Manpage of FUN

Content-type: text/html

FUN

Section: Misc. Reference Manual Pages (1es)
Index Return to Main Contents

NAME

fun - Experimental Functional Programming Language (version 1.3, Jan 1996)

SYNOPSIS

fun [-dhpvw] [files [-]]

DESCRIPTION

fun is the name of a simple functional programming language with fully lazy semantics and also the name of the program that reads a piece of source text in the language and evaluates it. The language is based on lambda-expressions, although thanks to the available extensive syntax most of the peculiarities of lambda-expressions are hidden from the user. *fun* accepts zero or more file name arguments optionally followed by a dash (-) to indicate that after processing all files, input is read from standard input: stdin. If no file arguments are present, input is read from stdin regardless of an explicit dash at the end of the command line.

fun mimics the functionality found in mature functional programming languages like Miranda (TM), Haskell, and Gofer. Don't get confused: all built-in functions in *fun* are defined as prefix operators. *fun* does however support a crude facility for infix expressions: any prefix operator or user defined function may be enclosed in back-quotes (‘) and as such used inbetween operands. Instead of back-quotes, a single dollar-sign in front of the operator name may be used as an alternative. All such infix operators, unlike the built-in ones, will assume the same (lowest) precedence and group from left-to-right (= left-associative), unless, of course, this default behaviour is explicitly overridden with the use of parentheses. Here is an example of using built-in operators in an infix expression: 123 $+ 4 $* 2. This gets interpreted as (123 $+ 4) $* 2, and thus is equivalent to (* (+ 123 4) 2).

Missing in *fun* are function definitions by pattern-matching and any notion of modularisation, like modules or abstract data types. Also, *fun* is weakly typed and type-checking is done at run-time, e.g. lists may contain elements of any type. In contrast, Miranda requires lists to have elements of a single type, and supplies the fixed-sized tuples data type to handle record/struct-like data. For now, tuples may be emulated by lists and uninterpreted names can be used as tags.

OPTIONS

General options:

-d

prints debug info to stderr; implies -v. This is of no interest to a regular user.

-h

prints a brief description of each option to stderr and exits.

-v

verbose, prints action summary to stderr.

-w

suppresses warning messages.

fun program specific options:

-p

Useful for running fun scripts. For that purpose *fun* interprets the combination of characters ‘#!’ at the beginning of a line as the start of a comment line. These comments should not be used for any other purpose; Use the regular C-style comment delimiters instead, like /* this is a comment */. To build a script, simply mention ‘#! <path>fun -p’ on the first line of a *fun* input source file.

<path> stands for the Unix directory of the *fun* program executable. Make the script executable, and you can then run it as a stand-alone (batch) program.

INTRODUCTION TO LAMBDA-CALCULUS AND FUN

The *fun* program is in a highly experimental stage and mainly intended as a compact and simple test bench for functional programming, and therefore the following is undoubtedly subject to change sooner than later. *fun* reads, compiles, and then reduces so-called extended lambda-expressions. There are a number of built-in types and functions, and also a standard library (in source code format) of some useful functions. This library is automatically loaded at program start-up. The location of the file is determined by the installer of the program and should be of no concern to a user. Unless, of course, you want to bypass the standard definitions in order to include your own. For that purpose you can define the FUN_STDDEFS environment variable to whatever file you like to get loaded instead, or even leave it empty which simply causes no load at all. You may want to initially load other files as well. This is accomplished by having a .fun file in your home directory which gets loaded right after the standard definitions file. There is also a top-level 'load' command that may be used to explicitly load fun source files. Loads may be nested; see below.

Expressions can be applications, abstractions, compositions, and atoms. In interactive mode you simply type an expression followed by a ';'. It gets compiled and evaluated, and its result is output to stdout. There are a few top-level commands that allow expressions to be named and stored under that name for later access in other expressions. The following constructs are recognized:

let Defs

See with letrec below.

letrec Defs

These lets and letrecs introduce so-called global or top-level definitions. The (partial) syntax of 'Defs' is:

Defs : { FunName [FormalArgs] = Expression / ',' }+

where FunName is some identifier which subsequently will serve as a name for the expression. letrec allows for recursive function definitions. There are also local let and letrec statements. For precise details see the file doc/syntax. The semantics of globally defined names is that they are globally visible from their point of definition onwards. However, globally defined names may become obscured by local definitions of the same FunName. Any free variables appearing in the global definition's body will remain free forever. It therefore doesn't make much sense to redefine a global name when it's also used in previous definitions; the latter will still see the old definition. To define functions recursively you

must use letrec. For instance the following definition will still have fac as a free variable in its body (right side of the = symbol):

```
let fac = L n . if (<= n 0) 1 (* n (fac (- n 1)));
```

