```
class Rat {
  type rat = new (num:int, den:int);
  ctor rat (x:int, y:int) =>
    let d = gcd (x,y) in
    _make_rat (num=x/d, den-y/d)
  ;

  fun + (a:rat, b:rat) =>
    let a = _repr_ a in
    let b = _repr_ b in
    _make_rat (
      a.num * b.den + b.num * a.den,
      a.den * b.den
    )
  ;
}
```

Here, the abstract type `rat` is represented by a record of two integers, `num} and @{den`, but this type is hidden.

Inside the class `Rat} the operator @{_make_rat` casts the implementation value to an abstract value, and the operator `_repr_` casts the abstract value to its implementation.

These casts cannot be used outside the class, thereby hiding the implementation outside the class.

# Chapter 15

# Polymorphism

## 15.1 Type Constraints

Felix provides 4 kinds of type constraints.

### 15.1.1 Type class constraints

Class constraints have no impact on phase 1 overload resolution when selecting a binding.

However, when binding the definition of a symbol, specified class views are introduced in a shadow scope as if the class were opened.

During monomorphisation, any virtuals must be mapped to instance functions and this is done by a process similar to that used for overload resolution.

### 15.1.2 Typeset membership constraints

Typesets were introduced to Felix to simplify bindings to C functions. For example:

```
typedef someints = typesetof (int, long);
typedef someuints = typesetof (uint, ulong);
fun someadd[T in someints] : T * T -> T = "$1+$2";
fun someadd[T in someuints] : T * T -> T = "$1+$2";
println$ someadd (1,2); // OK, first function
println$ someadd (1u,2u); // OK, second function
// println£ someadd (1,2u); // fails
```

In this example, a call to `someadd` will only work if the arguments types are the same and both either `int` or `long`. Overload resolution first treats the functions as if there were no constraints. Then, any candidate which fails the constraint it thrown out of the set.

This method saves writing out all the cases individually, which may be exponential in the number of individual types involved, whilst at the same time preventing invalid cases which an unconstrained binding would allow.

If two candidates exist with the same signatures, and one has a constraint and the other does not (or has the constant constraint true), the constrained candidate is considered more specialised than the unconstrained one.

However in general, constraints cannot be compared to see which is most specialised. In particular if for all values of type variables constraint A implies constraint B, A is more specialised, however there is no operational method for measuring this in general (halting problem!).

### 15.1.3 Equational Constraints

Felix allows a constraint to be For example:

```
fun f[T,U where T * int == int * U] (x:T * U) => x;
println$ f (1,2);
```

This example uses an equational constraint.

Here is another example using a typematch:

```
typedef fun ispair (x:TYPE) : TYPE =>
  typematch x with
  | _ * _ => 1
  | _ => 0
  endmatch
;

fun f[T where ispair T] (x:T)=>x;

println$ f (42,33L); // OK, signature ispair (int * long) true
```

# Part IV

# Definitions

# Chapter 16

# Variable Definitions

Syntax

A definition is a statement which defines a name, but does no cause any observable behavior, or, a class statement, or, a var or val statement. The latter two exceptions define a name but may also have associated behaviour.

## 16.1 The `var` statement

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

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

The first form specifies the type and an initialising expression which must be of the specified type.

The second form specifies a variable of the given type without an explicit initialiser, however the variable will be initialised anyhow with the default contructor for the underlying C++ type, although that constructor may be trivial.

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

If the initialiser has observable behaviour it will be observed if at all, when control passes through the variable statement.

If the variable introduced by the `var` statement is not used, the variable and its initaliser will be elided and any observable behaviour will be lost.

To be used means to have its address taken in a used expression, to occur in a used expression. A used expression is one which initialises a used variable, or,

is an argument to function or generator in a used expression, or an argument to a procedure through which control passes.

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

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

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

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

A variable may have its address taken:

```
var x = 1;
var px = &x;
```

it may be assigned a new value directly or indirectly:

```
x = 2;
px <- 3;
*px = 4;
```

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

### 16.1.1 Multiple variables

Multipls variables can be defined at once:

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

With this syntax, no type annotation may be given.

## 16.2 The `val` statement.

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

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

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

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

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

Since a `val` simply names an expression, it is associated with a value not an object and cannot be addressed or assigned to. However this does NOT mean its value cannot change:

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

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

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

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

### 16.2.1 Multiple values

Multipls values can be defined at once:

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

With this syntax, no type annotation may be given.

# Chapter 17

# Functions

Syntax

## 17.1 Functions

A felix function definition takes one of three basic forms:

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

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

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

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

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

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

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

Functions may not have side effects.

All these function have a type:

```
D -> C
```

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

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

```
f x
x.f
```

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

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

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

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

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

This hash operator binds more tightly than operator dot so

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

## 17.2 Pre- and post-conditions

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

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

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

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

## 17.3 Higher order functions

A function may be written like

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

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

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

They are equivalent to

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

that is, a function which returns another function.

Such a function can be applied like

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

since whitespace application is left associative.

## 17.4 Procedures

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

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

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

A procedure may return with a simple return statement:

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

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

Procedures can also have pre- and post-conditions.

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

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

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

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

## 17.5 Generators

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

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

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

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

will always print an even number because it is equivalent to

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

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

### 17.5.1 Yielding Generators

A generator may contain a `yield` statement:

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

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

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

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

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

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

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

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

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

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

## 17.6 Constructors

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

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

The constructions are equivalent to

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

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

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

## 17.7 Special function `apply`

When Felix finds and application

```
f a
```

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

```
F * A
```

where A is the type of a. For example

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

## 17.8 Objects

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

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

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

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

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

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

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

The `implements` clause is optional.

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

# Part V

# Executable statements

## 17.9 Assignment

Syntax

## 17.10 The `goto` statement and label `prefix`

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

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

The label must be visible from the goto statement.

There are two kinds of gotos. A local goto is a jump to a label in the same scope as the goto statement.

A non-local goto is a jump to any other visible label.

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

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

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

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

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

  error:> println$ error;
}
```

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

### 17.10.1 halt

Stops the program with a diagnostic.

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

### 17.10.2 try/catch/entry

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

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

### 17.10.3 goto-indirect/label_address

The `label-address` operator captures the address of code at a nominated label.

The address has type `LABEL` and can be stored in a variable.

Provided the activation record of the procedure containing the label remains live, a subsequent @goto-indirect) can be used to jump to that location.

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

### 17.10.4 Exchange of control

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

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

A good example is here, which shows an example of coroutines.

## 17.11 match/endmatch

The form:

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

is an extension of the C switch statement. The patterns are composed of these forms:

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

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

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

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

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

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

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

## 17.12 if/goto

The conditional goto is an abbreviation for the more verbose conditional:

```
    if c goto lab; // equivalent to
    if c do goto lab; done
```

### 17.12.1 if/return

The conditional return is an abbreviation for the more verbose conditional:

```
    if c return; // equivalent to
    if c do return; done
```

### 17.12.2 if/call

The conditional call is an abbreviation for the more verbose conditional:

```
    if c call f x; // equivalent to
    if c do call f x; done
```

## 17.13 if/do/elif/else/done

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

The `else} and @{elif` clauses are optional.

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

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

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

One or more statements may be givn in the selected control path.

A simple conditional is an abbreviation for a statement match:

```
if c do stmt1; stmt2; else stmt3; stmt4; done
// is equivalent to
match c with
| true => stmt1; stmt2;
| false => stmt3; stmt4;
endmatch;
```

## 17.14 call

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

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

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

```
p 1;
```

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

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

## 17.15 procedure return

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

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

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

### 17.15.1 return from

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

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

### 17.15.2 jump

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

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

## 17.16 function return

The functional return statement returns a value from a function.

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

Control may not fall through the end of a function.

### 17.16.1 yield

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

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

```
gen counter () = {
  var x = 0;
next_integer:>
  yield x;
  ++x;
  goto next_integer;
}

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

## 17.17 spawn_fthread

Library Reference

The `spawn_fthread` library function invokes the corresponding service call to schedule the initial continuation of a procedure taking a unit argument as an fthread (fibre).

