[JGRASS] Console Core

Architectural Concept

(Introduction)

Andreas Hamm

November 19, 2007

Table of Contents

Chapter 1

1 Introduction

The provided information by this documentation is to be seen as an addition to the source code documentation and as a general introduction of the principles and techniques used to realize the *JGRASS* ***console engine***, which is primarily responsible for the execution of *GRASS* native based and of *JGRASS* Java based, *OpenMI – Open Modelling Interface* – compliant commands with the ability to execute one or many of them in a script file, formally known as *batch-mode*.

Simply stated, the objective fact that JGRASS and GRASS, each of them, are following their self-defined syntax and grammar definition – sometimes commonly referred to as *model language* – which have nothing in common, inspired to realize the console engine based upon the concept of a translator or more precisely of a language processor.

This preliminary chapter is an introduction of the concept of a translator, or more precisely pre-processor because this the key. It serves as an introduction to the rest of the documentation.

1.1 Translation, Interpretation and Pre-processing

A language processor is a special kind of a *translator*:

Definition 1.1 (Translation)

A *compiler* is a software system respectively a computer program, which can read a program in one language – the *source* language – and translate it into an equivalent program in another language – the *target* language.

Typically a language processor performs many or all of the following operations: pre-processing, lexical analysis, parsing (syntax analysis, semantic analysis), code generation, code optimization.

Definition 1.2 (Interpretation)

An *interpreter* is another common kind of a language processor. Instead of producing a *target* program as the result of the translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Definition 1.3 (Pre-processing)

A *pre-processor* is a an entrusted, separate program used by compilers to a) extend the syntax and grammar definition of a programming-language with abbreviations, sometimes referred to as *macros*, for complex source programming constructs, and again others b) have the power of fully-fledged programming languages. *Macro expansion* is probably the most powerful feature of a pre-processor: The pre-processor expands the macro with its definition into one or many programming constructs based upon the source language and then it replaces the macro with the produced programming constructs. The modified source program is then fed to a translator.

1.2 Compiler

A compiler maps source code, programming constructs of a source program, into semantically equivalent target code. The mapping is roughly divided into two parts: analysis and synthesis – the part analysis is often called the *front end*; the synthesis part is often called the *back end*.

Definition 1.4 (Analysis)

The *analysis* breaks up the source program into its constituent pieces and imposes a tree-like structure on them, which a) depicts the grammatical structure of the source program and is then used to create a more abstract tree-like intermediate representation of it or b) creates immediately the tree-like abstract intermediate representation.

If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it reports informative messages and the user can take corrective actions. The analysis part also collects information about the source code and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

Definition 1.5 (Synthesis)

The *synthesis* part constructs the desired target program – code generation – from the intermediate representation and the information in the symbol table.

Definition 1.6 (Symbol table)

Symbol tables are data structures used to hold information about source program constructs. The information is collected incrementally during the analysis phases (syntactical, semantically) and is used by the synthesis phases to generate the target code. Entries in the symbol table contain information about identifiers. The information about an identifier may consists of its character string, its type, and any other relevant information.

1.3 Lexical Analysis

The first phase of the analysis part is called *lexical analysis* or *scanning*. The main task of the lexical analyzer is to read the stream of characters making up the source program, groups them into meaningful sequences – called *lexemes* – and produces for each of them as output a *token*. The stream of tokens it passes on to the subsequent phase, the *syntax analysis*.

Tokens

A token is the smallest element of a source program that is meaningful to a language processor. A *token* typically consists of a token name, an attribute value and the information, where the token has been found in the source program – the line number. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword – sometimes referred to as reserved word – or a sequence of input characters denoting e.g. an identifier, an operator or a delimiter.

Patterns

A pattern describes the form of a lexical unit that the lexemes of a token may take. In the case of a keyword, the pattern is just the sequence of characters that form the word, and then again for other tokens, a more complex structure that is matched by many strings.

Lexemes

A *lexeme* is a sequence of input characters that matches the pattern of a lexical unit for a token.

1.4 Syntax and Semantic Analysis

The second phase of the analysis part is *syntax analysis* or *parsing*. The parser uses syntactic units – *tokens* – to create a tree-like intermediate representation that depicts the grammatical structure of the obtained token stream from the lexical analyzer. A typical representation of the grammatical structure of a source program is a syntax tree in which each interior node represents an *operator* and the leaf nodes respectively the children of the node represents the arguments of the operator – the *operands*.

