Compiler Compiler System

Technical Specification

A complete reference for the Compiler Compiler System (CCS):
its parsing model, runtime and binary representations,
source grammar definition language, generator subsystem, and operational use.

A. F. Urakhchin

Inventor, US Patent 8,464,232

Draft 1.0 · May 2026

Status: working draft for review

Abstract

The Compiler Compiler System (CCS) is a compiler-construction tool that, given a context-free grammar specification, produces a complete compiler for the language that grammar describes. CCS departs from the YACC, GNU Bison, and ANTLR lineage in six specific ways: (i) it replaces the parse tree as the central intermediate representation with a structured first-class model of *Context*, *Name*, *Symbol*, and *Rule* and their relationships, instantiated as a `SyntaxControlledRuntime` object during parsing; (ii) it strictly separates syntax processing from semantic processing, with semantics consuming a grammar-defined application programming interface rather than being embedded in grammar actions; (iii) it defines syntax processing as a two-phase, formally bidirectional procedure between the runtime and a serialized form, the `SyntaxControlledBinary`, so that runtime, binary, and source text are interconvertible; (iv) it makes formal de-compilation back to source text a first-class operation rather than a debugging aid; (v) it derives the runtime and binary application programming interfaces from the grammar itself rather than from hand-written interface definitions; and (vi) the source grammar definition language SGDL is self-bootstrapped — its own grammar is defined in itself, and the compiler compiler executable can regenerate itself from that meta-grammar. The runtime and binary forms together serve as a multi-platform program-to-program data exchange substrate, supporting binary-to-binary transformations for obfuscation, access control, content management, and binary file format processing.

This specification describes the Compiler Compiler System as embodied by the C++ reference implementation that builds on top of US Patent 8,464,232. It is intended to be sufficient for a compilers engineer to use, extend, or reimplement the system. The document is organized in five parts: an overview and conceptual model (Part I); a description of each of the five subsystems — management, runtime, binary, generator, and parsing procedure (Part II); operational use of the `cppcc` executable and worked targeting examples (Part III); transformation algorithms over the syntax-controlled binary (Part IV); and a set of reference appendices (Part V).

Notation and Conventions

This document uses the following conventions consistently throughout.

Typography

- `Monospaced text` denotes the names of classes, methods, files, and command-line tokens as they appear in the C++ reference implementation. For example, `SyntaxControlledRuntime`, `Tag`, `cppcc`, and `meta.sgr` are all rendered monospaced.
- *Italic text* marks a term at its point of definition or emphasizes a word. The four entities of the parsing schema — *Context*, *Name*, *Symbol*, *Rule* — appear in italic in conceptual discussion and in monospaced when referring to the corresponding C++ classes.
- **Bold text** marks important properties, requirements, or warnings.

Cross-References and Citations

Citations to the source materials are inline, in square brackets, and abbreviated as follows. [*claim N*] refers to claim *N* of US Patent 8,464,232. [*FIG N*] refers to figure *N* of the same patent. [*whitepaper §X.Y*] refers

to section *X.Y* of the 2015 Compiler Compiler System whitepaper. Cross-references within this specification use the form *Chapter N*, *Section N.M*, or *Figure N.M* as appropriate.

Voice

Specification voice. The text uses the present indicative throughout — "the runtime maintains a collection of contexts" rather than "you can think of the runtime as maintaining a collection of contexts." First and second person are avoided. Hedging phrases ("typically," "in general," "may") appear only where genuine optionality or implementation freedom exists.

Numbering

Chapters are numbered with arabic numerals (Chapter 1, Chapter 2, …). Sections are numbered hierarchically (1.1, 1.2, 2.3, …). Figures are numbered by chapter (Figure 1.1, Figure 2.1, …). Phase labels in figures and prose follow the patent numbering when reproducing patent figures (Phase 2.1a, Phase 2.2c, etc.) and use chapter-prefixed numbering when introducing new procedural decompositions.

Chapter 1 — Introduction

1.1 Purpose and Audience

This document is the technical specification of the Compiler Compiler System (CCS). It is intended for engineers and researchers working with compiler construction tools who need a precise, complete description of the CCS architecture: its parsing model, the layout of its in-memory and on-disk representations, the application programming interfaces these representations expose, the source grammar definition language SGDL, and the operational behavior of the `cppcc` executable.

The audience is the specification reader. The text assumes familiarity with general compiler-construction concepts at the level of the canonical reference texts, with C++ at the level required to read class declarations and header files, and with the BNF notation used to define context-free grammars. No familiarity with the patent literature on CCS or with prior CCS documentation is assumed.

This is not a tutorial. Worked examples appear in Chapter 10, but they are organized as illustrations of mechanisms defined elsewhere in the specification rather than as a hand-held introduction to the system. A separate user guide is anticipated as a future companion document; readers seeking step-by-step instruction are advised to wait for that document or to read this specification in conjunction with the C++ reference implementation source code.

1.2 Relationship to the Source Materials

Two prior documents describe CCS: US Patent 8,464,232 (*Compiler Compiler System with Syntax-Controlled Runtime and Binary Application Programming Interfaces*, granted 11 June 2013) and the Compiler Compiler System whitepaper of June 2015 (the *whitepaper*, 173 pages). The present specification supersedes neither. It draws on both, restructures their material into a form suitable for engineering reference, and resolves several presentational issues.

Specifically, this specification:

- Adopts the abbreviation SGDL (Source Grammar Definition Language) from the whitepaper rather than CCSGDL from the patent. The two terms denote the same language; SGDL is shorter and matches the working code base. The patent’s term is mentioned at this single point and not used again.
- Uses the C++ reference implementation class names (`SyntaxControlledRuntime`, `SyntaxControlledBinary`, `Tag`, `TypeInstance`, and the rest) rather than the patent’s wordier "compiler compiler runtime" and "compiler compiler binary."
- Introduces the term *Parsing Model* (Chapter 2) for the high-level architectural abstraction that the patent describes informally and the whitepaper does not name. The Parsing Model is the central organizing concept of this specification.
- Provides material missing or stubbed in the whitepaper, including completed worked examples for XML and JSON compilers (Chapter 10), a transformation chapter unifying the patent’s scattered transformation use cases (Chapter 12), and a set of reference appendices including a complete glossary, class reference, and patent map (Appendices A through F).

Where this specification states properties of CCS that originate in the patent or whitepaper, the inline citations *[claim N]*, *[FIG N]*, and *[whitepaper §X.Y]* identify the source. Statements without citation are either definitional, derived in this specification, or reflect the C++ reference implementation directly.

1.3 What CCS Is and What It Is Not

CCS is a compiler-compiler. Given a grammar specification written in SGDL, it generates a complete compiler — parser, generator, and runtime — for the language defined by that grammar. The generated compiler can be built, distributed, and used to translate programs written in the source language into whatever target form the grammar's author has implemented.

Figure 1.1 shows the structure of a prior-art compiler-compiler system of the YACC or GNU Bison family, reproduced from *[FIG 1]* for orientation. An input file describing the grammar feeds a parser generator; an input file describing the lexical structure feeds a tokenizer generator; the outputs of both, together with hand-written compiler source files, feed a compiler that produces an executable program. The developer's effort is concentrated in the compiler source files: every grammar rule whose recognition triggers semantic processing must have an action attached to it, and the semantic processing is interleaved with parse-tree construction.

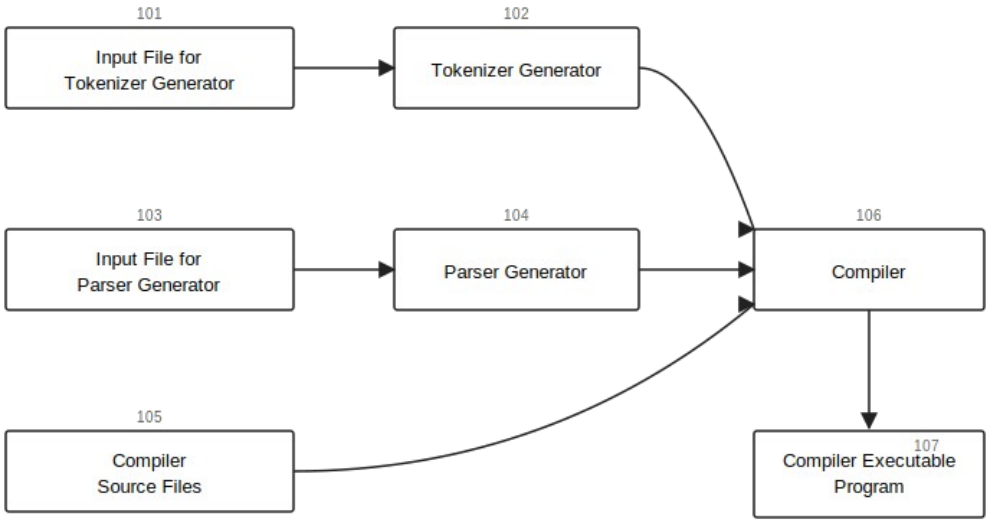

Figure 1.1 — Prior-art compiler-compiler architecture (after [FIG 1]). Tokenizer and parser generators consume input files describing lexical and grammatical structure; the compiler combines their outputs with hand-written source files to produce an executable.

CCS preserves the outermost flow shown in Figure 1.1 — a grammar specification feeds a generator; the generator's output, combined with implementation code, produces an executable — but it changes nearly everything internal to that flow. There is no parse tree. Grammar actions are not embedded in grammar rules; semantics is a downstream consumer of a grammar-defined application programming interface. The intermediate representation maintained by the parser is not a tree but a structured collection of named entities; that collection has a serialized binary form that is bidirectionally convertible with both the in-memory representation and the original source text. Chapter 2 develops these properties in full.

CCS is not a verified compiler. Its bidirectionality and round-trip properties are demonstrated empirically by self-test sequences (Chapter 8); they are not mechanically proved. CCS is also not a multi-target back-end framework in the sense of LLVM or MLIR: it produces source-level intermediate code in C++ which is then compiled by the host C++ toolchain. Its target portability comes from the host C++ toolchain, not from a CCS-internal back-end abstraction.

CCS is also not a tree-sitter-style incremental parser, a Protocol Buffers-style data interchange format, or a metaprogramming system embedded in a host language. It is closest in spirit to YACC and ANTLR — it generates compilers from grammar specifications — but it reformulates that activity around a very different intermediate representation. The detailed positioning against contemporary systems is left to a separate landscape document; the present specification's job is to define what CCS *is*, precisely.

1.4 Document Map

The remainder of this specification is organized as follows.

Part I — Overview and Conceptual Model. Chapter 2 develops the CCS Parsing Model in self-contained form: the three forms of the parsing artifact (grammar, runtime, binary), their relationships, the four-entity schema (Context, Name, Symbol, Rule) that the runtime and binary share, and the bootstrap loop by which CCS defines itself. Chapter 3 specifies SGDL as a language: its lexical structure, its grammar, the structure of its rule definitions and grammar actions, the meta-grammar `meta.sgr` that defines SGDL in itself, and the precise class of grammars CCS supports.

Part II — The Five Subsystems. Chapter 4 specifies the management subsystem (the `Logger`, `Shell`, `Compiler`, `Tokenizer`, and `Language` classes and their interfaces). Chapter 5 specifies the runtime (`SyntaxControlledRuntime` and its inner classes). Chapter 6 specifies the binary (`SyntaxControlledBinary`, its layout in memory, and the auxiliary `BinaryHelper` indexing classes). Chapter 7 specifies the generator subsystem and its style hierarchy. Chapter 8 specifies the compilation, de-compilation, and round-trip procedures, including the self-test sequences.

Part III — Use and Operation. Chapter 9 specifies the `cppcc` executable: its command-line interface, its build artifacts, and the foundation library architecture. Chapter 10 walks through the procedure for targeting CCS at a new language, with worked examples for XML and JSON. Chapter 11 specifies multiplatform operation and the use of the syntax-controlled binary as an inter-program data exchange substrate.

Part IV — Transformations. Chapter 12 specifies the binary-to-binary transformation algorithms enumerated in the patent (*[claim 13, claim 14, claim 15]*): obfuscation, access control, content management, and binary file format processing.

Part V — Appendices. Appendix A maps each section of this specification to specific patent claims and figures. Appendix B is a class reference. Appendices C and D give complete BNF for SGDL and for the syntax-controlled binary respectively. Appendix E reproduces the self-test command sequences. Appendix F is a glossary; Appendix G is a bibliography. Appendix H documents the differences between this specification and the 2015 whitepaper.

Chapter 2 — The CCS Model

This chapter is the conceptual entry point of the specification. It is the only chapter that spends pages on motivation rather than mechanism. The reader who absorbs this chapter has the model in hand; subsequent chapters fill in the precise definitions, layouts, and procedures.

2.1 The Compiler-Tax Problem

Every new programming abstraction, domain-specific language, file format, or interchange protocol bears an engineering cost: a parser must be written, a tokenizer must be written, a representation of the parsed program must be designed, code that traverses that representation to perform semantic analysis must be written, and code that emits some target form (executable, transformed source, reformatted output, transmitted message) must be written. Compiler-construction tools such as YACC, GNU Bison, and ANTLR address part of this cost — the parser and tokenizer — but they leave several costs in place. The intermediate representation is the developer's responsibility to design and maintain; the semantic actions are interleaved with grammar rules so that altering one affects the other; and there is no built-in story for serializing the parsed program, transmitting it across a process boundary, or reading it back into another tool. The cumulative effect is an engineering burden — the *compiler tax* — that is paid in full for every new language and that scales poorly as languages proliferate.

CCS addresses the compiler tax by reorganizing the activity of compilation around a representation that is, by construction: structured (so that the developer does not design it), grammar-defined (so that the application programming interface is automatic), serializable (so that the in-memory and on-disk forms are interconvertible without extra machinery), reversible (so that any of the three forms — source text, in-memory representation, on-disk binary — can be obtained from any other), and transformation-friendly (so that operations such as obfuscation, access control, content rewriting, and inter-program communication are first-class binary-to-binary operations).

2.2 Six Departures from Prior Art

CCS departs from the YACC/Bison/ANTLR family in six specific ways. These six points, taken together, constitute the technical content of the system. The remainder of this specification expands each of them into a full mechanism.

1. **No parse tree.** The intermediate representation maintained during parsing is not a parse tree. It is a structured collection of named entities — instances of the four classes `Context`, `Name`, `Symbol`, `Rule` — and their relationships, instantiated in a `SyntaxControlledRuntime` object. The parser populates this object as it reads the source.
2. **Total separation of syntax and semantics.** Grammar rules in SGDL contain no semantic actions in the YACC sense. Semantic processing is a downstream phase that consumes the runtime — or the binary — through a grammar-defined application programming interface. Altering semantics does not require recompiling the grammar; altering the grammar does not invalidate the semantic code beyond the API surface that changed.
3. **Two-phase, formally bidirectional syntax processing.** Compilation produces a `SyntaxControlledRuntime` (Phase 2.1a). The runtime serializes to a

`SyntaxControlledBinary` (Phase 2.2a) — a flat tagged vector that allocates all parsed data into a single read-only memory region. The binary deserializes back to a runtime (Phase 2.2b), and either form de-compiles back to source text (Phase 2.1b from the runtime, Phase 2.2c from the binary).

4. **Formal de-compilation.** De-compilation from runtime or binary back to source text is not a debugging convenience. It is a defined procedure that is provided automatically by the framework and is part of the standard CCS executable's interface. The de-compiled source is identical to the original modulo indentation rules.
5. **Grammar-derived application programming interfaces.** The runtime and binary expose application programming interfaces whose entry points are derived from the grammar specification — not hand-written. Every grammar non-terminal yields an automatically generated set of accessor methods on both `SyntaxControlledRuntime` and `SyntaxControlledBinary`. The grammar *is* the interface.
6. **Self-bootstrapping.** SGDL is defined in itself. The file `meta.sgr` is an SGDL specification of SGDL's own grammar. The CCS executable, given `meta.sgr` as input, regenerates its own parser, generator, and key-word definitions. The bootstrap loop is the central operational property of the system and is documented in detail in Section 2.5.

2.3 The Parsing Model

The *Parsing Model* is the high-level architectural abstraction of CCS. It comprises three entities and the morphisms (procedural transformations) between them. The three entities are:

Grammar. A specification, written in SGDL, of the source language whose programs CCS is to compile. In the case where CCS is bootstrapping itself, the grammar is the meta-grammar `meta.sgr` — an SGDL specification of SGDL itself.

SyntaxControlledRuntime. The in-memory representation of a successfully parsed program. The runtime is maintained by the parser as parsing proceeds: each grammar rule whose right-hand side is recognized causes the corresponding `Context`, `Name`, `Symbol`, and `Rule` instances to be created or updated within the runtime. Section 2.4 specifies the four-entity schema in detail.

SyntaxControlledBinary. The serialized representation of a successfully parsed program. The binary is generated as an independent phase, after compilation has succeeded — that is, after the front end has produced a runtime that is consistent with the grammar. The binary is a flat tagged `vector<Tag>` that holds the same parsed information as the runtime in a layout suitable for memory-mapped read-only access, file storage, and inter-process transmission.

Figure 2.1 shows the morphisms among these three entities. The five labeled phases — Phase 2.1a, 2.1b, 2.2a, 2.2b, and 2.2c — together constitute the bidirectional behavior of the system. Phase 2.1a (compile) is the only phase that processes source text; Phase 2.1b and Phase 2.2c are the two de-compilation phases, returning to source text from the runtime and binary respectively. Phases 2.2a and 2.2b are pure conversion between the two interconvertible in-memory and on-disk forms.

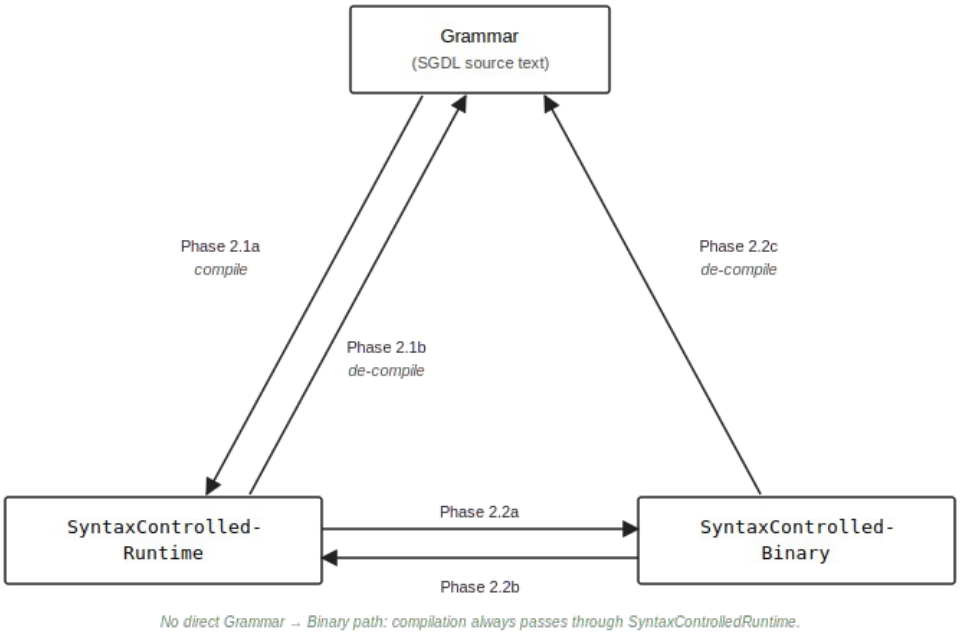

Figure 2.1 — The CCS Parsing Model. Three entities — grammar, runtime, binary — connected by five named morphisms. Compilation always passes through the runtime: there is no direct grammar-to-binary phase. De-compilation may be performed from the runtime (Phase 2.1b) or from the binary (Phase 2.2c).

A property worth noticing on first encounter: there is no direct **grammar → binary** arrow. Compilation always passes through the runtime. The binary is a serialized form of the runtime, and any process that produces a binary must first produce a runtime. This is not an arbitrary architectural choice; it reflects the responsibilities assigned to each of the three forms. The runtime is the form in which the parser *builds* the parsing artifact, with full freedom to insert, modify, and rearrange entries; the binary is a finalized, read-only summary suitable for transport. Allowing direct grammar-to-binary compilation would require either a parser whose output were the read-only binary (impractical, since parsing is incremental and exploratory) or a separate code path that duplicated the parser's logic. CCS chooses neither.

2.4 The Four-Entity Schema

The runtime and binary forms share a common schema: the four entities *Context*, *Name*, *Symbol*, and *Rule*, and their relationships. Both forms hold the same information, organized in the same way; they differ only in the physical representation (object graph in the runtime, flat tagged vector in the binary). The schema is most precisely specified in Chapters 5 and 6, where each entity is given its full class definition. Here, the conceptual content suffices.

Context — the top-level scope of a parsing artifact. A program parsed by CCS is decomposed into one or more contexts. Each context owns its own collections of names, symbols, and rules. Most simple grammars produce a single context per program; nested or modular grammars may produce many. Contexts are identified by a numeric `contextID` of type `Tag`.

Name — an identifier appearing in the source program: a variable name, a function name, a structure tag, a label. Each name lives in some context and is reachable by the textual identifier the source used as well as by a sequential index assigned during parsing. Both lookups are constant-time.

Symbol — a non-terminal grammar symbol — that is, an entry in the grammar — together with the parsing record of all instances in which the parser recognized it. A symbol owns a collection of `Rule` instances, one per recognition occurrence. Symbols are identified by `symbolID`.

Rule — a single recognition of a non-terminal symbol — a single point in the source where the parser matched the symbol's right-hand side. A rule has fixed and dynamic parts; the fixed part holds the values produced by terminals and embedded actions, while the dynamic part holds references to other rules invoked during this recognition. Rules are identified by `ruleInstanceID`.

The relationships among these four entities are: each runtime contains a collection of *contexts*; each context contains a collection of *names* (indexed both alphabetically and sequentially) and a collection of *symbols*; each symbol contains a collection of *rules*. The binary form preserves these collections in a flat memory layout that supports the same access patterns as the runtime form, with the same asymptotic costs. Chapter 5 specifies the runtime layout; Chapter 6 specifies the binary layout.

A reader familiar with parse-tree-based compiler-compilers will note an important absence: there is no node-and-edge tree structure here. The four-entity schema is not a tree. A `Rule` instance does refer to other `Rule` instances through its dynamic part — and so the rule graph is, technically, walkable as a tree from the axiom rule downward — but the schema's primary access patterns are not tree-walks. The runtime's API exposes lookups by name within a context, lookups by symbol within a context, lookups by rule instance within a symbol, and so on. These are constant-time index operations, not tree traversals. Semantic processing is therefore typically structured as a series of indexed accesses to the runtime, rather than as a recursive tree walk.

2.5 The Bootstrap Loop and Self-Definition

SGDL is defined in itself. The file `meta.sgr` is an SGDL specification of the SGDL grammar — an instance of SGDL whose object of description *is* SGDL. The full text of this file is reproduced and analyzed in Section 3.6; for present purposes the file's existence and the property it embodies are what matter. CCS is constructed so that the executable can consume `meta.sgr` as input and regenerate its own internals from it. This is the bootstrap loop, shown in Figure 2.2.

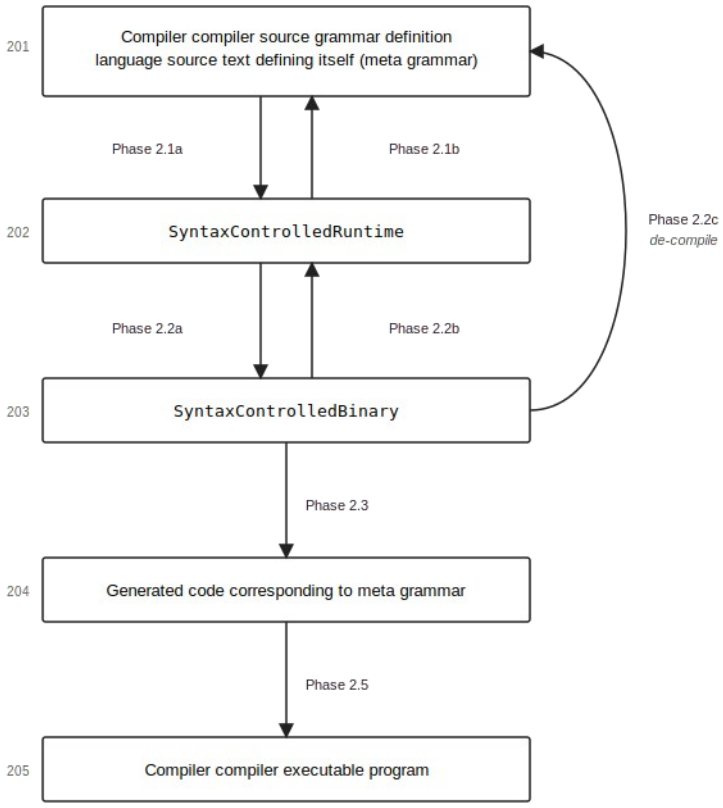

Figure 2.2 — The bootstrap loop (after [FIG 2]). The CCS executable reads the meta-grammar (201) and produces a runtime (202), then a binary (203), then generated code (204), then a new executable (205). The new executable is identical in capability to the one that produced it. Phase 2.4 (developer-supplied semantic implementation, not shown) refines the auto-generated default in (204) before (205) is built.

Each box in Figure 2.2 is a stable artifact; each labeled phase is a procedural transformation between artifacts. The CCS executable program shown at the bottom (205) is the same artifact, in capability, as the executable that initiated the cycle. Self-definition therefore means: the executable can rebuild itself from a single SGDL file. There is no separate boot grammar, no special-cased C++ code that handles SGDL parsing as opposed to user-grammar parsing. The CCS executable parses `meta.sgr` using the same machinery it would apply to any other SGDL specification.

It is worth being precise about what self-definition does **not** mean. CCS does not generate its own C++ code without human intervention. Phase 2.4 — not labeled as an inter-artifact arrow in Figure 2.2 because it is not a transformation between artifacts — is the step in which the developer fills in semantic implementations against the auto-generated parser and generator stubs produced by Phase 2.3. The default version of the generated semantics is empty; the developer's contribution is to provide the semantic processing that the application requires. For the meta-grammar bootstrap specifically, the developer's contribution at Phase 2.4 is the implementation of SGDL's own meaning — what it *does* to recognize a grammar and produce a runtime. The CCS distribution ships with this implementation; the developer of a new target language does not need to repeat it.

The bootstrap loop has a useful operational consequence beyond elegance. Because the loop closes — because the executable produced at (205) can regenerate itself from (201) — the system can be tested for round-trip fidelity. Take any grammar (not necessarily the meta-grammar): compile it through Phase 2.1a to a runtime, de-compile via Phase 2.1b back to text, and compare the de-compiled text against the original source modulo indentation. The expected result is identity. Take any binary: de-compile via Phase 2.2c back to text, and compare. Again the expected result is identity. The patent and whitepaper document the empirical demonstration of this property in self-test sequences (s.1)–(s.4) and (b.1)–(b.6); Chapter 8 of this specification reproduces those sequences and specifies their expected outputs precisely.

The bidirectionality property is empirically demonstrated, not formally proved. This specification does not claim that round-trip is bijective for every grammar that SGDL can express; it claims that the CCS implementation maintains round-trip identity for all grammars that have been tested against it, including `meta.sgr` itself. The precise grammar class for which round-trip is bijective is not characterized in this specification or in the source materials it draws upon. Section 3.7 returns to this question.

Chapter 3 — SGDL: Source Grammar Definition Language

SGDL — the Source Grammar Definition Language — is the input language of the CCS executable. An SGDL file describes a context-free grammar in a notation that extends Backus–Naur Form (BNF) with three iteration constructs, two optionality constructs, and a macro substitution mechanism for embedded actions. SGDL is itself defined in SGDL: the file `meta.sgr` is an SGDL specification of SGDL, and the CCS executable parses `meta.sgr` using the same machinery it would apply to any user-supplied grammar. This chapter specifies SGDL as a language: its lexical structure (Section 3.1), its grammar (Section 3.2), the section structure of an SGDL file (Section 3.3), the syntactic constructs available within rule definitions (Section 3.4), the action macro mechanism (Section 3.5), the meta-grammar (Section 3.6), and the precise class of grammars CCS supports (Section 3.7).

3.1 Lexical Structure

SGDL source text is a sequence of tokens separated by whitespace. The lexer recognizes five categories of token: identifiers, reserved words, integer literals, string literals, and grammar terminals. Whitespace and comments separate tokens but are otherwise insignificant. The CCS executable's tokenizer is the `Tokenizer` class specified in Chapter 4; the present section describes the lexical content the tokenizer recognizes when processing SGDL source.

3.1.1 Identifiers

An identifier in SGDL is a sequence of characters beginning with a letter, followed by zero or more letters, digits, or underscores. Identifiers are used to name grammar non-terminals and to refer to predefined token classes. Identifiers are case-sensitive. Examples:

```
grammar
grammarNameDef
rule
identAlt
iterItem
```

An identifier appearing on the left-hand side of a rule definition introduces a non-terminal of the grammar being defined. An identifier appearing on the right-hand side either references another non-terminal (defined elsewhere in the same SGDL file) or names one of the predefined tokens documented in Section 3.1.4.

3.1.2 Reserved Words

SGDL defines a small set of reserved words. These names are recognized by the tokenizer as distinct token classes and may not be used as identifiers in user grammars. The reserved words include the names of the predefined token classes — `integerToken`, `stringToken`, `termToken`, `identifier` — and the macro action delimiters `METAACTBEG` and `METAACTEND` that appear in the meta-grammar. The complete list is given in Appendix C.

3.1.3 Integer and String Literals

An *integer literal* is a sequence of one or more decimal digits. Integer literals are recognized by the predefined token class `integerToken`.

A *string literal* is a sequence of arbitrary characters enclosed in double-quote characters. The string literal `"hello"` represents the seven-character sequence `"hello"` (the surrounding quotes are part of the literal as recognized by the tokenizer; the string's value is the five-character content `hello`). String literals are recognized by the predefined token class `stringToken`. There is no escape mechanism specified for embedding a double-quote character within a string literal in the SGDL itself; grammar designers requiring such literals define the escape mechanism in their target language's grammar.

