finL

Language Reference Manual

Manager Lauren O'Connor, leo2118
System Architect Robert Cornacchia, rlc2160
Language Güru Josh Fram, jpf2141
Tester Paddy Quinn, pmq2101

Contents

I. Introduction

As quantitative and algorithmic trading strategies are being increasingly adopted by investment firms, the goal of finL is to provide a language to simplify the development of financial programs. Via Yahoo's web query language (YQL) and Finance API, finL provides easy access to current stock information. finL's special currency, percent, stock, order, and portfolio data types make software development simpler for financial analysts. In addition, reserved words and built-in functions allow users to easily utilize more functionality, such as switching between portfolios, converting different currency denominations, or exporting results to a .csv file.

II. Lexical Conventions

Tokens:

There are five types of tokens in finL. They are: identifiers, reserved words, whitespace, operators, and separators.

Identifiers:

Identifiers are any sequence of letters and underscores that signify the name of a variable or function. Identifiers in finL are case-sensitive and ny other punctuation or digits are not allowed.

Comments:

Only single-line comments are supported in finL. Comments are indicated with a single # symbol, i.e:

```
# this is a one-line comment
```

Whitespace:

All whitespace (spaces, tabs, new lines) in finL is ignored.

Separators:

finL uses the following characters as separators:

- `{}` – code block separator
- `()` – grouping separator and parameter list separator
- `;` - statement delimiter

Reserved Words & Symbols:

Types & Null (see III: Types):

`int`
`float`
`percent`
`null`
`array <type>`
`string`
`currency`
`stock`
`order`
`portfolio`

Boolean Logic Operators (see IV: Expressions):

`and`
`or`
`not`

Control Flow & Loops (see V: Control):

`?`
`??`
`!`
`for`
`when`
`break`
`index`

Functions & Related (see VI: Functions):

`function`
`return <expression>`
`void`
`main`
`use`
`export`

III. Types

Primitive

`int`

- signed 64 bit type
- a string of numeric characters without a decimal point, and an optional sign character
- an integer literal may be expressed in decimal (base 10).
- example: `-4`

`float`
- signed 64 bit type
- a string of numeric characters that can be before and/or after a decimal point, with an optional sign character
- a float literal may be expressed in decimal (base 10)
- example: `34.55`

`percent`
- signed 64 bit type
- a string of numeric characters that can be before and/or after a decimal point, with an optional sign character
- value of percent treated in calculations as `<value>/100`
- example:
```
percent myRate<<50;
myRate*80 # evaluates to 40, since 50 is treated as 0.5
```
- makes using percentages more intuitive to/readable for the user

`null`
- the value of an uninitialized object
- example: `null`

Non-Primitive

`array`
- arrays only hold elements of the same type
- finL arrays have the same behavior as Java ArrayLists, i.e. dynamic sizing
- adding an element:
  - <identifier>.add(<element>);
  - example: `myArray.add("hello");`
- accessing an element:
  - <identifier>.at(<index>);
  - example: `myArray.at(3);`
- removing an element:
  - <identifier>.at(<index>).remove;
  - example: `myArray.at(2).remove;`
- setting an element:
  - <identifier>.at(<index>) << <value>;
  - example: `myArray.at(2) << "world";`
- finding the maximum element:
  - <identifier>.max();
  - example: `myArray.max();`
- finding the minimum element:
  - <identifier>.min();
  - example: `myArray.min();`
- finding the average of the array:

        - <identifier>.avg();
        - example: `myArray.avg();`
    - sorting an array in ascending order:
        - <identifier>.sort();
        - example: `myArray.sort();`

`string`

- a finite sequence of ASCII characters, enclosed in double quotes
- example: `"Hello"`

`currency`

- tuple with float and string
- currency has two attributes: <identifier>.value (float determining the value) and <identifier>.denomination (string determining which currency)
- example: `myCurrency.value<<312.40;`
- currency conversions supported:
  USD (US Dollar)
  EUR (Euro)
  JPY (Japanese Yen)
  GBP (Great Britain Pound)
  CHF (Swiss Franc)
  CNY (Chinese Yuan)
  (later versions of finL will support more currency conversions)
- *string() function*:
  `<identifier>.string` converts currency into a string (adds corresponding currency symbol and rounds off to hundredths place)
  example:

```
currency lb;
lb.value<<100.4567;
lb.denomination<<"GBP";
print(lb.string()); # prints £100.46
```

`stock`

- wrapper type that contains the stock ticker and all data from Yahoo Finance YQL database table yahoo.finance.quotes (see Appendix for Yahoo Finance YQL database reference)
- *.populate() function:*
  `<identifier>.populate(<optional string array>)`
  populates/refreshes all data if no string is specified or specified data if an input string array is specified
  example:

```
stock stk << @GOOG;
stk.populate(); # populates all data from yahoo
array string arr;
arr.add("EBITDA");
```

```
arr.add("PERatio");
stk.populate(arr); # refreshes only data within arr
```

