# TS Calculator

## Introduction

The tool is used analyze streams of ticks using a domain specific language.

## Implementation details

There are five main subsystems:

* a language (an expression) compiler to VM byte-code
* a stack VM byte-code interpreter
* a user functions module
* a context manager
* a tick stream processor

### Language compiler

Source files: parser.h, parser.cpp, tokenizer.h.

The compiler is based on the Lemon parser (<http://www.hwaci.com/sw/lemon/>). Lemon is similar to bison and yacc. Lemon’s beneficial advantage is that it is a “passive” parser. The parser doesn’t invoke the tokenizer when it needs a next lexeme (as bison and yacc do). Instead the parser is being called to process a next lexeme when it’s ready. This approach allows logically to separate the parser and the tokenizer and also programmatically switch tokenizers on-the-fly. As a tokenizer GNU Flex is used. Also Lemon is designed such way that it doesn’t work with a global namespace. It’s possible to create as many independent instances of Lemon as required in one compilation unit and run them simultaneously in isolation. The last advantage of Lemon over bison and yacc is the grammar syntax. It’s clearer and more bug-free due to a terminal and non-terminal naming convention.

The Lemon’s grammar file is “parser\_grammar.y” (compiled “parser\_grammar.h” and “parser\_grammar.c”).

Flex tokenizer grammar file is “tokenizer.l” (compiled “tokenizer.yy.cc”).

### Stack VM byte-code interpreter

Source files: executor.h, executor.cpp, expression.h, expression.cpp

The expression is being compiled to the VM program. The VM has three commands: a constant (any double), a function call (@) and a symbol substitution (#). Arithmetic operations (+, -, /, \*) are also functions. For example, “(10 + eur) / 2 + moving\_avg(gbp, 10)” is compiled to:

“10” “#eur” “@+” “2” “@/” “#gbp” “10” “@moving\_avg” “@+”

Such approach allows adding new functions without changing grammar and can be done on-the-fly.

The program is being executed by a stack based interpreter.

### User functions module

Source files: function.h, function.cpp, runtime.h, runtime.cpp

This module contains user defined functions. New functions can be added without changing the grammar. Arithmetic operators (+, -, \*, /) are either implemented as functions.

### Context manager

Source files: tscalc.h, tscalc.cpp

The context manager ships the particular values of symbols (tickers) for every execution (evaluation) of an expression. For example, when the example (above) is being interpreted with the following context:

{ “eur”: 100.1, “gbp”: 105.5 }

It will be treated as:

“10” “100.1” “@+” “2” “@/” “105.5” “10” “@moving\_avg” “@+”

### Tick stream processor

Source files: tick\_stream.h, tick\_stream.cpp

The tick stream processor parses data files based on given a time frame and provides appropriate input to the context manager.

## Compilation

Visual Studio 2008 must be available to build the project. VS2005 or VS2010 can be also used. It requires modifications of the build scripts (see below) accordingly to set up the compiler environment properly.

The project is shipped with pre-compiled grammars of Lemon and Flex. The “generate-grammar.cmd” script can be used to re-generate the parser and the tokenizer grammars.

Note: The GNU Flex 2.5.35 has to be available to compile “tokenizer.l”. Lemon is compiled automatically by the build script from “lemon.c” or “lempar.c” files.

The “build-tests.cmd” script compiles the project unit test runner (including main modules as well). The unit test framework is Google Test (“./gtest/” subdirectory). This script produces “tscalc\_unittest\_runner.exe” file. The executable runs through all unit tests.

After the successful tests execution the “build-main.cmd” script can be used to build the target executable “tscalc.exe”.

Additionally “tscalc\_vs2008.sln” solutions is available to work on the project in IDE.

The sources are portable and should be ported to UNIX quite without major issues.

## The data generator

The “test\_data\_generator” directory contains a generator of test data.

## Further possible improvements

The domain specific language can be easily extended. New user functions can be added (in “function.h” and “function.cpp”) without changing the grammar.

A byte code produced by the compiler can optimized before the interpretation. For example, it’s possible to eliminate repeatable sub-expressions.