The spawned fthread begins executing immediately. If coutrol returns before yielding by a synchronous channel operation, the action is equivalent to calling the procedure.

Otherwise the spawned fthread is suspended when the first write, or the first unmatched read operation occurs.

### 17.17.1 read/write/broadcast schannel

Library Reference

## 17.18 spawn_pthread

Library Reference

### 17.18.1 read/write pchannel

Library Reference

### 17.18.2 exchange

## 17.19 loops

Library Reference

Felix has some low level and high level loop constructions.

The low level for, while, and repeat loops are equivalent to loops implemented with gotos.

The bodies of do loops do not constitute a scope, therefore any symbol defined in such a body is also visible in the surrounding code.

Low level loops may be labelled with a loop label which is used to allow break, continue, and redo statements to exit from any containing loop.

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

### 17.19.1 redo

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

### 17.19.2 break

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

### 17.19.3 continue

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

### 17.19.4 for/in/upto/downto/do/done

A basic loop with an inclusive range.

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

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

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

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

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

For loops over unsigned types cannot handle the empty case. For loops over signed types cannot span the whole range of the type.

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

### 17.19.5 while/do/done

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

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

### 17.19.6 until loop

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

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

### 17.19.7 for/match/done

TBD

### 17.19.8 loop

TBD

## 17.20 Assertions

Library Reference

## 17.21 assert

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

```
assert x > 0;
```

### 17.21.1 axiom

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

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

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

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

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

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

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

### 17.21.2 lemma

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

### 17.21.3 theorem

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

There is currently no standard way to prove such hints.

### 17.21.4 reduce

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

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

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

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

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

### 17.21.5 invariant

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

The invariant need not hold during execution of a method.

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

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

## 17.22 code

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

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

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

### 17.22.1 noreturn code

Similar to code, however noreturn code never returns.

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

## 17.23 Service call

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

It is equivalent to an OS kernel call.

The available operations include:

```
union svc_req_t =
/*0*/ | svc_yield
/*1*/ | svc_get_fthread         of &fthread    // CHANGED LAYOUT
/*2*/ | svc_read                of address
/*3*/ | svc_general             of &address    // CHANGED LAYOUT
/*4*/ | svc_reserved1
/*5*/ | svc_spawn_pthread       of fthread
/*6*/ | svc_spawn_detached      of fthread
/*7*/ | svc_sread               of _schannel * &gcaddress
/*8*/ | svc_swrite              of _schannel * &gcaddress
/*9*/ | svc_kill                of fthread
/*10*/ | svc_reserved2
/*11*/ | svc_multi_swrite       of _schannel * &gcaddress
/*12*/ | svc_schedule_detached  of fthread
;
```

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

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

## 17.24 with/do/done

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

It is the statement equivalent of the let expression.

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

## 17.25 do/done

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

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

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

```
do something; done
```

## 17.26 begin/end

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

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

# Part VI

# Expressions

Syntax

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

We will often exhibit expressions in the form

```
var x = expr;
```

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

## 17.27 Chain forms

### 17.27.1 Pattern let

The traditional let binding of ML. The syntax is

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

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

### 17.27.2 Function let

A let form which makes a function available in the expression

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

### 17.27.3 Match chain

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

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

println$ x;
```

### 17.27.4 conditional chain

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

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

## 17.28 Alternate conditional chain

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

## 17.29 Dollar application

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

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

## 17.30 Pipe application

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

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

## 17.31 Tuple cons constructor

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

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

## 17.32 N-ary tuple constructor

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

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

## 17.33 Logical implication

An operator for function `implies`.

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

## 17.34 Logical disjunction

A chaining operator for function `lor`.

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

## 17.35 Logical conjunction

A chaining operator for function `land`.

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

## 17.36 Logical negation

A bool operator for function `lnot`.

## 17.37 Comparisons

Non-associative.

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

## 17.38 Name temporary

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

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

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

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

## 17.39 Schannel pipe operators

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

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

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

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

## 17.40 Right Arrows

Right associative arrow operators.

List cons operator.

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

Function types (type language only):

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

## 17.41 Case literals

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

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

The case constructor with integer caseno has two uses.

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

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

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

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

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

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

The tuple projection function names a tuple projection:

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

## 17.42 Bitwise or

Left associative.

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

## 17.43 Bitwise exclusive or

Left associative.

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

## 17.44 Bitwise and

Left associative.

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

## 17.45 Bitwise shifts

Left associative.

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

## 17.46 Addition

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

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

## 17.47 Subtraction

Left associative.

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

## 17.48 Multiplication

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

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

## 17.49 Division operators

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

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

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

## 17.50 Prefix operators

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

## 17.51 Fortran exponentiation

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

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

preserving the usual mathematical syntax.

## 17.52 Felix exponentiation

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

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

## 17.53 Function composition

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

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

## 17.54 Dereference

For function `deref`.

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

For builtin dereference operator:

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

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

### 17.54.1 Operator new

Copies a value onto the heap and returns a pointer.

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

## 17.55 Whitespace application

Operator whitespace is used for applications.

### 17.55.1 General

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

### 17.55.2 Caseno operator

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

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

### 17.55.3 Likelyhood

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

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

## 17.56 Coercion operator

left associative. The right operand is a type.

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

## 17.57 Suffixed name

The most general form of a name:

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

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

## 17.58 Factors

### 17.58.1 Subscript

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

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

### 17.58.2 Subsstring

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

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

### 17.58.3 Copyfrom

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

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

### 17.58.4 Copyto

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

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

### 17.58.5 Reverse application

Left associative.

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

### 17.58.6 Reverse application with deref

Left associative

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

### 17.58.7 Reverse application with addressing

Left associative

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

### 17.58.8 Unit application

Prefix operator applies argument to empty tuple.

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

### 17.58.9 Addressing

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

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

### 17.58.10 C pointer

Used in type language for pointer to type or NULL.

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

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

### 17.58.11 Label address

Used to find the machine address in the code text of a label. Used with computed goto instruction.

```
proc f (a: int) {
  var target: LABEL =
    if a < 0 then label_address neg
    elif a > 0 then label_address pos
    else label_address zer
  ;
  goto-indirect target;
  pos:> println$ "pos"; return;
  neg:> println$ "neg"; return;
  zer:> println$ "zer"; return;
}
```

### 17.58.12 Macro freezer

Used to disable macro expansion of a symbol.

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

### 17.58.13 Pattern variable

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

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

### 17.58.14 Parser argument

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

## 17.59 Qualified name

A name in Felix has the form:

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

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

# Chapter 18

# Atoms

## 18.1 Record expression

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

## 18.2 Alternate record expression

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

## 18.3 Variant type

Denotes a variant type.

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

## 18.4 Wildcard pattern

Used in a pattern match, matches anything.

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

## 18.5 Ellipsis

Used only in C bindings to denote varags.

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

## 18.6 Truth constants

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

## 18.7 callback expression

??

