PSL: Portfolio Simulation Language
Language Reference Manual

Alexander Besidski ab2012@columbia.edu
Jian Huang jh2353@columbia.edu
Xin Li x174@columbia.edu
Wei-Chen Lee wl2135@columbia.edu

October 18, 2004

Introduction

PSL is a compiled financially oriented programming language. It aims to deliver easy-to-learn powerful programming tools for financial engineers, traders, investment bankers as well as finance major students. The basic syntax structure of PSL is similar to C++ and Java, yet much simpler in comparison. A lot of complex features, such as pointers and object overloading, do not appear in PSL. Anyone with a minimum amount of programming or financial experience could easily delve into PSL and write powerful programs to achieve quite interesting and practical financial trading tasks without spending much time or effort. Therefore PSL is a rather high-level language.

PSL compiler is written in Java. Although PSL provides users with a procedure-based programming interface, internally PSL is a pure object-oriented implementation. This is reflected in the complex types defined in PSL, which are meant to model real financial securities. It enables users to manipulate these objects through their corresponding member functions, which enable them to perform some highly professional practices such as portfolio management and dynamic stock market simulation.

1. Lexical Conventions

PSL defines 4 types of tokens – identifiers, keywords, constants, expression operators and separators.

1.1 Comments

Single line C-style // comments

1.2 Identifiers

Identifier is a sequence of letters and digits where the first character must be either a character or an underscore. PSL is a case-sensitive language so that identifiers with different casing are distinguished.

1.3 Keywords

The following is a list of all keywords supported by PSL.

for	if	then	else	break
string	continue	function	true	false
return	while	int	real	void
stock	bond	portfolio	portfolio	boolean
program				

1.4 Constants

PSL supports 4 types of constants – Boolean, string, integer and real.

1.4.1 Strings

A string is a sequence of characters surrounded by double quotes (“”). In order to express double quotes within a string they need to be escaped via the backslash character. (‘\”’)

1.4.2 Integers

An integer constant is simply a sequence of digits. Integer values are stored as 32 bit signed numbers.

1.4.3 Real

Real numbers are stored as 64 bit fixed precision values and consist of an integer part followed by a decimal point, a fraction part and an optionally signed integer exponent.

1.4.4 Boolean

These constant can only take on two values “true” or “false”.

2. Expressions

PSL has 7 expression precedence levels that are explained in the sections below in the order of decreasing priority.

2.1 Primary Expressions

2.1.1 All four types of constants

integer, real, string, boolean

2.1.2 Identifiers

All identifiers must be properly declared before being used.

2.1.3 (*expression*)
Any parenthesized expression is considered to be a primary expression that maintains all of the attributes of the un-parenthesized version.

2.1.4 *primary-expression* [*expression*]
This notation is only applicable to arrays with the type of the expression being an integer.
PSL will only support one-dimensional arrays.

2.1.5 *identifier* (*expression-list* $_{opt}$)
This notation is applicable to function calls where primary-expression. PSL allows recursive function calls. Simple types are passed in by value while complex types such as portfolio and stock and passed in by reference.

2.1.6 *identifier*. *method-name* (*expression-list* $_{opt}$)
This notation applies to complex types such as stock and bond. It is used to modify and read their underlying data. The dot operator is left associative. (yahoo.getExpectedFuturePrice())

2.1.7 *identifier*.*property-name*
This notation applies to complex types such as stock and bond. It is used to access properties in a Style similar to that of C# programming language. In PSL all properties can be read and set. (yahoo.price)

2.2 Unary expressions.
Unary operators are right associative

2.2.1 - *expression*
Negative of an expression.

2.3 Multiplicative operators
These types of operators are left associative.

2.3.1 *expression* * *expression*, *expression* / *expression*
The multiplication operators are valid for real and integer data types. If a real is multiplied by an integer the resulting value is a real.

2.4 Additive operators
These types of operators are left associative

2.4.1 *expression* + *expression*
The + operator is valid for all types except for boolean. If either expression is a string then both expressions must be strings and the resulting value is a string. Otherwise the result is converted to a real if at least one expression is a real and to an integer if bother operands are integers.

2.4.2 *expression* - *expression*
Same semantics as for the + operator with the exception of strings not being allowed.

2.5 Relational operators

2.5.1 *expression* >= *expression*
2.5.2 *expression* > *expression*
2.5.3 *expression* < *expression*
2.5.4 *expression* <= *expression*

These operators yield a boolean true if the specified expression is true and false otherwise.

2.6 Equality operators

2.6.1 *expression* == *expression*
2.6.2 *expression* != *expression*

These operators are right associative and have boolean return values.

