# ALFL Reference Manual and Programmer's Guide

**Paul Hudak**

**Technical Report YALEU/DCS/TR-322**
**Second Edition -- October 1984**

Yale University
Department of Computer Science
New Haven, CT

This research was supported in part by NSF Grant MCS-8106177.

# Table of Contents

# ALFL Reference Manual and Programmer's Guide

## Second Edition

**Paul Hudak**
**Yale Department of Computer Science**
**October 1984**

## 1. Introduction

ALFL is a functional language developed at Yale that historically grew out of a toy language called Mini-FPL used in CS-521, a compiler course taught at Yale. It is essentially a blend of the better ideas from David Turner's SASL [8] and Robert Keller's FEL [5], although a few significant new features have been added, including a more powerful pattern-matcher, the distinction between "list generators" and "ordered set generators", and the inclusion of a fail semantics for equations. The language's semantics tries to be as "lazy as possible" (i.e., a function is strict in as few of its arguments as semantically makes sense), and the base language has (of course) no assignment statement or "impure" functions (i.e., those that produce side-effects). However, since ALFL will undoubtedly be implemented on conventional computers with conventional file servers and device handlers, there exists ways to force an evaluation and produce side-effects in the implementation environment. Our goal is to provide a *practical* programming language based on the functional style, by allowing "controlled" use of side-effects in that great big side-effect pit known as the real world.

This manual is intended to describe ALFL's syntax and semantics, together with the pragmatics of its primary implementation to date, a system built upon T [6] called *Alpha-Tau* that runs on Apollo Aegis, Vax Unix, and Vax VMS. This manual is not intended to be a tutorial on functional programming, although enough examples are given that the experienced programmer may find it adequate. The reader interested in learning more about functional programming is encouraged to read Peter Henderson's book, *Functional Programming: Application and Implementation* [3], as well as the collection of papers in [2].

### Comment on Preliminary Version of this manual:

This is the first-ever attempt at documenting ALFL and Alpha-Tau -- there are guaranteed to be errors and omissions -- BEWARE! The auther welcomes all comments, criticisms, suggestions, and bug reports on either the language, its implementation, or this document.

## 2. Overview

A detailed BNF-style syntax for ALFL may be found in the Appendix. A denotational description of its semantics is forthcoming. In this section we attempt to give an informal overview of the language, after which we will treat each semantic feature in more detail.

An identifier in ALFL is represented by any non-empty sequence of alpha-numeric characters plus the symbol "_". There are many pre-defined identifiers such as `add`, `or`, and others, but they may all be redefined at will -- the only reserved word in the language is `result`, whose purpose is explained in the next paragraph. ALFL also has many infix operators, all represented by *symbols* (i.e., non-alphabetic characters). Each operator also has an equivalent "curried" function (assigned to a specific identifier), and when introducing an infix operator we will always include its curried version in parentheses, as in "`+ (add)`" and "`| (or)`". A summary of these operator/function equivalences is given in the appendix.

ALFL is block-structured, and lexically-scoped. Its largest syntactic object is an *equation group*, which is delimited by curly brackets ("{...}"). Within an equation-group is a collection of *equations* that map identifiers to particular values (which may be any of the allowable types described in the next section), together with a single *result clause* that expresses the value to which the equation-group will evaluate. A double equal-sign ("==") is used for equations to distinguish it from the infix operator for equality. An equation-group is a special case of an *expression*, and is thus valid wherever an expression is allowed. A conditional expression has the form "`pred -> cons, alt`" and is equivalent to the more conventional "`if pred then cons else alt`". Here is a simple example:

```
{ fac n == n=0 -> 1, n*fac(n-1);   % Definition of factorial
  x == 10;
  result fac x }
```

Note that comments are preceded by "%" and continue till the end of the line.

ALFL scoping rules are similar to those for most block-structured languages, in that expressions may reference any identifier defined locally in the current equation-group or in any surrounding equation-group. However, local references are allowed to be mutually recursive. Indeed, equations may appear in any order, consistent with ALFL's lazy evaluation semantics in which expressions are evaluated "by demand". Thus in the above example, the two equations plus the result clause may appear in any of six different orders. Of course, it is illegal to define an identifier more than once within the same scope -- the equations should be thought of as a *naming* discipline that maps identifiers to values, and not as an *assignment* operation.

All function applications are "curried". That is, all functions are assumed to take just one argument, which is no restriction since that function may return another function that takes one argument, etc. If we define function application to associate to the left, curried functions facilitate the use of higher-order functions, as in:

```
{ twice f x == f (f x);
  twofacs == twice fac;
  result twofacs 10 }
```

which is precisely equivalent to:

```
{ twice f == { result g;
               g x == f (f x) };
  twofacs == twice fac;
  result twofacs 10 }
```

Since it is useful at times to perform right-associative function application, ALFL provides the infix binary operator ":" (`apply`), which is defined to have *higher* precedence than normal (left-associative) application (thus in the above example we may write `twice f x == f:f:x` or even `twice f x == f f:x`). This convention turns out to be visually pleasing, since expressions such as `f hd:x tl:y` group together in the way they appear; i.e., as `f (hd x) (tl y)`.

One final note on function application: "." (compose) is the infix operator for function composition, and thus "`f.g`" denotes the composition of f with g; "`h == f.g`" is equivalent to "`h x == f:g:x`".

## 3. Data Types

Types do not exist explicitly in ALFL; rather, an object's type may be inferred from the behavior of one of several *type predicates* when passed the object as an argument. This weak characterization of type is consistent to that used in most Lisp dialects.