```
  callback [ x ]
```

## 18.8 Lazy expression

Function of unit.

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

## 18.9 Sequencing

Function dependent on final expression.

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

## 18.10 Procedure of unit.

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

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

## 18.11 Grouping

Parentheses are used for grouping.

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

## 18.12 Object extension

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

## 18.13 Conditional expression

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

# Part VII

# Library

# Chapter 19

# C bindings

Felix is specifically designed to provide almost seamless integration with C and C++.

In particular, Felix and C++ can share types and functions, typically without executable glue.

However Felix has a stronger and stricter type system than C++ and a much better syntax, so binding specifications which lift C++ entities into Felix typically require some static glue.

## 19.1 Type bindings

In general, Felix requires all primitive types to be first class, that is, they must be default initialisable, copy constructible, assignable, and destructible. Assignment to a default initialised variable must have the same semantics as copy construction.

It is recommended C++ objects provide move constructors as Felix generated code uses pass by value extensively.

The Felix type system does not support C++ references in general, you should use pointers instead.

However, there is a special lvalue annotation for C++ functions returning lvalues that allows them to appear on the LHS of an assignment. Only primitives can be marked lvalue.

The Felix type system does not support either const or volatile. This has no impact when passing arguments to C++ functions. However it may be necessary to cast a pointer returned from a primitive function in order for the generated code to type check.

## 19.2 Expression bindings

TBD

## 19.3 Function bindings

TBD

## 19.4 Floating insertions

TBD

## 19.5 Package requirements

TBD

# Chapter 20

# Core Primitive Types

## 20.1 Boolean type

The type `bool` which is also called `2` provides the usual boolean logic values `false` and `true`. The name `bool` is actually an alias for type unitsum of two cases. The name `false` is an alternate name of the value `case 0 of 2` and the name `true` is an alternate name for the value `case 1 of 2`.

See ?? ?? for more information.

## 20.2 Integer types

Library Reference

There is a table of the types Table 20.1 Felix Integer Types.

Note that all these types are distinct unlike C and C++. The types designated are not the complete set of available integer like types since not all have literal representations.

Signed integers are expected to be two's complement with one more negative value that positive value. Bitwise and, or, exclusive or, and complement operations do not apply with signed types.

The effect of overflow on signed types is unspecified.

Unsigned types use the standard representation. Bitwise operations may be applied to unsigned types. Basic arithmetic operations on unsigned types are all well defined as the result of the operation mathematically modulo the maximum value of the type plus one.

Table 20.1: Felix Integer Types

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

The maximum value of an unsigned type is one less than two raised to the power of the number of bits in the type. The number of bits is 8, 16, 32, or 64 or 128 for all unsigned types.

There is a table of the operators Table 20.2 Integer Operators.

### 20.2.1 Classification of integers

Library Reference

Integer types can be grouped into sets. In Felix this can be done with a `typeset` construction. The defined typesets are show in Table 20.3 Integer Typesets.

Fast integers correspond to the usual C distinct integer types.

Exact integers are aliases to fast integers of specified sizes, in Felix these are distinct types.

The weird integers are special purposes aliases to fast integers which have special uses, again in Felix these are distinct types.

- The type `ptrdiff` is a signed type intended to represent the difference between two pointers.
- The `size` type is for measuring the sizes of objects and number of elements in an array.
- The `intmax` and `uintmax` types are the largest available signed and unsigned integer types, respectively.
- Finally the `intptr` and `uintptr` types are the same size as a pointer and can be used for low level numerical operations on pointers either by a conversion or reinterpret cast. They're typically used for to pack extra data into the low bits of a pointer to a type with an alignment which ensures the low bits of the pointer are zero.

Typesets can be used to simplify specification of C function bindings by leveraging constrained polymorphism.

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

// or equivalently
fun add[T : ints]: T * T -> T = "$1+$2";
```

A special form which implies a quantifer for every use also exists, by specifying a `!` followed by a typeset name. For example,

```
fun intsum: !ints * !ints -> long = "(long)($1+$2)";

// or equivalently
fun intsum[T:ints, U:ints] : T * U -> long = "(long)($1+$2)";
```

Table 20.2: Integer Operators

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

nassoc: non-associative
lassoc: left associative
func: function
note prefix `-` maps to function `neg`

Table 20.3: Integer Typesets

| | |
|---|---|
| fast_sints | (tiny, short, int, long, vlong) |
| exact_sints | (int8,int16,int32,int64) |
| fast_uints | (utiny, ushort, uint, ulong,uvlong) |
| exact_uints | (uint8,uint16,uint32,uint64) |
| weird_sints | (ptrdiff, ssize, intmax, intptr) |
| weird_uints | (size, uintmax, uintptr) |
| sints | fast_sints ∪ exact_sints ∪ weird_sints |
| uints | fast_uints ∪ exact_uints ∪ weird_uints |
| fast_ints | fast_sints ∪ fast_uints |
| exact_ints | exact_sints ∪ exact_uints |
| ints | sints ∪ uints |

Note `intsum` can add any different kinds of integers, returning a `long`.

Polymorphism with typeset constraints is *only* useful for C bindings because the constraints are not propagated. It simply saves writing out all the finite combinations when genericity in the C or C++ target code means the C code would be the same in each case.

## 20.3 Floating point types

Library Reference

There is a table of the operators Table 20.5 Floating Point Operators, we also have Table 20.4 Trig functions.

Felix also provides `FINFINITY`, `DINFINITY` and `LINFINITY` for positive infinity representation for `float`, `double` and `ldouble` types, respectively.

## 20.4 Complex types

There are three complex types, `fcomplex`, `dcomplex` and `lcomplex` corresponding to cartesian representation using a pair of `float`, `double` and `ldouble` values, respectively.

There is a table of the operators Table 20.6 Complex Operators, we also have Table 20.4 Trig functions.

Table 20.4: Trig functions

| symbol | type | semantics |
|---|---|---|
| | | circular |
| `sin` | `T -> T` | sine |
| `cos` | `T -> T` | cosine |
| `tan` | `T -> T` | tangent |
| `asin` | `T -> T` | arc (inverse) sine |
| `acos` | `T -> T` | arc (inverse) cosine |
| `atan` | `T -> T` | arc (inverse) tangent |
| | | hyperbolic |
| `sinh` | `T -> T` | hyperbolic sine |
| `cosh` | `T -> T` | hyperbolc cosine |
| `tanh` | `T -> T` | hyperbolic tangent |
| `asinh` | `T -> T` | arc (inverse) hyperbolic sine |
| `acosh` | `T -> T` | arc (inverse) hyperbolic cosine |
| `atanh` | `T -> T` | arc (inverse) hyperbolic tangent |
| Note: Inverses return primary branch | | |

