The **fab** Programming Language Reference Manual

© 2010 Andrew Tolmach
Dept. of Computer Science
Portland State University

(version of November 26, 2010)

1 Introduction

The **fab** language is a small imperative programming language with first class functions, extensible record values with implicit pointers, arrays, integer and real variables, and a few simple structured control constructs. It bears many similarities to an earlier language, PCAT, designed by Andrew Tolmach and Jingke Li.

This manual gives an informal specification for the language. Fragments of EBNF syntax are introduced at relevant points in the text; the complete grammar is given in Section 13.

2 Lexical Issues

fab's character set is the standard 7-bit ASCII set. **fab** is case sensitive; upper and lower-case letters are *not* considered equivalent.

Whitespace (blank, tab or newline characters) serves to separate tokens; otherwise it is ignored. Whitespace is needed between two adjacent keywords or identifiers, or between a keyword or identifier and a number. However, no whitespace is required between a number and a keyword, since this causes no ambiguity. Delimiters and operators don't need whitespace to separate them from their neighbors on either side. Whitespace may not appear in any token except a string (see below).

Comments are enclosed in the pair `[*` and `*]`; they cannot be nested. Any character is legal in a comment. Of course, the first occurrence of the sequence of characters `*]` will terminate the comment. Comments may appear anywhere a token may appear; they are self-delimiting, i.e., they do not need to be separated from their surroundings by whitespace.

2.1 Tokens

Tokens consist of keywords, literal constants, identifiers, operators, and delimiters.

The following are reserved *keywords*.

```
and       by        div       do        else      elsif
exit      extends   for       func      if        loop
mod       not       of        or        read      record
return    then      to        var       while     write
```

Literal constants are either integer, real, or string. *Integers* (denoted `INTEGER` in the grammar) contain only digits; they must be in the range 0 to $2^{31} - 1$. *Reals* (denoted `REAL` in the grammar) consist of one or more digits, followed by a decimal point, followed by zero or more digits. There is no specific range constraint on reals, but the literal as a whole is limited to 255 characters in length. Note that no numeric literal can be negative, since there is no provision for a minus sign. *Strings* (denoted `STRING` in the grammar) begin and end with a double quote (`"`) and contain any sequence of printable ASCII characters (i.e., having decimal character codes in the range 32 – 126) except double

quote. Note in particular that strings may not contain tabs or newlines. String literals are limited to 255 characters in length, not including the delimiting double quotes.

Identifiers (denoted `ID` in the grammar) are strings of letters and digits starting with a letter, excluding the reserved keywords. Identifiers are limited to 255 characters in length.

The following are the *operators*:

```
@   ->  :=  +   -   *   /   <   <=  >   >=  =   <>
```

and the *delimiters*:

```
:   ;   ,   .   (   )   [   ]   {   }
```

For clarity, these are written within single quotes in the grammar.

3 Programs

program → {*recordtype-decl*} *block*

A program is the unit of compilation for **fab**. It consists of (optional) record type declarations and a *top-level block*. It is executed by executing its top-level block and then terminating.

Each file read by the compiler must consist of exactly one program. There is no facility for linking multiple programs or for separate compilation of parts of a program.

4 Blocks

block → *'{' block-items '}'*
block-items → [*block-item* {*';' block-item*}]
block-item → *declaration* | *statement*

A block is a sequence of declarations and statements, which may be freely intermixed. It is executed by elaborating each declaration and executing each statement in order.

5 Declarations

All identifiers occurring in a program must be introduced by a declaration, except for a small set of pre-defined identifiers: `real`, `integer`, `boolean`, `unit`, `true`, `false` (see Section 6.2), and `nil` (see Section 6.5). Each declaration serves to specify whether the declared identifier represents a type, a variable, or a function (all of which live in a single *name space*) or a record component name (which live in separate name spaces; see Section 6.5).

The only kind of types that can be declared are record types. Record type declarations occur at the beginning of the program, are mutually recursive, and are in scope throughout the program. Description of their syntax is defered to Section 6.5.