### 3.1. Primitive Types

There are six primitive data types in ALFL, together with a predicate that characterizes each by returning `true` when given an argument of that type, `false` otherwise:

- *Boolean*, containing values `true` and `false`, and with predicate `boolp`.
- *Integer*, the set of positive and negative integers with an implementation-dependendent range, and with predicate `intp`.
- *Floating-point*, the set of floating-point numbers whose range is also implementation-dependent, and with predicate `floatp`. A floating-point number may be created explicitly by including a decimal point in its representation; e.g., `2.`, `1.23`, etc.
- *String*, the set of arbitrary-length strings, with predicate `stringp`. Strings are represented in the standard way, using a pair of double-quotes surrounding an arbitrary sequence of characters (including carriage returns). A double-quote may be included in the string by juxtaposing two double quotes, as in: `"this is a string with a double-quote "" in it"`. The null string is denoted by `""`.
- *Function*, the set of primitive functions plus any user-defined functions, with predicate `functionp`.
- *Pair*, the set of composite objects usually formed by the infix pairing operator "^" (`fby`). The predicate for this type is `pairp`. (Pairs and lists are discussed in much more detail in the next section.)

In addition, there is a composite type *number*, which is essentially the join of integer and floating-point, and has predicate `nump`. The sub-type *list* contains the primitive value [] (the empty, or null list) together with all pairs whose second component is either [] or a pair; the predicate for lists is `listp`. Finally, the special value $\perp$ (read "bottom") is the undefined value, used in this manual to describe undefined behavior, which is an implementation-dependent concept that on most systems will result in either an "error-break" or non-termination.

## 3.2. Equality in ALFL

The binary infix operators = (`equal`) and <> (`notequal`) are used to establish equivalence between objects in ALFL. Equality is in general a surprisingly difficult concept to define semantically and then get implementationally correct; we have tried (as usual) to take the most functional approach, resulting in the following recursive equality semantics. Two objects are `equal` if:

1. They are identically the same object, or
2. They are integers and represent the same number, or
3. They are floating-point numbers and represent the same number, or
4. They are strings and have the same character sequence, or
5. They are booleans and have the same truth value, or
6. They are both the empty list [], or
7. They are pairs and each of their corresponding elements are equal.

Two objects are `notequal` if they are not `equal`.

The subtle aspect of the above definition is the statement that two objects are equal if they are "identically the same object" -- this is unfortunately an implementation-dependent concept. For example, can one compare two identical infinite lists? As in:

```
{ ones == 1 ^ ones;
  more_ones == ones;
  result ones = more_ones }
```

One would like this result to be `true` (and indeed it is in Alpha-Tau), but it is conceivable that some implementation might copy the expression for `ones` when assigning it to `more_ones`, so that the objects are not identical; argument passing in funciton calls provides further complication. Or consider:

```
{ ones == 1 ^ ones;
  more_ones == 1 ^ more_ones
  result ones = more_ones }
```

A clever implementation might recognize common-subexpressions and collapse `ones` and `more_ones` into one (pardon the pun), thus yielding `true` for the result. An implementation not doing this will try to compare the lists element-by-element, and thus not terminate.

Infinite lists are not the only place where "identical objects" can cause implementation-

dependent behavior -- consider the comparison of functions. The correct solution to this problem is to eliminate the clause defining equality on "identical objects", but the utility of that clause has so far dominated, and the current semantics has prevailed. The author welcomes opinions on this matter.

## 4. Lists

Finite lists may be constructed explicity using square brackets surrounding the elements separated by commas, as in "`[x,y,z]`", which is the list of three elements, `x`, `y`, and `z`. The empty list is denoted by `[]`. There is also an infix pairing operator `^` (`fby`); `x^y` is read "x followed by y". `fby` is more primitive than the the list expression just described, in that `[x1,x2,...,xn]` is equivalent to `x1^x2^...^[]` (the operator `^` is right-associative). Semantically, lists are constructed "lazily" in that the pairing function is not strict in either of its arguments. That is, both `x^`$\perp$ and $\perp$`^x` are well-defined.

A list's components (or, more correctly, a pair's components) are selected by the primitive functions `hd` (read "head") and `tl` (read "tail"), defined by:

```
hd (x^y) → x
hd []    → ⊥
tl (x^y) → y
tl []    → ⊥
```

This implies, of course, that:

```
hd [x1,x2,...,xn] → x1
tl [x1,x2,...,xn] → [x2,x3,...,xn]
```

"Infinite lists" may be defined in the obvious way. For example, the infinite stream of numbers starting at `n` may be defined by "`numsfrom n == n^numsfrom(n+1)`". As a more interesting example, consider this definition of the Fibonacci sequence:

```
{ fib == 1^1^addstreams[fib,tl fib];
  addstreams[x^S1,y^S2] == (x+y)^addstreams[S1,S2]; ... }
```

Of course, elements of an infinite list are not computed until they are selected ("demanded") for evaluation. The infix operator `^^` (**append**) is used to append lists together, and it too is lazy. Thus one can even append infinite lists together, as in "`fib^^fib`"; the second infinite stream of Fibonacci numbers is simply never reached!

### 4.1. Mapping Functions for Lists

There are four useful mapping functions for lists, essentially borrowed from FEL. They are best explained by examples.

**Reduction**, similar to that in APL, has the form: `[fn,init]//list`, where `fn` is (the curried version of) any binary associative function, `init` is the "default" value for empty lists, and `list` is the list to be reduced. For example:

```
[add,0]//[]       → 0
[add,0]//[1,2,3] → 6
[add,1]//[1,2,3] → 7
```

The curried version of // is reduce.

