

Imagen

for Windows/Linux

Language Module Documentation

January 12, 2001. Document Revision 1

Netclime Inc.

P.O.Box 251666, LA, CA-90025
Tel (310) 571-3135 Fax (646) 514-0412

email: imagen@netclime.com

Responsibilities

Purpose

This module provides custom C like scripting language.

Provided Functionality

Simple C like script preprocessor and interpreter is provided as log as interface for built in functions, callable through the script.

Interface

Integration Instructions

Just run it and pass as first parameter the script file name, the next command line parameter are passed directly to the interpreter:

The format is:

`<interpreter executable name> <script file name> [<parameter n name> = <parameter n value>]*`

or:

`<interpreter executable name>?<script file name>[&<parameter n name> = <parameter n value>]*`

`<parameter n value>` - could be a simple expression

Required Headers and Header Paths

Required LIBs and LIB paths

Dependencies

C++ Objects

Common

Types (typedefs, structures, etc.)

struct config – interpreter configuration structure, loaded at start time from the configuration file dic.cfg.

struct lexeme – basic unit for scanner and parser, as it name says, it represents a lexeme, describing it's type, identifier, the source file and line where it appears. The scanner produces lexemes. The parser interprets them.

struct atoms – set of atomic tables containing keywords, identifiers and constant strings; it also contains an array of lexemes; Atom tables provide an easy way for translating source code to sequence of lexemes and recovering the original source code form a lexeme sequence.

The keywords atomic table is filled at startup time and is never changed afterwards. The other atomic tables are filled by the scanner in the process of translating the program source into a sequence of lexemes. The array of lexemes and the atomic tables then are passed to the parser, which parses and interprets them appropriately.

Constants

All keywords codes are defined in the `enum keyword_code`.

Functions

config& the_config() – function providing access to the single instance of the `config` class

Static Objects

atoms atom_context – the `atoms` instance passed between the scanner and parser.

Scanner

The Scanner implements the preprocessor of the language. Its result is list of lexemes to be passed to the parser.

Namespaces

All definitions are in namespace `scanner`.

Types (typedefs, structures, etc.)

struct processor

The processor is responsible for converting character stream to array of lexemes.

It receives several atom tables: for keywords, identifiers and strings. The keywords table is input while the others are output. Note that *Names* (as specified in the Language Reference) are returned either as keywords or as identifiers. When a name is read from the input it is checked for keyword, if not, it's assumed identifier.

The class also implements search tree for fast recognition of keywords stored as punctuation.

Public members

`processor(atoms &atom_tables, std::istream &source);`

Initializes the object. The keywords member of the atom table must be initialized.

```
~processor();
```

Releases resources allocated by the object.

```
int line_number;
```

This member specifies the line number of following lexemes. It is automatically incremented by new-lines in the stream.

```
const char *file_name;
```

This member is directly copied to new lexemes.

```
bool get(lexeme &);
```

The method retrieves one more lexeme from the input stream. Returns `true` on success, `false` on EOF and throws errors through one of the `message_XXX` routines.

```
void get_all(std::vector<lexeme> &list);
```

This helper reads all lexemes until EOF is found. The resulting lexemes are stored into the `list` argument.

```
inline std::string lexeme2string(const lexeme &) const;
```

This helper returns the string representation of a lexeme by using the atom tables.

Private members

```
inline int skip_whitespace();
```

This method skips all white space and comments measuring the line numbers. It returns the first character of the next lexeme or EOF.

```
inline void setup_puncts();
```

This method called by constructor. It initializes the punctuation search tree stored in the `puncts` array.

```
inline bool get_identifier(lexeme &, int ch);
```

Handles name stored in the stream. Returns lexeme that is either identifier or keyword.

```
inline bool get_number(lexeme &, int ch);
```

Handles integral number stored in the stream.

```
inline bool get_text(lexeme &, int ch);
```

Handles string literal stored in the stream.

```
inline bool get_punctuation(lexeme &, int ch);
```

Handles punctuation sequence in the stream.

struct `preprocessor`

The preprocessor object handles all work that a preprocessor has to do: By given file name it loads the file, converts it to lexemes (by using the `processor` class above) and processes preprocessor directive.

NOTE: The lexemes are valid only during the lifetime of the preprocessor object! After destroying the preprocessor, the `file_name` member of all lexemes becomes invalid.

Public members

```
std::vector<lexeme> lexemes;
```

This is the resulting array of parsed lexemes.

```
preprocessor(atoms &atom_tables, std::string file_name);
```

The constructor initializes the object and preprocesses the file name. Upon return the `lexemes` member is valid. On error, exception is thrown through one of the `message_XXX` routines.