- *.get() function:*
  `<identifier>.get(<string>)` gets specified data from stock type
  example:
  ```
  stock stk << @GOOG;
  stk.populate(); # populates all data from yahoo
  stk.get("EBITDA"); # returns the EBITDA data in the
                     # stk variable
  ```
- *.string() function:*
  `<identifier>.string()` converts stock into a string
  example:
  ```
  stock stk<<@TSLA;
  print(stk.string()); # prints @TSLA
  ```

`order`

- buy or sell order
- contains:
  - `<identifier>.shares`: the number of shares (populated at initialization)
  - `<identifier>.stock`: a stock type (populated at initialization)
  - `<identifier>.price`: the price of the stock at time of sale (populated when the order is executed)
  - `<identifier>.time`: timestamp (populated when the order is executed)
- in a buy order, the number of shares is positive, but in a sell order, the number of shares is negative
- *.execute() function:*
  `<identifier>.execute()` executes the buy or sell order immediately (places a market order); to place a limit order (execute the order based on a trigger) use a when loop
  example:
  ```
  order myOrder;
  myOrder.shares<<10;
  myOrder.stock<<@AAPL;
  myOrder.execute(); # executes order
                     # edits current portfolio
  ```

`portfolio`

- type that contains order history, stocks owned, and capital on hand
- a finL program can work on multiple portfolios at a time through when loops, however each thread can have at most one active portfolio
- program starts with a default portfolio, imported from a csv
- new portfolios can be created or switched to using the `use` keyword

Casting

finL does not support casting.

IV. Expressions

Declaration & Assignment

To declare what an identifier means, the general syntax is as follows:

- `<type> <identifier>;`
- examples: `int x; float number; string name;`
- *arrays*
  - arrays are declared `array <type> <identifier>;`
  - example: `array int numberList;`

To set or reset a value of an identifier, the syntax is as follows:

- `<identifier> << <value>;`
- examples: `x<<3; dollars<<2.2;`
- *currency*
  - currency attributes should be initialized separately
  - `<identifier>.value<<43.432;`
    `<identifier>.denomination<<"EUR";`
- *arrays*
  - arrays can be initialized via declaration and then continuous calling of the "add" function

Shorthand Operators

finL also supports shorthand operators (for ints, floats, and percents) that combine arithmetic and assignment, for the following operations: addition (`+<<`), subtraction (`-<<`), multiplication (`*<<`), and division (`/<<`):

example: `x<<x+3;` can also be written `x+<<3;`
example: `x<<x*4;` can also be written `x*<<4;`

Arithmetic Operators

finL supports the following arithmetic operators, in order of precedence:

power (`**`)
multiplication(`*`)/division(`/`)
addition(`+`)/subtraction(`-`)
modulus(`%`)

Relational Operators

finL supports the following relational operators:

- `<` (less than)
- `>` (greater than)
- `<=` (less than or equal to)
- `>=` (greater than or equal to)
- `=` (equal to)
- `not =` (not equal to)

All return a result of 0 if the condition is false or 1 if the condition is true. Equality operators support integers, floats, percents, currencies (currency.value is compared), and strings, while all other relational operators support integers, percents, currencies, and floats.

Logic Operators

finL supports the following boolean operators as reserved keywords:

- `and`
  - logical intersection of two expressions
  - example: `0 and 1` evaluates to `0` (false)
- `or`
  - logical union of two expressions
  - example: `1 or 0` evaluates to `1` (true)
- `not`
  - logical negation of an expression
  - example: `not 1` evaluates to `0` (false)

Any non-zero number is considered true, and zero is considered false.

V. Control

Conditionals

All conditional statements must terminate with a colon.

if

`?(<boolean expression>): { … }`

- if conditionals are represented with a single question mark (`?`) at the end of the statement
- denotes an if statement, which must be followed by a boolean expression, and a block that is executed if the condition is true
- example: `?(x = 2): { print("X is 2!");}`

else if

`??(<boolean expression>): { … }`

- all else if statements follow an if statement. They are denoted by two question marks (`??`).
- example: `??(x = 3) { print("X is actually 3!");}`

else

`! { … }`

- denotes an else statement that occurs following a `?` or `??` statement. It is not followed by a boolean expression, but it is still followed by a block
- example: `! { print("X is 2!");}`

Full conditional example:

```
int number << 0;
?(number>0)
      {number+<<1;}
??(number<0)
      {number-<<1;}
!
      {number+<<5;}
print(number);
# prints 5
```

Loops

for

`for <integer> to <integer> by <integer>: { … }`

- perform iterations of a loop a specified number of times until the boolean condition (evaluated at the start of each iteration) is false
- range is inclusive (i.e. 1 to 5 would iterate 1, 2, 3, 4, 5 times)
- example: `for 1 to 10 by 1: { … }`
- NOTE: for loops should not be used to continually query the YQL database; use a `when` loop

when

`when(<boolean expression>):`

- a key control loop in finL
- starts a new thread to support being able to place multiple limit orders
- the when loop checks the conditional periodically based on the Yahoo Finance YQL database rate limiting and the number of active when loops
- when the when loop has a boolean expression that evaluates to true it executes its body once and terminates