After the syntax analysis, the syntax-tree is passed to the subsequent phase, the *semantic analysis*. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency and also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate code generation.

1.5 Code Generation

In the process of translating a source program into target code, a translator typically construct one or more intermediate representations. Syntax trees are a form of an intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many translators generate an intermediate representation of the target code, which is easy to produce and easy to translate into the target language. The code generator takes then as input the intermediate representation of the target program and maps it into the target language.

Chapter 2

2 The Console Engine of JGRASS

This chapter introduces the compiling techniques and a short how to use the engine.

2.1 Use of the Engine

The engine is really easy to use. A simple call of the method `dispatch` provided by the `JGrass` interface and a command, model or a script file is being immediately executed or only compiled, if specified.

In general, the developer of a console uses one `ConsoleEngine` object during the life-cycle of the program. Additionally, the console application must provide the user preferences and map them into a `ProjectOptions` object, each time the engine is used to compile or execute a command, model or script file.

2.2 The Engine

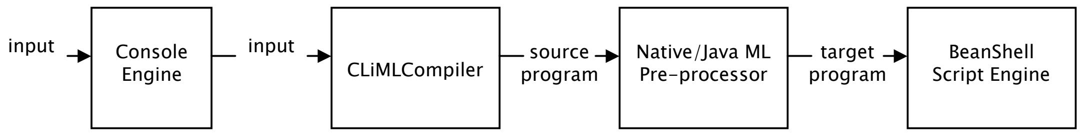
It is expected that any statement (command, model or the name of a script file) is passed through the `dispatch` method of the console engine object, because of, only the `ConsoleEngine` object provides the mechanism to execute or compile more than one concurrently.

Simply stated, the engine delegate the process of translation and execution to a entrusted so called "Command Line interpreter / Model Language Compiler" – *CLiMLCompiler* or compiler for short. For each statement passed through the `dispatch` method, the engine creates a project space (`Projectspace`) object that holds, e.g., the declarations of native based commands and Java based models, the directory of source script files and any other required information needed by the compiler to perform a compiler/interpreter run in its own thread.

However, the compiler treats the commands and models as keywords which can be setup in separated XML-files, additionally requires the information about input/output linkable components and for an interpreter run also their full qualified names – the engine parses all XML-files of a specified include directory for the desired information about commands and models.

2.3 Command and Model Processing

The so called "Command Line interpreter / Model Language Compiler" - CLiMLCompiler or compiler for short - controls the command line analysis and the subsequent translation phases; for the subsequent execution phase, a third-party script engine is being in use, unless that phase can be skipped, in the case of the command line should only be compiled - **/compile** directive.



Phases of the command and model processing

Facing the fact that the input stream is ambiguous - at one hand it can be a source program and at the other a GRASS command or JGRASS model - the compiler assumes a source program respectively the source code of a script file in its input - the compiler expects a **jgrass** or **grass** directive; if none was found the compiler treats the input as a statement of a command line assuming a JGRASS model in its input and attempts to translate a JGRASS model statement, if this also fails the compiler then attempts to translate a GRASS command. However, the command line will be executed in any case, except the command line should only be compiled.

CLiMLCompiler

The CLiMLCompiler is not a translator as expected. The so-called compiler only searches in the source program a **jgrass** or **grass** directive, then reads the content of the directive and delegate it either to the pre-processor that processes a model statement or to pre-processor that process a native command. The CLiMLCompiler then uses the output produced by the pre-processor to replace the directive in the source program.

The CLiMLCompiler uses a third-party script engine, the BeanShell script engine, to execute the source program.

ML pre-processor's

BeanShell is a Java compatible scripting engine - that means the scripting language of BeanShell is almost the same as the Java language, for that reason the pre-processors have the postfix *4j* in their class name.