2.7 Logical operators

2.7.1 *expression* && *expression*
2.7.2 *expression* || *expression*

These operators have the semantics that are similar to C and Java. Both are left associative and require both operands to be of type boolean.

2.8 Assignment operator

2.8.1 *lvalue = expression*

In PSL the assignment operator is not recursive unlike its C and Java counterparts. The operator is applicable to all PSL data types and arrays. ***lvalue*** concept is analogous to that of C.

3. Declarations

Declarations are used within function definitions and the main body of a PSL program. Their purpose is to provide the compiler with information about storage and behaviors of the identifiers. A declaration consists of a type specifier followed by a declarator.

3.1 Type specifiers

PSL allows the following type specifiers:
int, real, string, boolean, portfolio, stock, bond, cash

3.2 Declarators

Two forms or declarators are permitted.
identifier – declares a single reference to a type object.
identifier[*expression*] – declares an array of specified size of for the type. Expression must evaluate to an integer.

4. Statements

A PSL program is composed of a series of statements that are executed sequentially.

4.1 Expression statements

This is the simplest type of all statements and has the form shown below. Expressions are usually either assignment statements of function calls.
expression **;**

4.2 Compound statements

It is often necessary to treat a several statements as one. For that purpose statements can be grouped together using the C-style bracket notation.

{
***expression*$_1$;**
***expression*$_2$;**
…
}

4.3 Conditional statements

These statements should be used when program flow needs to be changed based on the value of a boolean expression.
Two syntactic alternatives are possible.

if* (*expression*) then *statement
***if* (*expression*) then *statement*$_1$ else *statement*$_2$**

Else ambiguity is resolved by associating the else with the last encountered if.

4.4 While statement

C-style loop where expression must evaluate to a boolean.

***while* (*expression*)**
{ statement }

4.5 For loop

C-style loop where *expression*$_2$ must evaluate to a boolean.
expression$_1$ is evaluated upon entering the loop for the first time.
The loop is executed while *expression*$_2$ is true.
expression$_3$ is evaluated after every successful loop iteration.

for (*expression*$_1$; *expression*$_2$; *expression*$_3$)
{ *statement* }

4.6 Break statement

Causes termination of the inner most while or for loop.

```
while ( expression )
{
        …
        break;
        …
}
```

4.7 Continue statement

Causes termination of the current while or for loop iteration and passing control onto the next interation.

```
for (expression1; expression2; expression3)
{
        …
        continue;
        …
}
```

4.8 Return statement

Causes the function to return to the caller possibly with a return value.
Examples below illustrate two possible scenarios.

```
return;
return ( expression );
```

5. Scope Rules

PSL defines scope rules similar to those of Java or C++. A variable can only be used for the duration of its inner most enclosing brackets. If a variable is declared within a function it cannot be referenced from any other function or the main body of the program. Since PSL programs consist of only a single file these scope definitions suffice.

6. Complex Types

PSL supports a set of built-in complex types and provides a user-friendly object-oriented interface for users to manipulate them in a convenient way. The interface for accessing each built-in type is similar, and users can manipulate the target financial instruments by calling the member functions and properties. The current version of PSL supports three kinds of complex types: Stock, Bond, and Portfolio, which incorporate the three most common kinds of instruments in the financial market. In general, stock is the most common type of equity securities and bond is the most common type of fix-income securities.

6.1 Stock

User creates a Stock object in the following way:

Stock <ID> = Stock([member initialization list]);

Where <member initialization list> composed one or more member initializations, separated by “,”, each member initialization takes the form:

<**member initialization**> : **<ID>=><Value>**

Where the <ID> to the left of the “=>”denotes one particular property of the Stock object. The possible properties of a Stock object includes

Name <String> [compulsory, read only once defined]
Price <String> [compulsory, must be positive]
Return <Real> [optional, default to the constant INTEREST__RATE, must be between 0 and 1]
Volatility <Real> [optional, default to the constant DEFAULT_VOLATILITY]
Dividend <Real> [optional, default to 0.0]

A simple example of a Stock definition is

Stock S = Stock(Name=>"IBM", Price=>24.5, Return=>0.30, Volatility=>0.75, Dividend=>0.20);

Which defines a Stock object called S0 that intends to characterize a real stock share issued by IBM and priced at $24.5, with an expected return rate of 30% and volatility of 75%. In addition, the stock comes with a yearly dividend rate of 20%.
The order of member initializations within the member initialization list is not relevant. For the above example, the definition could be well rewritten as

Stock S = Stock(Price=>24.5, Dividend=>0.20, Return=>0.3, Volatility=>0.75, Name=>"IBM");

Or any other order desired.