Note that the `atom_tables` argument is read/write:

- ◆ The members `identifiers` and `strings` are updated with the new objects.
- ◆ The member `keywords` is extended to contain preprocessor keywords.
- ◆ Note that the member `lexemes` is NOT updated.

```
static std::string advance_include(std::string source_include, std::string new_include);
```

This helper finds the name of file being included by another file by using the source file name and the string passed to the include directive. The operation is performed according to the rules for file inclusion: the include directive specifies file name relative to the directory where the referring file resides.

Private members

```
std::vector<lexeme> include(std::string file_name);
```

This method does the whole work related to loading and preprocessing of single file. Include directives are handled by recursive self invocation. Self-including files are recognized and reported. It's called by the constructor.

```
inline void lock_include(std::string include_file);
```

Helper that records some file as included. Second inclusion of the same file (without corresponding `unlock_include`) generates "recursive include" error.

```
inline void unlock_include(std::string include_file);
```

This method removes the specified file from the list of currently included files.

```
struct include_life
```

This structure provides simple life-time interface to the `lock_include` and `unlock_include` methods.

Parser

The parser processes a list of lexemes and executes the program stored there.

Namespaces

All classes and functions are declared in the global namespace except the profiling ones. They are declared in namespace `profile`.

Types (typedefs, structures, etc.)

struct parser : private `lexeme_process`, private `restrictions` - this class performs the parsing and interpreting of the lexeme sequence, generated by the scanner from the program source code. The parser uses a method called recursion diving.

Public members:

parser(bool profiler_enabled) – object constructor; initializes parser - creates global scope and profiler (`profile::table`) if profiling is enabled; the parameter passed toggles profiling on or off.

~parser() – object destructor; cleanup parser

void parse_args() – member function for parsing lexeme sequences where the only statement allowed is assignment. The lvalue variable type is not initially specified, it's set to the type of rvalue in the assignment. The rvalue can be a simple expression. This member function is used for parsing the parameters passed to the script, it's also used for parsing the config file.

void program() – after calling this function, parsing and interpreting of the program begins;

inline std::string get_profile_nfo() – after program has finished, call this function to create human readable profile information, of course if profiling is enabled...

Private members:

Recursion diving functions:

void statement()

void block(bool create_scope=true)

variable *terminal_expression()

variable *unary_expression()

variable *multiplicative_expression()

variable *additive_expression()

variable *relational_expression()

variable *and_expression()

variable *or_expression()

inline variable *Lexpression() – logical expression

inline variable *Aexpression() – arithmetic expression

inline bool global_definition()

inline bool var_definition()

inline bool func_definition()

inline void return_statement()

inline void if_statement()

inline void while_statement()

inline void do_while_statement()

Other member functions:

inline bool is_type() – checks if current lexeme is type

inline void clear_stack(unsigned clear_size) - unrolls temporary variables stack to the needed size & frees the memory allocated by variables in stack

inline variable *get_actual_param() – method for retrieving an actual function parameter

inline variable *create_formal_parm() – method for creating formal function parameter; it's created in the current scope(the function scope)

inline void function_call(function *func, variable *ret_var, bool main=false) - calls the specified function

inline bool function_call_executed(variable **ret_var) - check for a function call and if its one - executes it

inline void execute_main() – calls program main function

inline bool func_param() – checks for function formal parameters

inline void count_braces(unsigned open=1, unsigned close=0) – method for counting braces – used when skipping statements

inline void skip_statement() – skips a statement

inline void skip_block_or_statement() – skips one block or statment

inline variable *create_variable(unsigned type, unsigned id) – creates a variable in the current scope

inline variable *clone_var(variable *var) - clones the variable passed and pushes the cloning into the temporary variable stack

scope *_global_scope – the global scope, created upon parser creation

scope *_current_scope – pointer to current scope, initially is set to the global scope

typedef std::map<scope*, scope*> scope_map

scope_map _scope_map – all created scopes are registered in this map upon their creation and unregistered just before their destruction. When parser is destroyed all scopes registered in the map are destroyed too, thus preventing scope memory leaking.

typedef std::map<unsigned, function*> function_map

function_map _functions – contains all functions found while parsing the program;

function_stack _func_stack – contains information for occurred function calls

variable_stack_temp_var_stack – contains the temporary variables needed to evaluate expressions

profile::table *_profiler – if profiling is enabled; the parser upon its construction creates the profiling table, which later is filled with profile entries

struct lexeme_process – class for navigation through lexeme sequence; it serves like a pointer in the lexeme array; when the pointer is moved a lexeme structure is filled with the value of the current lexeme form the lexeme array

