Compiler Compiler System

Aleksandr F Urakhchin

2015.06.07

This document describes C++ version of "Compiler Compiler System with Syntax-Controlled Runtime and Binary Application Programming Interfaces" protected by US Patent 8,464,232 having the same title.

Contents

Introduction

The first Unix implementation had YACC, compiler compiler [1]. There are versions of C and C++ language compilers based on YACC. I have M.S. in Applied Mathematics and Computer Science from Moscow Institute of Physics and Technology, Moscow, Russia, http://mipt.ru/, since 1979. Also I have Ph.D. in Computer Science from Institution of Russian Academy of Sciences Dorodnicyn Computing Centre of RAS, Moscow, Russia, http://www.ccas.ru/index-e.htm, since 1984. During those years at schools from 1973 to 1982 (I graduated Computing Centre in 1982 getting Ph.D. two years later) I was learning different Computer Science subjects including compilers (see classic book [2]). I still remember my first impression about bootstrapping method and compiler compiler based on bootstrapping: it is not just "amazing", "cool", etc..., I do not have simple words for it, all this document is to explain it by introducing my own C++ compiler compiler system and related US patent [3].

In 80s and 90s I implemented few compilers for specialized programming languages including my Ph.D. thesis. In December 1996 I moved to US getting US citizenship in December 2006. In September 2010 I was laid off. Around that time a friend of mine told me his story. He said that his company just bought Workflow Engine from some company with a license price of $500,000 per installation. He said he could implement a similar Workflow Engine by himself including everything except front-end; that is a custom XML compiler for industry Workflow Standard defining that custom XML. I remarked to him that I had implemented a lot of compilers for specialized languages including my Ph.D. thesis in the 80s and 90s. Moreover, to automate the process of compiler development, in 1991, I created a compiler compiler system written on C programming language. After being laid off, the first thing I did, was, I wrote a patent. After finding the proper patent attorney, I spent the end of 2010 polishing the patent's text, and furthermore, the patent was submitted on Dec 27, 2010. While I was working on the patent in 2010, I also implemented from scratch a C++ Compiler Compiler System which is equivalent to the old one that I had implemented in 1991 as C Compiler Compiler System. In June 2013, it took me four days working with USPTO Patent Examiner to finish the patent application modifications that he had requested. As a result, US patent 8,464,232 was granted to me on June 11, 2013 [3]. So, this document is all about C++ Compiler Compiler System and the Patent [3]. In this document I will refer the C++ Compiler Compiler System, 2010, as C++ CCS; and C Compiler Compiler System, 1991, as C CCS.

Chapter "US 8,464,232 patent summary" describes main ideas of the Patent[3] that has a canonical structure: Abstract, List of Figures/Drawings (mainly UML diagrams), Background of the Invention, Summary of the Invention, Brief description of the Drawings, Detailed description of the invention, and Claims. In that Chapter some Patent details are omitted to be an introduction to C++ CCS.

Chapter "Parsing Model" introduces Parsing Model, the new conception defined by the Patent[3] and implemented by C++ CCS and C CCS. Just want to emphasize here, parsing tree, [1], [2] as a core data structure for representing parsing results is not used by C++ CCS or C CCS, it is replaced by Parsing Model with related Syntax-Controlled Runtime and Binary Application Programming Interfaces [3].

Chapter "Source Grammar Definition Language" defines C++ CCS (and also C CCS) input language, that is modified Backus-Naur Form (BNF) defined that way to be able to support so called LL(1) context-free grammars with some custom actions and grammar building blocks such as iteration, alternative, optional element, etc...

Chapter "C++ CCS overview" is a brief explanation of C++ CCS files and their role.

Chapter "C++ CCS Management" is a is a reference for C++ CCS Management classes.

Chapter "C++ CCS Generator" is a is a reference for C++ CCS Generator classes.

Chapter " C++ CCS Syntax-Controlled Runtime API" is a reference for C++ CCS Syntax-Controlled Runtime API classes.

Chapter " C++ CCS Syntax-Controlled Binary API" is a reference for C++ CCS Syntax-Controlled Binary API classes.

Chapter "C++ generated code library" is a reference for a code that is generated from input SGDL specification.

Chapter "Use Case: XML compiler" contains use case implementing XML compiler based on C++ CCS.

Chapter "Use Case: JSON compiler" contains use case implementing JSON compiler based on C++ CCS.

Chapter "Obfuscation" contains two use cases: text file and account number. In both cases a conversion from original source into Syntax-Controlled Runtime is presented. An obfuscation process is discussed as a grammar transformation process performed formal way using Syntax-Controlled Runtime and Binary APIs. Some simple obfuscation operations are presented.

US Patent 8,464,232 summary

A compiler compiler system with a design paradigm different from traditional compiler compiler systems in many aspects is defined by the Patent [3]. First, instead of parsing tree, compiler compiler runtime and binary are designed according to compiler compiler parsing model. Second, any semantics processing is totally separated from syntax processing. Third, the whole compilation process is defined as syntax processing and semantics processing followed by syntax processing performed under compiler compiler management supervision. Fourth, syntax processing has two phases: building compiler compiler runtime, and converting compiler compiler runtime into compiler compiler binary with available option to convert back compiler compiler binary to compiler compiler runtime. Fifth, compiler compiler runtime and binary syntax-controlled APIs are defined in terms of syntax. Sixth, there are formal methods de-compiling compiler compiler runtime and/or binary into original program text accordingly to syntax. Seventh, compiler compiler runtime and binary with their syntax-controlled APIs serve as a multiplatform for obfuscation, security, binary files processing, and program-to-program communication.

Background of the Invention

The present invention relates to information technology, and more specifically, to the generation of source code from formal descriptions that can be compiled and linked, creating an executable program.

In applications performing compiler constructions those formal descriptions are defined in terms of context-free grammars. Such tools take a definition of context-free grammar usually in Backus-Naur Form (BNF) and generate a source code of compiler components; and that compiler is able to process source code according to the input grammar definition. Such tools are called compiler compilers or compiler generators. One of the earliest and still most common form of compiler compiler is a parser generator [1]. In this case, the compiler compiler takes a grammar definition and generates parser source code. Also, a traditional compiler compiler generating parser source code from a grammar specification has another component that takes regular expressions and generates a tokenizer capable of processing specified tokens as a sequence of characters.

Compiler compilers as parser and tokenizer generators have been implemented since the late 1960's.

When a generated parser executes its actions during parsing a source program in accordance with a language grammar, it builds some form of parsing tree [2]. A developer who implements a compiler based on conventional compiler compiler technology is responsible for writing code in terms of a parsing tree and some attributes assigned to parsing tree nodes.

An ideal compiler compiler is supposed to take an input grammar specification and generate source code in automatic mode without any further manual procedures. Unfortunately this is far from what current information technology can offer in compiler compiler products available for developers.

Summary of the Invention

According to one embodiment of the present invention, a compiler compiler system includes compiler compiler executable program, compiler compiler management, compiler compiler runtime, compiler compiler binary, compiler compiler generator, compiler compiler source grammar definition language, and compiler compiler parsing model.

In another embodiment of the present invention, parsing results processing is totally separated from any subsequent semantics processing; parsing results are not represented in the form of any parsing tree; and parsing results are represented in the form of compiler compiler runtime that can be formally converted into/from compiler compiler binary.

In another embodiment of the present invention, target language defined in compiler compiler source grammar definition language as a source file is compiled by compiler compiler executable program, and a target parser and related source files are generated totally automatically providing to the target compiler a basic set of compile and de-compile operations.

In another embodiment of the present invention, the compiler compiler runtime and binary have a common compiler compiler parsing model that considers parsing results to be represented in the form of entities such as Context, Name, Symbol, and Rule and their relationships. That model itself is another embodiment of the present invention, i.e., an alternative view to a traditional parsing tree. The compiler compiler runtime implements the model in efficient form required by parser performing create, update, search, and other low level operations in terms of Context, Name, Symbol, and Rule classes and their relationships. The compiler compiler binary implements the model in efficient form providing read only operations having all data allocated in a vector of Tag instances but still is logically equivalent to the compiler compiler runtime.

In another embodiment of the present invention, a compiler compiler binary is a multiplatform data exchange protocol that allows interaction between programs running on different platforms, running on different operating systems, and built using different programming languages such as C, C++, C#, Java, Objective C, Ruby, Pyton, Perl, and others.

In another embodiment of the present invention, a compiler compiler system can be used for binary files processing. In this case, a corresponding binary file format has to be designed in the form of compiler compiler source grammar definition language specification. After that, a custom convertor from binary file format into compiler compiler runtime format is implemented. Having a compiler compiler runtime built for a binary file with a given format allows all other compiler compiler phases to work automatically without any extra code development.

In another embodiment of the present invention, an input compiler compiler binary can be formally transformed using a transformation algorithm into/from an output compiler compiler binary providing generic operations for applications such as obfuscation, security/access control, and content management. The transformation algorithm may be any suitable transformation algorithm typically used for obfuscation, access control, content management, or any other suitable algorithm as appropriate to the application. Consider a credit card account number represented in compiler compiler source grammar definition language as a sequence of digits, each of them represented as an integerToken. Then at the compiler compiler binary level the account number will be represented as a sequence of integerToken tokens. A simple obfuscation algorithm can transform those integerToken token values into different values in the output compiler compiler binary before transmitting output compiler compiler binary over the wire, protecting the account number, with subsequent de-obfuscation on the receiving side. Those algorithms can be changed dynamically also. For security control, compiler compiler binary can be transformed by adding user information with subsequent validation. Consider an audio or video file to be converted into compiler compiler binary. After that conversion, any custom transformations are possible including security control and obfuscation for content protection. Also, any kind of advertising incorporated into compiler compiler binary is possible with programmed options enabling/disabling ads.

In another embodiment of the present invention, compiler compiler management, generator, runtime and binary source code are compiled into a compiler compiler foundation library that is used in compiler compiler executable program and any other target compiler built by means of a compiler compiler system.

In another embodiment of the present invention, for any compiler compiler source grammar language description, the compiler compiler executable program generates code that is compiled into a generated library and a target compiler executable program is compiled and built with the compiler compiler foundation library and generated library. Built this way, the target compiler executable program has default compile and de-compile operations ready.

All semantics processing can be done as independent subsequent operations implemented using a compiler compiler binary application programming interface (API).

In another embodiment of the present invention, the compiler compiler source grammar definition language is defined using the same source grammar definition language that allows a bootstrapping method to be used for implementing compiler compiler executable program using itself. In other words, the compiler compiler generated code for a given meta grammar description is compiled into a generated library and a newly created compiler compiler executable program is compiled and built with compiler compiler foundation library and generated library. Built this way, the newly created compiler executable program has compile and de-compile operations ready. All semantics processing in this case is done as a code generation for parser and related generated source files implemented using compiler compiler binary API. If the abbreviation CCSGDL stands for compiler compiler source grammar definition language, then meta grammar is CCSGDL for CCSGDL.

Brief Description of the Drawings

This chapter contains drawings/figures (mainly UML diagrams) with their brief descriptions.

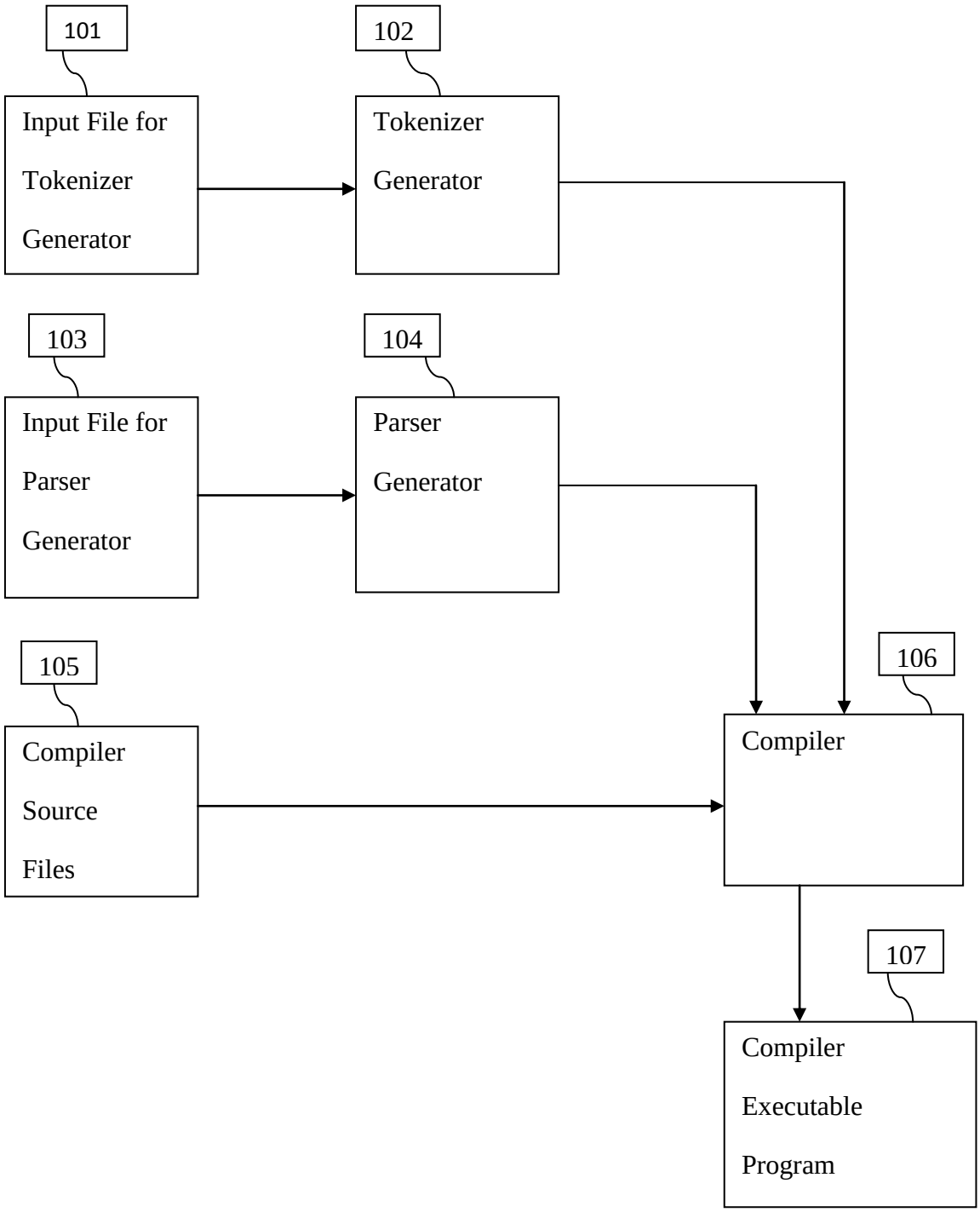
101
Input File for Tokenizer Generator
102
Tokenizer Generator
103
Input File for Parser Generator
104
Parser Generator
105
Compiler Source Files
106
Compiler
107
Compiler Executable Program

FIG. 1

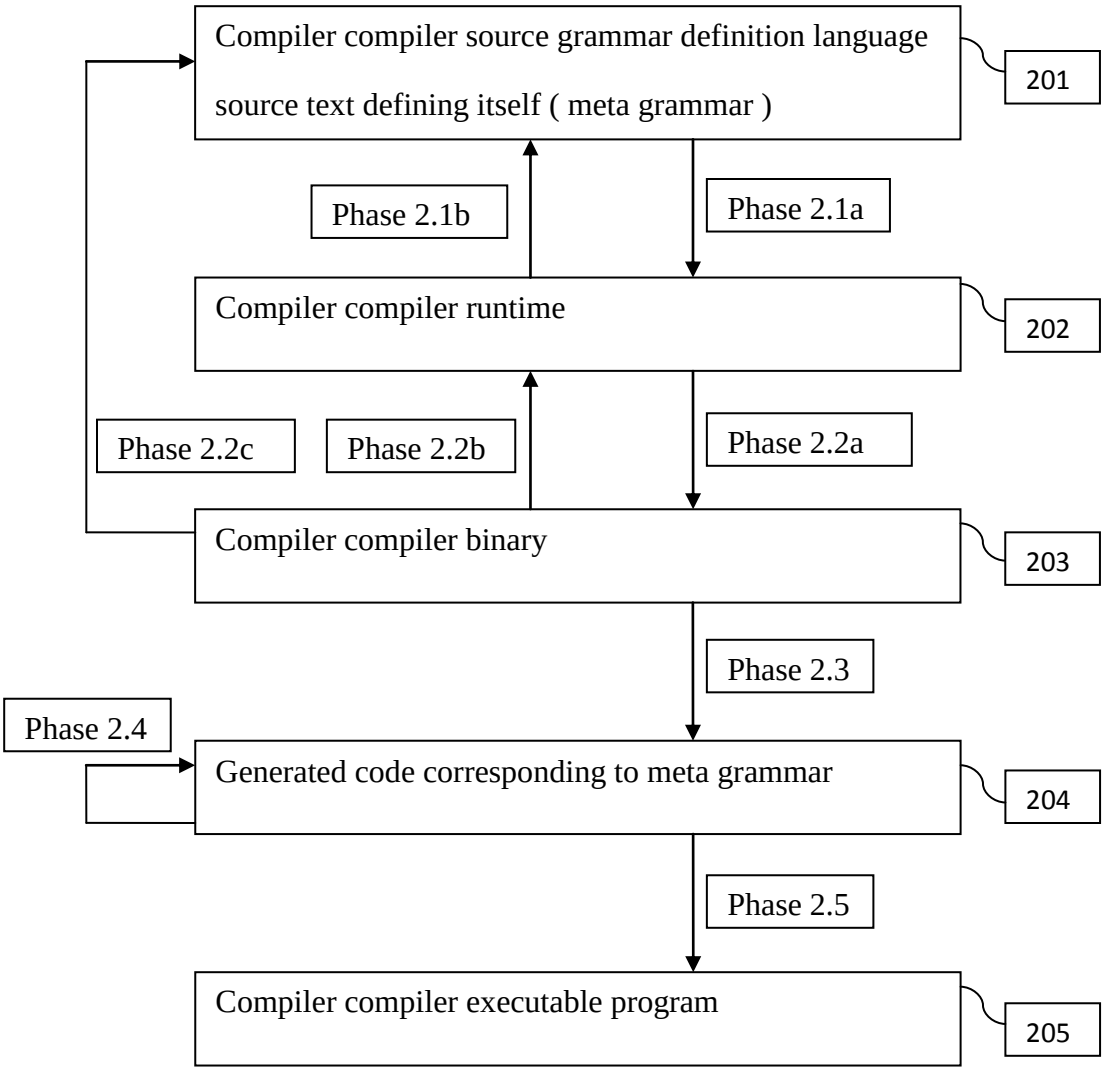
Compiler compiler source grammar definition language source text defining itself (meta grammar)
201
Phase 2.1b
Phase 2.1a
Compiler compiler runtime
202
Phase 2.2c
Phase 2.2b
Phase 2.2a
Compiler compiler binary
203
Phase 2.3
Phase 2.4
Generated code corresponding to meta grammar
204
Phase 2.5
Compiler compiler executable program
205

FIG. 2

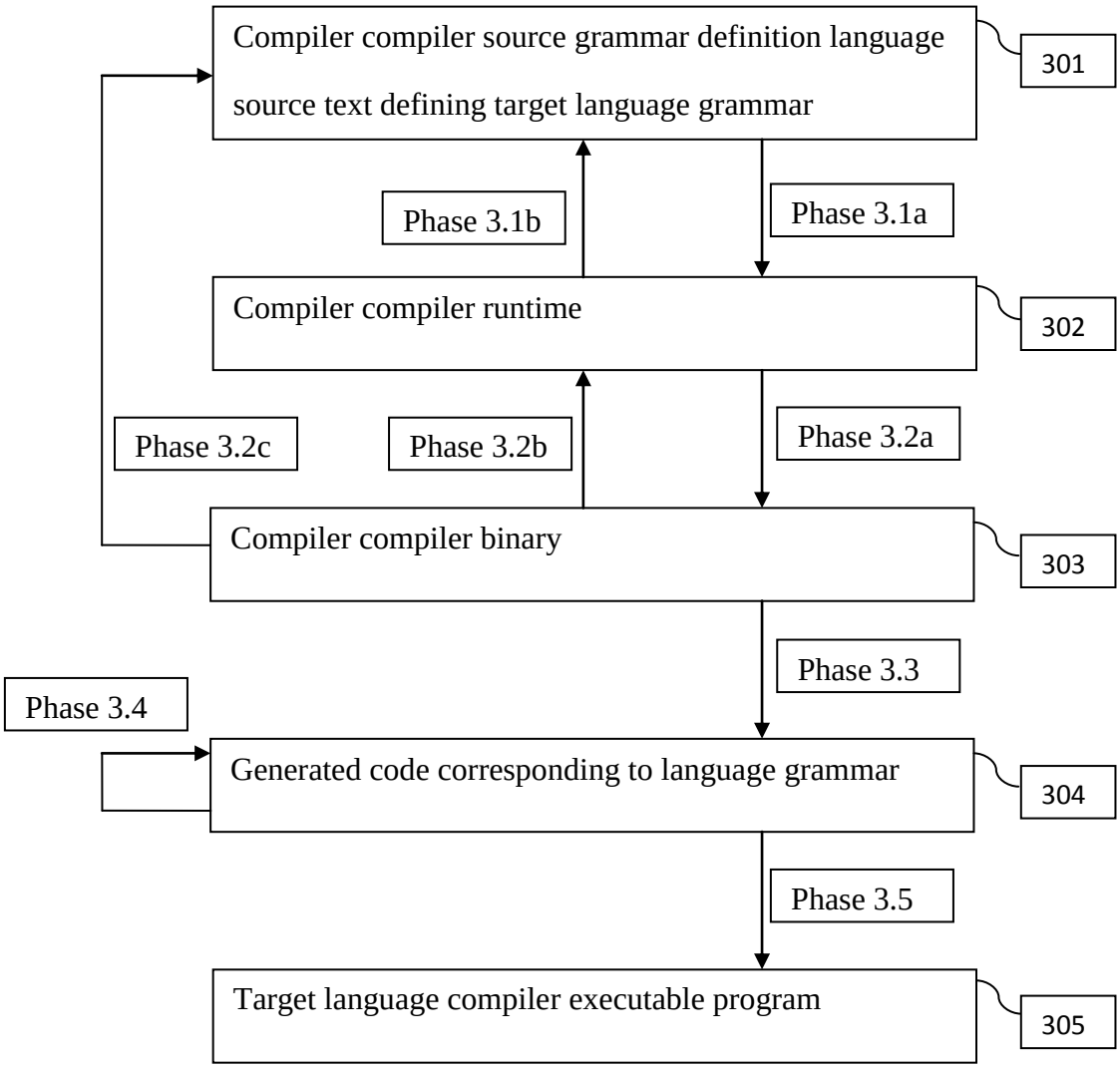
Compiler compiler source grammar definition language
source text defining target language grammar
301
Phase 3.1b
Phase 3.1a
Compiler compiler runtime
302
Phase 3.2c
Phase 3.2b
Phase 3.2a
Compiler compiler binary
303
Phase 3.3
Phase 3.4
Generated code corresponding to language grammar
304
Phase 3.5
Target language compiler executable program
305

FIG. 3

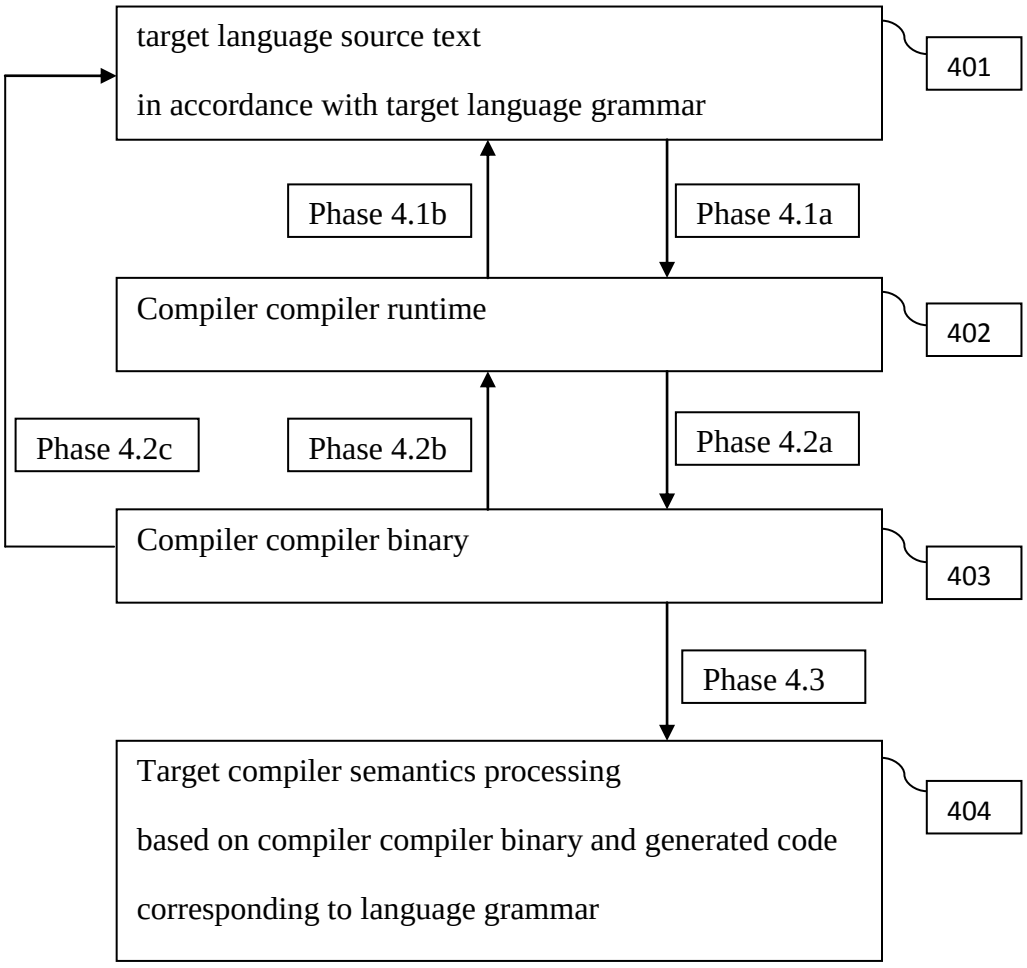
target language source text
in accordance with target language grammar
401
Phase 4.1b
Phase 4.1a
Compiler compiler runtime
402
Phase 4.2c
Phase 4.2b
Phase 4.2a
Compiler compiler binary
403
Phase 4.3
Target compiler semantics processing
based on compiler compiler binary and generated code
corresponding to language grammar
404

FIG. 4

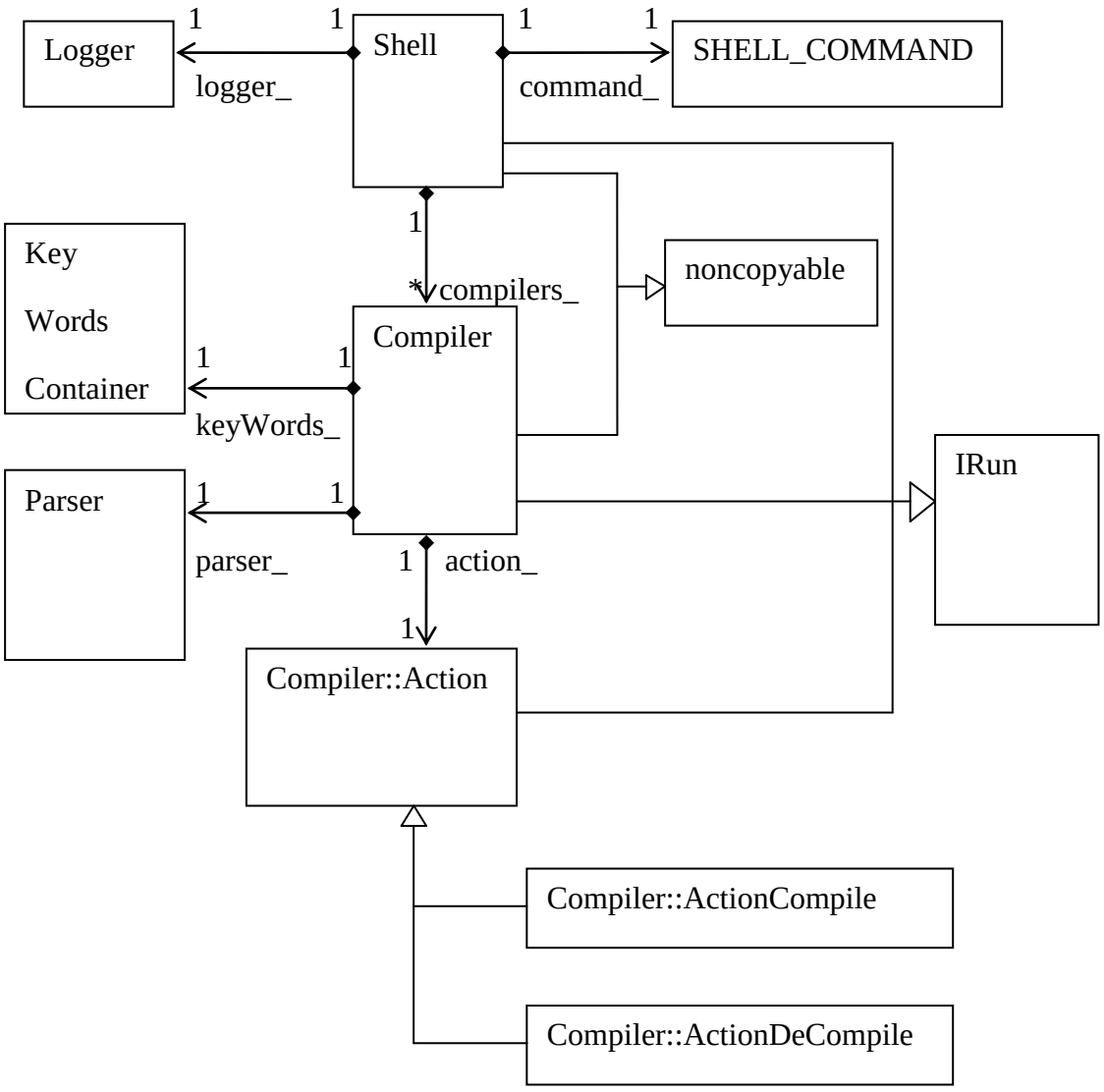
Logger
1
1
Shell
1
1
SHELL_COMMAND
logger_
command_
1
Key
Words
Container
*
compilers_
noncopyable
Compiler
1
1
keyWords_
Parser
1
1
IRun
parser_
1
action_
1
Compiler::Action
Compiler::ActionCompile
Compiler::ActionDeCompile

FIG. 5

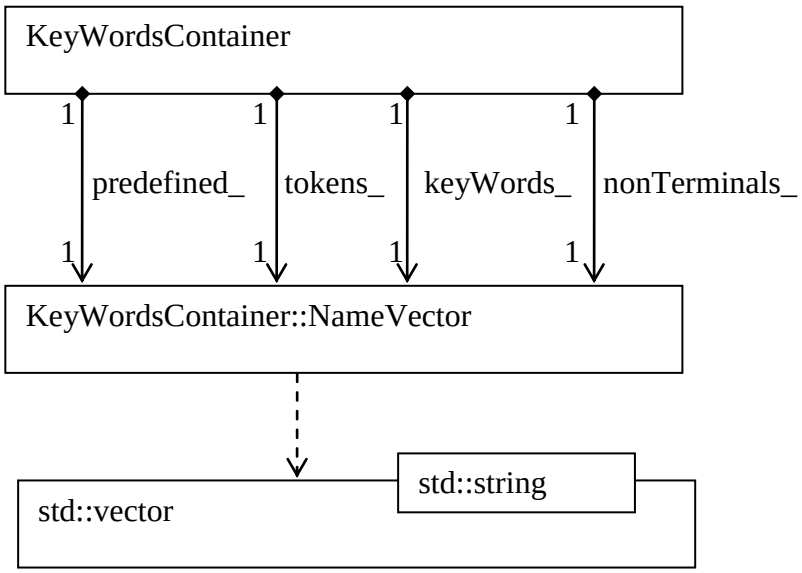
KeyWordsContainer
1
1
1
1
predefined_
tokens_
keyWords_
nonTerminals_
1
1
1
1
KeyWordsContainer::NameVector
std::string
std::vector

FIG. 6

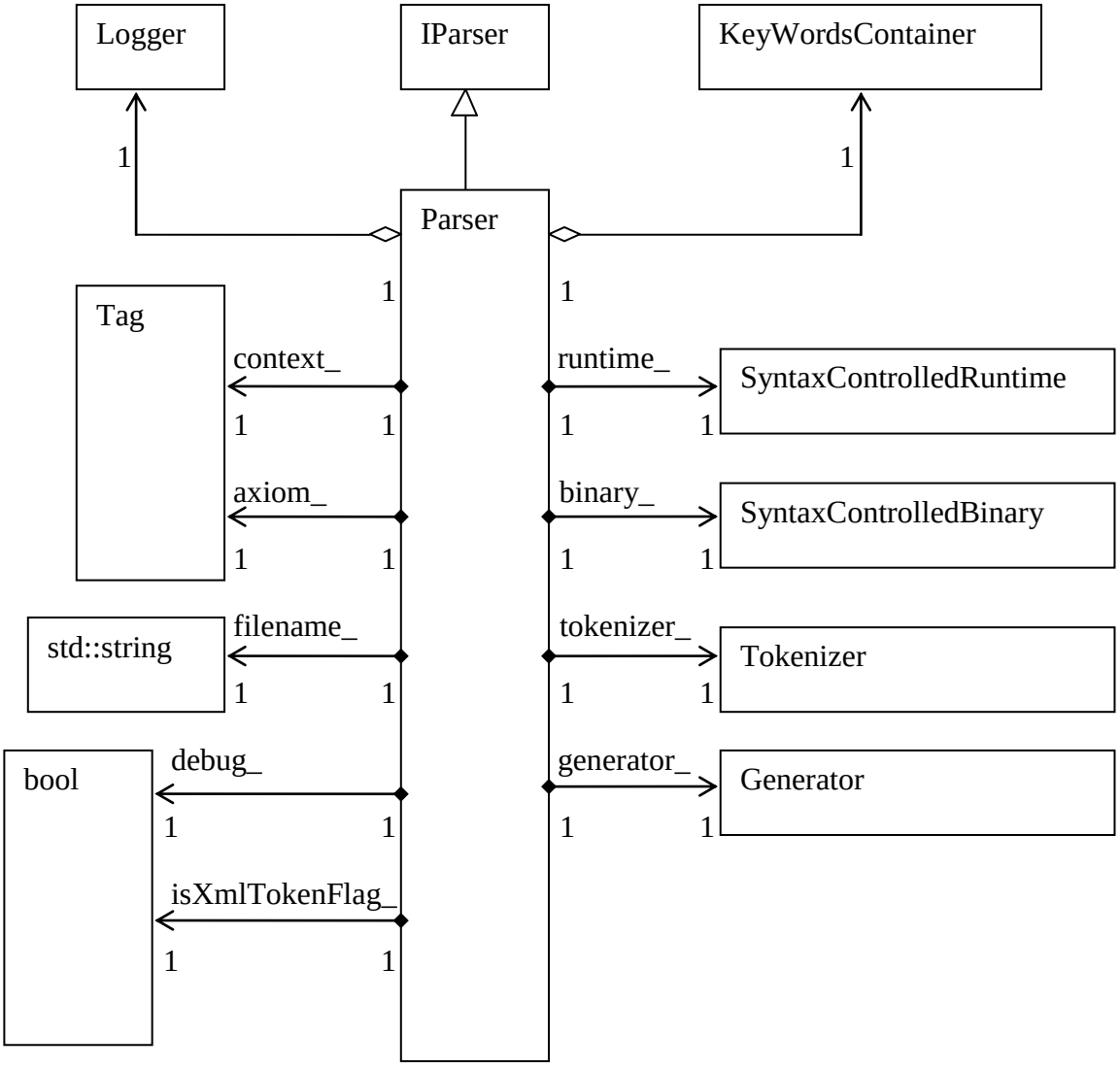
Logger
IParser
KeyWordsContainer
1
1
Parser
1
1
Tag
context_
runtime_
SyntaxControlledRuntime
1
1
1
1
axiom_
binary_
SyntaxControlledBinary
1
1
1
1
std::string
filename_
tokenizer_
Tokenizer
1
1
1
1
bool
debug_
generator_
Generator
1
1
1
1
isXmlTokenFlag_
1
1

FIG. 7

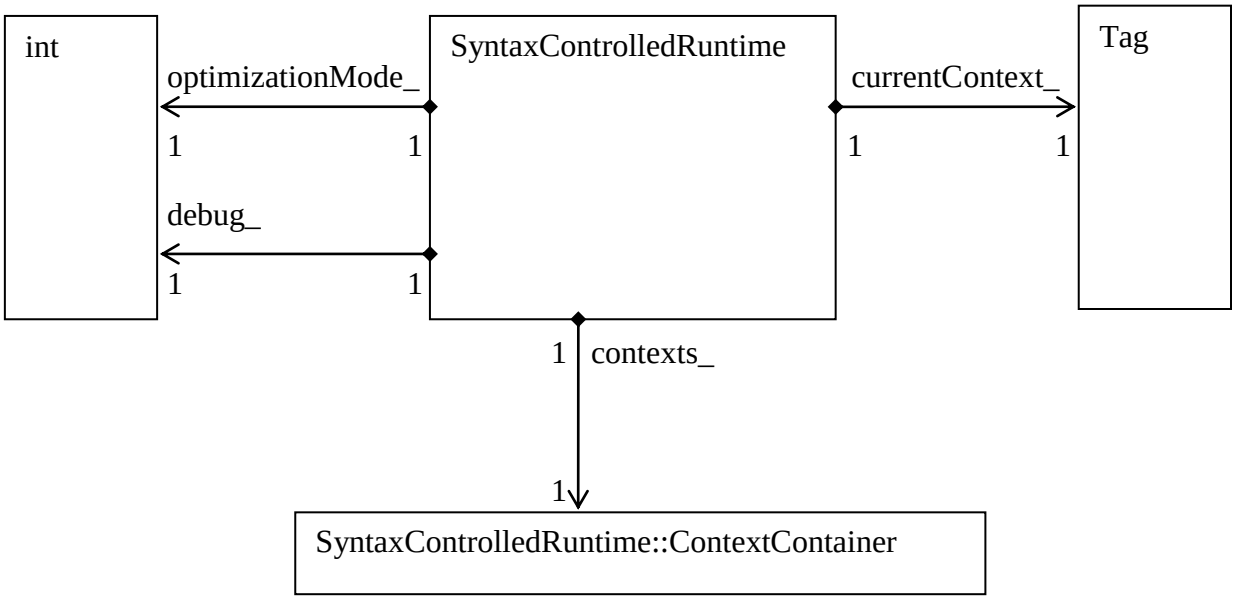
int
optimizationMode_
1
1
debug_
1
1
SyntaxControlledRuntime
currentContext_
1
1
Tag
1
contexts_
1
SyntaxControlledRuntime::ContextContainer