Table 20.5: Floating Point Operators

| symbol | kind | type | semantics |
|---|---|---|---|
| `==` | infix-nassoc | `T * T -> bool` | equality |
| `!=` | infix-nassoc | `T * T -> bool` | inequality |
| `<` | infix-nassoc | `T * T -> bool` | less |
| `<=` | infix-nassoc | `T * T -> bool` | less or equal |
| `>` | infix-nassoc | `T * T -> bool` | greater |
| `>=` | infix-nassoc | `T * T -> bool` | greater or equal |
| `+` | infix-nassoc | `T * T -> T` | addition |
| `-` | infix-nassoc | `T * T -> T` | subtraction |
| `*` | infix-nassoc | `T * T -> T` | multiplication |
| `/` | infix-nassoc | `T * T -> T` | quotient |
| `-` | prefix | `T -> T` | negation |
| `abs` | func | `T->T` | absolute value |
| `log10` | func | `T->T` | base 10 logarithm |
| `sqrt` | func | `T->T` | square root |
| `ceil` | func | `T->T` | ceiling |
| `floor` | func | `T->T` | floor |
| `trunc` | func | `T->T` | truncate |
| `embed` | func | `int->T` | embedding |

nassoc: non-associative
lassoc: left associative
func: function
note prefix `-` maps to function `neg`

Table 20.6: Complex Operators

| symbol | kind | type | semantics |
|---|---|---|---|
| `fcomplex` | ctor | `float * float -> fcomplex` | cartesian |
| `fcomplex` | ctor | `float -> fcomplex` | from real |
| `dcomplex` | ctor | `double * double -> dcomplex` | cartesian |
| `dcomplex` | ctor | `double -> dcomplex` | from real |
| `lcomplex` | ctor | `ldouble * ldouble -> lcomplex` | cartesian |
| `lcomplex` | ctor | `ldouble -> lcomplex` | from real |
| `==` | infix-nassoc | `T * T -> bool` | equality |
| `!=` | infix-nassoc | `T * T -> bool` | inequality |
| `+` | infix-lassoc | `T * T -> T` | addition |
| `-` | infix-lassoc | `T * T -> T` | subtraction |
| `*` | infix-lassoc | `T * T -> T` | multiplication |
| `/` | infix-lassoc | `T * T -> T` | quotient |
| `-` | prefix | `T -> T` | negation |
| `real` | func | `T->R` | real part |
| `imag` | func | `T->R` | imaginary part |
| `abs` | func | `T->R` | norm |
| `arg` | func | `T->R` | argument |

nassoc: non-associative
lassoc: left associative
func: function
ctor: constructor
note prefix `-` maps to function `neg`

Table 20.7: Quaternion Operators

| symbol | kind | type | semantics |
|---|---|---|---|
| `==` | infix-nassoc | `T * T -> bool` | equality |
| `!=` | infix-nassoc | `T * T -> bool` | inequality |
| `+` | infix-lassoc | `T * T -> T` | addition |
| `-` | infix-lassoc | `T * T -> T` | subtraction |
| `*` | infix-lassoc | `T * T -> T` | multiplication |
| `/` | infix-lassoc | `T * T -> T` | quotient |
| `-` | prefix | `T -> T` | negation |
| nassoc: non-associative | | | |
| lassoc: left associative | | | |
| func: function | | | |
| note prefix `-` maps to function `neg` | | | |

## 20.5 Quaternion type

There is a table of the operators Table 20.7 Quaternion Operators.

## 20.6 Char Type

Felix has a type `char` corresponding to C's `char`. There are no literals of type `char`, instead there is a function, also called `char` to construct a `char` from a string literal. If the string at least length 1, the first character is returned, if the string is empty the `NUL` char `\0x00` is returned.

An overload of the `char` function also accepts an `int` argument and constructs a `char` with the designated C code value which must be in the range 0 through 255.

The function `ord` extracts the C code value of a `char` as an `int` in the range 0 through 255.

We note that C code values were originally used for the ASCII character set. Strings of `char` may be used to represent ISO-10646 (Unicode) code points via the UTF-8 encoding method. ASCII characters in the range 0 through 127 agree with Unicode interpretation.

### 20.6.1 ASCII classification functions

There is a set of Table 20.9 ASCII classification functions used to test the ASCII class of char.

Table 20.8: ASCII charsets

| symbol | definition |
|---|---|
| `upper` | `ABCDEFGHIJKLMNOPQRSTUVWXYZ` |
| `lower` | `abcdefghijklmnopqrstuvwxyz` |
| `letters` | `upper + lower` |
| `digits` | `0123456789` |
| `alphanum` | `letters + digits` |
| `cidcont` | `alphanum+"_"` |
| `flxidcont` | `alphanum+"_-'"` |
| `camlidcont` | `alphanum+"_'"` |
| `numeric` | `digits + ".eEdD_"` |

The library also provides character sets represents as strings, show in Table 20.8 ASCII charsets

Table 20.9: ASCII classification functions

| symbol | type | semantics |
|---|---|---|
| | ISO C functions | |
| `isupper` | `char -> bool` | A-Z |
| `islower` | `char -> bool` | a-z |
| `isalnum` | `char -> bool` | A-Za-z0-9 |
| `isalpha` | `char -> bool` | A-Za-z |
| `isdigit` | `char -> bool` | 0-9 |
| `isxdigit` | `char -> bool` | 0-9A-Za-z |
| `iscntrl` | `char -> bool` | 0x0-0x1F |
| `isspace` | `char -> bool` | 0x20 |
| `isblank` | `char -> bool` | 0x20,0x09 |
| `isprint` | `char -> bool` | 0x20-0x7e |
| `ispunct` | `char -> bool` | punctuation |
| | Lexing functions | |
| `isidstart` | `char -> bool` | First char identifier |
| `iscamlidcont` | `char -> bool` | Ocaml, subsequent chars |
| `iscidcont` | `char -> bool` | C, subsequent chars |
| `isflxidcont` | `char -> bool` | Felix, subsequent chars |
| `isnumeric` | `char -> bool` | `0-9+-.Ee` |
| `isalphanum` | `char -> bool` | A-Za-z0-9 |
| `isletter` | `char -> bool` | A-Za-z |
| `issq` | `char -> bool` | ’ |
| `isdq` | `char -> bool` | " |
| `isslosh` | `char -> bool` | \ |
| `isnull` | `char -> bool` | 0x0 |
| `iseol` | `char -> bool` | `\n` LF: Unix, CR: Windows |

# Chapter 21

# Algebraic Structure of numeric types

Basic Algebra Package Number Package

The functionality identified in the previous chapter is not provided on a type by type basis, but instead leverages the mathematic structure of the types. This is modelled using Felix type classes.

In this section we display a subset of the definitions from the library. The real code also defines some other classes and also a rich set of alternate names for operators utilising the presence of LaTeX symbols: this allows for pleasant typesetting but usually doesn't provide essential semantic operations.