**Layered application** has the form fn\\list, where fn is (the curried version of) any *n-ary* function, and list has length *n*, each element being another list. This mapping function essentially "strips off" layers of list, applying fn to each layer, and returning the result in another list. For example:

```
add\\[[1,2,3],[4,5,6]] → [5,7,9]
add\\[[],[]] → []
```

If the sub-lists are of unequal length, the list returned is as long as the *first* sub-list, as long as the others are at least that long. Remember that because of lazy evaluation, infinite lists work equally well. For example:

```
add\\[numsfrom 1, numsfrom 2] → [3,5,7, ... ]
```

The curried version of \\ is layered_apply.

**Nested map** has the form fn||list, where fn is a unary function and list is an arbitrailiy-nested list. This operation returns a list having the same structure as list, except that fn has been applied to each of the atomic elements. For example:

```
succ||[1,[2,3],4] → [2,[3,4],5]
succ||[] → []
```

The curried version of || is nested_map.

**Structured application** has the form list::x where list is an arbitrary list of unary functions (and thus any curried function is allowed), and x is any object. This operation returns a list having the same structure as list, except that each function has been replaced by that function applied to x. For example:

```
[succ,pred]::3 → [4,2]
[]::3 → []
```

Note that since all functions in ALFL are curried, one can do things like:

```
([add,sub]::3)::2 → [5,1]
```

The curried version of :: is structured_apply.

## 4.2. Other List Utilities

As mentioned earlier, the predicate for lists is listp, and for pairs is pairp. There is also a predicate for atoms, atomp, which answers true to anything that is not a pair. The predicate nullp answers true only to the empty list [].

The remaining functions on lists are:

- `pre int list` returns the first `int` elements of `list`.
- `suf int list` removes the first `int` elements from `list`, and returns what's left.
- `nth int list` returns the `int`'th element from `list` (`nth 1 list` ≡ hd list).
- `length list` returns the number of top-level elements in `list`. (Guess what happens if `list` is infinite?)
- `reverse list` reverses the top-level structure of `list`.
- `member x list` returns true if `x` is equal to any of the top-level elements in `list`, false otherwise. `mem` is a synonym for `member`.
- For convenience, the 12 combinations of 2 or 3 `hd`'s and `tl`'s juxtaposed together are provided as built-in functions. They are: `hhd`, `htl`, `thd`, `ttl`, `hhhd`, `hhtl`, `hthd`, `httl`, `thhd`, `thtl`, `tthd`, and `tttl`. For example, `thtl x` is the same as `tl:hd:tl:x`.

## 5. The Pattern-Matcher

An interesting aspect of ALFL in its own right is its *pattern-matcher*, a feature that has become rather popular in several functional languages, including FEL, SASL, HOPE [1], and others. The idea is to provide a way for the programmer to define a function by writing patterns for its formal parameters on the left-hand side of an equation -- if the actual parameters in a function call match a particular pattern, then the expression on the right-hand side is evaluated. This gives a functional program very much of a "logical" style similar to Prolog. In ALFL we have attempted to carry the concept of pattern-matching as far as possible, including a fail semantics that provides a primitive level of back-tracking that is useful in search-oriented algorithms.

As a simple example, factorial can be defined by:

```
{ fac 0 == 1;
  fac n == n*fac(n-1); ... }
```

Here the first equation tries matching against a *constant*, namely 0. If that fails the second equation is tried, which always succeeds since the formal parameter `n` matches anything. The pattern-matcher also tests for the equivalence of multiple instances of the same formal parameter, as in:

```
{ eq x x == true;
  eq x y == false; ... }
```

which essentially defines `eq` as `equal`.

It should be noted that when using the pattern-matcher, the number of arguments in each equation defining the same function must be the same. Also, the equations must all be juxtaposed, and their *order* is important -- the left-hand-sides are tested sequentially for a match. Since this style of defining functions is very common, a short-hand is allowed for the equations following the first:

```
{ fac 0 == 1;
  '  n == n*fac(n-1); ... }
```

This is very convenient when defining functions whose identifier is rather long.

## 5.1. Destructuring

The pattern-matcher is even more useful when used with lists. For example, the function member may be defined by:

```
{ member x  []   == false;
    '    x (x^L) == true;
    '    x (y^L) == member x L;
 result member 2 [1,2,3] }
```

This style of "destructuring" may be carried to an arbitrary depth. It may also be useful in performing "multiple assignments" such as `[x,y] == [1,2]`.

## 5.2. Pattern Expressions

Sometimes it is convenient to match against some value that is unknown at compile-time; this may be accomplished by preceding the pattern with a # sign, denoting that the pattern is to be *evaluated* before the match is attempted -- such a pattern is called a *pattern expression*. For example:

```
{ one_less x #(x+1) == true;
     '     x   y    == false; ... }
```

Here the # sign indicates that the expression is to be evaluated rather than be interpreted as a pattern -- since x is also a formal parameter, its value in the pattern expression is gotten from the argument that will eventually match the formal parameter instance of x. Hence this function returns true if its first argument is one less than its second. The free variables in a pattern-expression reference values by observing the standard lexical-scoping rules. Thus in:

```
{ x == 5;
  equal_to_six #(x+1) == true;
        '         y    == false; ... }
```

x is not a formal-parameter to equal_to_six, so its value (5) is gotten from the surrounding environment.