Variable and function declarations are local to a block and all its sub-blocks. Global declarations are simply those that appear in the top-level block.

declaration → *var-decl* | *funcs-decl*

The *scope* of a declaration extends roughly from the point of declaration to the end of the enclosing block. The exact scope rules depend on the kind of declaration (see Sections 8 and 9). A declaration of an identifier within a nested function *hides* any declarations in outer functions and makes them inaccessible in the scope of the inner declaration. No identifier may be declared twice in the same function. The built-in identifiers may not be redeclared anywhere in the program. An identifier declared as a record type name may not be redeclared anywhere in the program.

Declaration elaboration can have computational side-effects, so the order of declarations and statements matters even when scope is not an issue.

6 Types

fab is a strongly-typed language; every expression has a unique type, and types must match at assignments, calls, etc. There is a simple notion of subtyping based on records extensions (see Section 6.5) and for the numeric types (see Section 6.2).

The built-in *basic types* (see Section 6.2) and declared record types are refered to by *type names*. New record types are created by *record type declarations* (see Section 6.5). Function and array types cannot be named; they are always created "on the fly" by applying the type constructors `->` and `@`, respectively, to existing types.

fab uses a *mixed* equivalence model for types. For records, *name* equivalence is used: each record type declaration produces a new, unique type, incompatible with all the others (except possibly for subtyping). For functions and array, *structural* equivalence is used: two types are equivalent if they result by applying the same constructor to equivalent types (again, with possible subtyping).

6.1 Type Expressions

type-expr	→	`ID`
	→	'`@`' *type-expr*
	→	*type-args* '`->`' *type-expr*
	→	'`(`' *type-expr* '`)`'
type-args	→	'`(` `)`'
	→	*type-expr*
	→	'`(`' *type-expr* {'`,`' *type-expr*} '`)`'

The function type constructor (`->`) is right-associative and has lower precedence than the array constructor (`@`). Parentheses may be used in type expressions to alter associativity or to improve readability.

6.2 Built-in Types

There are four *built-in* basic types: `integer`, `real`, `boolean`, and `unit`. Integer literal constants all have type `integer`, real literal constants all have type `real`, and the built-in values `true` and `false` have type `boolean`. The type `unit` is used to specify the return type of functions that don't return a useful value (similar to `void` in C/C++/Java); its (sole) value cannot be denoted in programs.

`integer` and `real` collectively form the *numeric* types. An `integer` value will always be implicitly *coerced* to a `real` value if necessary. The boolean type has no relation to the numeric types, and a boolean value cannot be converted to or from a numeric value.

6.3 Array Types

An array is a structure consisting of zero or more elements of the same *element type*. An array type is written as `@` followed by an expression for the element type. The elements of an array can be accessed by *dereferencing* using an *index*, which ranges from 0 to the length of the array minus 1. The length of an array is not fixed by its type, but is determined when the array is created at runtime. It is a checked runtime error to dereference outside the bounds of an array.

6.4 Function Types

Function types, written with an `->` constructor, describe functions taking zero or more parameters and returning a result (possibly `unit`). Normally, the parameter types are written as a comma-separated list within parentheses; the parentheses may be omitted when there is exactly one parameter. The body of a function is a block.

Functions are fully first-class in **fab**; that is, in addition to being called, they can also be stored in variables, arrays or records, or returned as the result of other functions. Function declarations can appear at any level of block nesting, and their bodies can freely dereference local variables and functions of enclosing blocks. However, the body of a nested function can only *update* variables declared within the function's own block or in the top-level block.

6.5 Record Types

A record type is a structure consisting of a fixed number of *components* of (possibly) different types. Unlike the other types, record types must be declared by name before they can be used.

recordtype-decl → `record ID` [`extends ID`] '`{`' [*ids-and-types*] '`}`' '`;`'
ids-and-types → *id-and-type* {'`,`' *id-and-type*}
id-and-type → `ID` '`:`' *type-expr*

The record type declaration specifies the name of the record type, an optional named super-type that this record type extends, and the name and type of each component. Component names are used to initialize and dereference components; the components for each record type form a separate namespace, so different record types may reuse the same component names. Component names must be unique within each record type.

