fraccalc

an exact precision calculator language
version 1.0

Mikhail V. Sinitcin

This manual documents `fraccalc`, an exact precision calculator language. `fraccalc` may be considered as the direct descendant of B.~Kernighan and R.~Pike `hoc`.

This manual is part of GNU `fraccalc`.

You may contact the author by: e-mail: `sinitcinmv@rambler.ru`

1 Introduction

1.1 Description

`fraccalc` [-h/–help] [-f/–file] [-l/–lib] [-V/–version]

`fraccalc` is a language that supports proper and improper fractions with interactive execution of statements. There are some similarities in the syntax to the C programming language. `fraccalc` starts by processing code from the standard input. `fraccalc` may also read code from the file on the command line after `-f` or `-l` option. The option `-l` allows to add commands from standard input. The option `-f` causes immediate exit after executing last command from file. All code is executed as it is read.

The author would like to thank Vladimir Lidovski for guidance and some advice.

Email bug reports to `sinitcinmv@rambler.ru`. Be sure to include the word "fraccalc" somewhere in the "Subject:" field.

1.2 Command Line Options

`fraccalc` takes the following options from the command line:

`-h, --help`
: Print the usage and exit.

`-l, --lib` Obtain commands from file.

`-f, --file`
: Obtain commands from file end exit.

`-V, --version`
: Print the version number.

2 Basic Elements

2.1 Fractions

The basic element in `fraccalc` is the fraction. The fractions may be proper, improper, and decimal. All numbers are represented internally in the fraction form and all computations are done with fractions. The integer part of a number must be separated from fractional part by floating point sign. The numerator and the denominator are separated by spacing underscore or low line sign.

2.2 Variables

Numbers are stored in variables. Names begin with a letter followed by any number of letters, digits and underscores.

2.3 Comments

Comments in `fraccalc` start with the characters `#` and end with the end of line marker. The end of line character is not part of the comment and is processed normally.

3 Expressions

3.1 About Expressions and Special Variables

The numbers are manipulated by expressions and statements. Since the language was designed to be interactive, statements and expressions are executed as soon as possible. There is no main program. Instead, code is executed as it is encountered. (Functions, discussed in detail later, are defined when encountered.)

A simple expression is just a constant. `fraccalc` converts constants into fractions. Input numbers may contain the characters 0-9, "`.`" and "`_`".

Full expressions are similar to many other high level languages. Since there is only one kind of number, there are no rules for mixing types.

3.2 Basic Expressions

In the following descriptions of legal expressions, "expr" refers to a complete expression and "*var*" refers to a simple variable. A simple variable is just a *name*

`- expr` The result is the negation of the expression.

`++ var` The variable is incremented by one and the new value is the result of the expression.

`-- var` The variable is decremented by one and the new value is the result of the expression.

`var ++` The result of the expression is the value of the variable and then the variable is incremented by one.

`var --` The result of the expression is the value of the variable and then the variable is decremented by one.

`expr + expr`
The result of the expression is the sum of the two expressions.

`expr - expr`
The result of the expression is the difference of the two expressions.

`expr * expr`
The result of the expression is the product of the two expressions.

`expr / expr`
The result of the expression is the quotient of the two expressions.

`expr ^ expr`
The result of the expression is the value of the first raised to the second. The second expression must be an integer. (If the

second expression is not an integer, the expression is truncated to get an integer value.) It should be noted that expr^0 will always return the value of 1.

`( expr )` This alters the standard precedence to force the evaluation of the expression.

var `= expr`
The variable is assigned the value of the expression.

3.3 Relational Expressions

Relational expressions are a special kind of expression that always evaluate to 0 or 1, 0 if the relation is false and 1 if the relation is true. These may appear in any legal expression. The relational operators are

`expr1 < expr2`
The result is 1 if expr1 is strictly less than expr2.

`expr1 <= expr2`
The result is 1 if expr1 is less than or equal to expr2.

`expr1 > expr2`
The result is 1 if expr1 is strictly greater than expr2.

`expr1 >= expr2`
The result is 1 if expr1 is greater than or equal to expr2.

`expr1 == expr2`
The result is 1 if expr1 is equal to expr2.

`expr1 != expr2`
The result is 1 if expr1 is not equal to expr2.

3.4 Boolean Expressions

Boolean operations are also legal. The result of all boolean operations are 0 and 1 (for false and true) as in relational expressions. The boolean operators are:

`!expr` The result is 1 if expr is 0.

`expr && expr`
The result is 1 if both expressions are non-zero.

`expr || expr`
The result is 1 if either expression is non-zero.