As a final example of the use of the pattern-matcher, suppose one wishes to create one's own "type abstraction". For example, one might wish to have a type *tree* representing binary trees containing leaves and interior nodes. A possible implementation in ALFL might look like:

```
mk_tree lst rst == ["tree",lst,rst];    % creates a binary tree

treep ["tree",l,r] == true;             % predicate for tree
  '   x            == false;

lst ["tree",l,r] == l;                  % get left subtree
 '  x             == error "Can't take left subtree of non-tree";

rst ["tree",l,r] == r;                  % get right subtree
 '  x             == error "Can't take right subtree of non-tree";

f ["tree",l,r] == ... ;          % way to define a function
                                 % only defined on trees

["tree",l,r] == ...body...       % way to do "multiple assignment"
                                 % that will cause an error if the
                                 % object is not a tree
```

### 5.3. Anonymous Functions

It is often convenient to define "anonymous functions" (i.e., a function that does not have a name), especially when they are small and in general not worth cluttering up the name-space. ALFL provides a convenient way to do this that looks almost exactly like a normal equation, except that the identifier has been replaced by the symbol `@`. The general form is: "`@ args == body`". As a simple example, `(@ x == x+1) 2` evaluates to 3.

The pattern matcher may also be used as usual when defining nameless functions, except that only one equation is allowed. For example, `(@ [x,y] == x+y-2) [3,4]` → 5.

## 6. Ordered Bags

A useful attribute of ALFL is the ability to generate *ordered bags* in a convenient way. These objects are really just standard ALFL *lists*, but we provide a special syntax to generate them in a way that looks very much like Zermelo-Skolen-Frankel set notation. Despite this similarity, they are *not* true sets or bags, since they are represented as lists and are thus ordered. They are very useful nonetheless.

An ordered set is simply a list of objects with no duplicates, whereas an ordered bag is a list of objects possibly with duplicates. To create ordered bags we use *generators* to generate elements from other lists, and *filters* to restrict the values thus generated. An odered set may be generated from an ordered bag simply by writing a function that removes duplicates.

## 6.1. Ordered Bags

The form of an ordered bag in ALFL is:

```
[* exp ! gen1; gen2; ... genn ! fil1; fil2; ... filn *]
```

The square brackets are used to emphasize that what is actually generated is a standard ALFL list. The symbol ! is read "such that". The form of `exp` is arbitrary -- any valid expression is allowed (we discuss this further shortly). Each of the $gen_i$ is a *generator* of the form `pattern <- exp`, where `<-` is read "is an element of". Each of the $fil_i$ is a *filter*. A simple example is:

```
[* [a,b] ! a<-L1; b<-L2 *]
```

This example creates a list of pairs representing the Cartesian product of lists (presumable ordered sets) L1 and L2. As this example shows, a filter is not always necessary (but at least one generator is). The order of the generators determines the order in which the elements of the result are computed: the generators are "nested" such that the first (left-most) one is at the outer-most level, and the others are nested from left to right within the preceding one. In the above example, `b<-L2` is nested within `a<-L1`. Thus if `L1` is `[1,2]` and `L2` is `[3,4]`, then the example generates the ordered bag `[[1,3], [1,4], [2,3], [2,4]]`.

The pattern-matcher may be used (yet again!) with generators, as in:

```
[* ["bush",x] ! ["tree",x] <- L *]
```

Here only those two-element lists whose head is the string `"tree"` are taken from the list `L`. Note that although `["tree",x]` is a pattern, `"bush",x` is *not* -- it is simply an *expression* that denotes the form of the elements to be added to the ordered bag.

A filter is an arbitrary boolean expression, any number of which are allowed. Each filter $fil_i$ must evaluate to `true` if the currently created element is to be added to the ordered bag. For example, in:

```
[* a ! a <- L ! hd a > 0; length a = 3 *]
```

an ordered bag is generated of 3-element lists each of whose first element is positive. Note that the generator creates a "binding" for `a` whose scope includes both the filters and the result expression.

As a final example, consider this perspicuous definition of quicksort:

```
qs []    == [];
' (a^s) == qs [* b ! b<-s ! b<a *]^^[a]^^qs [* b ! b<-s ! b>=a *];
```

## 6.2. A Note on Scoping

If the reader keeps in mind the fact that lexical, mutually recursive scoping is used consistently throughout ALFL, and that the patterns in the set of generators introduce a lexical contour that surrounds the ordered set/bag being created (just as the patterns in a function call create a lexical contour for the body of the definition), then little confusion will result. For clarification, in this contrived example:

```
ob == [* [a,d] ! [#(b+1),a] <- L1; [b,#c] <- L2 ! f a = d *]
a == 1;      b == 2;
c == 3;      d == 4;
f == succ;
```

c, d, and f are free in the ordered bag, and thus the following bindings hold:

1. In [a,d], a is bound to the a in [#(b+1),a], and d is bound to 4.
2. In [#(b+1),a], b is bound to the b in [b,#c], and a is a formal parameter.
3. In [b,#c], b is a formal parameter, and c is bound to 3.
4. In f a = d, a is bound to the a in [#(b+1),a], and f and d are bound to succ and 4, respectively.

## 7. Arithmetic

ALFL provides a reasonably rich set of arithmetic operators and functions for integer and floating-point arithmetic. As mentioned earlier, the type predicates are intp, floatp, and nump, for integers, floats, and numbers, respectively.

### 7.1. Operators

There are several binary infix operators for standard arithmetic: + (add) for addition, - (sub) for subtraction, * (mult) for multiplication, and / (div) for division. All of these operators are defined for integers and floating-point numbers -- if at least one of the operands is floating-point, the result is floating-point, otherwise the result is integer.

There is also the standard set of relational operators, namely < (lt), for "less-than"; > (gt), for "greater-than"; <= (le), for "less-than-or-equal-to"; and >= (ge), for "greater-than-or-equal-to." Equality and inequality are established by = and <>, respectively, as discussed earlier.

In addition, there are two other operators, - (negate) for unary negation (context distinguishes this from the binary version), and \ (rem) for remainder (which requires that both operands be integers).