The special built-in value `nil` belongs to every record type. It is a checked runtime error to dereference a component from the nil record.

All record types are potentially mutually recursive; that is, all record type names are in scope to define all components of all records. Note the utility of the `nil` record for building values of recursive types.

A record type declaration can optionally *extend* another record type, called its *super-type*. In this case, the present type contains all the components of the super-type *in addition to* the components listed. If the super-type is itself an extension, *its* super-type components are included as well, and so forth. Component names must be unique across the entire set of components. For example, given the declarations

```
record T{a:int}
record U extends T {b:bool}
```

records of type `T` contain a single component `a:int` and records of type `U` contain two components `a:int` and `b:bool`. No chain of super-types may contain a cycle.

An extended type is automatically coerced to its super-type (or the super-type of its super-type, and so on) whenever this is demanded by the context in which it is used. This makes sense because the context that wants a super-type can just ignore the additional fields defined by the extension. But this coercion only works when one record type is explicitly declared as the extension of another, not merely when one type has a superset of another's components. Given the declaration

```
record W {a:int,b:bool}
```

records of `W` contain the same components as those of type `U`, but they are not automically coerced to type `T`.

6.6 Subtyping

Type equivalence checks are performed "up to subtyping" so that a subtype can always be used in place of a supertype. Subtyping is transitive: if t is a subtype of u and u is a subtype of w, then t is a subtype of w.

The two basic forms of subtyping have already been described: integers can be treated as reals, and the value of an extended record type can be treated as having its super-type. Record subtyping is *nominal*: one record type is a subtype of another only when it is explicitly declared as an extension.

For functions, **fab** uses a *structural* subtyping rule, as follows: a function type `(`t_1`,...,`t_n`) -> t` is a subtype of `(`u_1`,...,`u_n`) -> u` if u is a subtype of t and t_i is a subtype of u_i for each $i \leq n$. Note that the subtyping relation is *covariant* on the result type but *contravariant* on the parameter types.

Each array types has only itself as a (trivial) subtype. So the array type `@`t is a subtype of `@`u only if $t = u$; in particular, it is not sufficient for t to be a subtype (or supertype) of u.

6.7 Constructed Type Values

Arrays and records are always manipulated by value, so a value of an array or record type is "really" a pointer to a heap object containing the array or record, though this pointer cannot be directly manipulated by the programmer. Thus, a record type that appears to contain other record types as components actually contains pointers to these types. In particular, a record type may contain (a pointer to) itself as a component, i.e., it may be recursive.

Similarly, values of function types are also represented as heap objects, called *closures* (see Section 9).

Records, arrays, and closures have unlimited lifetimes; the heap object containing one of them exists from the moment when its defining expression or declaration is evaluated (see Sections 11.6 11.7, and 9) until the end of the program. In principle, a garbage collector could be used to remove heap objects when no more pointers to them exist, but this would be invisible to the **fab** programmer.

7 Constants

There are three *built-in constant* values: `true` and `false` of type `boolean`, and `nil`, which belongs to every record type. There is no provision for user-defined constants.

8 Variables

var-decl → `var` `ID` [`':'` *type-expr*] `':='` *expression*

Every variable must have an initial value, given by `expression`. The type declaration can be omitted whenever the type can be deduced from the initial value (which is always possible except when the initial value is `nil`).

A `var` declaration is elaborated by evaluating the initializing expression and storing the resulting value into the specified variable.

The scope of each declared variable begins just after the declaration; it does *not* include the variable's own initializing expression, so declarations are never recursive.

9 Functions

funcs-decl → `func` *func-decl* {`and` *func-decl*}
func-decl → `ID` `'('` [*ids-and-types*] `')'` [`'->'` *type-expr*] *block*
ids-and-types → *id-and-type* {`','` *id-and-type*}
id-and-type → `ID` `':'` *type-expr*

Functions may or may not explicitly return a value. If they do not, they are considered to have return type `unit`; in this case, the return type can be omitted altogether from the function declaration. Functions returning `unit` can only be a invoked by the execution of a call *statement*; those that return a non-`unit` value can only be invoked by evaluating a call *expression* and their return value becomes the value of the call expression.