FIG. 8

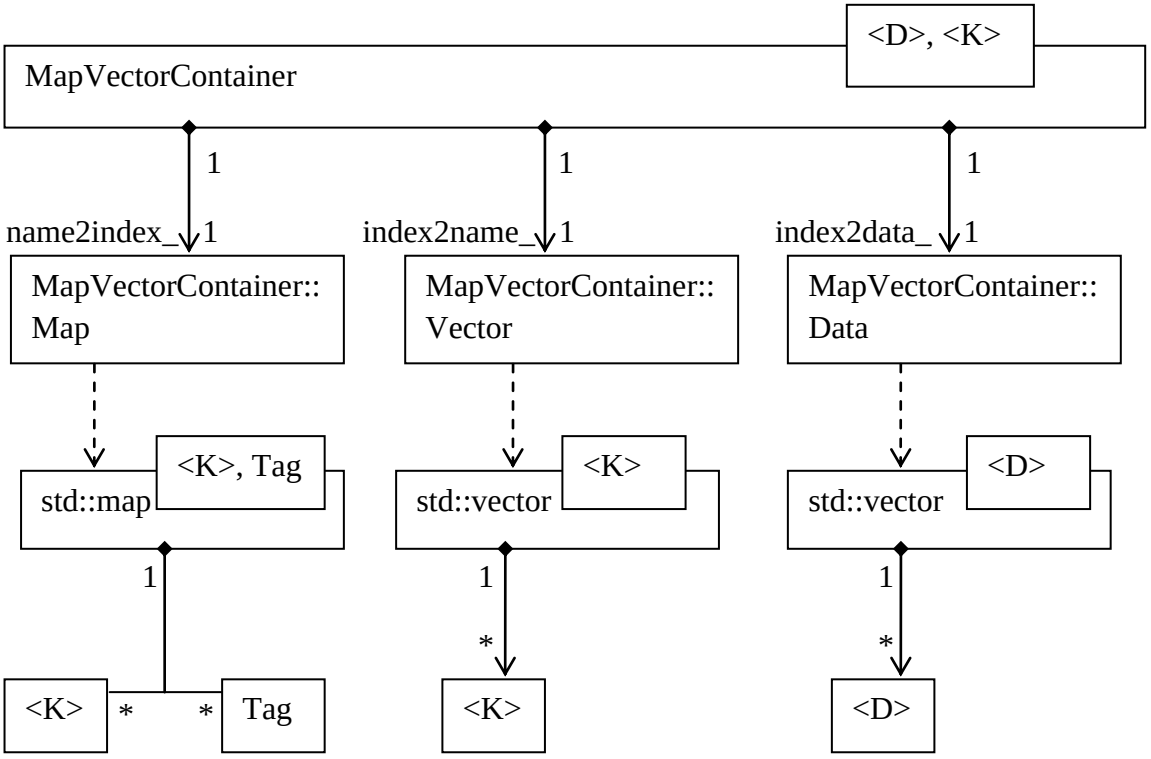
MapVectorContainer
<D>, <K>
1
1
1
name2index_
1
index2name_
1
index2data_
1
MapVectorContainer::
Map
MapVectorContainer::
Vector
MapVectorContainer::
Data
std::map
<K>, Tag
std::vector
<K>
std::vector
<D>
1
1
1
*
*
<K>
*
*
Tag
<K>
<D>

FIG. 9

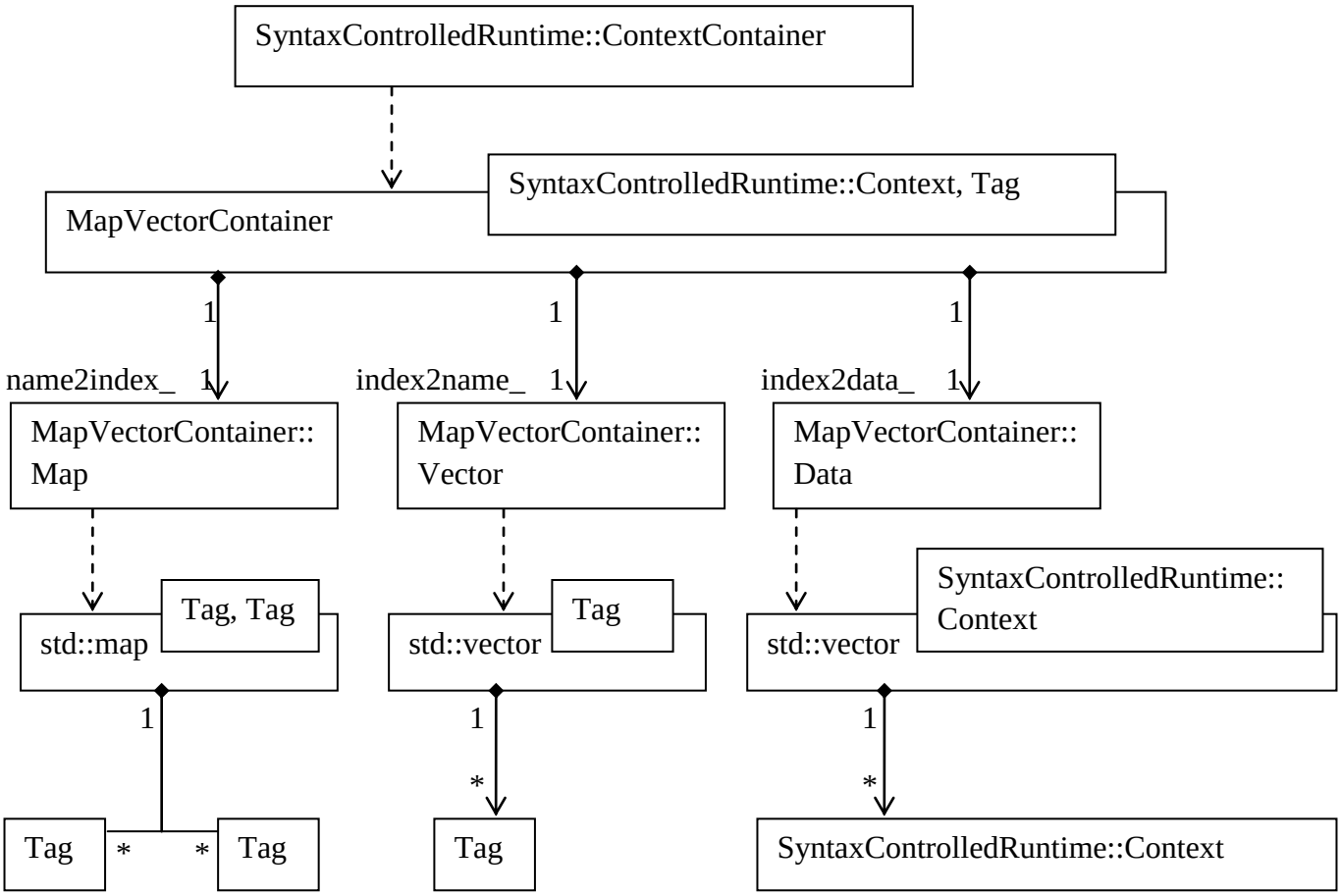
SyntaxControlledRuntime::ContextContainer
MapVectorContainer
SyntaxControlledRuntime::Context, Tag
1
1
1
name2index_ 1
index2name_ 1
index2data_ 1
MapVectorContainer::
Map
MapVectorContainer::
Vector
MapVectorContainer::
Data
std::map
Tag, Tag
std::vector
Tag
std::vector
SyntaxControlledRuntime::
Context
1
1
1
*
*
Tag
*
*
Tag
Tag
SyntaxControlledRuntime::Context

FIG. 10

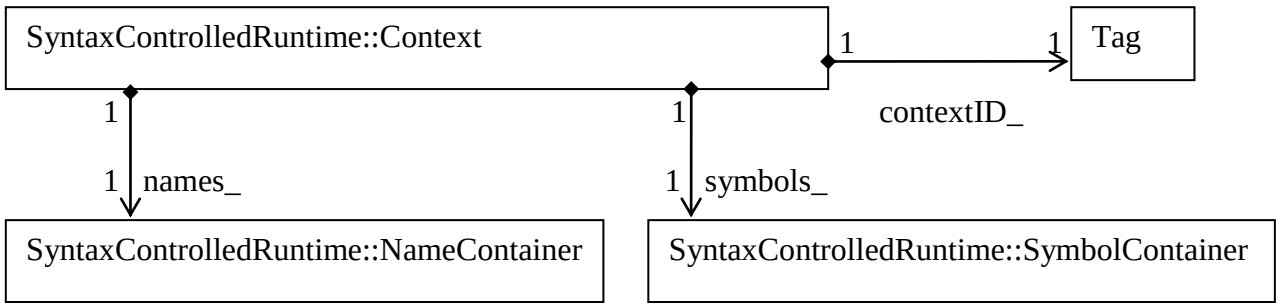
SyntaxControlledRuntime::Context
1
1
Tag
contextID_
1
1
names_
1
1
symbols_
SyntaxControlledRuntime::NameContainer
SyntaxControlledRuntime::SymbolContainer

FIG. 11

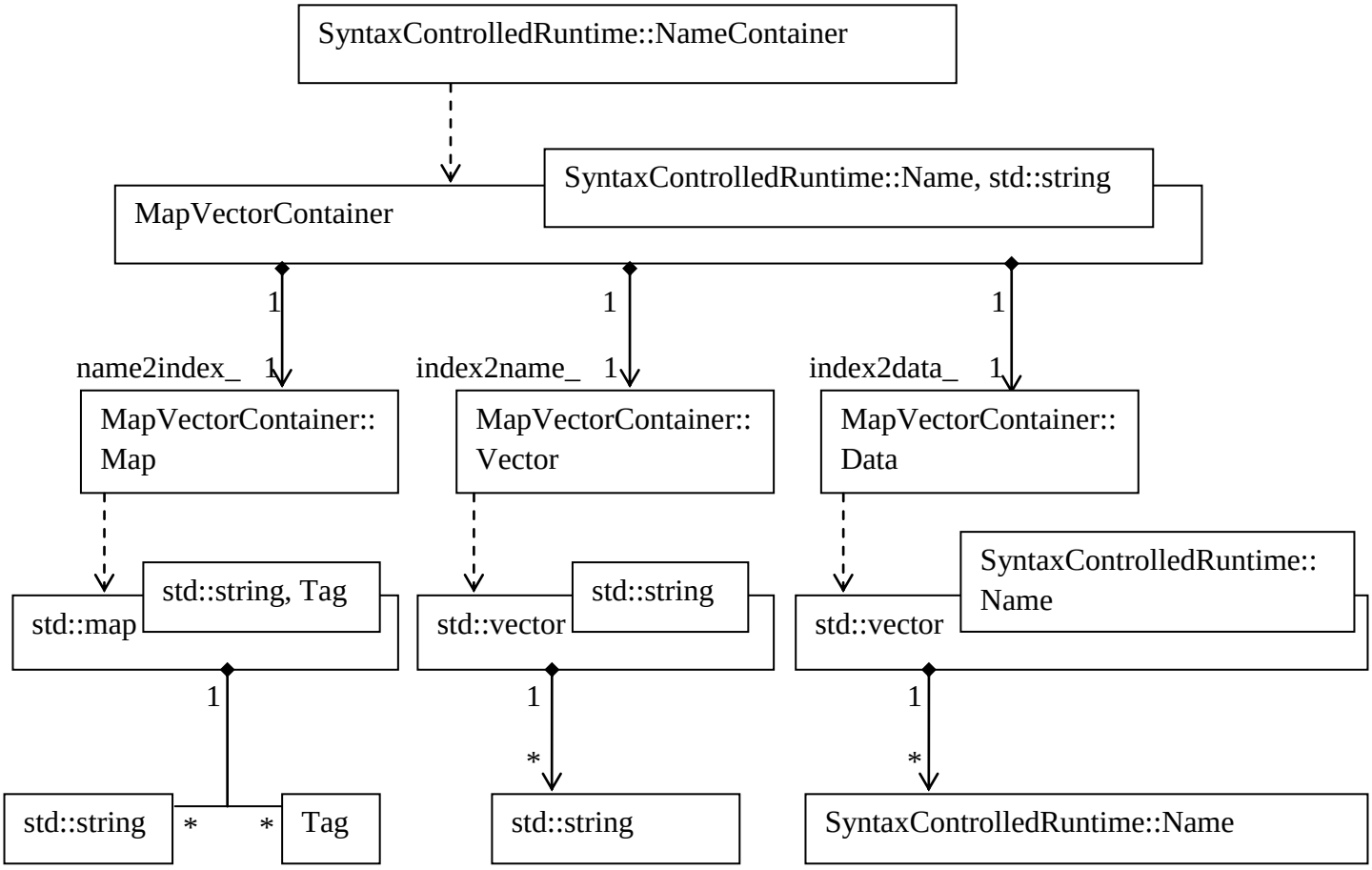
SyntaxControlledRuntime::NameContainer
MapVectorContainer
SyntaxControlledRuntime::Name, std::string
1
1
1
name2index_ 1
index2name_ 1
index2data_ 1
MapVectorContainer::
Map
MapVectorContainer::
Vector
MapVectorContainer::
Data
std::map
std::string, Tag
std::vector
std::string
std::vector
SyntaxControlledRuntime::
Name
1
1
1
*
*
std::string
*
*
Tag
std::string
SyntaxControlledRuntime::Name

FIG. 12

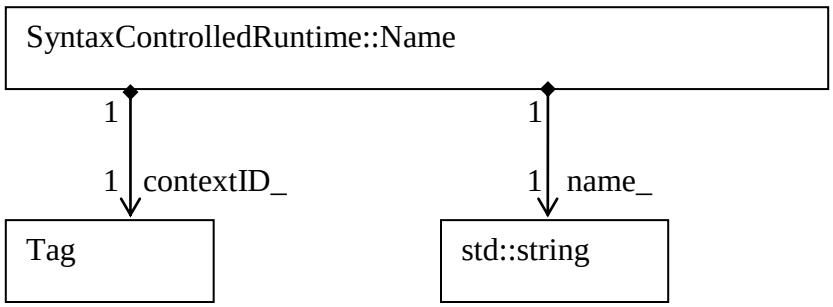
SyntaxControlledRuntime::Name
1
1
contextID_
1
1
name_
Tag
std::string

FIG. 13

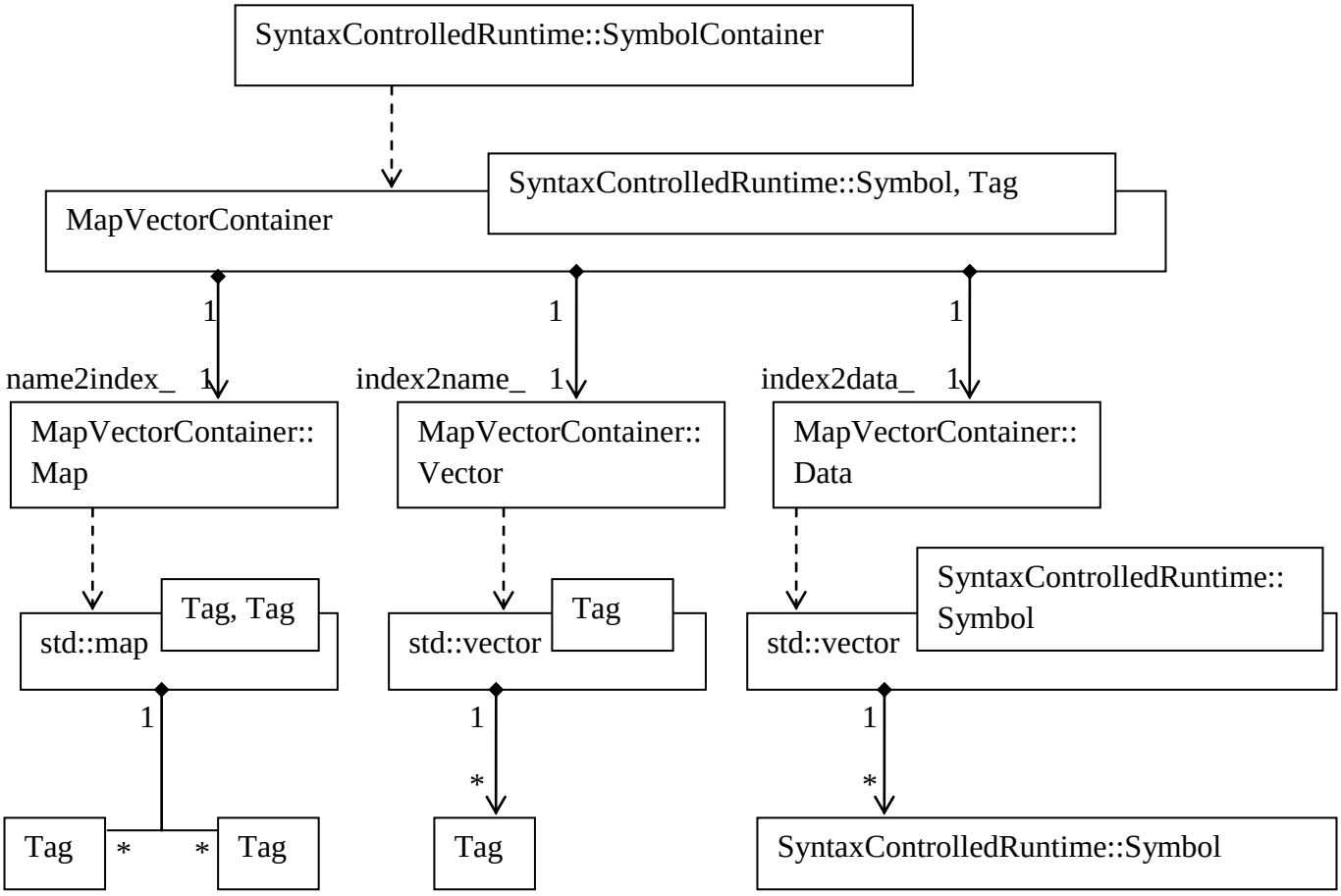
SyntaxControlledRuntime::SymbolContainer
MapVectorContainer
SyntaxControlledRuntime::Symbol, Tag
1
1
1
name2index_ 1
index2name_ 1
index2data_ 1
MapVectorContainer::
Map
MapVectorContainer::
Vector
MapVectorContainer::
Data
std::map
Tag, Tag
std::vector
Tag
std::vector
SyntaxControlledRuntime::
Symbol
1
1
1
*
*
Tag
*
*
Tag
Tag
SyntaxControlledRuntime::Symbol

FIG. 14

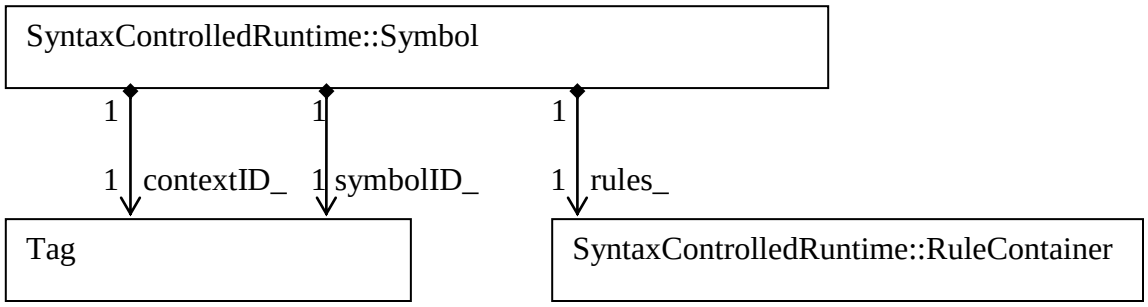
SyntaxControlledRuntime::Symbol
1
1
1
1 contextID_
1 symbolID_
1 rules_
Tag
SyntaxControlledRuntime::RuleContainer

FIG. 15

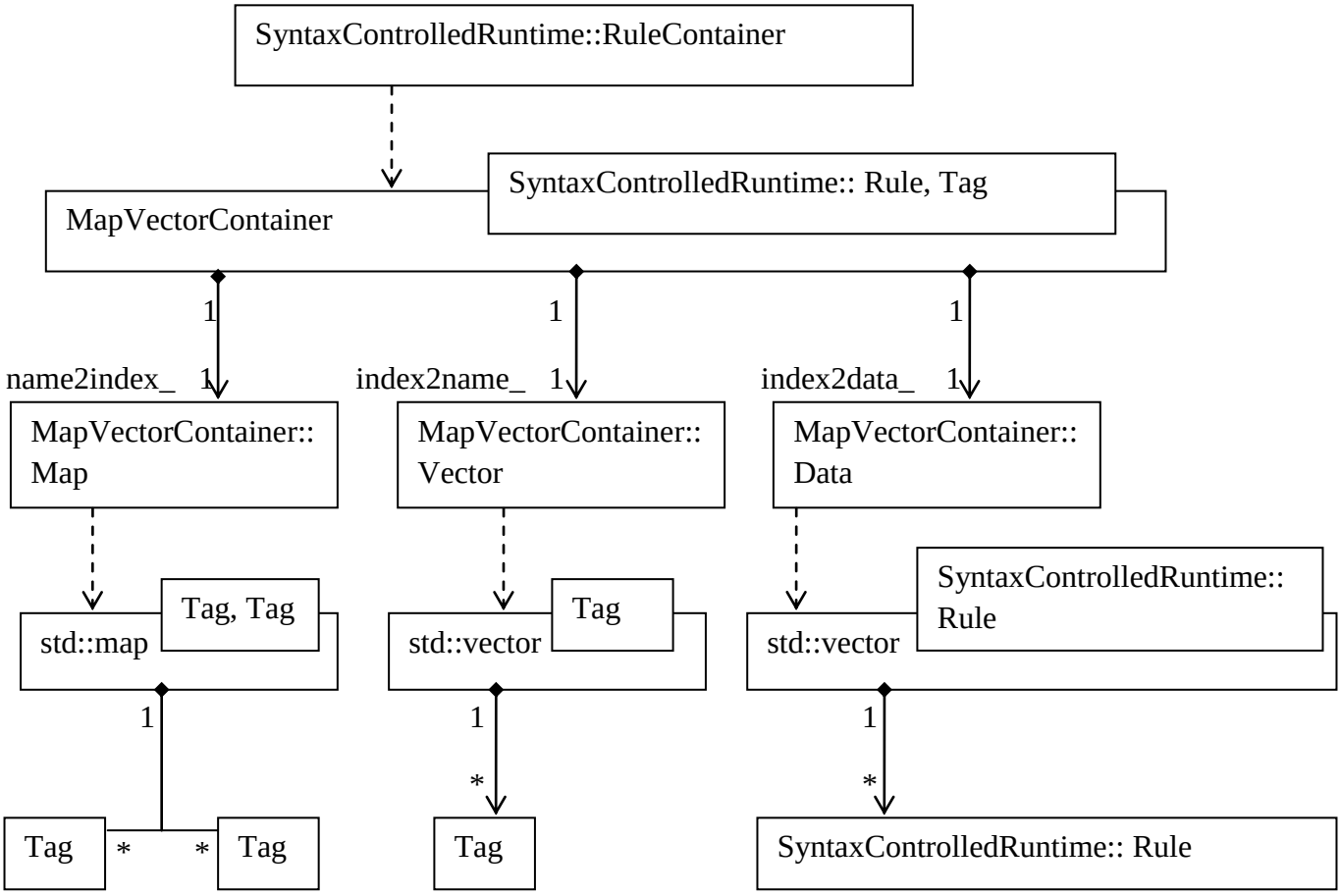
SyntaxControlledRuntime::RuleContainer
MapVectorContainer
SyntaxControlledRuntime:: Rule, Tag
1
1
1
name2index_ 1
index2name_ 1
index2data_ 1
MapVectorContainer:: Map
MapVectorContainer:: Vector
MapVectorContainer:: Data
std::map
Tag, Tag
std::vector
Tag
std::vector
SyntaxControlledRuntime:: Rule
1
1
1
*
*
Tag
*
*
Tag
Tag
SyntaxControlledRuntime:: Rule

FIG. 16

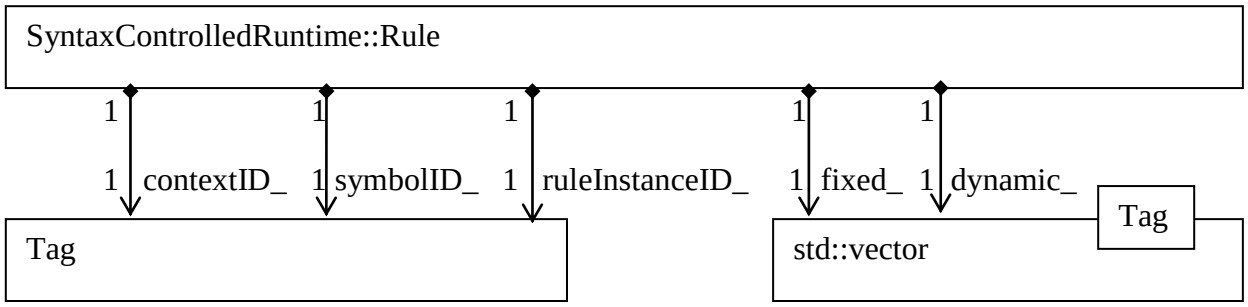
SyntaxControlledRuntime::Rule
1
1
contextID_
1
1
symbolID_
1
1
ruleInstanceID_
1
1
fixed_
1
1
dynamic_
Tag
Tag
std::vector

FIG. 17

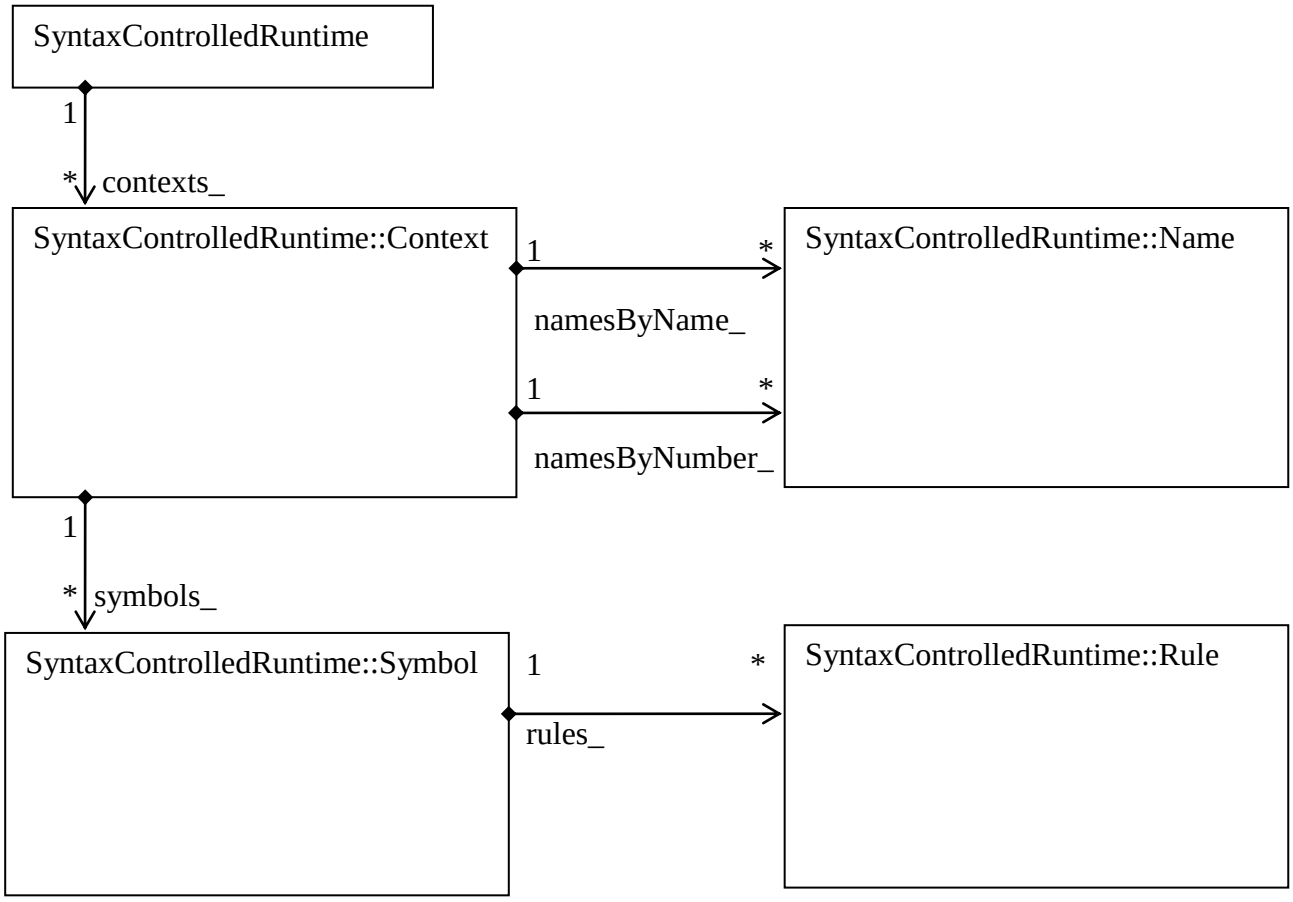
SyntaxControlledRuntime
1
*
contexts_
SyntaxControlledRuntime::Context
1
*
SyntaxControlledRuntime::Name
namesByName_
1
*
namesByNumber_
1
*
symbols_
SyntaxControlledRuntime::Symbol
1
*
SyntaxControlledRuntime::Rule
rules_

FIG. 18

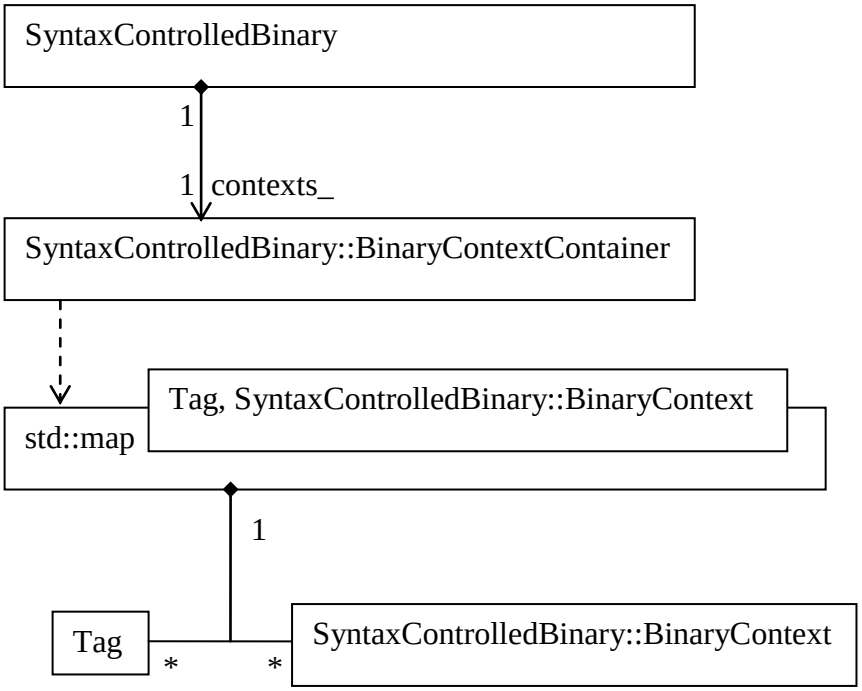
SyntaxControlledBinary
1
1
contexts_
SyntaxControlledBinary::BinaryContextContainer
Tag, SyntaxControlledBinary::BinaryContext
std::map
1
Tag
*
*
SyntaxControlledBinary::BinaryContext

FIG. 19

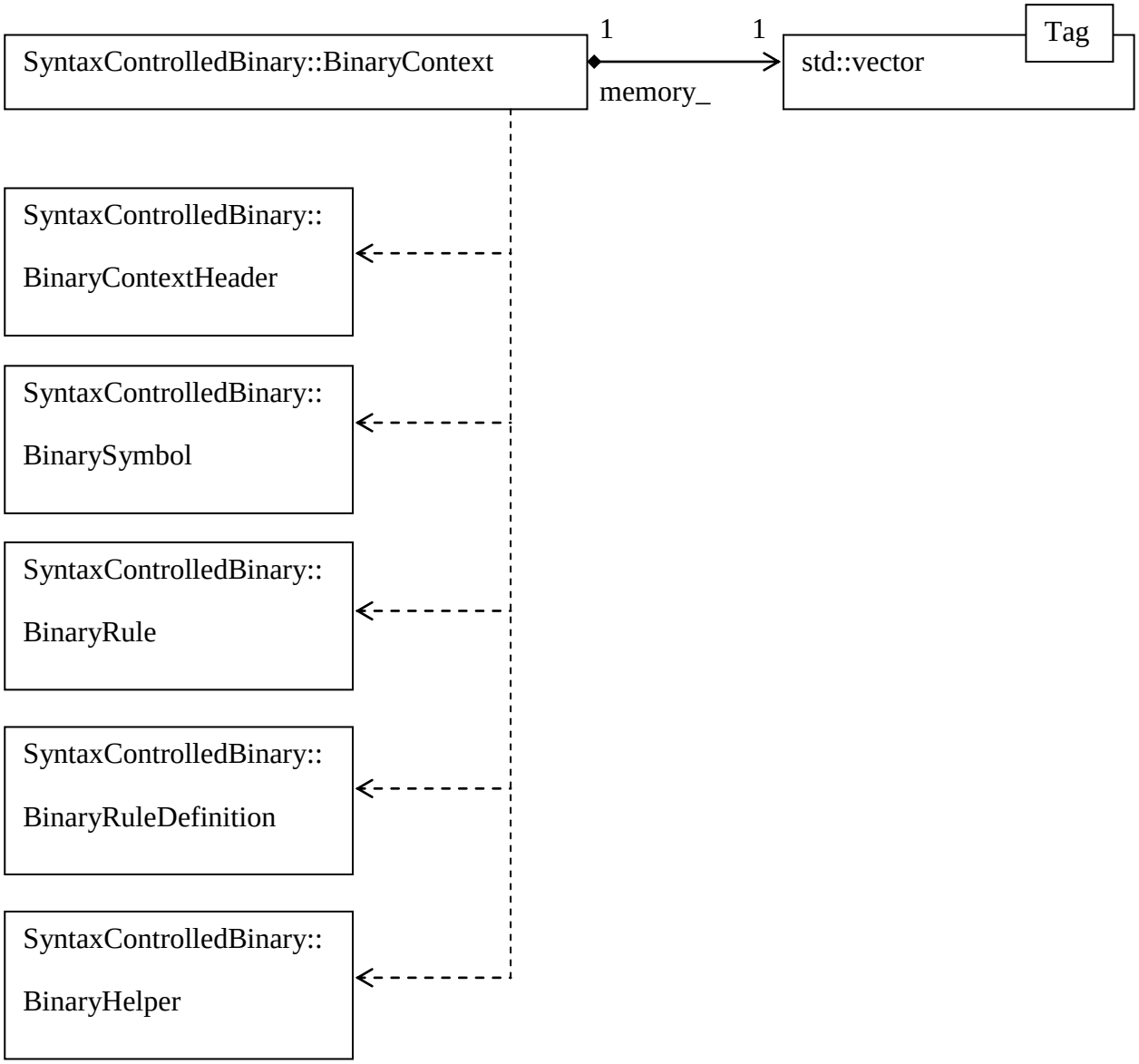
SyntaxControlledBinary::BinaryContext
1
1
memory_
std::vector
Tag
SyntaxControlledBinary::
BinaryContextHeader
SyntaxControlledBinary::
BinarySymbol
SyntaxControlledBinary::
BinaryRule
SyntaxControlledBinary::
BinaryRuleDefinition
SyntaxControlledBinary::
BinaryHelper

FIG. 20

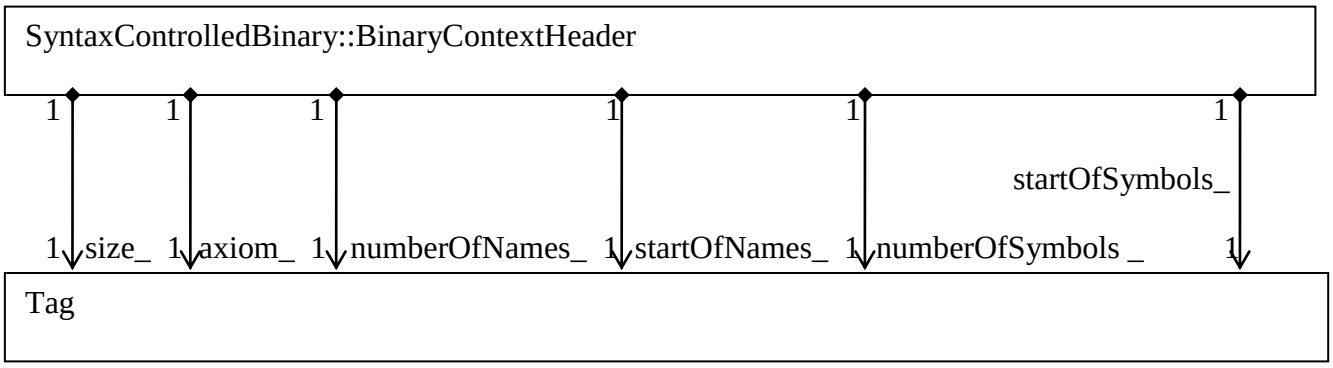
SyntaxControlledBinary::BinaryContextHeader
1
1
1
1
1
1
startOfSymbols_
1
size_
1
axiom_
1
numberOfNames_
1
startOfNames_
1
numberOfSymbols _
1
Tag

FIG. 21

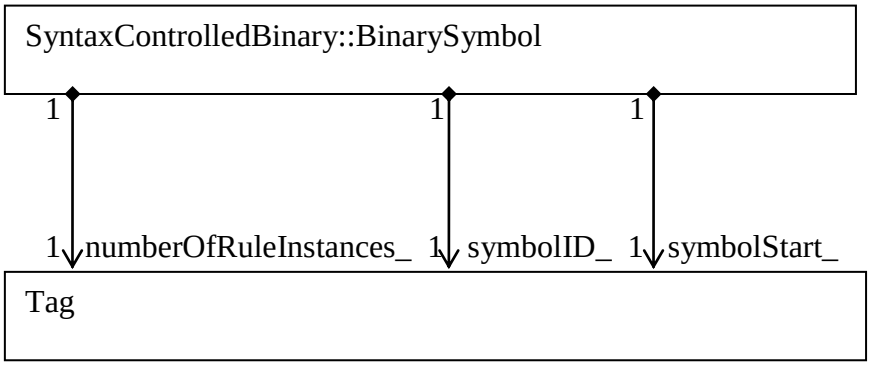
SyntaxControlledBinary::BinarySymbol
1
1
1
1
numberOfRuleInstances_
1
symbolID_
1
symbolStart_
Tag

FIG. 22

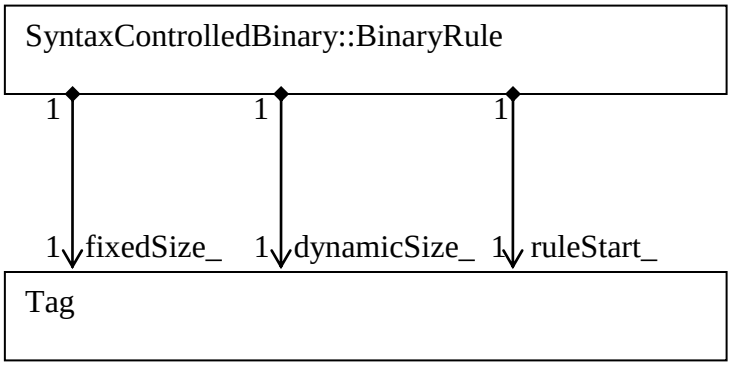
SyntaxControlledBinary::BinaryRule
1
1
1
1 fixedSize_
1 dynamicSize_
1 ruleStart_
Tag

FIG. 23

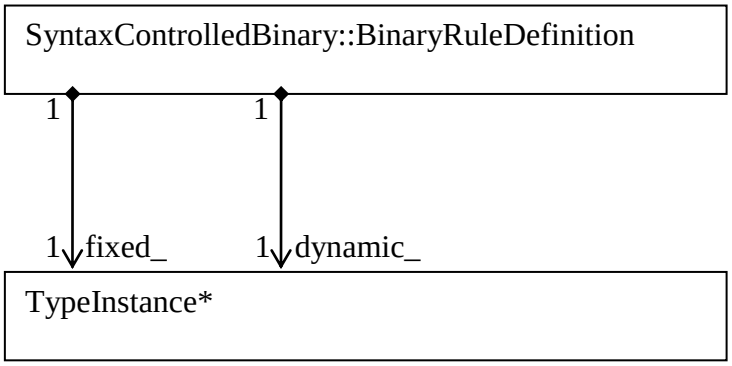
SyntaxControlledBinary::BinaryRuleDefinition
1
1
1 fixed_
1 dynamic_
TypeInstance*

