Big Picture on a Small Use Case

End-to-End Walkthrough of CCS through a Bank Account Grammar

A. F. Urakhchin

Draft 1.0 — May 2026

Companion to

CCS Technical Specification (102 pages)

1. Introduction

This document walks through a complete use of the Compiler Compiler System (CCS) on a small example: a grammar describing a bank account as a sequence of integers. The grammar is small enough that every artifact CCS produces — the seven generated source files, the compiled executable, the runtime, the binary, the diagnostic outputs — fits on a few pages and can be inspected end-to-end. The example exercises every concept the *CCS Technical Specification* (the 102-page reference document this walkthrough accompanies) describes abstractly: SGDL grammar authoring, the bootstrap procedure, the syntax-controlled runtime and binary, the round-trip de-compilation property, and the packed Tag identifier scheme.

The document is organized to mirror what a developer encounters in practice. Section 2 presents the grammar. Section 3 shows the project directory layout. Section 4 walks through building the target compiler with the `makeall` script. Section 5 shows the seven generated source files. Section 6 invokes the resulting compiler on a sample input. Section 7 reads the four output files — `.urt`, `.log`, `.lis`, `.fgr` — in turn. Section 8 dissects the binary file word by word against the binary BNF. Section 9 decodes the packed-tag arithmetic that the spec specifies in §4.7. Section 10 names what the example demonstrates about the Parsing Model as a whole.

Cross-references to the technical specification appear throughout in *italic small caps* notation. A reader unfamiliar with the spec can follow this document standalone; a reader who has consulted the spec will find that this walkthrough makes its abstractions concrete in a way that complements the formal text.

2. The Grammar: cca.sgr

The grammar is named `cca` (for *credit card account*). It describes a bank account as a non-empty sequence of integer numbers, with two non-terminal rules and an embedded action macro pair around the axiom. The complete grammar:

```
(cca
  (account ::=
    0 ="  METAACTBEG();"=
    numbers
    0 ="  METAACTEND();"=
  )

  (numbers ::=
    { integerToken }
  )
)
```

Three observations.

- The first non-terminal, `account`, is the *axiom* — the entry-point rule from which parsing begins (*spec §3.3.1*). It expands to the embedded action `METAACTBEG()`, the non-terminal `numbers`, and the embedded action `METAACTEND()`. Action macros are the developer's Phase 2.4 extension

points (*spec §3.5*); their bodies are written by the developer and inserted into the generated parser at the corresponding positions during code emission.

- The second non-terminal, `numbers`, uses the iteration operator `{ … }` to denote zero or more occurrences of `integerToken` — the predefined token class that recognizes decimal-digit sequences (*spec §4.2*). The iteration is greedy and terminates on the first non-integer input.
- The grammar is trivially LL(1) (*spec §3.7.1*): one token of lookahead suffices to decide between continuing the iteration and exiting it. CCS's recursive-descent parser handles it without any developer intervention.

The file is named `cca.sgr` and lives in the project's `build` subdirectory.

3. Project Layout

The project lives at `~/ccsUseCases/creditCardAccount` and follows the standard CCS project layout. Subdirectories:

```
~/ccsUseCases/creditCardAccount/
├── build/    grammar source, generated artifacts
├── incloc/   .h files copied from build/ after generation
├── srcloc/   .cc files copied from build/ after generation
├── lib/      libccagenerate2.a built from srcloc/
├── bin/      ccacompiler executable built from lib/
├── src/      reserved for hand-authored sources (empty here)
├── include/  reserved for hand-authored headers (empty here)
├── run/      example sample inputs
└── test/     reserved for test harnesses (empty here)
```

The split between `build/`, `incloc/`, and `srcloc/` merits explanation. CCS generates fresh source files into `build/` on every invocation; the `makeall` script then copies them to `incloc/` (headers) and `srcloc/` (implementations), where the host C++ toolchain consumes them. This separation lets a developer regenerate from grammar without disturbing a stable build snapshot, and lets the host build system depend on the stable copies rather than on files that change with every grammar edit.

