SimComp Macro Processor

Description

This project is a sub-project of the SimComp software toolkit. It is a macroprocessor for SimComp assembly language. The macroprocessor is a textual expander of macros defined in assembly source code file. It is the first component in a SimComp ASM build chain:

```
|infile| -> Macroprocessor -> Assembler -> |binary| -> Simulator
```

Usage example

Given a macro definition somewhere in the file:

```
ADD macro &a,&b,&result
    lda &a
    ldx &b
    add x
    sta &result
mend
```

the macro processor will expand any occurrence of `ADD fst,snd,res` into a macro definition with positional parameters substituted in appropriate places, for the example above the result would be:

```
    lda fst
    lds snd
    add x
    sta res
```

Features and boundaries

- Error handling is very weak for a moment.
- Recursive macro expansion is supported (macros can call other macros inside their body).
- Recursive macro definitions are not supported (one simply cannot declare a macro inside a macro).

Requirements

The macroprocessor is implemened according to **C++0x** standart and depends on **STL** library. The codebase was compiled and tested with `Linux + g++` and `Windows + VS2010`. Note that earlier versions of Microsoft Visual Studio may not be supported.

Build instructions

The macroprocessor is shipped with `CMakeLists.txt` file that is an input file for CMake utility. It can be used for generating both Makefiles and VCProjects for Linux and Windows environments. For Linux (or Cygwin) folks `build.sh` script will perform the building. The binary file is stored into the `bin/` directory. So, a Linux user should type

```
$ ./build.sh && ./test.sh && ./use.sh
```

to perform complete shipping cycle.

Processing given assembly file

Compiled macroprocessor can be used in the following way:

```
$ ./bin/macro infile.asm outfile.asm
```

Testing

No unit tests are provided. A simple end-to-end test script is used instead. It is called `test.sh` and is in a project root directory. It takes all files matching `*_in.asm` pattern in `fixture/` directory, runs macroprocessor against it and `diff`s the result with corresponding `*_expect.asm` file.

IMPLEMENTATION OF A MACROPROCESSOR

Macroprocessing is a two-pass procedure. Macro Definition Table (MDT) is generated on the first pass, Macro Expansion is performed on the second. Furthermore, declaration finder strips macro definitions from the source file so they are not present during macro expansion. The dataflow diagram of implemented macroprocessor is listed below:

```
 --------           ----------------         --------
| source |---+---->| Macro Expander |------>| output |
|________|   |     |________________|       |________|
             |            ^
             |            | MDT
             |      ______|______
             |     | Declaration |
             |---->|   Finder    |
                   |_____________|
```

Fig. 1. Macroprocessor dataflow diagram.

Project structure

The project is organised in a following way:

- `MacroExpander` - ME, a component that performs output code generation
- `DeclarationFinder` - finds macro declarations and fills the MDT.
- `DefinitionTable` - key data structure for managing macro definitions.
- `MacroDefinition` - handles the definition of a particular macro and provides the routines to expand itself with a list of arguments.
- `MacroProcessor` - wires the system up and provides user interface.

The choice of data structures

The most important data structure in a project is a macro definition table. It needs to suite for several conditions:

```
- Frequent reads.
- Non-frequent writes.
- Addressing by macro name.
- Storage of macro definition objects.
- Storage of unique elements.
- Unordered.
```

Facing the requirements described, `std::unordered_map<Name, MacroDefinition>` from STL is chosen as the one that handles all of recuirements properly. It is based on hash tables, having therefore **O(1)** complexity for reading.

MacroDefinition is the next data structure of importance; it is organized as follows:

```
MacroDefinition:
    name:     string   -- name of a macro
    argnames: [string] -- positional argument names of the macro
    body:     string   -- the body of the macro.

    string expand(argvalues: [string]) -- returns the expansion of the macro;
        'argvalues' is a list of arguments of the macro.
```