Alternatively, the volatility may be calculated from a time series of historical quotas, provided by the user in the form of an array of real numbers, as illustrated by the following example:

```
Real a[10];
// Fill up the elements of a[10]
….
S.setVolatilityFromTimeSeries(a, 10);
```

The available member functions and attributes for the Stock objects include:

```
// Get/Set the initial quoted price of the stock at time 0
Real Price;

// Get/Set volatility of the stock
Real  Volatility;

// Get/Set return of the stock
Real Return;

// Get/Set dividend of the stock
Real  Dividend;

// Get/Set name of the stock
String Name;

// Get the expected future price at time t(measured in years)
Real  getExpectedFuturePrice(real t);

// Calculate the volatility of the stock  from a time series, or an array, of historical quotas
Void setVolatilityFromTimeSeries (real a[], int N);

// Simulate a particular path of the stock movement The Stock will follow the lognormal //stochastic process. The plot
//parameter indicates whether a graphic output is desired or not.
Void Simulation(Real timestep, Int nstep, boolean plot);
```

An important property of stocks, which makes the portfolio optimization an interesting and non-trivial practice, is the correlation of stocks. Basically, this means that the stocks are correlated in some way or the other. The correlation between two different kinds of stocks is quantified statistically by the covariance, or correlation coefficient. PSL provides a global function for specifying the correlation coefficient between two (defined) stock objects:

Void SetCov(Stock s1, Stock s2, real corr);

Where corr must be a real number between –1 and 1.

One can also get the correlation coefficient between two stock objects through another global function:

Real getCov(Stock s1, Stock s2);

By default, all stocks are mutually uncorrelated (correlation coefficient is 0).

6.2 Bond

User creates a Bond object in pretty much the same way as the Stock object, i.e:

Bond <ID> = Bond([member initialization list]);

Where again <member initialization list> composed one or more member initializations, separated by “,”, each member initialization takes the form:

<member initialization> : <ID>=><Value>

Where the <ID> to the left of the “=>”denotes one particular property of the Bond object.

The possible properties of a Stock object includes

Name <String> [compulsory, read only once defined]
Price <String> [compulsory, must be positive]
Yield <Real> [optional, must be between 0 and 1]
Maturity <Real> [optional, must be positive]
Coupon<Real> [optional, must be positive]

Which gives the name, market price, yield, maturity and coupon of the bond (we assume coupon to be just a single cash flow along with the face value paid at the end of maturity). It is important that either Yield, or Maturity, but not both, must be specified explicitly in the member initialization list (as they are functionally related).

A simple example of a Bond definition is

Bond b = Bond(Name=>”Gov”, Price=>97.5, Yield=>0.30, Dividend=>0.20);

Which defines a Bond object called b that intends to characterize a real bond (either corporate or governmental) with an issue price of $97.5 (corresponding to a face value of $100), a yield of 30% and dividend rate of 20% (yearly).
The order of member initializations within the member initialization list is not relevant, just as in the Stock object. Users may specify the member initialization list in whichever order they like.

The available member functions for the Bond objects include:

```
// Get/Set the initial price of the bond at time 0
Real Price;

// Get/Set yield of the bond
// This affects the maturity as well as the yield and maturity are functionally related.
Real Yield;

// Get/Set coupon rate of the bond
Real Coupon;

// Get/Set name of the bond
String Name;
```

6.3 Portfolio

Portfolio an even higher-level object than Stock or Bond, in the sense that it is actually nothing but an assembly of these basic financial instruments. In the financial world, a portfolio consisting of a bunch of different financial instruments is often used to hedge the potential market risk, as by carefully select the ratio (or percent) of the constituent financial instruments (which is known as portfolio optimization), the risks of any individual investment instrument tend to be “diversified away” and the whole portfolio would be insensitive to any particular movement in the financial market.

To define a portfolio, one should conform to the following syntax:

```
Portfolio <ID> = Portfolio
(
        Capital => <REAL>;
        Components => {{<COMPONENT_LIST>}
);
```

Where <ID> is the name of the portfolio and must be a valid identifier name. The <REAL> followed by the “Capital=>” is a positive real number specifying the total amount of capital available for the portfolio investment.

The <COMPONENT LIST> consists of one or more elements separated by the comma, which must be either the array or single element of the defined financial instrument types.

An example:

```
Portfolio P1 = Portfolio
(
        capital => 1000,
        components => {s[10], b[5]}}
);
```

Where we defined a portfolio called P1, with an initial investment of $1000 (at time zero). The portfolio is made up of 10 shares of stocks s and 5 bonds b (see the previous two sections for the definition of s and b).