FIG. 24

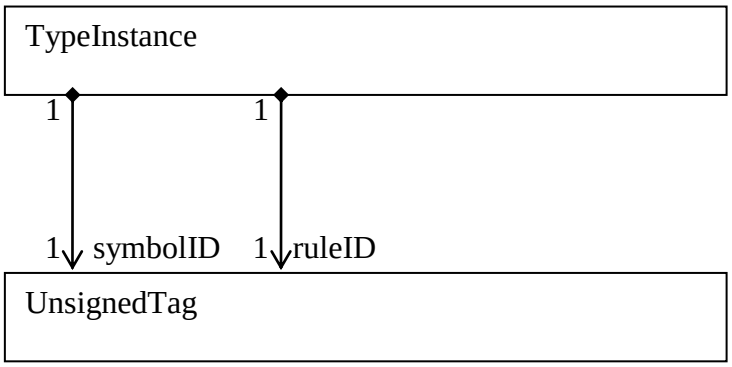

FIG. 25

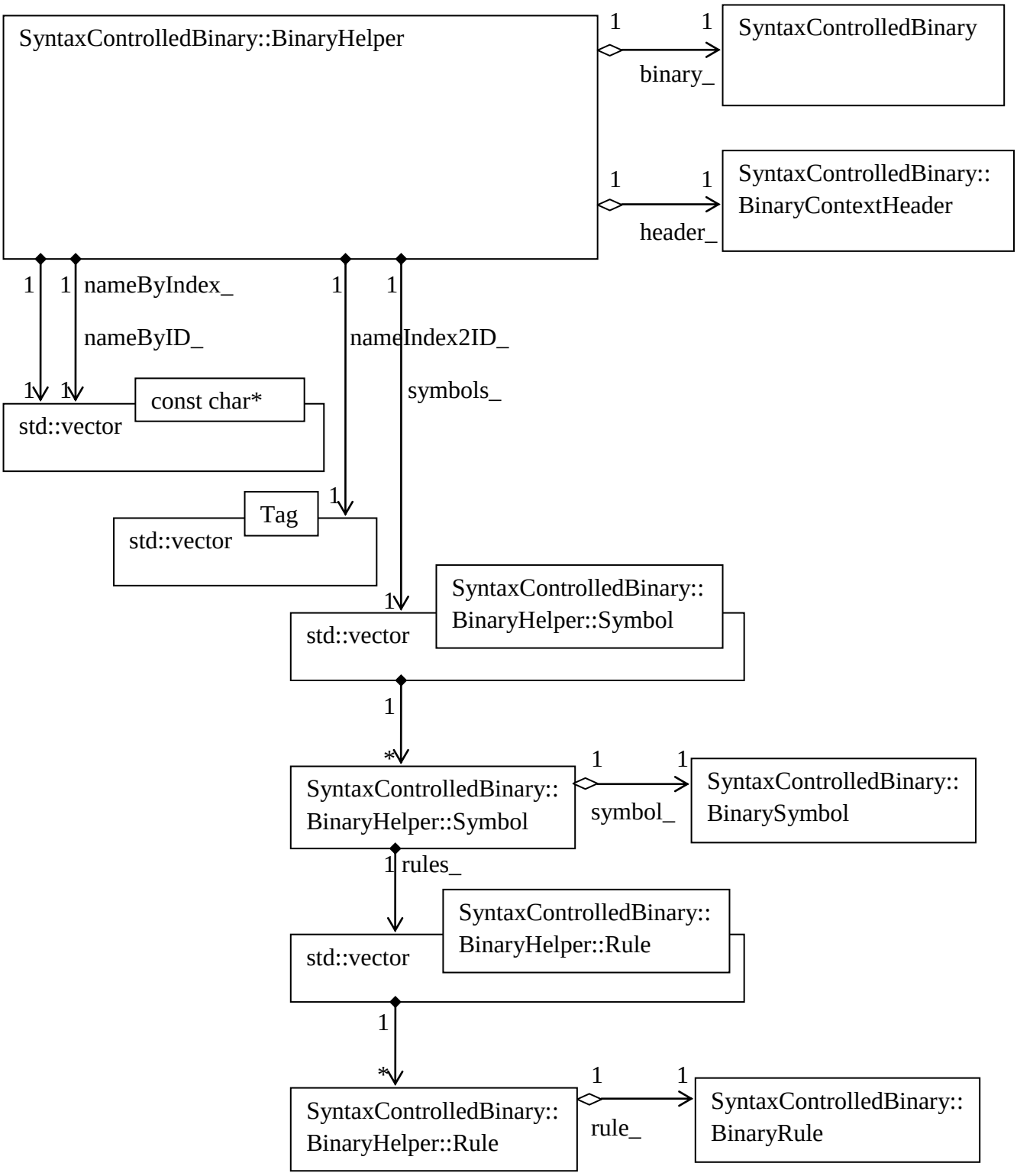
SyntaxControlledBinary::BinaryHelper
1
1
SyntaxControlledBinary
binary_
1
1
SyntaxControlledBinary::
BinaryContextHeader
header_
1
1
nameByIndex_
1
1
nameByID_
1
1
const char*
std::vector
1
nameIndex2ID_
symbols_
Tag
1
std::vector
1
SyntaxControlledBinary::
BinaryHelper::Symbol
std::vector
1
*
SyntaxControlledBinary::
BinaryHelper::Symbol
1
1
symbol_
SyntaxControlledBinary::
BinarySymbol
1
rules_
SyntaxControlledBinary::
BinaryHelper::Rule
std::vector
1
*
SyntaxControlledBinary::
BinaryHelper::Rule
1
1
rule_
SyntaxControlledBinary::
BinaryRule

FIG. 26

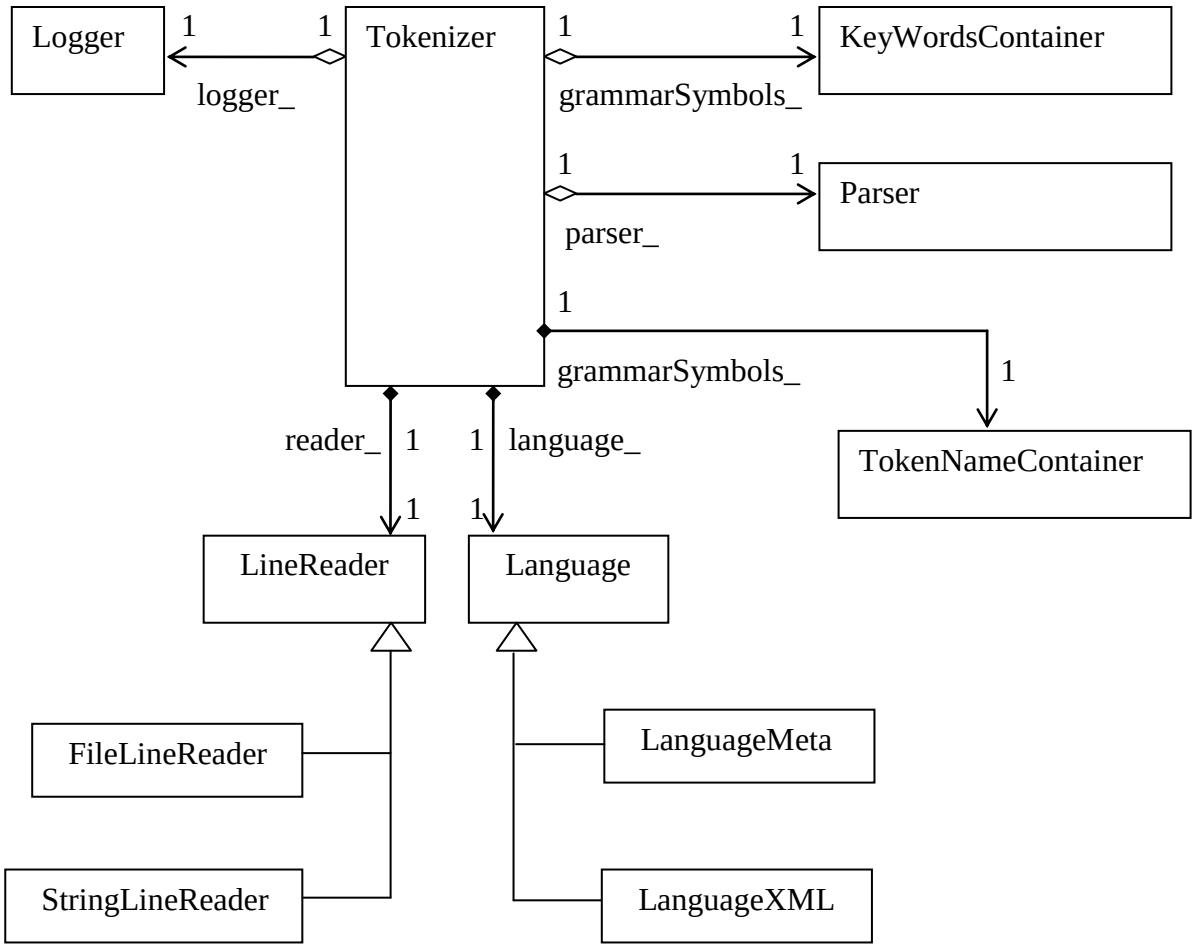
Logger
1
1
Tokenizer
1
1
KeyWordsContainer
logger_
grammarSymbols_
1
1
Parser
parser_
1
grammarSymbols_
1
TokenNameContainer
reader_
1
1
language_
1
1
LineReader
Language
FileLineReader
LanguageMeta
StringLineReader
LanguageXML

FIG. 27

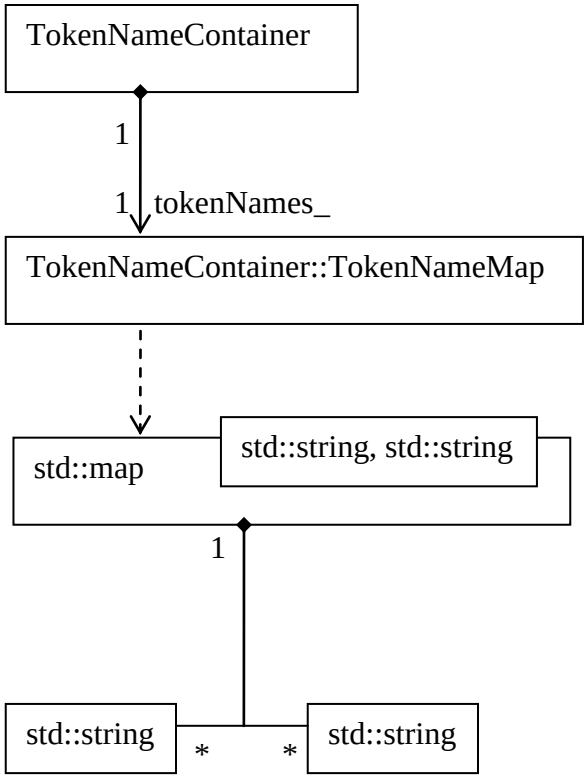
TokenNameContainer
1
1
tokenNames_
TokenNameContainer::TokenNameMap
std::map
std::string, std::string
1
std::string
*
*
std::string

FIG. 28

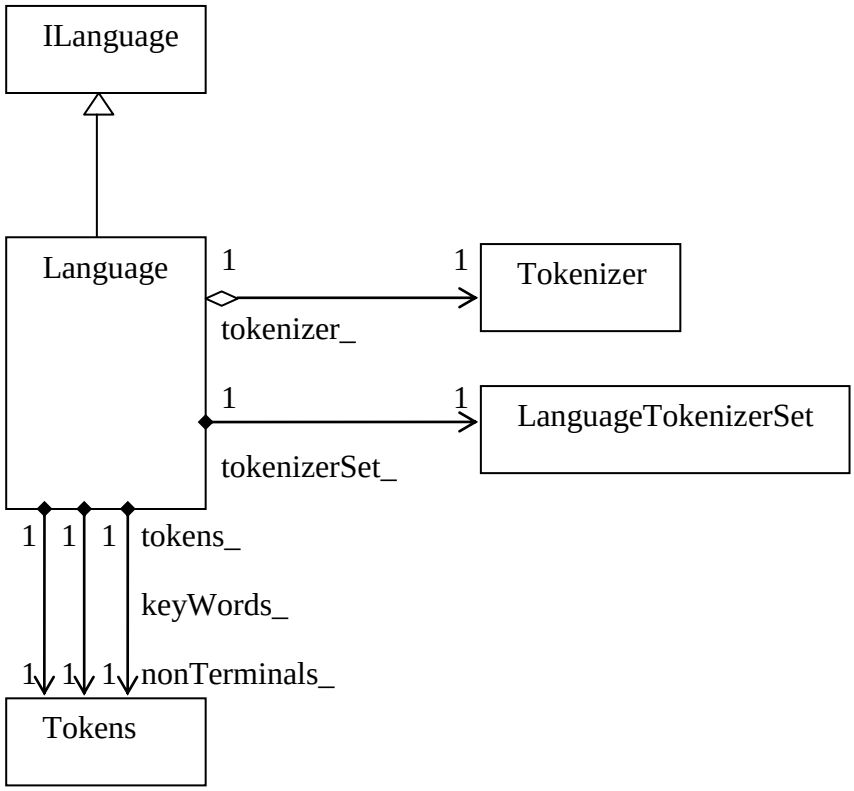
ILanguage
Language
1
1
Tokenizer
tokenizer_
1
1
LanguageTokenizerSet
tokenizerSet_
1 1 1 tokens_
keyWords_
1 1 1 nonTerminals_
Tokens

FIG. 29

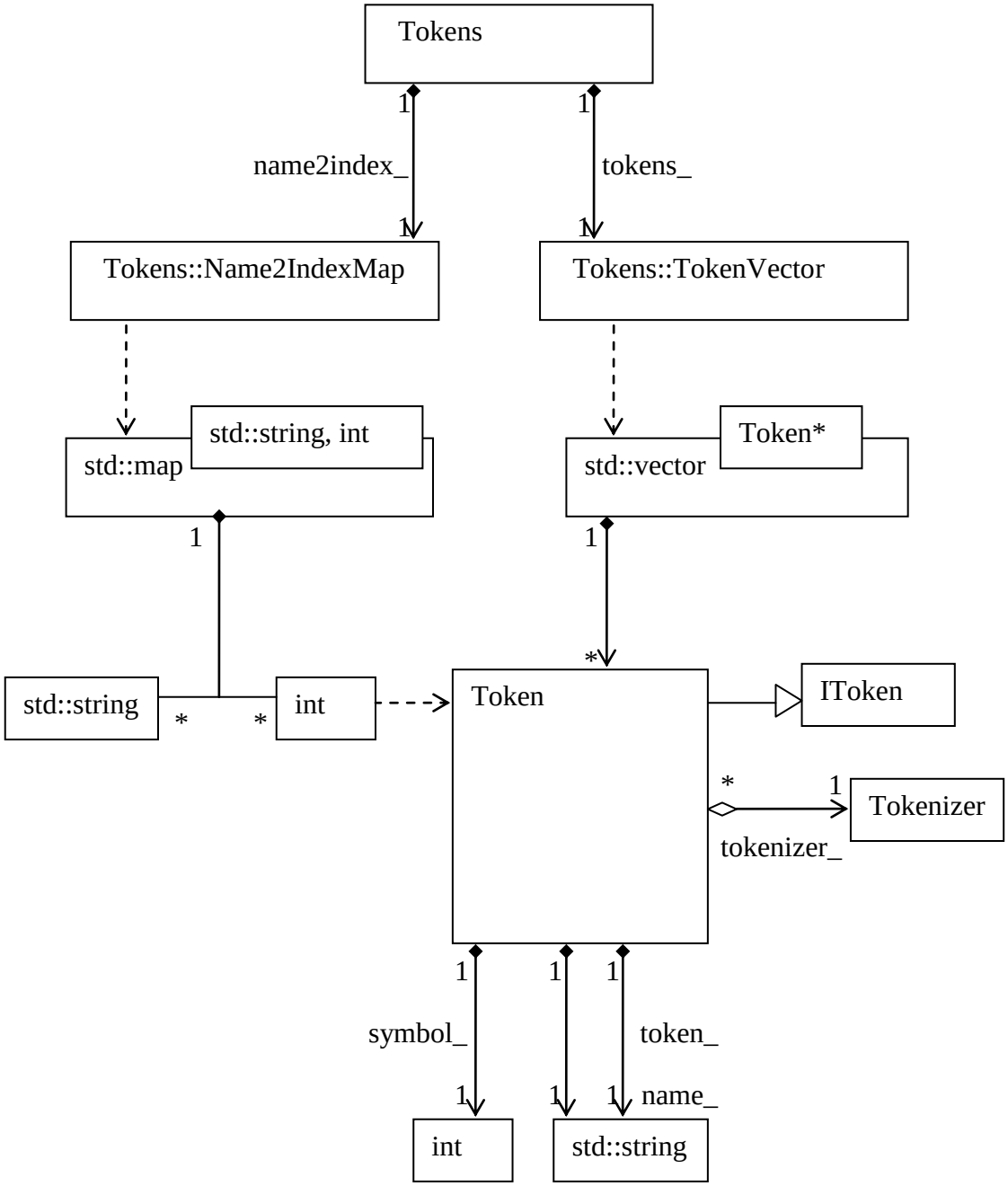
Tokens
1
1
name2index_
tokens_
1
1
Tokens::Name2IndexMap
Tokens::TokenVector
std::string, int
std::map
Token*
std::vector
1
1
*
std::string
*
*
int
Token
IToken
*
1
Tokenizer
tokenizer_
1
1
1
symbol_
token_
1
1
1
name_
int
std::string

FIG. 30

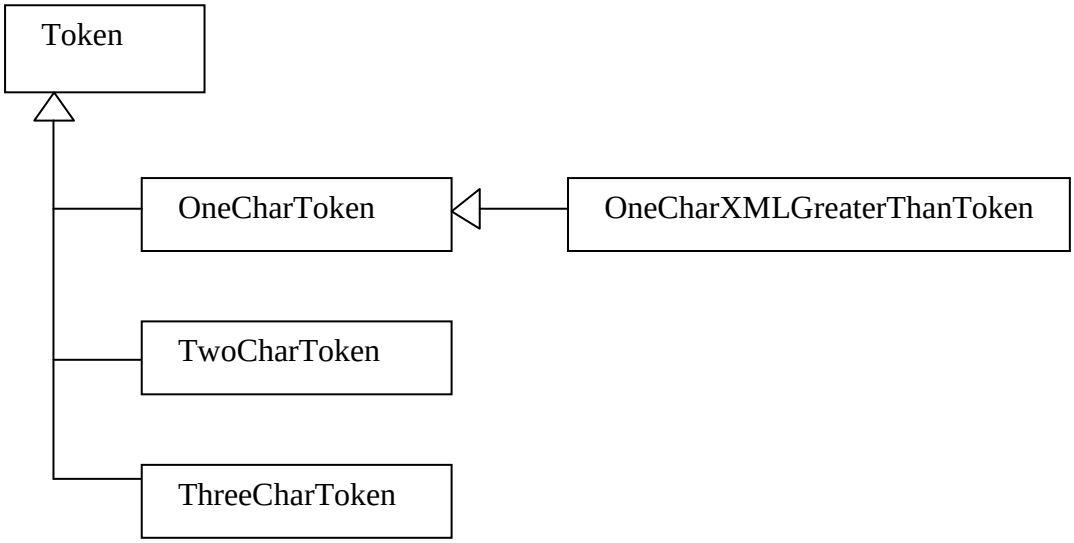
Token
OneCharToken
OneCharXMLGreaterThanToken
TwoCharToken
ThreeCharToken

FIG. 31

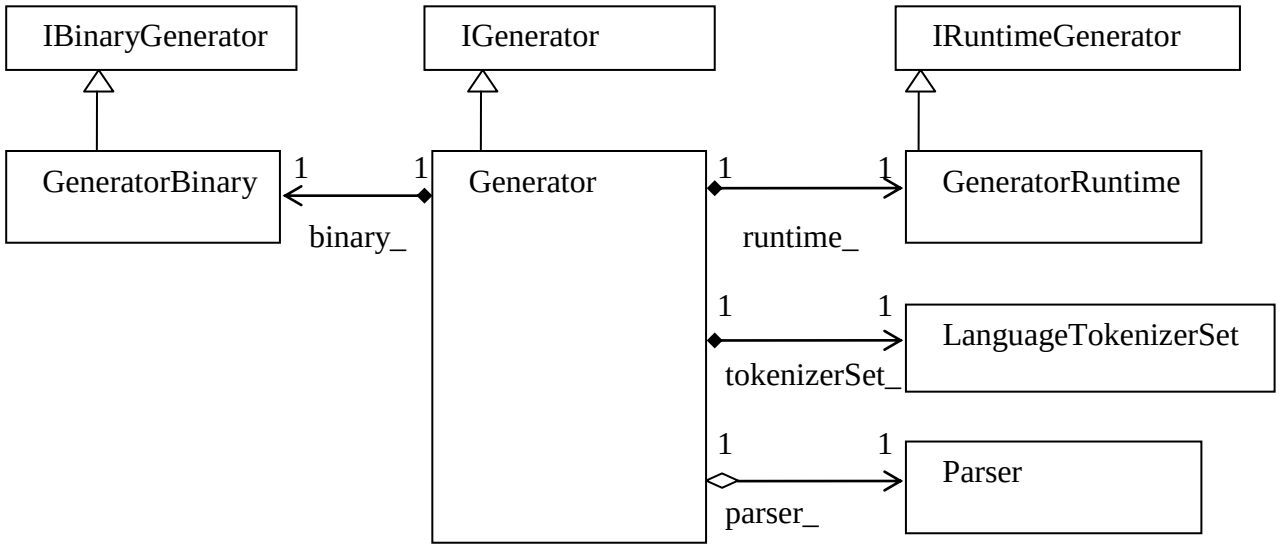
IBinaryGenerator
IGenerator
IRuntimeGenerator
GeneratorBinary
1
1
Generator
1
1
GeneratorRuntime
binary_
runtime_
1
1
LanguageTokenizerSet
tokenizerSet_
1
1
Parser
parser_

FIG. 32

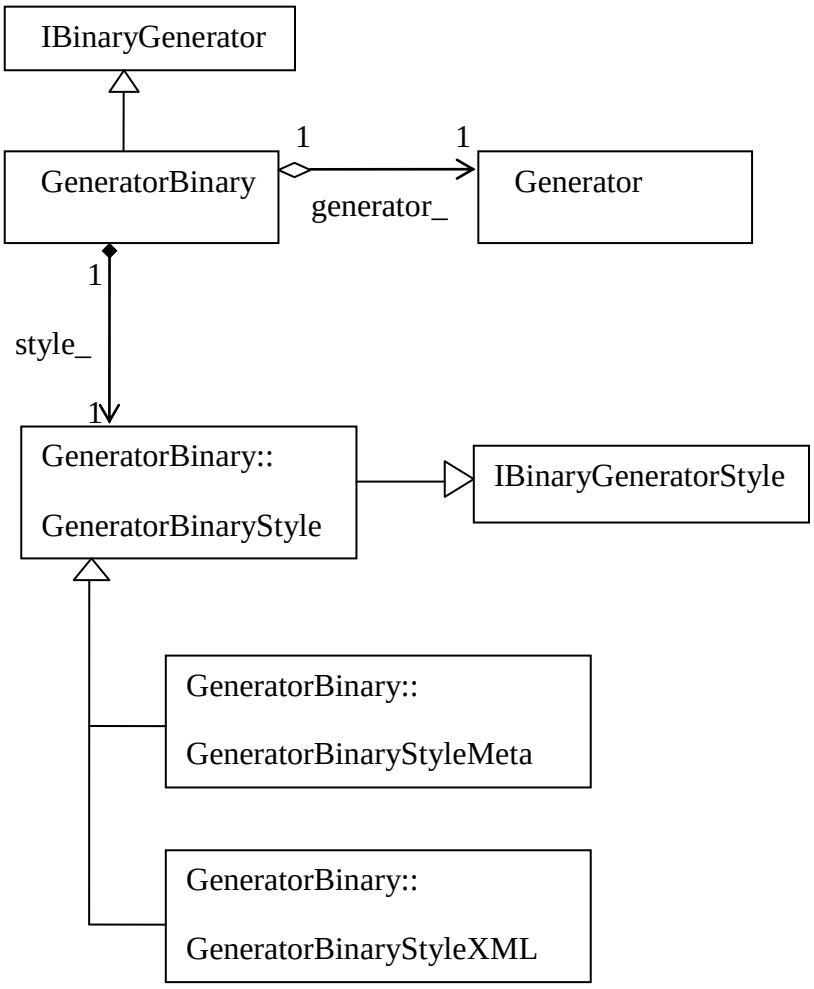
IBinaryGenerator
GeneratorBinary
1
1
generator_
Generator
1
style_
1
GeneratorBinary::
GeneratorBinaryStyle
IBinaryGeneratorStyle
GeneratorBinary::
GeneratorBinaryStyleMeta
GeneratorBinary::
GeneratorBinaryStyleXML

FIG. 33

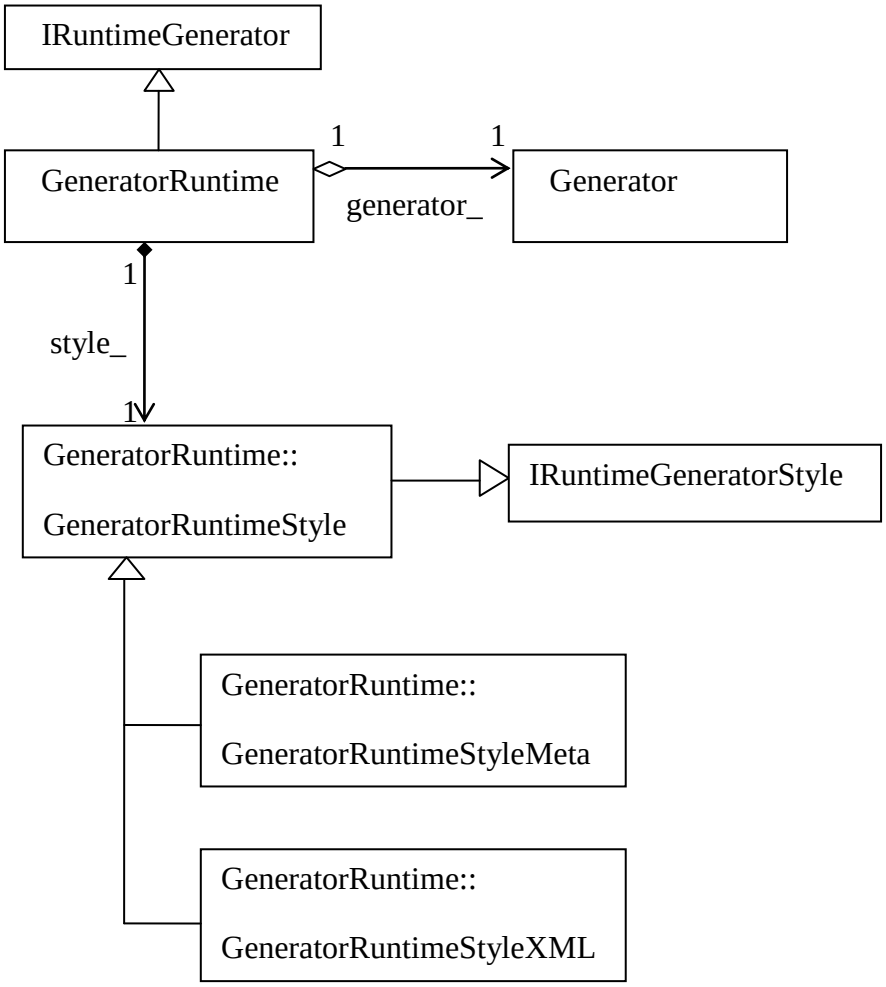
IRuntimeGenerator
GeneratorRuntime
1
1
generator_
Generator
1
style_
1
GeneratorRuntime::
GeneratorRuntimeStyle
IRuntimeGeneratorStyle
GeneratorRuntime::
GeneratorRuntimeStyleMeta
GeneratorRuntime::
GeneratorRuntimeStyleXML

FIG. 34

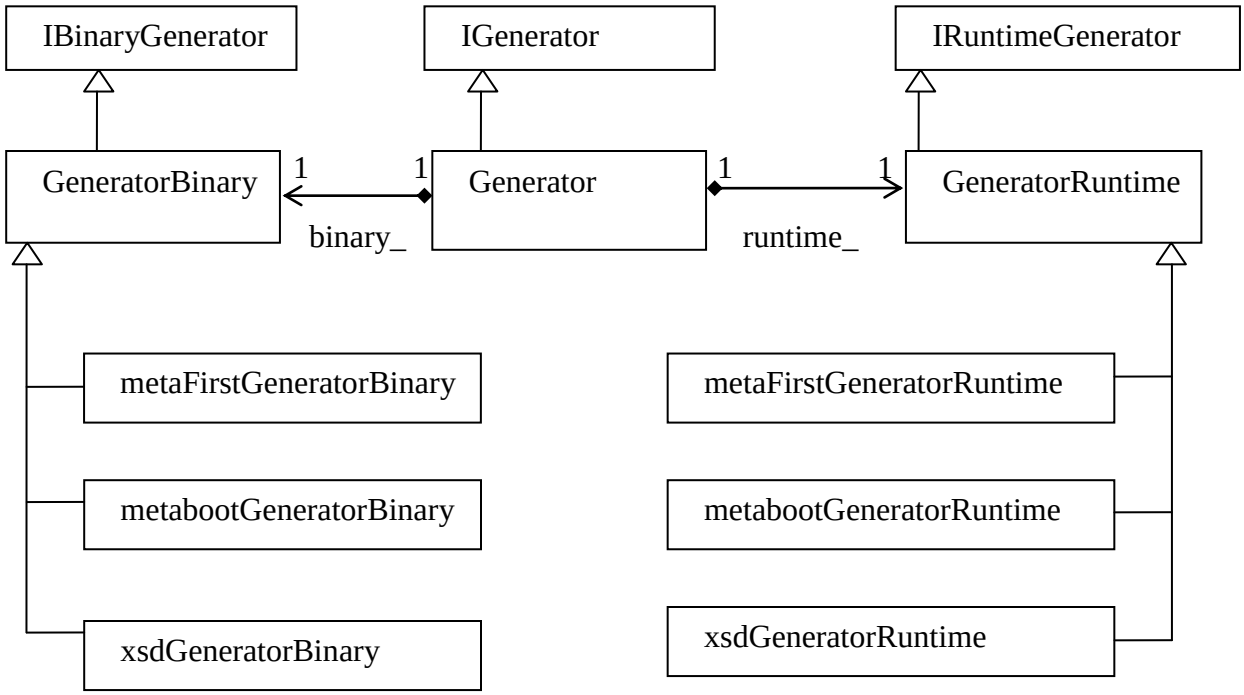
IBinaryGenerator
IGenerator
IRuntimeGenerator
GeneratorBinary
1
1
Generator
1
1
GeneratorRuntime
binary_
runtime_
metaFirstGeneratorBinary
metabootGeneratorBinary
xsdGeneratorBinary
metaFirstGeneratorRuntime
metabootGeneratorRuntime
xsdGeneratorRuntime

FIG. 35

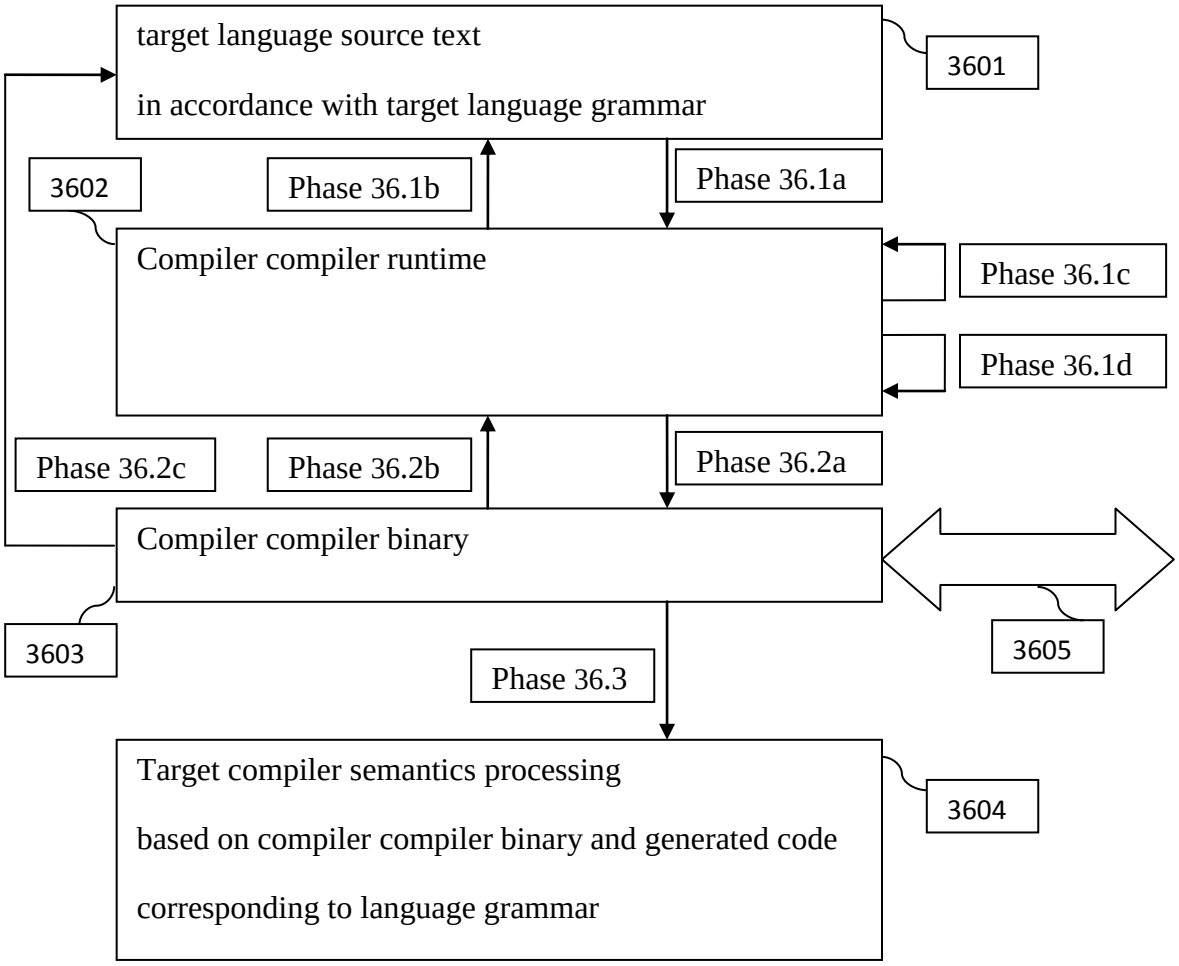
target language source text
in accordance with target language grammar
3601
3602
Phase 36.1b
Phase 36.1a
Compiler compiler runtime
Phase 36.1c
Phase 36.1d
Phase 36.2c
Phase 36.2b
Phase 36.2a
Compiler compiler binary
3603
3605
Phase 36.3
Target compiler semantics processing
based on compiler compiler binary and generated code
corresponding to language grammar
3604

FIG. 36

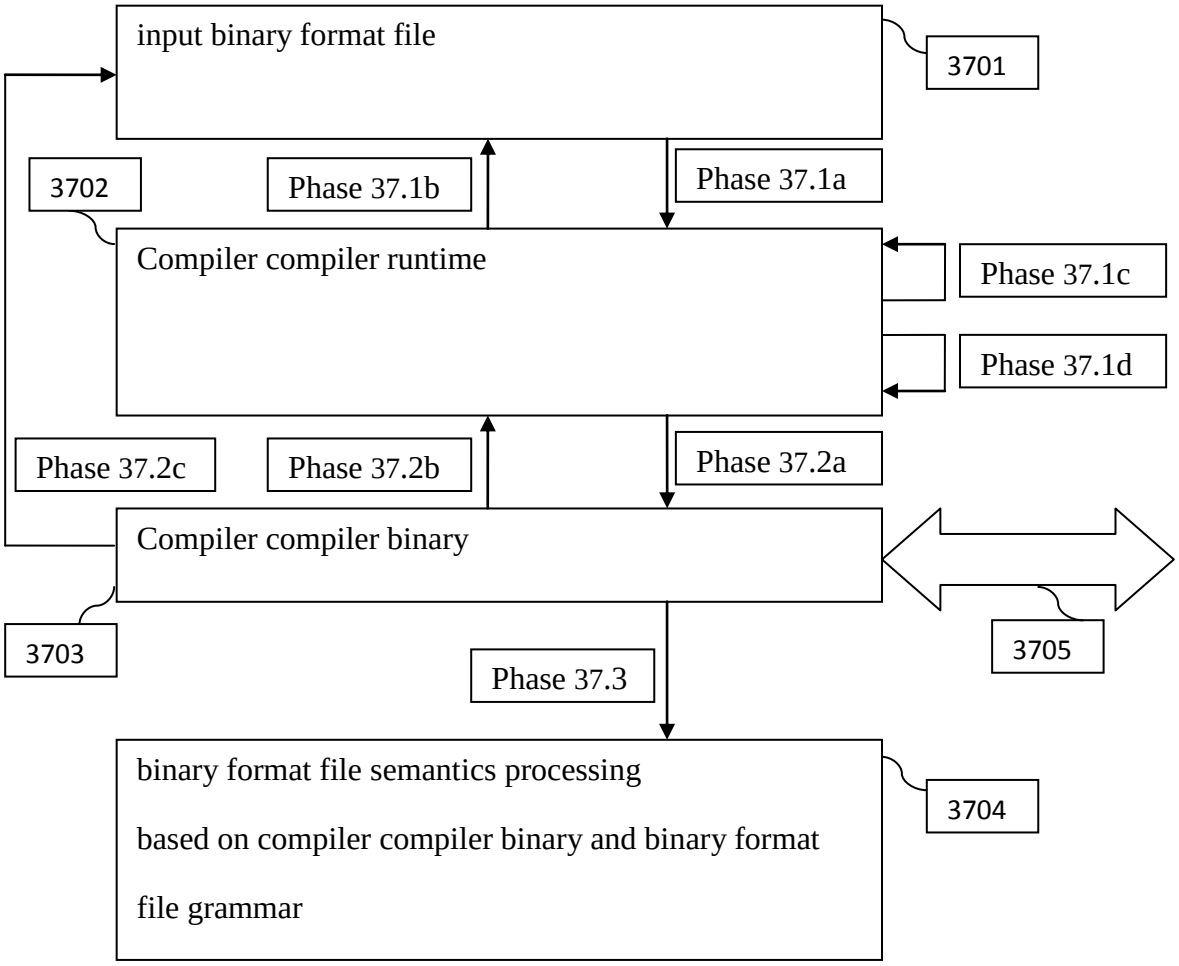
input binary format file
3701
3702
Phase 37.1b
Phase 37.1a
Compiler compiler runtime
Phase 37.1c
Phase 37.1d
Phase 37.2c
Phase 37.2b
Phase 37.2a
Compiler compiler binary
3703
3705
Phase 37.3
binary format file semantics processing
based on compiler compiler binary and binary format
file grammar
3704

FIG. 37

FIG. 1 shows a prior art compiler compiler such as YACC [1] or GNU BISON.

FIG. 2 shows compiler compiler system phases according to the invention for building the compiler compiler itself.

FIG. 3 shows compiler compiler system phases according to the invention for building a target compiler.

FIG. 4 shows target compiler phases when a target compiler is built by a compiler compiler system according to the invention.

FIG. 5 shows a UML diagram of compiler compiler management classes such as Shell, Compiler, Parser, and other related classes and their relationships.

FIG. 6 shows a UML diagram of compiler compiler management KeyWordsContainer class.

FIG. 7 shows a UML diagram of a compiler compiler management Parser class and its relationships with other classes.

FIG. 8 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime class.

FIG. 9 shows a UML diagram of a compiler compiler runtime MapVectorContainer template class.

FIG. 10 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::ContextContainer inner class based on a MapVectorContainer template class.

FIG. 11 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::Context inner class and its relationships with SyntaxControlledRuntime::NameContainer and SyntaxControlledRuntime::SymbolContainer inner classes.

FIG. 12 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::NameContainer inner class based on a MapVectorContainer template class.

FIG. 13 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::Name inner class.

FIG. 14 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::SymbolContainer inner class based on a MapVectorContainer template class.