### 7.2. Arithmetic Functions

Many standard arithemtic functions are provided:

- Exponentiation: expt <num> <int> → <num>; so that expt x y → $x^y$.
- Absolute value: abs <num> → <num>.
- Greatest common divisor: gcd <int> <int> → <int>.
- Minimum: min <list-of-numbers> → <num>; returns smallest element in the list.
- Maximum: max <list-of-numbers> → <num>; returns greatest element in the list.
- Factorial: fac <pos-int> → pos-int; standard factorial.
- Coercion: int_to_float <int> → <float>; float_to_int <float> → <int>. float_to_int rounds the number down.

- Successor: `succ <num>` → `<num>`; adds one to a number.
- Predecessor: `pred <num>` → `<num>`; subtracts one from a number.

## 7.3. Trigonometric Functions

The following trigonometric functions are provided, defined only on floating-point numbers: `exp` (exponential function), `log` (natural logarithm), `sqrt` (square root), `cos` (cosine), `sin` (sine), `tan` (tangent), `acos` (arccosine), and `asin` (arcsine).

## 7.4. Bitvector Functions

The following functions are defined on integers treated as bitvectors: `logand` (the logical and of the bits), `logior` (the logical inclusive-or of the bits), `logxor` (the logical exclusive-or of the bits), and `lognot` (the logical complement of the bits). In addition, `shift int1 int2` shifts `int1` by the number of places specified by `int2` -- left if positive, right if negative -- filling in zeros as it goes. `bit_field int pos size` (all arguments integers) extracts a bit-field of length `size` from `int` starting in position `pos`.

## 8. Logical Operators

The conditional expression was described earlier; but note that the value of `p` in "`p --> cons, alt`" must be of type boolean. The right arrow may either be `->` or `-->`. The predicate for boolean values is `boolp`.

There are three logical functions provided as infix operators in ALFL: `&` (and), `|` (or), and `~` (not), which perform the logical and, or, and complement of their operands, respectively. `&` and `|` are binary, `~` is unary. The order of precedence, from highest to lowest, is `~`, `&`, `|`.

Semantically, the binary logical operators `&` and `|` are *sequential* -- that is, their operands are evaluated from left to right. This means that they realize the truth tables shown below:

| x \ y | true | false | ⊥ | <non-bool> |
|---|---|---|---|---|
| true | true | false | ⊥ | error |
| false | false | false | false | false |
| ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| <non-bool> | error | error | error | error |

x & y

| x \ y | true | false | ⊥ | <non-bool> |
|---|---|---|---|---|
| true | true | true | true | true |
| false | true | false | ⊥ | error |
| ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| <non-bool> | error | error | error | error |

x | y

The reader may question why we have chosen the "sequential" versions of these operators instead of the more pleasing "parallel" versions which have symmetry with respect to their operands. This would allow truly "lazy" semantics in that, for example, ⊥ | true could be defined as true. The problem here is unfortunately again implementation related -- to implement the lazy semantics implied by the parallel operators requires either a true parallel machine (that we don't have), or the simulation of a non-deterministic Turing machine (which we just don't want to do). Future implementations of ALFL on parallel machines will have curried versions of the parallel operators.

## 9. Strings

As mentioned earlier, the predicate for strings is `stringp`. There are also two other predicates, `alphap` which returns `true` if the first character in its string argument is one of the 26 upper- or lower-case letters, and `digitp` which returns `true` if the first character in its string argument is one of the digits 0 through 9.

Other operations on strings include:

- `str_append str1 str2` returns the concatenation of strings `str1` and `str2`.
- `str_hd str` returns the first character in `str` as a string of length one.
- `str_tl str` returns `str` with the first character removed.
- `str_nth int str` returns the `int`'th character in `str` as a string of length one (`str_nth 1 s` ≡ `str_hd s`).
- `str_length str` returns the number of characters in `str`.
- `str_to_ascii str` returns the ascii integer corresponding to the first character in `str`.
- `ascii_to_str int` returns the character (as a string of length one) corresponding to the ascii integer `int`.
- `format_string str list` is like `format` in T; it returns a string formed by splicing in the values of the elements of `list` in their successive positions in `str`, according to the following convention: Whenever a ~ (tilde) is encountered in `str` it is replaced by the next element in `list` in a way specified by the character following the ~:
  - ~a Insert the next element in the list in its standard representation.

- ~b If the next element is an integer, insert it in binary.
- ~o If the next element is an integer, insert it in octal.
- ~p If the next element, which must be a number, is not equal to one, insert an s (useful for plurals).
- ~nr If the next element is an integer, insert it in radix n.
- ~nt Move to a position n characters from the beginning of the string, inserting spaces in between ("tab").
- ~x If the next element is an integer, insert it in hexidecimal.
- ~% Insert a line-feed/carriage-return (newline).
- ~& Insert a line-feed/carriage-return if not already at the beginning of a line (freshline).
- ~_ Insert a space.
- ~~ Insert a ~.

A tilde followed by any whitespace is ignored, along with all following whitespace. Here are some simple examples:

```
format_string "Hi ~a; I have ~b apple~p." ["Cristy", 3, 3] →
        "Hi Cristy, I have 11 apples."
format_string "To break up a long line, ~
                                  do this" [] →
        "To break up a long line, do this"
format_string "Here's what a list looks like: ~a" [[1,2,3]]; →
        "Here's what a list looks like: (1 2 3)"
```

- str_explode str returns a list of the characters in str, each represented as a string of length one.
- str_implode list concatenates all of the top-level elements of list, which should be strings (str_implode [str1,str2] ≡ str_append str1 str2).
- gen_str str returns a unique string everytime it is called, and is thus an impure function. The string returned begins with the characters in str and ends with an implementation-dependent sequence of characters. Care should be taken when using this function: "everytime it is called" is itself an implementation-dependent concept since, for example, common-subexpression elimination may fold several calls into one!