The above definition actually has another implications. Since when the portfolio is created (implicitly at the time zero), stock s is priced at $24.5 and bond b priced at $97.8, the total amount of capital spent on the financial instruments is 24.5*10 + 5*97.5= $732.5. Therefore there is an extra amount of capital amounting to 1000-732.5 = $267.5. This extra amount of $267.5 capital is implicitly treated as cash deposited in the money market which is stored in a member variable called Cash and may be accessed by the member property Cash. The cash earns a fixed interest rate specified by the INTEREST_RATE constant whose value is 0.020 (more or less the average interest rate for the saving account in U.S. banks for the previous three years). Note that Cash may be negative, in which case the portfolio achieves the investment by borrowing money at the same interest rate. In finance this is called "leverage".

The portfolio object lies at the heart of our PSL language. Basically we can perform a lot of interesting tasks on a portfolio. Available member functions are:

```
// Get/Set the cash amount in the portfolio
Real Cash;

// Get number of different kinds of stocks
int getNumStock();

// Get number of different kinds of bonds
int getNumBond();

// Explicitly set the percent ratio of the Stock with a specify name, which must be a <String> //type such as "IBM"
Void setStockPortion(String Name);

// return the percent ratio of the Stock  with a specify name, which must be a <String> type such //as "IBM"
Real getStockPortion(String Name);

// Explicitly set the percent ratio of the Bond with a specify name, which must be a <String> //type such as "Gov"
Void setBondPortion(String Name);

// return the percent ratio of the Bond  with a specify name, which must be a <String> type such //as "Gov"
Real getBondPortion(String Name);

// Test whether the portfolio is leveraging or not, i.e., whether the Cash amount is less than zero //or not.
Boolean isLeverage();

// Optimize the portfolio composition (only if there are more than 1 different kind of correlated //stocks).
Void Optimize();

// Perform a dynamic simulation of the portfolio. The Stocks will follow the lognormal //stochastic process. The plot
//parameter indicates whether a graphic output is desired or not.
Void Simulation(Real timestep, Int nstep, boolean plot);
```

7. Functions

7.1 Defining

PSL uses the following syntactic notation for function definitions.
When a function needs to terminate it must invoke the return statement.

function ***return-type identifier*** **(** ***expression-list*** $_{opt}$ **)**
{
statement-list**;**
}

7.2 Invocation

The following syntax is used in order to invoke a function, where *identifier* represents the name of the function.
identifier (*expression-list* $_{opt}$);

8. Examples

The following is a complete sample PSL program.

```
//Test.psl

program
{
        int   i;
        real  r;
        i= 5;

        // Defining a stock of IBM with initial price of $25.2 and volatility 30%.
        // Using Perl-like syntax to assign the data members
        Stock s = Stock(Price => 25.2, Volatility => 0.3, Name => "IBM");

        // Defining a stock of INTL (Intel) with initial price of $25.2 and volatility 30%.
        Stock s2 = Stock(Price => 10.0, Volatility => 0.3, Name => "INTL");

        // Defining a bond with price 99.0 (face value of bonds are always 100), with
        // a dividend of $30.0 at maturity and maturity of 5 years.
        Bond b = Bond(Price => 99.0, Dividend => 30.0, maturity => 5.0);

        // Set the correlation coefficient between stocks s and s2 to be 0.25
        SetCov(Stock s, Stock s2, 0.25);
// Construct a portfolio with a total capital of $10000, and consists of 100 shares of
// stock s, 50 shares of stock s1 and 50 bond b. Therefore the rest capital 10000 – 100*25.2 – 50* 10.0
// 99.0*50=$2030 will all be cash, which by default will earn a yearly interest rate of 1%
// (compounded).
Portfolio P1 = Portfolio
(
capital => 1000,
components => {s[100], s2[50], b[50]}
);


// get the portion of cash of p1 and save it in variable r;
r = P1. Cash;

// optimize the portfolio;
P1.optimize();

// Set portion of s to 0.30
P1.set(s,0.30);

// get portion of s after optimization
P1.get(s);

// get portion of b after optimization
P1.get(b);

// Varying the portion of s from 0.05 to 1.0
// simulate 100 days performance of portfolio and save output to out.dat, and plot the  results as well
        // (as specified in the third argument).

        for(i = 1; i<= 20; I=I+1)
        {
           P1.set*(s,0.05*i)
           P1.simulate(100,'out.dat', TRUE);
        }
}
```