A function have zero or more *formal parameters*, whose names and types are specified in the function declaration, and whose actual values are specified when the function is activated. The scope of formal parameters is the function block. All parameter names must be distinct. Parameters are always passed by value.

The body of a function is a block. A function is activated by binding the formal parameters to the actual argument values, executing the function's defining block, and finally returning to the calling function. There is an implicit `return` statement at the bottom of every function body.

Each set of functions declared following a single `func` keyword (and separated by `and` keywords) is treated as (potentially) mutually recursive; that is, the scope of each function name begins at the point of declaration of the first function in the set, and includes the bodies of all the functions in the set as well as the remainder of the enclosing block.

Elaboration of a function declaration causes creation of a *closure* object on the heap, which contains a pointer to the function's code together with the *current values* of any variables dereferenced in the function body that are declared in outer enclosing blocks other than the top level. Storing the value of each non-local variable in the closure allows a function body to access it even after the activation of the outer block where it was declared has terminated. However, because the values in the closure are only copies, it does not make sense to update them, so this is prohibited. The need to store values in closures and the restriction on update do not apply to top-level variables, since these are active for the entire duration of the program.

A **fab** implementation may perform optimizations to avoid constructing closure objects in some circumstances, but such optimizations will not be visible to the **fab** programmer.

10 L-values

An *l-value* is a location whose value can be either read or assigned. Variables, function parameters, record components, and array elements are all l-values.

lvalue → `ID`
→ *lvalue* '`[`' *expression* '`]`'
→ *lvalue* '`.`' `ID`

The square brackets notation (`[]`) denotes array element dereferencing; the expression within the brackets must evaluate to an integer expression within the bounds of the array.

The dot notation (`.`) denotes record component dereferencing; the identifier after the dot must be a component name within the record.

11 Expressions

11.1 Simple expressions

expression → *number*
→ *lvalue*
→ '`(`' *expression* '`)`'
number → `INTEGER` | `REAL`

A number expression evaluates to the literal value specified. Note that reals are distinguished from integers by lexical criteria (see Section 2). An l-value expression evaluates to the current contents of the specified location. Parentheses can be used to alter precedence in the usual way.

11.2 Arithmetic operators

expression → *unary-op expression*
→ *expression binary-op expression*
unary-op → '`-`'
binary-op → '`+`' | '`-`' | '`*`' | '`/`' | `div` | `mod`

Operators `+`, `-`, `*` require integer or real arguments. If both arguments are integers, an integer operation is performed and the integer result is returned; otherwise, any integer arguments are coerced to reals, a real operation is performed, and the real result is returned. Operator `/` requires integer or real arguments, coerces any integer arguments to reals, performs a real division, and always returns a real result. Operators `div` (integer quotient) and `mod` (integer remainder) take integer arguments and return an integer result. All the binary operators evaluate their left argument first.

11.3 Logical operators

expression → *unary-op expression*
→ *expression binary-op expression*
unary-op → `not`
binary-op → `or` | `and`

These operators require boolean operands and return a boolean result. `or` and `and` are "short-circuit" operators; they do not evaluate the right-hand operand if the result is determined by the left-hand one.

11.4 Relational operators

expression → *expression binary-op expression*
binary-op → '`>`' | '`<`' | '`=`' | '`>=`' | '`<=`' | '`<>`'

These operators all return a boolean result. These operators all work on numeric arguments; if both arguments are integer, an integer comparison is made; otherwise, any integer argument is coerced to real and a real comparison

is made. Operators = and <> also work on pairs of boolean arguments, or pairs of record or array arguments of the same type; for the latter, they test "pointer" equality (that is, whether two records or arrays are the same instance, not whether they have the same contents). These operators all evaluate their left argument first.

11.5 Function call

expression → *expression* '`(`' *expressions* '`)`'
expressions → [*expression* {'`,`' *expression*}]

This expression is evaluated by evaluating the operator expression to obtain the function closure value, and then the argument expressions left-to-right to obtain actual parameter values, and finally executing the function with its formal parameters bound to the actual parameter values. The function returns by executing an explicit `return` statement (with an expression for the value to be returned), so the return type must not be `unit`. The returned value becomes the value of the function call expression.