JavaML4j pre-processor

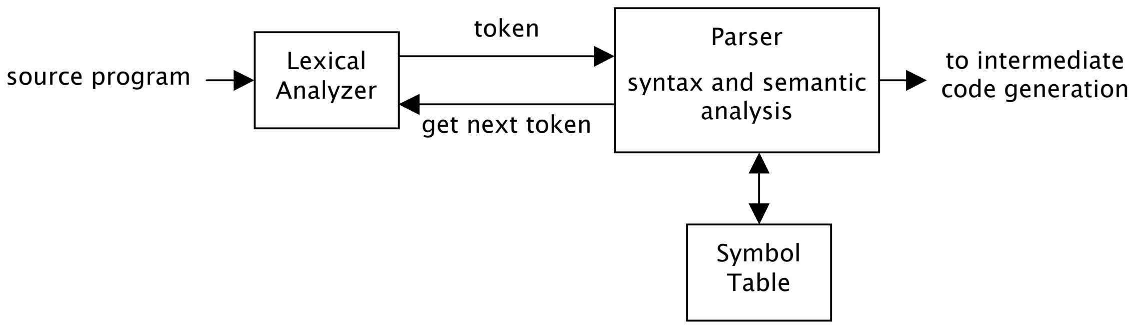
The JavaML4j translates a model statement: lexical analysis, syntax and partial semantic analysis, intermediate code generation, code generation of the target language.

NativeML4j pre-processor

The JavaML4j translates a native command: lexical analysis, syntax and partial semantic analysis, intermediate code generation, code generation of the target language.

2.4 Analysis

A parser of the JGRASS console engine analysis the syntax and semantic at once and then uses the tokens to create a tree-like intermediate representation. Instead of creating a syntax tree that depicts the grammatical structure, it creates a tree-like intermediate representation that depicts a abstract component structure.



Interactions between the lexical analyzer and the parser

The subsequent phase, the intermediate code generation, uses the tree-like intermediate representation produced by the parser and translate it into a equivalent syntax tree that depicts the programming constructs of the target language. The syntax tree is passed to the code generator.

Definition of Grammars

This section introduces a notation – called the "context-free grammar", or "grammar" for short – that is used to specify the syntax of a language. A context-free grammar has four components:

Definition 2.1 (Terminal)

A set of *terminal* symbols, sometimes referred to as "tokens". The terminals are the elementary symbols of the language defined by the grammar.

Definition 2.2 (Non-terminal)

A set of *non-terminals*, sometimes called "syntactic variables". Each non-terminal represents a set of strings of terminals, in a manner which has to be described.

Definition 2.3 (Production)

A set of *productions*, where each production consists of a non-terminal, called the *head* or *left* side, followed by an arrow and a sequence of terminals and/or non-terminals, called the body or right side.

A grammar is specified by listing their productions.

Definition 2.4 (Start symbol)

A designation of one of the non-terminals as the start symbol.

JGRASS Syntax Definition

This section is an introduction to the *JGRASS* and *GRASS* syntax and grammar definitions.

model_language	→	**jgrass** block
block	→	{ statement }
statement	→	java_model expressions
	|	ϵ
java_model	→	variable = java_model_identifier*
	|	variable = java_model_identifier
	|	java_model_identifier*
	|	java_model_identifier
expressions	→	expressions$_1$ output_expression nested_statement
	|	expressions$_1$ input_expression nested_statement
	|	expressions$_1$ expression’
	|	expressions$_1$ expression
	|	ϵ
expression’	→	output’
	|	input’
expression	→	argument
	|	output
	|	input
nested_statement	→	[java_model expressions expression’ expressions]
argument	→	argument_flag literal
	|	argument_flag variable
	|	argument_flag word
	|	literal
	|	--**help**
	|	variable
	|	word
input	→	input_expression literal
	|	input_expression --**help**
	|	input_expression variable
	|	input_expression word
input’	→	input_expression *
output	→	output_expression literal
	|	output_expression --**help**
	|	output_expression variable
	|	output_expression word
output’	→	output_expression *
argument_flag	→	--word
input_expression	→	--input_identifier*-quantity_identifier
	|	--input_identifier-quantity_identifier
output_expression	→	--output_identifier*-quantity_identifier
	|	--output_identifier-quantity_identifier

variable	→	$*variable_name*

GRASS Syntax Definition

model_language	→	**grass** *block*
block	→	{ *statement* }
statement	→	*native _model expressions*
	|	*native _model*
	|	ϵ
expressions	→	*expressions*$_1$ *expression*
	|	ϵ
expression	→	*literal*
	|	**--help**
	|	*variable*
	|	*word*
native_model	→	*native_model_identifier*

Tokens

In this section: the token descriptions, their associated pattern (regular expressions), and the token name respectively the token identifier.