3.5 Precedence

The expression precedence is as follows: (lowest to highest)

```
|| operator, left associative
&& operator, left associative
! operator, nonassociative
Assignment operator, right associative
Relational operators, left associative
+ and - operators, left associative
* and / operators, left associative
^ operator, right associative
unary - operator, nonassociative
++ and -- operators, nonassociative
```

So `fraccalc` uses the same expression precedence as in C programming language. Consider the expression:

```
a = 3 < 5
```

This would assign the result of "3 < 5" (the value 1) to the variable "a".

3.6 Special Expressions

There are a few more special expressions that are provided in `fraccalc`. These have to do with user-defined functions and standard functions. They all appear as "*name*(*parameters*)". See Chapter 5 [Functions], page 8, for user-defined functions. The standard functions are:

`integer ( expression )`
: The value of the integer part of a number.

`numerator ( expression )`
: The value of the numerator of a fraction.

`denominator ( expression )`
: The value of the denomiminator of a fraction.

4 Statements

Statements (as in most algebraic languages) provide the sequencing of expression evaluation. In `fraccalc` statements are executed "as soon as possible." Execution happens when a newline in encountered and there is one or more complete statements. Due to this immediate execution, newlines are very important in `fraccalc`. In fact, both a semicolon and a newline are used as statement separators. An improperly placed newline will cause a syntax error. A statement list is a series of statements separated by semicolons and newlines.

expression The expression is evaluated and printed to the output. After the number is printed, a newline is printed. The expression terminated by semicolon is not printed to the output.

`if` *(expression) statement1 [*`else` *statement2]*
The if statement evaluates the expression and executes statement1 or statement2 depending on the value of the expression. If the expression is non-zero, statement1 is executed. If statement2 is present and the value of the expression is 0, then statement2 is executed.

`while` *(expression) statement*
The while statement will execute the statement while the expression is non-zero. It evaluates the expression before each execution of the statement. Termination of the loop is caused by a zero expression value or the execution of a `break` statement.

`for` *([expression1] ; [expression2] ; [expression3]) statement*
The `for` statement controls repeated execution of the statement. *Expression1* is evaluated before the loop. *Expression2* is evaluated before each execution of the statement. If it is non-zero, the statement is evaluated. If it is zero, the loop is terminated. After each execution of the statement, *expression3* is evaluated before the reevaluation of expression2. If *expression1* or *expression3* are missing, nothing is evaluated at the point they would be evaluated. If *expression2* is missing, it is the same as substituting the value 1 for *expression2*. The following is equivalent code for the `for` statement:

```
expression1;
while (expression2) {
   statement;
   expression3;
}
```

`break` This statement causes a forced exit of the most recent enclosing `while` statement or `for` statement.

`return` *expression*
: Return the value of the expression from a function. (See Chapter 5 [Functions], page 8.)

`include` *filename*
: Obtain commands from file with filename.

exit
: When the `exit` statement is read, the `fraccalc` processor is terminated.

4.1 Pseudo Statements

These statements are not statements in the traditional sense. They are not executed statements. Their function is performed at "compile" time.

`proper`
: Set default proper fraction format for output. This mode is set initially.

`improper`
: Set default improper fraction format for output.

5 Functions

Functions provide a method of defining a computation that can be executed later. Functions in `fraccalc` always compute a value and return it to the caller. Function definitions are "dynamic" in the sense that a function is undefined until a definition is encountered in the input. That definition is then used until another definition function for the same name is encountered. The new definition then replaces the older definition. A function is defined as follows:

name (*parameters*) { *newline*
statement_list }

A function call is just an expression of the form "`name (`*parameters*`)`".

Parameters are numbers. In the function definition, zero or more parameters are defined by listing their names separated by commas. Numbers are only call by value parameters. It is possible to miss some parameters values at function call — these missed parameters are set to 0. These missed parameters may be used as local variables inside function body only.

The function body is a list of `fraccalc` statements. Again, statements are separated by semicolons or newlines. Return statements cause the termination of a function and the return of a value. There are two versions of the return statement. The first form, "`return`", returns the value 0 to the calling expression. The second form, "`return` *expression*", computes the value of the expression and returns that value to the calling expression. There is an implied "`return 0`" at the end of every function. This allows a function to terminate and return 0 without an explicit `return` statement.

At the description of function the opening brace be on the same line and all other parts must be on following lines.

```
 d (n) { return (2*n); }
 d (n) {
return (2*n);
}
```

The following is the definition of the recursive factorial function.

```
fact (x) {
  if (x <= 1) return 1;
  return f(x-1) * x;
}
```

6 Space in numbers

Some implementations of `bc` allow spaces in numbers. For example, `"x=1 3"` would assign the value 13 to the variable x. The same statement would cause a syntax error in this version of `fraccalc`.

7 Errors in execution

If a syntax error in expression is found then the processor skips this expression.

Table of Contents