3.1.4 Grammar Terminals

A *grammar terminal* is a string of one or more characters enclosed in single quotes. Examples include `'('`, `'::='`, and `'|'`. Grammar terminals are recognized by the predefined token class `termToken`. The single-quote-enclosed form distinguishes a literal piece of the source language being defined from an SGDL identifier or reserved word that names a non-terminal of that source language.

Four predefined token classes are available in every SGDL grammar: `identifier` (a sequence of letters, digits, and underscores beginning with a letter), `integerToken` (a sequence of decimal digits), `stringToken` (a double-quote-delimited string), and `termToken` (a single-quote-delimited terminal). A grammar that requires additional token classes — floating-point literals, hexadecimal literals, character classes for XML, and so on — extends this set through a `Language` subclass registered with the tokenizer at parser construction (Chapter 4).

3.1.5 Whitespace and Comments

Whitespace — space, tab, newline, and carriage return — separates tokens and is otherwise insignificant. SGDL does not specify a comment syntax: SGDL files in the patent and reference implementation contain no comments. Designers wishing to annotate a grammar may use whitespace and indentation freely, but no character sequence is recognized as a comment that the tokenizer would skip.

A consequence of this design choice is that round-trip de-compilation (Phase 2.1b and Phase 2.2c) preserves exactly the grammar content that the parser recognized. There is no comment information that could be lost in the round-trip. The de-compiled output differs from the original source only in indentation, as specified in Chapter 8.

3.2 The Grammar of SGDL

SGDL is itself a context-free language defined in itself. The complete and authoritative grammar of SGDL is the file `meta.sgr`, reproduced verbatim below. This is not an abstracted BNF restatement of SGDL: it is the actual grammar file that the CCS executable consumes when bootstrapping itself, and every property of SGDL stated elsewhere in this chapter follows from what is written 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 '>'
  )
)
```

Several observations about this listing.

Self-definition. The file is an SGDL grammar that defines SGDL. Its grammar name is `meta` (the first identifier inside the outer parenthesis). Its first rule defines the non-terminal `grammar` as the file structure

```
'(' grammarNameDef { rule } ')'
```

— that is, an opening parenthesis, an identifier naming the grammar, a sequence of zero or more rule definitions, and a closing parenthesis. The file `meta.sgr` itself satisfies this structure: it opens with `(meta`, contains the rules listed above, and closes with `)`. The grammar describes itself.

Reserved-word and terminal tokens. The grammar uses the following terminals, each quoted with single quotes: `'('`, `')'`, `'::='`, `'|'`, `'['`, `']'`, `'{'`, `'}'`, `'<'`, `'>'`, `'='`. Each is a single-character token except for `'::='`, which is three characters. The tokenizer recognizes these by longest match: `'::='` is preferred over the prefix `':'` when both could match.

Action macros on the outermost rule only. The first rule, `grammar`, is bracketed by two action macros — `METAACTBEG();` and `METAACTEND();`, each written using the literal action form `0 ="..."=`. These actions emit code at the start and end of the generated parsing routine for an SGDL grammar. No other rule in `meta.sgr` carries actions: every other rule defines its right-hand side in pure SGDL with no embedded code. This is intentional and is discussed further in Section 3.6.

Whitespace is significant only as a token separator. The mixture of spaces, tabs, and indentation in the listing above is the form in which `meta.sgr` is maintained in the reference implementation. Any whitespace pattern that separates tokens correctly produces an equivalent grammar; the de-compilation phases (Phase 2.1b and Phase 2.2c) emit a normalized indentation that may differ from the original source's, as discussed in Section 3.7.3.

3.3 Structure of an SGDL File

An SGDL file has two structural elements: a grammar name and a sequence of rule definitions, both enclosed in a single outer pair of parentheses. This is the structure that the `grammar` rule of Section 3.2 specifies, and it is the only structure SGDL recognizes. Unlike YACC, GNU Bison, and ANTLR — whose input files are organized into multiple distinct sections (declarations, rules, user code) separated by markers such as `%%` — an SGDL file is one self-contained parenthesized list. Implementation-language code, when present, appears within rules as action macros (Section 3.5), not as a separate section.

3.3.1 Grammar Name and Generated Artifact Naming

The grammar name is the first identifier appearing inside the outer parenthesis, and it is also the grammar's axiom — the non-terminal whose definition begins the rule sequence and which the parser attempts to recognize as it consumes the input. By convention the grammar name matches the file basename: the file `meta.sgr` has grammar name `meta`, and the file `xsd.sgr` has grammar name `xsd`. The CCS executable enforces this convention only loosely: the grammar name in the file determines the prefix of generated artifacts; the file name itself is informational. The generated artifacts for a grammar named `xyz` are:

- `xyzParser.h` and `xyzParser.cc` — parser source
- `xyzGenerator.h` and `xyzGenerator.cc` — generator source
- `xyzKeyWordDefinition.h` and `xyzKeyWordDefinition.cc` — keyword definitions
- `xyzMakeGenerators.cc` — generator instantiation stub
- `xyz.sgr.fgr` — `SyntaxControlledBinary` (when binary output is enabled)
- `xyz.sgr.urt` — de-compiled grammar text from runtime (when de-compilation is enabled)

3.3.2 Rule Definitions

The rule definitions section is the body of the SGDL file. It is a sequence of zero or more rule definitions, each of which has the form

```
(nterm ::= right)
```

where `nterm` is the non-terminal being defined and `right` is its right-hand side: a sequence of zero or more elements drawn from the constructs documented in Section 3.4. The non-terminal named first in the rule sequence is also the grammar's axiom; subsequent rules may appear in any order, and forward references between rules are permitted.

Lexical extensions to the four predefined token classes — additional token types that a target language requires beyond `identifier`, `integerToken`, `stringToken`, and `termToken` — are not specified within the SGDL file. They are provided by `Language` subclasses written in C++ and registered with the tokenizer at parser construction. Chapter 4 specifies the `Language` class hierarchy; Chapter 10 walks through the procedure for adding a new lexical extension when targeting CCS at a language whose lexical structure exceeds the four predefined token classes.

3.4 Rule Syntax

A rule's right-hand side is a sequence of *elements*. SGDL provides five kinds of element: a plain non-terminal or terminal reference, an alternative group, an optional group, an iteration group, and an embedded action. The first three are extensions of standard BNF; iteration is the SGDL analogue of the Kleene closure, and the optional groups together cover the two distinct senses of "optional" that arise in language design. This section specifies each construct, gives its desugaring into pure BNF where applicable, and notes the situations in which each is appropriate.

3.4.1 Plain References

A non-terminal or terminal name appearing on the right-hand side names that symbol directly. The rule

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

defines `rule` as the sequence of: the terminal `'('`, the non-terminal `nterm`, the terminal `'::='`, the non-terminal `right`, and the terminal `')'`. Plain references desugar to themselves; no further analysis is required.

3.4.2 Alternatives

A vertical bar `|` between symbols on the right-hand side denotes alternation: the parser may match either side of the bar to satisfy the rule. The construct is associative and left-binding. The rule

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

is equivalent to the four rules

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

Alternation may be combined with parentheses to localize the alternative within a larger sequence. The construct `"(" identAlt ")"` in the SGDL grammar permits writing

```
(combinedExample ::= prefix (A | B | C) suffix)
```

which means: a `prefix` followed by exactly one of `A`, `B`, or `C`, followed by a `suffix`.

3.4.3 Iteration

An iteration group, written with curly braces, denotes Kleene closure: zero or more occurrences of the enclosed content. The rule

```
(iterationExample ::= { element })
```

is equivalent to the recursive pair

```
(iterationExample ::= element iterationExample)
(iterationExample ::=)
```

That is, `iterationExample` matches either an `element` followed by another `iterationExample`, or the empty sequence. This is the standard right-recursive expansion of Kleene closure.

Iteration groups support internal alternatives. An iteration whose contents are a sequence of items separated by `|` permits each iteration to choose among the alternatives:

```
(altIterationExample ::= { A | B | C })
```

matches a sequence of zero or more elements, each of which is independently chosen from `A`, `B`, or `C`.

3.4.4 Optional Groups: Square Brackets

An element enclosed in square brackets `[ ]` may occur zero or one times. The rule

```
(optionalExample ::= A [ B ])
```

is equivalent to the pair

```
(optionalExample ::= A B)
(optionalExample ::= A)
```

Square brackets are intended for cases where the alternative-omission idiom is more natural than an explicit alternation. They desugar to the same form as the equivalent alternation, and the choice between `A [ B ]` and `(A B | A)` is purely stylistic.

3.4.5 Optional Groups: Angle Brackets

A non-terminal enclosed in angle brackets `< >` represents a constrained iteration: the non-terminal may occur zero or one times **within an iteration group**. Outside an iteration group, the angle-bracket form is not used; square brackets serve that purpose. The construct exists because iteration groups can otherwise force a non-terminal to be repeatable, and certain grammar designs require a non-repeatable item to be intermixed with repeatable items inside the same iteration. The rule

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

is equivalent to

```
(anotherIterationExample ::= elem anotherIterationExample)
(anotherIterationExample ::=)
```

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

with the constraint that `X` and `Z` are permitted to appear at most once across the entire iteration. The constraint is enforced at parse time, not in the desugared rule structure; the angle bracket is therefore a parser-level construct rather than a pure BNF transformation.

3.4.6 Element Composition

Elements compose by juxtaposition. A rule's right-hand side is a sequence of elements written next to one another with whitespace between them; the parser matches each element in turn against the input. There is no explicit sequencing operator; juxtaposition is the sequence. The rule

```
(complexExample ::= prefix [ optionalA ] { iterB } (X | Y) suffix)
```

matches: `prefix`, then optionally `optionalA`, then zero or more `iterB`, then exactly one of `X` or `Y`, then `suffix` — in that order, with no other content between the elements.

3.5 Grammar Actions

SGDL's *action* construct is the mechanism by which implementation-language code is associated with grammar elements. An action is a numeric tag followed by a sequence of string literals, the whole enclosed in `'='` delimiters. The form is:

```
action ::= integerToken "=" { stringToken } "="
```

A complete action looks like:

```
17 = "// generated code body" "more lines" =
```

The numeric tag (the `integerToken` that opens the action) is the action's identifier within the grammar. The string sequence is the action's payload — the implementation-language text to be emitted at this position by the generator. The delimiters `'='` (which open and close the payload) are reserved for this purpose.

Two specific action tags appear in the meta-grammar and in the patent's presentation:

- `METAACTBEG` — a reserved word equivalent to `0`, marking the beginning of a multi-action group.
- `METAACTEND` — a reserved word equivalent to `0`, marking the end of a multi-action group.

Both expand to the integer `0` followed by the customary delimiters; they are not distinct token classes, but reserved-word abbreviations that improve the readability of the meta-grammar. Patent §6, columns 5–6 documents this convention. User grammars may use these reserved words or write the equivalent literal `0=` …`=` form directly.

Actions are attached to grammar elements through the `actions` construct (note the plural):