## 10. I/O

Because most file systems and interfaces to the real world are very side-effect oriented, we felt it necessary to introduce similar features into ALFL. The set of impure functions described in this section have been implemented in Alpha-Tau, although a different set may be more appropriate for some other system -- indeed, perhaps a future "applicative architecture" will make all this unneccessary.

Despite the "impurity" of the I/O operations that we have introduced, we have attempted to make them as consistent with the functional style as possible; for example, all operations on files are done through *streams* (i.e., a *list*, not a stream in the T sense). The judicious programmer may still write purely functional programs with these functions.

## 10.1. Forcing Sequential Execution

One of the primary complications of side-effects is that one must be able to *order their execution*. In conventional languages the flow of control explicit in the semantics accomplishes this ordering. In ALFL, however, there is no explicit flow of control (a feature!), and so it is necessary to introduce a special operator to do this if we are to admit any form of side-effects. The simple function `force` accomplishes this by taking a list argument and sequentially evaluating each top-level element in turn, returning the *last* element in the list. For example, `force[print "foo", print " on", print " you"]` will guarantee that the message appears as intended, and will return the string `" you"` (since `print` functionally mimics the identity function).

## 10.2. Terminal Output

When an ALFL program is executed (in Alpha-Tau), the result is printed on the terminal. This is the simplest form of I/O, and is the recommended way to output to the terminal. However, together with `force`, the following two functions may provide an alternative technique: First, `print ob` causes the value of `ob` to be printed on the terminal; a "freshline" preceeds the value, and a "newline" follows it. The value of `ob` is returned. Second, `format str list` causes the values of the top-level elements in `list` to be inserted in the successively specified locations in the string `str`, according to the same conventions used by `format_string` as described in the last section. The list `list` is returned. Thus `format str list` prints the same as `print (format_string str list)`, except that the latter call will print a leading freshline and a trailing newline.

In addition to `print` and `format`, there is an error function that behaves very much like `format` except that execution of the program is interrupted, and the printed output is preceeded by an error flag. The form of the call is `error str list`.

## 10.3. File I/O and Terminal Input

A file may be opened for input in two ways:

- `char_in_stream file` will open the file identified by the string `file` and return a list (stream) that contains the successive ***characters*** (represented as single-length strings) found in the file. This list is lazily evaluated just as any other in ALFL; the successive elements are not evaluated (and are therefore not read from the file) until they are selected ("demanded") from the list. When the end of the file is reached (i.e., when the end of the stream is reached) then the file is automatically closed.
- `ob_in_stream file` is just like `char_in_stream` except that the elements in the returned stream are ***objects***, not characters; that is, numbers are parsed properly, lists are read intact, and successive words separated by blanks are returned as successive strings. For example, if the file `"foo"` looks like:

```
12.3 hello (1 harumph 2) there
```

then `ob_in_stream "foo"` returns the list `[12.3, "hello", [1, "harumph", 2], "there"]`. Note that lists as represented in the file have s-expression format -- this is because Alpha-Tau is built upon T. A future version of Alpha-Tau should fix this.

If the file specification in either of the above two function calls is the null string, then the *terminal* is opened for input. The end-of-file character is what will cause the terminal stream to be closed.

A stream (list) may be output to a file by the call `out_stream file list`, where `file` is the name (a string) of the file to be output to, and `list` is the stream of elements to be output. The value of the list is returned. For example, the call `out_stream "foo" [12.3, "hello", [1, "harumph", 2], "there"]` will create the file foo described earlier. The value `true` is returned from a successful call.

It should be noted that all output is done "eagerly"; that is, we assume that some daemon is demanding all of the elements of the list to be output. This is true even of terminal output.

It shouldn't be necessary, but if the user ever needs to explicitly close a file it may done via `close file`. One place this may be useful is at the ALFL read-eval-print-loop after an error has ocurred (see Section 12).

## 11. Miscellaneous Features

`hash ob` returns a unique hash-number (an integer) for the value of `ob`. `unhash int` has the property that `unhash hash:ob` ≡ `ob`.

## 12. The Alpha-Tau Implementation

The Alpha-Tau implementation of ALFL is built upon the Tau implementation of T [7]. It is available on Apollo Aegis, Vax Unix, and Vax VMS. The system takes advantage of T's functionality and lexical scoping by translating the ALFL source program into an equivalent T program. The resultant code (which is not fit for human consumption) may then be loaded into a special environment in which it is interpreted, or it may be compiled first and loaded into the same environment. Thus the user may mix compiled code (which may include libraries of commonly-used functions) with interpreted code (which is typically the program being developed). An interactive "read-eval-print-loop" using standard ALFL syntax interfaces to the ALFL environment to allow the convenience afforded by most Lisp systems. This section assumes some familiarity with T.

### 12.1. Starting Up Alpha-Tau

To run Alpha-Tau, simply start up T and load the file `alfl` from the appropriate directory:

```
(load "alfl")
```

Once Alpha-Tau is loaded, the REPL prompt will change to `ALFL>`. This is primarily a cosmetic change, since the REPL environment is still what it was at the time ALFL was loaded. However, a new environment has been created, `*ALFL-env*`, along with several functions that interface to it:

- `(alfl-parse "filename")` will parse and translate the file `filename`, writing the result to the file `filename.t`.
- `(alfl-load "filename")` will load the translated file `filename` into the ALFL environment, execute it, and print the result to the terminal.
- `(alfl "filename")` essentially has the effect of doing both of the above; the ALFL program found in the file `filename` is parsed, translated, loaded, and executed in the ALFL environment. The file `filename.t` containing the translation to T is written out as well.
- `(alfl)` will enter the user into a read-eval-print-loop in the ALFL environment. The session may be terminated with an end-of-file character, which on Aegis and VMS is control-Z, and on Unix is control-D.