Replacing let by letrec defines fac as the familiar factorial function. In order to define mutual recursive functions you must use letrec in the following way:

```
letrec f = ... g ... ,
       g = ... f ... ;
```

Two separate letrecs will not work. For global lets, the following forms are equivalent:

```
let f = ..., g = ...;  == let f = ...; let g = ...;
```

There are several ways to specify the formal parameters or better arguments for a function. In the above examples no FormalArgs were present, still the functions did have arguments as indicated by the explicit abstraction in the 'let f = L n . E' syntax. The latter is equivalent to 'let f n = E'. In general we have:

```
f a b c ... = E  ==  f = L a b c ... . E
```

The following syntax is also allowed:

```
f (a b c ...) = E, and also f (a, b, c, ...) = E
```

Choose the one you like best. In serious applications we recommend to only use the Miranda style syntax, 'f a b c ... = E', i.e., formal arguments to functions directly follow the function name and are separated by blanks.

Instead of using local lets and letrecs in the form 'let v = B in E', you may also use 'where-clauses'. They appear right after the definition in a let or letrec, for instance as in 'let v = B where w = C end in E'. Use of lets and letrecs imposes a define-before-use style of programming, whereas with where-clauses you first introduce a name and afterwards define it. Note that the 'where' must be followed by a matching 'end' keyword. This is not the case in Miranda, because Miranda uses a layout based syntax much like for instance OCCAM. More formally, the full syntax of Defs and the syntax of a WhereClause are:

```
Defs : { Def / ',' }+ .

Def : FunName [ FormalArgs ] = Expr [ WhereClause ] .

WhereClause : 'where' Defs 'end' .
```

The Defs within the where-clause are considered potentially mutually recursive. For efficiency, the Defs are analysed to obtain minimal sets of recursive definitions and thus maximal non-recursive ones. Internally, the recursive ones will be turned into appropriate letrecs, and the non-recursive definitions are cast into simple lets. This also holds for letrecs: if possible they are turned into simple lets. As a user you therefore need not worry about any efficiency issues in these cases.

undef FunName

Often, global definitions need to be modified (especially in interactive mode where you likely will program in a trial-and-error kind of style). Redefining an existing FunName is allowed but will cause a warning message. You may of course switch off the generation of warnings by the '-w' option. Perhaps it's more secure to explicitly undefine the function before redefining it. In that case use 'undef FunName'. Note that when globally defined names are used in other definitions, redefining that name does not automatically get propagated to all its points of usage. This is so because global definitions get compiled into other definitions and as such no knowledge is kept about the fact a global name was used. To resolve this, the only thing to do is to reload all definitions that use the redefined name.

print (string)

This is just for debugging purposes. 'print ("test\n")' for instance outputs 'test' on a line by itself to stdout. This command has nothing to do with any I/O functions that will be described below.

load string

A file named by string (any Unix file name path) of *fun* source text may be loaded into your current work-space. Loading entails the reading and parsing of the file, executing all top-level commands in sequence, which itself might cause new function definitions to be added to the ones that already existed, and returning to read from the current input file. Loads may be nested. So a file that is loaded may itself also contain load commands. This is similar to include directives found in other programming languages.

The built-in (prefix) functions are:

for booleans:

! (not), && (and), || (or), -> (implies), if. The built-in boolean constants are 'False' and 'True'.

for integers:

^, +, -, *, /, %, <, <=, =, !=, >, >=, << (shift left), >> (shift right). Integer numbers must be entered in decimal format with an optional '-' sign for negative numbers. There is no unary minus operator, but a predefined "neg" function is available instead. Depending upon the installation, numbers are either represented as 32 bit or as 64 bit signed integers. The predefined constants 'INT_MIN' and 'INT_MAX' define the available range. An integer will be coerced to a float in the context of a floating point operation or function.

for floats:

^, +, -, *, /, <, <=, =, !=, >, >=, sin, cos, tan, asin, acos, atan, sinh, cosh, tanh, asinh, acosh, atanh, exp, log, log10, log2, sqrt, ceil, abs, floor. Floats numbers must be entered in fractional or scientific notation with an optional '-' sign for negative numbers. When a decimal point is present, it must be followed

by at least 1 more digit. (Floats will also be printed as such). There is no unary minus operator, but a predefined "neg" function is available instead. Floats are represented as C doubles.