## 21.1 Equality

We start with class `Eq` which models equality based on the notion it is an equivalent relation:

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

  axiom reflex(x:t): x == x;
  axiom sym(x:t, y:t): (x == y) == (y == x);
  axiom trans(x:t, y:t, z:t): x == y and y == z implies x == z;
}
```

By providing an instance with `==`, inequality `!=` is automatically provided by the defaulty definition. An instance can also provide the inequality, as this is

sometimes more efficient.

Conformance to the axioms in instances is required, these axioms define the required semantics.

## 21.2 Total Ordering

Now we can define a total order:

```
class Tord[t]{
  inherit Eq[t];
  virtual fun < : t * t -> bool;
  fun >(x:t,y:t):bool => y < x;
  fun <= (x:t,y:t):bool => not (y < x);
  fun >= (x:t,y:t):bool => not (x < y);

  fun max(x:t,y:t):t=> if x < y then y else x endif;
  fun min(x:t,y:t):t => if x < y then x else y endif;

  axiom trans(x:t, y:t, z:t): x < y and y < z implies x < z;
  axiom antisym(x:t, y:t): x < y or y < x or x == y;
  axiom reflex(x:t, y:t): x < y and y <= x implies x == y;
  axiom totality(x:t, y:t): x <= y or y <= x;
}
```

To have a total order, we first have to have equality, this is effected by use of the `inherit` statement.

Next we define the less than function as virtual, instances must define this. The other functions are provided automatically.

The semantics are specified by the axioms which define a total ordering: reflexivity, anti-symmetry, transitivity, and finally totallity.

## 21.3 Addition

Now we define addition in two steps. First we define approximate addition

```
class FloatAddgrp[t] {
  inherit Eq[t];
  virtual fun zero: unit -> t;
  virtual fun + : t * t -> t;
  virtual fun neg : t -> t;
  virtual fun prefix_plus : t -> t = "$1";
  virtual fun - (x:t,y:t):t => x + -y;
  virtual proc += (px:&t,y:t) { px <- *px + y; }
  virtual proc -= (px:&t,y:t) { px <- *px - y; }

  reduce id (x:t): x+zero() => x;
  reduce id (x:t): zero()+x => x;
  reduce inv(x:t): x - x => zero();
  reduce inv(x:t): - (-x) => x;
  axiom sym (x:t,y:t): x+y == y+x;

  fun add(x:t,y:t)=> x + y;
  fun plus(x:t)=> +x;
  fun sub(x:t,y:t)=> x - y;
  proc pluseq(px:&t, y:t) {  += (px,y); }
  proc  minuseq(px:&t, y:t) { -= (px,y); }
}
```

Most the functions here are virtual without defaults. For the procedures we provide either virtual defaulted definitions or just the definitions.

The key requirement here is simply that some number added to zero returns the original number, as does negating a number twice. We also require subtracting a number from itself yields zero.

Normally addition has stronger requirements such as associativity, but this class is designed to support floating point numbers, and floating addition is not associative.

However since integer addition is, we provide another class:

```
class Addgrp[t] {
  inherit FloatAddgrp[t];
  axiom assoc (x:t,y:t,z:t): (x + y) + z == x + (y + z);
  reduce inv(x:t,y:t): x + y - y => x;
}
```

for an additive group, which just throws in the associative law. The semantic axiom requires cancellation.

Although technically signed integers in Felix do not obey these semantics due to overflow, the real axiom required cannot be modelled in Felix. The true axiom for computer signed integers is that the axioms are obeyed locally, that is, in a compact set close to zero.

This axiom would say, for example, that there exists some $d$ such that if $|a| < d$ and $|b| < d$ and $|c| < d$ then

$$(a + b) + c = a + (b + c)$$

where the restriction is there to ensure there is no overflow. However constructive mathematics cannot model existentials, and instead must provide an actual formula for calculating $d$. In this case, given the range of the integers, we can actually do that, however in general, even if it can be done, it is all too hard to bother.

Instead we just rely on the programmer understanding that the laws only apply locally and using calculations which they believe will not overflow. In practive many calculations with integers do not overflow, however there are a few very nasty cases.

# Chapter 22

# More core types

## 22.1 Slices

## 22.2 String Type

### 22.2.1 Equality and total ordering

```
fun == (s:string, c:string): bool
fun < (s:string, c:string): bool
fun <= (s:string, c:string): bool
fun > (s:string, c:string): bool
fun >= (s:string, c:string): bool
```

### 22.2.2 Equality of `string` and `char`

```
fun == (s:string, c:char): bool
fun == (c:char, s:string): bool
fun != (s:string, c:char): bool
fun != (c:char, s:string): bool
```

### 22.2.3 Append to `string` object

```
proc  += : &string * string
proc  += : &string * +char
proc  += : &string * char
```

### 22.2.4 Length of string

```
fun len: string -> size
```

### 22.2.5 String concatenation

```
fun + : string * string -> string
fun + : string * carray[char] -> string
fun + : string * char -> string
fun + : char * string -> string
fun + ( x:string,  y: int):string
```

### 22.2.6 Repetition of `string` or `char`

```
fun * : string * int -> string
fun * : char * int -> string
```

### 22.2.7 Application of `string` to `string` or `int` is concatenation

```
fun apply (x:string, y:string):string
fun apply (x:string, y:int):string
```

### 22.2.8 Construct a char from first byte of a `string`

Returns nul char (code 0) if the string is empty.

```
ctor char (x:string)
```

### 22.2.9 Constructors for `string`

```
ctor string (c:char)
ctor string: +char
ctor string: +char  * !ints
fun utf8: int -> string
```

### 22.2.10 Substrings

```
fun subscript: string * !ints -> char
fun copyfrom: string * !ints -> string
fun copyto: string * !ints -> string
fun substring: string * !ints * !ints -> string
fun subscript (x:string, gs:gslice[int]):string
proc store: &string * !ints * char
```

### 22.2.11 Map a string `char` by `char`

```
fun map (f:char->char) (var x:string): string = {
```

### 22.2.12 STL string functions

These come in two flavours: the standard C++ operations which return `stl_npos` on failure, and a more Felix like variant which uses an @option type.

```
const stl_npos: size

fun stl_find: string * string -> size
fun stl_find: string * string * size -> size
fun stl_find: string * +char -> size
fun stl_find: string * +char * size -> size
fun stl_find: string * char -> size
fun stl_find: string * char * size -> size

fun find (s:string, e:string) : opt[size]
fun find (s:string, e:string, i:size) : opt[size]
fun find (s:string, e:+char) : opt[size]
fun find (s:string, e:+char, i:size) : opt[size]
fun find (s:string, e:char) : opt[size]
fun find (s:string, e:char, i:size) : opt[size]

fun stl_rfind: string * string -> size
fun stl_rfind: string * string * size -> size
fun stl_rfind: string * +char-> size
fun stl_rfind: string * +char * size
fun stl_rfind: string * char -> size
fun stl_rfind: string * char * size -> size

fun rfind (s:string, e:string) : opt[size]
fun rfind (s:string, e:string, i:size) : opt[size]
fun rfind (s:string, e:+char) : opt[size]
fun rfind (s:string, e:+char, i:size) : opt[size]
fun rfind (s:string, e:char) : opt[size]
fun rfind (s:string, e:char, i:size) : opt[size]

fun stl_find_first_of: string * string -> size
fun stl_find_first_of: string * string * size -> size
fun stl_find_first_of: string * +char -> size
fun stl_find_first_of: string * +char * size -> size
fun stl_find_first_of: string * char -> size
fun stl_find_first_of: string * char * size -> size

fun find_first_of (s:string, e:string) : opt[size]
fun find_first_of (s:string, e:string, i:size) : opt[size]
fun find_first_of (s:string, e:+char) : opt[size]
fun find_first_of (s:string, e:+char, i:size) : opt[size]
fun find_first_of (s:string, e:char) : opt[size]
fun find_first_of (s:string, e:char, i:size) : opt[size]

fun stl_find_first_not_of: string * string -> size
fun stl_find_first_not_of: string * string * size -> size
fun stl_find_first_not_of: string * +char -> size
fun stl_find_first_not_of: string * +char * size -> size
fun stl_find_first_not_of: string * char -> size
fun stl_find_first_not_of: string * char * size -> size

fun find_first_not_of (s:string, e:string) : opt[size]
fun find_first_not_of (s:string, e:string, i:size) : opt[size]
fun find_first_not_of (s:string, e:+char) : opt[size]
fun find_first_not_of (s:string, e:+char, i:size) : opt[size]
fun find_first_not_of (s:string, e:char) : opt[size]
```

### 22.2.13 Construe `string` as set of `char`