```
actions ::= "=" { action } "="
```

which permits zero or more actions to be enclosed in a single `'='` pair. An action group attached to a non-terminal terminal pair (the construct `ntermtermact` in the SGDL grammar) means: when the parser recognizes the preceding non-terminal or terminal, the actions in this group are recorded against the corresponding rule instance and are emitted by the generator in the same order they appear here.

A subtle but important property of SGDL's action mechanism is that actions do **not** carry semantic meaning at parse time. The parser stores the actions in the runtime; the generator emits them into the produced source files; the developer's implementation in Phase 2.4 is what gives them meaning. This is the formal expression of the strict syntax–semantics separation declared in Section 2.2 (departure 2). A grammar with no actions parses cleanly and produces a runtime; a grammar with actions parses identically and produces a runtime that includes the action data. The semantic processing the actions ultimately drive happens in code that is downstream of the runtime, not in the grammar itself.

3.6 meta.sgr — SGDL on SGDL

Section 3.2 reproduces `meta.sgr` verbatim. This section examines what that grammar reveals about SGDL — properties that follow from the listing rather than from any document about it. Three observations are worth surfacing explicitly.

First, the meta-grammar exercises every element kind defined in Section 3.4. Plain references appear in `grammarNameDef` and `nterm` (which both reduce to `identifier`). Alternation appears in `element` (five-way: `identAlt | alternative | identMiss | iteration | action`) and in `ntermterm` (two-way: `nterm | termToken`). Iteration appears in `right`, `action`, `actions`, `identAlt` (via `Altpart`), `iterItems` (via `altIterItem`), and in the `{ rule }` of `grammar` itself. Square-bracket optionality appears in `ntermtermact` and `iterItemact` (each making the `actions` element optional). Angle-bracket optionality does not appear, because no rule in the meta-grammar requires the constrained-iteration semantics; the construct exists for use in user grammars where it is occasionally needed.

Second, the file structure is fully parenthesized. Every rule's right-hand side begins with `'('` and ends with `')'`, as the rule `rule ::= '(' nterm '::=' right ')'` specifies. The whole file is wrapped in a single outer parenthesis pair, as the rule `grammar ::= '(' grammarNameDef { rule } ')'` specifies. The outer parenthesis directly encloses the rule sequence; there is no separate parenthesization around `{ rule }`. The result is a Lisp-like nesting: an SGDL file is a single S-expression whose first element is the grammar name and whose remaining elements are rule definitions, each itself an S-expression. This parenthesization replaces the role that statement-terminator characters such as semicolons play in other notations: rule boundaries are fully determined by parenthesis balancing.

Third, the meta-grammar contains actions only on the outermost `grammar` rule. Every other rule defines its right-hand side in pure SGDL with no embedded code. This is intentional: the meta-grammar describes the *shape* of SGDL grammars; the implementation of what to do with that shape is in the generator subsystem (Chapter 7), which is itself written in C++ and operates on the runtime that the meta-grammar produces. The two action macros on `grammar` — `METAACTBEG();` opening and `METAACTEND();` closing — emit code at the start and end of the generated parsing routine, providing the framework hooks that connect the parser's recognition of an SGDL file to whatever surrounding scaffolding the host code requires.

3.7 The Class of Supported Grammars

CCS supports a subset of context-free grammars suitable for top-down recursive-descent parsing. The reference implementation's parser uses LL(1) parsing with one token of lookahead, augmented by the action machinery of Section 3.5. This section specifies the precise constraints a grammar must satisfy to be

supported, the constraints the implementation does not enforce, and the boundaries of the round-trip property.

3.7.1 Constraints Enforced by the Implementation

The CCS executable rejects, at compile time, grammars that violate the following constraints.

- **Single definition per non-terminal.** Each non-terminal is defined by exactly one rule. The alternation construct `|` is used to express a non-terminal that has multiple right-hand-side forms; multiple top-level rules sharing the same left-hand side are not permitted. A grammar with two rules both defining `foo` is rejected; rewrite as `(foo ::= form1 | form2)`.
- **No undefined non-terminals.** Every non-terminal that appears on a right-hand side must be defined somewhere in the file. The order of definitions does not matter (forward references are permitted), but every non-terminal must eventually be defined. Predefined token classes are exempt; they are not non-terminals.
- **LL(1) determinacy.** At every point where the parser must choose among alternatives — whether expressed by `|` alternation, by `{ … }` iteration entry/exit, or by `[ … ]` presence/absence — the choice must be determinable from a single token of lookahead. Grammars with longer-lookahead requirements, or with ambiguous alternatives whose first sets overlap, are rejected.
- **No left recursion.** A non-terminal whose first right-hand side begins with a reference to itself, directly or transitively, causes the recursive-descent parser to loop. Such grammars are rejected. Iteration constructs (the `{ … }` braces) are the SGDL idiom for what would otherwise be expressed as left-recursive rules in a bottom-up notation.

3.7.2 Constraints Not Enforced

Some grammar properties that are common subjects of analysis in compiler construction are not checked by the CCS executable.

- **Reachability.** A non-terminal that is defined but never referenced from the grammar's axiom is permitted. The parser will simply never invoke it. This permits incremental grammar development, where a designer drafts new rules before connecting them to the rest of the grammar.
- **Productivity.** A non-terminal whose every alternative ultimately depends on itself, with no escape to a base case, is permitted by the static check but causes the parser to fail at run time on any input that requires that non-terminal. The grammar designer is responsible for ensuring productivity.
- **Action consistency.** The numeric tags appearing in action constructs are not checked for collision or coverage. Two actions in the same grammar may share a tag; an action tag may be unused by the generator. Both situations are the developer's responsibility.

3.7.3 The Round-Trip Property

The de-compilation phases (Phase 2.1b from runtime to source, Phase 2.2c from binary to source, and the implicit Phase 2.2b from binary to runtime) are specified to produce output identical to the original input *modulo indentation rules*. This property is empirically demonstrated in the self-test sequences of Chapter 8, but it is not formally proved. Three caveats apply.

First, "identical modulo indentation" means the de-compiled text differs from the original only in whitespace placement: the same tokens appear in the same order, but the line breaks and indent depths chosen by the de-compiler may differ from those chosen by the original author. The de-compiler's indentation policy is a property of the generator-style class instantiated for the grammar; `GeneratorRuntimeStyleMeta` implements one indentation convention, `GeneratorRuntimeStyleXML` another, and additional generator-style classes can be added.

Second, the round-trip property has been verified empirically only for grammars in the LL(1) subclass that the parser accepts. Grammars that exceed this subclass are rejected at compile time and the question does not arise for them. Within the LL(1) subclass, the property has been observed to hold for the meta-grammar itself and for the grammars supplied with the reference implementation, but no formal proof of the property's general validity has been published.

Third, the bidirectionality between runtime and binary forms (Phases 2.2a and 2.2b) is logically separate from the de-compilation property: it concerns whether `SyntaxControlledRuntime` and `SyntaxControlledBinary` carry isomorphic information, not whether either can be projected back to source text. Chapter 6 specifies the isomorphism in detail through the binary's `vector<Tag>` layout. The empirical demonstration of `runtime → binary → runtime` identity, given in the self-test sequence (b.1)–(b.6) reproduced in Chapter 8, is independent of the demonstrations of `source → runtime → source` and `source → runtime → binary → source`.

Characterizing the precise grammar class for which the runtime-to-source and binary-to-source round-trips are bijective is identified in Section 1.3 as a property CCS demonstrates rather than proves. This specification documents the property as it operates on tested inputs; a formal characterization is left as an open question and would be a natural subject for a separate technical note.

Chapter 4 — Compiler Compiler Management

The management subsystem orchestrates compilation. It receives the user's command-line arguments, instantiates the compilers needed for the requested operation, drives them through their phases, and routes diagnostic output. It also owns the lexical and parsing infrastructure that every grammar uses: the keyword catalog, the tokenizer, the language abstraction, and the parser front end.

This chapter specifies seven cooperating components: the `Shell` and its `Compiler` children with the `Action` hierarchy (Section 4.1); the `KeyWordsContainer` that catalogs the four kinds of grammar symbol (Section 4.2); the `Tokenizer` and the `TokenNameContainer` that supplies its naming layer (Section 4.3); the `Tokens` container and the `Token` class hierarchy (Section 4.4); the `Language` abstraction and its built-in subclasses (Section 4.5); the `Parser` (Section 4.6); and the `Tag`, `UnsignedTag`, `Real`, and `TypeInstance` types that carry packed identifiers across the entire system (Section 4.7).

4.1 Shell, Compiler, and the Action Hierarchy

The top of the management subsystem is `Shell` — the class that the executable's `main` function instantiates. `Shell` parses command-line arguments, allocates a `Logger` for diagnostic output, allocates a vector of `Compiler` instances corresponding to the operation requested, and dispatches the run. Figure 4.1 shows the management classes and their relationships.

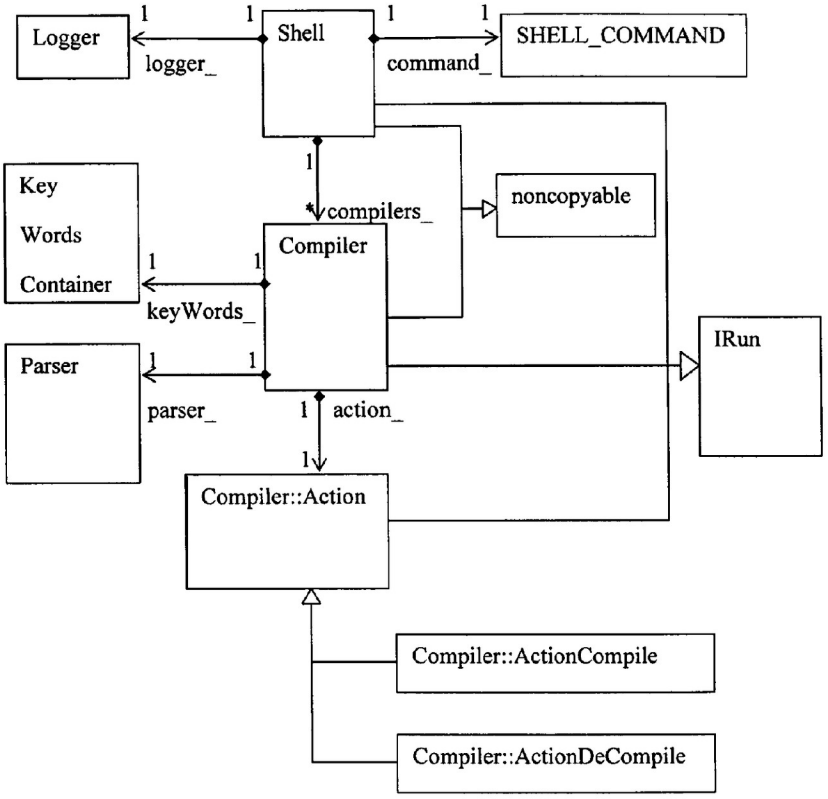

FIG. 5

Figure 4.1 — Compiler compiler management classes (FIG 5 of US 8,464,232). Shell owns a Logger, a SHELL_COMMAND code, and a vector of Compiler instances. Compiler owns a KeyWordsContainer, a Parser, and an Action. Action is an abstract base for ActionCompile and ActionDeCompile.

Several details of Figure 4.1 are worth noting.

- `Shell` is derived from the `IRun` interface (shown at the right of Figure 4.1). `IRun` declares a single virtual method, `run`, that `Shell` implements as the entry point of compilation. `Compiler` and `Action` are also derived from `IRun`, which makes the three classes interchangeable as the recipient of a top-level run instruction. The chain `Shell::run → Compiler::run → Action::run` is the path along which control flows from the command line to the action that performs compilation or de-compilation.
- `SHELL_COMMAND` is an enumerated type whose values name the operations the executable can perform: compile, de-compile from runtime, de-compile from binary, generate code, and so on. `Shell` stores the value parsed from the command line in its `command_` member and uses it to select which `Action` subclass to instantiate within each `Compiler`.
- `Shell` owns a *vector* of `Compiler` instances rather than a single one. The vector holds one `Compiler` per grammar that the operation traverses. For the bootstrap loop of Chapter 2, the vector contains a single `Compiler` instance for the meta-grammar; for a multi-grammar operation (such as building a target compiler whose source grammar imports declarations from another grammar) the vector contains one `Compiler` per source file involved.
- The `noncopyable` base class connected to `Compiler` enforces a deletion of the copy constructor and copy-assignment operator. `Compiler` instances own significant state (a parser, a runtime, a binary) and are referenced by pointer through the system; copying one is never the right operation, and the type system reflects this.

Within each `Compiler` the `Action` hierarchy implements the operation. `Action` is abstract; `ActionCompile` performs Phase 2.1a (source text → runtime) and, if the operation includes binary generation, Phase 2.2a (runtime → binary). `ActionDeCompile` performs Phase 2.1b (runtime → source) or Phase 2.2c (binary → source) depending on which form is currently in `Compiler`. The decision about which subclass to instantiate is made by `Shell` from the `SHELL_COMMAND` value at the start of the run.

The `Logger` class, allocated by `Shell` and shared by reference with every `Compiler` and every `Parser` (Section 4.6), accumulates diagnostic messages produced during compilation. It is the single point through which all parse errors, semantic errors, and informational messages flow to the user. Its interface is straightforward — append a message at a designated severity level, query whether any messages above a threshold have been recorded, write the accumulated log to a stream — and is not specified in further detail here.

4.2 KeyWordsContainer

`KeyWordsContainer` is the catalog of grammar symbol names that the parser builds as it processes a grammar specification. It is owned by `Compiler` (visible as the `keyWords_` member in Figure 4.1) and shared by reference with the `Tokenizer` and `Parser` that `Compiler` instantiates. It exists to give the rest

of the system constant-time access from a textual symbol name (such as `grammar` or `'::='`) to the integer index by which the runtime, binary, and generator refer to that symbol.

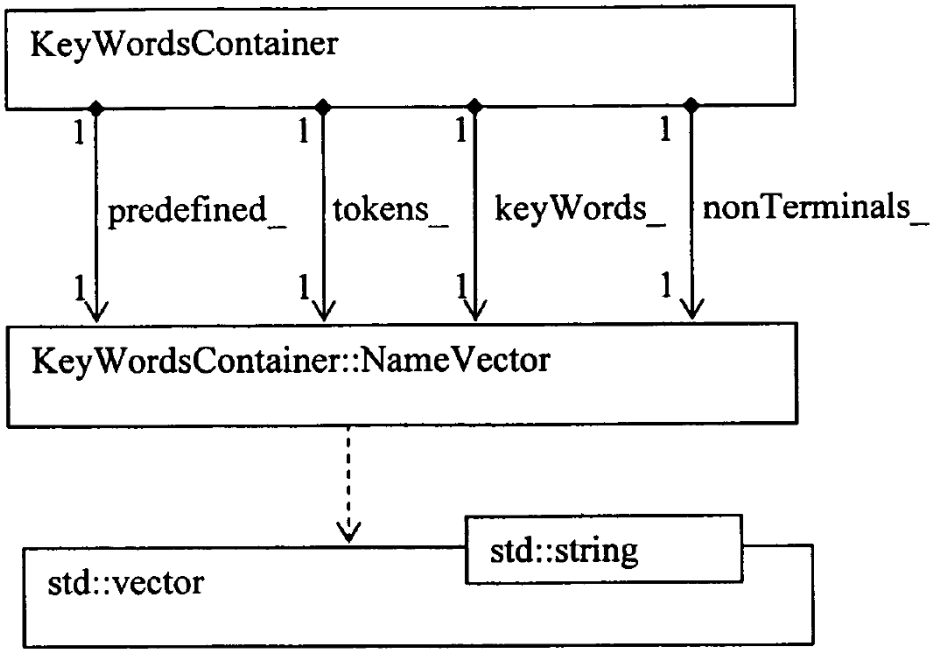

FIG. 6

Figure 4.2 — KeyWordsContainer (FIG 6 of US 8,464,232). Four parallel name vectors — predefined_, tokens_, keyWords_, nonTerminals_ — categorize every name appearing in the grammar. Each is an instance of KeyWordsContainer::NameVector, which is implemented as std::vector<std::string>.

The four name vectors partition the grammar symbol space into disjoint categories.

- `predefined_` — the four predefined token classes that exist in every grammar: `identifier`, `integerToken`, `stringToken`, `termToken`. These are populated by the `KeyWordsContainer` constructor before any grammar processing begins.
- `tokens_` — terminal tokens that the grammar uses, beyond the four predefined classes. Each entry is the textual content of a single-quote-enclosed terminal in the SGDL source: `'('`, `'::='`, `'|'`, and so on. The tokenizer recognizes these as fixed-character lexemes.
- `keyWords_` — reserved words specific to the grammar. For SGDL itself this includes `METAACTBEG` and `METAACTEND`; for a target grammar, the reserved words declared by the grammar's `Language` subclass (Section 4.5).
- `nonTerminals_` — the non-terminals declared by the grammar. For `meta.sgr`, this vector contains `grammar`, `grammarNameDef`, `rule`, `nterm`, and the rest of the meta-grammar's left-hand-side names. The vector is populated by `Parser` as it encounters each rule definition.

Every name across the four vectors has a unique sequential index when the four vectors are concatenated in the order shown. This is the *symbol ID* that the runtime, binary, and generator use to refer to the symbol

elsewhere. The mapping from name to ID is the textual-key lookup `std::find` against the appropriate vector; the mapping from ID to name is the index dereference `vec[id]`. Both operations are constant-time on small grammars and remain efficient at scale because the name vectors are bounded by the size of the grammar specification, not by the size of the input being parsed.

The inner type `KeyWordsContainer::NameVector` is a thin alias for `std::vector<std::string>` (the `std::vector` and `std::string` boxes at the bottom of Figure 4.2). The alias exists for documentary purposes; methods that take a `NameVector` parameter are clearer about their intent than methods that take a generic `vector<string>`.

4.3 Tokenizer

The `Tokenizer` converts a stream of characters into a stream of tokens. It is the lexical layer that sits between the source text and the parser. Every `Compiler` instantiates a `Tokenizer` as part of its construction; the tokenizer reads from a `LineReader` subclass and delivers tokens to the `Parser` on demand. Figure 4.3 shows the tokenizer's class structure.

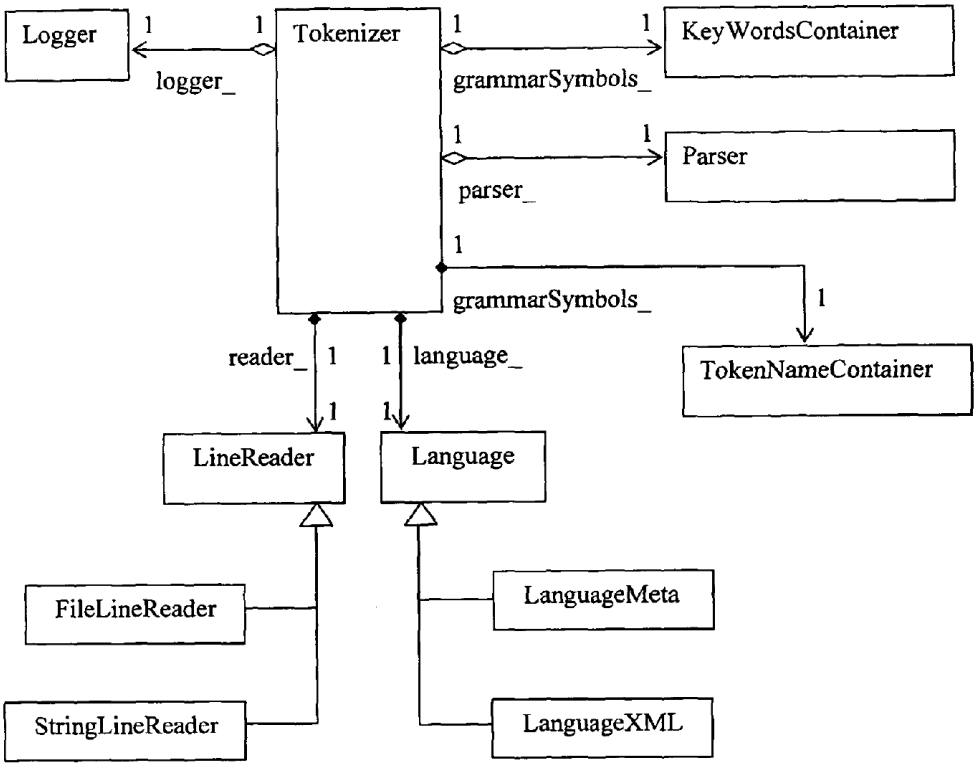

FIG. 27

Figure 4.3 — Tokenizer and its collaborators (FIG 27 of US 8,464,232). Tokenizer references a Logger, a KeyWordsContainer (as grammarSymbols_), a Parser, a TokenNameContainer (also as grammarSymbols_), a LineReader (as reader_), and a Language (as language_). LineReader has subclasses FileLineReader and StringLineReader; Language has subclasses LanguageMeta and LanguageXML.

The `LineReader` abstraction lets `Tokenizer` read from any character source. Two concrete subclasses are provided:

- `FileLineReader` — reads from an open file, line by line, and maintains a listing of the source content for diagnostic purposes.
- `StringLineReader` — reads from an in-memory `std::string`, delimiting lines on a configurable separator. Used by code that compiles grammar text held in memory rather than read from disk — for example, when a generated test harness compiles a small grammar embedded as a literal in the test source.

Both subclasses derive from a pure-virtual `LineReader` base whose interface comprises a `get` method (return the next line by reference), `hasListing` (return true if the reader maintains a listing), `isEof` (true at end of source), `start` (initialize), and `put` (append to the listing). Additional subclasses can be added by deriving from `LineReader` and implementing these methods.

Two associations on the right side of Figure 4.3 are worth highlighting. The `grammarSymbols_` member appears *twice* on `Tokenizer` — once typed as `KeyWordsContainer` and once typed as `TokenNameContainer`. The two members are not the same: the `KeyWordsContainer` reference (shared with `Compiler` and `Parser`) gives the tokenizer access to the four name vectors of Section 4.2 for assigning symbol IDs to recognized tokens. The `TokenNameContainer` reference (owned by the tokenizer or by the `Language`, depending on the grammar) maps token-class names to display strings used in diagnostics.

Figure 4.4 shows `TokenNameContainer` in detail.

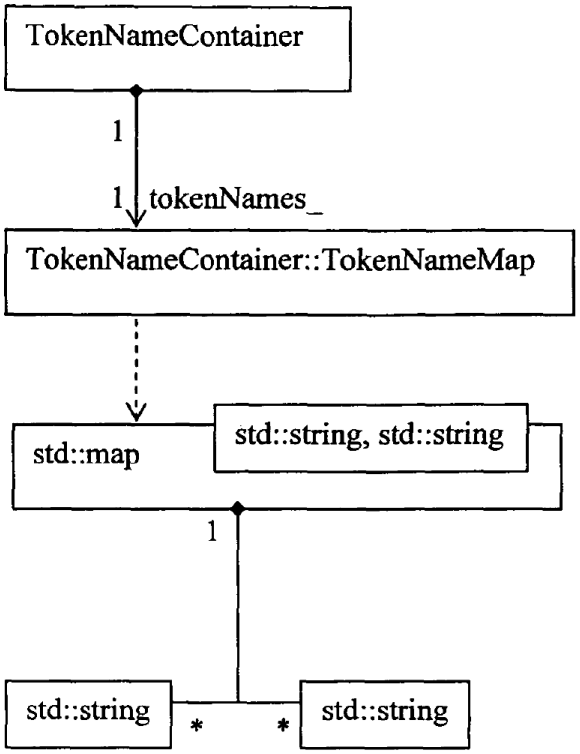

FIG. 28

Figure 4.4 — TokenNameContainer (FIG 28 of US 8,464,232). TokenNameContainer owns a TokenNameMap, which is an instance of std::map<std::string, std::string>.

The single member, `tokenNames_`, maps from a token-class name (the key) to a human-readable string (the value). For instance, `integerToken` may map to the descriptive string `"integer literal"` for use in error messages. The mapping is populated by the `Language` subclass active for the grammar (Section 4.5); the mapping is queried whenever the tokenizer or parser produces a diagnostic that needs to refer to a token class by name.

4.4 Tokens and the Token Class Hierarchy

Where `TokenNameContainer` (Section 4.3) maps token-class names to display strings, the `Tokens` container holds the actual `Token` instances that recognize input characters. `Tokens` is owned by `Language` and queried by `Tokenizer` through the `language_` member of Figure 4.3. Figure 4.5 shows the `Tokens` class.

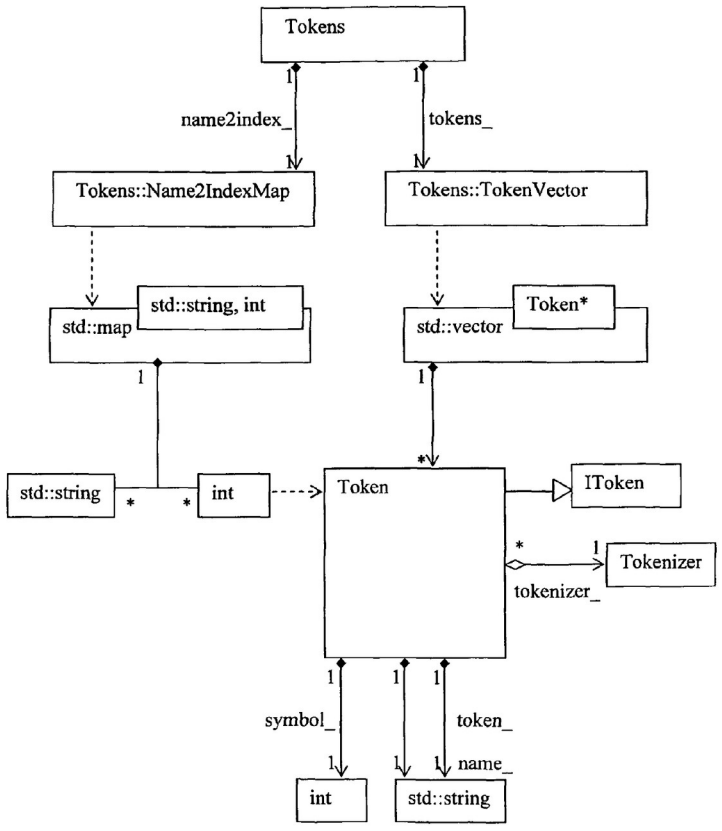

FIG. 30

Figure 4.5 — Tokens container (FIG 30 of US 8,464,232). Tokens owns a Name2IndexMap (std::map<std::string, int>) for name-to-index lookup and a TokenVector (std::vector<Token>) for index-to-token lookup. Each Token derives from IToken, holds a reference to its Tokenizer, and stores its symbol_ id, token_ string content, and name_ class name.*

The dual representation — `Name2IndexMap` and `TokenVector` — is the same idiom that the runtime's `MapVectorContainer` template uses (Chapter 5): a map from a textual key to a sequential index, paired with a vector that recovers the original object from the index. The result is constant-time access by name and constant-time access by sequential ID, with the vector preserving the order in which tokens were registered.

The `Token` class is abstract; concrete subclasses implement specific recognition strategies. Figure 4.6 shows the hierarchy.

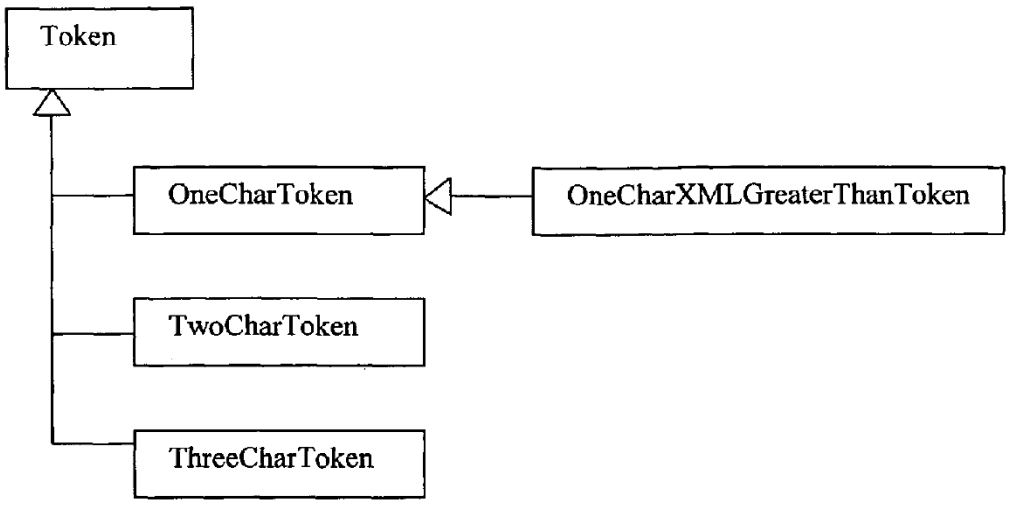

FIG. 31

Figure 4.6 — Token class hierarchy (FIG 31 of US 8,464,232). The abstract Token class has three subclasses recognizing fixed character sequences of length one, two, or three. OneCharXMLGreaterThanToken further specializes OneCharToken for XML's context-sensitive treatment of the > character.

The three primary subclasses correspond to the longest-match needs of typical grammars:

- `OneCharToken` — recognizes a single specified character. Used for the punctuation tokens `'('`, `')'`, `'|'`, `'='`, and the others enumerated in Section 3.2.
- `TwoCharToken` — recognizes a fixed two-character sequence such as ==, <=, or >= that a target grammar may use.
- `ThreeCharToken` — recognizes a fixed three-character sequence such as `'::='` itself.
- `OneCharXMLGreaterThanToken` — a context-sensitive specialization of `OneCharToken` for XML, where the > character's interpretation depends on whether the tokenizer is currently inside an XML tag or in element content. The class exists because the patent's XML grammar requires this distinction to be made at the tokenizer level rather than at the grammar level.

Each `Token` subclass implements three pure-virtual methods inherited from the `IToken` interface: `isToken` (does the current input position match this token?), `flushToken` (advance the tokenizer past the matched characters and skip whitespace), and the constructors and accessors for the symbol ID, the matched character sequence, and the token-class name.

4.5 Language and Language Subclasses

`Language` is the abstraction that bundles together the lexical machinery for a particular source language. It owns a `Tokens` container holding the language's token classes; it owns the keyword and non-terminal sets; it knows how to populate the tokenizer's state when a particular token class is recognized; and it provides the language-specific behavior for primitive operations such as recognizing a numeric literal or a string literal. Figure 4.7 shows the class.

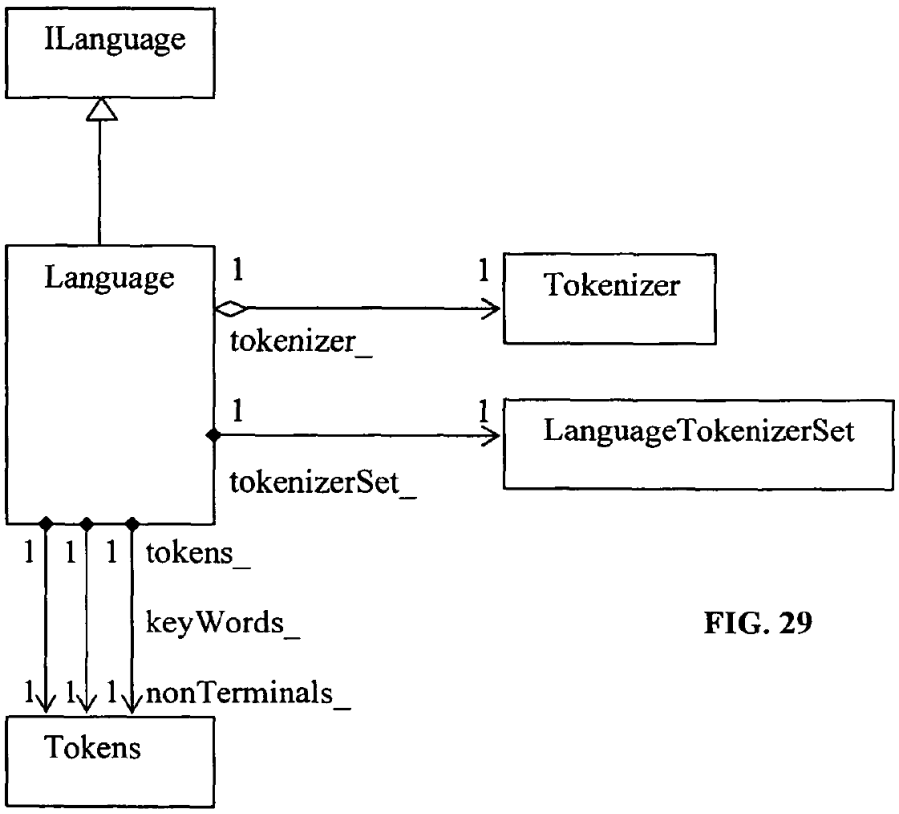

Figure 4.7 — Language and its data members (FIG 29 of US 8,464,232). Language derives from ILanguage, references a Tokenizer, owns a LanguageTokenizerSet (an enumerated value selecting which built-in language tokenizer set is in use), and owns three Tokens instances — tokens_, keyWords_, and nonTerminals_ — corresponding to the categories of grammar symbol.

The three `Tokens` instances inside `Language` parallel three of the four name vectors of `KeyWordsContainer` (Section 4.2): the language's terminals, its reserved keywords, and its non-terminals. The fourth name vector — `predefined_` — does not appear in `Language` because the four predefined token classes are common to every language; they are populated by `KeyWordsContainer` itself, not by language-specific code.

`LanguageTokenizerSet` is an enumerated type whose values name the built-in tokenizer configurations the system ships with. The two values present in the reference implementation correspond to the two concrete `Language` subclasses:

- `LanguageMeta` — the language of SGDL itself. Its `Tokens` instances populate with the SGDL terminals (`'('`, `')'`, `'::='`, and the rest), the SGDL reserved keywords (`METAACTBEG`, `METAACTEND`), and the SGDL non-terminals as they are encountered in the grammar being parsed.
- `LanguageXML` — the language of XML. Its `Tokens` instances populate with XML's terminals (including the context-sensitive `OneCharXMLGreaterThanToken` of Section 4.4), reserved attribute names, and non-terminals derived from the XML schema being processed.

Additional `Language` subclasses are added when targeting CCS at a new source language whose lexical structure exceeds what the predefined classes provide. Chapter 10 walks through this procedure in the context of targeting a new grammar end-to-end.

The `ILanguage` base class defines the contract that every `Language` subclass must satisfy: pure-virtual methods including `getCharacter` (advance to the next input character), `getIdentifier` (recognize an identifier-class token), `getNumeric` (recognize a numeric literal), `getString` (recognize a string literal), and `getNextToken` (the top-level dispatch). The default implementations in `Language` itself handle the common cases and delegate the language-specific behavior to the `Tokens` instances; subclasses override only the methods whose behavior they need to alter.

4.6 Parser

The `Parser` is the heart of the front end. It consumes tokens from the `Tokenizer` and constructs the `SyntaxControlledRuntime` that the rest of the system operates on. Every grammar produces, at code-generation time, a parser subclass derived from the abstract `Parser` class shown in Figure 4.8; the generated parser implements the recognition logic for the specific grammar, while the base class supplies the infrastructure that all parsers share.

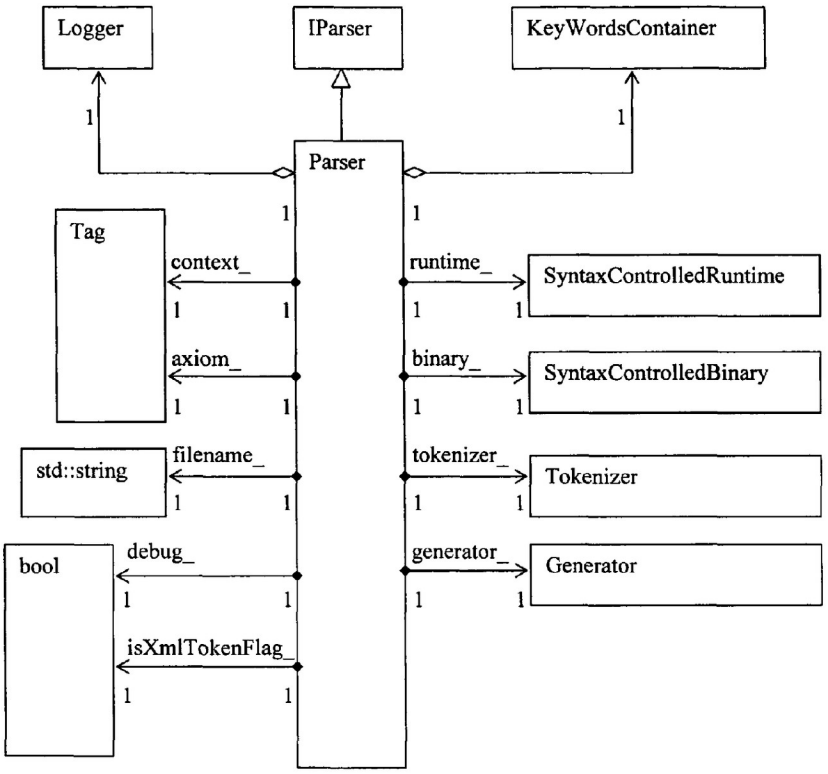

FIG. 7

Figure 4.8 — Parser class (FIG 7 of US 8,464,232). Parser derives from IParser, references a Logger and KeyWordsContainer, holds Tag-typed members context_ and axiom_, a std::string filename_, two boolean flags debug_ and isXmlTokenFlag_, and pointers to its SyntaxControlledRuntime, SyntaxControlledBinary, Tokenizer, and Generator collaborators.

The `Parser` class's data members fall into three groups: the diagnostic and lexical infrastructure (top-left and top-right of Figure 4.8), the parsing state (center-left), and the artifacts the parser produces or operates on (right column).

4.6.1 Lexical and Diagnostic Infrastructure

The `logger_` member is a reference to the `Logger` shared by the entire `Compiler` (Section 4.1). Every parsing diagnostic — syntax error, unexpected token, lookahead failure — flows through this reference. The `keyWords_` reference is a similar shared reference to the `KeyWordsContainer` of Section 4.2; the parser uses it to assign symbol IDs to non-terminals as their definitions are encountered.

4.6.2 Parsing State

Two `Tag`-typed members hold the parser's parsing state: `context_` is the `Tag` identifying the runtime context currently being populated, and `axiom_` is the `Tag` identifying the symbol that the grammar names as its axiom (the rule whose right-hand side defines the entire program). At the start of parsing, `axiom_` is the symbol ID of the grammar's first non-terminal definition; `context_` is the ID of the initial context. Both values change during parsing as the grammar's structure dictates.

Two boolean members hold parser-mode flags. `debug_` enables verbose tracing of the parser's actions, used during development of new grammars. `isXmlTokenFlag_` is a parser-state flag specific to XML processing: it indicates whether the tokenizer should treat `>` as an XML close-tag terminator (inside a tag) or as element content (outside one). The flag is set and cleared by the parser as it enters and exits tag-recognition rules; the tokenizer reads it through the parser reference shown in Figure 4.3.

The `filename_` member is the name of the source file being parsed, retained in the parser for error message construction. It is not used by the parsing logic itself.

4.6.3 Artifact References

Four members hold the artifacts the parser interacts with. `runtime_` points to the `SyntaxControlledRuntime` the parser is populating (Chapter 5). `binary_` points to the `SyntaxControlledBinary` that the runtime will be serialized into when Phase 2.2a runs (Chapter 6). `tokenizer_` is the `Tokenizer` supplying the parser with tokens (Section 4.3). `generator_` is the `Generator` that emits source code at Phase 2.3 (Chapter 7); the parser holds a reference to it so that action macros encountered during recognition can be routed to the generator immediately.

4.6.4 IParser and Generated Parser Subclasses

The base class `IParser` (top center of Figure 4.8) declares the pure-virtual methods that every grammar-specific parser subclass must implement. The principal methods correspond to compilation entry points: `compileFromFile` (parse from a named file, with or without preserving a listing), `compileFromString` (parse from an in-memory string). The base `Parser` class provides default implementations of these in terms of the recursive descent procedures that the generator emits per grammar; the subclass for a specific grammar overrides the procedures themselves but inherits the entry-point structure.

4.7 Tag, UnsignedTag, Real, and TypeInstance

Throughout the management subsystem, the runtime, and the binary, identifiers and small integer values are carried in a small family of typedef and struct types collectively named the *tag types*. Their definition is conditional on the target platform's word size:

```
#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
```

On a 32-bit platform, `Tag` and `UnsignedTag` are 4-byte words; `Real` is a 4-byte single-precision floating-point value; `TypeInstance` packs a 10-bit symbol ID and a 22-bit rule ID into a single 32-bit word. On a 64-bit platform, the corresponding sizes are 8 bytes for the integer and floating-point types, with a 16-bit symbol ID and a 48-bit rule ID inside `TypeInstance`.

The bit allocations within `TypeInstance` are not arbitrary. The 10-bit `symbolID` on 32-bit platforms permits up to 1,024 distinct grammar symbols across the four name-vector categories of `KeyWordsContainer` (Section 4.2); this is sufficient for grammars at the scale of full programming languages, and considerably more than is needed for SGDL itself, XML, or JSON. The 22-bit `ruleID` permits up to roughly 4 million rule instances to be referenced from within a single context, supporting source files of essentially unlimited practical size. On 64-bit platforms the limits are correspondingly higher: 65,536 distinct symbols and approximately 281 trillion rule instances.

`TypeInstance` is the type that appears in the dynamic part of a `Rule` (Chapter 5) when the rule's recognition involves invoking another non-terminal's rule: the embedded `symbolID` identifies which symbol the inner rule belongs to, and the `ruleID` identifies which instance of that symbol's recognition was matched. Packing both into a single word — rather than using a pair of separate `Tag` values — halves the memory footprint of the rule's dynamic part and is one of several measures by which CCS keeps the runtime footprint small even for large input files.

The choice between the 32-bit and 64-bit packings is made at compile time of the CCS executable through the `_CPPCC_TAG_32BITS_` preprocessor symbol. A binary produced on a 32-bit build is not interchangeable with a binary produced on a 64-bit build — the `Tag` word size differs, so the tagged vector layout is incompatible. Chapter 11 specifies the multiplatform considerations that follow from this and the file-format provisions for cross-platform binary exchange.

Chapter 5 — SyntaxControlledRuntime

`SyntaxControlledRuntime` is the in-memory representation of a successfully parsed program. The parser builds it; semantic processing consumes it; the binary form serializes it. Chapter 2 introduced the runtime as one of the three entities of the Parsing Model and named the four-entity schema (Context, Name, Symbol, Rule) that organizes it. This chapter specifies the runtime in full: the logical view that Section 5.1 presents corresponds class-for-class to the C++ implementation laid out in Sections 5.2 through 5.7, and the API enumerated in Section 5.8 is the surface through which generated parsers and semantic-processing code read and write the runtime.

A reader who has internalized Chapter 2 will find this chapter mostly definitional: every claim made about the runtime in Chapter 2 is realized as a specific C++ class member or method here. A reader skipping Chapter 2 should at least be aware that the runtime's organization — contexts containing names and symbols, symbols containing rules — is the same organization that the binary form preserves in flat memory (Chapter 6). The two forms are interconvertible because they share this schema.

5.1 Logical View

Figure 5.1 shows the runtime's logical view: the four entity classes and the multiplicity relationships among them. The logical view is identical to the binary's logical view (Chapter 6) because the two forms are different physical representations of the same parsing schema.

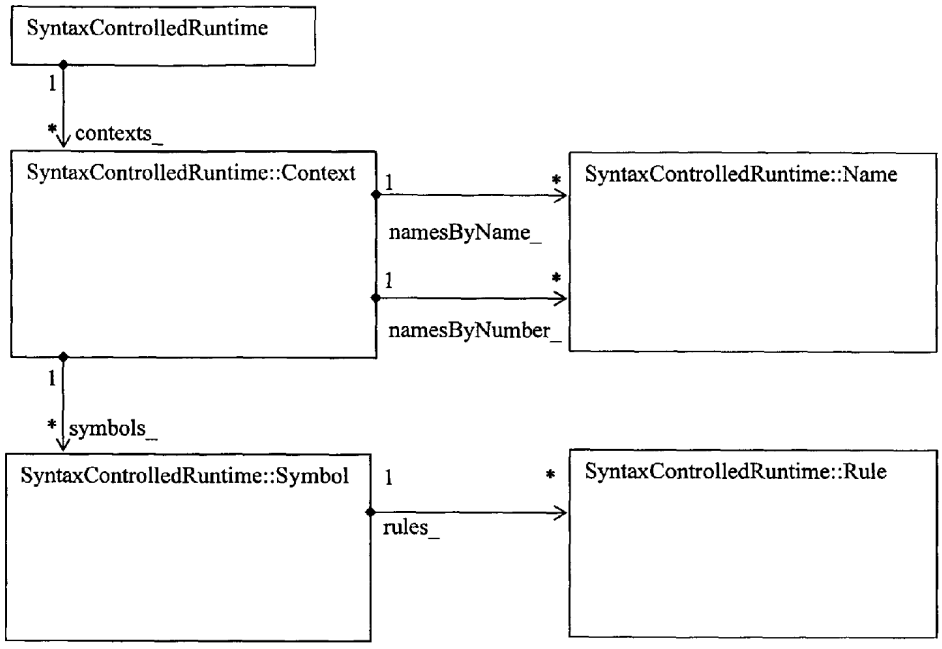

FIG. 18

Figure 5.1 — Runtime logical view (FIG 18 of US 8,464,232). SyntaxControlledRuntime owns multiple Context instances. Each Context owns multiple Name instances, indexed by name and by number. Each Context owns multiple Symbol instances. Each Symbol owns multiple Rule instances.

Three relationships in Figure 5.1 are worth noting before the detailed class-by-class specification.

- Each `Context` has *two* associations to its names: `namesByName_` (a name-keyed lookup) and `namesByNumber_` (an index-keyed lookup). Both associations point to the *same* set of `Name` instances; they differ only in which key the lookup uses. The dual indexing is the central performance idiom of the runtime, generalized in Section 5.2 as the `MapVectorContainer` template.
- Symbols and rules have only a single association each (no by-name lookup for them). The reason is that grammar symbols are referenced by their numeric `symbolID` once parsing begins, not by their textual name; the textual-to-numeric mapping is a one-time operation handled by `KeyWordsContainer` (Section 4.2). Names — which can be arbitrary identifiers introduced by the source program — require both lookup directions because both the source program and downstream analysis use textual names.
- There is no top-level association between `SyntaxControlledRuntime` and `Name` or `Rule`. Names live inside contexts; rules live inside symbols. To reach a particular name, code descends from the runtime to the relevant context to the names; to reach a particular rule, it descends through context and symbol. This nesting is enforced by the class structure rather than by access modifiers.

5.2 The MapVectorContainer Template

Almost every container in the runtime is an instance of the same template: `MapVectorContainer<D, K>`. Figure 5.2 shows the template.

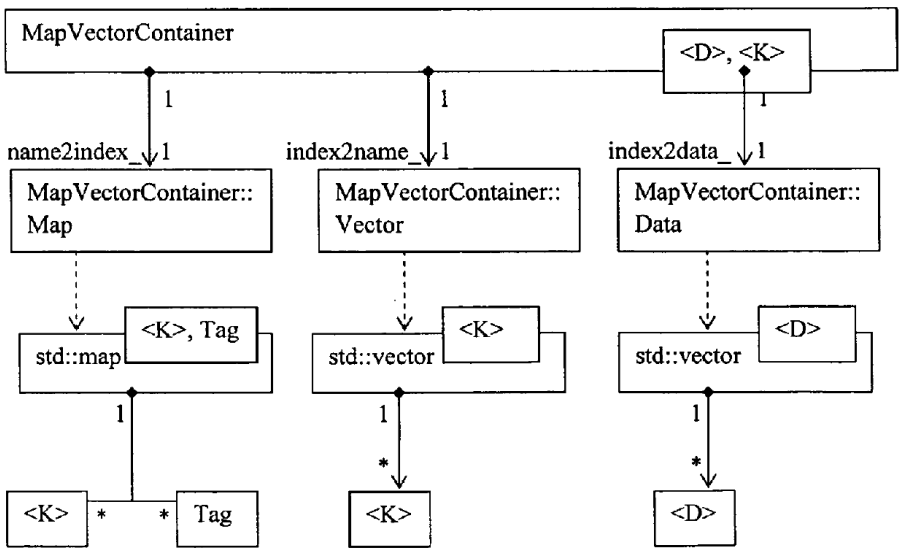

FIG. 9

Figure 5.2 — MapVectorContainer template (FIG 9 of US 8,464,232). The template is parameterized by D (the data element type) and K (the key type). It owns three members: name2index_ (a Map from K to Tag), index2name_ (a Vector of K), and index2data_ (a Vector of D). Each member is a thin alias for std::map<K, Tag>, std::vector<K>, or std::vector<D> respectively.

The template provides three operations of constant amortized cost:

- **Lookup by key.** Given a value of type `K` (typically a name string), `name2index_` returns the sequential index `Tag` assigned when the entry was inserted.
- **Lookup by index.** Given a sequential index `Tag`, `index2data_` returns the data of type `D` associated with that index.
- **Original-key recovery.** Given a sequential index, `index2name_` returns the key `K` originally inserted at that index. This is the inverse of lookup-by-key and is needed when de-compilation produces the source-level name from a runtime instance.

The two-vector representation — `index2name_` and `index2data_` — might appear redundant: `index2data_` alone could be enough if the data structure included the key. But the template is used in cases where the data type `D` is large (e.g., a full `SyntaxControlledRuntime::Context` instance) and the key is small (e.g., a `Tag`). Storing the key separately in `index2name_` lets de-compilation iterate keys without dereferencing data values, which can be expensive when the data values themselves contain nested containers.

The template is instantiated five times in the runtime, with different `<D, K>` bindings. Sections 5.3 through 5.7 specify each instantiation.

5.3 SyntaxControlledRuntime and ContextContainer

Figure 5.3 shows the outer `SyntaxControlledRuntime` class.

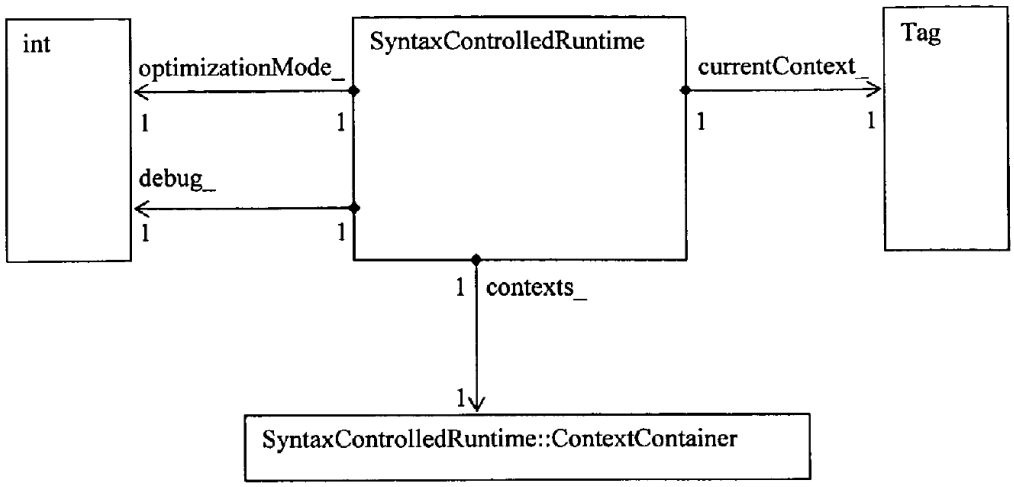

FIG. 8

Figure 5.3 — SyntaxControlledRuntime outer class (FIG 8 of US 8,464,232). The class has three direct data members: an int optimizationMode_, an int debug_, and a Tag currentContext. It also owns one ContextContainer (the contexts_ member) which holds the actual Context instances.

The three direct members:

- `optimizationMode_` (an `int`) — a flag controlling whether the runtime performs internal optimizations such as merging duplicate symbols across contexts. The default value is zero (no

optimization). The flag is read by the parser at the start of parsing and is otherwise stable for the lifetime of the runtime.

- `debug_` (an `int`) — a flag enabling verbose runtime diagnostics. When set, methods that modify the runtime emit trace messages through the `Logger` that the runtime's owning `Compiler` provides. Used during grammar development.
- `currentContext` (a `Tag`) — the ID of the context currently being populated by the parser. As the parser processes a grammar that introduces nested contexts, this value changes; it identifies the context to which subsequent name, symbol, and rule operations should be applied.

The fourth member, `contexts_`, is the `ContextContainer` that holds all the runtime's contexts. Figure 5.4 shows `ContextContainer` in detail.

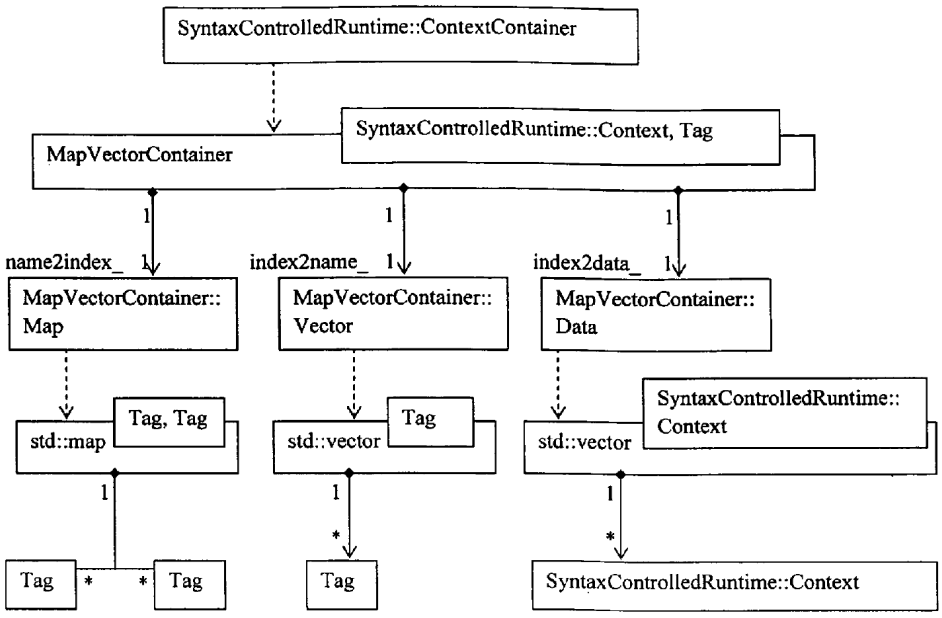

FIG. 10

Figure 5.4 — ContextContainer (FIG 10 of US 8,464,232). ContextContainer is an instantiation of MapVectorContainer with K = Tag and D = SyntaxControlledRuntime::Context. The three inner members are an std::map<Tag, Tag> (name2index_), an std::vector<Tag> (index2name_), and an std::vector<SyntaxControlledRuntime::Context> (index2data_).

The unusual property of `ContextContainer` is that both the key `K` and the index are of type `Tag` — there is no string-name lookup for contexts. The map's key is the context's `contextID` (which the parser assigns sequentially as new contexts are created), and the value is the same `Tag` returned as the index. This degenerate use of the template signals that the lookup by key is the same as the lookup by index for contexts; the template is used here for uniformity with the other containers rather than for distinct lookup semantics.

5.4 Context

Figure 5.5 shows the `Context` inner class.

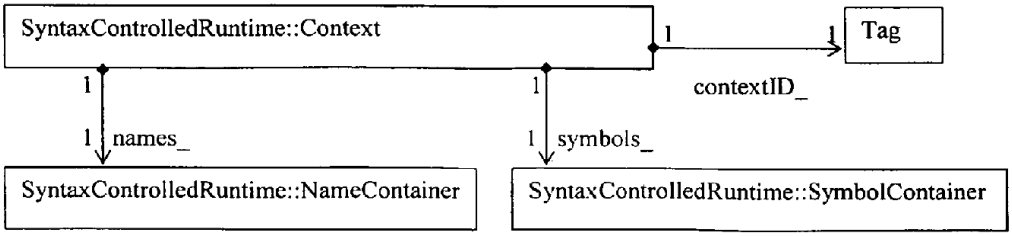

FIG. 11

Figure 5.5 — Context (FIG 11 of US 8,464,232). Context has three data members: contextID_ (a Tag identifying this context), names_ (a NameContainer holding the names declared within the context), and symbols_ (a SymbolContainer holding the symbol instances that the parser recognized within the context).

A `Context` is the unit of scoping in the runtime. Most simple grammars produce a single context per program: all names live in that context, all symbols are recognized within it. Grammars that introduce nested scopes — procedure bodies, namespace blocks, module declarations — produce one context per scope, with the parser switching `currentContext` on entry and exit of each scope.

The `contextID_` member is the `Tag` that the enclosing `ContextContainer` uses as the index for this context. It is also the value the parser stores in `SyntaxControlledRuntime::currentContext` while operating inside this context. Storing the ID inside the context as well as in the container index simplifies code that holds a `Context` reference and needs to know its ID without consulting the container.

5.5 Names: NameContainer and Name

Figures 5.6 and 5.7 specify the name layer of the runtime.

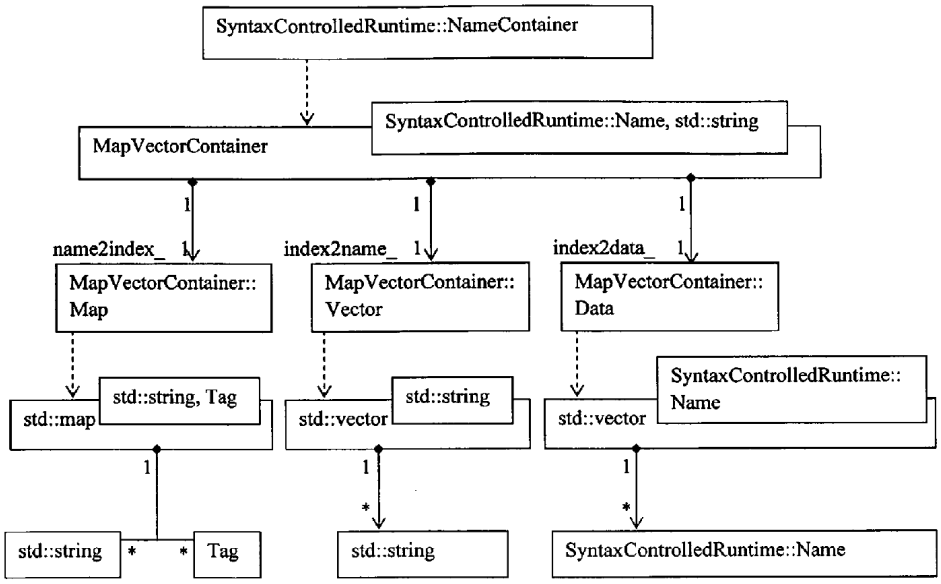

FIG. 12

Figure 5.6 — NameContainer (FIG 12 of US 8,464,232). NameContainer is an instantiation of MapVectorContainer with K = std::string and D = SyntaxControlledRuntime::Name. The map associates name strings with sequential indices; the vectors recover the original string and the Name data from the index.

Of the runtime's five `MapVectorContainer` instantiations, `NameContainer` is the one that uses `std::string` as its key type. The other four (one for contexts, one for symbols, one for rules, and one for the binary catalog discussed in Chapter 6) all use `Tag` as the key. Names are the only entities the runtime catalogs that originate as textual identifiers in source programs; everything else is an integer ID assigned at parse time.

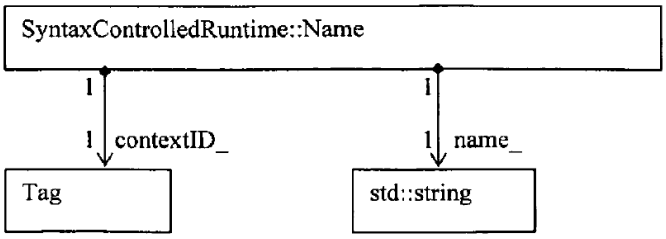

FIG. 13

Figure 5.7 — Name (FIG 13 of US 8,464,232). Name has two data members: contextID_ (a Tag identifying the context this name belongs to) and name_ (a std::string holding the name itself).

The `contextID_` member duplicates information already implicit in the surrounding `NameContainer` (which is itself owned by a particular `Context` whose ID is known). The redundancy is intentional: a `Name` instance, once retrieved, is self-describing — it knows which context it lives in without needing additional reference to its container. Code that accumulates `Name` references in working data structures (for example, a symbol table during semantic analysis) does not need to maintain context information separately.

5.6 Symbols: SymbolContainer and Symbol

Figures 5.8 and 5.9 specify the symbol layer.

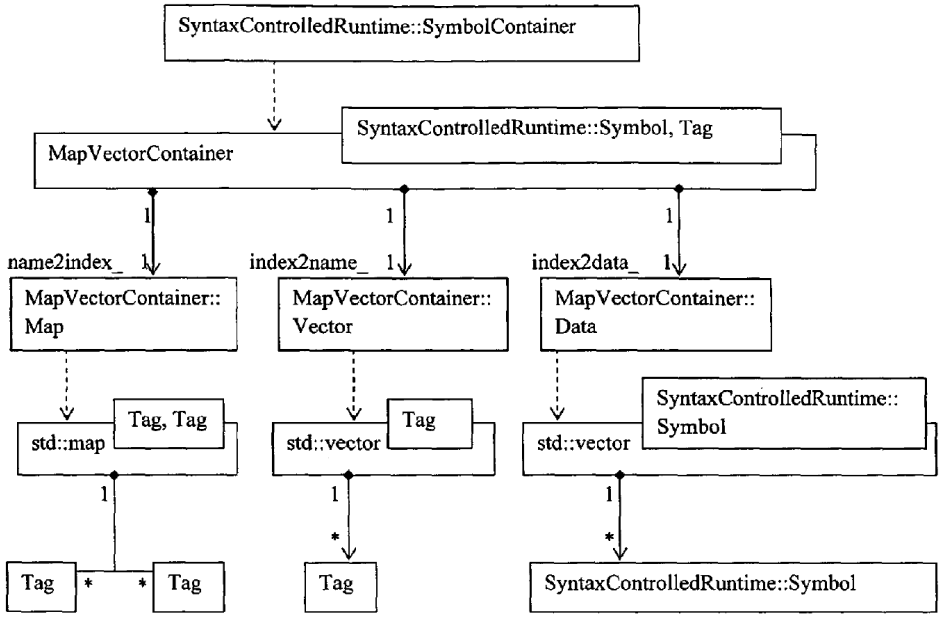

FIG. 14

Figure 5.8 — SymbolContainer (FIG 14 of US 8,464,232). SymbolContainer is an instantiation of MapVectorContainer with K = Tag and D = SyntaxControlledRuntime::Symbol. The map associates symbol IDs with sequential indices; the vectors recover the original Tag key and the Symbol data from the index.

Symbols are grammar non-terminals that the parser has *encountered during recognition* — not the symbol declarations from the grammar specification, but the run-time records of which non-terminals were exercised while parsing this particular source. For a grammar with 100 non-terminals defined in SGDL, a particular source file may exercise only 30 of them; the `SymbolContainer` for the parsed source's context contains 30 entries, not 100. The other 70 are present in `KeyWordsContainer` (which catalogs everything the grammar declared) but absent from the runtime's symbol container (which catalogs only what the parser actually recognized).

Like `ContextContainer`, `SymbolContainer` uses `Tag` as both key and index type: the symbol ID assigned by `KeyWordsContainer` is the natural identifier and is used directly.

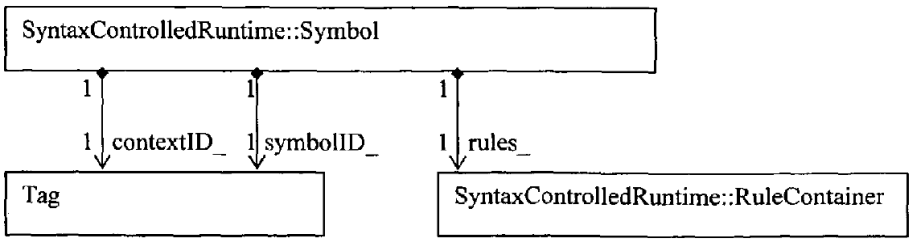

FIG. 15

Figure 5.9 — Symbol (FIG 15 of US 8,464,232). Symbol has three data members: contextID_, symbolID_, and rules_. The first two are Tag values identifying the context and the symbol within KeyWordsContainer; the third is a RuleContainer holding the Rule instances representing this symbol's recognition occurrences.

Each `Symbol` instance owns its own `RuleContainer`. A symbol that is recognized by the parser ten times during parsing of a source file produces a `Symbol` instance whose `rules_` container holds ten `Rule` instances. The `Rule` instances are the per-occurrence records: each one captures the specific tokens and sub-rule references that the parser matched at one particular point in the source text.

5.7 Rules: RuleContainer and Rule

Figures 5.10 and 5.11 specify the rule layer.

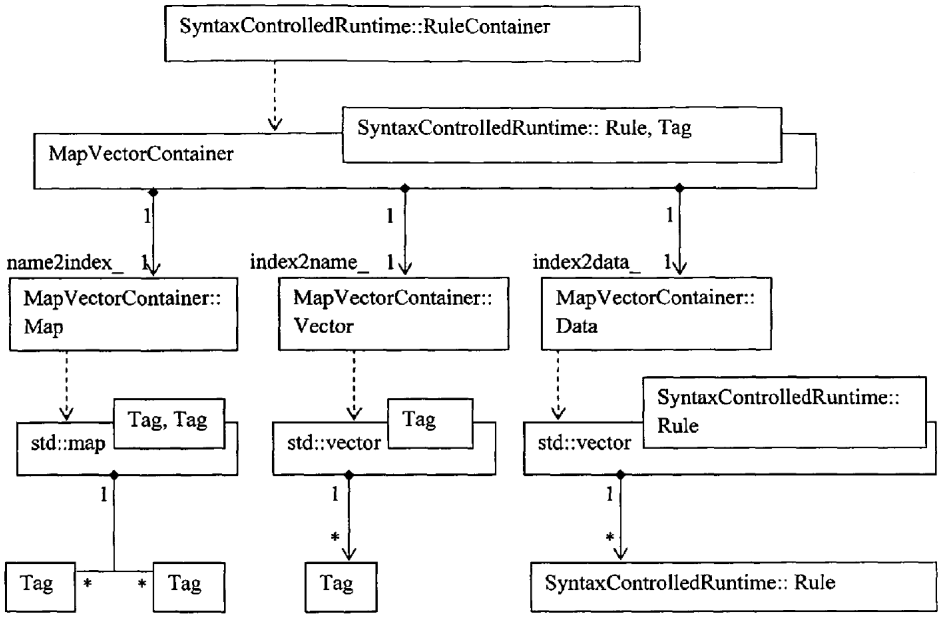

FIG. 16

Figure 5.10 — RuleContainer (FIG 16 of US 8,464,232). RuleContainer is an instantiation of MapVectorContainer with K = Tag and D = SyntaxControlledRuntime::Rule. The map associates rule instance IDs with sequential indices; the vectors recover the Tag key and the Rule data from the index.

The `Tag` key into `RuleContainer` is the `ruleInstanceID` that the parser assigns to each successful rule recognition. Recall from Section 4.7 that `TypeInstance` packs a `symbolID` and a `ruleID` into a single `Tag`-sized word; the `ruleID` field of that pack is what `RuleContainer` uses as its key. So a downstream consumer holding a `TypeInstance` can dereference it into a `Rule` instance by: looking up the symbol via `symbolID`, then looking up the rule within that symbol's `rules_` via `ruleID` — two constant-time operations.

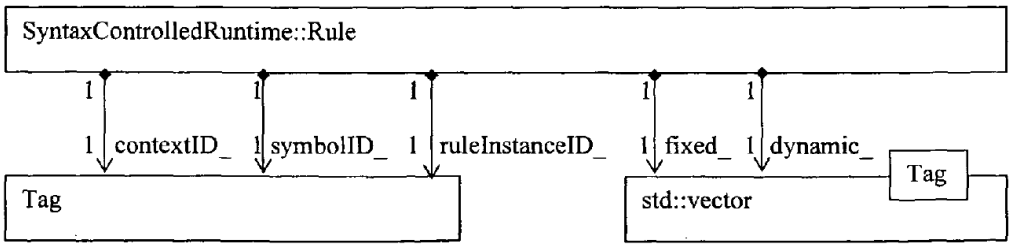

FIG. 17

Figure 5.11 — Rule (FIG 17 of US 8,464,232). Rule has five data members: three Tag values (contextID_, symbolID_, ruleInstanceID_) identifying which context, symbol, and instance this rule represents, and two std::vector<Tag> members (fixed_ and dynamic_) holding the parsed content of this rule recognition.

The two vectors `fixed_` and `dynamic_` carry the rule's data:

- `fixed_` — the values produced by terminals matched during this rule's recognition, plus any embedded action data. Each element is a `Tag` interpreted according to the rule's grammar definition. For a grammar element such as `integerToken`, the corresponding `Tag` carries the integer value that the tokenizer parsed; for a string literal, the `Tag` carries an index into the context's name container. Fixed values are the parser's record of *what it saw* — the source's data — at the points where the rule's right-hand side called for a terminal.
- `dynamic_` — references to other rules invoked during this rule's recognition. Each element is a `Tag` interpretable as a `TypeInstance` (Section 4.7) packing the `symbolID` and `ruleID` of the invoked rule. Dynamic values are the parser's record of *what it called* — the recursive structure of recognition — at the points where the rule's right-hand side referenced another non-terminal.

Together, `fixed_` and `dynamic_` carry exactly the information that a parse tree would carry, but in a flat indexed form rather than a recursive node-and-edge structure. Tree-shaped traversal is still possible — starting from the axiom's rule and following `dynamic_` references recursively — but it is not the runtime's primary access pattern. Indexed access by `symbolID` / `ruleID` is, and that access is uniformly constant-time.

5.8 The Runtime Syntax-Controlled API

The runtime exposes an application programming interface comprising the operations that generated parsers and semantic-processing code use to read and write the runtime. The patent enumerates the API as follows (*[col 13, lines 50-58]*, restated for the C++ reference implementation):

- **Create a new Context.** Returns the `contextID` (a `Tag`) of the new context. The context becomes the value of `SyntaxControlledRuntime::currentContext` if no other context is currently active; otherwise the parser is responsible for tracking the change.
- **Create a new Rule instance.** Given a `symbolID` identifying the non-terminal being recognized in the current context, the API returns a `Tag` that is actually a `TypeInstance` packing the `symbolID` and the newly assigned `ruleID`. The new rule instance is added to the context's `symbols_` container (allocating a new `Symbol` if this is the first occurrence of the symbol in this context).
- **Get a Symbol reference.** Given a `symbolID`, return a reference to the `Symbol` instance in the current context, allowing inspection of its `rules_` container.
- **Get a Rule reference.** Given a `Tag` interpretable as a `TypeInstance` (with packed `symbolID` and `ruleID`), return a reference to the corresponding `Rule` instance.
- **Modify a Rule's dynamic part.** Given a rule reference and a `std::vector<Tag>` representing the dynamic content, replace the rule's `dynamic_` member with the supplied vector. Used by the parser as it completes a rule's recognition and finalizes the references to invoked rules.
- **Modify a Rule's fixed part.** Given a rule reference and a `std::vector<Tag>` representing the fixed content, replace the rule's `fixed_` member.
- **Create or look up an identifier.** Given a string holding the textual identifier, return its index (`Tag`) within the current context's `NameContainer`. If the identifier was already present, the existing index is returned; otherwise a new `Name` instance is allocated.

Note what these operations **do not** include. There is no method to delete a rule, no method to reorder rules within a symbol, no method to merge contexts. The runtime is a write-once, append-only structure during parsing; once the parser completes, the runtime is read-only for downstream code. This monotonic discipline simplifies reasoning about runtime state and is the property that makes serialization to `SyntaxControlledBinary` (Chapter 6) a straightforward memory-copy operation rather than a full traversal.

5.9 Optimization Mode and Debug Flags

The two `int` members on `SyntaxControlledRuntime` — `optimizationMode_` and `debug_` — deserve a brief specification because their meaning is not self-evident from the class layout.

`optimizationMode_` takes one of a small set of values defined as enumeration constants by `SyntaxControlledRuntime`. The default value is zero, indicating no optimization: every name, symbol, and rule is recorded exactly as the parser delivers it. Non-zero values enable runtime-level optimizations such as canonicalizing duplicate names across contexts, de-duplicating identical rules, or compacting the `fixed_` and `dynamic_` vectors of small rules. The optimizations are lossless with respect to the round-trip property of Chapter 8 — a runtime built with optimization enabled de-compiles to identical source modulo indentation — but they may produce a binary form that differs in size or layout from the unoptimized form.

`debug_` takes the values 0 (off) or 1 (on). When on, every API operation that modifies the runtime emits a trace message through the runtime's `Logger` identifying the operation, its arguments, and the resulting state. The trace volume is large — a non-trivial parse can produce thousands of messages — and the flag is

intended for grammar developers debugging the interaction between a generated parser and the runtime, not for general use.

Chapter 6 — SyntaxControlledBinary

`SyntaxControlledBinary` is the serialized form of a successfully parsed program. Where `SyntaxControlledRuntime` (Chapter 5) holds the parsing artifact as an object graph in heap memory, `SyntaxControlledBinary` holds the same artifact as a single contiguous `vector<Tag>` — a flat tagged memory region with no embedded pointers, suitable for memory-mapped read-only access, file storage, and inter-process transmission. The two forms carry the same information; they differ only in physical representation. Phase 2.2a (runtime → binary) and Phase 2.2b (binary → runtime) are the morphisms between them.

This chapter specifies the binary in full: the outer class structure (Section 6.2), the layout of the in-memory tagged vector (Section 6.3), the BNF describing that layout precisely (Section 6.4), the auxiliary `BinaryHelper` indexing classes that provide constant-time access (Section 6.5), the binary syntax-controlled API (Section 6.6), and the file format used for persistence (Section 6.7).

6.1 Logical Equivalence to the Runtime

The logical view of the binary is identical to the logical view of the runtime presented in Figure 5.1. The binary owns a collection of contexts; each context owns a collection of names and a collection of symbols; each symbol owns a collection of rules; each rule has fixed and dynamic parts. The four-entity schema (Context, Name, Symbol, Rule) of Chapter 2 applies unchanged.

What differs is the physical realization. The runtime's containers are `std::map` and `std::vector` instances, holding heap-allocated entries linked by pointer. The binary's containers are *regions* of a single contiguous memory block, with offsets in place of pointers and length-prefix headers in place of container metadata. The differences and what they imply for use:

- **Single allocation.** The binary occupies one `std::vector<Tag>`. There is no per-entry heap allocation, no per-container `std::map` overhead. The footprint is therefore typically smaller than the runtime's, often by a factor of two or three for non-trivial parses.
- **Read-only after construction.** The binary is built once — by Phase 2.2a serializing a runtime — and is read-only thereafter. There is no API method that modifies a binary in place. To produce a modified binary, code must deserialize to a runtime, modify the runtime, and re-serialize. (Transformations of Chapter 12 are an exception: they operate on the binary directly through carefully scoped overlay operations rather than through the standard API.)
- **Position-independent layout.** All references inside the binary are byte offsets relative to the start of the memory block, not absolute pointers. The binary can be loaded at any address, mapped from a file, or transmitted between processes without relocation.
- **No external dependencies.** The binary contains complete copies of every name, symbol identifier, and rule reference. A binary on disk is self-contained; reading it requires no auxiliary data structures beyond the binary itself and a compatible `Tag` type definition (Section 4.7).

6.2 SyntaxControlledBinary Outer Class

Figure 6.1 shows the outer class.

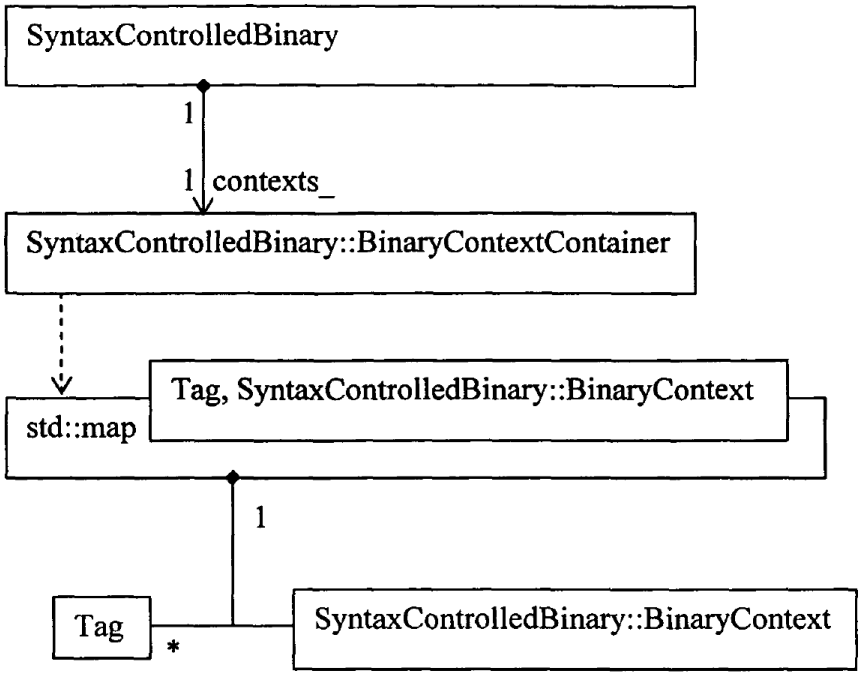

FIG. 19

Figure 6.1 — SyntaxControlledBinary outer class (FIG 19 of US 8,464,232). SyntaxControlledBinary owns a BinaryContextContainer (the contexts_ member), implemented as a std::map from Tag to BinaryContext.

Like its runtime counterpart, `SyntaxControlledBinary` is a thin outer class whose principal member is a container holding the actual contexts. The container type, `SyntaxControlledBinary::BinaryContextContainer`, is a `std::map<Tag, BinaryContext>` — simpler than the runtime's `MapVectorContainer<Context, Tag>` because the binary form does not need the by-name and by-index dual-access pattern. Each binary context is keyed only by its `contextID`; the dual indexing for names lives inside the context (Section 6.5) rather than at the top level.

6.3 BinaryContext and the Tagged Vector Memory

The substance of the binary is the `BinaryContext` inner class. Figure 6.2 shows it.

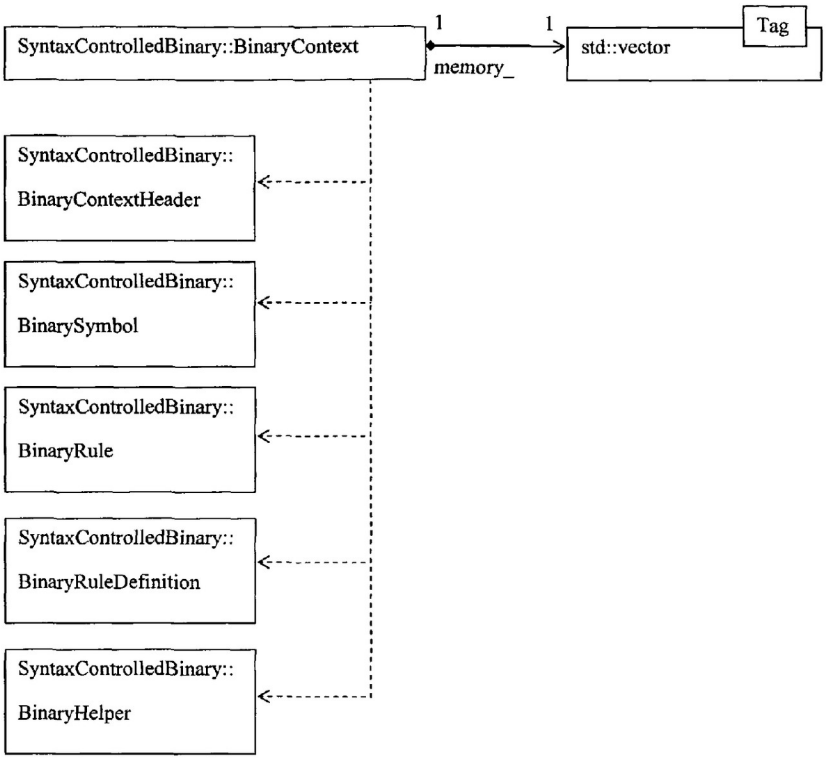

FIG. 20

Figure 6.2 — BinaryContext (FIG 20 of US 8,464,232). BinaryContext owns a single std::vector<Tag> (the memory_ member). The structure types BinaryContextHeader, BinarySymbol, BinaryRule, BinaryRuleDefinition, and BinaryHelper are overlays that interpret regions of memory_; they are not separately allocated.

The single data member, `memory_`, is the entirety of the binary's state for one context. It is a `std::vector<Tag>` — a flat array of tag-sized words — whose contents follow the BNF specified in Section 6.4. The five structure types attached to `BinaryContext` by dashed-arrow associations in Figure 6.2 (the patent's notation for "is overlaid by") are not separately allocated objects: they are C++ struct types whose members are reference-typed and whose constructors take a pointer to a position within `memory_` as their argument. Constructing one of these structures is a constant-time operation that produces a typed view onto the underlying tag vector; destroying it does nothing because there is nothing to deallocate.

The five overlay types and their purposes:

- `BinaryContextHeader` — the fixed-size header at the start of `memory_` giving the sizes and offsets of all the regions that follow. Specified in Section 6.4.1.
- `BinarySymbol` — the per-symbol record holding the count of recognized rule instances and the offset to those instances within `memory_`. Specified in Section 6.4.3.
- `BinaryRule` — the per-rule record holding the sizes of the rule's fixed and dynamic parts and the offset to its definition. Specified in Section 6.4.4.

- `BinaryRuleDefinition` — the actual fixed and dynamic data of a recognized rule. Specified in Section 6.4.5.
- `BinaryHelper` — an auxiliary class providing constant-time index lookup over the binary by precomputing offset tables. Specified in Section 6.5.

6.4 The Binary Structure

The contents of `memory_` follow a strict layout that is best stated as a BNF over `Tag`-sized words. Each non-terminal below denotes a contiguous region of `memory_` beginning at some offset; each terminal denotes a single `Tag` word. The complete BNF for one context's `memory_` vector is:

```
binaryContext      ::= header sequentialNames alphaNames symbols

header             ::= size axiom numberOfNames startOfNames
                       numberOfSymbols startOfSymbols

sequentialNames    ::= { name }
name               ::= { Tag }                  -- packed string

alphaNames         ::= { nameIndex }
nameIndex          ::= Tag                      -- offset into
                                                   sequentialNames

symbols            ::= { binarySymbol }
binarySymbol       ::= numberOfRuleInstances
                       symbolID
                       symbolStart
                       { binaryRule }

binaryRule         ::= fixedSize
                       dynamicSize
                       ruleStart
                       binaryRuleDefinition

binaryRuleDefinition ::= fixed dynamic
fixed              ::= { Tag }
dynamic            ::= { TypeInstance }
```

Each non-terminal occupying two or more lines in the listing is a structure with named `Tag` fields; each non-terminal whose right-hand side is `{ Tag }` is a length-prefixed array of `Tag` words. The remainder of this section walks through the structure top-down, with the patent's figures of each component.

6.4.1 BinaryContextHeader

The header is the first six tag-sized words of `memory_` and gives every other position in the binary as an offset relative to the start of `memory_` or relative to the previous region. Figure 6.3 shows its structure.

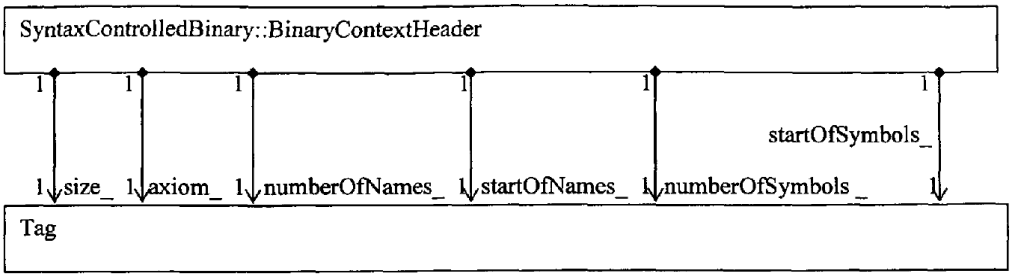

FIG. 21

Figure 6.3 — BinaryContextHeader (FIG 21 of US 8,464,232). The header has six Tag-typed fields: size_, axiom_, numberOfNames_, startOfNames_, numberOfSymbols_, startOfSymbols_.

The six fields:

- `size_` — the total size of the binary in `Tag` words, including the header itself. A reader knows from this field alone how many bytes constitute the entire binary, which is essential when reading from a stream of length-prefixed binaries packed into one file.
- `axiom_` — the `symbolID` of the grammar's axiom (the entry-point non-terminal). When de-compilation begins, it begins by locating this symbol's rule and following references from there.
- `numberOfNames_` — the count of distinct names declared in this context. Used by `BinaryHelper` (Section 6.5) to size the name index tables.
- `startOfNames_` — the offset, in `Tag` words from the start of `memory_`, at which the sequential names region begins. Equivalently, the offset just after the header (since names follow the header immediately in the standard layout).
- `numberOfSymbols_` — the count of distinct symbols recognized by the parser within this context.
- `startOfSymbols_` — the offset at which the symbols region begins. Equivalent to `startOfNames_` plus the total size of the names region.

6.4.2 Names: Sequential and Alphabetical Catalogs

Names occupy the region between `startOfNames_` and `startOfSymbols_` within `memory_`. The region is internally divided into two sub-regions:

- **Sequential names.** Each name is stored as a length-prefixed sequence of `Tag`-words holding its UTF-8 (or platform-character) bytes packed into the words. Names appear in the order they were inserted into the runtime's `NameContainer` — the same order in which the parser encountered them.
- **Alphabetical name index.** A vector of `Tag` offsets, one per name, sorted alphabetically by the name strings the offsets point to. Lookup by name is therefore O(log N): binary search the alphabetical index, dereference to the sequential names region for string comparison at each search step.

The dual representation reproduces the runtime's `NameContainer` (Section 5.5) functionality — lookup by name and lookup by index — in flat memory. The original-key recovery is the dereference `sequentialNames[index]`; lookup by name is binary search through `alphaNames` followed by string comparison.

6.4.3 BinarySymbol

Each entry in the symbols region is a `BinarySymbol` record. Figure 6.4 shows its structure.

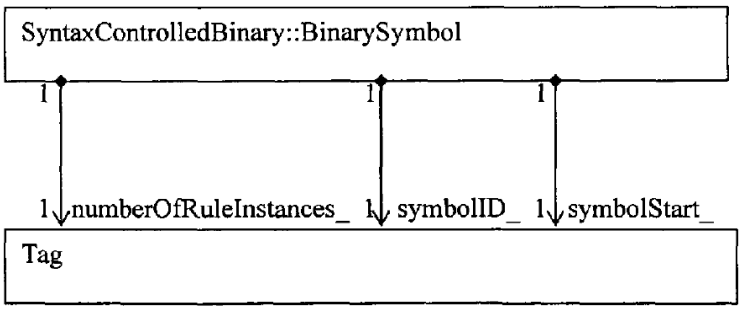

FIG. 22

Figure 6.4 — BinarySymbol (FIG 22 of US 8,464,232). BinarySymbol has three Tag-typed fields: numberOfRuleInstances_, symbolID_, symbolStart_.

- `numberOfRuleInstances_` — the count of rule instances (recognition occurrences) of this symbol within the context.
- `symbolID_` — the symbol's identifier within the grammar's `KeyWordsContainer` (Section 4.2). The same ID that the runtime's `Symbol` instance carries (Section 5.6).
- `symbolStart_` — the offset within `memory_` at which the run of `BinaryRule` records for this symbol's recognized instances begins.

A `BinarySymbol` record is therefore three `Tag` words. Constant-time access to the rules of a particular symbol involves: index into the symbols region by `symbolID` (with the help of `BinaryHelper`), read the three-word record, and use `symbolStart_` to dereference into the rule region.

6.4.4 BinaryRule

Each rule instance is stored as a `BinaryRule` record. Figure 6.5 shows its structure.

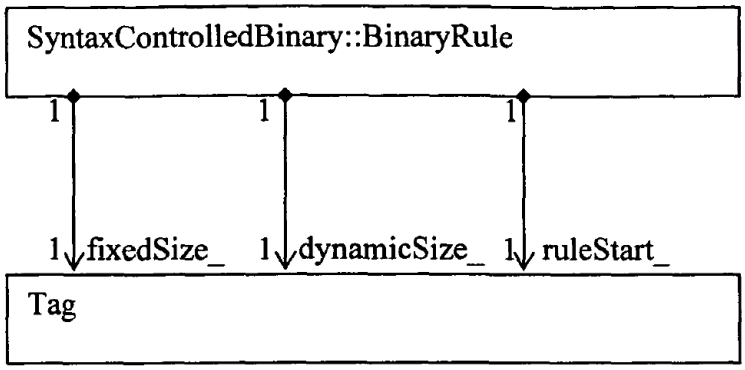

FIG. 23

Figure 6.5 — BinaryRule (FIG 23 of US 8,464,232). BinaryRule has three Tag-typed fields: fixedSize_, dynamicSize_, ruleStart_.

- `fixedSize_` — the count of `Tag` words in this rule's fixed part.
- `dynamicSize_` — the count of `TypeInstance` words in this rule's dynamic part. (Each `TypeInstance` is a single tag-sized word per Section 4.7, so `dynamicSize_` is also the count of words in the dynamic part — the type distinction is interpretive, not a size difference.)
- `ruleStart_` — the offset within `memory_` at which this rule's `BinaryRuleDefinition` begins.

6.4.5 BinaryRuleDefinition

The actual content of a recognized rule — what tokens it matched and what other rules it invoked — lives in `BinaryRuleDefinition`. Figure 6.6 shows the structure.

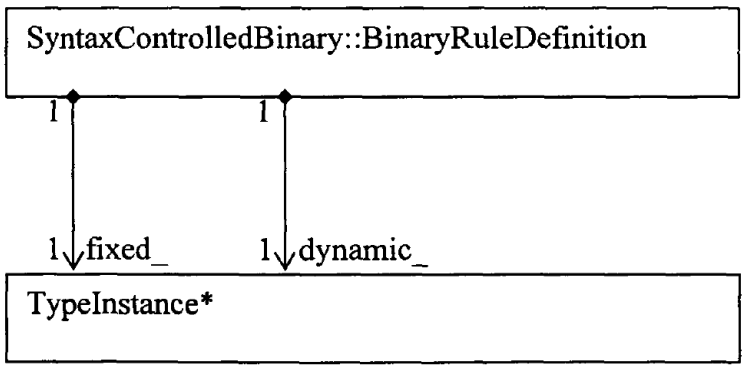

FIG. 24

Figure 6.6 — BinaryRuleDefinition (FIG 24 of US 8,464,232). BinaryRuleDefinition has two pointer-typed fields: fixed_ and dynamic_, each pointing to a region of TypeInstance words within memory_.

A `BinaryRuleDefinition` is a transient overlay constructed when a reader needs to access a rule's data. Its `fixed_` field points to the rule's fixed region and is `fixedSize_` words long; its `dynamic_` field points

to the rule's dynamic region and is `dynamicSize_` words long. Both regions immediately follow the `BinaryRule` record at offset `ruleStart_`: the fixed region first, the dynamic region after it.

Note that `BinaryRuleDefinition::fixed_` and `::dynamic_` are typed as `TypeInstance*` in Figure 6.6 — not as `Tag*`. The patent's convention is that every Tag-sized word is potentially interpretable as a `TypeInstance` (Section 4.7), and the type signature reflects this. Code reading the fixed region treats each word as a plain `Tag`; code reading the dynamic region treats each word as a `TypeInstance` with packed `symbolID` and `ruleID` fields. Figure 6.7 shows the `TypeInstance` layout for reference.

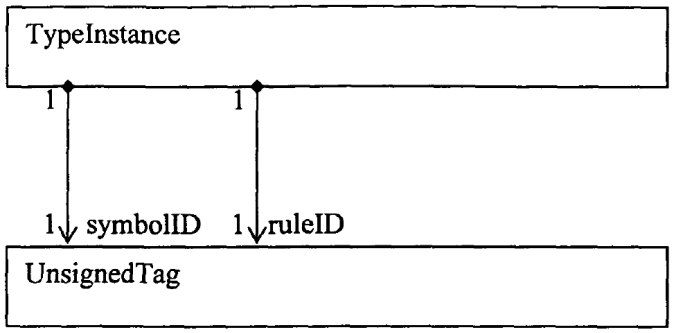

FIG. 25

Figure 6.7 — TypeInstance (FIG 25 of US 8,464,232). TypeInstance has two UnsignedTag-typed bit-fields: symbolID and ruleID, packed into a single tag-sized word as specified in Section 4.7.

6.5 BinaryHelper

Walking the binary by following offsets word-by-word is correct but inefficient for repeated access. `BinaryHelper` is the auxiliary class that precomputes index tables giving constant-time access to any name, symbol, or rule by ID. It is built once per binary, on demand, and is held alongside the binary by the code that operates on it. Figure 6.8 shows its structure.

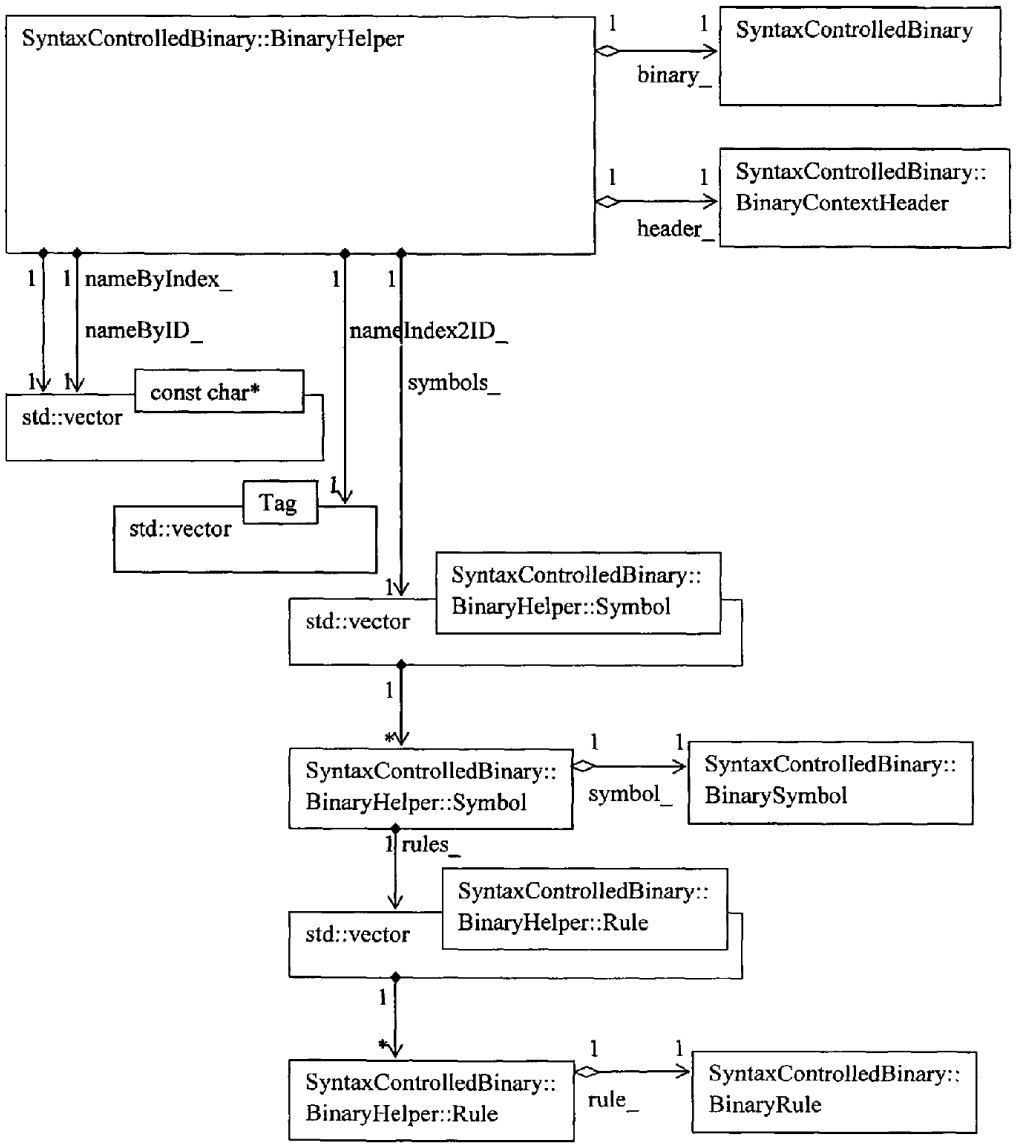

FIG. 26

Figure 6.8 — BinaryHelper (FIG 26 of US 8,464,232). BinaryHelper references the SyntaxControlledBinary it indexes (binary_) and the BinaryContextHeader at the start of memory_ (header_). It owns four lookup tables: nameByIndex_ (a vector<const char> indexing names in sequential order), nameByID_ (the alphabetical index), nameIndex2ID_ (a vector<Tag> mapping sequential index to alphabetical ID), and symbols_ (a vector<BinaryHelper::Symbol> with one entry per symbol). Each Symbol entry holds a pointer to its BinarySymbol overlay (symbol_) and a vector<BinaryHelper::Rule> (rules_), each of which holds a pointer to its BinaryRule overlay.*

The four top-level members of `BinaryHelper`:

- `nameByIndex_` — a `vector<const char*>` indexed by the name's sequential position in the binary. Element `i` is a pointer into `memory_` at the start of the `i`th name's character data. Lookup by sequential index is therefore one indexed read; the result is a `const char*` pointing into the binary itself — no copy.

- `nameByID_` — a `vector<const char*>` indexed by the alphabetical ID, which is the position the name occupies in the alphabetically-sorted `alphaNames` region. Element `id` is a pointer to the same character data that `nameByIndex_[seq(id)]` would yield, where `seq` is the inverse of the alphabetical sort.
- `nameIndex2ID_` — a `vector<Tag>` giving the inverse of the previous map: from sequential index to alphabetical ID. Used by code that has a sequential index in hand and needs the alphabetical ID for stable cross-binary references.
- `symbols_` — a `vector<BinaryHelper::Symbol>`, one entry per recognized symbol in the context. Each `Symbol` entry holds the offset of its `BinarySymbol` record (in `symbol_`) and a vector of `Rule` entries (in `rules_`), one per recognized rule instance of that symbol, each holding the offset of its `BinaryRule` record.

Constructing a `BinaryHelper` is an O(N + M + R) operation where N is the number of names, M is the number of symbols, and R is the total number of rule instances across all symbols. After construction, every read access to the binary is O(1).

6.6 The Binary Syntax-Controlled API

The binary exposes the same logical operations as the runtime API of Section 5.8, but only the read operations — the binary is not modifiable in place. The patent enumerates the binary API as the set of constant-time accessors built atop `BinaryHelper` (*[col 14, lines 30-50]*, restated for the C++ reference implementation):

- **Get a Name.** Given a sequential name index or an alphabetical ID, return a pointer to the name's character data within `memory_`. No allocation; constant-time.
- **Get a Symbol overlay.** Given a `symbolID`, return a `BinarySymbol` overlay positioned at the symbol's record. The overlay's methods read `numberOfRuleInstances_`, `symbolID_`, and the offset to the rule region.
- **Get a Rule overlay.** Given a `TypeInstance` (with packed `symbolID` and `ruleID`), return a `BinaryRule` overlay. Two-step constant-time lookup: dereference `symbolID` through `symbols_`, then dereference `ruleID` through that symbol's `rules_`.
- **Get a Rule's fixed part.** Given a `BinaryRule` overlay, return a `BinaryRuleDefinition` overlay whose `fixed_` field points to the start of the fixed region.
- **Get a Rule's dynamic part.** Same as above for `dynamic_`.
- **Iterate symbols within a context.** Provided as a forward iterator over `symbols_` yielding `BinarySymbol` overlays.
- **Iterate rule instances within a symbol.** Provided as a forward iterator over a particular symbol's `rules_` yielding `BinaryRule` overlays.

Notice the absence of any "create" or "modify" operations. A binary is read-only after construction. Code that needs to alter the parsed artifact must either deserialize to a runtime (Phase 2.2b), modify the runtime through the API of Section 5.8, and re-serialize (Phase 2.2a); or it must apply a transformation algorithm of Chapter 12 that produces a new binary from an existing one without going through a runtime.

6.7 File Input and Output

A `SyntaxControlledBinary` on disk is the contents of `memory_` written byte-for-byte to a file. There is no envelope format, no checksum, no compression: the bytes of the binary in memory *are* the bytes of the binary on disk. The file extension `.fgr` (for *foreign grammar* — the binary form being foreign to the source language) identifies a CCS binary file.

Writing a binary is therefore a single `std::ofstream::write` call passing `memory_`'s `data()` pointer and its `size() * sizeof(Tag)` byte count. Reading a binary is correspondingly the inverse: open the file, read its byte size to determine the vector capacity required, allocate the vector, and read the bytes directly into it. The reader's only post-processing step is to overlay a `BinaryContextHeader` on the start of `memory_` and read its `size_` field to validate that the byte count read matches the header's declaration.

Cross-platform considerations apply when transferring a binary between machines with different endianness or different `Tag` word sizes. The binary format does not include byte-order marks or word-size headers; reading machinery must know in advance whether to byte-swap and how to interpret the `TypeInstance` bit-field layouts. Chapter 11 specifies the conventions for cross-platform binary exchange and the file naming or wrapping that should be used to disambiguate.

Chapter 7 — Generator

The generator subsystem performs Phase 2.3 of compilation: it consumes a `SyntaxControlledRuntime` produced by the parser and emits the source files that, when compiled by the host C++ toolchain, become the executable target compiler. The subsystem comprises the `Generator` class itself (Section 7.2), a pair of cooperating sub-generators — `GeneratorBinary` for the binary-form output and `GeneratorRuntime` for the runtime-form output (Section 7.3) — a hierarchy of `Style` classes that abstract over per-target-language emission patterns (Sections 7.4 and 7.5), and a family of grammar-specific generator children that bind the abstract pieces to particular grammars (Section 7.6).

The generator is the only subsystem that the developer normally extends when targeting CCS at a new language. The runtime, the binary, the parser front end, and the management hierarchy operate uniformly across all grammars; a new grammar produces a new style class plus a new generator child. Section 7.7 specifies the seven files the generator produces per grammar, the contract each file satisfies, and where the developer's Phase 2.4 contribution fits into the result.

7.1 Position in the Compilation Phases

Phase 2.3 sits between the parser and the developer-implemented semantic processing of Phase 2.4. Its inputs are: the `SyntaxControlledRuntime` the parser produced (or, equivalently, the `SyntaxControlledBinary` if invoked from a binary), the grammar's `KeyWordsContainer`, and the parser instance itself (used for symbol-ID lookup during emission). Its outputs are seven C++ source files specified in Section 7.7. Phase 2.3 is fully automatic and grammar-driven: every rule in the grammar produces its share of generated code by the same emission template, and there is no developer involvement at this phase.

Phase 2.4, by contrast, is entirely the developer's. The seven files Phase 2.3 produces include implementation stubs for the semantic operations the target language requires; Phase 2.4 is the editing of those stubs to provide actual semantic content. Phase 2.5 then compiles the result with the host C++ toolchain to produce the final target compiler executable.

7.2 Generator

Figure 7.1 shows the `Generator` class.

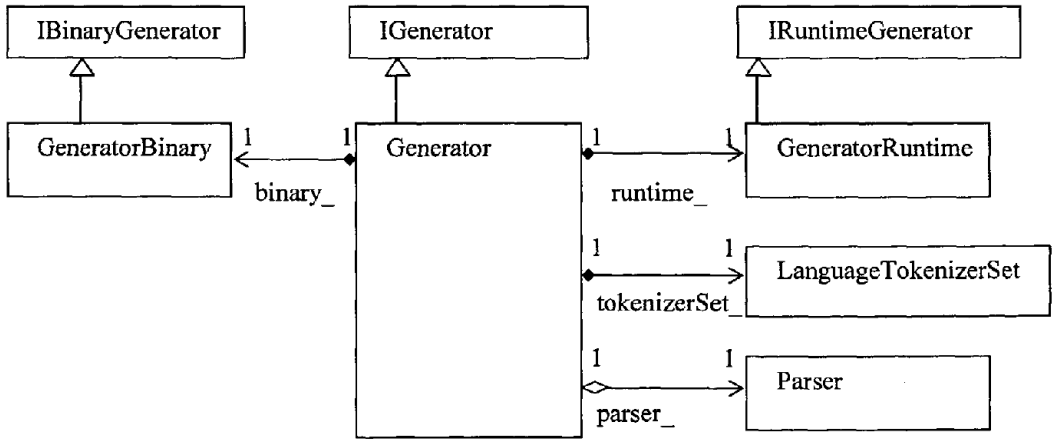

FIG. 32

Figure 7.1 — Generator and its collaborators (FIG 32 of US 8,464,232). Generator derives from IGenerator, owns a GeneratorBinary (binary_) and a GeneratorRuntime (runtime_), and references a LanguageTokenizerSet and a Parser.

`Generator` is the orchestrator. It does not itself emit code; it owns the two specialized sub-generators that do (the `binary_` and `runtime_` members) and delegates emission to them. It also holds references to the `LanguageTokenizerSet` (the enumeration value identifying which language-specific tokenizer the generated code should instantiate) and the `Parser` (used for symbol-name lookup when emitting cross-references between generated artifacts).

The split between `GeneratorBinary` and `GeneratorRuntime` is a separation of concerns. `GeneratorBinary` emits the source files responsible for serializing parsed programs into the on-disk binary form; `GeneratorRuntime` emits the source files responsible for de-compiling the in-memory runtime form back to source text. Both sub-generators consume the same `SyntaxControlledRuntime` as input (the meta-runtime, when CCS is being bootstrapped; the user grammar's runtime when targeting a user grammar), but they emit files with different responsibilities.

The `IGenerator` base class is the abstract interface. It declares pure-virtual methods including `generate` (the top-level emission entry point invoked by `Compiler::run`), `getRuntime` and `getBinary` (accessors for the two sub-generators), and `setStyle` (used by `Generator` subclasses to install the appropriate style class for the grammar being processed). The default implementations provided by `Generator` itself are sufficient for grammars that emit standard CCS-flavored C++; subclasses override them for grammars whose target output is structurally different (for example, the XML-emitting subclass of Section 7.6).

7.3 GeneratorBinary and GeneratorRuntime

Figure 7.2 shows `GeneratorBinary` and its style hierarchy.

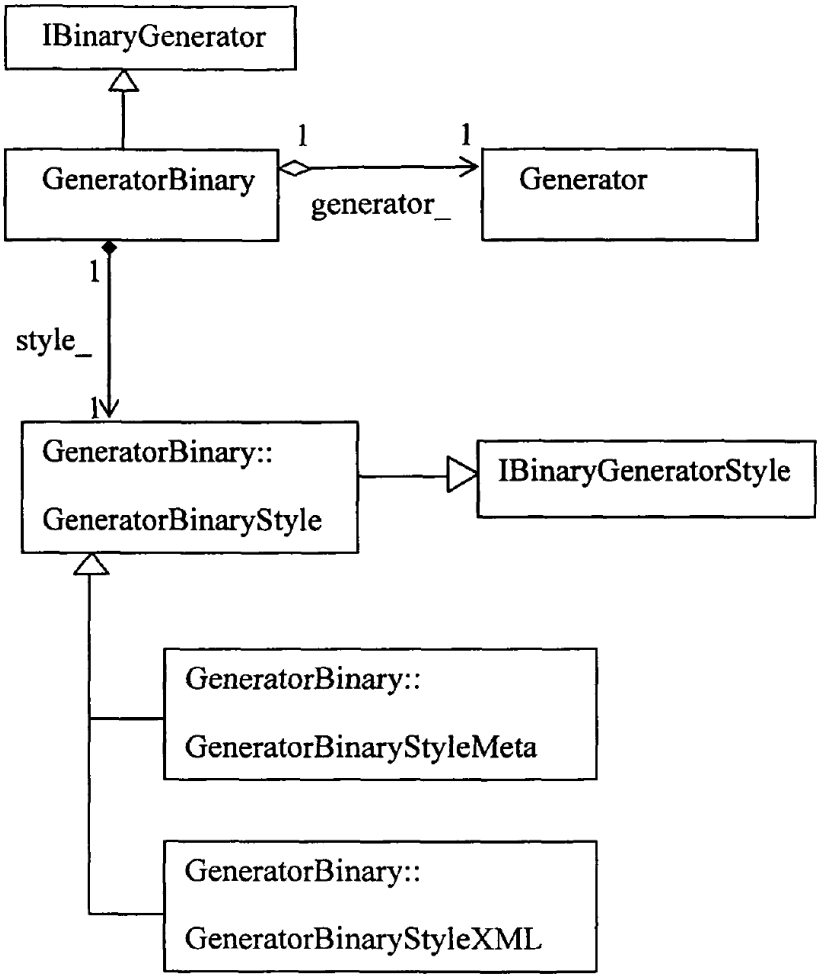

FIG. 33

Figure 7.2 — GeneratorBinary and its style hierarchy (FIG 33 of US 8,464,232). GeneratorBinary derives from IBinaryGenerator and references its parent Generator. It owns a GeneratorBinary::GeneratorBinaryStyle (the style_ member), which in turn derives from IBinaryGeneratorStyle. Two concrete style subclasses are shipped: GeneratorBinaryStyleMeta and GeneratorBinaryStyleXML.

`GeneratorBinary` produces the source files that read and write `SyntaxControlledBinary` for the target grammar. Its main responsibility is emitting the `<Prefix>Generator.cc` source file that, when compiled into the target compiler, knows how to serialize and de-serialize binaries for grammars built on the target language. The actual emission patterns — what C++ code to write for what grammar construct — vary by target language and are encoded in the `GeneratorBinaryStyle` subclass that `GeneratorBinary` is configured with.

Figure 7.3 shows the parallel structure for runtime emission.

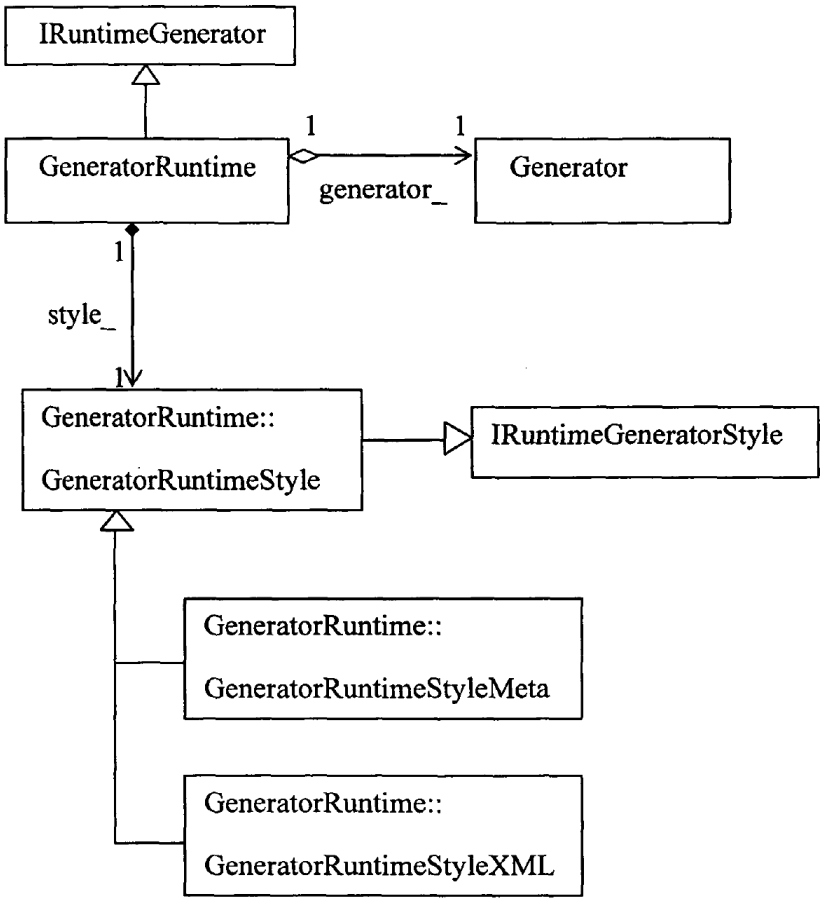

FIG. 34

Figure 7.3 — GeneratorRuntime and its style hierarchy (FIG 34 of US 8,464,232). The structure mirrors GeneratorBinary: GeneratorRuntime derives from IRuntimeGenerator and references its parent Generator. It owns a GeneratorRuntime::GeneratorRuntimeStyle (the style_ member), with concrete subclasses GeneratorRuntimeStyleMeta and GeneratorRuntimeStyleXML.

`GeneratorRuntime` produces the source files that traverse `SyntaxControlledRuntime` for de-compilation. Its principal output is the `<Prefix>DeCompile` implementation in `<Prefix>Generator.cc` (the same file that `GeneratorBinary` contributes to). The split of responsibility is: `GeneratorBinary` contributes binary I/O code; `GeneratorRuntime` contributes de-compilation code; the two streams are interleaved into a single output file by the `Generator` parent.

The deliberate parallelism between Figures 7.2 and 7.3 is the reason the abstraction exists. Both binary I/O and de-compilation traverse the four-entity schema of Chapter 5 (Context, Name, Symbol, Rule); both produce per-rule emission code; both vary by target language only in the surface pattern of what to emit at each schema visitation point. Encoding the per-target variation in `Style` subclasses lets the variation be small and localized. A new target language requires writing one `GeneratorBinaryStyle` subclass and one `GeneratorRuntimeStyle` subclass; the rest of the generator subsystem applies unchanged.

7.4 The Style Hierarchy

Each `Style` subclass implements a target-language-specific code-emission strategy. The base classes `IBinaryGeneratorStyle` and `IRuntimeGeneratorStyle` declare pure-virtual methods that correspond to the emission points the generator visits during traversal of the runtime: `emitContextStart`, `emitContextEnd`, `emitName`, `emitSymbolStart`, `emitSymbolEnd`, `emitRuleStart`, `emitRuleFixed`, `emitRuleDynamic`, `emitRuleEnd`, and so on. Concrete subclasses override these to produce the code their target language requires at each point.

The four concrete style subclasses shipped with the reference implementation:

- `GeneratorBinaryStyleMeta` — the binary-form emission strategy for SGDL itself. Used during the bootstrap loop of Chapter 2: when the CCS executable processes `meta.sgr`, this is the style class that `GeneratorBinary` instantiates.
- `GeneratorBinaryStyleXML` — the binary-form emission strategy for XML grammars. Used when the target compiler being built is an XML-aware compiler.
- `GeneratorRuntimeStyleMeta` — the runtime-form emission strategy for SGDL. Its `emitRuleFixed` and other methods produce the de-compilation code that re-renders an SGDL grammar from its runtime form back to text.
- `GeneratorRuntimeStyleXML` — the runtime-form emission strategy for XML, including XML-specific concerns such as attribute reordering and entity-reference reconstruction.

Adding a new target language is the procedure of writing two new style subclasses (one binary, one runtime) plus a generator-child that selects them. Chapter 10 walks through this procedure in detail for a worked example.

7.5 Indentation and Round-Trip Identity

Section 3.7.3 noted that the round-trip property holds modulo indentation: de-compiled source text is identical to the original modulo whitespace placement, and the indentation policy is determined by the runtime-form style class active at the time of emission. This is now precise: the policy is implemented by `GeneratorRuntimeStyle::emit*` methods, with each style subclass making different whitespace choices.

`GeneratorRuntimeStyleMeta` implements an indentation convention suitable for SGDL grammars: opening parentheses align to the previous nesting level, rule definitions align two spaces inside the grammar’s outer parenthesis, and right-hand-side elements may be split onto multiple lines for readability. `GeneratorRuntimeStyleXML` implements an indentation convention suitable for XML: tags align to their nesting depth in the document tree, attributes appear on the tag line in source order, and content text is preserved character-by-character.

A consequence: round-trip identity for a particular grammar depends on which style subclass the de-compilation invokes. Compiling an SGDL grammar to runtime, then de-compiling with `GeneratorRuntimeStyleMeta`, then comparing against the original, will succeed (the de-compiler’s indentation choices are stable across invocations). Compiling the same grammar and de-compiling with a different style — should one ever be authored for a "compact" SGDL emission, for example — would not

produce identical output. The empirical demonstration of round-trip identity in Chapter 8 is therefore implicitly parameterized by the style subclasses in use; that parameterization is normal and is not a caveat in the round-trip property itself.

7.6 Target-Specific Generator Children

Figure 7.4 shows the generator hierarchy as it appears in the reference implementation, with the concrete grammar-specific subclasses below the base classes.

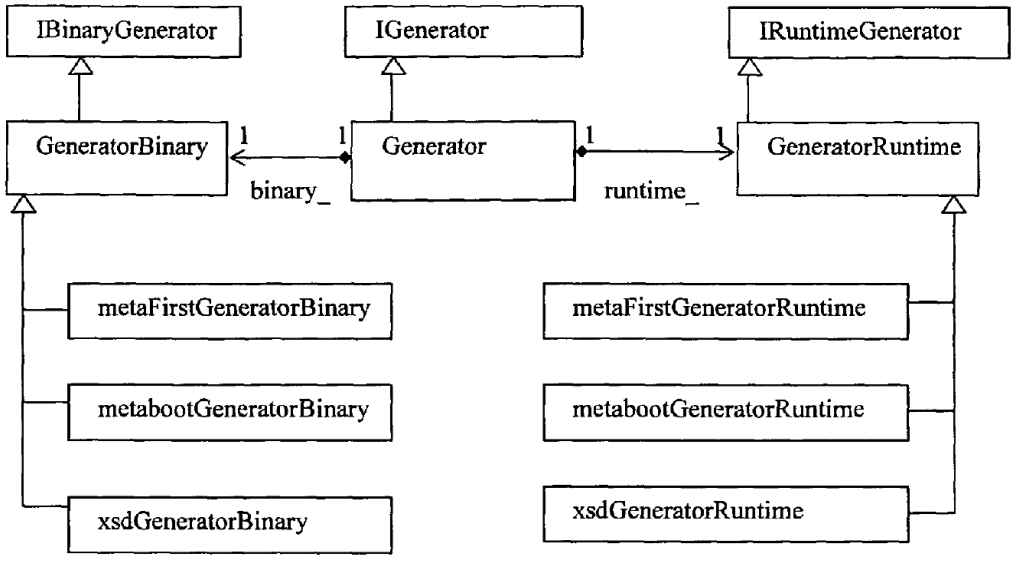

FIG. 35

Figure 7.4 — Concrete generator subclasses (FIG 35 of US 8,464,232). On the left, GeneratorBinary has three concrete subclasses: metaFirstGeneratorBinary, metabootGeneratorBinary, and xsdGeneratorBinary. On the right, GeneratorRuntime has the parallel three: metaFirstGeneratorRuntime, metabootGeneratorRuntime, and xsdGeneratorRuntime.

The three pairs of subclasses correspond to the three target compilers the reference implementation builds during a full self-hosting cycle:

- **metaFirst*** — the very first instantiation of the generator pair, used when CCS is built from sources with no pre-existing CCS executable available. The `metaFirstGeneratorBinary` and `metaFirstGeneratorRuntime` classes contain hand-authored code that handles the SGDL grammar without depending on any auto-generated artifact. This is the "stage-zero" compiler that the bootstrap chain begins with.
- **metaboot*** — the second-stage instantiation, used after `metaFirst` has produced the meta-grammar's artifacts. `metabootGeneratorBinary` and `metabootGeneratorRuntime` implement the same emission strategies as their `metaFirst` counterparts but consume their inputs through the auto-generated parsing infrastructure rather than hand-authored equivalents. The fact that their output matches `metaFirst`'s output (modulo whitespace) is the closure of the bootstrap loop.

- **xsd*** — the example application target: a compiler for XML schemas described by XSD. `xsdGeneratorBinary` and `xsdGeneratorRuntime` are concrete grammar-specific subclasses that demonstrate how a developer extends the generator hierarchy to target a new language. They are also the working example referenced in Chapter 10.

Notice the symmetry of Figure 7.4: every `GeneratorBinary` subclass has a matching `GeneratorRuntime` subclass with the same name prefix. This is by convention rather than enforcement; the generator-pair selection logic in `Compiler` assumes the convention holds and would fail to find one of the pair if the convention were violated. A new grammar added to the reference implementation contributes a matching pair.

7.7 Generated Files

A successful Phase 2.3 invocation against a grammar named `xyz` produces seven files in the output directory (*[whitepaper §8.5]*):

- `xyzParser.h` — the header for the grammar-specific parser subclass. Declares the recursive-descent procedures, one per non-terminal in the grammar.
- `xyzParser.cc` — the implementation of the parser declared in `xyzParser.h`. Contains the body of each recursive-descent procedure, with each procedure's body derived from the corresponding right-hand side in the grammar specification. The action macros embedded in the grammar appear here as inline code, in the positions the grammar placed them.
- `xyzGenerator.h` — the header for the grammar-specific generator subclass. Declares the binary-emission and runtime-emission methods that target compilers will invoke when serializing or de-compiling parsed `xyz` programs.
- `xyzGenerator.cc` — the implementation of the generator. Combines the contributions of `GeneratorBinary` and `GeneratorRuntime` (Section 7.3) into a single file.
- `xyzKeyWordDefinition.h` — the header declaring the keyword and non-terminal name constants for the `xyz` grammar. Each non-terminal in the grammar receives a named integer constant whose value is its symbol ID; semantic-processing code references symbols by name through these constants rather than by raw integer.
- `xyzKeyWordDefinition.cc` — the implementation of the keyword definition table. Populates the constants declared in the header.
- `xyzMakeGenerators.cc` — a stub source file containing the `makeGeneratorBinary` and `makeGeneratorRuntime` factory functions that `Compiler` calls when instantiating the generator pair for the `xyz` grammar. Two-line file in most cases; exists separately so that target build systems can re-link without touching the larger generator implementation.

Of these seven files, six are fully generated and should not be edited: any change is liable to be overwritten the next time Phase 2.3 runs. The seventh — `xyzGenerator.cc` — contains the generated emission code, but it is also the file in which the developer's Phase 2.4 contribution lives. The convention is that the generator emits implementation stubs marked with comment delimiters; the developer fills the stubs with target-specific semantic content; subsequent regenerations of `xyzGenerator.cc` preserve the developer's additions by re-injecting them between the same comment delimiters. The mechanics of this preservation

are part of the action-macro system specified in Section 3.5; the developer-visible result is that `xyzGenerator.cc` is regenerable but the developer's edits are not lost.

7.8 Default Empty-Implementation Generator

Phase 2.3's output is fully buildable as soon as it is generated. The semantic stubs in `xyzGenerator.cc` have empty bodies (or, for stubs whose return type is non-void, return a default-constructed value). A target compiler built with these empty stubs is functional in the limited sense that it can parse `xyz` programs, build runtimes, and round-trip them through binary form and back; what it cannot do is anything semantically meaningful with the parsed program, because the empty stubs ignore their arguments.

This default behavior is intentional and is the property that lets the bootstrap loop of Chapter 2 close. When Phase 2.3 produces `xyzGenerator.cc` with empty stubs, Phase 2.5 immediately compiles it into an executable that can re-parse `xyz` grammars and reproduce its own source. The developer's Phase 2.4 work — filling the stubs — transforms the executable from a round-trip-faithful identity transformer into the actual target-language compiler the developer set out to build.

Chapter 8 — Compilation, De-Compilation, and the Two-Phase Procedure

The Parsing Model of Chapter 2 identified five named morphisms between the three forms of a parsing artifact: Phase 2.1a (source text → runtime), Phase 2.1b (runtime → source text), Phase 2.2a (runtime → binary), Phase 2.2b (binary → runtime), and Phase 2.2c (binary → source text). Chapter 7 introduced the additional Phases 2.3, 2.4, and 2.5 that close the loop from a parsed artifact to a built target compiler. This chapter specifies all eight phases procedurally: what each phase reads, what it produces, what invariants it preserves, and how round-trip equivalence is empirically demonstrated by the self-test sequences (s.1)–(s.4) and (b.1)–(b.6).

A reader who has internalized Chapters 2 through 7 has the conceptual content of this chapter; what this chapter adds is the operational level. Phase by phase, what command does the user run? What file does the executable read? What memory state is built? What file does it write? How is correctness checked? The chapter concludes with the self-test sequences that exercise the system end-to-end.

8.1 The Eight Phases

Table 8.1 summarizes the eight phases of the CCS compilation procedure. Phases 2.1a through 2.2c are the parsing-model morphisms; Phases 2.3 through 2.5 are the build phases that produce a working target compiler.