FIG. 15 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::Symbol inner class.

FIG. 16 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::RuleContainer inner class based on MapVectorContainer template class.

FIG. 17 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime::Rule inner class.

FIG. 18 shows a UML diagram of a compiler compiler runtime SyntaxControlledRuntime class and its inner classes with their relationships regardless of the implementation details of the containers.

FIG. 19 shows a UML diagram of compiler compiler binary SyntaxControlledBinary class and its inner classes with their relationships.

FIG. 20 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinaryContext class and other SyntaxControlledBinary inner classes with their relationships.

FIG. 21 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinaryContextHeader class.

FIG. 22 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinarySymbol class.

FIG. 23 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinaryRule class.

FIG. 24 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinaryRuleDefinition class.

FIG. 25 shows a UML diagram of a compiler compiler management TypeInstance class.

FIG. 26 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary::BinaryHelper class.

FIG. 27 shows a UML diagram of a compiler compiler management Tokenizer class and its relationships with LineReader, Language, andTokenNameContainer classes.

FIG. 28 shows a UML diagram of a compiler compiler management TokenNameContainer class.

FIG. 29 shows a UML diagram of a compiler compiler management Language class and its relationships with related classes.

FIG. 30 shows a UML diagram of a compiler compiler management Tokens class and its relationships with related classes.

FIG. 31 shows a UML diagram of a compiler compiler management Token class hierarchy.

FIG. 32 shows a UML diagram of a compiler compiler generator Generator class and its relationships with related classes.

FIG. 33 shows a UML diagram of a compiler compiler generator GeneratorBinary class and its relationships with related classes.

FIG. 34 shows a UML diagram of a compiler compiler generator GeneratorRuntime class and its relationships with related classes.

FIG. 35 shows a UML diagram of a target compiler's child classes derived from GeneratorBinary and GeneratorRuntime classes.

FIG. 36 shows target compiler standard and transformation phases performed in a multiplatform environment when a compiler compiler system builds a target compiler.

FIG. 37 shows binary files processing based on a compiler compiler system performing standard and transformation phases in a multiplatform environment.

Detailed Description of the Invention

FIG.1 shows an example of a prior art compiler compiler system, e.g. [1]. A compiler executable program 107 is a final step here and it is done as a compiling and linking of a compiler code 106 that consists of compiler source files 105, generated compiler parser, and generated compiler tokenizer. The generated compiler parser is an output of executing parser generator 104 having input file for parser generator 103. The generated compiler tokenizer is an output of executing a tokenizer generator 102 having input file for tokenizer generator.

The input file for parser generator 103 contains a grammar definition in the form of a few sections such as implementation language declarations, compiler compiler declarations, compiler compiler grammar rules specification, and additional implementation language code. For YACC [1] and GNU BISON the implementation language is C language.

The compiler compiler grammar rules specification is a sequence of individual grammar rules. Each grammar rule is defined as a non-terminal on the left side and a potentially empty sequence of terminals and non-terminals on the right side followed by actions. The grammar rule actions are specified as implementation language code with additional features the compiler compiler can understand and convert into implementation language code during generation. Those additional features are related to attributes attached to parsing tree nodes. Usually those attributes have a prefix '$' and the compiler compiler acts like a specialized preprocessor that for each rule finds all occurrences of '$' and generates corresponding implementation language code based on compiler compiler executing environment rules for processing parsing results.

For each grammar rule specified actions are executed every time the parser recognizes that the rule is supposed to be invoked during parsing.

The input file for the tokenizer generator mainly consists of language tokens definitions in the form of token name and token regular expression specifying a valid set of characters comprising the token.

To summarize, the prior art compiler compiler system model forces a compiler developer to define compilation tasks in the form of individual grammar rule actions for the isolated parsing tree context identified with the given rule invocation. As a result, when parsing is done a parsing tree is built along with a custom environment implemented by the compiler developer. Subsequent compilation phases followed by parsing are implemented in terms of that parsing tree and custom environment.

FIG. 2 shows the present invention compiler compiler system phases for building itself. A compiler compiler executable program 205 takes compiler compiler source grammar definition language source text defining itself (meta grammar) 201 and performs a phase 2.1a generating compiler compiler runtime 202 for meta grammar 201. Compiler compiler executable program 205 has an option to de-compile meta grammar generated compiler compiler runtime 202 into a text file (not shown) containing meta grammar executing phase 2.1b. This newly de-compiled meta grammar as a text is identical to meta grammar 201 except for some differences related to supported indentation rules.

Compiler compiler executable program 205 for meta grammar 201 performs phase 2.2a generating compiler compiler binary 203 for compiler compiler runtime 202. The phase 2.2a is implemented as a formal procedure that converts compiler compiler runtime 202 into compiler compiler binary 203. Compiler compiler executable program 205 has an option to de-compile meta grammar from generated compiler compiler binary 203 into a text file (not shown) containing meta grammar executing phase 2.2c. This newly de-compiled meta grammar as a text is identical to meta grammar 201 except for some differences related to supported indentation rules. Compiler compiler executable program 205 has an option to re-create a compiler compiler runtime that is identical to original compiler compiler runtime 202 having compiler compiler binary 203 executing phase 2.2b.

Compiler compiler executable program 205 performs phase 2.3 creating a compiler compiler generated code 204 corresponding to meta grammar 201. The compiler compiler source grammar definition language consists of a grammar name section followed by a sequence of rules where the first rule is also a grammar axiom. As used herein, the grammar name section consists of a single identifier that defines a name of grammar. As an example, when C++ compiler compiler executable program 205 takes the following meta grammar source file:

```
(meta
 (grammar ::=
        0 ="  METAACTBEG();"=
        '(' grammarNameDef
                { rule }
        ')'
        0 ="  METAACTEND();"=
 )
 (grammarNameDef ::= identifier
 )
 (rule ::=          '(' nterm '::=' right ')'
 )
 (nterm ::=       identifier
 )
```

```
 (right ::=          { element }
 )
 (element ::=    identAlt | alternative | identMiss | iteration | action
 )
 (action          ::=        integerToken '=' { stringToken } '='
 )
 (actions ::=     '=' { action } '='
 )
 (identAlt ::=    ntermtermact { Altpart }
 )
 (Altpart ::=      '|' ntermtermact
 )
 (ntermtermact   ::=     ntermterm [ actions ]
 )
 (ntermterm     ::=        nterm | termToken
 )
 (alternative    ::=        '(' identAlt ')'
 )
 (identMiss      ::=        '[' identAlt ']'
 )
 (iteration       ::=        '{' iterItemact iterItems '}'
 )
 (iterItems       ::=        { altIterItem }
 )
 (altIterItem     ::=        '|' iterItemact
 )
 (iterItemact     ::=        iterItem  [ actions ]
 )
 (iterItem        ::=        nterm | maybeNterm
 )
 (maybeNterm  ::=        '<' nterm '>'
 )
)
```

as a result of phase 2.3 the following C++ source files are generated:

- metaGenerator.h
- metaKeyWordDefinition.h
- metaParser.h
- metaGenerator.cc
- metaKeyWordDefinition.cc
- metaParser.cc
- metaMakeGenerators.cc

Note, that the 'meta' prefix in file names corresponds to the grammar name - the first section identifier in source grammar definition language. Note also that

0 =" METAACTBEG();"=

0 =" METAACTEND();"=

are used as a special macro substitution actions defined in form of integer number followed by sequence of string literals enclosed in '=' and '='. Further elements and rules are explained in the following paragraphs.

The compiler compiler source grammar definition language elements such as '(' and ')' are grammar terminals defined as a string literal with enclosed single quotes.

The compiler compiler source grammar definition language element such as

{ rule }

is BNF extension called iteration meaning that enclosed by { and } non-terminal is actually may occur zero or any other number of times.

Note, that, e.g., rule

(iterationExample ::= { element })

is equivalent to rules

(iterationExample ::= element iterationExample)

(iterationExample ::=)

The compiler compiler source grammar definition language rule

(rule ::= '(' nterm '::=' right ')'

)

defines rule as a terminal *'('* followed by non-terminal *nterm* followed by terminal *'::='* followed by non-terminal *right* followed by terminal *')'*.

The compiler compiler source grammar definition language rule

(nterm ::= identifier

)

defines non-terminal *nterm* as *identifier*.

The compiler compiler source grammar definition language rule

(right ::= { element }

)

defines non-terminal *right* as an iteration of *element* non-terminals.

The compiler compiler source grammar definition language rule

(element ::= identAlt | alternative | identMiss | iteration | action

)

defines non-terminal *element* as an alternative of non-terminals on the right side of rule definition separated by '|'. The alternative is BNF extension similar to iteration extension; it is used in cases when non-terminal on the left side of rule definition can be one of non-terminals from the right side. Note, that, e.g., rule

(alternativeExample ::= A | B | C | Z)

is equivalent to rules

(alternativeExample ::= A)

(alternativeExample ::= B)

(alternativeExample ::= C)

(alternativeExample ::= Z)

The compiler compiler source grammar definition language rule

(action ::= integerToken '=' { stringToken } '='

)

defines non-terminal *action* as an *integerToken* followed by terminal *'='* followed by iteration of *stringToken* followed by terminal *'='*. Here *integerToken* and *stringToken* are another compiler compiler source grammar definition language reserved key words similar to *identifier*. *integerToken* defines token that holds integer value. *stringToken* defines token that holds string literal value as an arbitrary sequence of any characters enclosed with double quotes, i.e., ".

The compiler compiler source grammar definition language rule

(actions ::= '=' { action } '='

)

defines non-terminal *actions* as a iteration of *action* enclosed with *'='*.

The compiler compiler source grammar definition language rule

(identAlt ::= ntermtermact { Altpart }

)

defines non-terminal *identAlt* as a *ntermtermact* followed by iteration of *Altpart* non-terminals.

The compiler compiler source grammar definition language rule

(Altpart ::= '|' ntermtermact

)

defines non-terminal *Altpart* as a terminal *'|'* followed by non-terminal *ntermtermact.*

The compiler compiler source grammar definition language rule

(ntermtermact ::= ntermterm [actions]

)

defines non-terminal *ntermtermact* as a non-terminal *ntermterm* followed by *[actions]* meaning that non-terminal *actions* may be omitted. Non-terminal enclosed with [and] is another compiler compiler source grammar definition language BNF extension representing elements that can be omitted. Note, that, e.g., rule

(ommitedElementExample ::= A [W])

is equivalent to rules:

(ommitedElementExample ::= A Welement)

(Welement::= W)

(Welement::=)

The compiler compiler source grammar definition language rule

(ntermterm ::= nterm | termToken

)

defines non-terminal *ntermterm* as an alternative of *nterm* of *termToken. nterm* is defined above.

termToken is another compiler compiler source grammar definition language reserved key word that defines terminal token specification as a string literal enclosed with single quotes.

The compiler compiler source grammar definition language rule

(alternative ::= '(' identAlt ')'

)

defines non-terminal *alternative* as an *identAlt* enclosed with terminals *'('* and *')'*.

The compiler compiler source grammar definition language rule

(identMiss ::= '[' identAlt ']'

)

defines non-terminal *identMiss* as an *identAlt* enclosed with terminals *'['* and *']'*.

The compiler compiler source grammar definition language rule

(iteration ::= '{' iterItemact iterItems '}'

)

defines non-terminal *iteration* as an *iterItemact* followed by non-terminal *iterItems* enclosed with terminals *'{'* and *'}'*.

The compiler compiler source grammar definition language rule

(iterItems ::= { altIterItem }

)

defines non-terminal *iterItems* as an iteration of *altIterItem* non-terminals.

The compiler compiler source grammar definition language rule

(altIterItem ::= '|' iterItemact

)

defines non-terminal *altIterItem* as terminal *'|'* followed by non-terminal *iterItemact.*

The compiler compiler source grammar definition language rule

(iterItemact ::= iterItem [actions]

)

defines non-terminal *iterItemact* as non-terminal *iterItem* followed by *[actions]*.

The compiler compiler source grammar definition language rule

(iterItem ::= nterm | maybeNterm

)

defines non-terminal *iterItem* as an alternative of non-terminals *nterm* and *maybeNterm.*

The compiler compiler source grammar definition language rule

(maybeNterm ::= '<' nterm '>'

)

defines non-terminal *maybeNterm* as non-terminal *nterm* enclosed between terminals *'<'* and *'>'*. The compiler compiler source grammar definition language iteration is actually defined as a sequence of terminals or non-terminals may be followed by actions, and also non-terminals may be enclosed between terminals *'<'* and *'>'* meaning that such non-terminal is allowed to be in iteration only zero or one time. Note, that, e.g., the rule

(anotherIterationExampe :: = { A | B | <X> | <Z> })

is equivalent to the rules

(anotherIterationExampe :: = elem anotherIterationExampe)

(anotherIterationExampe :: =)

(elem ::= A | B | X | Z)

with the limitation that *X* and *Z* are allowed to be defined zero or one times only.

If phase 2.5 is performed for newly compiler compiler generated code corresponding to meta grammar 204, then a new default version of compiler compiler executable program 205 is created with default 'metaGenerator.cc' that has empty implementation. When that default version of compiler compiler executable program 205 is running it has all phase 2.1a, phase 2.1b, phase 2.2a, phase 2.2b, and phase 2.2c available to be executed.

The phase 2.3 is actually implemented in the '<prefix>Generator.cc' file. In case of the C++ compiler compiler system mentioned above, seven files are created automatically without any manual intervention from the developer, not only for meta grammar but also for any other grammar defined in the compiler compiler source grammar definition language. In other words, compiler compiler semantics processing is performed by phase 2.4 and it is done by the '<prefix>Generator.cc' file. That code is actually implemented on top of the generated initial default version.

FIG. 3 shows compiler compiler system phases for building a target compiler. Compiler compiler executable program 205 takes compiler compiler source grammar definition language source text defining target language grammar 301 and performs phase 3.1a generating compiler compiler runtime 302. Compiler compiler executable program 205 has an option to de-compile target language grammar from generated compiler compiler runtime 302 into a text file containing compiler compiler source grammar definition language source text defining target language grammar executing phase 3.1b. This newly de-compiled target language grammar as a text is identical to target language grammar 301 except for some differences related to supported indentation rules.

Compiler compiler executable program 205 performs phase 3.2a generating compiler compiler binary 303 for target language grammar compiler compiler runtime 302. The phase 3.2a is implemented as a formal procedure that converts compiler compiler runtime 302 into compiler compiler binary 303. Compiler compiler executable program 205 has an option to de-compile target language grammar from generated compiler compiler binary 303 into a text file containing compiler compiler source grammar definition language source text defining target language grammar executing phase 3.2c. This newly de-compiled target language grammar as a text is identical to target language grammar 301 except for some differences related to supported indentation rules. Compiler compiler executable program 205 has an option to create compiler compiler runtime 302 having compiler compiler binary 303 executing phase 3.2b.

Compiler compiler executable program 205 performs phase 3.3 creating compiler compiler generated code 304 corresponding to target language grammar 301 directly from 303. When C++ compiler compiler executable program 205 takes target language grammar source file 301 the following C++ source files are generated:

- <Prefix>Generator.h
- <Prefix>KeyWordDefinition.h
- <Prefix>Parser.h
- <Prefix>Generator.cc
- <Prefix>KeyWordDefinition.cc
- <Prefix>Parser.cc
- <Prefix>MakeGenerators.cc

Note, that the <Prefix> prefix in file names corresponds to the grammar name - the first section identifier in source grammar definition language.

If phase 3.5 is performed for newly compiler compiler generated code corresponding to target language grammar 304, then a new default version of target language compiler executable program 305 is created with a default

'<Prefix>Generator.cc' that has empty implementation. When that default version of target language compiler executable program 305 is running, it has all phase 3.1a, phase 3.1b, phase 3.2a, phase 3.2b, and phase 3.2c available to be executed.

The phase 3.3 is actually implemented in '<Prefix>Generator.cc' file corresponding to target language grammar 301. In the case of the C++ compiler compiler system mentioned above, seven files are created automatically without any manual intervention from the developer, not only for meta grammar but also for any other grammar defined in the compiler compiler source grammar definition language. In other words, compiler compiler semantics processing is performed by phase 3.4, and it is done by the '<Prefix>Generator.cc' file corresponding to the target language compiler. That code is implemented on top of the generated initial default version with no manual actions. So, phase 3.3 performs the same steps as phase 2.3, but phase 3.4 is related to semantics processing of the target language compiler.

FIG.4 shows phases of target language compiler executable program 305. When the target language compiler executable program 305 takes target language source text in accordance with target language grammar 401 its parser goes through phase 4.1a creating a compiler compiler runtime 402 corresponding to target language grammar 401. In accordance with FIG.3 target language compiler executable program 305 is built with the same set of operations as the compiler compiler executable program 205 shown in FIG.2. The target language executable program 305 has an option to de-compile source program from generated compiler compiler runtime 402 into a source text file in accordance with target language grammar executing phase 4.1b. This newly de-compiled source text is identical to original target language grammar source text 401 except for some differences related to supported indentation rules.

Target language compiler executable program 305 performs phase 4.2a generating compiler compiler binary 403 for compiler compiler runtime 402. Phase 4.2a is implemented as a formal procedure that converts compiler compiler runtime 402 into compiler compiler binary 403. Target language compiler executable program 305 has an option to de-compile source text from generated compiler compiler binary 403 into a source text file in accordance with target language grammar 401 executing phase 4.2c. This newly de-compiled target language source text is identical to original target language source text 401 except for some differences related to supported indentation rules. The target language compiler executable program 305 has an option to create compiler compiler runtime 402 having compiler compiler binary 403 executing phase 4.2b.

The target language compiler executable program 305 performs phase 4.3, a target compiler semantics processing 404 based on compiler compiler binary 403.

To summarize FIG. 2, FIG. 3, and FIG. 4, compiler compiler management, generator, runtime and binary source code are compiled into a compiler compiler foundation library that is used when compiler compiler executable program compiles compiler compiler source grammar definitions and when target compiler executable program compiles target language programs.

FIG. 5 shows a Unified Modeling Language (UML) diagram of compiler compiler management classes such as Logger, Shell, Compiler, Parser, and other related classes and their relationships. Logger class is defined for logging any activities performed during compilation/de-compilation. Shell class is defined for interacting with main routine performing compilation/de-compilation operations. Given Shell instance maintains a container of Compiler instances. Shell instance allocates Logger instance. Compiler class is defined for performing compilation/de-compilation operations under Shell control. Action abstract base class is defined for compilation/de-compilation interface to be instantiated inside Compiler. ActionCompile and ActionDecompile are classes derived from Action class to perform compilation/de-compilation operations. KeyWordsContainer class is defined to maintain set of names related to grammar such as predefined key word, tokens, key words, non-terminals. This class is instantiated inside Compiler instance. Parser class defines generic class as a parent for all generated parser classes for given

grammar. This class is instantiated inside Compiler instance. IRun defines common interface in form of abstract class for Shell, Compiler, and Action classes. The compiler compiler executable program source code contains main function implementation by instantiating Shell with provided program arguments and executing run method of Shell instance.

FIG. 6 shows a UML diagram of compiler compiler management KeyWordsContainer class. It has inner type KeyWordsContainer::NameVector defined as std::vector<std::string>. KeyWordsContainer data members are of KeyWordsContainer::NameVector type; they are predefined_ , tokens_ , keyWords_ , and nonTerminals_. So, all grammar symbol names are categorized into those four groups, the last one for grammar non-terminals, while the first three divide grammar terminals into predefined tokens such as identifier, integerToken, etc..., tokens as sequences of any characters, and tokens as reserved key words.

FIG. 7 shows a UML diagram of compiler compiler management Parser class and its relationships with other classes. Parser class is derived from abstract class IParser that has few methods for compilation from file source with/without listing and from string source with/without listing. Parser class has references to Logger and KeyWordsContainer actually instantiated by Shell class and propagated to Parser through Compiler. Parser class has data members context_ and axiom_ of type Tag defined in C++ version at some namespace as follows:

```
#ifdef _CPPCC_TAG_32BITS_
typedef unsigned long    UnsignedTag;
typedef long             Tag;
typedef float            Real;
struct  TypeInstance {
  UnsignedTag            symbolID:10;
  UnsignedTag            ruleID:22;
};
#else
typedef unsigned long long      UnsignedTag;
typedef long long               Tag;
typedef double                  Real;
struct  TypeInstance {
  UnsignedTag           symbolID:16;
  UnsignedTag           ruleID:48;
};
#endif
```

Along with Tag, UnsignedTag, Real, and TypeInstance are defined the same way. For 32 bit computer architecture all those types occupy a four-byte word; for 64 bit computer architecture all those types occupy an eight-byte word. TypeInstance instance in many cases is represented as a Tag value with subsequent fields packing/unpacking operations.

The Parser class instantiates some simple data members such as file name as a string class instance, as well as Boolean flags such as debugging flag, XML token indicator flag, etc... shown in FIG. 7.

The Parser class shown on FIG. 7 also instantiates other important members such as runtime_ of type SyntaxControlledRuntime, a binary_ of type SyntaxControlledBinary, a tokenizer of type Tokenizer_, and a generator_ of type Generator.

FIG. 8 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime class. This class has currentContext_ data member of type Tag, debug_ and optimizationMode_ flags of type int, and contexts_ of type SyntaxControlledRuntime::ContextContainer. The purpose of SyntaxControlledRuntime class is to define syntax-controlled runtime API as a collection of its methods and methods of inner classes and their relationships designed in accordance with compiler compiler parsing model.

As described in the following paragraphs with reference to FIGS. 9-18, the compiler compiler runtime syntax-controlled API, including various inner classes, is designed to perform the following set of operations to be invoked from any parser generated by compiler compiler system:

- Create new 'Context' instance returning 'ContextID'.

- Create new 'Rule' instance for the current 'Context' instance having 'SymbolID', returning 'Tag' instance actually mapped to 'TypeInstance' class instance with packed 'symbolID' and 'ruleID'.

- Return 'Symbol' instance reference for the current 'Context' instance having 'SymbolID'.

- Return 'Rule' instance for the current 'Context' instance having 'Tag' instance actually mapped to 'TypeInstance' class instance with packed 'symbolID' and 'ruleID'.

- Modify 'Rule' instance dynamic part for the current 'Context' instance having 'Rule' instance reference and vector reference representing dynamic part.

- Modify 'Rule' instance fixed part for the current 'Context' instance having 'Rule' instance reference and vector reference representing fixed part.

- Create identifier representation having string representing identifier and returning identifier index.

FIG. 9 shows a UML diagram of compiler compiler runtime MapVectorContainer template class. This template class is designed to support different SyntaxControlledRuntime containers that are defined as SyntaxControlledRuntime inner types instantiating MapVectorContainer template class with different SyntaxControlledRuntime inner classes. In many compilation tasks a sequence of objects must be identified by object name or sequentially by sequential number in accordance with objects definition.

Consider a definition like this:

int z, a, b, w;

If a compiler builds a map from a string representing object name to object instance, then it would be easy and efficient to manipulate those objects by finding them by name. However, the original order of z, a, b, w would disappear since in the map they are ordered differently. MapVectorContainer is designed to provide both effective operations by name and preserving original sequence in the way objects were originally defined having direct access by name (object key) and by index (object sequential number).

On FIG. 9 MapVectorContainer template class takes two formal arguments, <D> representing object type and <K> representing object key type. MapVectorContainer defines three inner types such as MapVectorContainer::Map, MapVectorContainer::Vector, and MapVectorContainer::Data. The MapVectorContainer::Map type is defined as std::map<K, Tag>. The MapVectorContainer::Vector Map type is defined as std::vector<K>. MapVectorContainer::Data type is defined as std::vector<D>. MapVectorContainer template class instantiates name2index_ instance of type MapVectorContainer::Map, index2name_ instance of type MapVectorContainer::Vector, and index2data_ instance of type MapVectorContainer::Data.

FIG. 10 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::ContextContainer inner class based on MapVectorContainer template class. SyntaxControlledRuntime::ContextContainer inner class is MapVectorContainer template class instance with arguments K=Tag and D=SyntaxControlledRuntime::Context.

FIG. 11 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::Context inner class and its relationships with SyntaxControlledRuntime::NameContainer and SyntaxControlledRuntime::SymbolContainer inner classes. SyntaxControlledRuntime::Context has data members contextID_ of type Tag, names_ of type SyntaxControlledRuntime::NameContainer, and symbols_ of type SyntaxControlledRuntime::SymbolContainer.

FIG. 12 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::NameContainer inner class based on MapVectorContainer template class. SyntaxControlledRuntime:: NameContainer inner class is MapVectorContainer template class instance with arguments K=std::string and D=SyntaxControlledRuntime::Name.

FIG. 13 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::Name inner class having contextID_ data member of type Tag and name_ data member of type std::string.

FIG. 14 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::SymbolContainer inner class based on MapVectorContainer template class. SyntaxControlledRuntime::SymbolContainer inner class is MapVectorContainer template class instance with arguments K=Tag and D=SyntaxControlledRuntime::Symbol.

FIG. 15 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::Symbol inner class having contextID_ and symbolID_ data members of type Tag and rules_ data member of type SyntaxControlledRuntime::RuleContainer.

FIG. 16 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::RuleContainer inner class based on MapVectorContainer template class. SyntaxControlledRuntime:: RuleContainer inner class is MapVectorContainer template class instance with arguments K=Tag and D=SyntaxControlledRuntime::Rule.

FIG. 17 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime::Rule inner class class having contextID_, symbolID_, and ruleInstanceID_ data members of type Tag and fixed_ and dynamic_ data members of type std::vector<Tag>.

FIG. 18 shows a UML diagram of compiler compiler runtime SyntaxControlledRuntime class and its inner classes with their relationships regardless containers implementation details. Actually FIG. 18 defines compiler compiler runtime logical view as a collection of individual context. Logically each compiler compiler runtime context

maintains its own collection of names ordered alphabetically or by sequential number with direct access by name or sequential number.

Logically each compiler compiler runtime context maintains its own collection of symbols ordered by symbolID with direct access by symbolID.

Logically each compiler compiler runtime symbol maintains its own collection of rules representing each rule invocation during parsing for a given symbol.

Actually, compiler compiler runtime logical view and compiler compiler binary logical view are the same since compiler compiler runtime and compiler compiler binary are interchangeable. However, compiler compiler runtime is designed to be an effective environment for processing parsing results during parsing itself, and compiler compiler binary is designed to be an effective environment for processing final parsing results in read only mode serving as a multiplatform interchange format.

In other words, FIG.18 shows one form of compiler compiler parsing model with entities such as Context, Name, Symbol, and Rule with their relationships.

As described in the following paragraphs with reference to FIGS. 19-26, the compiler compiler binary syntax-controlled API, including various inner classes, is designed to perform the following set of operations to be invoked from any application responsible for any semantics processing based on compiler compiler system:

- Get const pointer to 'SyntaxControlledBinary::Context::BinaryHeader'.

- Get pointer to 'SyntaxControlledBinary::Context::BinaryHeader'.

- Get identifier as a const char* having its sequential number.

- Get identifier as a const char* having its alphabetic number.

- Get identifier sequential number having its alphabetic number.

- Get pointer to 'SyntaxControlledBinary::Context::BinarySymbol' having symbol sequential number.

- Get pointer to 'SyntaxControlledBinary::Context::BinarySymbol' having 'SymbolID'.

- Get pointer to char having offset in 'memory_' data member.

- Get const pointer to char having offset in 'memory_' data member.

- Return aligned size of anything to 'Tag' size with given initial size.

- Get pointer to 'SyntaxControlledBinary::Context::BinaryRule' having 'SyntaxControlledBinary::Context:: BinarySymbol' instance pointer and rule instance number given for that symbol.

- Populate 'SyntaxControlledBinary::Context::BinaryRuleContent' by its reference having 'SyntaxControlledBinary::Context:: BinarySymbol' instance pointer and 'SyntaxControlledBinary::Context::BinaryRule' instance pointer.

- Get rule fixed part as a pointer to 'Tag' having 'SyntaxControlledBinary::Context::BinaryRule' instance pointer.

- Get rule fixed part as a reference to 'TypeInstance' having 'SyntaxControlledBinary::Context::BinaryRule' instance pointer and rule instance number.

- Get rule dynamic part as a pointer to 'Tag' having 'SyntaxControlledBinary::Context::BinaryRule' instance pointer.

- Get rule dynamic part as a reference to 'TypeInstance' having 'SyntaxControlledBinary::Context::BinaryRule' instance pointer and rule instance number.

- Write binary into file having file name.

- Read binary from file having file name.

FIG. 19 shows a UML diagram of a compiler compiler binary SyntaxControlledBinary class and its inner classes with their relationships. SyntaxControlledBinary class has contexts_ of type SyntaxControlledBinary::BinaryContextContainer, i.e., SyntaxControlledBinary is a collection of individual binary contexts each of them is represented by inner class SyntaxControlledBinary::BinaryContext. SyntaxControlledBinary::BinaryContextContainer is defined as std::map from Tag to SyntaxControlledBinary::BinaryContext.

FIG. 20 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinaryContext class and other SyntaxControlledBinary inner classes with their relationships. SyntaxControlledBinary::BinaryContext class has only one data member, memory_ that is of type std::vector<Tag> meaning that SyntaxControlledBinary::BinaryContext is represented as a raw memory where SyntaxControlledBinary::BinaryContextHeader, SyntaxControlledBinary::BinarySymbol, SyntaxControlledBinary::BinaryRule class instances are actually allocated.

SyntaxControlledBinary shown in FIG. 19 can be converted into a single binary object specified by the following Backus-Naur Form (BNF):

(1) SyntaxControlledBinary ::= header catalog elements
(2) header ::= headerSize headerTotal headerCatalogOffset headerElementsOffset
(3) headerSize ::= integerToken
(4) headerTotal ::= integerToken
(5) headerCatalogOffset ::= integerToken
(6) headerElementsOffset ::= integerToken
(7) catalog ::= { catalogElement }
(8) catalogElement ::= catalogElementContextID catalogElementOffset catalogElementSize
(9) catalogElementContextID ::= integerToken
(10) catalogElementOffset ::= integerToken
(11) catalogElementSize ::= integerToken
(12) elements ::= { SyntaxControlledBinaryContext }
(13) SyntaxControlledBinaryContext ::=
BinaryContextHeader
BinaryContextSequentialNames
BinaryContextAlphabeticalNames
BinaryContextNames
BinaryContextSymbols
BinaryContextRules
BinaryContextRuleElements
(13.1) BinaryContextHeader ::=
BinaryContextHeaderLength
BinaryContextHeaderCurrent

BinaryContextHeaderAxiom
BinaryContextHeaderNumberOfIdentifiers
BinaryContextHeaderStartOfIdentifiersMap
BinaryContextHeaderNumberOfSymbol
BinaryContextHeaderStartOfSymbols

(13.1.1) BinaryContextHeaderLength ::= integerToken
(13.1.2) BinaryContextHeaderCurrent ::= integerToken
(13.1.3) BinaryContextHeaderAxiom ::= integerToken
(13.1.4) BinaryContextHeaderNumberOfIdentifiers ::= integerToken
(13.1.5) BinaryContextHeaderStartOfIdentifiersMap ::= integerToken
(13.1.6) BinaryContextHeaderNumberOfSymbol ::= integerToken
(13.1.7) BinaryContextHeaderStartOfSymbols ::= integerToken
(13.2) BinaryContextSequentialNames::= { BinaryContextSequentionalNameOffset }
(13.2.1) BinaryContextSequentionalNameOffset ::= integerToken
(13.3) BinaryContextAlphabeticalNames ::= { BinaryContextAlphabeticalName }
(13.3.1) BinaryContextAlphabeticalName ::=
BinaryContextAlphabeticalNameOffset
BinaryContextAlphabeticalNameID
(13.3.1.1) BinaryContextAlphabeticalNameOffset ::= integerToken
(13.3.1.2) BinaryContextAlphabeticalNameID ::= integerToken
(13.4) BinaryContextNames ::= { BinaryContextNameAligned }
(13.4.1) BinaryContextNameAligned ::= stringToken
(13.5) BinaryContextSymbols ::= { BinaryContextSymbol }
(13.5.1) BinaryContextSymbol ::=
BinaryContextSymbolNumberOfRuleInstances
BinaryContextSymbolID
BinaryContextSymbolStart
(13.5.1.1) BinaryContextSymbolNumberOfRuleInstances ::= integerToken
(13.5.1.2) BinaryContextSymbolID ::= integerToken
(13.5.1.3) BinaryContextSymbolStart ::= integerToken
(13.6) BinaryContextRules ::= { BinaryContextRule }
(13.6.1) BinaryContextRule ::=
BinaryContextRuleFixed
BinaryContextRuleDynamic
BinaryContextRuleStart
(13.6.1.1) BinaryContextRuleFixed ::= integerToken
(13.6.1.2) BinaryContextRuleDynamic ::= integerToken
(13.6.1.3) BinaryContextRuleStart ::= integerToken
(13.7) BinaryContextRuleElements ::= { BinaryContextRuleElement }
(13.7.1) BinaryContextRuleElement ::=
BinaryContextRuleElementFixed | BinaryContextRuleElementDynamic
(13.7.1.1) BinaryContextRuleElementFixed ::= integerToken
(13.7.1.2) BinaryContextRuleElementDynamic ::= integerToken

where memory_ data member of SyntaxControlledBinary::BinaryContext class is represented by rule *(13) SyntaxControlledBinaryContext.*

Note, that this BNF is extended by special extensions such as iteration when some non-terminal is enclosed with '{' and '}' meaning that that non-terminal can occur zero or unlimited number of times; alternative when some non-terminals are separated by '|' meaning that one of them can occur; 'integerToken' is used to represent 4 bytes integer or 8 bytes integer; 'stringToken' is used to represent string literal properly aligned. So,

BinaryContextRuleElements ::= { BinaryContextRuleElement }

is equivalent to

BinaryContextRuleElements ::= BinaryContextRuleElement BinaryContextRuleElements
BinaryContextRuleElements ::=
So,

BinaryContextRuleElement ::=
BinaryContextRuleElementFixed | BinaryContextRuleElementDynamic
is equivalent to

BinaryContextRuleElement ::= BinaryContextRuleElementFixed
BinaryContextRuleElement ::= BinaryContextRuleElementDynamic

FIG. 21 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinaryContextHeader class. It is a struct that has the following fields of type Tag: size_, axiom_, numberOfNames_, startOfNames_, numberOfSymbols_, and startOfSymbols_. The instance of this struct is actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member.

FIG. 22 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinarySymbol class. It is a struct that has the following fields of type Tag: numberOfRuleInstances_ , symbolID_, and symbolStart_. The instances of this struct are actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member.

FIG. 23 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinaryRule class. It is a struct that has the following fields of type Tag: fixedSize_, dynamicSize_, and ruleStart_. The instances of this struct are actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member.

FIG. 24 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinaryRuleDefinition class. It is a struct that has the following fields of type TypeInstance*, pointer to TypeInstance: fixed_ and dynamic_. This struct is an auxiliary type that is not actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member, it has two pointers fixed_ and dynamic_ for fixed and dynamic rule parts actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member.

FIG. 25 shows a UML diagram of compiler compiler management TypeInstance class. The description of FIG.7 also has a definition of TypeInstance class. Its fields symbolID_and ruleID_ are actually packed fields that form 4- or 8- byte word depending on the computer architecture.

FIG. 26 shows a UML diagram of compiler compiler binary SyntaxControlledBinary::BinaryHelper class. This class is an auxiliary type that is not actually allocated inside SyntaxControlledBinary::BinaryContext memory_ data member. This class implements SyntaxControlledBinary logical view similar to FIG. 18. SyntaxControlledBinary::BinaryHelper class has references to SyntaxControlledBinary and SyntaxControlledBinary::BinaryContextHeader, binary_ and header_ respectively. Also it has data members nameByIndex_ and nameByID_ of type std::vector<const char*>, nameIndex2ID_ of type std::vector<Tag>, symbols_ of type std::vector<SyntaxControlledBinary::BinaryHelper::Symbol>. The nameByIndex_ represents all context names ordered sequentially. The nameByID_ represents all context names ordered alphabetically. The nameIndex2ID_ represents mapping from alphabetic order to sequential order. The symbols_ represents a vector container of all context symbols with individual elements of type SyntaxControlledBinary::BinaryHelper::Symbol. The SyntaxControlledBinary::BinaryHelper::Symbol has a pointer reference to SyntaxControlledBinary::BinarySymbol instance as symbol_ member, data member rules_ of type std::vector<SyntaxControlledBinary::BinaryHelper::Rule>. The SyntaxControlledBinary::BinaryHelper::Rule has a pointer reference to SyntaxControlledBinary::BinaryRule instance as rule_ member.

FIG. 27 shows a UML diagram of compiler compiler management Tokenizer class and its relationships with LineReader, Language, and TokenNameContainer classes. The Tokenizer class has references logger_ to Logger, grammarSymbols_ to KeyWordsContainer, and parser_ to Parser. The Tokenizer class instantiates an instance of TokenNameContainer as grammarSymbols_ member. The Tokenizer class instantiates LineReader as reader_ and Language as language_.

The LineReader class is an abstract base class. The FileLineReader and StringLineReader classes are derived from LineReader. The FileLineReader class provides implementation for reading source text of compiled program from a file, it also maintains the corresponding listing as a file. The StringLineReader class provides implementation for reading source text of compiled program from a std::string as a sequence of lines separated by a given delimiter, it also maintains the corresponding listing. Any other line reader classes can be implemented as derived from LineReader abstract base class.

The LineReader class comprises the following pure virtual functions or their equivalents:

- 'get' returning argument by reference, the next line as a string to be processed by 'Tokenizer'.

- 'put' having const reference to string to be put into listing by 'Tokenizer'.

- boolean 'hasListing' returning true if listing is maintained.

- boolean 'isEof' indicating end of source lines to be processed.

- 'start' having file name and listing file name.

- 'start' having const reference to string to be processed as a sequence of lines and boolean flag if listing is required.

The Language class is an abstract base class. The LanguageMeta and LanguageXML classes are derived from the Language class. The LanguageMeta class implements tokens set and Language interface for compiler compiler source grammar definition language. The LanguageXML class implements tokens set and Language interface for XML type of languages.

FIG. 28 shows a UML diagram of compiler compiler management TokenNameContainer class. It has a data member tokenNames_ of type TokenNameContainer::TokenNameMap defined as std::map<std::string, std::string>.

FIG. 29 shows a UML diagram of compiler compiler management Language class and its relationships with related classes. The Language is an abstract base class derived from abstract base class ILanguage that consists of a set of pure virtual functions defining its interface. The Language class has a reference tokenizer_ to Tokenizer class. The Language class has a data member tokenizerSet_ of enumeration (a.k.a. enum) type LanguageTokenizerSet. The Language class instantiates three instances such as tokens_, keyWords_, and nonTerminals_ of type Tokens.

The 'Language' abstract class comprises the following pure virtual functions or their equivalents from the set of functions defining the ILanguage interface:

- 'getCharacter' populating 'Tokenizer' 'currentCharacter'.

- 'getIdentifier' populating 'Tokenizer' 'currentIdentifier' and 'currentKeyWord'.

- 'getNumeric' populating 'Tokenizer' 'currentDoubleValue' or 'currentIntegerValue' depending on token value, and 'Tokenizer' 'currentKeyWord'.

- 'getString' populating 'Tokenizer' 'currentStringValue' and 'currentKeyWord'.

- 'getNextToken' populating 'Tokenizer' next token and 'currentKeyWord'.

In addition, the 'Language' class comprises generic functions common to all languages as derived classes from the 'Language' class; those generic functions are:

- 'populateTerminals' populating 'SymbolID' for each token defined in 'terminalTokens_', 'terminalKeyWords_', and 'nonTerminals_'.

- 'isDefinedToken' returning true or false depending on check of all tokens defined in 'terminalTokens_' if that token matches the sequence of characters starting from 'Tokenizer' 'currentCharacter'.

- 'isKeyWord' returning true or false depending on check if a given formal parameter is defined in corresponding name to index container of 'terminalKeyWords_'.