- finL does type checking on the boolean expression to determine what it needs to repopulate and if it should change the time between calls
- example:

```
stock chill << @NFLX;
chill.populate();
order netflixOrder;
netflixOrder.shares << 70;
netflixOrder.stock << @NFLX;
when (chill.get("FiftydayMovingAverage") >
     chill.get("TwoHundreddayMovingAverage")): {
     netflixOrder.execute();
}
netflixOrder.execute();
# this mini-program executes the netflixOrder once
# no matter what and once in the when loop
# according to the boolean, the when loop starts a
# new thread so the other order can be executed
# immediately
```

index

`index`

- used to get the value of the current iteration in a for loop
- example:

```
for 1 to 5 by 1: {
     print(index);
} # prints 12345
```

break

`break;`

- break out of the current iteration of the loop and exit the function block
- example:

```
for 1 to 10 by 1: {
     if(index=2){break;}
}
```

Scope

Identifiers declared within bracket (`{}`) separators have a local scope and cannot be accessed outside that bracket. Functions have a global scope, and must be declared with unique identifiers. Portfolios have a global scope. When a when loop is entered the current portfolio is passed into the scope of the when loop. The current portfolio can then be changed without affecting the portfolio within the when loop.

VI. Functions

Reserved Words:

`function`

- a reserved word used to define a user-defined function that takes 0 or more parameters, and returns an *<expression>* or `void`
- example: `function void tester(int testParam): { … }`

`return` *<expression>*

- return *expression* to the caller from the function block
- example: `return 1;`

`void`

- a reserved word returned to the caller from the function block that indicates the function does not return a value
- example: `return void;`

`main`

- name of the entry function, the first function called by the program at run-time. Every finL program must have a `main` function
- example: `function void main(): { … }`

`use <string>`

- keyword that sets the current portfolio to the csv file specified in the string
- example: `use "secondPortfolio.csv";`

`export <string>`

- keyword that exports the current portfolio to the csv file specified at that point in the program
- example: `export "middayPortfolio.csv";`

Built-in Functions:

`print(<primitive data type>)`

- system function that prints a primitive data type to the console
- example: `percent pct << 10; print(pct); # prints 10%`

`convert(<currency>, <string>)`

- system function that converts one denomination to another
- paramaters: takes in a currency (i.e. value and denomination), and a string indicating the desired new denomination
- returns a currency object with the new value and denomination
- example:

```
currency x;
x.value<<102.3;
x.denomination <<"USD";
currency newX << convert(x, "GBP");
#newX now holds value in pounds and "GBP"
```

Function Definition

Functions in finL are defined with the following syntax:

```
function <return type> <identifier> (<optional parameters>):
{...}
```

example:

```
function int foo(currency myEuro): {
     print(myEuro.string());
     return 1;
}
```

VII. Appendix

Yahoo Finance YQL yahoo.finance.quotes Database Table:

"symbol"
"Ask"
"AverageDailyVolume"
"Bid"
"AskRealtime"
"BidRealtime"
"BookValue"
"Change_PercentChange"
"Change"
"Commission"
"Currency"
"ChangeRealtime"
"AfterHoursChangeRealtime"
"DividendShare"
"LastTradeDate"
"TradeDate"
"EarningsShare"
"ErrorIndicationreturnedforsymbolchangedinvalid"
"EPSEstimateCurrentYear"
"EPSEstimateNextYear"
"EPSEstimateNextQuarter"
"DaysLow"
"DaysHigh"
"YearLow"
"YearHigh"
"HoldingsGainPercent"
"AnnualizedGain"
"HoldingsGain"
"HoldingsGainPercentRealtime"

"HoldingsGainRealtime"
"MoreInfo"
"OrderBookRealtime"
"MarketCapitalization"
"MarketCapRealtime"
"EBITDA"
"ChangeFromYearLow"
"PercentChangeFromYearLow"
"LastTradeRealtimeWithTime"
"ChangePercentRealtime"
"ChangeFromYearHigh"
"PercentChangeFromYearHigh"
"LastTradeWithTime"
"LastTradePriceOnly"
"HighLimit"
"LowLimit"
"DaysRange"
"DaysRangeRealtime"
"FiftydayMovingAverage"
"TwoHundreddayMovingAverage"
"ChangeFromTwoHundreddayMovingAverage"
"PercentChangeFromTwoHundreddayMovingAverage"
"ChangeFromFiftydayMovingAverage"
"PercentChangeFromFiftydayMovingAverage"
"Name"
"Notes"
"Open"
"PreviousClose"
"PricePaid"
"ChangeinPercent"
"PriceSales"
"PriceBook"
"ExDividendDate"
"PERatio"
"DividendPayDate"
"PERatioRealtime"
"PEGRatio"
"PriceEPSEstimateCurrentYear"
"PriceEPSEstimateNextYear"
"Symbol"
"SharesOwned"
"ShortRatio"
"LastTradeTime"

"TickerTrend"
"OneyrTargetPrice"
"Volume"
"HoldingsValue"
"HoldingsValueRealtime"
"YearRange"
"DaysValueChange"
"DaysValueChangeRealtime"
"StockExchange"
"DividendYield"
"PercentChange":