The seven generated files — specified in *spec §7.7* — appear in `build/` immediately after grammar processing:

```
ccaParser.h           parser declarations
ccaParser.cc          parser implementation
ccaGenerator.h        generator declarations
ccaGenerator.cc       generator implementation (Phase 2.4 stubs)
ccaKeyWordDefinition.h     symbol-id constants
ccaKeyWordDefinition.cc    symbol-id table population
ccaMakeGenerators.cc       factory functions
```

The first six contain the parsing and emission logic for the `cca` grammar. The seventh, `ccaMakeGenerators.cc`, contains the factory functions through which `Compiler` instantiates a generator pair for the `cca` grammar at runtime (*spec §7.6*).

4. Building the cca Frontend

The build is driven by a small shell script, `makeall`, that invokes `cppcc` on the grammar and then compiles the generated files with the host C++ toolchain. The script:

```
set -v
echo "CPPCCHOME:$CPPCCHOME"
$CPPCCHOME/bin/cppcc cca.sgr
cp *.h ../incloc
cp *.cc ../srcloc
ls -l ../incloc
ls -l ../srcloc
cd ../lib
rm -rf *.a
make -f makeboot2
cd ../bin
rm -rf ccacompiler
make -f makeboot2
```

The script executes the four phases of *spec §10.1* that follow grammar authoring: Phase 2.3 (the `cppcc cca.sgr` invocation), the file-copying that places generated artifacts where the build system expects them, Phase 2.5's library construction (the `lib/libccagenerate2.a` archive), and Phase 2.5's executable link (the `bin/ccacompiler` binary). On a successful build, the executable is roughly 2.7 megabytes.

The grammar specifies no developer action bodies (the `METAACTBEG` and `METAACTEND` macros expand to empty in the default generator), so Phase 2.4 — the developer's semantic-implementation step (*spec §10.5*) — is not exercised in this build. The resulting `ccacompiler` is therefore a round-trip-faithful identity transformer: it can parse `cca` inputs, build runtimes, serialize them to binaries, and de-compile them back to source identical to the original modulo whitespace (*spec §7.8*).

5. The Generated Parser

Of the seven generated files, the parser implementation (`ccaParser.cc`) is the most informative for understanding what `cppcc` produced. The two procedures corresponding to the grammar's two non-terminals appear below.

5.1 ccaParser::account

```
void
ccaParser::account(cppcc::scr::tag::Long& tag)
/*
( account ::= 0 = "  METAACTBEG();"= numbers
              0 = "  METAACTEND();"= )
*/
{
  TagVector f(1);
  METAACTBEG();
  numbers(f[0]);
  crefixe(KW_ACCOUNT, f, &tag);
  METAACTEND();
}
```

The procedure mirrors the grammar definition almost literally. A fixed-part vector `f` of length 1 is allocated to hold the result of the embedded `numbers` invocation. The two action macros bracket the body. The runtime API call `crefixe` ("create *rule with* `fixed` and *embedded reference"*) registers a new `Rule` instance under the symbol `KW_ACCOUNT` with the fixed-part vector as its data, and writes the resulting tag into the output `tag` parameter (*spec §5.8*).

5.2 ccaParser::numbers

```
void
ccaParser::numbers(cppcc::scr::tag::Long& tag)
/*
( numbers ::= { integerToken } )
*/
{
  TagVector d;
  for(;;) {
    if(iseof()) break;
    if((kword() == KW_INTEGERTOKEN) ||
       (kword() == KW_FLOATTOKEN)) {
      d.push_back(0);
      integerToken(d.back());
    }
    else break;
  }
  crelasdyn(KW_NUMBERS, d, &tag);
}
```

The iteration operator `{ … }` becomes a `for` loop with a two-clause termination check: end-of-file, or a token that is neither integer nor float. Each iteration appends a fresh slot to the dynamic-part vector `d` and calls the predefined `integerToken` recognizer to fill it. After the loop, the runtime API call `crelasdyn` ("create *rule with last* `dynamic` …") registers a `Rule` instance under the symbol `KW_NUMBERS` with the populated dynamic-part vector.

Each invocation of `integerToken` yields a packed `TypeInstance` containing both the symbol ID `KW_INTEGERTOKEN` (3) and the rule ID assigned to that occurrence (0, 1, 2, …). Section 9 below decodes the resulting `Tag` values.

