fp egg

An interpreter/translator for a dialect of John Backus' FP language
Extension for Chicken Scheme
Version 1.1

felix

Table of Contents

1 About this egg

1.1 Version history

`1.1`	Added some builtin functions, extended atom syntax
`1.0`	Initial release

1.2 Requirements

This egg requires the following extensions:

`silex`, `lalr`

1.3 Usage

Load this egg like so:

```
(require-extension fp)
```

2 Documentation

2.1 Introduction

This extension translates programs in a dialect of the FP programming language into Scheme. You can use the translator interactively, as a library or as a compiler extension (the latter allows you to compile FP code into executables and/or libraries).

To use it interactively, invoke the `fp-repl` procedure (see below). To use it as a library, call `fp-eval`.

If you want to compile FP programs, pass the `-X fp` option to the CHICKEN compiler driver, like this:

```
% csc -X fp myprogram.fp
```

A program consists of a list of definitions separated by semicolons, like this:

```
square == x  [id, ~2];
main   == square  tonum
```

The left hand side of a definition specifies a name and the right hand side should be a functional form. A definition may be followed by auxilliary definitions enclosed in `{ ... }` which are only visible in the preceding definition.

Identifiers may consist of lowercase letters or an underscore. Any character with an ASCII/ISO-8859-1 code below or equal 32 is ignored. Any other character is treated as an identifier of length 1.

Comments follow C-style (`/* ... */`) and may not be nested. `#!` is also parsed as a comment and ignores everything up to the next line.

2.2 Objects

An object is an atom (a symbol consisting of uppercase characters or `_`, a character (`'char`) a sequence (`<x1, ...>`), a character sequence `" ... "` or a number. The atom `F` is also used as the boolean false value. Atoms may also be given as `| ... |` when they should contain special characters.

If the numbers extension is loaded, then FP programs are capable of calcuating with bignums and exact rationals.

2.3 Builtin functions

`(f -> g; h): x`	`if f:x then g:x else h:x`	`[conditional]`
`(f  g): x`	`f:(g:x)`	`[composition]`
`f x`	`f:x`	`[application]`
`/f:<x1, x2, ...>`	`f:<x1, f:<x2, ...>>`	`[insertion]`
`@f:<x1, x2, ...>`	`<f:x1, f:x2, ...>`	`[mapping]`
`N:<x1, x2, ...>`	`xN (negative number select from the right)`	`[selection]`
`OP:<x, y>`	`x OP y, where OP is "+", "-", "x" (multiply), "" (divide), "bior" (bitwise or), "band" (bitwise`	

	and), or "bxor" (bitwise xor)	
bnot:x	bitwise not	
(~O):x	O, where O is an object	[constant]
[f1, f2, ...]:x	<f1:x, f2:x, ...>	[construction]
id:x	x	[identity]
hd:<x1, x2, ...>	x1	
tl:<x1, x2, ...>	<x2, ...>	
null:x	if x = <> then T else F	
atom:x	if x = <...> then F else T	
apndl:<x, <y1, ...>>	<x, y1, ...>	
apndr:<<x1, ...>, y>	<x1, ..., y>	
cat:<<x1, ...>, <y1, ...>>	<x1, ..., y1, ...>	
reverse:<x1, x2, ...>	<xN, ..., x2, x1>	
length:<x1, ..., xN>	N	
eq:<x, y>	if x = y then T else F	
lt:<x, y>	less than (numbers only)	
gt:<x, y>	greater than (numbers only)	
not:x	if x != F then T else F	
tonum:x	converts character sequence to number	
tochar:x	converts number to char	
tostring:x	converts number into character sequence	
tosym:x	converts character sequence to atom	
(error "..."):x	prints error message and argument and exits	
(debug "..."):x	prints debug message and argument	
show:x	prints x and returns it	
read:s	read contents for file with the name s	
write:<s1, s2>	write string s2 into file with the name s1	
(*f):<x1, x2, ...>	removes elements from the sequence for which f:xI is false	
(bu f x):y	f:<y, x>, x must be an object	
?:n	returns a random integer between 0 and x-1	
system:s	execute shell command and return status code	
load:s	load FP source code or compiled .so/.dll	
(while p f):x	if p(x) is true, (while p f) : f(x), otherwise p(x)	
and:<x, y>	if x != F then y else F	
or:<x, y>	if x != F then x else y	
f & g	f -> g; ~F	
f ^ n	ff... (n times)	
gensym:symbol	return fresh symbol with name "xN", where "N" is some number	

(Alternative symbols are "." for "" and "%" for "")

2.4 Grammar

```
PROGRAM      --> DEFINITION | APPLICATION ...
DEFINITION   --> ID "==" EXPR ["{" PROGRAM "}"] [";"]
```

```
              |  ID "==" EXPR [";"]
APPLICATION  --> EXPR ":" OBJECT ...
EXPR         --> EXPR0 "->" EXPR0 ";" EXPR
              |  "while" EXPR2 EXPR
              |  EXPR0
EXPR0        --> EXPR1 "&" EXPR
              |  EXPR1
EXPR1        --> EXPR2 "" EXPR1
              |  EXPR2 [EXPR1]
EXPR2        --> "/" EXPR2
              |  "@" EXPR2
              |  "*" EXPR2
              |  EXPR2 "^" NUMBER
              |  "bu" EXPR2 OBJECT
              |  "(" EXPR ")"
              |  CONSTRUCTION
              |  VALUE
VALUE        --> NUMBER
              |  "~" OBJECT
              |  "debug" STRING
              |  "error" STRING
              |  BUILTIN
              |  OBJECT
OBJECT       --> CHAR
              |  NUMBER
              |  SEQUENCE
              |  ATOM
              |  STRING
CHAR         --> "‘" CHARACTER
SEQUENCE     --> "<" [OBJECT {"," OBJECT}] ">"
CONSTRUCTION --> "[" [EXPR {"," EXPR]} "]"
```

2.5 Example

```
/* fac.fp */

fac == eq0 -> ~1; x  [id, fac  sub1]
  { eq0  == eq  [id, ~0];
    sub1 == -  [id, ~1] }

main == fac  tonum
```

2.6 API

fp-parse [procedure]

`(fp-parse INPUT)`

Parses the FP code given in `INPUT`, which should be a string or an input port and returns its Scheme representation as a list of Scheme toplevel expressions.

This Scheme code can be directly evaluated.

`fp-eval` [procedure]

```
(fp-eval INPUT)
```

Parses and evaluates the FP code given in `INPUT`.

`fp-repl` [procedure]

```
(fp-repl [PROMPT])
```

Executes a read-eval-print-loop that prints `PROMPT`, reads a line of FP code and evaluates it, printing the returned result.

2.7 Interfacing to/from Scheme

All top-level definitions in FP will result in a Scheme procedure definition of a procedure of one argument, with the name prefixed with `fp:`, so for example

```
fac == /x  !
```

will result in a procedure named `fp:fac` that you can call from Scheme like any other procedure.

FP programs can call Scheme procedures, provided they have a name with the `fp:` prefix and accept a single argument, returning a single value and accept/return values that are meaningful in FP programs. Scheme and FP data types are related in the following manner:

Scheme	**FP**
symbol	atom
char	char
list or string	sequence
number	number

2.8 Standard library

A small library of useful functions is installed in the CHICKEN extension repository under the name `stdlib.fp`, which you can access by putting `load:"stdlib.fp"` at the start of your FP program.

Index