Comments

A comment is text that a language processor ignores. Comments are normally used to annotate code for future reference. The lexical analyzer treats them as white-space.

A comment is written in one of the following ways:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters to terminate this form of comment – commonly called a “multi-line comment”.
- The // (two slashes) characters, followed by any sequence of characters. A new line in the source program terminates this form of comment. Therefore, it is commonly called a “single-line comment”.

The comment characters (/*, */, and //) have no special meaning within a character constant, a string literal, or a comment. Comments using the first syntax, therefore, they cannot be nested.

Token	Pattern / Regular Expression (Lexeme)	Token name/identifier
multi-line comment	([\u002f][\u002a])(.)*([\u002a][\u002f])	COMMENT_BLOCK
single-line comment	([\u002f][\u002f])(.)*([\r]?[\n])	COMMENT_LINE

Keywords

Keywords are predefined reserved identifiers – sometimes referred to as reserved words – that have special meanings. They cannot be used as identifiers in the source program.

Token Pattern / Regular Expression (Lexeme)	Token name/identifier
compile directive /compile	DIRECTIVE_COMPILE
grass directive grass	DIRECTIVE_GRASS
jgrass directive jgrass	DIRECTIVE_JGRASS
R directive R	DIRECTIVE_R
usage directive --help	DIRECTIVE_USAGE
JGRASS argument flag --[a-z0-9]+	ARGUMENT
JGRASS input/quantity identifier, argument flag --i[a-z0-9]+[*]?-[a-z0-9]+	INPUT
JGRASS java based model identifier [a-z][\u002e][a-z0-9\u002e]+[\u002a]?	JAVA_MODEL
GRASS native command identifier [a-z][\u002e][a-z0-9\u002e]+[\u002a]?	NATIVE_MODEL
JGRASS output/quantity identifier, argument flag --o[a-z0-9]+[*]?-[a-z0-9]+	OUTPUT
external variable identifier [\u0024][a-zA-Z_][a-zA-Z0-9_]+	VARIABLE

Other Tokens

A so called further token e.g. is a token that identifies an identifier, an integer number, a floating point number, a string literal or a word literal.

Token Pattern / Regular Expression (Lexeme)	Token name/identifier
integer number, floating point number, string literal [\"].*[\"] [\'].*[\'] [0-9]+[\u002e][0-9]+ [0-9]+	LITERAL
pathname ([\\w[\\u005c]/]+[:]?[\\w[\u002e][\\u005c]/]*)?([\\w[\u002e]]+[\u002e]{0}) [\"]([\\w[\\u005c]/]+[:]?[\\w[\u002e][\\u005c]/]*)?([\\w[\u002e]]+[\u002e]{0})[\"]?	PATHNAME

unknown [.]*	UNKNOWN
word literal [a-zA-Z0-9_]+	WORD

Operators

Operators specify an evaluation to be performed: all of them are following a strict precedence which defines the evaluation order of expressions containing these operators. Operators associate with either the expression on their left or the expression on their right.

Token Pattern / Regular Expression (Lexeme)	Token name/identifier
assignment operator =	CHARACTER_ASSIGN
asterisk operator [\u002a]	CHARACTER_ASTERISK

Delimiter

Tokens are delimited (bounded) by other tokens, such as operators and punctuation.

Token Pattern / Regular Expression (Lexeme)	Token name/identifier
start model statement block {	CHARACTER_BRACE_OPEN
end model statement block }	CHARACTER_BRACE_CLOSE
begin nested model statement [	CHARACTER_BRACKET_OPEN
end nested model statement]	CHARACTER_BRACKET_CLOSE
ignored; reserved for future use (	CHARACTER_PRENS_OPEN
Ignored; reserved for future use)	CHARACTER_PRENS_CLOSE