## 12.2. More on the Interactive ALFL Environment

The interactive ALFL environment invoked by `(alfl)` is especially useful for program development, and requires further explanation. Once invoked, the user may type arbitrary expressions, including equation groups, and have their results computed immediately. For example:

```
{ fac 0 == 1;
   ' n == n*fac(n-1);
  result fac 5 };

    Result: 120
```

Note that it is necessary to delimit the end of the expression to be evaluated with a semicolon since, for example, the result of the expression may be a function which is to be applied to a yet-to-be-typed argument.

In addition, functions may be defined at the "top-level" of the ALFL read-eval-print-loop using the normal syntax for equations, except that (again for delimiting purposes) the set of equations must begin with the key-word `let` and end with a (extra) semicolon. For example:

```
let fac 0 == 1;
     ' n == n*fac(n-1);;

    Defining FAC
```

Note the extra semicolon after the last equation. The function `fac` is now defined (and may subsequently be redefined):

```
fac 5;

    Result: 120
```

There is one other useful feature of the ALFL interactive environment: If the "pragma" `$library` is placed at the beginning of a file containing an arbitrary number of equations, then those equations are simply translated and loaded into the ALFL environment, just as if they had been typed in using the `let` syntax just described. For example, if the file "`foo`" contains:

```
$library
fac 0 == 1;
  ' n == n*fac(n-1);
x == 5;
```

then after it is loaded into the ALFL environment by (`alfl "foo"`), `fac` and `x` will be defined just as if they were defined using the interactive `let` syntax.

If the user wishes to start a "new" ALFL environment -- that is, devoid of all definitions caused by the `let` or `$library` features -- then simply type (`alfl-reset`) to the `ALFL>` prompt.

## 12.3. Compiling ALFL Programs

As discussed earlier, an ALFL program in the file `filename` gets translated into a T program written out to the file `filename.t`. This program may then be loaded into the ALFL environment for execution (interpretation). Alternatively, one may compile the program first, using the optimizing T compiler (TC), and then load the resulting object file. As with most T programs, this can result in a speedup of ten-fold or more.

To compile the translated ALFL program, simply start up TC, and then load the support file `alfl_sup.t`. The file containing the translated ALFL program may then be compiled in the standard way. Once compiled, it may be loaded into the ALFL environment by using `alfl-load` to load the object file. `$library` files may be compiled in the same way.

## 12.4. Future additions to Alpha-Tau

1. Functional vectors and arrays (with an implementation technique that avoids the overhead of copying).
2. Fail semantics for equations; a primitive form of back-tracking (is this really necessary?).
3. More pragmas such as `$include` to automatically load libaray files, and `$expose_top_level` to load an ALFL program in such a way that the outermost definitions are exposed at the top-level of the ALFL environment (essentially "stripping away" the outermost curly-brackets).
4. A much fancier interactive environment that exploits the graphical capabilities of a bit-mapped display (this will most likely be a successor to Alpha-Tau; perhaps Beta-Tau?).
5. Conversion of normal-order to applicative-order evaluation where possible, as done in [4].

## 13. Acknowledgements

Many people have contributed to the development of ALFL and Alpha-Tau, either directly or indirectly. The work of David Turner and Robert Keller has of course been instrumental in formulating many of the ideas in the language. Alan Perlis provided implicit guidance and forewarnings in designing "yet another programming language." David Kranz helped me build the first implementation of ALFL, a combinator-based version described in [4]. Since that first implementation, several people have contributed to Alpha-Tau, especially Fred Douglis, Jonathan Young, and Lauren Smith. David McDonald helped me formulate ideas about how I/O should be done. Work is continuing with Jonathan Young, Lauren Smith, and Adrienne Bloss, and should account for further improvements to the language and implementation.

Special thanks are extended to Lauren Smith, who provided helpful comments on an earlier draft of this manuscript and helped formulate the concepts of ordered sets and ordered bags, including building an implementation of the latter in Alpha-Tau and giving assistance in writing Section 6.

# Appendix I

## Curried Equivalences of Infix Operators

| | | | |
|---|---|---|---|
| ^ | fby | - (unary) | negate |
| ^^ | append | - (binary) | sub |
| : | apply | + (unary) | plus (not implemented) |
| . | compose | + (binary) | add |
| = | equal | * | mult |
| <> | notequal | / | div |
| > | gt | \ | rem |
| < | lt | & | and |
| >= | ge | \| | or |
| <= | le | ~ | not |
| // | reduce | | |
| \\ | layered_apply | | |
| \|\| | nested_map | | |
| :: | structured_apply | | |

## Operator Precedence

```
 // \\ || ::
       .              (function composition)
       :
    <space>           (function application)
      + -             (the unary  versions)
     * / \
      + -             (the binary versions)
= < > >= <= <>
      -
      &
      |
      ^^
      ^
      -> ,            (conditional)
     @ ==             (infix lambda)
```

# Appendix II
# Syntax of ALFL

In the following, literal characters (in the terminal alphabet) are enclosed by quotes. Token names are enclosed in angle brackets ("<...>"). Parentheses are used to group a set of terminals and/or non-terminals together. An object or group of objects followed by a * means zero or more occurrences of that object; use of a + denotes one or more occurrences. A vertical bar denotes alteration. Finally, @ denotes the empty string.

## Lexical Definitions

Comments are represented by a percent sign ( "%" ) followed by the comment text followed by an end-of-line character (carriage return or line-feed). Comments are treated as "white-space" and thus do not qualify as a token.