- 'setNextToken' setting 'Tokenizer' 'currentKeyWord'

by performing loop for each token size 'k' starting from maximum size while 'k' greater than zero reducing 'k' by one

by performing loop for each token 't' in 'terminalTokens_'

checking if 'k' equals to token 't' size and token 't' isToken predicate is true then

executing token 't' 'flushToken' function (see description of the Token class below);

setting 'Tokenizer' 'currentKeyWord' as token 't' 'SymbolID'.

- 'getToken' having a formal argument as an input with token 'SymbolID' returning corresponding reference to 'Token' class instance.

- 'skipToken' having a first formal argument as an input with token 'SymbolID' and a second formal argument as an output with token reference to 'Tag' actually mapped to 'TypeInstance' class instance with token 'SymbolID' and token 'RuleID'; performing call to 'getToken' with the first formal argument as an input with token 'SymbolID' getting reference 't' to 'Token' class instance; performing call to 't' method 'skipToken' with the first formal argument as an input with token 'SymbolID' and the second formal argument as an output with token reference to 'Tag'; performing call to 'getNextToken'.

- 'getTerminalString' setting 'Tokenizer' 'currentTerminalStringValue' and related 'Tokenizer' data members processing sequence of input characters enclosed with single quotes representing language defined terminal as a sequence of characters enclosed with single quotes.

FIG. 30 shows a UML diagram of compiler compiler management Tokens class and its relationships with related classes. The Tokens class has data members name2index_ of type Tokens::Name2IndexMap and tokens_ of type Tokens::TokenVector. The type Tokens::Name2IndexMap is defined as std::map<std::string, int> mapping token name to token index for getting access to Token in tokens_ container. The type Tokens::TokenVector is defined as std::vector<Token*> and it will be replaced by std::vector<std::shared_ptr<Token>> when std::shared_ptr will be widely available as C++ standard library class.

The Token class is an abstract base class that is derived from abstract base class IToken that consists of a set of pure virtual functions defining its interface. The Token class has a reference tokenizer_ to Tokenizer class instance. The

Token class has a data member symbol_ that is actually corresponds to SymbolID of Symbol related to the given Token. The Token class has data members token_ and name_ of std::string type.

The 'Token' class comprises pure virtual functions or their equivalents from the set of pure virtual functions defining the IToken interface, such as:

- 'isToken' predicate returning true or false depending on if 'Tokenizer' sequence of characters starting from 'Tokenizer' 'currentCharacter' matches given token 'token_' data member.

- 'flushToken' checking if 'isToken' predicate returns true, then advancing 'Tokenizer' 'currentCharacter' to the next one beyond the given token sequence of characters; and executing 'Tokenizer' 'flushBlanks' function skipping white space characters such as blank, tabulation, etc.

FIG. 31 shows a UML diagram of compiler compiler management Token class hierarchy. The OneCharToken, TwoCharToken, ThreeCharToken are derived from Token class abstract class implementing its interface. The OneCharXMLGreaterThanToken is derived from OneCharToken providing custom implementation of '>' token in case of XML type of languages.

FIG. 32 shows a UML diagram of compiler compiler generator Generator class and its relationships with related classes. The Generator class is derived from IGenerator abstract base class that consists of a set of pure virtual functions or their equivalents defining its interface, such as:

- 'decompileRuntime' having file name as a formal argument where runtime content is de-compiled.

- 'decompileBinary' having file name as a formal argument where binary content is de-compiled.

- 'generateFromRuntime' having file name as a formal argument that is used to generate output based on runtime content.

- 'generateFromBinary' having file name as a formal argument that is used to generate output based on binary content.

The Generator class implements IGenerator interface; i.e., implements the foregoing functions or their equivalents. The Generator class has a reference parser_ to the Parser instance. The Generator class has a data member tokenizerSet_ of enumerated type LanguageTokenizerSet. . The Generator class instantiates runtime_ as GeneratorRuntime* and binary_ as GeneratorBinary*. Pointer to GeneratorRuntime, GeneratorRuntime*, and Pointer to GeneratorBinary, GeneratorBinary*, will be replaced by std::shared_ptr<GeneratorRuntime> and std::shared_ptr<GeneratorBinary> when std::shared_ptr will be widely available as C++ standard library class. The GeneratorBinary class is a base class derived from IBinaryGenerator abstract base class that consists of a set of pure virtual functions defining its interface. The GeneratorRuntime class is a base class derived from IRuntimeGenerator abstract base class that consists of a set of pure virtual functions defining its interface.

FIG. 33 shows a UML diagram of compiler compiler generator GeneratorBinary class and its relationships with related classes. The GeneratorBinary class is a base class derived from IBinaryGenerator abstract base class that consists of a set of pure virtual functions defining its interface. The GeneratorBinary class has a reference generator_ to Generator class instance. The GeneratorBinary class instantiates style_ data member of type GeneratorBinary::GeneratorBinaryStyle* that will be replaced by std::shared_ptr< GeneratorBinary::GeneratorBinaryStyle> when std::shared_ptr will be widely available as C++ standard library class. The GeneratorBinary::GeneratorBinaryStyle is an abstract base class that is derived from IBinaryGeneratorStyle that consists of a set of pure virtual functions defining its interface. The

GeneratorBinary::GeneratorBinaryStyleMeta and GeneratorBinary::GeneratorBinaryStyleXML classes are derived from GeneratorBinary::GeneratorBinaryStyle abstract base class.

The IBinaryGenerator abstract base class consists of the following pure virtual functions or their equivalents:

- makeAlignment having formal argument token as a reference to Token returning void;

- decompile having formal argument filename as a const reference to string returning void.

- generate having formal argument filename as a const reference to string returning void.

The IRuntimeGenerator abstract base class consists of the following pure virtual functions or their equivalents:

- makeAlignment having formal argument token as a reference to Token returning void;

- decompile having formal argument filename as a const reference to string returning void;

- generate having formal argument filename as a const reference to string returning void.

The IRuntimeGeneratorStyle abstract base class consists of the following pure virtual functions or their equivalents:

- makeAlignment having formal argument token as a reference to Token returning void.

The IBinaryGeneratorStyle abstract base class consists of the following pure virtual functions or their equivalents:

- makeAlignment having formal argument token as a reference to Token returning void.

FIG. 34 shows a UML diagram of compiler compiler generator GeneratorRuntime class and its relationships with related classes. The GeneratorRuntime class is a base class derived from IRuntimeGenerator abstract base class that consists of a set of pure virtual functions defining its interface. The GeneratorRuntime class has a reference generator_ to Generator class instance. The GeneratorRuntime class instantiates style_ data member of type GeneratorRuntime::GeneratorRuntimeStyle* that will be replaced by std::shared_ptr< GeneratorRuntime::GeneratorRuntimeStyle> when std::shared_ptr will be widely available as C++ standard library class. The GeneratorRuntime::GeneratorRuntimeStyle is an abstract base class that is derived from IRuntimeGeneratorStyle that consists of a set of pure virtual functions defining its interface. The GeneratorRuntime::GeneratorRuntimeStyleMeta and GeneratorRuntime::GeneratorRuntimeStyleXML classes are derived from GeneratorRuntime::GeneratorRuntimeStyle abstract base class.

FIG. 35 shows a UML diagram of target compilers child classes derived from GeneratorBinary and GeneratorRuntime classes. Target compiler related classes such as metaFirstGeneratorBinary, metabootGeneratorBinary, xsdGeneratorBinary, and similar are derived from GeneratorBinary class. Target compiler related classes such as metaFirstGeneratorRuntime, metabootGeneratorRuntime, xsdGeneratorRuntime, and similar are derived from GeneratorRuntime class.

FIG. 36 shows built by compiler compiler system target compiler standard and transformation phases performed in multiplatform environment. This FIG.36 is similar to FIG. 4 with additional elements. The phase 36.1c is a transformation of input compiler compiler runtime into output compiler compiler runtime. This type of transformation could be obfuscation, security protection, or content management. The phase 36.1d is a back transformation of output compiler compiler runtime into input compiler compiler runtime. The compiler compiler binary 3603 could be transmitted or received by 3605 in a binary format to/from other program. Those two sides of

program to program communication can be implemented in different platforms, different operating systems, and different programming languages.

FIG. 37 shows binary files processing based on compiler compiler system performing standard and transformation phases in multiplatform environment. FIG. 37 is similar to FIG. 36. It shows an idea of converting any binary file into compiler compiler runtime. This can be done in form of designing binary file format in term of compiler compiler source grammar definition language. A custom binary file format convertor is required that loads binary file content into compiler compiler runtime. Having compiler compiler runtime built all other operations of FIG.37 are available for free.

While the invention has been described with respect to certain preferred embodiments, as will be appreciated by those skilled in the art, it is to be understood that the invention is capable of numerous changes, modifications and rearrangements and such changes, modifications and rearrangements are intended to be covered by the Patent claims [3].

Parsing Model

The Patent [3] introduces a new conception of Parsing Model for representing parsing results. All parsing results are maintained in a form of collection of Context instances. The Context is an entity representing parsing results dedicated for compiled elements under a single scope having common collection of Name instances. The Name is an entity representing language names: identifiers and reserved key words. There are two relationships between Context and Name: namesByName and namesByNumber providing Name look up by name and by number. In other words, within a given Context each Name is identified by name and by number. Each Context maintains a collection of Symbol instances. Each Symbol represents grammar terminal or non-terminal symbol. Each Symbol maintains a collection of Rule instances.

FIG.18 shows one form of compiler compiler parsing model with entities such as Context, Name, Symbol, and Rule with their relationships defined in SyntaxControlledRuntime class as related inner classes. FIG. 10 - 16 shows other details of those classes based on MapVectorContainer template class presented on FIG. 9. Additional SyntaxControlledRuntime class information will be presented in Chapter "Syntax-Controlled Runtime API".

In this Chapter let's look into details representing parsing results while parser performs its job, i.e., parsing analysis creating parsing results.

metabootParser.h source file

In Chapter "US Patent 8,464,232 summary" FIG. 2 is explained with additional details about source grammar definition language. For bootstrapping C++ compiler compiler an initial handcrafted compiler compiler parser was created in files metabootParser.h and metabootParser.cc files. Here is the content of metabootParser.h file:

```
////////////////////////////////////////////////////////////////////
//  metabootParser.h
////////////////////////////////////////////////////////////////////

#ifndef  _CPPCC_metaboot_PARSER_H_
#define  _CPPCC_metaboot_PARSER_H_

#include "Parser.h"

namespace cppcc {

namespace cmp {
  class Compiler;
}

namespace metaboot {

class metabootParser
  : public cppcc::syn::Parser
{

public:
  metabootParser
    (cppcc::log::Logger&          logger
    ,cppcc::com::KeyWordsContainer&   theKeyWords
    ,cppcc::com::LanguageTokenizerSet   theLanguage
    ,cppcc::lex::TokenizerReader    theReader)
  : Parser(logger, theKeyWords, theLanguage, theReader)
  {
  }
```

```
  ~metabootParser()
  {
  }

  void compile
    (const char *sour, const char *list);

  void compile
      (const std::string& filename
      ,const std::string& sourceString
    ,bool         isListing);


  void grammar(cppcc::scr::tag::Long& tag);
  void grammarNameDef(cppcc::scr::tag::Long& tag);
  void rule(cppcc::scr::tag::Long& tag);
  void nterm(cppcc::scr::tag::Long& tag);
  void right(cppcc::scr::tag::Long& tag);
  void element(cppcc::scr::tag::Long& tag);
  void action(cppcc::scr::tag::Long& tag);
  void actions(cppcc::scr::tag::Long& tag);
  void identAlt(cppcc::scr::tag::Long& tag);
  void Altpart(cppcc::scr::tag::Long& tag);
  void ntermtermact(cppcc::scr::tag::Long& tag);
  void ntermterm(cppcc::scr::tag::Long& tag);
  void alternative(cppcc::scr::tag::Long& tag);
  void identMiss(cppcc::scr::tag::Long& tag);
  void iteration(cppcc::scr::tag::Long& tag);
  void iterItems(cppcc::scr::tag::Long& tag);
  void altIterItem(cppcc::scr::tag::Long& tag);
  void iterItemact(cppcc::scr::tag::Long& tag);
  void iterItem(cppcc::scr::tag::Long& tag);
  void maybeNterm(cppcc::scr::tag::Long& tag);
};

}
}

#endif
```

It is clear from that file that `class metabootParser` is derived from `cppcc::syn::Parser.` The role of `metabootParser` constructor parameters will be explained later in Chapter "C++ CCS overview" along with `cppcc::syn::Parser` class.

Two compile members implement `cppcc::syn::IParser` methods (note that `cppcc::syn::Parser` is derived from `cppcc::syn::IParser`).

metaboot source grammar definition file

The `metaboot` SGDL specification is presented here:

```
(metaboot
 (grammar ::=
        0 ="  METAACTBEG();"=

        '(' grammarNameDef
                        { rule }
        ')'

        0 ="  METAACTEND();"=
 )

 (grammarNameDef ::= identifier
 )

 (rule ::=          '(' nterm '::=' right ')'
 )

 (nterm ::= identifier
 )

 (right ::= { element }
 )

 (element       ::=
                identAlt | alternative | identMiss | iteration | action
 )

 (action        ::= integerToken '=' { stringToken } '='
 )

 (actions       ::= '=' { action } '='
 )

 (identAlt      ::=
                ntermtermact { Altpart }
 )

 (Altpart       ::= '|' ntermtermact
 )

 (ntermtermact ::= ntermterm [ actions ]
 )

 (ntermterm     ::=
                nterm | termToken
 )
```

```
 (alternative     ::=        '(' identAlt ')'
 )

 (identMiss       ::=        '[' identAlt ']'
 )

 (iteration       ::=        '{' iterItemact iterItems '}'
 )

 (iterItems       ::=        { altIterItem }
 )

 (altIterItem     ::= '|' iterItemact
 )

 (iterItemact     ::= iterItem  [ actions ]
 )

 (iterItem        ::=
                    nterm | maybeNterm
 )

 (maybeNterm  ::=
                  '<' nterm '>'
 )

)
```

It has one little difference v.s. meta.sgr: we have here (metaboot instead of (meta. This name (metaboot v.s. meta) acts like a grammar name that is also used as a namespace in a generated C++ code.

metabootParser.cc source file

Also note that each grammar non-terminal has a related method with the same name having one argument defined as: `(cppcc::scr::tag::Long& tag).` We will talk about meaning of that argument after discussing parser code presented in metabootParser.cc file with the following content:

```
////////////////////////////////////////////////////////////////////
//  metabootParser.cc
////////////////////////////////////////////////////////////////////

#include "Shell.h"
#include "Logger.h"
#include "Compiler.h"
#include "Parser.h"

#include "metabootParser.h"
#include "metabootKeyWordDefinition.h"

namespace cppcc {
```

```
namespace metaboot {

void
metabootParser::compile
  (const char *sour, const char *list)
{
  CPPCC_LOG_INFO(logger_,
    << "metabootParser::compile() "
    << " source:" << "'" << std::string(sour?sour:"") << "'"
    << " listing:" << "'" << std::string(list?list:"") << "'"
  )

  filename_ = sour?sour:"";
  tokenizer_.start(sour, list);

  grammar(axiom_);

  tokenizer_.erfini();
}

void
metabootParser::compile
    (const std::string& filename
    ,const std::string& sourceString
  ,bool         isListing)
{
  filename_ = filename;
  tokenizer_.start(sourceString, isListing);

  grammar(axiom_);

  tokenizer_.erfini();
}

void
metabootParser::grammar(cppcc::scr::tag::Long& tag)
/*
(grammar::=0 ="  METAACTBEG();"='(' grammarNameDef{rule}')' 0 ="  METAACTEND();"=)
*/
{
  TagVector d;
  TagVector f(2);

  METAACTBEG();

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  grammarNameDef(f[1]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_LEFTPARENTHESISTERMTOKEN) {
         d.push_back(0);
         rule(d.back());
    }
```

```
    else break;

  }

  d.push_back(0);
  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,d.back());
  crelasedp(KW_GRAMMAR,f,d,&tag);
  METAACTEND();

}

void
metabootParser::grammarNameDef(cppcc::scr::tag::Long& tag)
/*
(grammarNameDef::=identifier)
*/
{

  identifier(tag);

}

void
metabootParser::rule(cppcc::scr::tag::Long& tag)
/*
(rule::='(' nterm'::=' right')' )
*/
{
  TagVector f(5);

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  nterm(f[1]);

  skipToken(KW_DEFISTERMTOKEN,f[2]);

  right(f[3]);

  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,f[4]);
  crefixe(KW_RULE,f,&tag);


}

void
metabootParser::nterm(cppcc::scr::tag::Long& tag)
/*
(nterm::=identifier)
*/
{

  identifier(tag);

}

void
```

```
metabootParser::right(cppcc::scr::tag::Long& tag)
/*
(right::={element})
*/
{
  TagVector d;

  for(;;) {
    if(iseof()) break;

    if( (kword() == KW_LEFTPARENTHESISTERMTOKEN)
      || (kword() == KW_IDENTIFIER)
      || ((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN))
      || (kword() == KW_TERMTOKEN)
      || (kword() == KW_LEFTSQUAREBRACKETTERMTOKEN)
      || (kword() == KW_LEFTBRACETERMTOKEN)
    ){
         d.push_back(0);
         element(d.back());
    }
    else break;

  }

  crelasdyn(KW_RIGHT,d,&tag);

}

void
metabootParser::element(cppcc::scr::tag::Long& tag)
/*
(element::=identAlt|alternative|identMiss|iteration|action)
*/
{

  if((kword() == KW_IDENTIFIER)
   || (kword() == KW_TERMTOKEN)
  ){
    identAlt(tag);
  }
  else if((kword() == KW_LEFTPARENTHESISTERMTOKEN)
  ){
    alternative(tag);
  }
  else if((kword() == KW_LEFTSQUAREBRACKETTERMTOKEN)
  ){
    identMiss(tag);
  }
  else if((kword() == KW_LEFTBRACETERMTOKEN)
  ){
    iteration(tag);
  }
  else if(((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN))
  ){
    action(tag);
  }
```

```
  else {
    pdbkwmis(KW_LEFTPARENTHESISTERMTOKEN);
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_INTEGERTOKEN);
    pdbkwmis(KW_TERMTOKEN);
    pdbkwmis(KW_LEFTSQUAREBRACKETTERMTOKEN);
    pdbkwmis(KW_LEFTBRACETERMTOKEN);
    edber();
  }

}

void
metabootParser::action(cppcc::scr::tag::Long& tag)
/*
(action::=integerToken'=' {stringToken}'=' )
*/
{
  TagVector d;
  TagVector f(2);

  integerToken(f[0]);

  skipToken(KW_EQUALTERMTOKEN,f[1]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_STRINGTOKEN) {
         d.push_back(0);
         stringToken(d.back());
    }
    else break;

  }

  d.push_back(0);
  skipToken(KW_EQUALTERMTOKEN,d.back());
  crelasedp(KW_ACTION,f,d,&tag);

}

void
metabootParser::actions(cppcc::scr::tag::Long& tag)
/*
(actions::='=' {action}'=' )
*/
{
  TagVector d;
  TagVector f(1);

  skipToken(KW_EQUALTERMTOKEN,f[0]);

  for(;;) {
    if(iseof()) break;
```

```
    if((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN)) {
         d.push_back(0);
         action(d.back());
    }
    else break;

  }

  d.push_back(0);
  skipToken(KW_EQUALTERMTOKEN,d.back());
  crelasedp(KW_ACTIONS,f,d,&tag);

}

void
metabootParser::identAlt(cppcc::scr::tag::Long& tag)
/*
(identAlt::=ntermtermact{Altpart})
*/
{
  TagVector d;
  TagVector f(1);

  ntermtermact(f[0]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_VERTICALBARTERMTOKEN) {
         d.push_back(0);
         Altpart(d.back());
    }
    else break;

  }

  crelasedp(KW_IDENTALT,f,d,&tag);

}

void
metabootParser::Altpart(cppcc::scr::tag::Long& tag)
/*
(Altpart::='|' ntermtermact)
*/
{
  TagVector f(2);

  skipToken(KW_VERTICALBARTERMTOKEN,f[0]);

  ntermtermact(f[1]);
  crefixe(KW_ALTPART,f,&tag);


}
```

```
void
metabootParser::ntermtermact(cppcc::scr::tag::Long& tag)
/*
(ntermtermact::=ntermterm[actions])
*/
{
  TagVector f(2);

  ntermterm(f[0]);

  f[1]=0;
    if(kword() == KW_EQUALTERMTOKEN) {
actions(f[1]);
  }
  crefixe(KW_NTERMTERMACT,f,&tag);


}

void
metabootParser::ntermterm(cppcc::scr::tag::Long& tag)
/*
(ntermterm::=nterm|termToken)
*/
{

    if(kword() == KW_IDENTIFIER) {
    nterm(tag);
  }
  else if((kword() == KW_TERMTOKEN)
  ){
    termToken(tag);
  }
  else {
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_TERMTOKEN);
    edber();
  }

}

void
metabootParser::alternative(cppcc::scr::tag::Long& tag)
/*
(alternative::='(' identAlt')' )
*/
{
  TagVector f(3);

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  identAlt(f[1]);

  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,f[2]);
  crefixe(KW_ALTERNATIVE,f,&tag);
```

```
}

void
metabootParser::identMiss(cppcc::scr::tag::Long& tag)
/*
(identMiss::='[' identAlt']' )
*/
{
  TagVector f(3);

  skipToken(KW_LEFTSQUAREBRACKETTERMTOKEN,f[0]);

  identAlt(f[1]);

  skipToken(KW_RIGHTSQUAREBRACKETTERMTOKEN,f[2]);
  crefixe(KW_IDENTMISS,f,&tag);


}

void
metabootParser::iteration(cppcc::scr::tag::Long& tag)
/*
(iteration::='{' iterItemact iterItems'}' )
*/
{
  TagVector f(4);

  skipToken(KW_LEFTBRACETERMTOKEN,f[0]);

  iterItemact(f[1]);

  iterItems(f[2]);

  skipToken(KW_RIGHTBRACETERMTOKEN,f[3]);
  crefixe(KW_ITERATION,f,&tag);


}

void
metabootParser::iterItems(cppcc::scr::tag::Long& tag)
/*
(iterItems::={altIterItem})
*/
{
  TagVector d;

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_VERTICALBARTERMTOKEN) {
         d.push_back(0);
         altIterItem(d.back());
    }
```

```
    else break;

  }

  crelasdyn(KW_ITERITEMS,d,&tag);

}

void
metabootParser::altIterItem(cppcc::scr::tag::Long& tag)
/*
(altIterItem::='|' iterItemact)
*/
{
  TagVector f(2);

  skipToken(KW_VERTICALBARTERMTOKEN,f[0]);

  iterItemact(f[1]);
  crefixe(KW_ALTITERITEM,f,&tag);


}

void
metabootParser::iterItemact(cppcc::scr::tag::Long& tag)
/*
(iterItemact::=iterItem[actions])
*/
{
  TagVector f(2);

  iterItem(f[0]);

  f[1]=0;
    if(kword() == KW_EQUALTERMTOKEN) {
actions(f[1]);
  }
  crefixe(KW_ITERITEMACT,f,&tag);


}

void
metabootParser::iterItem(cppcc::scr::tag::Long& tag)
/*
(iterItem::=nterm|maybeNterm)
*/
{

    if(kword() == KW_IDENTIFIER) {
    nterm(tag);
  }
  else if((kword() == KW_LESSTHENTERMTOKEN)
  ){
    maybeNterm(tag);
```

```
  }
  else {
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_LESSTHENTERMTOKEN);
    edber();
  }

}

void
metabootParser::maybeNterm(cppcc::scr::tag::Long& tag)
/*
(maybeNterm::='<' nterm'>' )
*/
{
  TagVector f(3);

  skipToken(KW_LESSTHENTERMTOKEN,f[0]);

  nterm(f[1]);

  skipToken(KW_GREATERTHENTERMTOKEN,f[2]);
  crefixe(KW_MAYBENTERM,f,&tag);


}

}

namespace com {
cppcc::syn::Parser*
makeParser(cppcc::cmp::Compiler& compiler)
{
  CPPCC_LOG_INFO((*compiler.shell_.logger_),
    << "com::makeParser() started..."
    << " tokensSetID:" << compiler.shell_.tokensSetID_
  )

  return
    new cppcc::metaboot
      ::metabootParser
      ((*compiler.shell_.logger_)
      ,compiler.keyWords_
      ,compiler.shell_.tokensSetID_? compiler.shell_.tokensSetID_ :
cppcc::com::LanguageTokenizerSet_Meta
      ,cppcc::lex::TokenizerReader_File)
    ;
}
}

}
```

Note, that presented parser is so called recursive descent parser for LL(1) grammar [2].

C++ compiler compiler parser generator rules

The two metabootParser::compile methods are similar, they set up file name, start tokenizer, call grammar axiom, and finalize tokenizer work. When grammar axiom is called (grammar axiom is the first rule in grammar definition, and in our case it is metabootParser::grammar) the tag it returns is saved in axiom_ member.

The `metabootParser::grammar` method is defined as follows:

```
void
metabootParser::grammar(cppcc::scr::tag::Long& tag)
/*
(grammar::=0 ="  METAACTBEG();"='(' grammarNameDef{rule}')' 0 ="  METAACTEND();"=)
*/
{
  TagVector d;
  TagVector f(2);

  METAACTBEG();

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  grammarNameDef(f[1]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_LEFTPARENTHESISTERMTOKEN) {
         d.push_back(0);
         rule(d.back());
    }
    else break;

  }

  d.push_back(0);
  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,d.back());
  crelasedp(KW_GRAMMAR,f,d,&tag);
  METAACTEND();

}
```

The compiler compiler generating parser applies the following rules:
(pg.0) Namespace name comes from grammar name, the first name followed by the first '(' in a grammar definition.
(pg.1) Each grammar non-terminal has its own parser method with the same name having a single output parameter with type cppcc::scr::tag::Long&.
(pg.2) Each non-terminal parser method has a C-style comment where grammar rule is presented, and that comment is a result of de-compilation of rule definition based on corresponding Syntax-Controlled Binary API.
(pg.3) Each non-terminal parser method has a compound statement enclosed in '{' and '}'.
(pg.4) Each non-terminal parser method compound statement may start with tag variable declarations; the structure of those declarations depends on grammar rule form. For the case of `metabootParser::grammar` the right part of the rule has two fixed elements, terminal '(' and non-terminal `grammarNameDef` followed by iteration in form of { rule }, followed by terminal ')'. In this case tag variable declarations are:

```
  TagVector d;
  TagVector f(2);
```

where variable 'f' is defined to keep tag results for terminal '(' and non-terminal grammarNameDef, i.e., f[0] keeps terminal '(' tag and f[1] keeps grammarNameDef tag.

The variable 'd' is defined to keep tag results for iteration of rule instances followed by terminal ')' tag, i.e., d[0], d[1], ..., d[n-1] keep rule 0, rule 1, ..., rule n-1 tags, in the order of 'rule' compilation; d[n] has terminal ')' tag.
(pg.5) The rule actions such as

```
0 ="  METAACTBEG();"=
0 ="  METAACTEND();"=
```

act like compiler compiler pragma statements and in this case 0 means a copy of enclosed string into generated code.
(pg.6) Each grammar terminal has an entry in corresponding enum; in our case for '(' terminal there is `KW_LEFTPARENTHESISTERMTOKEN` entry; each grammar terminal from a rule fixed part is processed by parser by calling `skipToken,` e.g.:

```
  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);
```

(pg.7) Each grammar non-terminal from a rule fixed part is processed by parser by calling corresponding (by name) parser method, e.g.:

```
grammarNameDef(f[1]);
```

(pg.8) Compiler compiler parser generator provides mapping between rule fixed part element and corresponding index; in case of `metabootParser::grammar` we have f[0] for '(' and f[1] for `grammarNameDef.`
(pg.9) All grammar rules have fixed part and dynamic part; fixed part can be empty; dynamic part can be empty; dynamic part corresponds to iteration defined as non-terminal enclosed in { and }; it is possible to have an additional terminal followed by iteration; fixed part is everything that is before dynamic part if iteration is defined; only one iteration can be on the right part of any rule.
(pg.10) Compiler compiler parser generator creates for(;;) loop corresponding to iteration; inside that for(;;) loop there is an if statement; that if statement has an entry for each first symbol [2] corresponding to iteration grammar symbol. In case of `metabootParser::grammar` we have:

```
  for(;;) {
    if(iseof()) break;

    if(kword() == KW_LEFTPARENTHESISTERMTOKEN) {
         d.push_back(0);
         rule(d.back());
    }
    else break;

  }
```

meaning that grammar `rule` non-terminal can only start with '(', i.e., the first symbol set of grammar `rule` non-terminal has only one terminal symbol, that is '('.
(pg.11) When fixed and dynamic rule parts are populated, then rule populating action is called; there are few rule populating actions depending on fixed/dynamic parts being empty. The `crelasedp` is called when fixed and dynamic rule parts both are not empty, e.g.:

```
crelasedp(KW_GRAMMAR,f,d,&tag);
```

The `crefixe` is called when dynamic rule part is empty, e.g.:

```
crefixe(KW_RULE,f,&tag);
```

The `crelasdyn` is called when fixed rule part is empty, e.g.:

```
crelasdyn(KW_ITERITEMS,d,&tag);
```

Those strange names `crelasedp, crefixe,and crelasdyn` came from C CCS version, I did not change them.
Those methods are defined in Parser class manipulating with `runtime_` member defined as follows:

```
class Parser : IParser
{

. . .
  cppcc::log::Logger&                   logger_;
  cppcc::com::KeyWordsContainer&        grammarSymbols_;
  cppcc::scr::SyntaxControlledRuntime   runtime_;
  cppcc::scb::SyntaxControlledBinary    binary_;
  cppcc::lex::Tokenizer                 tokenizer_;
  cppcc::scr::tag::Long                 context_;
```

```
  cppcc::scr::tag::Long                axiom_;
  cppcc::gen::Generator                generator_;
  std::string                          filename_;
  bool                                 debug_;
  bool                                 isXmlTokenFlag_;

. . .
```

Chapter "C++ CCS Overview" provides more details on Parser, FIG. 7 has UML diagram defining Parser, and Chapter "Syntax Controlled Runtime API" describes `cppcc::scr::SyntaxControlledRuntime` class and its methods.

The `metabootParser::grammarNameDef` method is defined as follows:

```
void
metabootParser::grammarNameDef(cppcc::scr::tag::Long& tag)
/*
(grammarNameDef::=identifier)
*/
{
  identifier(tag);
}
```

(pg.12) The 'identifier' here is a reserved key word in source grammar definition language representing identifier token. As usual identifier is a sequence of characters, digits, and '_' that starts with character or '_'.

(pg.13) The rule may not have any fixed or dynamic part, in this case the output parameter is populated directly by calling proper method. In case of `metabootParser::grammarNameDef` it is equivalent to reserved key word 'identifier'.

The `metabootParser::rule` method is defined as follows:

```
void
metabootParser::rule(cppcc::scr::tag::Long& tag)
/*
(rule::='(' nterm'::=' right')' )
*/
{
  TagVector f(5);

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  nterm(f[1]);

  skipToken(KW_DEFISTERMTOKEN,f[2]);

  right(f[3]);

  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,f[4]);
  crefixe(KW_RULE,f,&tag);
}
```

(pg.14) This rule has only fixed part with 5 elements. It defines grammar rule as a sequence of terminal '(', non-terminal nterm, terminal '::=', non-terminal right, and terminal ')'.
The `metabootParser::`nterm method is defined as follows:

```
void
metabootParser::nterm(cppcc::scr::tag::Long& tag)
```

```
/*
(nterm::=identifier)
*/
{

  identifier(tag);

}
```
(pg.15) Grammar non-terminal is defined as identifier.

The `metabootParser::`right method is defined as follows:

```
void
metabootParser::right(cppcc::scr::tag::Long& tag)
/*
(right::={element})
*/
{
  TagVector d;

  for(;;) {
    if(iseof()) break;

    if( (kword() == KW_LEFTPARENTHESISTERMTOKEN)
      || (kword() == KW_IDENTIFIER)
      || ((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN))
      || (kword() == KW_TERMTOKEN)
      || (kword() == KW_LEFTSQUAREBRACKETTERMTOKEN)
      || (kword() == KW_LEFTBRACETERMTOKEN)
    ){
         d.push_back(0);
         element(d.back());
    }
    else break;

  }

  crelasdyn(KW_RIGHT,d,&tag);

}
```
(pg.16) Grammar rule has non-terminal on the left and sequence of elements on the right. Grammar right rule represents rule elements on the right. In this case we have iteration of element.
(pg.17) The element first symbol set consists of terminals such as: '(', identifier, integerToken or floatToken, terminalToken, and '{'.
(pg.18) Grammar right rule has only dynamic part.
(pg.19) The integer token or double/float token is represented in source grammar definition language as a reserved key word integerToken. In general for integerToken in source grammar definition language the target language program may have either integer token or float/double token. To distinguish integer token from float/double token tokenizer represents token type as `KW_INTEGERTOKEN` or `KW_FLOATTOKEN,` as a result parser has to check for both, e.g.:

```
...
    if( (kword() == KW_LEFTPARENTHESISTERMTOKEN)
      || (kword() == KW_IDENTIFIER)
      || ((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN))
      || (kword() == KW_TERMTOKEN)
      || (kword() == KW_LEFTSQUAREBRACKETTERMTOKEN)
```

```
      || (kword() == KW_LEFTBRACETERMTOKEN)
    ){
         d.push_back(0);
         element(d.back());
    }
    else break;
...
```

The `metabootParser::`element method is defined as follows:

```
void
metabootParser::element(cppcc::scr::tag::Long& tag)
/*
(element::=identAlt|alternative|identMiss|iteration|action)
*/
{

  if((kword() == KW_IDENTIFIER)
   || (kword() == KW_TERMTOKEN)
  ){
    identAlt(tag);
  }
  else if((kword() == KW_LEFTPARENTHESISTERMTOKEN)
  ){
    alternative(tag);
  }
  else if((kword() == KW_LEFTSQUAREBRACKETTERMTOKEN)
  ){
    identMiss(tag);
  }
  else if((kword() == KW_LEFTBRACETERMTOKEN)
  ){
    iteration(tag);
  }
  else if(((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN))
  ){
    action(tag);
  }
  else {
    pdbkwmis(KW_LEFTPARENTHESISTERMTOKEN);
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_INTEGERTOKEN);
    pdbkwmis(KW_TERMTOKEN);
    pdbkwmis(KW_LEFTSQUAREBRACKETTERMTOKEN);
    pdbkwmis(KW_LEFTBRACETERMTOKEN);
    edber();
  }
}
```

(pg.19) The element itself is represented in form of identAlt (identifier or alternative elements) and they could be: `identAlt, alternative, identMiss, iteration, and action.`

(pg.20) The first symbol set of `identAlt` contains two terminal symbols: identifier and termToken.

(pg.21) The termToken is reserved key word of source grammar definition language representing a sequence of characters enclosed into single quotes, e.g., ')', '::=', 'class', 'struct', that are treated as a single terminal token comprised of those characters.

(pg.22) In case of syntax error, 'Key Word is Missing' error is printed for each terminal in first symbol set of element non-terminal into listing file.

The `metabootParser::`action method is defined as follows:

```
void
metabootParser::action(cppcc::scr::tag::Long& tag)
/*
(action::=integerToken'=' {stringToken}'=' )
*/
{
  TagVector d;
  TagVector f(2);

  integerToken(f[0]);

  skipToken(KW_EQUALTERMTOKEN,f[1]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_STRINGTOKEN) {
         d.push_back(0);
         stringToken(d.back());
    }
    else break;

  }

  d.push_back(0);
  skipToken(KW_EQUALTERMTOKEN,d.back());
  crelasedp(KW_ACTION,f,d,&tag);

}
```

(pg.23) The action rule defines compiler compiler pragma in a form of integerToken followed by terminal '=', followed by iteration of stringToken, followd by terminal '='. There is a fixed (predefined) set of integerToken values, i.e., 0, 1, 2, 3, etc.. Value 0 is used for operation when provided collection of string tokens is substituted into generated code as is.

The `metabootParser::`actions method is defined as follows:

```
void
metabootParser::actions(cppcc::scr::tag::Long& tag)
/*
(actions::='=' {action}'=' )
*/
{
  TagVector d;
  TagVector f(1);

  skipToken(KW_EQUALTERMTOKEN,f[0]);

  for(;;) {
    if(iseof()) break;

    if((kword() == KW_INTEGERTOKEN) || (kword() == KW_FLOATTOKEN)) {
         d.push_back(0);
         action(d.back());
```

```
    }
    else break;

  }

  d.push_back(0);
  skipToken(KW_EQUALTERMTOKEN,d.back());
  crelasedp(KW_ACTIONS,f,d,&tag);

}
```
(pg.24) The actions rule defines compiler compiler pragma in a form of individual action sequence.

The `metabootParser::`identAlt method is defined as follows:

```
void
metabootParser::identAlt(cppcc::scr::tag::Long& tag)
/*
(identAlt::=ntermtermact{Altpart})
*/
{
  TagVector d;
  TagVector f(1);

  ntermtermact(f[0]);

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_VERTICALBARTERMTOKEN) {
         d.push_back(0);
         Altpart(d.back());
    }
    else break;

  }

  crelasedp(KW_IDENTALT,f,d,&tag);

}
```
(pg.25) The identAlt rule defines alternative in a form of a sequence of terminal or non-terminal separated by '|'.

The `metabootParser::`Altpart method is defined as follows:

```
void
metabootParser::Altpart(cppcc::scr::tag::Long& tag)
/*
(Altpart::='|' ntermtermact)
*/
{
  TagVector f(2);

  skipToken(KW_VERTICALBARTERMTOKEN,f[0]);

  ntermtermact(f[1]);
  crefixe(KW_ALTPART,f,&tag);
```

```
}
```

The `metabootParser::`ntermtermact method is defined as follows:

```
void
metabootParser::ntermtermact(cppcc::scr::tag::Long& tag)
/*
(ntermtermact::=ntermterm[actions])
*/
{
  TagVector f(2);

  ntermterm(f[0]);

  f[1]=0;
    if(kword() == KW_EQUALTERMTOKEN) {
actions(f[1]);
  }
  crefixe(KW_NTERMTERMACT,f,&tag);


}
```

(pg.26) The ntermtermact rule defines terminal or non-terminal followed by optional actions.

The `metabootParser::`ntermterm method is defined as follows:

```
void
metabootParser::ntermterm(cppcc::scr::tag::Long& tag)
/*
(ntermterm::=nterm|termToken)
*/
{

    if(kword() == KW_IDENTIFIER) {
    nterm(tag);
  }
  else if((kword() == KW_TERMTOKEN)
  ){
    termToken(tag);
  }
  else {
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_TERMTOKEN);
    edber();
  }

}
```

(pg.27) The ntermterm rule defines terminal or non-terminal.

The `metabootParser::`alternative method is defined as follows:

```
void
metabootParser::alternative(cppcc::scr::tag::Long& tag)
/*
(alternative::='(' identAlt')' )
*/
```

```
{
  TagVector f(3);

  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);

  identAlt(f[1]);

  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,f[2]);
  crefixe(KW_ALTERNATIVE,f,&tag);


}
```
(pg.28) The alternative is defined as identAlt with enclosed '(' and ')'.

The `metabootParser::`identMiss method is defined as follows:

```
void
metabootParser::identMiss(cppcc::scr::tag::Long& tag)
/*
(identMiss::='[' identAlt']' )
*/
{
  TagVector f(3);

  skipToken(KW_LEFTSQUAREBRACKETTERMTOKEN,f[0]);

  identAlt(f[1]);

  skipToken(KW_RIGHTSQUAREBRACKETTERMTOKEN,f[2]);
  crefixe(KW_IDENTMISS,f,&tag);


}
```
(pg.29) The identMiss is defined as identAlt with enclosed '[' and ']'. It is used for cases when some elements could be omitted.

The `metabootParser::`iteration method is defined as follows:

```
void
metabootParser::iteration(cppcc::scr::tag::Long& tag)
/*
(iteration::='{' iterItemact iterItems'}' )
*/
{
  TagVector f(4);

  skipToken(KW_LEFTBRACETERMTOKEN,f[0]);

  iterItemact(f[1]);

  iterItems(f[2]);

  skipToken(KW_RIGHTBRACETERMTOKEN,f[3]);
  crefixe(KW_ITERATION,f,&tag);
```

```
}
```
(pg.30) The iteration is defined as a sequence of iterItemact separated by terminal '|' and enclosed in twrminals '{' and '}'.