Public members:

inline lexeme_process(atoms &atom_tables=atom_context) – constructor, initializes object, specifying atom tables to use(, the lexeme sequence is in the atom tables); by default is initialized with the global atom tables

inline unsigned curr_lex_index() const – member function returning the current lexeme position (“the pointer value”)

inline void curr_lex_index(unsigned new_index) – member function setting new index to the pointer in the lexeme sequence

inline void get_next_lexeme() – member function moving the pointer to the next lexeme in the sequence (“incrementing the pointer”)

Protected members:

lexeme _lex – current lexeme value is stored here for derived classes usage

atoms &_atom_context – reference to the atom tables and lexeme sequence; for derived classes ussage

struct end_of_script – empty notification structure thrown when end of script is found

struct restrictions – helper class for implementing parser restrictions – maximal iteration count & maximal recursion depth using. The class can’t be instantiated; only derived classes can use it.

Protected members:

restrictions() – constructor initializing object members

restrictions::counter_iterations – restriction counter, controlling allowed iteration count

restrictions::counter_recursive_func_calls – restriction counter, controlling allowed recursion depth

struct restrictions::counter – class implementing restriction counter. If restriction count is set to zero (0), restriction counting is not performed.

Public members:

inline counter(unsigned max_count, const char *err_explain) – constructor, specifying the maximal allowed count and explanation of the error when this count is reached

inline unsigned inc() – increase current count

inline unsigned dec(unsigned cnt=1) – decrease current count

struct variable – abstract class defining typeless variable interface.

Public members:

- virtual ~variable()** – virtual variable destructor
- virtual void assign(const variable *)=0** – variable assignment
- virtual operator bool() const=0** – cast to bool operator
- virtual bool equal(const variable *)=0** – checks if this variable is equal to the passed variable
- virtual bool not_equal(const variable *)=0** - checks if this variable is not equal to the passed variable
- virtual bool less(const variable *)=0** - checks if this variable is less than the passed variable
- virtual bool less_or_equal(const variable *)=0** - checks if this variable is less or equal to the passed variable
- virtual bool greater(const variable *)=0** - checks if this variable is greater than the passed variable
- virtual bool greater_or_equal(const variable *)=0** - checks if this variable is greater or equal to the passed variable
- virtual void add(const variable *)=0** – adds the passed variable value to the current variable value, the result is stored in the current variable value
- virtual void sub(const variable *)=0** - subtracts the passed variable value from the current variable value, the result is stored in the current variable value
- virtual void mul(const variable *)=0** - multiplies the passed variable value with the current variable value, the result is stored in the current variable value
- virtual void div(const variable *)=0** – divides the current variable value with the passed variable value, the result is stored in the current variable value
- virtual void mod(const variable *)=0** - divides the current variable value with the passed variable value, and stores the remainder in the current variable value
- virtual variable *unary_plus()=0** – does nothing
- virtual variable *unary_minus()=0** – negates the current variable value
- virtual variable *unary_not()=0** – performs logical not on the current variable value
- virtual variable *clone()=0** – creates a new variable, initialized with the current variable value

For typed variable behavior (, the interface is the same) see the templates section below.

Typed variables implementations:

typedef typed_variable<int> int_variable – integer variable implementation, defining specific integer variables behavior; All variable operations defined in variable interface are implemented;

typedef typed_variable<std::string> string_variable – string variable implementation, defining specific string variables behavior; All variable operations defined in variable interface supported, except:

```
inline void typed_variable<std::string>::sub(const variable *var)
```

```

inline void typed_variable<std::string>::mul(const variable *var)
inline void typed_variable<std::string>::div(const variable *var)
inline void typed_variable<std::string>::mod(const variable *var)
variable *typed_variable<std::string>::unary_minus()
variable *typed_variable<std::string>::unary_not()

```

typedef std::stack<variable*> variable_stack – variables stack class

struct scope – class implementing scope behavior. Variables are created in scopes.

Scopes can be chained in a list, thus allowing easy variable name resolution. Same variable names can reside in different scopes. Global scope is the first scope and is created at parser initialization and is set to be the current scope. The current scope is the first scope that a variable is looked for, if it's not found there the search continues in the next scope in the chain. When a new scope is defined, it is chained before the current scope, becoming the current scope.