11.6 Record construction

expression → `ID` '`{`' [*comp-inits*] '`}`'
comp-inits → *comp-init* {'`,`' *comp-init*}
comp-init → `ID` '`:=`' *expression*

If *typename* is a record type name, then *typename* $\{id_1$`:=`exp_1`,`id_2`:=`exp_2`,`$\ldots\}$ evaluates each expression left-to-right, and then creates a new record instance of type *typename* with named components initialized to the resulting values. The names and types of the component initializers must match those of the named type (including any components inherited from a super-type), though they need not be in the same order.

11.7 Array construction

expression → '`@`' *type-expr* '`{`' [*array-inits*] '`}`'
array-inits → *array-init* { '`,`' *array-init*}
array-init → [*expression* `of`] *expression*

The expression `@`*texpr* $\{expr_1^n$ `of` $expr_1^v$`,` $expr_2^n$ `of` $expr_2^v$`,`$\ldots\}$ evaluates each pair of expressions in left-to-right order to yield a list of pairs of integer counts n_i and initial values v_i, and then creates a new array instance with elements of type *texpr* whose contents consist of n_1 copies of v_1, followed by n_2 copies of v_2, etc. If any of the counts is 1, it may be omitted. For example, the specification `@integer{1,2 of 3,3 of 2,4}` yields an integer array of length 7 with contents `1,3,3,2,2,2,4`. If any of the n_i is less than 1, no copies of the corresponding v_i are included. The types of the v_i must match *texpr*.

11.8 Precedence and associativity

Function call and parenthesization have the highest (most binding) precedence; followed by unary `-`; followed by `*`, `/`, `mod`, and `div`; followed by `+` and `-`; followed by the relational operators; followed by `not`; followed by `and`; followed by `or`.

The binary arithmetic and logical operators are all left-associative. The relational operators are non-associative; in other words, an expression such as `a = b = c` is illegal, although one such as `(a = b) = c` is legal (presuming `c` has type `boolean`.

12 Statements

12.1 Block

statement → *block*

A block may be introduced at any point where a statement is expected.

12.2 Assignment

statement → *lvalue* `':='` *expression*

The l-value is evaluated to a location; then the expression is evaluated and its value is stored in the location. Assigning a record or array value actually assigns a pointer to the record or array.

12.3 Function Call

statement → *expression* `'('` *expressions* `')'`
expressions → [*expression* {`','` *expression*}]

This statement is executed by evaluating the operator expression to obtain the function closure value, and then the argument expressions left-to-right to obtain actual parameter values, and finally executing the function with its formal parameters bound to the actual parameter values. The function must have return type `unit`. The function returns when its final statement or an explicit `return` statement (with no expression) is executed.

12.4 Read

statement → `read` `'('` *lvalue* {`','` *lvalue*} `')'`

This statement is executed by evaluating the l-values to locations in left-to-right order, and then reading numeric literals from standard input, evaluating them, and assigning the resulting values into the locations. The l-values must have type integer or real, and their types guide the evaluation of the corresponding literals. Input literals are delimited by whitespace, and the last one must be followed by a carriage return.

12.5 Write

statement → `write` `'('` *write-params* `')'`
write-params → [*write-expr* {`','` *write-expr*}]
write-expr → `STRING` | *expression*

Executing this statement evaluates the specified expressions (which must evaluate to integers, reals, booleans, or string literals) in left-to-right order, and then writes the resulting values to standard output (with no separation between values), followed by a newline.

12.6 If-then-else

statement → `if` *expression* `then` *statement*
{`elsif` *expression* `then` *statement*}
[`else` *statement*]

This statement specifies the conditional execution of guarded statements. The expression preceding a statement sequence, which must evaluate to a boolean, is called its *guard*. The guards are evaluated in left-to-right order, until one evaluates to `true`, after which its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the `else` (if any) is executed.

In cases of ambiguity, an `else` or `elsif` is always attached to the nearest previous `if` statement.

12.7 While

statement → `while` *expression* `do` *statement*

The inner statement is repeatedly executed as long as the expression evaluates to `true`, or until an `exit` from the loop (see Section 12.10).