The `metabootParser::`iterItems method is defined as follows:

```
void
metabootParser::iterItems(cppcc::scr::tag::Long& tag)
/*
(iterItems::={altIterItem})
*/
{
  TagVector d;

  for(;;) {
    if(iseof()) break;

    if(kword() == KW_VERTICALBARTERMTOKEN) {
         d.push_back(0);
         altIterItem(d.back());
    }
    else break;

  }

  crelasdyn(KW_ITERITEMS,d,&tag);

}
```
(pg.31) The iterItems are used inside iteration.

The `metabootParser::`altIterItem method is defined as follows:

```
void
metabootParser::altIterItem(cppcc::scr::tag::Long& tag)
/*
(altIterItem::='|' iterItemact)
*/
{
  TagVector f(2);

  skipToken(KW_VERTICALBARTERMTOKEN,f[0]);

  iterItemact(f[1]);
  crefixe(KW_ALTITERITEM,f,&tag);


}
```
(pg.32) The altIterItem is individual item to be used inside iteration.

The `metabootParser::`iterItemAct method is defined as follows:

```
void
metabootParser::iterItemact(cppcc::scr::tag::Long& tag)
/*
(iterItemact::=iterItem[actions])
```

```
*/
{
  TagVector f(2);

  iterItem(f[0]);

  f[1]=0;
    if(kword() == KW_EQUALTERMTOKEN) {
actions(f[1]);
  }
  crefixe(KW_ITERITEMACT,f,&tag);


}
```
(pg.33) The iterItemAct is individual item with optional actions to be used inside iteration.

The `metabootParser::`IterItem method is defined as follows:

```
void
metabootParser::iterItem(cppcc::scr::tag::Long& tag)
/*
(iterItem::=nterm|maybeNterm)
*/
{

    if(kword() == KW_IDENTIFIER) {
    nterm(tag);
  }
  else if((kword() == KW_LESSTHENTERMTOKEN)
  ){
    maybeNterm(tag);
  }
  else {
    pdbkwmis(KW_IDENTIFIER);
    pdbkwmis(KW_LESSTHENTERMTOKEN);
    edber();
  }

}
```
(pg.34) The iterItem is individual item to be used inside iteration. It could be non-terminal represented by nterm or non-terminal enclosed in '<' and '>' represented by maybeNterm.

The `metabootParser::`maybeNterm method is defined as follows:

```
void
metabootParser::maybeNterm(cppcc::scr::tag::Long& tag)
/*
(maybeNterm::='<' nterm'>' )
*/
{
  TagVector f(3);

  skipToken(KW_LESSTHENTERMTOKEN,f[0]);

  nterm(f[1]);
```

```
  skipToken(KW_GREATERTHENTERMTOKEN,f[2]);
  crefixe(KW_MAYBENTERM,f,&tag);

}
```

(pg.35) The maybeNterm is a special iteration item meaning that specified non-terminal may be inside iteration only zero or one times. Multiple occurrences (2 or more times) of a specified non-terminal enclosed in '<' and '>' are forbidden within corresponding iteration.

Parsing Model Summary

Let's summarize Parsing Model main ideas.

A short description of Parsing Model is given by Patent claim 2. (see "Patent 8,464,232 claims").

C++ CCS and C CCS both have two implementations of Parsing Model: Syntax-Controlled Runtime and Binary. The Syntax-Controlled Runtime is used by parser performing parsing operations using corresponding API. If parser is successful analyzing source program, the Syntax-Controlled Runtime can be (formally) converted to Syntax-Controlled Binary. Any additional semantics processing can be done on Syntax-Controlled Binary using corresponding API. Having target language description on SGDL, the corresponding parser and additional code are generated, when executable program for target compiler is built, that compiler program is capable of compiling programs on specified language into Syntax-Controlled Runtime with sunsequent transformation into Syntax-Controlled Binary; also that compiler can de-compile provided Syntax-Controlled Binary file into original source code in accordance with language grammar. Ones again, all described operations are completely automated.

Chapter "Patent 8,464,232 claims" contains Patent claims. You may consider Patent claims as Parsing Model detailed formal specification.

Patent 8,464,232 claims

1. A compiler compiler system comprising a non-transitory computer readable medium including

a compiler compiler executable program comprising a source code for a main routine that accepts a common set of program arguments specifying compile/de-compile modes interacting with the compiler compiler runtime, the compiler compiler binary, and the compiler compiler generator by means of the compiler compiler management,

a compiler compiler management comprising a management environment composed of compiler compiler management classes and their relationships; the compiler compiler management classes including:

a logger class,

a run interface class,

a shell class,

a shell command enumerated type,

a key words container class,

a name vector container inner class of the key words container class,

a parser class,

a compiler class,

an action inner class of the compiler class,

a compile action inner class of the compiler class,

a de-compile action inner class of the compiler class,

a parser interface class,

a typedef tag,

a tokenizer class,

a line reader class,

a file line reader class,

a string line reader class,

a language class,

a token name container class,

a token name map container inner class of the token name container class,

a language interface class,

a language tokenizer set enumerated type,

a tokens class,

a name to index map container inner class of the tokens class,

a token vector container inner class of the tokens class,

a token interface class,

a token class; and

their relationships including:

the shell class, the compiler class, and the action inner class of the compiler class being derived from the run interface class,

the shell class allocating the logger class instance, having a data member of the shell command enumerated type, and allocating a vector of shared pointers to the compiler class instances,

the compiler class having a member reference to the shell class, allocating the parser class instance, and allocating a shared pointer to the action inner class of the compiler class,

the compile action inner class of the compiler class and the de-compile action inner class of the compiler class being derived from the action inner class of the compiler class,

the name vector container inner class of the key words container class being defined as a vector of strings,

the key words container class allocating the name vector container inner class of the key words container class for grammar terminals and non-terminals ,

the parser class being derived from the parser interface class,

the parser class having member references to the logger class and to the key words container class,

the parser class having data members of the tag type representing current context and grammar axiom,

the parser class having Boolean flags as data members,

the parser class allocating instances of the syntax controlled runtime class, the syntax controlled binary class, the tokenizer class, and a generator class, the generator class being a member of the compiler compiler generator,

the tokenizer class having member references to the logger class, the key words container class, and the parser class,

the tokenizer class allocating the token name container class instance, the line reader class instance, and the language class instance,

the line reader class being a base class for derived classes including a file line reader class and a string line reader class,

the language class being the base class for any target language specific derived classes,

the token name container class having an instance of the token name map container inner class of the token name container class,

the language class being derived from the language interface class,

the language class having a member reference to the tokenizer class,

the language class having a data member of the language tokenizer set enumerated type,

the language class allocating the tokens class instances for grammar tokens, key words, and non-terminals,

the tokens class allocating an instance of the name to index map container inner class of the tokens class and the token vector container inner class of the tokens class,

the name to index map container inner class of the tokens class being implemented as a map from string representing token name to integer representing token index in the token vector container inner class of the tokens class,

the token vector container inner class of the tokens class being implemented as a vector of shared pointers to the token class instances,

the token class being derived from the token interface class,

the token class having a data member pointer to the tokenizer class instance,

the token class having a data member of integer representing token id,

the token class having a data member of string representing token name and token value,

the token class being the base class to language specific tokens,

a compiler compiler runtime,

a compiler compiler binary,

a compiler compiler generator comprising a generator environment composed of compiler compiler generator classes and their relationships; the compiler compiler generator classes including:

a generator interface class,

a generator class,

a binary generator interface class,

a binary generator class,

a binary generator style interface class,

a binary generator style inner class of the binary generator class,

a runtime generator interface class,

a runtime generator class,

a runtime generator style interface class,

a runtime generator style inner class of the runtime generator class, and

and

their relationships including:

the generator class being derived from the generator interface class, the generator class implementing the generator interface class methods, the generator class having a member reference to a parser class comprised in the compiler compiler management, the generator class having a data member of language tokenizer set enumerated type which is a member of the compiler compiler management, the generator class owning an instance of a shared pointer to the runtime generator class and an instance of a shared pointer to the binary generator class,

the binary generator class being derived from the binary generator interface class, the binary generator class having a member reference to the generator class, the binary generator class owning an instance of a shared pointer to the binary generator style inner class,

the binary generator style inner class of the binary generator class being derived from the binary generator style interface class, the binary generator style inner class of the binary generator class being a parent class for any binary generator target language specific style inner class of the binary generator class,

any binary generator target language specific style inner class of the binary generator class implementing the binary generator style interface class methods,

the runtime generator class being derived from the runtime generator interface class, the runtime generator class having a member reference to the generator class, the runtime generator class owning an instance of a shared pointer to the runtime generator style inner class,

the runtime generator style inner class of the runtime generator class being derived from the runtime generator style interface class, the runtime generator style inner class of the runtime generator class being a parent class for any runtime generator target language specific style inner class of the runtime generator class, and

any runtime generator target language specific style inner class of the runtime generator class implementing the runtime generator style interface class methods,

a compiler compiler source grammar definition language,

a compiler compiler parsing model,

wherein the compiler compiler executable program performs an operation selected from the group consisting of:

calling a compiler compiler parser that performs a compilation from source text according to the compiler compiler source grammar definition language to the compiler compiler runtime;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from the compiler compiler runtime into source text according to the compiler compiler source grammar definition language;

calling the compiler compiler generator, the compiler compiler generator performing a formal conversion of the compiler compiler runtime into the compiler compiler binary;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from compiler compiler binary into source text according to the compiler compiler source grammar definition language;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from the compiler compiler binary into the compiler compiler runtime;

calling the compiler compiler generator, the compiler compiler generator performing target compiler code generation, and

wherein the compiler compiler executable program performs the compilation and de-compilation phases being performed automatically without any additional code implementation processing target language grammar defined in the compiler compiler grammar source definition language source text.

2. The compiler compiler system of claim 1,

the compiler compiler parsing model comprising a parsing logical view composed of compiler compiler parsing model classes and their relationships;

the compiler compiler parsing model classes including:

a context class,

a name class,

a symbol class, and

a rule class; and

their relationships including:

a collection of instances of the context class representing parsing results,

a collection of instances of the name class defined in each instance of the context class, the name instances ordered by a string name corresponding to each name instance,

a collection of instances of the name class defined in each of said context instances, said name instances ordered by a sequential index corresponding to each name instance,

a collection of instances of the symbol class defined in each of said context instances, said symbol instances ordered by a symbol ID corresponding each symbol instance, and

a collection of instances of the rule class defined in each of said symbol instances, said rule instances ordered by a rule invocation ID corresponding to each rule instance.

3. The compiler compiler system of claim 1, the compiler compiler runtime comprising a runtime environment composed of compiler compiler runtime classes and their relationships;

the compiler compiler runtime classes including:

a context class,

a name class,

a symbol class, and

a rule class; and

their relationships including:

a collection of instances of the context class representing parsing results,

a collection of instances of the name class defined in each instance of the context class, the name instances ordered by a string name corresponding to each name instance,

a collection of instances of the name class defined in each of said context instances, said name instances ordered by a sequential index corresponding to each name instance,

a collection of instances of the symbol class defined in each of said context instances, said symbol instances ordered by a symbol ID corresponding each symbol instance, and

a collection of instances of the rule class defined in each of said symbol instances, said rule instances ordered by a rule invocation ID corresponding to each rule instance.

4. The compiler compiler system of claim 1, the compiler compiler binary comprising a binary environment in form of compiler compiler binary classes and their relationships

the compiler compiler binary classes including

a context class,

a name class,

a symbol class, and

a rule class; and

their relationships including:

a collection of instances of the context class representing parsing results,

a collection of instances of the name class defined in each instance of the context class, the name instances ordered by a string name corresponding to each name instance,

a collection of instances of the name class defined in each of said context instances, said name instances ordered by a sequential index corresponding to each name instance,

a collection of instances of the symbol class defined in each of said context instances, said symbol instances ordered by a symbol ID corresponding each symbol instance, and

a collection of instances of the rule class defined in each of said symbol instances, said rule instances ordered by a rule invocation ID corresponding to each rule instance.

5. The compiler compiler system of claim 1, the compilation and de-compilation phases being performed automatically and the system being configured to implement any kind of semantics processing as subsequent phases based on compiler compiler binary.

6. The compiler compiler system of claim 1, designed to perform a whole compilation process under supervision of the compiler compiler management with parsing results to be represented in two interchangeable forms, namely, compiler compiler runtime and binary, both designed in accordance with the compiler compiler parsing model, the compiler compiler parsing model being common to the compiler compiler runtime and binary.

7. The compiler compiler system of claim 6, wherein the compiler compiler runtime is designed to serve parsing processing during source code compilation, and the compiler compiler binary is designed to serve as an optimized, interchangeable, and independent multiplatform data exchange protocol for any subsequent processing.

8. The compiler compiler system of claim 3,

the compiler compiler runtime defined as a syntax controlled runtime class designed to provide a compiler compiler runtime view of the compiler compiler parsing model;

the syntax controlled runtime class defining a compiler compiler runtime syntax-controlled API composed of methods of syntax controlled runtime inner classes and their relationships;

the compiler compiler runtime syntax-controlled API capable of being invoked from any generated parser.

9. The compiler compiler system of claim 4,

the compiler compiler binary defined as a syntax controlled binary class designed to provide a compiler compiler binary view of the compiler compiler parsing model;

the syntax controlled binary class defining a compiler compiler binary syntax-controlled API composed of methods of syntax controlled binary inner classes and their relationships;

the context inner class of the syntax controlled binary class having only one data member representing memory allocated as a standard library vector container to be used as a single memory resource for allocating all syntax controlled binary inner class instances and their relationships; and

the compiler compiler binary syntax-controlled API capable of being invoked from any application performing semantics processing.

10. The compiler compiler system of claim 9, wherein the compiler compiler binary is a multiplatform data exchange protocol for any subsequent semantics processing.

11. The compiler compiler system of claim 1, wherein source codes of the compiler compiler runtime, binary, generator, and management are compiled into a compiler compiler foundation library that is used by compiler compiler executable program and any other target compiler built by means of compiler compiler system.

12. The compiler compiler system according to claim 1, wherein the compiler compiler generator creates generated code for a given compiler compiler source grammar definition language description and that generated code is compiled into generated library and target compiler executable program is compiled and built with compiler compiler foundation library and generated library; built this way target compiler executable program has compile and de-compile operations ready; all semantics processing can be done as independent subsequent operations implemented using compiler compiler binary syntax-controlled API.

13. The compiler compiler system according to claim 1, wherein the compiler compiler management, generator, runtime and binary serve as a multiplatform foundation based on compiler compiler runtime and binary transformations for researching and developing new types of products such as binary files processing, and communication between programs running on different platforms, programs running on the same platform, programs running on different operating systems, programs running on the same operating system, programs built using different programming languages, and programs built using the same programming language.

14. The compiler compiler system according to claim 1, wherein the compiler compiler management, generator, runtime and binary serve as a multiplatform foundation based on compiler compiler runtime and binary transformations for researching and developing new types of products including obfuscation, security protection, content management, and any other formal compiler compiler binary transformations implemented for particular cases.

15. The compiler compiler system according to claim 1, wherein the compiler compiler management, generator, runtime and binary serve as a multiplatform foundation for researching and developing new types of products based on binary files processing when existing binary file formats representing documents, audio, and video are transformed in form of compiler compiler source grammar definition language binary file format specification; followed by implementation of binary file convertor from its binary content into compiler compiler runtime having compiler compiler binary created by formal compiler compiler phase.

16. A compiler compiler method comprising:

providing a non-transitory computer readable medium including

a compiler compiler executable program comprising a source code for main routine that accepts a common set of program arguments specifying compile/de-compile modes interacting with the compiler compiler runtime, the compiler compiler binary, and the compiler compiler generator by means of the compiler compiler management, binary class, the tokenizer class, and a generator class, the generator class being a member of the compiler compiler generator,

the tokenizer class having member references to the logger class, the key words container class, the parser class,

the tokenizer class allocating instances of the token name container class, the line reader class, and the language class,

the line reader class being a base class for derived classes including a file line reader class and a string line reader class,

the language class being the base class for any target language specific derived classes,

the token name container class having an instance of the token name map container inner class of the token name container class,

the language class being derived from language interface class,

the language class having a member reference to the tokenizer class,

the language class having a data member of the language tokenizer set enumerated type,

the language class allocating instances of the tokens class for grammar tokens, key words, and non-terminals,

the tokens class allocating an instance of the name to index map container inner class of the tokens class and the token vector container inner class of the token class,

the name to index map container inner class of the tokens class being implemented as a map from a string representing a token name to an integer representing a token index in the token vector container inner class of the tokens class,

the token vector container inner class of the tokens class being implemented as a vector of shared pointers to the token class instances,

the token class being derived from the token interface class,

the token class having a data member pointer to the tokenizer class instance,

the token class having a data member of integer representing token id,

the token class having a data member of string representing token name and token value,

the token class being the base class to language specific tokens,

a compiler compiler runtime,

a compiler compiler binary,

a compiler compiler generator comprising a generator environment composed of compiler compiler generator classes and their relationships; the compiler compiler generator classes including:

a generator interface class,

a generator class,

a binary generator interface class,

a binary generator class,

a binary generator style interface class,

a binary generator style inner class of the binary generator class,

a runtime generator interface class,

a runtime generator class,

a runtime generator style interface class,

a runtime generator style inner class of the runtime generator class, and

and

their relationships including:

the generator class being derived from the generator interface class, the generator class implementing the generator interface class methods, the generator class having a member reference to a parser class comprised in the compiler compiler management, the generator class having a data member of language tokenizer set enumerated type which is a member of the compiler compiler management, the generator class owning an instance of a shared pointer to the runtime generator class and an instance of a shared pointer to the binary generator class,

the binary generator class being derived from the binary generator interface class, the binary generator class having a member reference to the generator class, the binary generator class owning an instance of a shared pointer to the binary generator style inner class,

the binary generator style inner class of the binary generator class being derived from the binary generator style interface class, the binary generator style inner class of the binary generator class being a parent class for any binary generator target language specific style inner class of the binary generator class,

any binary generator target language specific style inner class of the binary generator class implementing the binary generator style interface class methods,

the runtime generator class being derived from the runtime generator interface class, the runtime generator class having a member reference to the generator class, the runtime generator class owning an instance of a shared pointer to the runtime generator style inner class,

the runtime generator style inner class of the runtime generator class being derived from the runtime generator style interface class, the runtime generator style inner class of the runtime generator class being a parent class for any runtime generator target language specific style inner class of the runtime generator class, and

any runtime generator target language specific style inner class of the runtime generator class implementing the runtime generator style interface class methods,

a compiler compiler source grammar definition language,

a compiler compiler parsing model, and

executing the compiler compiler executable program performs an operation selected from the group consisting of:

calling a compiler compiler parser that performs a compilation from source text according to the compiler compiler source grammar definition language to the compiler compiler runtime;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from the compiler compiler runtime into source text according to the compiler compiler source grammar definition language;

calling the compiler compiler generator, the compiler compiler generator performing a formal conversion of the compiler compiler runtime into the compiler compiler binary;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from compiler compiler binary into source text according to the compiler compiler source grammar definition language;

calling the compiler compiler generator, the compiler compiler generator performing formal de-compilation from the compiler compiler binary into the compiler compiler runtime;

calling the compiler compiler generator, the compiler compiler generator performing target compiler code generation, and

wherein the compiler compiler executable program performs the compilation and de-compilation phases being performed automatically without any additional code implementation processing target language grammar defined in the compiler compiler grammar source definition language source text.

17. A method of securely transmitting data comprising

performing the compiler compiler method of claim 16 and further comprising:

providing a computer to read the non-transitory computer readable medium,

using the computer to represent a data structure in the compiler compiler source grammar definition language;

using the computer to execute the compiler compiler executable program to create a data compiler executable program linked from compiled compiler compiler executable program text, the compiler compiler foundation library, and a data compiler generated library created as a result of compilation of the data structure represented in the compiler compiler source grammar definition language by executing the compiler compiler executable program;

using the computer to execute the data compiler executable program to convert the data into a data compiler compiler runtime;

using the computer to execute a transformation algorithm to convert the data compiler compiler runtime into a transformed data compiler compiler runtime based on the compiler compiler runtime syntax-controlled API;

using the computer to execute the data compiler executable program to convert the transformed data compiler compiler runtime into a transformed data compiler compiler binary;

using the computer to execute a transformation algorithm to convert the transformed data compiler compiler binary into secondary transformed data compiler compiler binary based on the compiler compiler binary syntax-controlled API;

using the computer to transmit the secondary transformed data compiler compiler binary to a receiving computer;

using the receiving computer to execute a backward transformation algorithm to convert the secondary transformed data compiler compiler binary back to the transformed data compiler compiler binary;

using the receiving computer to execute the compiler compiler executable program to convert the transformed data compiler compiler binary into the transformed data compiler compiler runtime;

using the receiving computer to execute a backward transformation algorithm to convert the transformed data compiler compiler runtime back to the data compiler compiler runtime;

using the receiving computer to execute the compiler compiler executable program to convert the data compiler compiler runtime into the data compiler compiler binary; and

using the receiving computer to perform subsequent semantics processing of the data compiler compiler binary based on compiler compiler binary syntax-controlled API.

Source Grammar Definition Language

SGDL on SGDL

The source grammar definition language was introduced in Chapter " US Patent 8,464,232 summary " with more clarification given in Chapter "Parsing Model". Ones again, below is a content of meta.sgr file, that is source grammar definition language program representing source grammar definition language, or, if SGDL is abreviation for Source Grammar Definition Language, then SGDL on SGDL is here:

```
(meta
  (grammar ::=
       0 ="  METAACTBEG();"=

       '(' grammarNameDef
                    { rule }
       ')'

       0 ="  METAACTEND();"=
  )

  (grammarNameDef ::= identifier
  )

  (rule ::=   '(' nterm '::=' right ')'
  )

  (nterm       ::= identifier
  )

  (right ::= { element }
  )

  (element    ::=
              identAlt | alternative | identMiss | iteration | action
  )

  (action     ::= integerToken '=' { stringToken } '='
  )

  (actions    ::= '=' { action } '='
  )

  (identAlt   ::=
              ntermtermact { Altpart }
  )

  (Altpart    ::= '|' ntermtermact
  )

  (ntermtermact ::= ntermterm [ actions ]
  )

  (ntermterm  ::=
              nterm | termToken
  )
```

```
  (alternative       ::=    '(' identAlt ')'
  )

  (identMiss  ::=    '[' identAlt ']'
  )

  (iteration  ::=    '{' iterItemact iterItems '}'
  )

  (iterItems  ::=    { altIterItem }
  )

  (altIterItem       ::= '|' iterItemact
  )

  (iterItemact       ::= iterItem  [ actions ]
  )

  (iterItem   ::=
               nterm | maybeNterm
  )

  (maybeNterm ::=
              '<' nterm '>'
  )

)
```

XML specification based on SGDL

In Chapter "Use Case: XML compiler" we will provide more details for XML compiler that is built completely in automated mode by means of C++ CCS having this SGDL program describying any XML [4] file:

```
(xsd2meta

  (grammar ::=
  0 ="  METAACTBEG();"=

  xmlVersion

  xmlTag

  0 ="  METAACTEND();"=
  )

  (xmlVersion ::=
    '<'
    '?'
    identifier
    xmlVersionAttributes
    '>'
  )

  (xmlVersionAttributes ::=
    attributes '?'
  )

  (attribute ::=
    identifier [ attributeExtention ] '=' stringToken
  )

  (attributeExtention ::=
    ':' identifier
  )

  (attributes ::=
    { attribute }
  )

  (xmlTag ::=
    '<' xmlTagID attributes xmlTagEndOrTags
  )

  (xmlTagEndOrTags ::= xmlTagEnd | xmlTags
  )

  (xmlTagEnd ::=
    '/'
    0 ="  xmlTokenOn();"=
    '>'
  )

  (xmlTagID ::=
    identifier [ xmlTagIDExtention ]
  )
```

```
  (xmlTagIDExtention ::=
    ':' identifier
  )

  (xmlTags ::=
    0 ="  xmlTokenOn();"=
    '>'
    xmlTagsList
    xmlTagID
    '>'
  )

  (xmlTagsList ::=
    { xmlTag }
    '</'
  )
)
```

JSON specification based on SGDL

In Chapter "Use Case: JSON compiler" we will provide more details for JSON compiler that is built completely in automated mode by means of C++ CCS having this SGDL program describying any JSON [5] file:

```
(json2meta
  (data ::=
         0 ="  METAACTBEG();"=

    object

         0 ="  METAACTEND();"=
  )

  (object ::=
        '{'
              members
        '}'
  )

  (members ::= pair { comma_pair }
  )

  (comma_pair ::= ',' pair
  )

  (pair ::= stringToken ':' value
  )

  (value ::= stringToken | integerToken | object | array | terminalValues
  )

  (terminalValues ::= 'true' | 'false' | 'null'
  )

  (array ::=
     '['
        arrayElements
     ']'
  )

  (arrayElements ::= value { comma_value }
  )

  (comma_value ::= ',' value
  )
)
```

C++ CCS overview

C++ CCS list of files

Below is a list of files in~/v2/cppcc directory.

```
~/v2/cppcc
$ wc include/* src/*.cc incloc/* srcloc/*.cc bin/make* lib/make*
  147   218  3013 include/CommonRoutines.h
  129   208  2659 include/Compiler.h
  332   617  8432 include/Generator.h
  132   265  3391 include/Logger.h
  746  1505 16756 include/Parser.h
   59    84  1028 include/Shell.h
  587  1466 11874 include/SyntaxControlledBinary.h
  764  1657 17461 include/SyntaxControlledRuntime.h
   34    53   613 include/SystemRoutines.h
   34    46   572 include/TokenNames.h
 1280  2501 24655 include/Tokenizer.h
   77   153  2143 include/cppccstd.h
  229   582  6213 src/CommonRoutines.cc
  136   220  2635 src/Compiler.cc
 1436  3195 35362 src/Generator.cc
  108   209  1993 src/Logger.cc
  563  1364 15719 src/Parser.cc
  222   516  4846 src/Shell.cc
  509  1322 10705 src/SyntaxControlledBinary.cc
  878  2111 22792 src/SyntaxControlledRuntime.cc
    9    16   264 src/SystemRoutines.cc
   79   143  1757 src/TokenNames.cc
 2585  5823 60965 src/Tokenizer.cc
   37    67   673 src/cppcc.cc

   65    80  1168 incloc/metaFirstGenerator.h
  132   257  2285 incloc/metaFirstKeyWordDefinition.h
   76   143  1949 incloc/metaFirstParser.h
 9289 24979 307182 srcloc/metaFirstGenerator.cc
  124   147  2079 srcloc/metaFirstKeyWordDefinition.cc
   36    42   762 srcloc/metaFirstMakeGenerators.cc
  495   732  8957 srcloc/metaFirstParser.cc
 1214  3277 37627 srcloc/tmp.cc

   33    73   695 bin/makefile
   92   173  2377 lib/makefile
22668 54244 621602 total
```

As you know wc output, the total number of lines of current C++ CCS is 22668 with 621602 bytes total.

The incloc and srcloc are directories for generated code by original SGDL specification. In the list above,

9289 24979 307182 srcloc/metaFirstGenerator.cc

was modified by real code that provides actual generation of the output code based on original SGDL on SGDL.

lib/makefile

The lib/makefile has the following content:

```
#      $CPPCCHOME/lib/makefile

cfl    = -g
CCFLAGS  = $(cfl) -I../include -I../incloc

DIR_SRC= ../src
DIR_SRCLOC= ../srcloc
LIBRUNTIME= libruntime.a
LIBGENERATE= libgenerate.a
LIBS= $(LIBRUNTIME) $(LIBGENERATE)

# LIBRUNTIME
LO_sys_routines= $(LIBRUNTIME)($(DIR_SRC)/SystemRoutines.o)
LO_com_routines= $(LIBRUNTIME)($(DIR_SRC)/CommonRoutines.o)
LO_lex_tokenizer= $(LIBRUNTIME)($(DIR_SRC)/Tokenizer.o)
LO_syn_parser= $(LIBRUNTIME)($(DIR_SRC)/Parser.o)
LO_cmd_shell= $(LIBRUNTIME)($(DIR_SRC)/Shell.o)
LO_cmp_compiler= $(LIBRUNTIME)($(DIR_SRC)/Compiler.o)
LO_log_logger= $(LIBRUNTIME)($(DIR_SRC)/Logger.o)
LO_scr_runtime= $(LIBRUNTIME)($(DIR_SRC)/SyntaxControlledRuntime.o)
LO_scb_binary= $(LIBRUNTIME)($(DIR_SRC)/SyntaxControlledBinary.o)
LO_tnm_tokennames= $(LIBRUNTIME)($(DIR_SRC)/TokenNames.o)
LO_gen_generator= $(LIBRUNTIME)($(DIR_SRC)/Generator.o)

# LIBGENERATE
LO_parser= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstParser.o)
LO_generator= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstGenerator.o)
LO_keywords= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstKeyWordDefinition.o)
LO_makeGenerators= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstMakeGenerators.o)

INCL   = \
              ../incloc/metaFirstKeyWordDefinition.h \
              ../include/CommonRoutines.h \
              ../include/SystemRoutines.h \
              ../include/cppccstd.h


.cc.a  :
       cd $(<D); $(CXX) -c $(CCFLAGS) $(<F)
       $(AR) $(ARFLAGS) $@ $*.o
       rm -f $*.o
#      @echo "<$(*D)*$(*F)=$*>,<$(@D)*$(@F)=$@>"
#      @echo "<$(<D)*$(<F)=$<>,<$(?D)*$(?F)=$?>"

all : $(LIBS)
```

```
	@echo  "/$(LIBS)/ done !"
	ranlib $(LIBRUNTIME)
	ranlib $(LIBGENERATE)


$(LIBRUNTIME) : \
	$(LO_sys_routines)   \
	$(LO_com_routines)   \
	$(LO_lex_tokenizer)  \
	$(LO_cmd_shell)      \
	$(LO_cmp_compiler)   \
	$(LO_log_logger)     \
	$(LO_scr_runtime) \
	$(LO_scb_binary) \
	$(LO_tnm_tokennames) \
	$(LO_gen_generator) \
	$(LO_syn_parser)
		@echo "$@ changed"

#	$(LO_makeGenerators)

$(LIBGENERATE) : \
	$(LO_parser)  \
	$(LO_generator) \
	$(LO_makeGenerators) \
	$(LO_keywords)
		@echo "$@ changed"

# $(LO_makeGenerators)

$(LO_sys_routines)   \
$(LO_com_routines)   \
$(LO_lex_tokenizer)  \
$(LO_syn_parser)     \
$(LO_cmd_shell)      \
$(LO_cmp_compiler)   \
$(LO_log_logger)     \
$(LO_scr_runtime) \
$(LO_scb_binary) \
$(LO_parser)  \
$(LO_tnm_tokennames) \
$(LO_keywords) \
$(LO_gen_generator) \
$(LO_makeGenerators) \
$(LO_generator)      :      \
		$(INCL)
```

As you can see two libraries are created when this makefile is executed:

```
LIBRUNTIME= libruntime.a
LIBGENERATE= libgenerate.a
```

The `libruntime.a` library is actually C++ CCS foundation library that has generic/common code for all compilers that may be created by means of C++ CCS. The following files are included into `libruntime.a` library:

```
# LIBRUNTIME
LO_sys_routines= $(LIBRUNTIME)($(DIR_SRC)/SystemRoutines.o)
LO_com_routines= $(LIBRUNTIME)($(DIR_SRC)/CommonRoutines.o)
LO_lex_tokenizer= $(LIBRUNTIME)($(DIR_SRC)/Tokenizer.o)
LO_syn_parser= $(LIBRUNTIME)($(DIR_SRC)/Parser.o)
LO_cmd_shell= $(LIBRUNTIME)($(DIR_SRC)/Shell.o)
LO_cmp_compiler= $(LIBRUNTIME)($(DIR_SRC)/Compiler.o)
LO_log_logger= $(LIBRUNTIME)($(DIR_SRC)/Logger.o)
LO_scr_runtime= $(LIBRUNTIME)($(DIR_SRC)/SyntaxControlledRuntime.o)
LO_scb_binary= $(LIBRUNTIME)($(DIR_SRC)/SyntaxControlledBinary.o)
LO_tnm_tokennames= $(LIBRUNTIME)($(DIR_SRC)/TokenNames.o)
LO_gen_generator= $(LIBRUNTIME)($(DIR_SRC)/Generator.o)
```

The `libgenerate.a` library is realated to generated code for a given SGDL specification. It contains the following files:

```
# LIBGENERATE
LO_parser= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstParser.o)
LO_generator= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstGenerator.o)
LO_keywords= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstKeyWordDefinition.o)
LO_makeGenerators= $(LIBGENERATE)($(DIR_SRCLOC)/metaFirstMakeGenerators.o)
```

bin/makefile

The bin/makefile has the following content:

```
# $CPPCCHOME/bin/makefile

cfl    = -g
CCFLAGS       = $(cfl) -I../include -I../incloc

#LIB   = ../lib/libruntime.a ../lib/libgenerate.a ../lib/libruntime.a
##LIB  = ../lib/libgenerate.a ../lib/libruntime.a ../lib/libgenerate.a
LIB    = ../lib/libruntime.a ../lib/libgenerate.a ../lib/libruntime.a
../lib/libgenerate.a

PROG_LIST = cppcc

PROG_OBJ = ../src/cppcc.o

SRC    = ../src
INCL   = ../include/Shell.h ../include/cppccstd.h

.cc.o:
       cd $(<D); $(CXX) -c $(CCFLAGS) $(<F)

all : $(PROG_LIST)
       @echo "all /$(PROG_LIST)/ done !"
       rm $(PROG_OBJ)

$(PROG_LIST) : $(PROG_OBJ) $(LIB)
       $(CXX) $(CCFLAGS) $(PROG_OBJ) -o $@ $(LIB) -lm
       @echo  "$@ created"

$(PROG_OBJ) : $(INCL)
```

That make file creats cppcc program in bin directory based on `../src/cppcc.cc source file and ../lib/libruntime.a ../lib/libgenerate.a` libraries.

Running C++ CCS cppcc program

To run C++ CCS cppcc program you can, e.g.:

$CPPCCHOME/bin/cppcc meta.sgr

Where envirinment variable CPPCCHOME is set to top level directrory where the whole cppcc is, i.e.,

$CPPCCHOME/bin has a cppcc executable program. Note that meta.sgr file is SGDL on SGDL (see Chapter "Source Grammar Definition Language" and Chapter "US Patent 8,464,232 summary ").

When you run that $CPPCCHOME/bin/cppcc meta.sgr, the following files will be created:

- meta.sgr.fgr
- metaKeyWordDefinition.h
- metaGenerator.h
- metaParser.h
- metaMakeGenerators.cc
- metaKeyWordDefinition.cc
- metaGenerator.cc
- metaParser.cc
- meta.sgr.urt
- meta.sgr.log

The meta.sgr.fgr is a file that contains Syntax-Controlled Binary for original meta.sgr. The *.h and *.cc files are generated files corresponding to original meta.sgr. They contain enum KeyWords, KeyWordsContainer, metaParser and metaGenerator files. The meta.sgr.log is a log file. The meta.sgr.urt is a result of a de-compilation of Syntax-Controlled Runtime into source text according with target language grammar, i.e., it is equivalent to meta.sgr with different formatting.

Self testing running C++ CCS cppcc program

Consider the following sequence:

```
(s.1)
$CPPCCHOME/bin/cppcc meta.sgr

(s.2)
cp meta.sgr.urt meta22.sgr

(s.3)
$CPPCCHOME/bin/cppcc meta22.sgr

(s.4)
diff meta22.sgr.urt meta.sgr.urt
```

The last diff should show no differences. Notes, that (s.1) creates `meta.sgr.urt` file as a result of de-compilation of Syntax-Controlled Runtime into that file. The (s.2) makes meta22.sgr file preserving default extention (.sgr). The (s.3) step compiles meta22.sgr creating `meta22.sgr.urt` file. And (s.4) should show no differences.

Consider the following sequence:

```
(b.1)
$CPPCCHOME/bin/cppcc -u  meta22.sgr.fgr
cat  meta22.sgr.fgr.ubr

(b.2)
cp meta22.sgr.fgr.ubr meta22ubr.sgr

(b.3)
$CPPCCHOME/bin/cppcc meta22ubr.sgr

(b.4)
$CPPCCHOME/bin/cppcc -u  meta22ubr.sgr.fgr

(b.5)
diff meta22ubr.sgr.fgr.ubr  meta22.sgr.fgr.ubr

(b.6)
diff meta22ubr.sgr.fgr.ubr  meta22ubr.sgr.urt
```

Note, that (b.1) is a de-compilation request from Syntax-Controlled Binary presented in file `meta22.sgr.fgr`, into `meta22.sgr.fgr.ubr` file, that is a program text in accordance with target language grammar. The (b.2) makes another copy, i.e., `meta22ubr.sgr` file. The (b.3) compiles `meta22ubr.sgr` again. The (b.4) does de-compilation again similar to (b.1) having different input/outputs. The (b.5) compares de-compilation results from different files with Syntax-Controlled Binary. The (b.6) compares de-compilation results from Syntax-Controlled Binary and Syntax-Controlled Runtime. Both (b.5) and (b.6) should generate no diffs.

As a final note, similar tests (s.1) - (s.4) and (b.1) - (b.6) can be done for any target compiler having SGDL specification. Please review FIG.2 - FIG.4 again.

C++ CCS Management

The C++ CCS management is defined in these files:

- include/cppccstd.h
- include/SystemRoutines.h and src/SystemRoutines.cc
- include/CommonRoutines.h and src/CommonRoutines.cc
- include/TokenNames.h and src/TokenNames.cc
- include/Logger.h and src/Logger.cc
- include/Compiler.h and src/Compiler.cc
- include/Parser.h and src/Parser.cc
- include/Tokenizer.h and src/Tokenizer.cc
- include/Shell.h and src/Shell.cc
- src/cppcc.cc

These files:

- include/cppccstd.h
- include/SystemRoutines.h and src/SystemRoutines.cc
- include/CommonRoutines.h and src/CommonRoutines.cc
- include/TokenNames.h and src/TokenNames.cc
- include/Logger.h and src/Logger.cc

provide auxiliry features used by CCS. In this Chapter we provide more details for these files:

- include/Compiler.h and src/Compiler.cc
- include/Parser.h and src/Parser.cc
- include/Tokenizer.h and src/Tokenizer.cc
- include/Shell.h and src/Shell.cc
- src/cppcc.cc

FIG.5 contains UML diagram defining relationships between Shell, Compiler, Parser, Logger, KeyWordsContainer, and Compiler::Action classes. Note, that src/cppcc.cc has this content:

```
#include "Shell.h"
int    main(int argc, char** argv)
{
  cppcc::cmd::Shell sh(argc, argv);
  sh.run();
  return 0;
}
```

So, Shell class is responsible for program arguments to be processed. As described in section "Running C++ CCS cppcc program" of Chapter "C++ CCS overview" these commands:

```
$CPPCCHOME/bin/cppcc meta.sgr
$CPPCCHOME/bin/cppcc -u  meta.sgr.fgr
```

are actions to compile meta.sgr and to de-compile Syntax-Controlled Binary from file `meta.sgr.fgr` into original SGDL specification.

FIG.5 shows that Shell owns Logger instance, logger_, and SHELL_COMMAND value, command_. Also Shell owns a collection of Compiler instances, compilers_. In general, under one Shell it is possible to keep many instances of Compiler with their own languages. The Shell and Compiler classes are derived from noncopyable class to make them noncopyable C++03 way. Also Shell and Compiler classes are derived from IRun interface having one pure virtual method:

```
virtual void run() = 0;
```

Both Shell and Compiler classes implement that `run()` method.

From FIG. 5 the Compiler class owns KeyWordsContainer instance, keyWords_, Parser instance, parser_, and Compiler::Action instance, action_. Also Compiler has a reference to the Shell instance.