Public members:

scope(scope *parent=NULL) – constructor, initializes object and specifies parent scope if any

~scope() – destructor, destroys all variables in the scope

void add_variable(unsigned index, variable *var) – inserts a variable in the scope

variable* find(unsigned index) – searches for variable in the current scope, if it's not found there the search continues in parent scopes

inline scope *get_parent_scope() const – returns the parent scope

struct function – class containing function information(where functions starts, function return type). It also creates `function::stack_info` used in function calls.

struct function_return - empty notification structure thrown when a return statement is found or function end is reached. (Can be used for passing return values, in this implementation return values are passed

struct function::stack_info – class for storing function call information, pushed in the function stack

typedef std::stack<function::stack_info *> function_stack – function stack, stores function calls info.

struct profile::table: `protected lexeme_process` – class for storing profile entries and generation human readable profile information.

Public members:

void append(entry &new_entry) – appends a new profile entry to the profile table

std::string create_humman_nfo() – form the profile table contents, creates human readable profile information

struct profile::entry – an element form profile table, containing time information and start and end positions in the lexeme array, indicating the profiled piece of code

Templates

template <class type> struct typed_variable: variable – template class defining common typed variable behavior

template <class restrict_cnt> struct lifetime – helper lifetime template class for usage with recursion depth restriction counter –

```
typedef lifetime<restrictions::counter> restriction_lifetime
```

template <class restrict_cnt> struct lifetime_loop - helper lifetime template class for usage with iteration restriction counter –

```
typedef lifetime_loop<restrictions::counter> loop_lifetime
```

Externals

The externals section provides mechanisms for registering run-time routines. Function names for all interfaces are atom ids obtained from the global `atom_context` variable.

This module exports the following interfaces:

- ◆ Parser Interface
- ◆ Registration Interface
- ◆ Registration Helpers

Parser Interface

This is the interface between the registered functions and the parser component.

Header file: `external.h`

Namespaces

All definitions are in namespace `external`.

Functions

```
bool function_exists(atom::index function);
```

This method queries for the existence of a function.

```
variable *call(atom::index function, const std::vector<variable *> args);
```

This method calls given function by passing function name and arguments. The return is either variable from given type or null pointer for `void` functions.

Registration Interface

This is the interface between the providers of registered functions and the externals component.

Header file: `external_reg.h`

Namespaces

All definitions are in namespace `external`.

Macros

REGISTER(function)

Shortcut to `register_function`

STUB(function)

Helper used for the image of external function.

IA(x), SA(x), CA(x)

Helper macros that return their argument (at position `x`) as either `int`, `string`, or `const char *`. If argument types do not match, error is issued.

ACNT(n)

This helper macro verifies that the correct number of arguments is provided.

Types

```
typedef variable *(* caller)(const std::vector<variable *> &args);
```

This is the prototype for external routine.

Functions

```
void register_function(const char *name, caller code);
```

Call this method to register external function.

Registration Helpers

This is additional interface for easier defining of new external routines.

Header file: `std_stub.h`

Macros

XSTUB_PROLOG, XSTUB_EPILOG

Redefine these to provide enter/leave code for your stubs. For example: to create custom exception handling. These macros are placed in the beginning/the end of the functions generated by `STD_STUB` below.

STD_STUB_SPACE

This is the namespace locating the function to call

STD_STUB(ret,name,args)

Note: `args` is in the form: `ARGn(type1,type2,...)`

This macro defines function with the given name and arguments in the current namespace. The produced function verifies the argument count to match the given one, converts the arguments to the specified types and invokes function with the same name (but in namespace `STD_STUB_SPACE`). The called function must have the same arguments and return type.

The arguments currently can be: `int` and `string`.

The return type can be also: `void`.

Example usage:

```
STD_STUB(int, func_name, ARG3(string, int, int))
```

Stubs

The built in routines currently available are divided into the following categories:

- ◆ Strings
- ◆ Graphics and Text
- ◆ Debug

Strings

String routines are located in `strings.cpp`.

Graphics and Text

Graphic routines are located in `magick.cpp`. This implementation uses ImageMagick for all operations related to the virtual display except for text output.

Text Output

Text output specific support is located in the following files:

<code>font_draw.*</code>	Generic interface for font renderers. Note: they render single glyphs only.
<code>freetype1.*</code>	Font renderer that uses the FreeType 1.3 library
<code>freetype2.*</code>	Font renderer that uses the FreeType 2.0 library
<code>antialias_simple.*</code>	Extension supporting 'simple' antialiasing method. Works on top of another font renderer.
<code>draw_text.*</code>	Engine for displaying text. The methods for alignment and clipping are implemented here.
<code>text_wrap.*</code>	Wrapping module.
<code>text2image.*</code>	Adapter between <code>draw_text</code> image format and ImageMagick's.

Debug

Debug routines are located in debug.cpp.

Examples of Usage

Common Usage

Show here the simplest and most obvious method of use of the module. Include essential source fragments in the text. Put complete example in the appendix.