In general upper- and lower-case are equivalent. The only exception to this is that character case is preserved in strings.

The lexer recognizes the following tokens:

There is one reserved word; viz: result.

Some are infix or prefix operators, like:

```
-> // \\ || :: | & ~ + - * / \ = > < >= <= <> ^^ ^ : . <-
```

and some are delimiters, such as:

```
! { } [ ] ( ) ; , == {* *} [* *] @
```

There are also three "structured" tokens:

```
        <number> --> <integer> | <float>
            <id> --> <letter><alpha>*
        <string> --> """(<any-char> | <blank>)*"""
```

where:

```
      <integer> --> <digit>+
        <float> --> <digit>*.<digit>+ | <digit>+.<digit>*
       <digit>  --> "0" | "1" | ... | "9"
      <letter>  --> "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
       <alpha>  --> <letter> | <digit> | "_"
    <any-char>  --> <alpha> | <special-symbol> | """"  (two double-quotes,
                                                        interpreted as one)
<special-symbol> --> any other printable character except "
```

Examples of single tokens:

```
        this_is_right     27938     .5     5.     0     foo
        "He said, ""foo on you!"""     >=     result     27.938
```

Non-examples of single tokens:

```
this-is-wrong    "this isn"t right"    9result    1.2.3
```

Tokens are separated by "white-space"; that is, anything that appropriately disambiguates successive tokens. For example, 9result is really two tokens, 9 and result, separated by the white space @ (the symbol for epsilon, the null string). Of course, the null string is not always sufficient.

### A Context-Free Grammar for ALFL

In the following productions, non-terminals are represented by an identifier reflecting its intuitive meaning (for example, "expression"). Terminals are tokens as defined above, and are represented in one of three ways; structured tokens are enclosed in angle brackets (as in <id>), reserved words are represented by all caps (as in RESULT), and operators and delimiters are enclosed in quotes (as in ">=").

```
       program --> equation-group
equation-group --> "{" ( equation ";" )*
                       RESULT expression
                       ( ";" equation )*
                       ( @ | ";" )
                   "}"
      equation --> lhs "==" expression
           lhs --> (<id> | "'") patterns | pattern
      patterns --> pattern patterns | pattern
       pattern --> constant | "#" e15 | <id> | "(" pat-ex ")"
                   "[" pat-ex ( "," pat-ex )*  "]"
        pat-ex --> pattern | pat-ex "^" pat-ex
    expression --> "@" patterns "==" e0 | e0
            e0 --> e1 "->" e0 "," e0 | e1 "->" e0 | e1
            e1 --> e2 "^"  e1 | e2
            e2 --> e3 "^^" e2 | e3
            e3 --> e4 "|"  e3 | e4
            e4 --> e5 "&"  e4 | e4
            e5 -->    "~"  e5 | e6
            e6 --> e7 ( "=" | "<" | ">" | "<=" | ">=" | "<>" ) e7 | e7
            e7 --> e8 ( "+" | "-" ) e7 | e8
            e8 --> e9 ( "*" | "/" | "\" ) e8 | e9
            e9 --> ( "+" | "-" ) e10 | e10
           e10 --> e11 e10 | e11
           e11 --> e12 ":" e11 | e12
           e12 --> e13 "." e12 | e13
           e13 --> e14 ( "::" | "||" | "\\" | "//" ) e13 | e14
           e14 --> "[" ( expression "," )* expression "]" | e15
           e15 --> <id> | constant | equation-group |
                   "(" expression ")" | setnot
      constant --> <number> | <string> | "[]"
```

```
    setnot --> [* expression ! quals *] | {* expression ! quals *}
    quals  --> gens (@ | "!" fils)
    gens   --> generator (";" generator)* (@ | ";")
    fils   --> filter  (";" filter)* (@ | ";")
    filter --> expression
 generator --> pat-ex "<-" expression
```

All infix operators associate to the right, except for "blank" (function application), which associates to the left. The "dangling else" is disambiguated in the normal way. Operator precedence is as follows, from highest to lowest:

```
// \\ || ::
      .                (function composition)
      :
   <space>             (function application)
     + -               (the unary  versions)
    * / \
     + -               (the binary versions)
= < > >= <= <>
     -
     &
     |
     ^^
     ^
     -> ,              (conditional)
    @ ==               (infix lambda)
```

# References

[1] Burstall, R.M., MacQueen, D.B., and Sannella, D.T.
HOPE: An experimental Applicative Language.
In Davis, R.E., and Allen, J.R. (editors), *The 1980 LISP Conference*, pages 136-143. Stanford University, August 1980.

[2] Darlington, J., Henderson, P., and Turner, D.A. (editor).
*Functional Programming and its Applications.*
Cambridge University Press, Cambridge, England, 1982.

[3] Henderson, P.
*Functional Programming: Application and Implementation.*
Prentice-Hall, Englewood Cliffs, NJ, 1980.

[4] Hudak, P. and Kranz, D.
A combinator-based compiler for a functional language.
In *11th ACM Sym. on Prin. of Prog. Lang.*, pages 121-132. acm, January 1984.

[5] Keller, R.M.
FEL programmer's guide.
AMPS TR 7, University of Utah, March 1982.

[6] Rees, J.A., and Adams, N.I.
T: a dialect of LISP or, Lambda: the ultimate software tool.
In Park et al. (editors), *Sym. on Lisp and Functional Prog.*, pages 114-122. ACM, August 1982.

[7] Rees, J.A., Adams, N.I., and Meehan, J.R.
The T Manual.
Technical Report 4th edition, Yale University, January 1984.

[8] Turner, D.A.
SASL language manual.
Technical Report, University of St. Andrews, 1976.

# Index