This KeyWordsContainer defines all grammar symbols grouped by different categories. FIG. 6 shows the corresponding UML diagram related to KeyWordsContainer class.
The Compiler::Action is also derived from IRun and Compiler::ActionCompile, Compiler::ActionDeCompile implement run() method for compile and de-compile modes.

FIG.7 contains UML diagram defining Parser class. On that FIG.7 and other UML diagrams and the Patent [3] Tag is a typedef that corresponds to C++ CCS source code as a TagLong defined as follows in SyntaxControlledBinary.h:

```
namespace cppcc {
namespace scb {
////////////////////////////////////////////////////////////////////////
//     scb - Syntax Controlled Binary
////////////////////////////////////////////////////////////////////////

struct SyntaxControlledBinary
{
       struct kwnirType;

       typedef cppcc::scr::tag::Long         TagLong;
       typedef TagLong*                      TagPointer;
       typedef cppcc::scr::tag::TypeInstance TagInstance;
       typedef std::vector<kwnirType*>       kwnirTypeVector;
       std::vector<TagLong>                  memory_;
. . .
```

Where `cppcc::scr::tag::Long` is denied in SyntaxControlledRuntime as follows:

```
namespace cppcc {
namespace scr {

////////////////////////////////////////////////////////////////////////
//     scr - Syntax Controlled Runtime
////////////////////////////////////////////////////////////////////////
namespace tag {

#ifdef _CPPCC_TAG_32BITS_

typedef unsigned long      ULong;
typedef long               Long;
typedef float              Real;

struct TypeInstance {
```

```
  ULong             t:10;
  ULong             d:22;
};

#else

typedef unsigned long long  ULong;
typedef long long           Long;
typedef double              Real;

struct TypeInstance {
  ULong             t:16;
  ULong             d:48;
};

#endif
...
...
```

Note, that `TypeInstance is` used interchangeably with `Long` by using these functions:

```
inline void setedflong(tag::Long* ll, tag::Long tt, tag::Long dd)
{
  if (ll) {
    ((TypeInstance*)ll)->t = tt;
    ((TypeInstance*)ll)->d = dd;
  }
}

inline void setlongedf(tag::Long* ll, tag::Long& tt, tag::Long& dd)
{
  if (ll) {
    tt = ((TypeInstance*)ll)->t;
    dd = ((TypeInstance*)ll)->d;
  }
}
```

On FIG. 7 Parser owns two instances of Tag, context_ and axiom_. In Parser.h they are defined as follows:

```
class Parser : IParser
{
. . .
  cppcc::log::Logger&                   logger_;
  cppcc::com::KeyWordsContainer&        grammarSymbols_;
  cppcc::scr::SyntaxControlledRuntime   runtime_;
  cppcc::scb::SyntaxControlledBinary    binary_;
  cppcc::lex::Tokenizer                 tokenizer_;
  cppcc::scr::tag::Long                 context_;
  cppcc::scr::tag::Long                 axiom_;
  cppcc::gen::Generator                 generator_;
  std::string                           filename_;
  bool                                  debug_;
  bool                                  isXmlTokenFlag_;

public:
```

```
  typedef std::vector<cppcc::scr::tag::Long>     TagVector;
```

. . .

On FIG. 7 Parser is derived from IParser with pure virtual methods, compile, dedicated for different compilation modes, compiling sources from files, strings, etc..., managing input sources and compilation listing . On FIG. 7 Parser shares Logger and KeyWordsContainer with the Shell. On FIG. 7 Parser owns filename_ as std::string and bool variables, debug_ and isXmlTokenFlag_ to control debug logging and to work with XML style. On FIG. 7 Parser owns runtime_ as SyntaxControlledRuntime instance and binary_ as SyntaxControlledBinary instance. On FIG. 7 Parser owns tokenizer_ as Tokenizer instance and generator_ as Generator instance.

FIG. 8 - FIG.18 are dedicated to Syntax-Controlled Runtime and will be discussed in Chapter "C++ CCS Syntax-Controlled Runtime API".

FIG. 19 - FIG.26 are dedicated to Syntax-Controlled Binary and will be discussed in Chapter "C++ CCS Syntax-Controlled Binary API".

FIG. 27 contains UML diagram defining Tokenizer class. It shares Logger and KeyWordsContainer instances. Also Tokenizer class serves a given Parser instance, and the Parser instance owns the Tokenizer instance. The Tokenizer owns instance of TokenNameContainer that keeps default token container with their names and values. The Tokenizer owns instance of LineReader and instance of Language. On FIG. 27 there are FileLineReader and StringLineReader classes derived from LineReader class. On FIG. 27 there are LanguageMeta and LanguageXML classes derived from Language class.

FIG. 28 contains UML diagram defining TokenNameContainer as a class having tokenNames_ member defined as std::map from std::string to std::string.

FIG. 29 contains UML diagram defining Language class derived from ILanguage with pure virtual methods such as:

```
  virtual void getchr() = 0;
  virtual void getid() = 0;
  virtual void getint() = 0;
  virtual void getstring() = 0;
  virtual void getNextToken() = 0;
  virtual void flush() = 0;
```

On FIG. 29 class Language serves a dedicated Tokenizer instance; the Tokenizer instance owns corresponding Language instance. On FIG. 29 the `LanguageTokenizerSet` is enum type with list of support language categories such as Meta, XML, etc... On FIG. 29 the Language class owns tokens_, keyWords_, and nonTerminal_ instances of Tokens class.

FIG. 30 contains UML diagram defining Tokens class. The Tokens class owns name2index_ member that is Tokens::Name2IndexMap instance and tokens_ instance that is std::vector of Token instances. The Tokens::Name2IndexMap is a typedef to std::map that is a map from std::string to int. The Token class is derived from IToken with these pure virtual functions:

```
  virtual bool isToken() = 0;
  virtual void flushToken() = 0;
  virtual void skipToken(const int t, cppcc::scr::tag::Long& l) = 0;
```

The Token class has a reference to Tokenizer it serves. The Token class owns token_ and name_ std::string members along with symbol_ as int.

FIG. 31 contains UML diagram defining Token derived classes such as OneCharToken, TwoCharToken, ThreeCharToken. Also OneCharToken is a parent to OneCharXMLGreaterThanToken.

Note, that components presented on FIG. 5 - FIG. 35 work the same way for compiler compiler (cppcc program) and for any other target language compiler. FIG. 2 shows compiler compiler workflow when it builds itself operating SGDL on SGDL (presented in meta.sgr), FIG.3 shows compiler compiler workflow when it builds any other compiler having SGDL specification (some <filename>.sgr), and FIG.4 shows target compiler workflow when it compiles programs on his own language.

C++ CCS Generator

The C++ CCS Generator is defined in these files:

- include/Generator.h and src/Generator.cc

FIG. 32 contains UML diagram defining Generator class. The Generator class owns binary_ instance of GeneratorBinary, runtime_ of GeneratorRuntime, and tokenizerSet_ instance of enum LanguageTokenizerSet. The Generator class is derived from IGenerator. The GeneratorBinary is derived from IBinaryGenerator. The GeneratorRuntime is derived from IRuntimeGenerator. The Generator serves Parser for this reason the Generator keeps a reference to Parser.

FIG. 33 contains UML diagram defining GeneratorBinary class having reference generator_ to Generator instance and style_ member of GeneratorBinary::GeneratorBinaryStyle inner class derived from IBinaryGeneratirStyle. There are GeneratorBinary::GeneratorBinaryStyleMeta and GeneratorBinary::GeneratorBinaryXML classes derived from GeneratorBinary::GeneratorBinaryStyle.

FIG. 34 contains UML diagram defining GeneratorRuntime class having reference generator_ to Generator instance and style_ member of GeneratorRuntime::GeneratorRuntimeStyle inner class derived from IRuntimeGeneratirStyle. There are GeneratorRuntime::GeneratorRuntimeStyleMeta and GeneratorRuntime::GeneratorRuntimeXML classes derived from GeneratorRuntime::GeneratorRuntimeStyle.

FIG. 35 shows Generator class with additional classes such as metaFirstGeneratorBinary, metabootGeneratorBinary, xsdGeneratorBinary derived from GeneratorBinary and metaFirstGeneratorRuntime, metabootGeneratorRuntime, xsdGeneratorRuntime derived from GeneratorRuntime. Note, that for a give <filename>.sgr having <grammar name> as a top level section in that <filename>.sgr the corresponding files such as:

- <grammar name>Generator.h and <grammar name>Generator.cc
- <grammar name>WordDefinition.h and <grammar name>KeyWordDefinition.cc
- <grammar name>Parser.h and <grammar name>Parser.cc
- <grammar name>MakeGenerators.cc

are generated by compiler compiler, cppcc program.

GeneratorBinary decompile method implementation

GeneratorBinary class implements decompile method this way:

```
void
GeneratorBinary::decompile(const std::string& fname)
{
  std::string gName = grammarName();
  filename_ =
       (!fname.size() || ("" == fname))
     ? (gName + ".ubr")
     : fname
  ;

  output_.open(filename_.c_str());
  if (!output_) {
      std::string   syserr = cppcc::com::CPPCCException::systemError();
      CPPCC_THROW_EXCEPTION(
      << "Can't open file for binary decompiling:'"
      << filename_
```

```
      << "' - Reason:'"
      << syserr
      << "'"
      )
  }

  // Dump the content first.
  //int isDump = 0;
  //if (isDump) {
  //  generator_.parser_.runtime_.dump(output_);
  //}

  scDecompileMain();
}
```

Where `scDecompileMain()` is defined in Generator.h as follows:

```
       void scDecompileMain()
       {
              scDecompileMain_(output_);
       }
```

Where `scDecompileMain_` is defined in Generator.cc as follows:

```
void
GeneratorBinary::scDecompileMain_(std::ofstream& output)
{
  cppcc::scb::SyntaxControlledBinary& bin =
       generator_.parser_.binary_;

  cppcc::scr::tag::Long axiom = bin.head()->cntbeg;
  scDecompile_(output, axiom);
  if (StateLine_) {
         output << std::endl;
  }

  output << std::endl;
}
```

Where `scDecompile_` is recursive method that decompiles the whole Syntax-Controlled Binary into specified file. The `scDecompile_ has the following code:`

```
void
GeneratorBinary::scDecompile_
       (std::ofstream& output
       ,cppcc::scr::tag::Long& tag)
{
// (gb.0)
       bool _isDebug_ = false;

       if (_isDebug_) {
         std::cout
         << "scDecompile"
         << " tag:" << tag
         << std::endl;
       }
```

```
  if(!tag) return;

// (gb.1)
  cppcc::scr::tag::Long kkk;
  cppcc::scr::tag::Long nnn;
  cppcc::scr::tag::setlongedf(&tag,kkk,nnn);

       if (_isDebug_) {
         std::cout
         << "scDecompile"
         << " k:" << kkk
         << " n:" << nnn
         << " s:" << generator_.parser_.grammarSymbols_.size()
         << std::endl;
       }

  if ((kkk <0) || (kkk >= generator_.parser_.grammarSymbols_.size())) {
         return;
  }

// (gb.2)
  scDecompileSpace(output, kkk);

  StateLine_= 1;

  cppcc::scb::SyntaxControlledBinary& bin =
       generator_.parser_.binary_;

  if(cppcc::com::PLS_IDENTIFIER == kkk) {
    output << bin.ptrids(nnn) << " ";

    if (_isDebug_) {
               std::cout
               << "scDecompile"
               << " k:" << kkk
               << " n:" << nnn
               << " s:" << generator_.parser_.grammarSymbols_.size()
               << " id:" << "'" << bin.ptrids(nnn) << "'"
               << std::endl;
         }

    LastKey_= kkk;
    return;
  }

// (gb.3)

  if(cppcc::com::PLS_TERMTOKENOFRULE_TOKEN == kkk) {
    std::string tn = bin.ptrids(nnn);

    std::string tn2 = generator_.parser_.tokenizer_.language_->tokens_.gid(tn);

    output
      << "'"
      << tn2
```

```
      << "'"
    ;

    LastKey_= kkk;
    return;
  }

// (gb.4)
  if(cppcc::com::PLS_TERMINAL_TOKEN == kkk) {
    cppcc::lex::Token& token =
      generator_.parser_.tokenizer_.language_->getToken(nnn);
    output << token.gid() << " " ;

    makeAlignment(token);

    LastKey_= kkk;
    Lastnnn_= nnn;
    return;
  }

// (gb.5)
  cppcc::scb::SyntaxControlledBinary::kwnirType* k =
    bin.setptrkwn(kkk);
  if(k) {

        if (_isDebug_) {
               std::cout
               << "scDecompile"
               << " tag:" << tag
               << " k:" << kkk
               << " n:" << nnn
               << " kwn:" << (*k)
               << std::endl;
        }


    cppcc::scb::SyntaxControlledBinary::edpirType* e =
         bin.ptredp(k, nnn);
    if (e) {
         if (_isDebug_) {
               std::cout
               << "scDecompile"
               << " tag:" << tag
               << " k:" << kkk
               << " n:" << nnn
                    << " kwn:" << (*k)
                    << " edp:" << (*e)
               << std::endl;
         }

// (gb.6)
      cppcc::scr::tag::Long* uf = bin.fixed(e);
      cppcc::scr::tag::Long* ud = bin.dynamic(e);

         switch (kkk)
         {
```

```
   case cppcc::com::PLS_TEXT_TOKEN:
   {
      if (e->fl && (2 == e->fl)) {
        cppcc::scr::tag::Long kkk0 = bin.fixed(e, 0).t;
        cppcc::scr::tag::Long nnn0 = bin.fixed(e, 0).d;

        cppcc::scr::tag::Long kkk1 = bin.fixed(e, 1).t;
        cppcc::scr::tag::Long nnn1 = bin.fixed(e, 1).d;

         if((cppcc::com::PLS_TERMINAL_TOKEN == kkk0)
            && (cppcc::com::PLS_STRING_TOKEN == kkk1)) {
               cppcc::lex::Token& token0 =
                  generator_.parser_.tokenizer_.language_->getToken(nnn0);

               cppcc::scb::SyntaxControlledBinary::kwnirType*k111 =
                 bin.setptrkwn(kkk1);
               if(k111) {
                    cppcc::scb::SyntaxControlledBinary::edpirType* e111 =
                        bin.ptredp(k111, nnn1);
                    cppcc::scr::tag::Long* u111f = bin.fixed(e111);

                    output
                      << token0.token_
                      << specialCharacterConversion((char*)u111f);

                    LastKey_= kkk;
                }
            }
      }
      return;
   }
   case cppcc::com::PLS_STRING_TOKEN:
   {
     if(uf) {
      output << "\"" << ((char*)uf) << "\"";

    LastKey_= kkk;
  }
  return;
}
case cppcc::com::PLS_INTEGER_TOKEN:
{
  if(uf) {
    output << (*((cppcc::scr::tag::Long*)uf));
    LastKey_= kkk;
  }
  return;
}
case cppcc::com::PLS_FLOAT_TOKEN:
{
  if(uf) {
    output << (*((cppcc::scr::tag::Real*)uf));
    LastKey_= kkk;
  }
  return;
}
```

```
// (gb.7)
        default:
        {
             if (uf) {
               for (std::size_t i = 0, z = e->fl; i < z; i++) {
                    scDecompile_(output, uf[i]);
               }
             }

             if (ud) {
               for (std::size_t i = 0, z = e->dl; i < z; i++) {
                 scDecompile_(output, ud[i]);
               }
             }
        }
      }
    }
  }
}
```

In the code above I put commens (gb.0), (gb.1), etc... Here are some related comments.

(gb.0)
The `GeneratorBinary::scDecompile_` for the given output file stream and reference to tag recursively calls itself decompiling the whole binary context affiliated with the tag. Originally, inside GeneratorBinary::scDecompileMain_, that tag represents grammar axiom.

(gb.1)

The decompile operation is ended if provided tag is 0.
Otherwise tag is unpacked into to variable, kkk, and nnn. Note, that in this case kkk represents Parsing Model Symbol as integer symbol ID, and nnn represents Parsing Model Rule as integer rule ID (for the given symbol ID).

(gb.2)

The decompile operation is ended if value of kkk is out of range.
Otherwise a proper space/indentation is generated by calling `scDecompileSpace` followed by `cppcc::com::PLS_IDENTIFIER` identifier processing. If current element represents identifier, then its representation is generated and in this case the decompile operation is ended.

(gb.3)

If current element represents terminal token originally encoded in a single quotes, then its representation is generated and in this case the decompile operation is ended.

(gb.4)

If current element represents terminal token originally encoded in a single quotes and belongs to delimiters category, then its representation is generated and in this case the decompile operation is ended.

(gb.5)

This case happenes when current element represents some Parsing Model Rule being populated. Variable k defined as:

```
  cppcc::scb::SyntaxControlledBinary::kwnirType* k =
    bin.setptrkwn(kkk);
```
represents Parsing Model Symbol. Variable e defined as:

```
    cppcc::scb::SyntaxControlledBinary::edpirType* e =
         bin.ptredp(k, nnn);
```
represents Parsing Model Rule.

(gb.6)

For the given Parsing Model Rule represented by variable e, fixed and dynamic arrays are populated this way:

```
      cppcc::scr::tag::Long* uf = bin.fixed(e);
      cppcc::scr::tag::Long* ud = bin.dynamic(e);
```
Followed by switch statement. For that switch the following cases are processed first:

```
case cppcc::com::PLS_TEXT_TOKEN:
case cppcc::com::PLS_STRING_TOKEN:
case cppcc::com::PLS_INTEGER_TOKEN:
case cppcc::com::PLS_FLOAT_TOKEN:
```

(gb.7)

The switch default case has the following code:

```
        default:
        {
             if (uf) {
               for (std::size_t i = 0, z = e->fl; i < z; i++) {
                    scDecompile_(output, uf[i]);
               }
             }

             if (ud) {
               for (std::size_t i = 0, z = e->dl; i < z; i++) {
                 scDecompile_(output, ud[i]);
               }
             }
        }
```
This way Parsing Model Rule fixed part is processed first, followed by Parsing Model Rule dynamic part processed second.

GeneratorBinary class provides implementation of its methods based on Syntax-Controlled Binary API presented in Chapter "C++ CCS Syntax-Controlled Binary API".

GeneratorRuntime decompile method implementation

GeneratorRuntime class implements decompile method is implemented similar to GeneratorBinary. The related `GeneratorRuntime::scDecompile_` is presented here:

```
void
GeneratorRuntime::scDecompile_
       (std::ofstream&            output
       ,cppcc::scr::tag::Long&    tag)
{
```

```
if(!tag) return;

cppcc::scr::tag::Long rrr = tag;
cppcc::scr::tag::Long kkk;
cppcc::scr::tag::Long nnn;
cppcc::scr::tag::setlongedf(&rrr,kkk,nnn);

scDecompileSpace(output, kkk);

StateLine_= 1;

cppcc::scr::SyntaxControlledRuntime::cnm&  currentContext =
     generator_.parser_.runtime_.cnms_.dataByKey
   (generator_.parser_.runtime_.cnmnumCurrent_);
cppcc::scr::SyntaxControlledRuntime::kwn&  kwnInstance =
     generator_.parser_.runtime_.kwnLookup(kkk);


if(cppcc::com::PLS_IDENTIFIER == kkk) {
     cppcc::scr::SyntaxControlledRuntime::nam namInstance=
      currentContext.namn_.dataByIndex(nnn);

output << namInstance.namkey_ << " " ;

  LastKey_= kkk;
  return;
}

if(cppcc::com::PLS_TERMTOKENOFRULE_TOKEN == kkk) {
  cppcc::scr::SyntaxControlledRuntime::nam namInstance=
    currentContext.namn_.dataByIndex(nnn);

  std::string tn = namInstance.namkey_;

  std::string tn2 = generator_.parser_.tokenizer_.language_->tokens_.gid(tn);


  output
    << "'" << tn2 << "'"
  ;

  LastKey_= kkk;
  return;
}

if(cppcc::com::PLS_TERMINAL_TOKEN == kkk) {
  cppcc::lex::Token& token =
    generator_.parser_.tokenizer_.language_->getToken(nnn);
  output << token.gid() << " " ;

  makeAlignment(token);


  LastKey_= kkk;
  Lastnnn_= nnn;
  return;
```

```
}

cppcc::scr::SyntaxControlledRuntime::edp& edpInstance =
    kwnInstance.edps_.dataByKey(nnn);

switch (kkk)
{
  case cppcc::com::PLS_STRING_TOKEN:
  {
    if(edpInstance.edpfix_.size()) {

     output << "\"" << ((char*)&edpInstance.edpfix_[0]) << "\"";

      LastKey_= kkk;
    }
    return;
  }
  case cppcc::com::PLS_INTEGER_TOKEN:
  {
    if(edpInstance.edpfix_.size()) {
     output << edpInstance.edpfix_[0];
      LastKey_= kkk;
    }
    return;
  }
  case cppcc::com::PLS_FLOAT_TOKEN:
  {
    if(edpInstance.edpfix_.size()) {
     output << (*((cppcc::scr::tag::Real*)&edpInstance.edpfix_[0]));
      LastKey_= kkk;
    }
    return;
  }
  case cppcc::com::PLS_TEXT_TOKEN:
  {
    if(edpInstance.edpfix_.size()) {
     if(2 == edpInstance.edpfix_.size()) {

       cppcc::scr::tag::Long kkk0;
       cppcc::scr::tag::Long nnn0;
       cppcc::scr::tag::setlongedf(&edpInstance.edpfix_[0],kkk0,nnn0);

       cppcc::scr::tag::Long kkk1;
       cppcc::scr::tag::Long nnn1;
       cppcc::scr::tag::setlongedf(&edpInstance.edpfix_[1],kkk1,nnn1);

       if((cppcc::com::PLS_TERMINAL_TOKEN == kkk0)
         && (cppcc::com::PLS_STRING_TOKEN == kkk1)) {
           cppcc::lex::Token& token0 =
              generator_.parser_.tokenizer_.language_->getToken(nnn0);

           cppcc::scr::SyntaxControlledRuntime::cnm&  currentContext1 =
           generator_.parser_.runtime_.cnms_.dataByKey
              (generator_.parser_.runtime_.cnmnumCurrent_);
           cppcc::scr::SyntaxControlledRuntime::kwn&  kwnInstance1 =
           generator_.parser_.runtime_.kwnLookup(kkk1);
```

```
            cppcc::scr::SyntaxControlledRuntime::edp& edpInstance1 =
                kwnInstance1.edps_.dataByKey(nnn1);

            output
              << token0.token_
              << specialCharacterConversion((char*)&edpInstance1.edpfix_[0]);

          }
        }

         LastKey_= kkk;
       }
       return;
     }
     default:
     {
          if (edpInstance.edpfix_.size()) {
            for (std::size_t i = 0, z = edpInstance.edpfix_.size(); i < z; i++) {
                 scDecompile_(output,edpInstance.edpfix_[i]);
            }
          }

          if (edpInstance.edpdyn_.size()) {
            for (std::size_t i = 0, z = edpInstance.edpdyn_.size(); i < z; i++) {
              scDecompile_(output,edpInstance.edpdyn_[i]);
            }
          }
     }
  }

}
```

GeneratorRuntime class provides implementation of its methods based on Syntax-Controlled Runtime API presented in Chapter "C++ CCS Syntax-Controlled Runtime API".

C++ CCS Syntax-Controlled Runtime API

The C++ CCS Syntax-Controlled Runtime API is defined in these files:

- include/SyntaxControlledRuntime.h and src/SyntaxControlledRuntime.cc

FIG. 8 - FIG.18 are dedicated to Syntax-Controlled Runtime. Files SyntaxControlledRuntime.h and SyntaxControlledRuntime.cc have this comment inherited from C CCS:

```
////////////////////////////////////////////////////////////////////////
//
//
//      sys |--------|        |--------|cnm     |--------|nam
//          |        |------->| cnmnum |------->| namnum |
//          |        |  cnms  |        |  namn  | namkey |
//          |        |        |        |------->| namend |*
//          |        |        |        |  nams  | namsou |*
//          |--------|        |--------|        |--------|
//                              |kwns
//                              V
//                            |--------|kwn     |--------|edp
//                            | kwnnum |------->| edpnum |
//                            |        |  edps  | edpfix |*
//                            |--------|        | edpdyn |*
//                                               |--------|
//
////////////////////////////////////////////////////////////////////////
```

That diagram is C CCS Syntax-Controlled Runtime database schema, a custom database management system was used in case of C CCS representing Syntax-Controlled Runtime as a database. The `SyntaxControlledRuntime` class keeps those names, i.e., 'sys', 'cnm', 'nam', 'kwn', and 'edp' for having easy mapping with C CCS version. The patent [3] and FIG. 8 - FIG.18 introducing Parsing Model have different names for entities and their relationships. Table below, Parsing Model entity name mapping between C and C++ CCS versions, provides corresponding mapping.

Table. Parsing Model name mapping between C and C++ CCS versions.

C CCS	C++ CCS	Patent	Parsing Model
sys	SyntaxControlledRuntime	SyntaxControlledRuntime FIG. 8	Syntax-Controlled Runtime
cnm	SyntaxControlledRuntime:: cnm	SyntaxControlledRuntime:: Context FIG.10, FIG.11	Syntax-Controlled Runtime Context
nam	SyntaxControlledRuntime:: nam	SyntaxControlledRuntime:: Name FIG.12, FIG. 13	Syntax-Controlled Runtime Name
kwn	SyntaxControlledRuntime:: kwn	SyntaxControlledRuntime:: Symbol FIG.14, FIG. 15	Syntax-Controlled Runtime Symbol
edp	SyntaxControlledRuntime:: edp	SyntaxControlledRuntime:: Rule FIG.16, FIG. 17	Syntax-Controlled Runtime Rule
cnms	SyntaxControlledRuntime:: SystemContextMap	SyntaxControlledRuntime:: ContextContainer FIG.8, FIG.10	Syntax-Controlled Runtime Context Container
namn	SyntaxControlledRuntime::	SyntaxControlledRuntime::	Syntax-Controlled Runtime

	ContextNameByKeyMap	NameContainer FIG. 12	NamesByName Container
nams	SyntaxControlledRuntime:: ContextNameByKeyMap	SyntaxControlledRuntime:: NameContainer FIG. 12	Syntax-Controlled Runtime NamesByNumber Container
kwns	SyntaxControlledRuntime:: ContextKeyWordByNumberMap	SyntaxControlledRuntime:: SymbolContainer FIG. 14	Syntax-Controlled Runtime Symbol Container
edps	SyntaxControlledRuntime:: KeyWordRuleDefinitionByNumberMap	SyntaxControlledRuntime:: RuleContainer FIG. 16	Syntax-Controlled Runtime Rule Container

FIG.9 shows UML diagram representing C++ CCS custom container as a template class that takes two template parameters <D> and <K>. The <D> stands for data type, and <K> is for key type.

struct SyntaxControlledRuntime

The SyntaxControlledRuntime.h has the following definition of SyntaxControlledRuntime:

. . .

```
struct SyntaxControlledRuntime
{

  struct cnm;
  struct nam;
  struct kwn;
  struct edp;

  typedef tag::MapVectorContainer<cnm,tag::Long>    SystemContextMap;
  typedef tag::MapVectorContainer<nam,std::string>  ContextNameByKeyMap;
  //typedef tag::MapVectorContainer<nam,tag::Long>    ContextNameByNumberMap;
  typedef tag::MapVectorContainer<kwn,tag::Long>    ContextKeyWordByNumberMap;
  typedef tag::MapVectorContainer<edp,tag::Long>    KeyWordRuleDefinitionByNumberMap;

  //typedef tag::MapVectorContainer<nam,std::string>::Map ContextNameByKeyMapMap;

  struct nam {
    tag::Long         cnmnum_;
    //tag::Long       namnum_;
    std::string       namkey_;
  };

  struct edp {
    tag::Long         cnmnum_;
    tag::Long         kwnnum_;
    tag::Long         edpnum_;
    std::vector<tag::Long>   edpfix_;
    std::vector<tag::Long>   edpdyn_;
  };

  struct kwn {
    tag::Long         cnmnum_;
    tag::Long         kwnnum_;
    KeyWordRuleDefinitionByNumberMap edps_;
  };
```

```
struct cnm {
  tag::Long                       cnmnum_;
  ContextNameByKeyMap             namn_;
  //ContextNameByNumberMap        nams_;
  ContextKeyWordByNumberMap       kwns_;
};


SystemContextMap     cnms_;
tag::Long            cnmnumCurrent_;
int                  optimizationMode_;
int                  debug_;

SyntaxControlledRuntime()
       : cnms_()
       , cnmnumCurrent_(0)
       , optimizationMode_(0)
       , debug_(0)
{}

SyntaxControlledRuntime(int optimizationMode)
       : cnms_()
       , cnmnumCurrent_(0)
       , optimizationMode_(optimizationMode)
       , debug_(0)
{}

~SyntaxControlledRuntime()
{}


kwn&        kwnLookup(tag::Long  k);

void        edfcnst(tag::Long* l);
void        celcnmstr(tag::Long* l) { edfcnst(l); }
void        edfepst(tag::Long k,tag::Long* l);
void        celedpstr(tag::Long k,tag::Long* l) { edfepst(k,l); }

edp& celedpfnd(tag::Long* kn);

void        edfeptd(edp& e, tag::Long d[],tag::Long m,tag::Long sh);
void        edfeptd(edp& e, const std::vector<tag::Long>& d);
void        celedpmit(edp& e, tag::Long d[],tag::Long m,tag::Long sh)
      {edfeptd(e,d,m,sh);}

void        edfepfx(edp& e, tag::Long d[],tag::Long m,tag::Long sh);
void        edfepfx(edp& e, const std::vector<tag::Long>& f);
void        celedpmfx(edp& e, tag::Long d[],tag::Long m,tag::Long sh)
      {edfepfx(e,d,m,sh);}

void        edfnmst(const std::string& n, tag::Long* l);
void        celnamstr(const std::string& n, tag::Long* l)
      {edfnmst(n,l); }

void        celedpfst_
```

```
        (int                        flag
        ,tag::Long  k
        ,tag::Long* l
        ,std::vector<tag::Long>& v);

  void        celedpfst
        (int                        flag
        ,tag::Long  k
        ,tag::Long* l
        ,tag::Long  t[]
        ,tag::Long  m)
  {
       std::vector<tag::Long>      fixed(t, t+m);
       return celedpfst_(flag,k,l,fixed);
  }

  void        celedpfst
        (int                        flag
        ,tag::Long  k
        ,tag::Long* l
        ,std::vector<tag::Long>& v)
  {
       return celedpfst_(flag,k,l,v);
  }

  void        celedpfss(tag::Long k, tag::Long*l, const std::string& t);

};
```

I hope that presented on 3 pages SyntaxControlledRuntime code would be easier to understand keeping in mind Parsing Model. Some additional implementation details are defined in SyntaxControlledRuntime.cc file. Also all SyntaxControlledRuntime members are based on `tag::MapVectorContainer` class template.

template struct MapVectorContainer

The `MapVectorContainer` is dedined in SyntaxControlledRuntime.h file in tag namespace as follows:

```
template<typename D, typename K = std::string>
struct      MapVectorContainer
{

  //typedef   MapVectorContainer<D,K> Container;
  typedef   K                        Key;
  typedef   tag::Long               Index;
  typedef   std::map<Key, Index>     Map;
  typedef   std::vector<Key>         Vector;
  typedef   std::vector<D>           Data;

  Map       name2index;
  Vector    index2name;
  Data      index2data;

  MapVectorContainer ()
    : name2index()
    , index2name()
```

```
  , index2data()
{}

K   lastKey(tag::Long* l, const K& defaultKey) {
  return
    l
      ? (
        index2name.size()
          ? index2name[index2name.size()-1]
          : defaultKey
      )
      : defaultKey
  ;
}


K   lastKey(const K& defaultKey) {
  return
           index2name.size()
                  ? index2name[index2name.size()-1]
          : defaultKey
  ;
}


D&  dataByIndex(Index  index) {

  if ((index < 0) || (index >= static_cast<Index>(index2data.size()))) {
    CPPCC_THROW_EXCEPTION(
      << "MapVectorContainer::dataByIndex("
      << index
      << ")"
      << " - failed! - Argument is out off range: [0.."
      << index2data.size()
      << ")"
    )
  }

  return  index2data[index];
}

D&  dataByKey(const Key&  theKey) {
  Index idx;
  bool z = lookup(theKey, idx);

  if (!z) {
    CPPCC_THROW_EXCEPTION(
      << "MapVectorContainer::dataByKey('"
      << theKey
      << "')"
      << " - failed! - data member has not been defined."
    )
  }

  return index2data[idx];
}
```

```
void  store(const D&   theData, const Key&  theKey) {
  Index idx;
  bool z = lookup(theKey, idx);

  if (z) {
    CPPCC_THROW_EXCEPTION(
      << "MapVectorContainer::store('"
      << theKey
      << "')"
      << " - failed! - data member has been already defined."
    )
  }

  idx = static_cast<Index>(index2name.size());
  index2name.push_back(theKey);
  index2data.push_back(theData);

  std::pair<typename Map::iterator, bool>
    insertPair = name2index.insert
      (std::pair<const Key, Index>
        (theKey, idx)
      );

  if (!insertPair.second) {
    CPPCC_THROW_EXCEPTION(
      << "MapVectorContainer::store('"
      << theKey
      << "')"
      << " - assert check failed! - data member has been already defined."
    )
  }
}

void  update(const D&   theData, const Key&  theKey) {
  Index idx;
  bool z = lookup(theKey, idx);

  if (z) {
    index2name[idx] = theKey;
    index2data[idx] = theData;
  }
  else {
    idx = static_cast<Index>(index2name.size());
    index2name.push_back(theKey);
    index2data.push_back(theData);

    std::pair<typename Map::iterator, bool>
      insertPair = name2index.insert
        (std::pair<const Key, Index>
          (theKey, idx)
        );

    if (!insertPair.second) {
      CPPCC_THROW_EXCEPTION(
        << "MapVectorContainer::update('"
```

```
        << theKey
        << "')"
        << " - assert check failed! - data member has been already defined."
      )
    }
  }
}

bool  lookup(const Key& kw, Index& idx) const {
  typename Map::const_iterator  iter = name2index.find(kw);

  if(iter == name2index.end()) {
    return false;
  }

  idx = iter->second;

  return true;
}

void    dump_(std::ostream& strm)
{
       strm
           << std::endl;

         for (register std::size_t i = 0; i < index2name.size(); i++) {
            strm
             << i
             << " "
             << "'"
             << index2name[i]
             << "'"
             << std::endl;
         }
         strm << std::endl;

         typename Map::const_iterator    iend = name2index.end();
         typename Map::const_iterator    iter = name2index.begin();
         for (; (iend != iter); ++iter) {
            strm
             << iter->second
             << " "
             << "'"
             << iter->first
             << "'"
             << " -> "
             << "'"
             << index2name[iter->second]
             << "'"
             << std::endl;

         }
}

void    dump() {
       dump_(std::cout);
```

```
  }

  std::string  all() {
    std::string r("");

    typename Map::const_iterator    iend = name2index.end();
    typename Map::const_iterator    iter = name2index.begin();

    for (;(iend != iter);++iter) {

      r
        .append(iter->first)
        .append(" ");

    }

    return r;
  }

};
```

FIG. 9 shows UML diagram for `MapVectorContainer` template.

Parser class methods using Syntax-Controlled Runtime API

The Parser class has runtime_ member defined as follows:

```
  cppcc::scr::SyntaxControlledRuntime    runtime_;
```

The Parser class is the parent class for all Parser classes generated by C++ CCS cppcc program. The Parser class has the following methods that are implemented as some actions using `runtime_` member:

- `generateBinaryFromRuntime`
- `METAACTBEG`
- `METAACTEND`
- `crelasedp`
- `crefixe`
- `crelasdyn`
- `identifierAction`
- `integerTokenAction`
- `stringTokenAction`
- `textTokenAction`
- `termToken`
- `grammarName`

Let's review SGDL on SGDL (meta.sgr) axiom defined by grammar rule and related `metaFirstParser::grammar` method:

```
void
metaFirstParser::grammar(cppcc::scr::tag::Long& tag)
/*
 ( grammar ::=
 0 = "  METAACTBEG();" = '(' grammarNameDef { rule }
 ')' 0 = "  METAACTEND();" = )
*/
{
  TagVector d;
  TagVector f(2);
```

```
  METAACTBEG();
  skipToken(KW_LEFTPARENTHESISTERMTOKEN,f[0]);
  grammarNameDef(f[1]);

  for(;;) {
       if(iseof()) break;

       if(kword() == KW_LEFTPARENTHESISTERMTOKEN) {

         d.push_back(0);
         rule(d.back());
       }
       else break;
  }

  d.push_back(0);
  skipToken(KW_RIGHTPARENTHESISTERMTOKEN,d.back());

  crelasedp(KW_GRAMMAR,f,d,&tag);

  METAACTEND();
}
```

Actions, `METAACTBEG()` and `METAACTEND()` are executed as first and last actions performed by the parser. The `METAACTBEG()` is defined in Parser.h as follows:

```
  void METAACTBEG()
  {
    runtime_.edfcnst(&context_);
  }
```

The purpose of `edfcnst` is to create new Parsing Model Context and to return its id into provided variable. The `SyntaxControlledRuntime::edfcnst` is defined as follows in `SyntaxControlledRuntime.cc`:

```
////////////////////////////////////////////////////////////////////////////
// Create new 'cnm' instance.
////////////////////////////////////////////////////////////////////////////
void
SyntaxControlledRuntime::edfcnst(tag::Long* l)
{
       cnm c;

       c.cnmnum_ = cnms_.lastKey(l,InitialContextNumber);
       c.cnmnum_++;
       cnms_.store(c, c.cnmnum_);

       if (l) *l = c.cnmnum_;
       cnmnumCurrent_ = c.cnmnum_;

       if (debug_) {
              std::cout << "edfcnst:"
              << " cnmnum:" << c.cnmnum_
              << " cnmnumCurrent_:" << cnmnumCurrent_
              << std::endl;
       }

       return;
}
```

So after calling `METAACTBEG()` the parser has a newly created Parsing Model Context that becomes an owner of all subsequent Parsing Model Name, Symbol, and Rule instances with their relationships.

The `METAACTEND()` is defined in Parser.h as follows:

```
  void METAACTEND()
  {
        // (A)
        /*
         *   static std::string   predef_[] =
  {
    "WrongKeyWord"
    ,"identifier"
    ,"stringToken"
    ,"integerToken"
    ,"floatToken"
    ,"textToken"
    ,"termToken"
    ,"termTokenOfRule"
  };
         */
        runtime_.edfnmst("floatToken",0);
        //runtime_.edfnmst("xmlToken",0);
        runtime_.edfnmst("textToken",0);
        runtime_.edfnmst("integerToken",0);
        //runtime_.edfnmst("identifier",0);

        // (A.1) - decompile binary - new phase v.s. ccc
        // check shell flag.
        // bool isRuntimeDecompiled = compiler_.shell_.isRuntimeDecompiled_;
        // bool isRuntimeDecompiled = true;
        // if (isRuntimeDecompiled) {
        //     //std::string decompilingRuntime = filename_ + ".urt";
        //     //runtime_.decompile(decompilingRuntime, context_, axiom_);
        // runtime_.decompile(context_, axiom_);
        //}
        bool isRuntimeDecompiled = true;
        if (isRuntimeDecompiled) {
              std::string decompilingRuntime = filename_ + ".urt";
              generator_.decompileRuntime(decompilingRuntime);
        }

        // (B)
        // Make binary_ generation
        // generateBinary();
        //if (generator_) {
        //  generator_->generateBinaryFromRuntime();
        //}
        generateBinaryFromRuntime();
        bool isTesting = false;
        if (isTesting) {
              std::string dName =  filename_ + ".ubrt";
              generator_.decompileBinary(dName);
        }

        // (C) unload binary_ into *.fgr file
```

```
        // unloadBinary();
        std::string binaryFilaName = filename_ + ".fgr";
        binary_.writeBinary(binaryFilaName);

        // (D)
        // generateParser();
        // create <>Parser.cc
        // create <>Parser.h

        // D.1
        bool isGeneratedFromBinary = true;
        if (isGeneratedFromBinary) {
               generator_.generateFromBinary(filename_);
        }

        // D.2
        bool isGeneratedFromRuntime = true;
        if (isGeneratedFromRuntime) {
               generator_.generateFromRuntime(filename_);
        }
}
```

The `edfnmst()` has the following definition in `SyntaxControlledRuntime.cc`:

```
void
SyntaxControlledRuntime::edfnmst(const std::string& n, tag::Long* l)
{
       if (debug_) {
              std::cout
              << "edfnmst:0:"
              << " id:" << "'" << n << "'"
              << " cnmnumCurrent_:" << cnmnumCurrent_
              << std::endl;
       }

  //Wrong::::::::::
  //cnm& currentContext = cnms_.dataByIndex(cnmnumCurrent_);
  cnm&  currentContext = cnms_.dataByKey(cnmnumCurrent_);
  tag::Long   idx;


       if (debug_) {
              std::cout
              << "edfnmst:1:"
              << " id:" << "'" << n << "'"
              << " cnmnumCurrent_:" << cnmnumCurrent_
              << std::endl;
       }


  if (!currentContext.namn_.lookup(n, idx)) {
       nam                  namInstance;
       namInstance.cnmnum_ = cnmnumCurrent_;
       namInstance.namkey_ = n;
       currentContext.namn_.store(namInstance,n);
       idx = currentContext.namn_.index2data.size() - 1;
```

```
        if (debug_) {
                std::cout
                << "edfnmst:2:"
                << " id:" << "'" << n << "'"
                << " cnmnumCurrent_:" << cnmnumCurrent_
                << " idx:" << idx
                << std::endl;
        }

  }

  if (l) *l = idx;

  return;
}
```

The `SyntaxControlledRuntime::edfnmst` creates Parsing Model Name instance if it does not exists in a given Parsing Model Context. Note, that `generateBinaryFromRuntime()` is called by `METAACTEND()` on step (B).