```
Phase    Operation                                          In       Out
-------------------------------------------------------------------------
2.1a     Compile source to runtime                          .sgr     runtime
2.1b     De-compile runtime to source                       runtime  .urt
2.2a     Serialize runtime to binary                        runtime  .fgr
2.2b     Deserialize binary to runtime                      .fgr     runtime
2.2c     De-compile binary to source                        .fgr     .urt
2.3      Generate target compiler source files              runtime  .h/.cc
2.4      Developer adds semantic implementation             (manual editing)
2.5      Build target compiler executable                   .h/.cc   binary
```

Sections 8.2 through 8.6 specify each of Phases 2.1a, 2.1b, 2.2a, 2.2b, and 2.2c in detail. Phases 2.3, 2.4, and 2.5 were specified in Chapter 7 (Sections 7.1, 7.7, and 7.8); they are referenced here only when the round-trip discussion needs them.

8.2 Phase 2.1a — Compilation

Phase 2.1a converts source text into a `SyntaxControlledRuntime`. It is the only phase that reads the source language directly; every other transformation operates on the runtime, the binary, or both.

8.2.1 Inputs and Outputs

- **Input:** a source file conforming to the grammar that the executable was generated for. For the bootstrap case, the input is `meta.sgr` and the executable is `cppcc` (or its predecessors

`cppccMetaFirst` and `cppccMetaBoot`); for a user grammar `xyz`, the input is some `foo.xyz` and the executable is the `xyz` compiler built from the seven generated files of Chapter 7.

- **Output:** a populated `SyntaxControlledRuntime` object held in memory by the executing `Compiler` instance. The runtime is not by itself written to disk by Phase 2.1a; subsequent phases (2.1b, 2.2a) consume it and produce file output.

8.2.2 Procedure

Phase 2.1a executes the following sequence inside `Compiler::run` when the `SHELL_COMMAND` indicates compilation:

- The `Compiler` constructs a `Tokenizer` (Chapter 4) wrapped around a `FileLineReader` attached to the input file, with the `Language` subclass appropriate to the grammar.
- The `Compiler` constructs the grammar-specific `Parser` subclass (named `xyzParser` for grammar `xyz`), which in its constructor allocates a fresh `SyntaxControlledRuntime` and stores it in the parser's `runtime_` member.
- The parser invokes the grammar's axiom procedure (the recursive-descent procedure named after the grammar's first non-terminal, e.g., `xyzParser::axiomGrammar` for the meta-grammar). The axiom procedure consumes tokens from the tokenizer and, on each successful rule recognition, calls the runtime API of Section 5.8 to record the new `Symbol`, `Rule`, `Name`, or `Context` instance.
- On successful parse (the axiom procedure returns and the next token is end-of-file), Phase 2.1a is complete. The runtime is consistent with the grammar; subsequent phases may consume it.
- On parse failure (the axiom procedure encounters input that no grammar production accepts), the parser writes a diagnostic through `Logger` and returns failure. The runtime's state is partial and must not be consumed by downstream phases.

8.2.3 Invariants

On successful completion, the runtime satisfies the four-entity schema invariants of Section 5.1: every `Rule` instance lies inside a `Symbol` whose ID matches the rule's `symbolID_`; every `Symbol` lies inside a `Context` whose ID matches its `contextID_`; every `TypeInstance` in a rule's `dynamic_` field references a rule that exists in the runtime. These invariants are the contract that downstream phases (2.2a, 2.1b) rely on.

8.3 Phase 2.1b — De-Compilation from Runtime

Phase 2.1b is the inverse of Phase 2.1a: it consumes a `SyntaxControlledRuntime` and produces source text. The source text is identical to what Phase 2.1a would have read modulo indentation rules (Section 8.7).

8.3.1 Inputs and Outputs

- **Input:** a `SyntaxControlledRuntime` object satisfying the invariants of Section 8.2.3. In practice, the runtime input is held in memory by the same `Compiler` instance whose Phase 2.1a populated it; Phase 2.1b is invoked immediately after Phase 2.1a as a verification step.
- **Output:** a text file with the conventional extension `.urt` (for *user runtime* — the de-compiled source from the runtime). For input `meta.sgr`, the output is named `meta.sgr.urt` by convention.

8.3.2 Procedure

Phase 2.1b executes the following sequence inside the `GeneratorRuntime` subsystem (Chapter 7):

- The `GeneratorRuntime` selects the `GeneratorRuntimeStyle` subclass appropriate to the grammar (Section 7.4) — `GeneratorRuntimeStyleMeta` for SGDL grammars, `GeneratorRuntimeStyleXML` for XML grammars.
- The de-compiler walks the runtime starting from its initial context and the symbol identified by the grammar's axiom. For each recognized `Symbol`, it iterates over the symbol's `rules_` container; for each `Rule` instance, it interleaves the `fixed_` values (the terminal data: token strings, numeric literals) with recursive descents into the rules referenced by the `dynamic_` field.
- At each emission point, the active style class's `emit*` methods are invoked (see Section 7.4). The methods write to an output stream that the `GeneratorRuntime` has opened on the `.urt` output file.
- When the recursion completes (the axiom rule's `dynamic_` field is exhausted), the output file is closed and Phase 2.1b is complete.

Phase 2.1b is deterministic: given the same runtime and the same style class, it produces byte-identical output every time. This determinism is essential for the round-trip property of Section 8.7.

8.4 Phase 2.2a — Serialization to Binary

Phase 2.2a converts a `SyntaxControlledRuntime` into a `SyntaxControlledBinary`: the in-memory object graph becomes a flat tagged vector laid out per the BNF of Section 6.4.

8.4.1 Inputs and Outputs

- **Input:** a `SyntaxControlledRuntime` satisfying the invariants of Section 8.2.3.
- **Output:** a `SyntaxControlledBinary` object held in memory, optionally written to a file with extension `.fgr` (Section 6.7). For input `meta.sgr`, the file is conventionally named `meta.sgr.fgr`.

8.4.2 Procedure

The serialization is a single-pass walk over the runtime that builds the binary's `memory_` vector in three appended segments per the layout of Section 6.4:

- The header (six tag-sized words) is written first with the size, axiom, and offset fields placeholdered; the actual values are back-patched once the segment lengths are known.
- The names region is written next: each `Name` instance from the runtime's `NameContainer` yields a length-prefixed sequence of tag words holding its character data. After all sequential names are emitted, the alphabetical name index is computed and written as a vector of offsets sorted by the names they point to.
- The symbols region is written last: each `Symbol` instance from the runtime's `SymbolContainer` yields a `BinarySymbol` record followed by the run of `BinaryRule` records for its rules, each followed by the rule's `fixed_` and `dynamic_` regions.

- Once all three segments are emitted, the offset fields in the header are back-patched and the `size_` field is set to the final word count of `memory_`.

Phase 2.2a is deterministic and lossless. The binary contains exactly the information that the runtime contained; no data is omitted, summarized, or compressed.

8.5 Phase 2.2b — Deserialization from Binary

Phase 2.2b is the inverse of Phase 2.2a: it consumes a `SyntaxControlledBinary` and produces a `SyntaxControlledRuntime` equivalent to the runtime that Phase 2.2a originally serialized.

8.5.1 Inputs and Outputs

- **Input:** a `SyntaxControlledBinary` object, typically loaded from a `.fgr` file by reading its bytes directly into `memory_` as specified in Section 6.7.
- **Output:** a populated `SyntaxControlledRuntime` object held in memory.

8.5.2 Procedure

The deserialization is a single-pass walk over the binary's `memory_` that reconstructs the runtime's containers in their original structure:

- A `BinaryHelper` (Section 6.5) is constructed over the binary to give constant-time index access during the walk.
- A fresh `SyntaxControlledRuntime` is allocated. Its `contexts_` container is populated by iterating over the binary's top-level context map, allocating one `Context` per binary context.
- For each context, the `NameContainer` is populated from the binary's sequential names region, the `SymbolContainer` from the binary's symbols region, and each `Symbol`'s `RuleContainer` from the run of `BinaryRule` records following the `BinarySymbol`. The rule's `fixed_` and `dynamic_` vectors are copied directly from the binary's memory into newly allocated `std::vector<Tag>` instances.

Phase 2.2b is the inverse of Phase 2.2a in the strong sense that `Phase2.2b(Phase2.2a(R)) == R` for every runtime `R` that satisfies the Section 8.2.3 invariants. Empirical demonstration of this identity is part of the self-test sequence (b.1)–(b.6) of Section 8.8.

8.6 Phase 2.2c — De-Compilation from Binary

Phase 2.2c produces source text from a binary directly, without first deserializing to a runtime. It is functionally equivalent to the composition `Phase2.1b(Phase2.2b(B))` but is implemented as a single pass over the binary for efficiency. Phase 2.2c is the operation that lets a tool consume a `.fgr` file and emit human-readable source without ever materializing a full runtime.

8.6.1 Inputs and Outputs

- **Input:** a `SyntaxControlledBinary` object.
- **Output:** a text file with extension `.urt` — the same convention used by Phase 2.1b, but the source is the binary rather than a runtime.

8.6.2 Procedure

Phase 2.2c walks the binary using a `BinaryHelper` (Section 6.5) and invokes the same `GeneratorRuntimeStyle::emit*` methods that Phase 2.1b uses (Section 8.3.2). The traversal differs only in the underlying access pattern: Phase 2.1b dereferences runtime container pointers, while Phase 2.2c reads from the binary's `memory_` vector through helper-cached offsets. The emission stream is identical; the output file is identical, modulo the indentation policy of the active style.

That Phase 2.2c and Phase 2.1b produce the same output is not coincidental; it is a property the implementation deliberately preserves. Both phases route their emission through the same `Style` interface and use the same traversal order; only the data-source layer differs. A grammar developer adding a new style class produces a class that works for both phases by construction.

8.7 Round-Trip Identity and the Bidirectionality Property

The five morphisms of Phases 2.1a through 2.2c, together, give the system its bidirectionality property: any of the three forms (source, runtime, binary) is recoverable from any other. This property is not a single theorem but a small family of round-trip identities, each exercised by a specific phase composition. Table 8.2 enumerates them.

```
Identity                              Composition
-------------------------------------------------------------
source       → source                 P2.1b ∘ P2.1a
source       → source via binary      P2.2c ∘ P2.2a ∘ P2.1a
runtime      → runtime via binary     P2.2b ∘ P2.2a
binary       → binary via runtime     P2.2a ∘ P2.2b
```

Each identity holds modulo a stated equivalence:

- **Source-text identities** hold modulo indentation. The active `GeneratorRuntimeStyle` subclass determines the whitespace placement; a grammar de-compiled and recompiled produces source text whose tokens, in order, are identical to the original's, but whose line breaks and indent depths follow the style class's convention rather than the original author's. Section 7.5 specifies this caveat in detail.
- **Runtime identity** holds in the strong sense. Two runtimes are equal if they have the same contexts (matched by `contextID`), the same names per context (matched by name string), the same symbols per context (matched by `symbolID`), the same rule instances per symbol (matched by `ruleInstanceID`), and the same `fixed_` and `dynamic_` vectors per rule. The composition `Phase2.2b ∘ Phase2.2a` preserves all five conditions.
- **Binary identity** holds in the byte-exact sense. The composition `Phase2.2a ∘ Phase2.2b` produces a binary whose `memory_` vector is byte-for-byte identical to the original's. This holds because Phase 2.2a is deterministic in its segment-write order and Phase 2.2b preserves all the data Phase 2.2a wrote.

Each of these properties is empirically demonstrated by the self-test sequences of Section 8.8 against the meta-grammar and against the worked-example grammars shipped with the reference implementation. None is mechanically proved. As Section 1.3 notes, the precise grammar class for which all four identities hold is not formally characterized in this specification or in the source materials it draws upon. The

expectation, validated empirically against every grammar tested, is that the identities hold for any LL(1) grammar that the parser accepts (Section 3.7.1).

8.8 Self-Test Sequences

Two self-test sequences are conventionally used to validate a CCS build. Sequence (s.1)–(s.4) exercises the source-runtime-source round trip; sequence (b.1)–(b.6) exercises the binary-mediated round trip including binary serialization and deserialization. Both sequences are run against the meta-grammar `meta.sgr` as part of the build verification (*[whitepaper §12.3]*).

8.8.1 Sequence (s.1)–(s.4): Source → Runtime → Source

The source-only round trip verifies that compilation followed by de-compilation reproduces the original source modulo indentation. Each step is a single command-line invocation of the CCS executable; Step (s.4) compares text files using the standard `diff` utility with whitespace normalization.

```
# (s.1) Compile the meta-grammar to a runtime
#       and immediately de-compile it back to text.
cppcc --compile --decompile-runtime  meta.sgr

# Result: meta.sgr.urt is written next to meta.sgr,
# containing the de-compiled source.

# (s.2) Inspect the de-compiled source.
less meta.sgr.urt

# (s.3) Compare original to de-compiled source,
#       ignoring indentation differences.
diff -B -w  meta.sgr  meta.sgr.urt

# (s.4) Expected result: no output, exit code 0,
#       meaning the two files are identical
#       modulo whitespace.
```

On a correct build, Step (s.3) produces no output and the shell's `$?` is 0. Any output indicates a token-level discrepancy and a failure of round-trip identity — either the parser failed to recognize a construct, the runtime failed to record what the parser recognized, or the de-compiler failed to emit a recognized rule. All three outcomes are bugs.

8.8.2 Sequence (b.1)–(b.6): Source → Runtime → Binary → Runtime → Source

The binary-mediated round trip verifies the full chain: source compiles to runtime, runtime serializes to binary, binary deserializes back to runtime, and the second runtime de-compiles to source identical to the original.