for characters:

some of the arithmetic operators, like + and -, also work on characters, as well as all comparisons. Combinations of chars and integers are also allowed; the type of the result is determined by the type of the first argument. Characters are denoted just like in C. You must enclose them in single quotes, and the usual escape mechanism works for some special characters like new-line, tab, et cetera; other non-printable characters may be denoted by a backslash followed by an octal number of at most 3 digits.

for lists:

Cons, car, cdr, null, len, ith, ++ (= append). The built-in constant value 'Nil' represents the empty list. Lists may be denoted explicitly by listing the elements separated by commas enclosed in square brackets. Hence '[]' is a synonym for 'Nil'. Lists have a rich syntax. *fun* offers a list notation for arithmetic series and so-called list comprehensions. If you are familiar with Miranda, you should be warned here that *fun* has no infix list constructor operator. Instead, lists allow a syntax where the constructor operator ':' is embedded. For instance, in Miranda you write 'a:as' to construct the list with 'a' being its first element and 'as' the rest of the list. In *fun* you are required to write 'Cons a as' or (which is much more eloquent) '[a : as]'. It then comes to no surprise that '[:]' denotes the function 'Cons'.

Input/Output: read, write, stdin, stdout, stderr.

The functions read and write accept a string as argument that names the file (i.e., a Unix path name). The file will be lazily read/written. For read its contents is returned as a list of characters. For write, the second argument must be a string and it is written character by character. The stderr, stdout functions write a data object to the file stderr, resp. stdout. Only strings can be output. Anything else must first be converted to a string e.g. by the built-in 'shows' or predefined 'show'. The return value is the function itself. This means that the following expression:

stdout "Hello " "World;" "What FUN!" "\n"

has the side-effect (i.e., writing to stdout of):

Hello World; What FUN!

But only once! You thus should not rely on this effect when the above stdout call is used within a function body: the side-effect of writing to the file stdout only happens once during evaluation and because the language is referentially transparent you will each time see the function's return value. Supplying the empty string as argument to stdout or stderr causes the file to be closed and the function itself to be returned. The next call will open a fresh stream.

'stdin' accepts no arguments. When called, it opens a new stream on stdin and starts reading characters from it till the end-of-file condition is met. The characters become available in a list, in other words the return value is a potentially infinite string. Of course the input is lazily read. Upon end-of-file stdin is closed and the list construction stops. Here is a minimalistic example of a copy script:

```
#! fun -p
/* Copies standard input to standard output. */
stdout stdin;
```

WARNING: there is no support for error diagnosis when opening, reading, and writing files. For now, *fun* will simply issue some error message to stderr. But you cannot catch the error yourself in a program.

miscellaneous

The program has a built-in constant Stddefs with has as value the list of the names (strings) of predefined objects. Another constant is Builtins that holds all names of built-in operators (prefix functions) in a similar way. Yet another useful constant is 'Argv' which is a list of all arguments to the call of the *fun* program. Each list element is a string.

EXAMPLE

Here are some examples that in fact are taken from the standard definitions file.

The following defines a global function '#' with 1 argument that returns the length of a list. It uses recursion, hence the 'letrec' definition.

```
letrec # = L list . if (null list) 0 (+ 1 (# (cdr list)));
```

Here is how this can be used:

```
# [ 1,2,3,4 ];
==> 4
```

(There is a built-in function to determine the length of a list; it's called 'len' and works faster than the one above). We here use the '==>' symbol to indicate the response of the program; in reality only the result '4' will be output. Note also that (in a restricted way) you may use non-alphanumeric identifiers. Already the possibilities for characters in an identifier include the characters '_' (underscore) and '@' (at-sign), and, apart from these and the usual letters and/or digits, any identifier character except for the first may also be either a '$' (dollar-sign), a '*' (asterisk), or a '' (single quote). In principle any non-whitespace character sequence that does not coincide with a punctuation symbol or a reserved word may be used as an identifier. This is the closed you may get to defining your own 'operator' symbols. Together with the possibility to use them in an infix

style this gives great versatility in expressing your definitions in a clear and concise way. The precise rule is (in terms of a lex regular expression) that the first character must match

```
[^ \t\n\v\f/‘’";.,(){}[\]$0-9_a-zA-Z]
```

and that any subsequent characters match

```
[^ \t\n\v\f‘’";.,(){}[\]0-9_a-zA-Z]*
```

This rule is applied as last rule during lexical analysis. Hence, if the character sequence can be matched by any other rule, that rule will take precendence. Still, this scheme allows for most of the Miranda operators to have identical counterparts (albeit only to be used in prefix form) in *fun:*

```
let ~ = !;
let \/ = ||;
let & = &&;
let # = len;
```

Mind that the rules for these special identifiers might get changed in the future when the need arises to reserve some of the now allowed special characters for exclusive use in the syntax.