The `crelasedp` is defined in Parser.h with the following signature:

```
  void  crelasedp
        (int          k
        ,TagVector&   f
        ,TagVector&   d
        ,cppcc::scr::tag::Long* l)
```

For the given Parsing Model Symbol id presented by argument k, fixed and dynamic Parsing Model Rule elements presented by arguments f and d, the new tag is created as `cppcc::scr::tag::Long* l.`

The `crefixe` is defined in Parser.h with the following signature:

```
  void  crelasedp
        (int          k
        ,TagVector&   f
        ,cppcc::scr::tag::Long* l)
```

For the given Parsing Model Symbol id presented by argument k and fixed Parsing Model Rule element presented by argument f, the new tag is created as `cppcc::scr::tag::Long* l.`

The `crelasdyn` is defined in Parser.h with the following signature:

```
  void  crelasdyn
        (int          k
        ,TagVector&   d
        ,cppcc::scr::tag::Long* l)
```

For the given Parsing Model Symbol id presented by argument k and dynamic Parsing Model Rule element presented by argument d, the new tag is created as `cppcc::scr::tag::Long* l.`

That variable l in all three cases of `crelasedp, crefixe, and crelasdyn` is packed as follows:

```
  tag::Long  edpnum;
  tag::Long  r;
. . .
  tag::setedflong(&r,k,edpnum);
. . .
  if (l) *l = r;
. . .
```

where `edpnum` represents Parsing Model Rule id corresponds to Parsing Model Rule instance id created for the given Parsing Model Symbol instance with Parsing Model Symbol id equals to argument k.

The identifierAction is used by identifier method as follows:

```
  void        identifier(cppcc::scr::tag::Long&  tag)
  {
        identifierAction(tag);
        tokenizer_.getNextToken();
  }
```

Where identifierAction is defined as follows:

```
  void        identifierAction(cppcc::scr::tag::Long&   tag)
  {
        cppcc::scr::tag::Long n;
        runtime_.edfnmst(tokenizer_.id_, &n);

        if (tokenizer_.debug_) {
             std::ostringstream o;
             o
                    << "identifier:"
                    << " kw:" << tokenizer_.kword_
                    << " id:" << tokenizer_.id_
                    << " n:" << n
             ;
             tokenizer_.infoMessage(o.str());
        }

        cppcc::scr::tag::setedflong(&tag, cppcc::com::PLS_IDENTIFIER, n);

  }
```

This way runtime_ action is separated from tokenizer_ action. Note that
runtime_.edfnmst(tokenizer_.id_, &n);
creates Parsing Model Name instance returning Parsing Model Name id presented in n variable. The tag variable, that is output argument of identifierAction is populated by packing cppcc::com::PLS_IDENTIFIER and n (Parsing Model Name id) into that tag variable.

The integerTokenAction is used by integerTokenmethod as follows:

```
  void        integerToken(cppcc::scr::tag::Long&      tag)
  {
        integerTokenAction(tag);
        tokenizer_.getNextToken();
  }
```

Where integerTokenAction is defined as follows:

```
  void  integerTokenAction(cppcc::scr::tag::Long& tag)
  {
        if (tokenizer_.debug_) {
             std::ostringstream o;
             o
                    << "integerToken:0:"
                    << " kw:" << tokenizer_.kword_
                    << " iv:" << tokenizer_.ival_
             ;
             tokenizer_.infoMessage(o.str());
        }


        if(cppcc::com::PLS_INTEGER_TOKEN != tokenizer_.kword_) {
               tokenizer_.infoMessage("here must be integer or float token!");
```

```
                tokenizer_.errorMessage("wrong token!");
                return;
        }

        if (!tokenizer_.modeintfloat_) {
                cppcc::scr::tag::Long    value = tokenizer_.ival_;
                runtime_.celedpfst
                        (0, cppcc::com::PLS_INTEGER_TOKEN, &tag, &value, 1);

                if (tokenizer_.debug_) {
                     std::ostringstream o;
                     o
                            << "integerToken:1:celedpfst:"
                            << " kw:" << tokenizer_.kword_
                            << " iv:" << tokenizer_.ival_
                            << " tag:" << tag
                     ;
                     tokenizer_.infoMessage(o.str());
                }

        } else {
                cppcc::scr::tag::Real value = tokenizer_.rval_;
                runtime_.celedpfst
                        (0, cppcc::com::PLS_FLOAT_TOKEN, &tag,
                        reinterpret_cast<cppcc::scr::tag::Long*>(&value), 1);
        }
}
```

In that code above `tokenizer_.modeintfloat_` keeps a boolean value that is false integer token and is true for double/float token. In both cases value variable is populated in runtime_ instance as a proper Parsing Model Rule instance with a single fixed element. The returned tag is a pair of (`cppcc::com::PLS_INTEGER_TOKEN`, Parsing Model Rule id) for integer token or pair of (`cppcc::com:: PLS_FLOAT_TOKEN`, Parsing Model Rule id) for float/double token.

The `stringTokenAction, textTokenAction` have a similar implementation in Parser.h.
The `termToken` is defined as follows in Parser.h:

```
void                termToken(cppcc::scr::tag::Long&   tag)
{
        if (tokenizer_.debug_) {
             std::ostringstream o;
             o
                    << "termToken:0:"
                    << " kw:" << tokenizer_.kword_
                    << " iv:" << tokenizer_.termString_
             ;
             tokenizer_.infoMessage(o.str());
             std::cout << o.str() << std::endl;
        }

        if(cppcc::com::PLS_TERMINAL_TOKEN != tokenizer_.kword_) {
               tokenizer_.infoMessage("here must be terminal token!");
               tokenizer_.errorMessage("wrong token!");
               return;
        }

        // find tokenizer_.termString_
```

```
        // in tokenizer_.language_->tokens_
        cppcc::lex::Tokens::Name2IndexMap::const_iterator cIter =
          tokenizer_.language_->tokens_.name2index_.find(tokenizer_.termString_);

        std::string e("");
        std::string&       t =
          (tokenizer_.language_->tokens_.name2index_.end() == cIter)
            ? tokenizer_.termString_
            : (
              ((cIter->second >= 0)
              && (cIter->second <
                static_cast<int>(tokenizer_.language_->tokens_.tokens_.size())))
                ? (tokenizer_.language_->tokens_.tokens_[cIter->second]
                  ? tokenizer_.language_->tokens_.tokens_[cIter->second]->name_
                  : e
                )
                : e
            )
        ;

        if (e == t) {
           tokenizer_.errorMessage("could not process terminal token:");
           tokenizer_.infoMessage(tokenizer_.termString_);
           return;
        }

        cppcc::scr::tag::Long n;
        runtime_.edfnmst(t, &n);

        if (tokenizer_.debug_) {
             std::ostringstream o;
             o
                    << "termToken:1:"
                    << " kw:" << tokenizer_.kword_
                    << " iv:" << tokenizer_.termString_
                    << " n:" << n
             ;
             tokenizer_.infoMessage(o.str());
        }

        cppcc::scr::tag::setedflong
          (&tag
             ,cppcc::com::PLS_TERMTOKENOFRULE_TOKEN
          ,n);

        tokenizer_.getNextToken();
  }
```

Note, that this way termToken is represented as Parsing Model Name instance with the tag as a pair of (`cppcc::com::PLS_TERMTOKENOFRULE_TOKEN`, Parsing Model Name id).

The `grammarName()` is defined in Parser.h, it returns grammar name using Syntax-Controlled Runtime API by taking axiom tag, unpacking axiom tag into Parsing Model Symbol id and Parsing Model Rule id; setting up Parsing Model Context by having current context id; setting up Parsing Model Symbol instance by Parsing Model Symbol id; setting up Parsing Model Rule instance for the given Parsing Model Symbol instance and Parsing Model Rule id; unpacking Parsing Model Rule fixed part with index 1 getting Parsing Model Name id; setting up Parsing Model

Name instance by Parsing Model Name id; setting up returned grammar name as Parsing Model Name instance namkey_ member.

C++ CCS Syntax-Controlled Binary API

The C++ CCS Syntax-Controlled Binary API is defined in these files:

- include/SyntaxControlledBinary.h and src/SyntaxControlledBinary.cc

FIG. 19 - FIG.26 are dedicated to Syntax-Controlled Binary.

struct SyntaxControlledBinary

The include/SyntaxControlledBinary.h contains the following definition of `SyntaxControlledBinary`:

```
struct SyntaxControlledBinary
{
	struct kwnirType;

	typedef cppcc::scr::tag::Long		TagLong;
	typedef TagLong*			TagPointer;
	typedef cppcc::scr::tag::TypeInstance	TagInstance;
	typedef std::vector<kwnirType*>		kwnirTypeVector;

	std::vector<TagLong>			memory_;


	// head of dynamic array with its context
	struct contextType {
	  TagLong   cntlen;		// total length
	  TagLong   cntcur;		// current index:0 ... cntlen - 1

	  TagLong   cntbeg;		// axiom:struct TagInstance

	  TagLong   cntnid;		// total number of identifiers
	  TagLong   cntdid;   // dynamic array of identifiers map:
			//   sequantial number to TagLong

	  TagLong   cntnkw;		// total number of kwn's
	  TagLong   cntdkw;		// dynamic array of kwn's map:
			//  sequantial number to ptrType
	};

	// contextType size aligned
	std::size_t   SOFCNT() {
		return
			(sizeof(contextType) + sizeof(TagLong) - 1)/sizeof(TagLong)
		;
	}

	// head of kwn representation
	struct kwnirType {
	  TagLong     l;		// total number of kwn edp instances
	  TagLong     k;		// enum cppcc::...::KeyWords kwn key number
	  TagLong     d;		// beginning of kwn instances
	};
```

```
// kwnirType size aligned
std::size_t   SOFKWN() {
       return
              (sizeof(kwnirType) + sizeof(TagLong) - 1)/sizeof(TagLong)
       ;
}

// head of edp representation
struct edpirType {
  TagLong     fl;           // number of items of fixed part
  TagLong     dl;           // number of items of dynamic part
  TagLong     d;            // begin of edp instances
};

// edpirType size aligned
std::size_t   SOFEDP() {
       return
              (sizeof(kwnirType) + sizeof(TagLong) - 1)/sizeof(TagLong)
       ;
}

// TagInstance size aligned - should be always 1
std::size_t   SOFEDFTD() {
       return
              (sizeof(TagInstance) + sizeof(TagLong) - 1)/sizeof(TagLong)
       ;
}

const contextType*   head()
{
   return
     reinterpret_cast<const contextType*>
          (&memory_[0])
   ;
}

contextType*  uhead()
{
   return
     reinterpret_cast<contextType*>
          (&memory_[0])
   ;
}


// get identifier by its number in a sequential order.
const char*   ptrids(const TagLong id)
{
       return
         reinterpret_cast<const char*>
              (&memory_[memory_[head()->cntdid + id]])
       ;
}

// get identifier by its number in an alphabetic order.
const char*   ptridn(const TagLong id)
```

```
	{
		return
		  reinterpret_cast<const char*>
			(&memory_[memory_[head()->cntnid + head()->cntdid + 2*id]])
		;
	}

	// get identifier number by its number in an alphabetic order.
	TagLong	ptridnum(const TagLong id)
	{
		return
		  //memory_[memory_[head()->cntnid + head()->cntdid + 2*id+1]]
		  memory_[head()->cntnid + head()->cntdid + 2*id+1]
		;
	}

TagLong pdbidnsea(const std::string& id)
{

  for(TagLong j=0, z = (head()->cntnid); j < z; j++) {
    if (id == std::string(ptridn(j))) {
      return ptridnum(j);
    }
  }

  return(-1);
}

TagLong find_id(const std::string& id)
{
  return pdbidnsea(id);
}

	// get kwn representation (kwnirType*) by kwn id.
	//const
	kwnirType*	ptrkwn(const TagLong i)
	{
		return
			reinterpret_cast<kwnirType*>
			   //((&memory_[memory_[head()->cntdkw + i*SOFKWN()]])
			   (&memory_[head()->cntdkw + i*SOFKWN()])
		;
	}

	// get edp representation (edpirType*) by kwn id.
	//const
	edpirType*	ptredp(kwnirType* i, const TagLong j)
	{
		return
			reinterpret_cast<edpirType*>
				//((&memory_[memory_[i->d + j*SOFEDP()]])
				(&memory_[i->d + j*SOFEDP()])
		;
	}

	// convert TagLong to char*
```

```
char*  ptrchar(const TagLong d)
{
       return
              reinterpret_cast<char*>
                 (&memory_[d])
       ;
}

// convert TagLong to const char*
const char*   ptrcchar(const TagLong d)
{
       return
              reinterpret_cast<const char*>
                 (&memory_[d])
       ;
}

// aligned size of anything with initial size 'e'
std::size_t   lenofedpchar(std::size_t e)
{
       return
              (e + sizeof(TagLong) - 1)/sizeof(TagLong)
       ;
}

// sizeof binary internal representation
std::size_t   lenofir()
{
       return
              (head()->cntlen)*sizeof(TagLong)
       ;
}

// edp - grammar rule representation
struct ruleType {
  TagInstance*       f;            // fixed    part of rule
  TagInstance*       d;            // dynamic    part of rule
};

// populate ruleType& ppu instance having
//     kwnirType*    ppk
//     edpirType*    ppd
void   setptrrule
       (kwnirType*   ppk
       ,edpirType*          ppd
       ,ruleType&           ppu)
{
  ppu.f = 0;
  if (ppd && ppd->fl) {
       ppu.f = reinterpret_cast<TagInstance*>(&memory_[ppd->d]);
  }
  ppu.d = 0;
  if (ppd && ppd->dl) {
       ppu.d = reinterpret_cast<TagInstance*>(&memory_[ppd->d + ppd->fl]);
  }
}
```

```
//ruleType  setrule(kwnirType*     ppk, edpirType*             ppd)
ruleType  setrule(edpirType*              ppd)
{
       ruleType r;
       r.f = 0;
       r.d = 0;

       if (ppd && ppd->fl) {
              //r.f = reinterpret_cast<TagInstance*>(&memory_[ppd->d]);
              r.f = (TagInstance*)(&memory_[ppd->d]);
       }

       if (ppd && ppd->dl) {
              //r.d = reinterpret_cast<TagInstance*>(&memory_[ppd->d + ppd->fl]);
              r.d = (TagInstance*)(&memory_[ppd->d + ppd->fl]);
       }

       return r;
}

 cppcc::scr::tag::Long*           fixed
        (edpirType* e);
 cppcc::scr::tag::TypeInstance& fixed
        (edpirType* e, cppcc::scr::tag::Long i);

 cppcc::scr::tag::Long*           dynamic
        (edpirType* e);
 cppcc::scr::tag::TypeInstance& dynamic
        (edpirType* e, cppcc::scr::tag::Long i);


// method for searching & creating a kwnType descriptor by its enum number
//  enum KeyNum             ppkw;
//  struct kwnirType *ppk;
//const
kwnirType*    setptrkwn(const TagLong ppkw);
// initialize tag vector elements
void pdbinitf(std::vector<TagLong>& f)
{
       for (register std::size_t i = 0, s = f.size(); i<s; i++) {
         f[i] = 0;
       }
}

void writeBinary(const std::string& filename);
void readBinary(const std::string& filename);

void setkeywor(kwnirTypeVector& kwv) {
       for (register std::size_t i = 0, s = kwv.size(); i<s; i++) {
              kwv[i] = setptrkwn(static_cast<TagLong>(i));
       }
}


void edpDump
```

```
        (std::ostream&       strm
        ,SyntaxControlledBinary::edpirType&  edp)
    {
        strm
               << "edp="
               << "("
               << "fl=" << edp.fl
               << " dl=" << edp.dl
               << " d=" << edp.d
               << ")"
               << std::endl
        ;


          for (cppcc::scr::tag::Long i = 0, z = edp.fl; i < z; i++) {
          strm
            << "edp="
            << "("
            << "fl=" << edp.fl
            << " dl=" << edp.dl
            << " d=" << edp.d
            << " fixed:" << i
                << " " << memory_[edp.d+i]
                << "(" << (((TagInstance*)&memory_[edp.d+i])->t)
                << " " << (((TagInstance*)&memory_[edp.d+i])->d)
                << ")"
            << ")"
            << std::endl
          ;

          }


          for (cppcc::scr::tag::Long i = 0, z = edp.dl; i < z; i++) {
          strm
            << "edp="
            << "("
            << "fl=" << edp.fl
            << " dl=" << edp.dl
            << " d=" << edp.d
            << " dyn:" << i
                << " " << memory_[edp.d+edp.fl+i]
                << "(" << (((TagInstance*)&memory_[edp.d+edp.fl+i])->t)
                << " " << (((TagInstance*)&memory_[edp.d+edp.fl+i])->d)
                << ")"
            << ")"
            << std::endl
          ;

          }


        //return strm;
    }

struct Helper {
```

```
struct edp
{
	SyntaxControlledBinary*	b_;
	edpirType*		e_;

	edpirType&		e();

	TagLong*		fixed();
	TagLong*		dynamic();
	TagInstance&		fixed(const TagLong i);
	TagInstance&		dynamic(const TagLong i);

	operator bool() const { return e_; }
};

struct kwn
{
	//123// SyntaxControlledBinary*	b_;
	kwnirType*			k_;
	std::vector<edp>		edps_;

	edp&			e(const TagLong i);

	//kwnirType&		k();

	operator bool() const { return k_; }
};

SyntaxControlledBinary&			b_;
SyntaxControlledBinary::contextType&	h_;
std::vector<const char*>		ids_;
std::vector<const char*>		idn_;
std::vector<TagLong>			idi_;
std::vector<kwn>			kwns_;

Helper(SyntaxControlledBinary& b, const TagLong kwnSize)
  : b_(b)
  , h_(*(b_.uhead()))
  , ids_(h_.cntnid)
  , idn_(h_.cntnid)
  , idi_(h_.cntnid)
  //, kwns_(h_.cntnkw)
  , kwns_(kwnSize)
{
	populate();
}

//const contextType& h();
kwn&			k(const TagLong i);

//const std::string&  id(const TagLong i);
const char*		id(const TagLong i);

void			dump(std::ostream&  strm);
```

```
private:
	void populate();
};


private:

	cppcc::scr::tag::Long* fixed_(edpirType* e);
	cppcc::scr::tag::Long* dynamic_(edpirType* e);

};
```

The conversion of Syntax-Controlled Runtime into Syntax-Controlled Binary is performed by `Parser::generateBinaryFromRuntime()`. Next section will have some examples of using Syntax-Controlled Binary API during parser generation.

Using C++ CCS Syntax-Controlled Binary API by Parser Generator.

Having metaboot.sgr on SGDL with grammar name as metaboot, metabootGenerator.h and metabootGenerator.cc are generated. The metabootGenerator.cc has the following content:

```
////////////////////////////////////////////////////////////////////
//  metabootGenerator.cc
////////////////////////////////////////////////////////////////////

#include "Shell.h"
#include "Logger.h"
#include "Compiler.h"
#include "Parser.h"

#include "metabootParser.h"
#include "metabootKeyWordDefinition.h"
#include "metabootGenerator.h"

namespace cppcc {
namespace metaboot {

void
metabootGeneratorBinary::generate(const std::string& filename)
{
}

}
}
```

So, initially metabootGenerator has an empty `metabootGeneratorBinary::generate` to be developed.

In case of initial version of compiler compiler, the corresponding file was metaFirstGenerator.cc. The `metaFirstGeneratorBinary::generate` has the following implementation presented in metaFirstGenerator.cc file:

```
void
metaFirstGeneratorBinary::generate(const std::string& filename)
{
	GenerateKeyWordsContainer		g0(*this, filename);
```

```
        GenerateGeneratorH                g1(*this, filename);

        // ub:
        //GenerateKeyWordDefinitionH      g2(*this, filename);
        V2GenerateKeyWordDefinitionH      g2(g0);

        GenerateParserH                   g3(*this, filename);
        GenerateGeneratorCC               g4(*this, filename);

        //GenerateKeyWordDefinitionCC     g5(*this, filename);
        V2GenerateKeyWordDefinitionCC     g5(g0);

        GenerateMakeGeneratorsCC          g6(*this, filename);
        GenerateParserCC                  g7(*this, filename);
}
```

This way `metaFirstGeneratorBinary::generate` has eight phases from 0 to 7. The Parser generation is performed by:

```
GenerateParserCC                  g7(*this, filename);
```

This class is defined as follows:

```
class GenerateParserCC
{
         struct  metantmType
         {
           typedef std::set<int>   SymbolsContainer;

           cppcc::scr::tag::Long   nterm;
           SymbolsContainer        first;
           SymbolsContainer        follow;
         };

         struct  metagrmType {
             typedef std::vector<metantmType>            NonTerminalsContainer;
             typedef std::vector<cppcc::scr::tag::Long>  NonTerminalRuleContainer;

           NonTerminalsContainer     rulval;
           NonTerminalRuleContainer  ntmrul;
         };

        cppcc::gen::GeneratorBinary&      binary_;
        std::string                       filename_;
        metagrmType                       grammar_;
        int                                           CheckMode_;
        int                                           GlobalReturnCode_;
        cppcc::scb::SyntaxControlledBinary&           bin_;
        cppcc::scb::SyntaxControlledBinary::Helper    help_;
        cppcc::lex::Tokenizer&                        tok_;
        int                         Gffin_;
        std::vector<std::string>    predefinedIDs_;
        std::string                 IITT_;
```

```
void metaalone
       (std::ofstream&            output
       ,std::size_t               nnrule
       ,cppcc::scr::tag::Long     t
       ,cppcc::scr::tag::Long     d
       ,cppcc::scr::tag::Long     iiii
       ,cppcc::scr::tag::Long     iact
       );

void generateKWAlt
       (std::ofstream&          output
       ,cppcc::scr::tag::Long    ndd);

void dumpSymbols
       (std::ofstream&                           output
       ,std::size_t                              nnrule
       ,metantmType::SymbolsContainer&    symbols);

void dumpFirstFollow(std::ofstream&             output);

std::string          keyword(cppcc::scr::tag::Long   ndd);

void metagrfsimp
(std::ofstream&                           output
,std::size_t                              nnrule
,cppcc::scr::tag::Long                    nit
,cppcc::scr::tag::Long                    nid
,metantmType::SymbolsContainer& symbols);


void metagrfaltr
(std::ofstream&         output
,std::size_t            nnrule
,cppcc::scr::tag::Long nal
//,std::set<int>&              symbols
,metantmType::SymbolsContainer& symbols);


void metagetstr
//(output, ruleIndex, bin_.dynamic(e,i).t,0)
(std::ofstream&                    output
,std::size_t                       ruleIndex
,cppcc::scr::tag::Long     acid
,const char**                      t);


void metagetint
(std::ofstream&                    output
,std::size_t                       nnrule
,cppcc::scr::tag::Long     n
,cppcc::scr::tag::Long*    v);

void
metaactionsfind
(std::ofstream&                    output
```

```
,std::size_t                        nnrule
,cppcc::scr::tag::Long      actp
,cppcc::scr::tag::Long      scid
,cppcc::scr::tag::Long      vv
,int*                                   yy);

void metarfaddset
(std::ofstream&                     output
,std::size_t                               nnrule
,metantmType&                                     ntm
,metantmType::SymbolsContainer& symbols);

void metagrfiden
(std::ofstream&                     output
,std::size_t                               nnrule
,cppcc::scr::tag::Long              nnn
,metantmType::SymbolsContainer& symbols);

void
metagrfiter
(std::ofstream&                     output
,std::size_t                               nnrule
,cppcc::scr::tag::Long              nid
,metantmType::SymbolsContainer& symbols);

void metagggg
(std::ofstream&                     output
,std::size_t                        nnrule
,cppcc::scr::tag::Long      nitr
,cppcc::scb::SyntaxControlledBinary::edpirType*  ei);

void metarral
       (std::ofstream&                            output
       ,std::size_t                               nnrule
       ,cppcc::scr::tag::Long                     nid
       ,cppcc::scr::tag::Long                     nal
       ,int*                                             yy);

void metarfalid
       (std::ofstream&                            output
       ,std::size_t                               nnrule
       ,cppcc::scr::tag::Long          d
       ,metantmType::SymbolsContainer&    symbols);

void metarfsimp
       (std::ofstream&                            output
       ,std::size_t                               nnrule
       ,cppcc::scr::tag::Long                     t
       ,cppcc::scr::tag::Long             d
       ,metantmType::SymbolsContainer&    symbols);

void ruleError
         (std::ofstream&             output
         ,cppcc::scr::tag::Long   dNTerm
         ,const std::string&         t);
```

```
void ruleWarning
       (std::ofstream&            output
       ,cppcc::scr::tag::Long     nnrule
       ,const std::string&        t);

void metaidal
(std::ofstream&            output
,cppcc::scr::tag::Long     nnrule
,cppcc::scr::tag::Long     nid
,cppcc::scr::tag::Long  nal
,int*                           yy);

void metachrr
(std::ofstream&            output
,cppcc::scr::tag::Long     nnrule
,cppcc::scr::tag::Long     nid
,cppcc::scr::tag::Long  rig
,int*                           yy);

void checkGrammar(std::ofstream&   output);

void metarfiden
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nnn
,metantmType::SymbolsContainer&    symbols);

void metarfiter
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nid
,metantmType::SymbolsContainer&    symbols);

void metachnt
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nnn
,cppcc::scr::tag::Long*            rull);

void metarfaddnxt
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nnn
,metantmType::SymbolsContainer&    symbols);

void metarfmiss
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nid
,metantmType::SymbolsContainer&    symbols);

void metarfaltr
(std::ofstream&                         output
,std::size_t                            nnrule
,cppcc::scr::tag::Long                  nid
,metantmType::SymbolsContainer&    symbols);
```

```
void metarfaltern
(std::ofstream&                          output
,std::size_t                             nnrule
,cppcc::scr::tag::Long                   nid
,metantmType::SymbolsContainer&    symbols);

void metarfoll
(std::ofstream&                          output
,std::size_t                             nnrule
,metantmType::SymbolsContainer&    symbols);

void metarfir
(std::ofstream&                          output
,std::size_t                             nnrule
,metantmType::SymbolsContainer&    symbols);

void metarfcheck
(std::ofstream&                          output
,std::size_t                             nnrule
,metantmType::SymbolsContainer&    symbols);

void generateFirstFollow(std::ofstream&   output);

void metaactglob
(std::ofstream&      output
,std::size_t         ruleIndex);

void metaactnewf
(std::ofstream&      output
,std::size_t         ruleIndex);

void  metaangl
 (std::ofstream&           output
 ,cppcc::scr::tag::Long  dNTerm
 ,cppcc::scr::tag::Long  dRight);


int  metadynp
 (std::ofstream&           output
 ,cppcc::scr::tag::Long  dNTerm
 ,cppcc::scr::tag::Long  dRight);


int  metafixe
  (std::ofstream&             output
  ,cppcc::scr::tag::Long   dNTerm
  ,cppcc::scr::tag::Long   dRight);


//sss// int
void
metaalid
(std::ofstream&                  output
,std::size_t                     ruleIndex
```

```
,cppcc::scr::tag::Long      nalid);

void
metaalid_ccc
(std::ofstream&                   output
,std::size_t                      ruleIndex
,cppcc::scr::tag::Long      nalid);


void metaiter
//(output, ruleIndex, e, eru, i)
(std::ofstream&                   output
,std::size_t                      ruleIndex
,cppcc::scb::SyntaxControlledBinary::edpirType* e
,cppcc::scb::SyntaxControlledBinary::edpirType* eru
,cppcc::scr::tag::Long      i);


void metaaction
 (std::ofstream&                  output
,std::size_t                      ruleIndex
,cppcc::scr::tag::Long      acid
,cppcc::scr::tag::Long      vv);

void metamiss
(std::ofstream&      output
,std::size_t         ruleIndex
,int                        nal
,int                        iiii
,int                        yyyy
,int*                       fmis
,int                        iact
,int                        iend
);

void metagrfalid
(std::ofstream&                           output
,std::size_t                              nnrule
,cppcc::scr::tag::Long                    nalid
,metantmType::SymbolsContainer& symbols);

void metaaltr
(std::ofstream&             output
,std::size_t                ruleIndex
,cppcc::scr::tag::Long      nal
,cppcc::scr::tag::Long      iiii
,cppcc::scr::tag::Long      iact
);

void metaactions
(std::ofstream&                   output
,std::size_t                      ruleIndex
,cppcc::scr::tag::Long            actp
,cppcc::scr::tag::Long            acid
,cppcc::scr::tag::Long            vv
);
```

```
int metasimp
(std::ofstream&                    output
,std::size_t                       ruleIndex
,cppcc::scr::tag::Long             uit
,cppcc::scr::tag::Long             uid
,cppcc::scb::SyntaxControlledBinary::edpirType* e
,cppcc::scr::tag::Long*     iiii
,cppcc::scr::tag::Long             iact
);


void  metarigh
  (std::ofstream&                  output
  ,std::size_t                     ruleIndex
  ,cppcc::scr::tag::Long    dNTerm
  ,cppcc::scr::tag::Long    dRight);

void generateRuleBody
           (std::ofstream&         output
              ,std::size_t         ruleIndex
              ,cppcc::scr::tag::Long  dNTerm
        ,cppcc::scr::tag::Long  dRight);

std::string axiomRuleName();

void generateKW
(std::ofstream&           output
,cppcc::scr::tag::Long    ndd);

void generateKWelse
(std::ofstream&           output
,cppcc::scr::tag::Long    ndd);

void generateKWLoop
(std::ofstream&           output
,cppcc::scr::tag::Long    ndd);

void generate();

public:
       GenerateParserCC
              (cppcc::gen::GeneratorBinary&  binary
              ,const std::string&           filename)
           : binary_(binary)
           , filename_(filename)
           , grammar_()
           , CheckMode_(0)
           , GlobalReturnCode_(0)
           , bin_(binary_.generator_.parser_.binary_)
           , help_(cppcc::scb::SyntaxControlledBinary::Helper(bin_,KW_KEYWORDS_TOTAL))
           , tok_(binary_.generator_.parser_.tokenizer_)
           , Gffin_(0)
           , predefinedIDs_(tok_.grammarSymbols_.predefined_.size())
           , IITT_()
       {
```

```
		for (std::size_t i = 0, iz = predefinedIDs_.size();i < iz; i++) {
			predefinedIDs_[i] = prefix +
upper(tok_.grammarSymbols_.predefined_[i]);
		}
		IITT_ =
upper(tok_.grammarSymbols_.predefined_[cppcc::com::PLS_INTEGER_TOKEN]);
		generate();
	}
};
```

This way `GenerateParserCC` constructor calls `generate()` method that performs all parser generation tasks and defined as follows:

```
void
GenerateParserCC::generate()
{
// Extract axiom rule name:
		std::string	axn = axiomRuleName();

		std::string	ext = "Parser.cc";
		std::string	gn  = name(filename_);
		std::string	fe  = extention(filename_);
		std::string	fn  = gn+ext;
		std::ofstream output;

		bool		isFullRun = false;

		output.open(fn.c_str());
		if (!output) {
			std::string	syserr = cppcc::com::CPPCCException::systemError();
			CPPCC_THROW_EXCEPTION(
				<< "Can't open file for writing generated code:'"
				<< fn
				<< "' - Reason:'"
				<< syserr
				<< "'"
			)
		}

		bool isHelpDump = false;
		if (isHelpDump) {
			output << "---------------------------" << std::endl;
			help_.dump(output);
			output << "---------------------------" << std::endl;
		}
// Parser prolog generation:
		for (std::size_t i = 0; i < ParserCCCodeStartSize; i++) {
			output << line(ParserCCCodeStart[i], gn,axn) << std::endl;
		}

// Variable k is for Parsing Model Symbol identified by KW_RULE:
		cppcc::scb::SyntaxControlledBinary::Helper::kwn& k =
		     help_.k(KW_RULE);
		if (k) {
		    checkGrammar(output);
		    generateFirstFollow(output);
```

```
                CheckMode_ = 1;

            for (std::size_t i = 0, z = k.edps_.size(); i < z; i++) {
// Variable e is for Parsing Model Rule identified by k and index i:
                cppcc::scb::SyntaxControlledBinary::Helper::edp& e = k.e(i);

// Grammar rule for non-terminal rule is:
//  (rule ::= '(' nterm '::=' right ')'
//  )
// having only fixed part with mapping:
//   e.fixed(0) ==>> '('
//   e.fixed(1) ==>> nterm
//   e.fixed(2) ==>> '::='
//   e.fixed(3) ==>> right
//   e.fixed(4) ==>> ')'
                cppcc::scr::tag::TypeInstance&  kn  = e.fixed(1);
                      if (cppcc::com::PLS_IDENTIFIER != kn.t) {
                            tok_.errorMessage(errorNoIdentifierFound);
                                  CPPCC_THROW_EXCEPTION(
                                 << errorNoIdentifierFound
                                 << " rule:" << i << " k:" << kn.t << " n:" << kn.d
                                  )
                      }
// The code above checks that element 1 presented in variable kn has to be
// PLS_IDENTIFIER.

                      cppcc::scr::tag::TypeInstance&  knr = e.fixed(3);
                      if (KW_RIGHT != knr.t) {
                            tok_.errorMessage(errorNoRightEntryFound);
                                  CPPCC_THROW_EXCEPTION(
                                 << errorNoRightEntryFound
                                 << " rule:" << i << " k:" << knr.t << " n:" << knr.d
                                  )
                      }
// The code above checks that element 1 presented in variable knr has to be
// non-terminal with type field, Parsing Model Symbol ID, as KW_RIGHT.

// The code below comprises variable rule having KW_RULE as Parsing Model Symbol ID
// and Parsing Model Rule ID as a current index i.
                      cppcc::scr::tag::Long rule;
                      cppcc::scr::tag::setedflong(&rule,KW_RULE,i);

// Action section generation:
                        metaactglob(output, i);
                        metaactnewf(output, i);

// Parser method generation corresponding to rule name presented by help_.id(kn.d):
                      output << "void" << std::endl;
                      output
                            << gn << "Parser" << "::"
                            << help_.id(kn.d)
                            << "(cppcc::scr::tag::Long& tag)"
                            << std::endl;
// Now we can use rule tag presented in variable rule de-compiling rule definition on SGDL:
                      output << "/*" << std::endl;
```

```
                    binary_.generator_.binary_->scDecompile_(output, rule);
                    output << std::endl;
                    output << "*/" << std::endl;

// Now we can generate Parser method body as a compound statement:
                    output << "{" << std::endl;

// Pay attention to generateRuleBody arguments:   (output, i, kn.d, knr.d);
                    generateRuleBody(output, i, kn.d, knr.d);

                    output << "}" << std::endl;
                    output << std::endl;
                  }
                }

// Parser epilog generation:
                for (std::size_t i = 0; i < ParserCCCodeFinishSize; i++) {
                    output << line(ParserCCCodeFinish[i], gn, axn, fe) << std::endl;
                }
}
```

I put comments in bold font.
I will provide more comments for some implemented functions.

```
std::string
GenerateParserCC::axiomRuleName()
{
// Get  access to current Parsing Model Context:
                std::string gn("");
                cppcc::scr::SyntaxControlledRuntime::cnm&  currentContext =
                  binary_.generator_.parser_.runtime_.cnms_.dataByKey
                       (binary_.generator_.parser_.runtime_.cnmnumCurrent_);
// Get  access to current Parsing Model Symbol by id equal to KW_RULE:
                cppcc::scr::SyntaxControlledRuntime::kwn&  kwnInstance =
                  binary_.generator_.parser_.runtime_.
                    kwnLookup(KW_RULE);
// Get  access to current Parsing Model Rule with index 0,
// the first grammar rule that is considered as axiom:
                cppcc::scr::SyntaxControlledRuntime::edp& edpInstance =
                  kwnInstance.edps_.dataByIndex(0);
// Check for correct size:
                if (edpInstance.edpfix_.size() >= 2) {
                  cppcc::scr::tag::Long k;
                  cppcc::scr::tag::Long n;
// Unpack element [1] into variables k, n:
               cppcc::scr::tag::setlongedf(&edpInstance.edpfix_[1],k,n);
// Check the element type, k,  to be equal to cppcc::com::PLS_IDENTIFIER:
               if (cppcc::com::PLS_IDENTIFIER == k) {
// Create cppcc::scr::SyntaxControlledRuntime::nam instance by index n:
                 cppcc::scr::SyntaxControlledRuntime::nam namInstance=
                      currentContext.namn_.dataByIndex(n);
// Set the name having namkey_ member:
                  gn = namInstance.namkey_;
               }
                }
```

```
            return gn;
}


void
GenerateParserCC::generateRuleBody
       (std::ofstream&         output
       ,std::size_t            ruleIndex
       ,cppcc::scr::tag::Long  dNTerm
       ,cppcc::scr::tag::Long  dRight)
{
// Generate dynamic rule part variable declaration:
         if (metadynp(output, dNTerm, dRight)) {
           output << "  TagVector d;" << std::endl;
             metaangl(output, dNTerm, dRight);
         }

// Generate fixed rule part variable declaration:
         int fixedSize = metafixe(output, dNTerm, dRight);
         if (fixedSize > 0) {
           output << "  TagVector f(" << fixedSize << ");" << std::endl;
         }

         output << std::endl;

// Generate code for the right part of the grammar rule:
         metarigh(output, ruleIndex, dNTerm, dRight);

         output << std::endl;
}
```

C++ CCS generated code library

For the given SGDL specification having grammar name as metaFirst, these files are generated and compiled to `libgenerate.a` library:

- incloc/metaFirstGenerator.h and srcloc/metaFirstGenerator.cc
- incloc/metaFirstKeyWordDefinition.h and srcloc/metaFirstKeyWordDefinition.cc
- incloc/metaFirstParser.h and srcloc/metaFirstParser.cc
- srcloc/metaFirstMakeGenerators.cc

Use Case: XML compiler

Use Case: JSON compiler

Obfuscation

Obfuscation Use Case: Text File

Obfuscation Use Case: Account Number

References

[1] John R. Levine, Tony Mason, Doug Brown. *Lex & Yacc*. O'Reilly & Associates, 1992. ISBN: 1565920007.

[2] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers:Principles, Techniques, and Tools.* Addison-Wesley, 1986. ISBN: 0201100886.

[3] Aleksandr F Urakhchin. *Compiler compiler system with syntax-controlled runtime and binary application programming interfaces.* June 11 2013. US Patent 8,464, 232.

[4] http://www.w3.org/XML/, *Extensible Markup Language (XML).*

[5] http://json.org/, *Introducing JSON.*