```
instance Set[string,char] {
  fun \in (c:char, s:string) => stl_find (s,c) != stl_npos;
}
```

### 22.2.14 Construe `string` as stream of `char`

```
instance Iterable[string, char] {
  gen iterator(var x:string) () = {
    for var i in 0 upto x.len.int - 1 do yield Some (x.[i]); done
    return None[char];
  }
}
inherit Streamable[string,char];
```

### 22.2.15 Test if a string has given prefix or suffix

```
fun prefix(arg:string,key:string)=
fun suffix (arg:string,key:string)
fun startswith (x:string) (e:string) : bool

// as above: slices are faster
fun endswith (x:string) (e:string) : bool

fun startswith (x:string) (e:char) : bool
fun endswith (x:string) (e:char) : bool
```

### 22.2.16 h2 Trim off specified prefix or suffix or both

```
fun ltrim (x:string) (e:string) : string
fun rtrim (x:string) (e:string) : string
fun trim (x:string) (e:string) : string
```

### 22.2.17 Strip characters from left, right, or both end of a string.

```
fun lstrip (x:string, e:string) : string
fun rstrip (x:string, e:string) : string
fun strip (x:string, e:string) : string
fun lstrip (x:string) : string
fun rstrip (x:string) : string
fun strip (x:string) : string
```

### 22.2.18 Justify string contents

```
fun ljust(x:string, width:int) : string
fun rjust(x:string, width:int) : string
```

## 22.3 Regexps

Syntax

Combinators

Google Re2 Binding

# Chapter 23

# Introspection

# Chapter 24

# Serialisation

Felix allows certain data structures to be serialised into a string, so that the data structure can be recovered from the string.

## 24.1 Operation

### 24.1.1 `encode_varray`

This function accepts the address of a serialisable object, and returns a string containing the serialised contents of the object.

### 24.1.2 `deccode_varray`

This accepts a string previously serialised by `encode_varray`, rebuilds a copy of the data structure it encoded, and returns the address of the object.

### 24.1.3 Usage

The data structure to be serialised must reside entirely on the heap and satisfy other requirements. A pointer to the object is cast to type `address`. When deserialising, the address returned should be cast to a pointer to the same data type used in serialisation.

```
include "std/felix/serialise";
var px: &list[int] = (1,2,3,4).list.new;
var serialised_x : string = px.addresss.Serialise::encode_varray;
var px_copy = serialised_x.Serialise::decode_varray.&list[int];
println$ *px == *px_copy;
```

### 24.1.4 Process Restrictions

Serialisation functions depend on the machine address of the Felix run time type information (RTTI) objects which also control garbage collection.

Therefore serialisation is only guarranteed to work within a single process. On some architectures with some linkage strategies it may be possible to save the string to disk, and reloaded it in a different process running the same program. In particular the core requirement must be met by the operating system: to load the process at the same virtual address. If you wish to try this, it is best to statically link your program since different orders of dynamic linkage may result in different addresses for shared library objects (DLLs).

If serialisation works across different processes, it should also work if the serialised data is transmitted to an remote machine running the same architecture, same operating system, and binary identical Felix program.

### 24.1.5 Data Restrictions

All data to be serialised must either be a C data type bound to Felix with supporting serialisation and deserialisation functions provided in the binding, or, any Felix combination of such primitive data types effected by a tuple, array, struct, union, sum or unit sum type, or pointer type. Recursive data types are supported. Circular data structures do not present a problem.

Most Felix primitive types provided in the library are supported, except for data types derived from Judy arrays.

You cannot serialise function or procedure closures because these refer to the global thread frame object, which is the root for most of the rest of the program data: serialisation would then attempt to capture the whole program. Even if the thread frame were serialisable, deserialisation would create a copy of it which is not supported.

Pointers to C functions, however, are serialisable since they're just addresses. Pointers to an foreign objects remain invariant. Most simple data structures are serialised by simply using a blit operation, namely `memcpy`.

C++ strings are serialisable. Since serialisation produces a C++ string, data structures containing serialised data are also serialisable.

### 24.1.6 How it works

Serialisation works by copying serialising primitives with user supplied serialisation functions . These currently must be written in C++. Non-primitives are serialised by copying the object and finding subobjects to also serialised using the same pointer location data the garbage collector uses from the types RTTI object.

Serialised data is tagged with the address of the RTTI object so that the deserialisation function and location of interior pointers can be found.

### 24.1.7 Future Directions

A more portable method of locating RTTI than using the machine address would remove many of the restrictions, in particular the restriction that the exact same program must be used for deserialisation as for serialisation, as well as the restriction that deserialisation should be in the same processes.

In fact the primary difficulty here is finding a unique universal representation of a data type. Were such a kind available a search could be used to map universal type tokens to the RTTI address.

More portable representations of the data may also remove the restriction that the same processor must be run the processes with the same underlying binary ABI (so that, for example, an `int` has the same binary representation on both architectures).

# Chapter 25

# Fibres and Schannels

Library Package

Fibres (or f-threads) are coroutines which exchange control with other fibres by reading and writing synchronous channels (schannels).

## 25.1 `spawn_fthread` procedure

The `spawn_fthread` procedure spawns a new fibre (fthread). The procedure being spawned must have type `unit->void` also written `1->0`.

The new fibre begins execution immediately and continues until suspended by either an unmatched I/O request on an schannel, or a read operation matched by a write.

Fibres are similar to threads, howver control exchange is synchronous, not pre-emptive. A communicating set of fibres runs in a pre-emptive single thread.

Fibres can also be suspended waiting for asynchronous events such as socket I/O or a timer.

## 25.2 Schannel types

There is only one kind of schannel, however the type system provides three views:

**ischannel[T** ] an input schannel for reading T values,

**oschannel[T** ] an outputput schannel for writing T values,

**ioschannel[T** ] an input/output schannel for either reading or writing T values.

Each of these types is also a constructor for the corresponding type of schannel.

## 25.3 `mk_ioschannel_pair[T]` function

This function creates a single schannel, returning a read end and a write end in a tuple:

```
    var ins, outs = mk_ioschannel_pair[int]();
```

## 25.4 `ioschannel[T]` constructor

A bidirectional channel can also be created:

```
    var ios = ioschannel[int]();
```

## 25.5 `read` generator and `write` procedure

The read and write operators on schannels are used for communication between fibres.

- When a read matches a write, the reader continues with the data, and the writer is placed by on the scheduler active list. If there is more than one writer, the writer chosen is unspecified.
- If a read has no matching write, the reading fibre joins a set of fibres waiting on the schannel for a write, and the scheduler begins execution of one fibre from its active list.
- A write will transfer control and data to one reader. The reader chosen is unspecified. The writer is then placed on the scheduler active list.
- If a write has no matching read, the writing fibres joins a set of fibres waiting for a read, and the scheduler begins execution of one fibre from its active list.

Thus, an schannel is, in effect, a localised scheduler, queueing either writers or readers until matching operations are performed.

## 25.6 Cross thread I/O

Schannel read and write operations are not thread safe. Schannels should usually not be shared between pthreads to ensure I/O cannot be performed between

pthreads using schannel I/O. Pthreads can use `pchannels` to communicate instead. On the other hand fibres must not use pchannels to communicate within the same pthread since this would be an automatic deadlock.

## 25.7 `broadcast` procedure

This procedure writes to all fibres waiting on an schannel, then returns. Unlike `write` it does not block.

`broadcast` can be used directly from C++. It is intended to permit custom C++ driver code to communicate with a suspended Felix system. The driver code must ensure the data sent with a broadcast remains live whilst target fthreads are processing it.

Felix `broadcast` is invoked by the member function `external_multi_write` of the object of type `async_sched` with type shown below, passing the schannel to broadcast over, and a pointer to the data to broadcast.