White-space

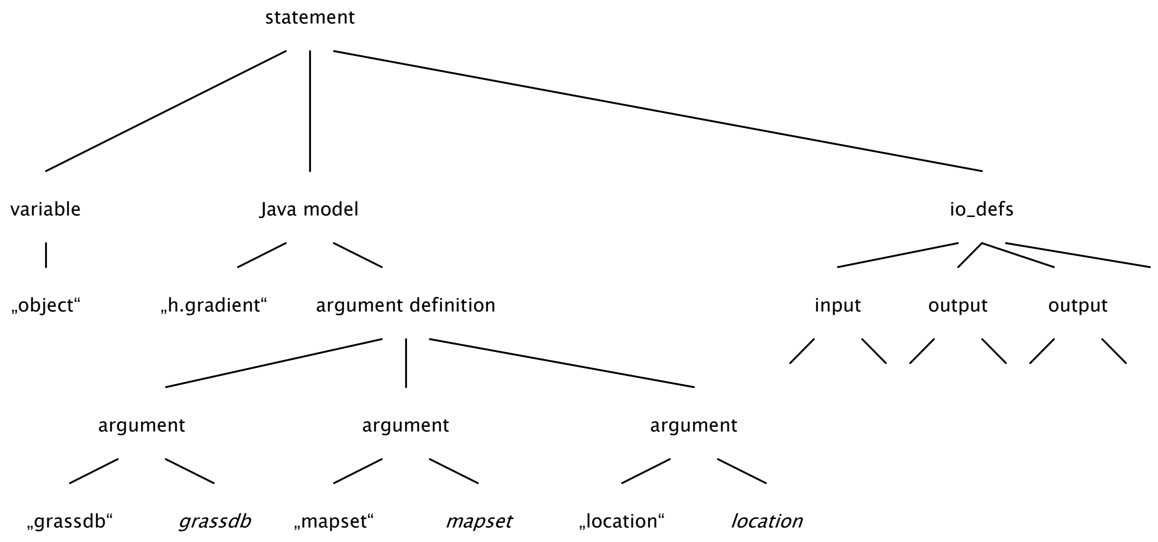
Tokens are delimited (bounded) by white-space characters and by other tokens, such as operators and punctuation.

Token Pattern / Regular Expression (Lexeme)	Token name/identifier
backspace character \u0008	CHARACTER_BACKSPACE

blank, character space \u0020	CHARACTER_BLANK
carriage return character \r	CHARACTER_CARRIAGE_RETURN
form feed character \f	CHARACTER_FORM_FEED
line feed character \n	CHARACTER_LINE_FEED
new-line [\r]?[\n]+	NEW_LINE
horizontal tabulator character \t	CHARACTER_TABULATOR

The Abstract Parse Tree

The parser uses the productions of the grammar to create a tree-like intermediate representation – an abstract parse tree, which has nothing in common with formally known parse or syntax trees.



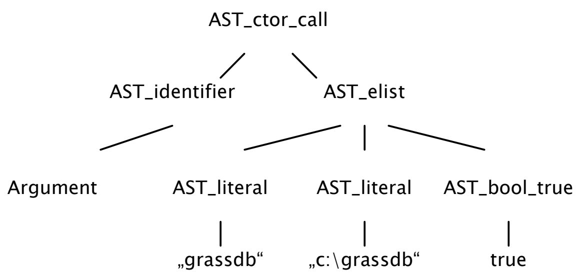
A sequence of abstract parse tree for a model

2.5 Intermediate Code Generation

In the process of translating a source program into target code, a translator may construct one or more intermediate representations, which can have a variety of forms. Syntax trees, in our case the abstract parse tree, are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many translators generate an explicit intermediate representation, which is easy to produce and easy to translate into the target language.

The intermediate code generation, uses the tree-like intermediate representation produced by the parser and translate it into a equivalent abstract syntax tree that depicts the programming constructs of the target language. The syntax tree is passed to the code generator.

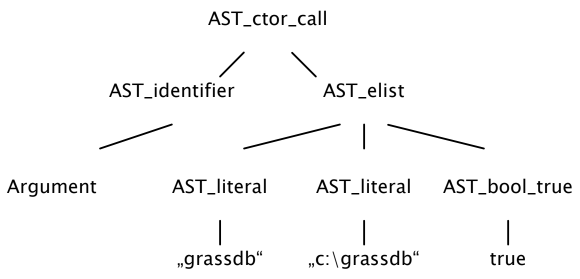


Sequence of an abstract syntax tree

2.6 Code Generation

The code generator takes as input a syntax tree of the target program and maps it into the target language. The mapping is done by the pre-processors of the CLiMLCompiler, namely by the JavaML4j or by the NativeML4j pre-processor.

Example:



Sequence of an abstract syntax tree

According the abstract syntax above, the code generator of the pre-processor generates the following target code:

```
    new Argument(“grassdb”,”c:\grassdb”,true)
```