12.8 Loop

statement → `loop` *statement*

The inner statement is repeatedly executed until an `exit` occurs (see Section 12.10).

12.9 For

statement → `for ID ':='` *expression* `to` *expression* [`by` *expression*]
`do` *statement*

Executing the statement `for` id `:=` exp_1 `to` exp_2 `by` exp_3 `do` st is equivalent to the following steps: (i) evaluate expressions exp_1, exp_2, and exp_3 in that order to values v_1, v_2, v_3 (which must be integers); (ii) if the value of id is less than or equal to v_2, execute st; otherwise terminate the loop. (iii) set id := id + v_3 and repeat step (ii).

If the `by` clause is omitted, v_3 is taken to be 1.

The loop index id is an ordinary integer variable; it must be declared in the scope containing the `'for'` statement, and it can be inspected or set above, within, or below the loop body.

The normal execution of the loop can be interrupted early by an `exit` statement (see Section 12.10).

12.10 Exit

statement → `exit`

Executing `exit` causes control to pass immediately to the next statement following the *nearest* enclosing `while`, `loop` or `for` statement. If there is no such enclosing statement, the `exit` is illegal.

12.11 Return

statement → `return` [*expression*]

Executing `return` terminates execution of the current function and returns control to the calling context. There can be multiple `return`s within one function body, and there is an implicit `return` at the bottom of every function. A `return` from a function with return type other than `unit` must have a return value expression of the return type; a `return` from a function with return type `unit` must not. The top-level program block must not include a `return`.

13 Complete Concrete Syntax

program → {*recordtype-decl*} *block*
recordtype-decl → `record ID` [`extends ID`] '{' [*ids-and-types*] '}' ';'
ids-and-types → *id-and-type* {',' *id-and-type*}
id-and-type → `ID` ':' *type-expr*
block → '{' *block-items* '}'
block-items → [*block-item* {';' *block-item*}]
block-item → *declaration* | *statement*
declaration → *var-decl* | *funcs-decl*
var-decl → `var ID` [':' *type-expr*] ':=' *expression*
funcs-decl → `func` *func-decl* {`and` *func-decl*}
func-decl → `ID` '(' [*ids-and-types*] ')' ['->' *type-expr*] *block*
type-expr → `ID`
→ '@' *type-expr*
→ *type-args* '->' *type-expr*
→ '(' *type-expr* ')'
type-args → '(' ')'
→ *type-expr*
→ '(' *type-expr* {',' *type-expr*} ')'
statement → *lvalue* ':=' *expression*
→ *expression* '(' *expressions* ')'
→ `read` '(' *lvalue* {',' *lvalue*} ')'
→ `write` '(' *write-params* ')'
→ `if` *expression* `then` *statement*
{`elsif` *expression* `then` *statement*}
[`else` *statement*]
→ `while` *expression* `do` *statement*
→ `loop` *statement*
→ `for ID` ':=' *expression* `to` *expression* [`by` *expression*]
`do` *statement*
→ `exit`
→ `return` [*expression*]
→ *block*
write-params → [*write-expr* {',' *write-expr*}]
write-expr → `STRING` | *expression*
expression → *number*
→ *lvalue*
→ '(' *expression* ')'
→ *unary-op expression*
→ *expression binary-op expression*
→ *expression* '(' *expressions* ')'
→ `ID` '{' [*comp-inits*] '}'
→ '@' *type-expr* '{' [*array-inits*] '}'
expressions → [*expression* {',' *expression*}]
lvalue → `ID`
→ *lvalue* '[' *expression* ']'
→ *lvalue* '.' `ID`
comp-inits → *comp-init* {',' *comp-init*}
comp-init → `ID` ':=' *expression*
array-inits → *array-init* { ',' *array-init*}
array-init → [*expression* `of`] *expression*
number → `INTEGER` | `REAL`
unary-op → '-' | `not`
binary-op → '+' | '-' | '*' | '/' | `div` | `mod` | `or` | `and`
→ '>' | '<' | '=' | '>=' | '<=' | '<>'