```
void async_sched::external_multi_swrite(
  ::flx::rtl::schannel_t *chan,
  void *data
)
```

Felix can be embedded in C++ code, in particular it can run in framework loop idle function as a callback. It runs until there are no active fibres. The `external_multi_write` is used to schedule more active fibres so the next resumption via the idle loop callback has work to do.

### 25.7.1 Fibre States

Fibres may be in one of five states:

**Running** Only one fibre can be running at a time in a given pthread.

**Waiting to Run** A fibre is active if it is on the scheduler active list but not running.

**Waiting on Schannel** A fibre may be waiting for either input or to output.

**Waiting for Asynchrons Event** Fibres may also wait for asynchronous events.

A fibre which is running or waiting to run is said to be *active*. A fibre which is waiting on an schannel or asynchronous event is said to be suspended.

### 25.7.2 Schannel States

An schannel may be in one of three states:

**Empty** The schannel has no fibres on its wait list.

**Waiting for a reader** The fibre has one or more writers on its wait list.

**Waiting for a writer** The fibre has one or more readers on its wait list.

### 25.7.3 Abbreviated Type names

Felix provides short hand notation for input and output schannels as follows:

| alias | full name | semantics |
|---|---|---|
| `%<T` | ischannel[T] | input schannel |
| `%>T` | oschannel[T] | output schannel |
| `%<>T` | ioschannel[T] | bidirectional schannel |

### 25.7.4 Deadlock, Livelock, and Suicide

Both fibres and schannels are garbage collected. Thus they either be reachable or unreachable.

If a fibre is active it is a root, and is always reachable. If a fibre is waiting for an asynchrons event it is also always reachable.

An schannel may be also be reachable if it is stored in a variable in the global thread frame. This is the global data object which is shared between all preemptive threads.

Alternatively it may be reachable from a procedure reachable from an active fibre.

A fibre may terminate by returning. In this case it is unreachable and will be collected. This is known as normal termination.

A fibre which can only be reached from an unreachable schannel will be garbage collected. This is known as suicide and ensures fibres cannot deadlock: deadlocked fibres suicide and so aren't deadlocked because they no longer exist. For example consider:

```
begin
  var i1,o1 = ml_ioschannel_pair[int]();
  var i2,o2 = ml_ioschannel_pair[int]();
  spawn_fthread { write (o1,1); println$ read i2; };
  spawn_fthread { write (o2,2); println$ read i1; };
end
```

The first fthread suspends on channel 1, waiting to write, the second fthread suspends on channel 2, waiting to write. The fthreads are suspended and deadlocked. However the main fthread is active and now runs, and the begin/end

block returns so its data frame, containg the two schannels, is no longer reachable. This means the two suspended fthreads are not reachable either. So the two fibres have suicided, they are no longer deadlocked because they no longer exist.

Suicide is not necessarily an error, it is, in fact, a common way to terminate fthreads. The problem above terminates correctly.

Now, consider the program above without the begin/end block. In this case, the fthreads will not be collected because they're reachable from global store (the so-called thread frame). So that program will not terminate, and is said to be *live-locked*.

It is a live-lock because the main fthread can read the channels and thus the suspended fibres and *could* unsuspend them with suitable I/O operations.

It is the responsibility of the holder of an schannel to ensure other holders I/O requests are satisfied. If a fibre does not intended to perform I/O on an schannel it should forget it. In the example the main thread forgets the schannels by forgetting the begin/end blocks data frame when it returns.

### 25.7.5 Restrictions on use

Fibres can only be spawned with a basic `spawn_fthread` by top level procedures or procedural desendants, or, from functions or generator which are inlined (eliminating the function).

This is because procedures spawned with a basic `spawn_fthread` with a service call which requires the machine stack be empty, since the service calls are effected by *returning* control to the scheduler.

Similarly, the same restrictions apply to schannel I/O.

However these restrictions can be partly overcome with nested schedulers.

## 25.8 Nested Scheduling

A nested scheduler of type `fibre_scheduler` may be created a constructor accepting a unit argument. It is must be deleted when no longer required by the procedure `delete_fibre_scheduler`.

Fibres may be added to the scheduler using the advanced `spawn_fthread` procedure, this procedure takes a `fibre_scheduler` argument and returns a closure accepting procedure of type `1->0`. Fibres starting with the procedure are added to the nested scheduler list, but are not yet run.

The `run` procedure takes a populated scheduler and runs the fibres on it.

Fibres on nested schedulers may communicate with each other using schannel operations.

The run procedure will not return until all the fibres on the nominated scheduler have completed. The fibres may not do asynchronous operations, or make service calls other than schannel I/O operations, however the scheduler handles schannel I/O properly.

The result of attempting to communicate with fibres on another scheduler such as the master scheduler are undefined.

```
fun foo () =
{
  var chin, chout = mk_ioschannel_pair[int]();
  var sched = fibre_scheduler();
  var res = 1;
  spawn_fthread sched { write (chout,42); };
  spawn_fthread sched { res = read chin; };
  run sched;
  delete_fibre_scheduler sched;
  return res;
}
println$ foo();
```

## 25.9 Synchronous pipelines

Library Reference lah.

A synchronous pipeline is a special case of fibres and channels in which data flow from one fibre to a series of others in sequence using a one schannel for each adjacent pair to connect them.

Special constructors are provided to make pipelines which ensure that only the fibres in the pipeline know the connecting schannels.

This ensures that the pipeline collapses when there is no more data to processes.

The starting point of a pipeline is known as a source, the ending point a sink, and the pieces in the middle are called transducers.

A transducer is a control inverted function. This means that instead of being called with data, it reads it, and instead of returning with a result it writes it.

Unlike functions, which form client/server relations, tranduces are all peers.

## 25.10 Buffers

Synchronous I/O is unbuffered. This means the sequencing of reads and writes is critical. If a fibre writes channel A then B, and another reads B then A, the system is deadlocked and the fibres may suicide.

In order to read a set of inputs in an arbitrary order, buffers can be added to each input.

A buffer is simply an fibre running an infinite loop which reads from one channel and writes on another:

```
var Ain,Aout = mk_ioschannel_pair[int]();
var Bin,Bout = mk_ioschannel_pair[int]();
var Cin,Cout = mk_ioschannel_pair[int]();
var Din,Dout = mk_ioschannel_pair[int]();

proc buffer (r:%<int, w:%>int) () {
  while true perform write (w,read r);
}

spawn_fthread { write (Aout, 1); write (Bout,2); };
spawn_fthread { var d = read Din; c = read Cin; println$ c,d; };
spawn_fthread$ buffer (Ain,Cout);
spawn_fthread$ buffer (Bin,Dout);
```

Now, as you can see, the reader procedure reads the data from A in the reverse order to that in which it is written, but this works because of the buffers. Note that the first fthread can now proceed as soon as the buffer has read its output, before the second fthread gets it. In fact the second fthread gets what it wrote first.

Buffers are mandatory for a feedback.

# Chapter 26

# Asychronous Events

blah.

# Chapter 27

# Pre-emptive Threading

Library Package Felix provides pre-emptive threading based on OS native threads called *pthreads*.

Felix pthreads are detached, there is no support for OS joinable threads. The primary synchronisation primitve is the *pchannel*.

## 27.1 `spawn_pthread` procedure

The `spawn_pthread` procedure spawns a new pthread. The procedure being spawned must have type `unit->void` also written `1->0`.

## 27.2 Pchannel types

There is only one kind of pchannel, however the type system provides three views:

**ipchannel[T** ] an input pchannel for reading T values,

**opchannel[T** ] an outputput pchannel for writing T values,

**iopchannel[T** ] an input/output pchannel for either reading or writing T values.

Each of these types is also a constructor for the corresponding type of pchannel.

## 27.3 mk_iopchannel_pair[T] function

This function creates a single pchannel, returning a read end and a write end in a tuple:

```
var inp, outp = mk_iopchannel_pair[int]();
```

## 27.4 iopchannel[T] constructor

A bidirectional channel can also be created:

```
var ios = iopchannel[int]();
```

## 27.5 read generator and write procedure

The read and write operators on pchannels are used for communication between pthreads.

Pchannels are technically *monitors*. There is no buffering. Unmatched reads block until a matching write occurs on the same underlying pchannel. Unmatched writes block until a matching read occurs. Neither reader nor writer may proceed until data is exchanged.

Unlike fibres and schannels, pthread can be deadlock by out of order I/O.

## 27.6 concurrently procedure

```
open class ForkJoin {
  proc concurrently_by_iterator (var it:1 -> opt[1->0])
  proc concurrently[T with Streamable[T,1->0]] (d:T)
}
```

Felix provides the procedure `concurrently_by_iterator` which uses an iterator returning procedures of type `1->0` to launch pthreads.

A more convenient procedure, `concurrently`, accepts any data structure with an iterator.

`concurrently` automatically joins the launched threads.

```
// using an array
concurrently (
  { println$ "Thread1"; },
  { println$ "Thread2"; },
  { println$ "Thread3"; }
);
```

## 27.7 Atomics

Felix atomics utilise the C++11 standard library atomics. Atomic variables are not managed by the GC.

```
open class Atomic
{
  type atomic[T];
  ctor[T] atomic[T]: T;
  proc delete[T] : atomic[T;

  gen load[T] : atomic[T];
  proc store[T] : atomic[T];
  proc store[T] (a:atomic[T]) (v:T);

  // integer types only
  proc pre_incr[T] : &atomic[T];
  proc pre_decr[T] : &atomic[T];

}
```

## 27.8 Mutual Exclusion Lock

```
open class Mutex
{
  type mutex;
  ctor mutex: unit;
  proc lock: mutex;
  proc unlock: mutex;
  proc destroy: mutex;
}
```

Felix provides a garbage collector aware mutual exclusion lock `mutex`. It is a timed mutex which periodically checks for a GC world stop request.

Condition variables are not garbage collected.

## 27.9 Condition Variables

Felix provides a GC aware condition variable based on C++11 condition variables. It uses a timed variant to periodically check for a GC world stop request.

```
open class Condition_Variable
{
  type condition_variable;
  ctor condition_variable: unit;
  proc destroy: condition_variable;

  //£ lock/unlock associated mutex
  proc lock : condition_variable;
  proc unlock : condition_variable;

  proc wait: condition_variable;
  proc signal: condition_variable;

  proc broadcast: condition_variable;

  gen timedwait: condition_variable * double -> int;
}
```

Note that unlike Posix condition variables, the associated mutex is built-in to the condition variable.

## 27.10 Bound Queue

Felix provides thread safe, GC aware, resizeble bound queue. The queue itself is garbage collected. The queue stores pointers to heap allocated copies of the queued objects, which are scanned by the GC.

```
open class TS_Bound_Queue
{
  ctor[T] ts_bound_queue_t[T]: !ints;
  proc enqueue[T] (Q:ts_bound_queue_t[T])  (elt:T);
  gen dequeue[T] (Q:ts_bound_queue_t[T]): T;
  proc wait[T]: ts_bound_queue_t[T]; // until Q empty
  proc resize[T]: ts_bound_queue_t[T] * !ints;

}
```

The constructor creates a queue with the specified maximum capacity or *bound*.

The `enqueue` operation may wait until there is space in the queue before depositing its payload and returning.

The `dequeue` operation may wait whilst the queue is empty until it can acquire a payload before returning.

The `wait` procedure blocks until the queue is empty.

The `resize` procedure changes the queue bound.

Note: When threads are enqueing and dequeing from the queue, there is no defined method for closing down the use of the queue. Neither `enqueue` nor `dequeue` can fail, nor can they be cancelled.

## 27.11 Thread Pool

```
class ThreadPool
{
  typedef job_t = 1 -> 0;
  fun get_nthreads () => nthreads;

  proc barrier();
  proc start ();
  proc start (n:int);
  proc queue_job (job:job_t);
  proc stop ();
  proc post_barrier();
  proc notify (chan:opchannel[int]) ();
  proc join ();

  proc pfor_segment (first:int, last:int) (lbody: int * int -> 1 -> 0)
  inline proc forloop (lbody: int -> 0) (first:int, last:int) ()
}
```

Felix provides a global thread pool which consists of a thread safe job queue and a set of running Felix pthreads. The number of threads is determined from the architecture if possible, otherwise set to a reasonable value such as 8.

The job queue is bound to 1024 jobs.

The `queue_job` procedure accepts a unit procedure of type `1->0` and adds it to the job queue. It may block whilst the job queue is full. Queing a job automatically starts up the thread pool.

Threads in the pool dequeue jobs from the queue and run them. When finished, a thread fetches another job if there is one, and runs it. If there are no jobs, the thread waits until a job is available.

The `start` procedure can be used to manually start the thread pool.

The `post_barrier` procedure posts $n$ `barrier` jobs, where $n$ is the number of threads in the pool. A barrier job causes a thread to stop acquiring jobs and

wait until all threads are waiting on a barrier.

The `join` procedure posts $n$ barriers, where $n$ is the number of threads in the pool. This causes each thread to suspend on a barrier until all threads have suspended. Then all threads may continue processing jobs.

The `stop` procedure attempts to stop the pool by posting a $n$ special `ThreadStop` jobs, where $n$ is the number of threads in the pool. A `ThreadStop` job causes the thread executing it to suicide.

### 27.11.1 Restrictions

Jobs submitted to the thread pool should be either be entirely independent or transiently coupled. In general, jobs should not communicate with each other or other threads using condition variables or pchannels. It is counter productive for jobs to spawn new threads although it is not disallowed, because the thread pool is designed to saturate multi-core CPUs (that is, keep all the cores busy).

Transient coupling, however, is permitted. For example if jobs are used to calculate some value which is then added to a single variable, a mutex can be used to serialise access to the variable, because the lock will only be held transiently.

## 27.12 Parallel For Loop

# Part VIII

# Data Structures

# Chapter 28

# Iterators

:

# Chapter 29

# Lists

# Chapter 30

# Arrays