Tricky Usage

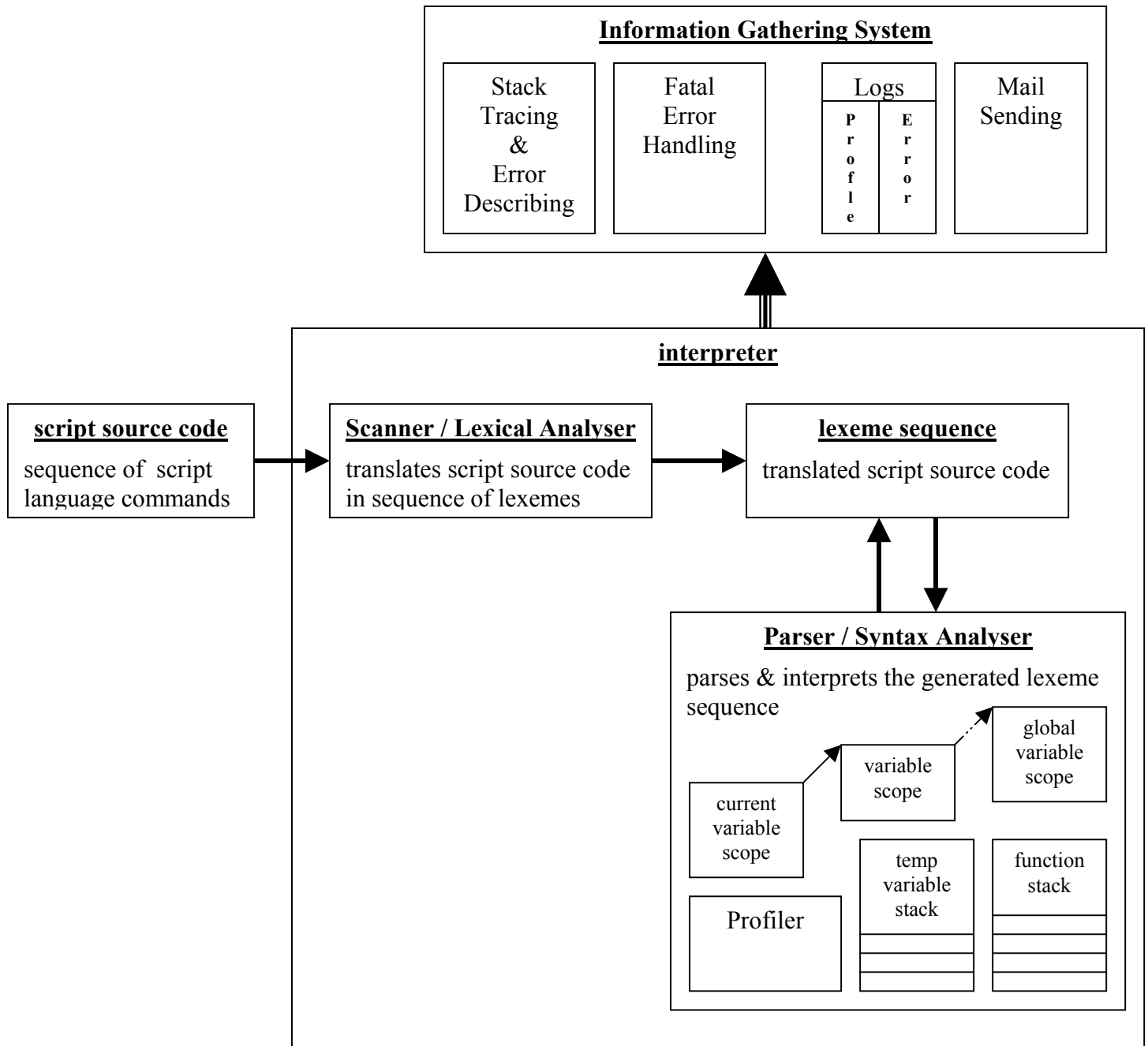
Show here the trickiest and not obvious method of use of the module. Include essential source fragments in the text. Put complete example in the appendix.

Design

Concept

Explain here all abstractions in the form of ideas methods and algorithms on which the module operation is based.

Ideas



Methods

Algorithms

Technologies

Structure

Overview of Internal Modules and Their Responsibilities

Scanner

The Scanner implements the preprocessor of the language. Its result is list of lexemes to be passed to the parser.

Parser

The parser processes a list of lexemes and executes the program stored there.

Externals

The externals section provides mechanisms for registering run-time routines.

Stubs

The built in routines currently available are located here.

Data Flow and Control Flow Diagram

State here the stages through which the data processing goes. Show dispatching of control between objects/functions.