To apply a function ‘f’ to every element of a list ‘l’ and then return the thus modified list we can define the general ‘map’ function:

```
letrec map = L f l . if (null l) []
                     (Cons (f (car l)) (map f (cdr l)));
```

Above ‘Cons’ is the list constructor function. We could also have used:

```
letrec map = L f l . if (null l) [] [ f (car l) : map f (cdr l) ];
```

or, hiding the explicit lambda ‘L’:

```
letrec map f l = if (null l) [] [ f (car l) : map f (cdr l) ];
```

Here is an example of using ‘map’:

```
/* Increment every list element: */
map (+ 1) [ 1,2,3,4,5 ];
==> [ 2, 3, 4, 5, 6 ];
```

You may use infinite data structures; here is how one can define the natural numbers 0,1,2,... as an infinite list:

```
let N = letrec G = L n . [ n : G (+ n 1)] in G 0;
```

or, alternatively:

```
let N = G 0 where G n = [ n : G (+ n 1) ] end;
```

or, even more concise:

```
let N = [0..];
```

Of course, you should never attempt to output an infinite data structure; typing 'N' would result in:

```
[ 0, 1, 2, 3, 4, 5, 6, and on, and on, and on ...
```

until a stack overflow occurs or memory is exhausted. But, thanks to fully lazy evaluation, it is possible to do the following:

```
let square n = * n n;
take 10 (map square N);
==> [ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
```

You will find that most of the Miranda standard function definitions are present under the same name in *fun.* Since there does not (yet) exist a *fun* reference manual, we advise to consult any good book on Miranda or on functional programming in general, to learn more about them.

FILES

/usr/es/bin/fun

executable

/usr/es/lib/stddefs.fun

standard definitions

~geert/fun/[src|doc|ex]

sources, documentation, and more examples.

SEE ALSO

"The Implementation of Functional Programming Languages"
Simon L. Peyton Jones,
Prentice-Hall, 1987

"Programming Language Theory and its Implementation"
Michael J.C. Gordon,
Prentice-Hall, 1988

"Programming with Miranda (TM)"
Chris Clack, Colin Myers, Ellen Poon,
Prentice-Hall, 1995

AUTHOR

(C) 1995-1996 Geert Janssen
Dept. Electrical Engineering, EH 9.26
Eindhoven University
P.O. Box 513
5600 MB Eindhoven
The Netherlands
Phone: (+31)-40-2473387
E-mail: geert@ics.ele.tue.nl

KNOWN DEFICIENCES

fun is merely an experimental program. Don't expect too much of its run-time efficiency. A crude form of mark-and-sweep garbage collection has been implemented. Be aware of infinite recursion and infinite data structures. No explicit tests for these situations are present (and theoretically are impossible to perform anyway). In interactive mode, you may interrupt reduction by sending the SIGINT signal to the process. This is usually bound to the ^C key. *fun* is rather loosely typed and no compile-time type-checking is done. Although most operations do type-check their operands at run-time, no provisions are made to signal an error message and/or abort reduction. Typically in case of errors, reduction can't proceed and the partially reduced, SK-compiled, input expression will be echoed to stdout, which will simply look like a lot of gibberish to the uninitiated user. On the other hand this allows for easy manipulation of symbolic expressions (containing uninterpreted names) and therefore greatly helps in understanding what functional languages are all about.

BUG REPORTS

Please report bugs to geert@ics.ele.tue.nl, or just: Have FUN!

Disclaimer: even if you think to have found a 'bug', I might still decide to consider it a useful (or even necessary) feature, and therefore won't feel obliged to 'fix' it. Of course, this does not apply to program crashes other than running out of memory (which, by the way, is supposed to be monitored and will cause an error message to be generated).

Index

This document was created by man2html, using the manual pages.
Time: 17:28:55 GMT, March 11, 2022