```
# (b.1) Compile the meta-grammar to a runtime
#       and serialize the runtime to a binary.
cppcc --compile --serialize-binary  meta.sgr

# Result: meta.sgr.fgr is written, holding the
# binary form of the parsed meta-grammar.
```

```
# (b.2) Read the binary back as a fresh runtime
#       and de-compile that runtime to text.
cppcc --read-binary --decompile-runtime  meta.sgr.fgr

# Result: meta.sgr.urt is written from the
# binary-derived runtime.

# (b.3) Alternatively, de-compile directly from
#       binary without materializing a runtime.
cppcc --decompile-binary  meta.sgr.fgr

# Result: meta.sgr.urt is written, identical to
# the (b.2) output.

# (b.4) Compare the two .urt files from (b.2) and (b.3).
diff  meta.sgr.urt.from-runtime  meta.sgr.urt.from-binary

# (b.5) Compare the de-compiled source against the
#       original source.
diff -B -w  meta.sgr  meta.sgr.urt

# (b.6) Compare the binary against itself after a
#       runtime round-trip (deserialize then
#       reserialize).
cppcc --read-binary --serialize-binary  meta.sgr.fgr  meta.sgr.fgr.2
cmp  meta.sgr.fgr  meta.sgr.fgr.2
```

On a correct build, Steps (b.4), (b.5), and (b.6) all complete with no output and exit code 0. Step (b.4) verifies that Phase 2.2c and the composition Phase 2.1b ∘ Phase 2.2b produce the same source text. Step (b.5) verifies the source-text round-trip identity. Step (b.6) verifies that `Phase2.2a ∘ Phase2.2b` is the identity on binaries (byte-for-byte).

The exact command-line flag names shown above (e.g., `--compile`, `--serialize-binary`) correspond to the operations the `SHELL_COMMAND` enumeration of Section 4.1 enumerates. Section 9.1 specifies the `cppcc` command-line interface in detail; sequences (s.1)–(s.4) and (b.1)–(b.6) above use representative flag names that may differ in surface syntax from the reference implementation.

8.8.3 Beyond meta.sgr: Self-Tests for User Grammars

Both self-test sequences generalize from `meta.sgr` to any user grammar. For a target grammar `xyz.sgr` that has been processed through the full bootstrap (yielding the seven generated files of Section 7.7 plus a working `xyzCompiler` executable from Phase 2.5), the analogous sequences are:

- (s.1)–(s.4) using `xyzCompiler` against any source file conforming to the `xyz` grammar.
- (b.1)–(b.6) using `xyzCompiler` against the same source files, exercising `xyzCompiler`'s own binary serialization and deserialization.

Running the user-grammar self-tests against representative source files is the standard validation step for a new grammar before it is considered ready for production use. A grammar that passes the analog of sequence (s.1)–(s.4) for a representative test corpus has demonstrated round-trip fidelity for that corpus; a grammar that passes (b.1)–(b.6) has additionally demonstrated correct binary-format handling. Failures in

either sequence point to specific phases (parser, runtime API, generator-style) where the grammar developer should focus debugging effort.

Chapter 9 — The cppcc Executable

The CCS executable in the C++ reference implementation is named `cppcc`. This chapter specifies the command-line interface that `cppcc` presents to its users: the operations it performs (Section 9.2), the file-naming conventions it follows (Section 9.3), the diagnostic output it produces (Section 9.4), and representative invocation patterns (Section 9.5). This chapter is reference material; readers familiar with the operations from Chapters 7 and 8 will find this is a brief restatement at the command-line level.

9.1 Synopsis

A typical cppcc invocation has the form:

```
cppcc <operation-flags>  <input-file>  [output-file]
```

The operation flags specify which morphisms of Chapter 8 should be applied; the input file is the file consumed by the first morphism in the chain; the output file, when supplied, names the file produced by the last morphism. When no output file is named, `cppcc` applies the file-naming convention of Section 9.3 to derive the output file from the input.

A given invocation may chain multiple morphisms. The flags `--compile --decompile-runtime` chain Phase 2.1a and Phase 2.1b; the flags `--compile --serialize-binary` chain Phase 2.1a and Phase 2.2a. The valid chains are those whose successive phases agree on intermediate type: Phase 2.1a produces a runtime, so it can be followed by Phase 2.1b (consumes runtime) or Phase 2.2a (consumes runtime); Phase 2.2b produces a runtime and can likewise be followed by Phase 2.1b or another Phase 2.2a. Section 9.5 enumerates the canonical combinations.

9.2 Operations

The operations the executable can perform correspond directly to the eight phases of Chapter 8. The current section describes their command-line spelling.

9.2.1 Phase 2.1a: Compile

Flag: `--compile` (or `-c`). Reads a source file conforming to the executable's grammar and produces a runtime in memory. Almost never used alone — the runtime is in-memory and disappears at process exit — but is the leading flag in most chains.

9.2.2 Phase 2.1b: De-compile from Runtime

Flag: `--decompile-runtime` (or `-d`). Walks the runtime currently in memory and emits source text. When chained with `--compile`, the runtime is the one Phase 2.1a just produced; when chained with `--read-binary`, the runtime is the one Phase 2.2b just produced.

9.2.3 Phase 2.2a: Serialize to Binary

Flag: `--serialize-binary` (or `-s`). Walks the runtime currently in memory and writes a `.fgr` binary to the output path.

9.2.4 Phase 2.2b: Read (Deserialize) Binary

Flag: `--read-binary` (or `-r`). Reads a `.fgr` binary file from disk and reconstructs the runtime in memory. Substitutes for `--compile` as the leading operation in chains that begin from a binary rather than from source.

9.2.5 Phase 2.2c: De-compile from Binary

Flag: `--decompile-binary` (or `-D`). Reads a `.fgr` binary file and emits source text by walking the binary directly, without first deserializing to a runtime. Equivalent in output to `--read-binary --decompile-runtime` but more efficient.

9.2.6 Phase 2.3: Generate

Flag: `--generate` (or `-g`). Walks the runtime currently in memory and emits the seven C++ source files of Section 7.7 into the output directory. Used during Phase 2.3 of a target compiler's build.

9.2.7 Convenience Combinations

Two combinations appear often enough that the reference implementation may provide single-flag aliases:

- `--self-test-source` (alias for `--compile --decompile-runtime`) — the source-only round trip of Section 8.8.1.
- `--self-test-binary` (alias for `--compile --serialize-binary` followed by `--read-binary --decompile-runtime` as a second invocation) — the binary-mediated round trip of Section 8.8.2.

Aliases for build orchestration scripts and for the self-test sequences are convenience only; the underlying operations are those of Sections 9.2.1 through 9.2.6.

9.3 File-Naming Conventions

When the user does not supply an output filename, `cppcc` derives one from the input filename by appending an extension. The conventions:

- Source-language input: extension `.sgr` (for SGDL) or whatever extension the target grammar uses (e.g., `.xml` for XML grammars or `.json` for JSON grammars).
- De-compiled source output: extension `.urt` appended to the original filename. So `meta.sgr` compiled and de-compiled produces `meta.sgr.urt`.
- Binary output: extension `.fgr` appended to the original filename. So `meta.sgr` serialized produces `meta.sgr.fgr`.
- Generated source files: named per the prefix of the grammar (Section 3.3.1). For grammar `xyz`, the seven files of Section 7.7 are written to the output directory with names `xyzParser.h`, `xyzParser.cc`, `xyzGenerator.h`, `xyzGenerator.cc`, `xyzKeyWordDefinition.h`, `xyzKeyWordDefinition.cc`, and `xyzMakeGenerators.cc`.

9.4 Diagnostic Output and Exit Codes

All diagnostic output flows through the `Logger` instance owned by the `Shell` (Section 4.1). The default destination is standard error; the `--log-file` flag redirects to a named file. Verbosity is controlled by `--debug` and `--quiet`: the former enables verbose tracing of parser and runtime operations (intended for grammar development), the latter suppresses informational messages and prints only errors.

Exit codes:

- `0` — success; all phases completed without error.
- `1` — parse error in the input source.
- `2` — file I/O error (input not found, output not writable).
- `3` — binary format error (input not a valid `.fgr` file, or version mismatch).
- `4` — internal error (assertion failure, allocator failure).

9.5 Examples

Common invocations and what they do.

Self-test of the meta-grammar:

```
cppcc --compile --decompile-runtime  meta.sgr
diff -B -w  meta.sgr  meta.sgr.urt
```

Compile a user grammar to its seven generated files:

```
cppcc --compile --generate  myGrammar.sgr
```

Serialize a parsed source to binary, then deserialize and verify identity:

```
cppcc --compile --serialize-binary  example.xyz
cppcc --read-binary --serialize-binary  example.xyz.fgr  example.xyz.fgr.2
cmp  example.xyz.fgr  example.xyz.fgr.2
```

De-compile a binary directly to text without materializing a runtime:

```
cppcc --decompile-binary  example.xyz.fgr
```

The above invocations show representative flag spellings. The reference implementation’s exact flag set may differ in surface syntax (long forms, short forms, equals-sign vs space separation); consulting `cppcc --help` on the local installation gives the authoritative list.

Chapter 10 — Targeting CCS at a New Language

The most common reason to use CCS is to build a compiler for a new language. This chapter walks through the procedure end-to-end with a worked example: targeting CCS at JSON. The example is small enough to follow completely yet exercises every step of the procedure: writing the grammar file (Section 10.2), running the compiler-compiler to produce the seven generated files (Section 10.3), recognizing where the predefined token classes are sufficient and where a `Language` subclass is required (Section 10.4), and providing the developer's Phase 2.4 contribution (Section 10.5). Section 10.6 generalizes the procedure into a checklist applicable to any new language.

10.1 Procedure Overview

Targeting CCS at a new language consists of these steps, in order:

- **Write the grammar.** Author a `.sgr` file describing the new language's syntax in SGDL (Chapter 3).
- **Identify lexical extensions.** Determine whether the four predefined token classes (`identifier`, `integerToken`, `stringToken`, `termToken`) are sufficient. If the language has tokens not covered — floating-point literals, hexadecimal literals, character classes, contextually-significant punctuation — author a `Language` subclass (Chapter 4.5).
- **Run cppcc.** Invoke `cppcc --compile --generate yourGrammar.sgr`. This produces the seven generated source files of Section 7.7.
- **Implement Phase 2.4.** Edit `yourGrammarGenerator.cc` to fill in the action-macro stubs with semantic content. The empty default Phase 2.4 implementation produces a round-trip-faithful identity transformer; useful semantic processing requires authored implementations.
- **Build the target compiler.** Phase 2.5: invoke the host C++ toolchain on the seven generated files plus your edited `yourGrammarGenerator.cc` to produce the target compiler executable.
- **Validate.** Run the self-test sequences of Section 8.8 against representative inputs in your language. A grammar that survives both (s.1)–(s.4) and (b.1)–(b.6) is ready for production use.

10.2 Worked Example: JSON Grammar

JSON is a useful working example because it is small (a complete grammar fits on one screen), well-known, and uses none of SGDL's more advanced features. The grammar in `json.sgr`:

```
(json
  (jsonText ::= value)

  (value ::=
      object | array
    | stringValue | numberValue
    | trueValue   | falseValue   | nullValue
  )

  (object ::= '{' [ memberList ] '}')
  (memberList ::= member { commaMember })
```

```
  (commaMember ::= ',' member)
  (member ::= stringValue ':' value)

  (array ::= '[' [ valueList ] ']')
  (valueList ::= value { commaValue })
  (commaValue ::= ',' value)

  (stringValue ::= stringToken)
  (numberValue ::= integerToken)
  (trueValue   ::= 'true')
  (falseValue  ::= 'false')
  (nullValue   ::= 'null')
)
```

The grammar declares the axiom `jsonText`, the recursive `value` rule with seven alternatives, the two compound forms (`object` and `array`) with their optional comma-separated lists, and three terminal rules for the JSON keywords `true`, `false`, `null`. No actions are present — the grammar describes only the shape of valid JSON. Filling in the actions (Section 10.5) turns the resulting compiler into something that does meaningful work.

Note that `memberList` is rewritten as a head element followed by an iteration of `commaMember` rather than the more direct form `member { ',' member }`. The two forms are equivalent in the language they recognize but differ in the parse-tree shape they produce; the rewritten form makes each comma-separated continuation a distinct rule instance, which is sometimes preferable for downstream traversal. SGDL accepts both forms.

10.3 Running the Compiler-Compiler

With `json.sgr` written, the next step is to run `cppcc` to produce the seven generated files:

```
cppcc --compile --generate  json.sgr
```

On success, the working directory contains:

- `jsonParser.h` — declares `class jsonParser : public Parser` with one recursive-descent procedure per rule.
- `jsonParser.cc` — implements the procedures, with each one's body derived from the corresponding right-hand side.
- `jsonGenerator.h` — declares the generator subclass and its emission methods.
- `jsonGenerator.cc` — implements the generator, with the action-macro stubs that Phase 2.4 will fill.
- `jsonKeyWordDefinition.h` — declares named integer constants for each non-terminal in the grammar.
- `jsonKeyWordDefinition.cc` — populates the constants.
- `jsonMakeGenerators.cc` — the factory functions for instantiating the generator pair.

Compiling these seven files together with the CCS runtime library and a small main wrapper produces an executable, `json`, that can parse JSON files, build runtimes, serialize them to binaries, and de-compile them

back to source. With no Phase 2.4 work, the executable is round-trip-faithful: any JSON it consumes can be re-emitted unchanged modulo whitespace. Section 10.5 specifies the additional Phase 2.4 work needed to make the executable do something more interesting than identity transformation.

10.4 When the Predefined Tokens Suffice — and When They Don't

The grammar above uses `integerToken` for `numberValue`. This works for JSON values like `42` or `-100` but does not handle JSON's full numeric literal syntax: `3.14`, `1e-5`, `1.5e10`. The predefined `integerToken` recognizes only digit sequences; everything richer requires extension.

The extension is a `Language` subclass (Section 4.5). The subclass overrides the `getNumeric` virtual method on `Language` to recognize the full JSON numeric form, then registers itself as the active language for the `json` grammar. A sketch of the relevant code:

```
class LanguageJSON : public Language {
public:
    LanguageJSON(Tokenizer& t)
        : Language(t, LanguageTokenizerSet::JSON) {}

    void getNumeric() override {
        // Parse: [-]? digits ( "." digits )?
        //                    ( ("e"|"E") ("+"|"-")? digits )?
        // Produce a Tag containing the numeric value
        // and register it as a numberToken in the runtime.
        // ... implementation ...
    }
};
```

With `LanguageJSON` registered, the `json.sgr` grammar above works unchanged: `numberValue ::= integerToken` is replaced by `numberValue ::= numberToken` (where `numberToken` is the token class `LanguageJSON` adds), and the parser recognizes the richer numeric syntax. The grammar file changes by one line; the C++ subclass file is new; everything else — the seven generated files, the developer's Phase 2.4 work, the build system — remains unchanged.

This is the general pattern. The four predefined token classes cover most of what most languages need; a small minority of token forms require a `Language` subclass; the subclass is local in scope and does not require modification of any other CCS subsystem.

10.5 Phase 2.4 — Implementing Semantic Processing

The seven generated files of Section 10.3 produce a round-trip-faithful identity transformer: it reads JSON in, builds a runtime, can serialize the runtime to binary, can de-compile back to source, and that's all. The interesting question — what does the executable *do* with the parsed JSON? — is the developer's to answer in Phase 2.4.

Phase 2.4 work happens in `jsonGenerator.cc`. The generated file contains action-macro stubs at the points where the grammar's actions appear (zero in the JSON example, since `json.sgr` has no actions) and at the points where the framework's implicit hooks fire — most importantly, the per-rule emission methods on the active `GeneratorRuntimeStyle` subclass. For JSON, the developer would typically:

- Add a member to the JSON generator that accumulates a parallel data structure (e.g., a tree of `std::variant<map, vector, string, double, bool, nullptr_t>` nodes) as parsing progresses.
- Override `emitObjectStart`, `emitObjectEnd`, `emitArrayStart`, etc., to push and pop nodes in the parallel structure.
- Override `emitNumber`, `emitString` to convert the rule's `fixed_` values into the parallel structure's leaf nodes.
- Add a public method `getRoot()` that returns the parallel structure's root, so that calling code can access the parsed JSON as a typed C++ data structure.

After Phase 2.4 editing, the executable is a JSON parser that produces a typed C++ representation. Subsequent compilations of `json.sgr` must preserve the developer's Phase 2.4 contributions; this preservation is handled by the action-macro mechanism described in Section 3.5 and is reliable in practice as long as the developer's edits stay between the comment-delimited stub regions that `cppcc` emits.

10.6 General Targeting Checklist

The procedure of Sections 10.1 through 10.5 generalizes. For any new language:

- **Step 1. Write the `.sgr` grammar.** Use the constructs of Chapter 3 (alternation, iteration, optionality). Make sure the grammar is LL(1) (Section 3.7.1).
- **Step 2. Audit the lexer.** List the kinds of tokens the language uses. If any are not covered by `identifier`, `integerToken`, `stringToken`, `termToken`, author a `Language` subclass.
- **Step 3. Run cppcc.** Generate the seven files. Compile them.
- **Step 4. Run self-tests.** Apply the sequences of Sections 8.8.1 and 8.8.2. Fix any discrepancies before proceeding to Phase 2.4 — a grammar that does not round-trip will not parse correctly with semantic processing layered on top.
- **Step 5. Implement Phase 2.4.** Add the data structures, methods, and action-macro fills that turn the executable from an identity transformer into the application you set out to build.
- **Step 6. Re-run self-tests.** Verify that Phase 2.4 modifications did not break round-trip behavior. The empty stubs preserved round-trip identity; non-trivial Phase 2.4 implementations should preserve it as well, since the stubs sit alongside emission code rather than replacing it.

In the reference implementation, the XML target follows this checklist as documented in *[whitepaper §12.4]*; it required a `LanguageXML` subclass for the context-sensitive treatment of `>` (Section 4.4) and substantial Phase 2.4 work to handle attribute reordering, entity references, and namespace resolution. The XML target is a useful precedent for any developer targeting a new structured-text language.

Chapter 11 — Multiplatform Operation and Binary as Exchange Substrate

The runtime form of a parsing artifact is C++-specific: it lives in heap memory through `std::map` and `std::vector` instances, with pointers between heap allocations. The runtime's representation is therefore not portable: a runtime constructed by one process cannot be passed to another process or to a different language.

The binary form is different. It is a single contiguous `std::vector<Tag>` (Section 6.3) with no embedded pointers; written to a file, it is byte-for-byte identical to its in-memory form (Section 6.7). The binary is therefore portable in three senses: across processes (Section 11.3), across machines with the same `Tag` configuration (Section 11.2), and — with appropriate adapters — across implementation languages (Section 11.4).

This chapter specifies the considerations that govern each of these portability dimensions, starting with the underlying platform variation in word size and byte order (Section 11.1).

11.1 Tag Word Size

Section 4.7 specified the `Tag`, `UnsignedTag`, and `TypeInstance` types as conditional on the preprocessor symbol `_CPPCC_TAG_32BITS_`. On 32-bit builds, `Tag` is 4 bytes and `TypeInstance` packs a 10-bit `symbolID` with a 22-bit `ruleID`; on 64-bit builds, `Tag` is 8 bytes and `TypeInstance` uses a 16/48 split.

The implication for the binary form: a `.fgr` file produced on a 32-bit build is **not** interchangeable with a `.fgr` file produced on a 64-bit build. Every word in the binary is sized differently, the bit layouts of `TypeInstance` fields differ, and the offsets within the binary's header reference word counts that mean different byte counts on the two builds. A reader compiled for 32-bit cannot read a 64-bit binary, and vice versa.

The reference implementation's convention is that a development organization picks one `Tag` size and uses it across all CCS deployments. 64-bit is the practical default for modern systems; the 32-bit build is retained for memory-constrained environments and for compatibility with legacy tooling. Mixing builds within a single deployment is not supported.

11.2 Endianness

The binary format does not specify byte order. The bytes of a `Tag` are written in whatever order the writing machine's CPU stores them; the bytes are read in whatever order the reading machine's CPU stores them. On homogeneous deployments — every machine running the same architecture — this is correct and efficient. On heterogeneous deployments, a binary written on (say) a big-endian machine cannot be directly read on a little-endian machine.

The reference implementation does not currently provide a portable binary format. Three approaches are available to a developer who needs cross-architecture interchange:

- **Convert at write time.** Modify the writer to emit the binary in a canonical byte order (typically big-endian, the network-byte-order convention) regardless of host endianness. The reader is correspondingly modified to swap to host order on read.

- **Convert at read time.** Add a marker byte to the binary header indicating writer endianness; readers detect mismatches and byte-swap at read time. Adds a few extra cycles per word read but keeps the writer simple.
- **Avoid the issue.** Constrain deployment to homogeneous architectures. This is the reference implementation's default position and is sufficient for many applications.

The marker-byte approach is the most general and is the one a future revision of the binary format would likely adopt. As of the present specification, the developer who needs cross-architecture interchange is responsible for adding the conversion themselves.

11.3 Binary as Inter-Process Exchange

Two cooperating processes on the same machine, both built from the same CCS sources with the same `Tag` configuration, can exchange parsed program artifacts as binaries with no conversion. The pattern:

- Process A parses a source program (Phase 2.1a) and serializes the runtime to a binary (Phase 2.2a), writing the binary to a file or to a pipe.
- Process B reads the binary (Phase 2.2b), reconstructing the runtime in its own memory. From process B's point of view, the runtime is indistinguishable from one process B parsed itself.

This pattern is the basis for a number of useful architectures. A *compiler service* parses source files and serves binaries to client processes that perform downstream analysis. A *cache* stores binaries indexed by source-file hash and serves them to repeated requests. A *build pipeline* separates the parsing stage (which is grammar-specific) from later stages (which can be generic over runtimes derived from any CCS-supported grammar).

The performance characteristics make these patterns practical. A `.fgr` binary is typically 3 to 10 times smaller than the source file it derives from (because the binary records only what the parser actually recognized, not the source's incidental whitespace and syntax tokens). Reading a binary is one `std::ifstream::read` call and one `BinaryHelper` construction — typically a few milliseconds for moderate inputs — versus the parsing time of the source, which scales with input length and grammar complexity. Producer-consumer patterns that would be infeasible at parse rates are routine at binary-read rates.

11.4 Binary as Inter-Language Exchange

The binary format makes no commitments to C++. The format is just bytes laid out per the BNF of Section 6.4, and any program in any language that can read those bytes can recover the parsed artifact.

A reader in another language needs:

- A way to read raw bytes (universal).
- A way to interpret `Tag`-sized words: 4-byte or 8-byte unsigned integers per the writer's configuration.
- A way to interpret `TypeInstance` bit-fields: a 10/22 or 16/48 unsigned integer split per the writer's configuration.

- A walk function that follows the BNF of Section 6.4: header, names, symbols, rules, fixed and dynamic data.

None of these requirements presupposes any C++ feature. A Python reader, a JavaScript reader, a Rust reader, or a reader written in any future language can be authored to consume CCS binaries given access to the BNF and the `Tag` configuration. The reader in the new language will not have a `SyntaxControlledRuntime` — that class is C++-specific — but it can construct an equivalent in-memory representation in its own language's idioms.

This property generalizes the inter-process exchange of Section 11.3 to inter-language exchange. A C++ CCS process can serialize a parsed artifact to a binary; a Python tool can read the binary and operate on it; a JavaScript front end can render a visualization of the same parsed artifact. The artifact moves between languages with no need for textual re-serialization or syntax-tree marshaling. The binary is the substrate.

A specification document for inter-language CCS interoperability is outside the scope of the present technical specification. The *Agentic AI Platform Requirements Specification* (a planned companion document) is intended to develop this direction in detail, including conventions for cross-language schema agreement, versioning, and the role of the binary as the canonical exchange form. The position is that CCS's binary already provides the technical foundation; what remains is the organizational layer that makes it routinely usable across language boundaries and across organizational boundaries.

Chapter 12 — Transformations

A *transformation* in CCS terminology is an operation that takes a parsed artifact — either a runtime or a binary — produces a modified parsed artifact, and either de-compiles the result to source text or hands it off to downstream semantic processing. Transformations are the natural extension of the bidirectionality property: because the runtime and binary forms are first-class data with stable APIs, they can be programmatically modified between parsing and semantic processing in ways that the source language's textual surface would not naturally support.

This chapter specifies the transformation framework (Section 12.1), a small set of canonical transformation patterns that the framework supports (Sections 12.2 through 12.5), and the role of the binary form as a substrate for transformations applied to languages whose surface form is itself a binary file format (Section 12.6).

12.1 The Transformation Framework

Figure 12.1 shows the transformation framework as it applies to source-language processing.

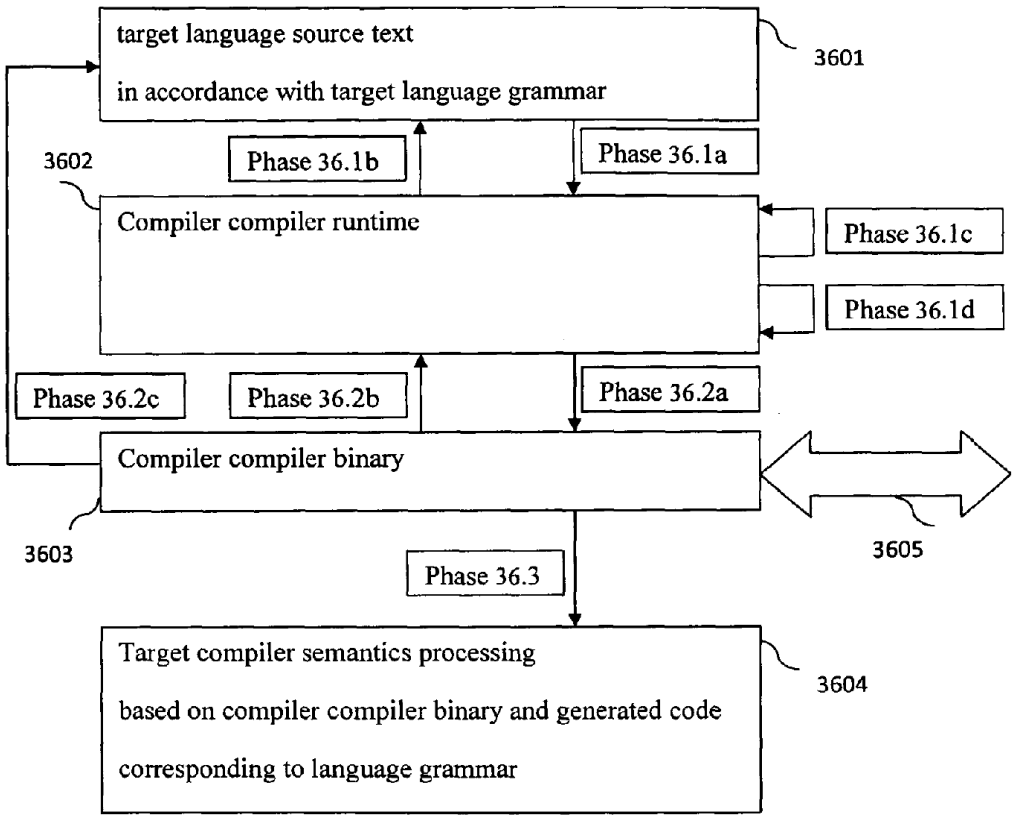

FIG. 36

Figure 12.1 — Transformation framework for source-language processing (FIG 36 of US 8,464,232). The diagram shows source text 3601 cycling through compiler-compiler runtime 3602, optionally transformed via Phase 36.1c or Phase 36.1d into a modified runtime, then either de-compiled back to source text via Phase 36.1b or serialized to compiler-compiler binary 3603, optionally exchanged 3605, and finally consumed by target compiler semantics processing 3604 via Phase 36.3.

The diagram identifies five phase categories.

- **Phase 36.1a / 36.1b.** The compilation and de-compilation phases of Chapter 8 — the runtime-mediated bidirectionality between source text and runtime.
- **Phase 36.1c.** A transformation that takes a runtime and produces a modified runtime *without* going through source. This is the runtime-level transformation point: code that operates directly on the runtime's containers to add, remove, rename, or restructure entities. Examples include the obfuscation, access-control, and content-management transformations of Sections 12.3 through 12.5.
- **Phase 36.1d.** A second runtime-level transformation, applied after Phase 36.1c, that performs further modification before the runtime is serialized or consumed. The framework permits chaining: a parsed artifact can pass through multiple Phase 36.1d transformations before reaching the binary or the semantic-processing stage.
- **Phase 36.2a / 36.2b / 36.2c.** The serialization, deserialization, and de-compilation phases connecting runtime and binary. A transformation can be applied at the runtime level (Phase 36.1c/d) or at the binary level (between Phase 36.2a and 36.2b on the same artifact, or by composing Phase 36.2c ∘ 36.2a).
- **Phase 36.3.** The hand-off from a transformed binary to target compiler semantic processing — the developer's Phase 2.4 work of Chapters 7 and 10. Transformations applied earlier are visible to Phase 36.3 by the time it runs; the semantic-processing code does not need to know whether the artifact it consumes was original or transformed.

Two properties of this framework are worth highlighting.

Composition. Transformations compose. A runtime that has passed through Phase 36.1c and Phase 36.1d is a runtime; it can be serialized through Phase 36.2a, deserialized through Phase 36.2b, and de-compiled through Phase 36.2c just like any other runtime. The output of one transformation is valid input to another. This permits transformation pipelines: organizational policies may apply (say) an obfuscation transformation followed by an access-control transformation followed by a content-management transformation, with each step taking the previous step's output as input.

Reversibility, when designed for. A transformation that records its inverse — by retaining a mapping from transformed entities back to their originals — is reversible. The runtime's monotonic, append-only structure (Section 5.8) makes this straightforward to implement: the inverse mapping lives in the same runtime as a parallel name container, and a follow-on inverse transformation reads that container to undo the changes. Section 12.3's obfuscation transformation is an example.

12.2 The Transformation API

Transformations are implemented as classes that derive from a small abstract `Transformation` base class and override a single virtual method:

```
class Transformation {
public:
    virtual void apply(SyntaxControlledRuntime& rt) = 0;
    virtual ~Transformation() = default;
};
```

The `apply` method receives the runtime by reference and modifies it in place using the runtime API of Section 5.8. A transformation that adds entities calls the create-context, create-rule, and create-name operations; a transformation that modifies existing entities calls the modify-fixed-part and modify-dynamic-part operations; a transformation that removes entities does so by overwriting their content with sentinel values that the de-compiler renders as empty (CCS does not support physical deletion of runtime entities; the monotonic discipline of Section 5.8 forbids it).

Binary-level transformations follow the same pattern but operate on a `SyntaxControlledBinary` rather than a runtime:

```
class BinaryTransformation {
public:
    virtual void apply(SyntaxControlledBinary& bin) = 0;
    virtual ~BinaryTransformation() = default;
};
```

Because the binary is read-only after construction (Section 6.1), a `BinaryTransformation::apply` that needs to modify content typically does so by deserializing to a runtime, applying a corresponding `Transformation`, and re-serializing. Transformations that only filter content (Section 12.4 below) can sometimes operate directly on the binary by producing a new binary whose `memory_` vector excludes the filtered regions and updates the offsets in the header.

12.3 Obfuscation

An *obfuscation* transformation replaces every textual identifier in the runtime with an opaque token, producing a runtime that is structurally identical to the original but whose names carry no information about the source program's intent. The transformed runtime can be de-compiled to source text that compiles and runs identically to the original but is unreadable.

A typical obfuscation procedure:

- Iterate the runtime's `NameContainer` for each context. For each `Name` instance, allocate a new opaque identifier (e.g., `_a01`, `_a02`, …) and replace the `Name`'s `name_` string with the opaque value.
- Optionally, retain the original—to—opaque mapping in a parallel name container. This makes the transformation reversible: a follow-on inverse transformation can restore the original names.
- No other modifications are needed. Symbol IDs and rule references in the runtime use numeric indices that are stable under name renaming; the structure of the runtime is unchanged.

De-compiling the transformed runtime through Phase 2.1b produces source text that uses the opaque identifiers wherever the original used meaningful names. The text is syntactically valid in the source language; it parses; if compiled by the target compiler (Phase 2.5's output), it produces the same machine behavior as the original. The information content visible to a human reader has been reduced, but the machine-readable behavior is preserved.

Obfuscation in this sense is a code-distribution technique. A vendor distributing source-form code to customers may apply obfuscation to protect implementation details while preserving the customer's ability to compile and run the code. The reverse transformation, retained at the vendor, lets the vendor de-obfuscate the code internally for development and debugging.

12.4 Access Control

An *access-control* transformation produces a filtered version of a parsed artifact in which entities that the recipient is not permitted to see have been removed or stubbed. This is the use case that distinguishes CCS from textual obfuscation tools: textual filtering of source code is fragile (a regex-based filter is easily defeated), but filtering at the runtime level operates on the parser's authoritative recognition of program structure, not on the lexical surface.

A representative access-control procedure:

- Tag every `Symbol` or `Rule` instance in the runtime with an access label — a small enumeration value indicating which permission level is required to view it.
- When producing a recipient-specific runtime, iterate the runtime's symbols and rules. For each entity whose access label exceeds the recipient's permission level, replace its `fixed_` and `dynamic_` vectors with a sentinel — typically a single rule reference to a synthetic `redacted` symbol whose presence the de-compiler renders as a placeholder.
- Serialize the filtered runtime to a binary. The binary is now a function of both the original parsed artifact and the recipient's permission level; different recipients receive different binaries from the same source.

De-compiling the filtered runtime produces source text in which redacted regions appear as comments or as a placeholder syntax of the source language's choosing. The binary itself can be exchanged through normal channels; the access-control logic does not require any auxiliary key-management infrastructure. Each recipient receives only the binary they are entitled to see.

12.5 Content Management

A *content-management* transformation operates on a binary as a versioned content artifact. Two patterns are useful:

- **De-duplication.** Two parsed artifacts whose `memory_` vectors are byte-for-byte identical represent the same parsed program. This identity is decidable in constant time per pair (compare lengths, then compare bytes), making it possible to maintain a content-addressable store of binaries indexed by hash. A new submission whose hash matches an existing binary is recognized as a duplicate without parsing.

- **Versioning.** Two parsed artifacts that differ in some rules but share most of their structure can be diffed structurally rather than textually. The `SymbolContainer` and `RuleContainer` of each are compared by symbol ID and rule instance ID; differences yield a structural diff that is independent of source-level whitespace, indentation, and incidental reformatting. A version-control system that stored binaries rather than source text would benefit from semantically meaningful diff and merge operations.

Both patterns rely on the byte-stable serialization of the binary form. A serialization that varied with allocation order (as the runtime does) would not support content addressability; the binary's deterministic layout (Section 8.4) makes the patterns possible.

12.6 Binary File Format Processing

The transformation framework of Section 12.1 applies symmetrically to processing of file formats whose surface form is itself binary. Figure 12.2 shows the analogous diagram for binary-format input.

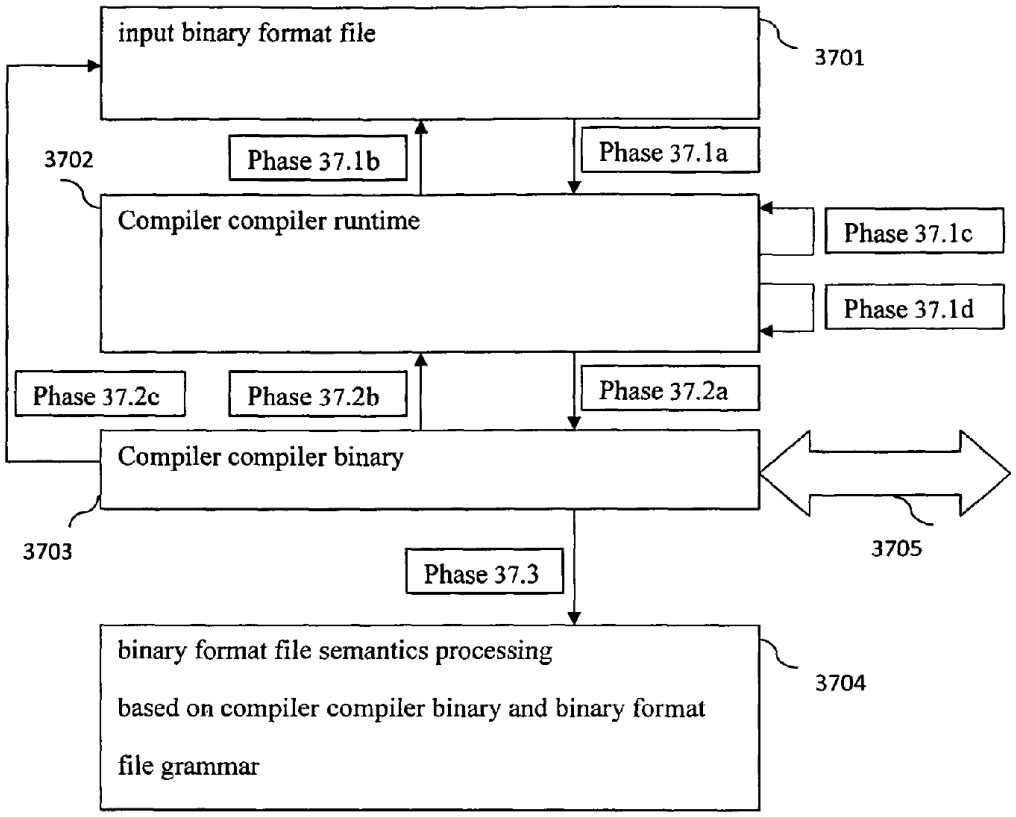

FIG. 37

Figure 12.2 — Transformation framework for binary file format processing (FIG 37 of US 8,464,232). The structure mirrors Figure 12.1: input binary format file 3701 cycles through compiler-compiler runtime 3702 with transformation phases 37.1a through 37.1d, then via 37.2a/b/c through compiler-compiler binary 3703, optionally exchanged 3705, and finally consumed by binary format file semantics processing 3704 via Phase 37.3.

Two observations on the symmetry between Figures 12.1 and 12.2.

First, the parsing-model machinery does not distinguish source-text inputs from binary-file-format inputs. As long as the input has a grammar describing its structure, CCS can parse it. The grammar for a binary file format is a different SGDL grammar from the grammar for a source language, but it is still SGDL, and the resulting compiler is built through the same procedure of Chapter 10. The reference implementation does not currently ship example binary-format grammars, but the framework is in place; *[whitepaper §15]* discusses example file formats (PNG, ELF, MP4) that lend themselves naturally to grammar-based descriptions.

Second, the transformation patterns of Sections 12.3, 12.4, and 12.5 apply to binary-format input as well as to source-text input. A PNG file that has been parsed into a CCS runtime can be obfuscated (rename internal chunk identifiers), filtered (remove ancillary chunks the recipient is not permitted to see), de-duplicated (two byte-identical PNG runtimes are the same image), and version-controlled (structural diffs between two PNG files' runtimes are more meaningful than textual diffs of their bytes). The same Phase 36.1c/36.1d / 37.1c/37.1d transformation hooks apply to both kinds of input.

This symmetry is the deepest of the parsing-model's implications. Source text and binary file formats are not architecturally distinct in CCS; they are both grammar-described inputs that produce parsed artifacts subject to the same transformations. A development organization that has built CCS-based tooling for one class of input — say, source code in a language they care about — has, by the structure of the framework, much of the tooling it would need for the other class as well. Chapter 11's discussion of binary as exchange substrate, combined with Section 12.6's observation here, points at the broader application: CCS's parsing model is naturally suited to systems in which text-form and binary-form artifacts must be uniformly described, transformed, and exchanged.

Appendix A — Patent Map

This appendix maps the specification's sections to the corresponding portions of US Patent 8,464,232. The mapping is at the level of patent column numbers and figure numbers; readers consulting the patent for the legal text underlying any specification claim should locate the cited column and read its surrounding paragraphs.

The patent's claims are not enumerated explicitly below: the patent's independent claims are claim 1 (the compiler-compiler system with syntax-controlled runtime and binary APIs) and the related independent claims that cover the binary form, the de-compilation procedure, and the bidirectionality property. Specification readers verifying coverage should examine the patent's claim section directly. The mapping below gives them the supporting figures and column ranges.

Part I — Foundations.

Spec section	Patent column(s)	Patent figure(s)
Chapter 1	cols 1–3	FIG 1
Chapter 2 (model)	cols 3–5	FIG 2, FIG 3, FIG 4
Chapter 3 (SGDL)	cols 5–10	(none; meta-grammar text)

Part II — Subsystems.

Spec section	Patent column(s)	Patent figure(s)
Chapter 4 management	cols 11–12	FIG 5, 6, 7
4.3 Tokenizer	col 22	FIG 27, 28
4.4 Token hierarchy	col 22–23	FIG 30, 31
4.5 Language	col 23	FIG 29
4.7 Tag/TypeInst	cols 12, 18	FIG 25
Chapter 5 runtime	cols 12–14	FIG 8, 9, 10, 11
5.5 Names	col 13	FIG 12, 13
5.6 Symbols	col 13	FIG 14, 15
5.7 Rules	col 14	FIG 16, 17
5.1 Logical view	col 12	FIG 18
Chapter 6 binary	cols 14–16	FIG 19, 20
6.4 Binary BNF	cols 15–16	FIG 21, 22, 23, 24, 25
6.5 BinaryHelper	col 16	FIG 26
Chapter 7 generator	cols 19–20	FIG 32, 33, 34, 35
Chapter 8 phases	cols 17–18	FIG 3, FIG 4

Parts III, IV, V.

Spec section	Patent column(s)	Patent figure(s)
Chapter 9 cppcc	(not in patent)	—
Chapter 10 targeting	cols 20–21	—
Chapter 11 multipl.	col 18	—
Chapter 12 transforms	cols 22–24	FIG 36, 37

Sections of the specification not mapped to a patent column — specifically the cppcc command-line interface (Chapter 9), the targeting checklist (Chapter 10's procedural framing), and the multiplatform

discussion of Chapter 11 — represent specification material that is not directly reproduced from the patent. They are derived from the reference implementation's practice and from the 2015 whitepaper. Appendix H discusses the scope of this material in more detail.

Appendix B — Class Reference

A compact catalog of the classes specified throughout the document. Each entry gives the class name, its containing subsystem, the chapter and section that specifies it, and a one-line summary of its role.

Management Subsystem (Chapter 4)

IRun	§4.1	Top-level run-method interface
Logger	§4.1	Diagnostic message accumulator
Shell	§4.1	main entry point, owns Compilers
SHELL_COMMAND (enum)	§4.1	Operation selector from command line
Compiler	§4.1	Per-grammar orchestration unit
Action	§4.1	Abstract base for compile/decompile
ActionCompile	§4.1	Phase 2.1a + 2.2a executor
ActionDeCompile	§4.1	Phase 2.1b + 2.2c executor
noncopyable	§4.1	Mixin disabling copy semantics
KeyWordsContainer	§4.2	Four-vector grammar symbol catalog
KeyWordsContainer:: NameVector	§4.2	Alias for std::vector<std::string>

Lexical Subsystem (Chapter 4)

Tokenizer	§4.3	Character-to-token converter
LineReader	§4.3	Pure-virtual character-source base
FileLineReader	§4.3	Reads from a file with listing
StringLineReader	§4.3	Reads from std::string
TokenNameContainer	§4.3	Token-class to display-string map
Tokens	§4.4	Container of Token instances
IToken	§4.4	Token virtual interface
Token	§4.4	Abstract recognizer base
OneCharToken	§4.4	Single-character recognizer
TwoCharToken	§4.4	Two-character recognizer
ThreeCharToken	§4.4	Three-character recognizer
OneCharXMLGreaterThan Token	§4.4	Context-sensitive XML > recognizer
ILanguage	§4.5	Language virtual interface
Language	§4.5	Lexical-machinery container base
LanguageMeta	§4.5	SGDL language subclass
LanguageXML	§4.5	XML language subclass
LanguageTokenizerSet	§4.5	Built-in language enumeration
IParser	§4.6	Parser virtual interface
Parser	§4.6	Recursive-descent base

Tag Family (Chapter 4.7)

Tag	§4.7	Signed word, 4 or 8 bytes
UnsignedTag	§4.7	Unsigned word, matching size
Real	§4.7	Floating-point, matching size
TypeInstance	§4.7	Packed (symbolID, ruleID) struct

Runtime Subsystem (Chapter 5)

SyntaxControlledRuntime	§5.3	Outer runtime class

```
MapVectorContainer<D,K> §5.2  Dual-indexed container template
ContextContainer        §5.3  Map of Tag to Context
Context                 §5.4  Per-scope unit
NameContainer           §5.5  Names per context
Name                    §5.5  Identifier instance
SymbolContainer         §5.6  Symbols per context
Symbol                  §5.6  Per-non-terminal recognition record
RuleContainer           §5.7  Rules per symbol
Rule                    §5.7  Per-occurrence recognition record
```

Binary Subsystem (Chapter 6)

```
SyntaxControlledBinary  §6.2  Outer binary class
BinaryContextContainer  §6.2  std::map<Tag, BinaryContext>
BinaryContext           §6.3  Owns the memory_ tagged vector
BinaryContextHeader     §6.4  Six-Tag overlay at memory_ start
BinarySymbol            §6.4  Per-symbol record overlay
BinaryRule              §6.4  Per-rule record overlay
BinaryRuleDefinition    §6.4  Fixed and dynamic-region overlay
BinaryHelper            §6.5  Index-cache for fast access
BinaryHelper::Symbol    §6.5  Per-symbol cache entry
BinaryHelper::Rule      §6.5  Per-rule cache entry
```

Generator Subsystem (Chapter 7)

```
IGenerator              §7.2  Generator virtual interface
Generator               §7.2  Orchestrator owning binary_/runtime_
IBinaryGenerator        §7.3  Binary-generator interface
GeneratorBinary         §7.3  Binary I/O code emitter
IRuntimeGenerator       §7.3  Runtime-generator interface
GeneratorRuntime        §7.3  De-compilation code emitter
IBinaryGeneratorStyle   §7.4  Binary-emission style interface
GeneratorBinary::
  GeneratorBinaryStyle  §7.4  Binary-style abstract base
GeneratorBinaryStyleMeta  §7.4  SGDL binary style
GeneratorBinaryStyleXML   §7.4  XML binary style
IRuntimeGeneratorStyle    §7.4  Runtime-emission style interface
GeneratorRuntime::
  GeneratorRuntimeStyle §7.4  Runtime-style abstract base
GeneratorRuntimeStyleMeta  §7.4  SGDL runtime style
GeneratorRuntimeStyleXML   §7.4  XML runtime style
metaFirstGeneratorBinary   §7.6  Stage-0 SGDL binary generator
metaFirstGeneratorRuntime  §7.6  Stage-0 SGDL runtime generator
metabootGeneratorBinary    §7.6  Stage-1 SGDL binary generator
metabootGeneratorRuntime   §7.6  Stage-1 SGDL runtime generator
xsdGeneratorBinary         §7.6  Example XSD binary generator
xsdGeneratorRuntime        §7.6  Example XSD runtime generator
```

Transformation Subsystem (Chapter 12)

```
Transformation          §12.2  Runtime transformation base
BinaryTransformation    §12.2  Binary transformation base
```

Appendix C — SGDL BNF (Consolidated)

The complete grammar of SGDL, reproduced verbatim from the file `meta.sgr` as it appears in the C++ reference implementation. This is the same listing as Section 3.2; it is reproduced here for single-place reference.

```
(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 '>'
  )
)
```

Appendix D — Binary BNF (Consolidated)

The complete BNF for the contents of `SyntaxControlledBinary::BinaryContext::memory_`, reproduced from Section 6.4. Each non-terminal is a contiguous region of the tagged-vector memory; each terminal is a single `Tag` word.

```
binaryContext      ::= header sequentialNames alphaNames symbols

header             ::= size axiom numberOfNames startOfNames
                       numberOfSymbols startOfSymbols

sequentialNames    ::= { name }
name               ::= { Tag }                  -- packed string

alphaNames         ::= { nameIndex }
nameIndex          ::= Tag                      -- offset into
                                                   sequentialNames

symbols            ::= { binarySymbol }
binarySymbol       ::= numberOfRuleInstances
                       symbolID
                       symbolStart
                       { binaryRule }

binaryRule         ::= fixedSize
                       dynamicSize
                       ruleStart
                       binaryRuleDefinition

binaryRuleDefinition ::= fixed dynamic
fixed              ::= { Tag }
dynamic            ::= { TypeInstance }
```

Each non-terminal occupying two or more lines in the listing is a structure with named `Tag` fields; each non-terminal whose right-hand side is `{ Tag }` is a length-prefixed array of `Tag` words.

Appendix E — Self-Test Reference

Single-page reference for the self-test sequences specified in Chapter 8.8. Both sequences are run against `meta.sgr` as part of build verification; both generalize to any user grammar.

Sequence (s.1)–(s.4): Source Round Trip

```
# Compile and immediately de-compile back to text.
cppcc --compile --decompile-runtime  meta.sgr

# Compare original to de-compiled, ignoring whitespace.
diff -B -w  meta.sgr  meta.sgr.urt

# Expected: empty diff output, exit code 0.
```

Sequence (b.1)–(b.6): Binary Round Trip

```
# (b.1) Compile and serialize to binary.
cppcc --compile --serialize-binary  meta.sgr

# (b.2) Read binary and de-compile via runtime.
cppcc --read-binary --decompile-runtime  meta.sgr.fgr

# (b.3) De-compile binary directly (no runtime materialization).
cppcc --decompile-binary  meta.sgr.fgr

# (b.4) Compare two .urt files.
diff  meta.sgr.urt.from-runtime  meta.sgr.urt.from-binary

# (b.5) Compare de-compiled source against original.
diff -B -w  meta.sgr  meta.sgr.urt

# (b.6) Verify byte-exact binary identity after round trip.
cppcc --read-binary --serialize-binary  meta.sgr.fgr  meta.sgr.fgr.2
cmp  meta.sgr.fgr  meta.sgr.fgr.2
```

Round-Trip Identities Validated

```
Sequence  Identity                                  Validated by
-----------------------------------------------------------------
(s.1–4)   source → source                           (s.3)
(b.1–6)   source → source via binary                (b.5)
(b.1–6)   binary → binary via runtime               (b.6)
(b.1–6)   Phase2.2c == Phase2.1b∘Phase2.2b         (b.4)
```

Appendix F — Glossary

Definitions of terms used throughout the specification, alphabetized for reference. Cross-references in italics point to the section in which the term is specified in detail.

Action — A numeric tag plus a string sequence appearing in an SGDL rule, used to associate implementation-language code with grammar elements. *§3.5*.

Axiom — The grammar's entry-point non-terminal, named first in the grammar's rule sequence. The parser begins recognition from the axiom. *§3.3.1*.

Binary form — The serialized form of a parsed artifact: a single contiguous `std::vector<Tag>` in memory, byte-identical to its `.fgr` file representation. *Chapter 6*.

Bidirectionality — The property that any of the three forms (source, runtime, binary) can be reconstructed from any of the others, modulo stated equivalences. *§8.7*.

Bootstrap loop — The procedure by which CCS re-builds itself from its own meta-grammar, demonstrating self-hosting. *§2.4*.

Compilation — Phase 2.1a: the morphism from source text to runtime. *§8.2*.

Context — A unit of scoping within a runtime, owning its own `NameContainer` and `SymbolContainer`. *§5.4*.

De-compilation — A morphism from runtime or binary to source text. *§8.3, §8.6*.

Dynamic part — The portion of a `Rule`'s data holding references to other rules, packed as `TypeInstance` values. *§5.7*.

Fixed part — The portion of a `Rule`'s data holding terminal-value records (token strings, integers, etc.). *§5.7*.

Four-entity schema — The internal organization of both runtime and binary: Context, Name, Symbol, Rule. *§5.1, §6.1*.

Grammar — A set of rules in SGDL describing a context-free language. *Chapter 3*.

LL(1) — The class of grammars supported by CCS's recursive-descent parser: those for which one token of lookahead suffices to choose between alternatives. *§3.7.1*.

meta-grammar — The SGDL specification of SGDL itself, contained in the file `meta.sgr`. *§3.6, Appendix C*.

Morphism — In this specification, any of the named operations connecting source, runtime, and binary forms (Phase 2.1a, 2.1b, 2.2a, 2.2b, 2.2c). *§8.1*.

Name — An identifier introduced by the source program; held in a context's `NameContainer`. *§5.5*.

Parsing Model — The architectural pattern of CCS: a grammar, a parser-maintained syntax-controlled runtime, and a syntax-controlled binary independent of the parsing process. *§2.3*.

Phase — A named morphism (2.1a through 2.5) of the compilation procedure. *§8.1*.

Round-trip identity — The property that compositions of the named morphisms produce results identical to their inputs, modulo specified equivalences. *§8.7*.

Rule — A per-occurrence record of a non-terminal's recognition by the parser, with fixed and dynamic parts. *§5.7*.

Runtime form — The in-memory object-graph representation of a parsed artifact: an instance of `SyntaxControlledRuntime`. *Chapter 5*.

Self-test — A standardized command sequence verifying that compilation, de-compilation, serialization, and deserialization produce consistent results. *§8.8, Appendix E*.

SGDL — Source Grammar Definition Language: the input language of the CCS executable. *Chapter 3*.

Style class — A target-language-specific code-emission strategy, derived from `GeneratorBinaryStyle` or `GeneratorRuntimeStyle`. *§7.4*.

Symbol — A per-non-terminal recognition record; owns the rules of all instances of that non-terminal recognized by the parser within a context. *§5.6*.

Tag — The integer type used throughout the runtime and binary as the unit of identifier or value. 4 or 8 bytes depending on build configuration. *§4.7*.

Target compiler — A compiler built using CCS for a particular source language. *Chapter 7*.

Transformation — An operation on a runtime or binary that produces a modified runtime or binary while preserving structure. *Chapter 12*.

TypeInstance — A struct packing `symbolID` and `ruleID` into a single Tag-sized word. *§4.7*.

Appendix G — Bibliography

Primary Sources

Urakhchin, A. F. (2013). *Compiler Compiler System with Syntax-Controlled Runtime and Binary Application Programming Interfaces.* US Patent 8,464,232. Filed June 29, 2010; granted June 11, 2013.

Urakhchin, A. F. (2015). *CCS White Paper: Compiler Compiler System with Syntax-Controlled Runtime and Binary Application Programming Interfaces.* Internal technical document, June 2015.

Foundational Compiler-Construction References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley. The standard reference for parsing theory, including LL(1) and LR(1) parsing classes referenced in Section 3.7.

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and Control*, 8(6), 607–639. The original LR-grammar paper, providing the framework within which CCS's LL(1) restriction is positioned.

Johnson, S. C. (1975). YACC: Yet Another Compiler Compiler. *Bell Laboratories Computing Science Technical Report* 32. The reference YACC paper, from which Section 1.2's "compiler tax" framing identifies departures.

Modern Compiler Infrastructure (Post-2015)

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). *MLIR: Scaling Compiler Infrastructure for Domain Specific Computation.* Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO). The contemporaneous infrastructure for multi-level IR, useful for comparison with CCS's syntax-controlled runtime as an IR alternative.

Brunsfeld, M. and contributors (2018–present). *Tree-sitter: An Incremental Parsing System.* Open-source project at github.com/tree-sitter. Tree-sitter's incremental GLR parsing represents an alternative point in the design space; the comparison with CCS's LL(1) runtime is informative for grammar-tooling discussion.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016). CompCert — a formally verified optimizing compiler. *Embedded Real Time Software and Systems* Conference (ERTS). The formally verified C compiler; comparable to CCS in its emphasis on correctness, contrasting in its proof-bearing rather than empirical approach.

Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). CakeML: A verified implementation of ML. *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (POPL). A second formally verified compiler, this one for an ML-family language; relevant to discussion of how the round-trip property of Section 8.7 might be formally stated.

de Moura, L. and Ullrich, S. (2021). The Lean 4 theorem prover and programming language. *Proceedings of the Conference on Automated Deduction* (CADE). Relevant to discussion of metaprogramming in dependently typed languages; Lean 4's elaborator as compiler-compiler offers another point of comparison.

Note on Citations

The references above are representative rather than exhaustive. The bibliography for the planned *Agentic AI Platform Requirements Specification* companion document will expand this list to include current work on multi-agent systems, programming-language interoperability, and AI-assisted compiler tooling.

Appendix H — Differences from the 2015 Whitepaper

This specification draws on the 2015 CCS Whitepaper as one of its primary sources. The whitepaper is comprehensive but is written as an exposition rather than as a reference specification, and its conventions differ from this document's in several places. This appendix documents the differences for readers consulting both.

Terminology

- **SGDL replaces CCSGDL throughout.** The patent uses "CCSGDL" (Compiler Compiler System Grammar Definition Language); the whitepaper and the C++ reference implementation use "SGDL". This specification uses "SGDL" throughout for consistency with the implementation, with one explicit cross-reference in Section 1.2 to acknowledge the patent's usage.
- **"Parsing Model" is new.** Chapter 2's framing of the system as a Parsing Model with three entities (grammar, syntax-controlled runtime, syntax-controlled binary) and four-entity internal schema (Context, Name, Symbol, Rule) is a specification-level abstraction not present in the whitepaper or the patent. The underlying classes and relationships are unchanged; the framing is intended to give readers a single-page conceptual handle on a system the whitepaper presents in a more incremental fashion.
- **Phase numbering.** This specification uses Phase 2.1a, 2.1b, 2.2a, 2.2b, 2.2c, 2.3, 2.4, 2.5 to enumerate the eight named procedures. The whitepaper does not number them as systematically; it uses prose names ("compile", "de-compile", "serialize", etc.) interchangeably. The specification's numbering is used uniformly in Chapter 8 and referenced from earlier chapters as needed.

Removed Material

- **YACC-style file sections.** Section 3.3 of this specification states that an SGDL file is a single self-contained parenthesized list with two structural elements: a grammar name and a sequence of rule definitions. The whitepaper at points presents an SGDL file as having multiple sections (language declarations, compiler-compiler declarations, embedded code) imported from the YACC tradition; those subsections were never part of SGDL itself, and have been removed from the specification's presentation.
- **Some implementation-detail discussions.** The whitepaper includes detailed discussions of implementation choices (memory pool design, specific allocator strategies) that are properly the domain of the implementation's code rather than its specification. This document omits those discussions; readers wanting implementation-level detail should consult the source code.

Added Material

- **Round-trip property formalization.** Section 8.7's table of round-trip identities (source → source, source → source via binary, runtime → runtime, binary → binary) is more explicit than any equivalent statement in the whitepaper. The whitepaper asserts the bidirectionality property generally; this specification states which compositions are valid and what equivalence each preserves.

- **Indentation policy as a Style class concern.** Section 7.5 specifies that the round-trip-modulo-indentation property is a function of the active `GeneratorRuntimeStyle` subclass. The whitepaper notes that de-compilation does not preserve original indentation but does not name the responsible class.
- **Multiplatform considerations and binary as exchange substrate.** Chapter 11 develops the role of the binary form as an inter-process and inter-language exchange substrate — a direction the whitepaper mentions briefly but does not develop. This direction is the bridge to the planned *Agentic AI Platform Requirements Specification* companion document.
- **Worked JSON example in Chapter 10.** The whitepaper's primary worked example is XML; this specification adds JSON in Section 10.2 because JSON is a smaller, more universally familiar example for new readers. The XML target remains documented in the whitepaper and is referenced throughout this specification as the reference-implementation use case.

Reading Order Recommendation

A reader with no prior CCS familiarity is best served by reading the whitepaper first — it is more expository and motivates the design choices — and then this specification for reference. A reader with implementation-level familiarity (having worked with the C++ source) is best served by reading this specification first, since it organizes the material at the level the implementer needs, and consulting the whitepaper for historical context.