6. Running the Compiler on ex.txt

A sample input file `ex.txt` contains a single line of integers:

```
1 2 3 4 5 6 7 100000
```

Invoking the compiler:

```
../bin/ccacompiler ex.txt
```

produces four output files in the same directory:

```
ex.txt.urt    de-compiled source (round-trip output)
ex.txt.log    diagnostic trace
ex.txt.lis    parser listing with token-by-token detail
```

```
ex.txt.fgr    syntax-controlled binary
```

The four extensions match the file-naming conventions specified in *spec §9.3*. The `.urt` file is the de-compilation output (Phase 2.1b); the `.fgr` file is the binary serialization (Phase 2.2a). Together they exercise the bidirectional core of the Parsing Model: the source text is parsed, the runtime is built, the runtime is serialized to binary, the runtime is de-compiled back to source. Each output is examined in turn below.

7. The Output Files

7.1 ex.txt.urt — Round-Trip De-Compilation

The de-compiled source:

```
1 2 3 4 5 6 7 100000
```

Identical to the original. The `cca` grammar uses no whitespace-significant constructs and the `GeneratorRuntimeStyleMeta` indentation policy preserves token order with simple space separation. The round-trip identity `Phase2.1b ∘ Phase2.1a == identity` specified in *spec §8.7* is empirically demonstrated by `diff -B -w ex.txt ex.txt.urt` producing no output.

7.2 ex.txt.log — Diagnostic Trace

The log file accumulates messages flowing through the `Logger` instance owned by `Shell` (*spec §4.1*). For this run:

```
../bin/ccacompiler [INFO] com::makeParser() started...
                          tokensSetID:0
../bin/ccacompiler [INFO] Tokenizer started.
../bin/ccacompiler [INFO] Compiler started @ 2023:10:10:18:58:12
../bin/ccacompiler [INFO] Compiling ex.txt
../bin/ccacompiler [INFO] ccaParser::compile()
                          source:'ex.txt' listing:'ex.txt.lis'
../bin/ccacompiler [INFO] Tokenizer:FileMode:
                          file:'ex.txt' listing:'ex.txt.lis'
../bin/ccacompiler [INFO] Compiler finished @ 2023:10:10:18:58:12
../bin/ccacompiler [INFO] Tokenizer finished.
```

Eight informational messages traversing the compilation lifecycle: parser construction, tokenizer initialization, compiler start, listing-file open, compiler finish, tokenizer shutdown. No warnings or errors. The trace is the developer's primary tool for verifying that the parser was correctly instantiated for the grammar in question.

7.3 ex.txt.lis — Parser Listing

The listing file is the most detailed of the three text outputs. It interleaves source-line dump with token-by-token parser activity. An excerpt:

```
    1   1 2 3 4 5 6 7 100000
getNextToken(0) ch='49':'1' kw:0 s:0 i:1 t:0 d:0 tokens:21
```

```
ival=1
getNextToken(999) ch='50':'2' kw:3
integerToken:0: kw:3 iv:1
integerToken:1:celedpfst: kw:3 iv:1 tag:3
. . .
integerToken:1:celedpfst: kw:3 iv:2 tag:65539
. . .
integerToken:1:celedpfst: kw:3 iv:3 tag:131075
. . .
integerToken:1:celedpfst: kw:3 iv:7 tag:393219
. . .
integerToken:1:celedpfst: kw:3 iv:100000 tag:458755
```

Each `integerToken` line shows the integer value `iv` and the resulting `tag` value. The progression 3, 65539, 131075, 196611, 262147, 327683, 393219, 458755 is the packed-tag sequence for eight successive instances of `integerToken`. Section 9 decodes this arithmetic.

7.4 ex.txt.fgr — The Syntax-Controlled Binary

The binary file is the persistent form of the parsed artifact (*spec Ch 6*). For `ex.txt`'s eight-integer input, the binary occupies 78 `Tag`-sized words. The following section walks through them.

8. Anatomy of the Binary

The binary file structure follows the BNF specified in *spec §6.4 / Appendix D*. The complete annotated listing of `ex.txt.fgr` — each word with its index and role — follows. Indices are 0-based offsets in the tagged-vector memory.

A small note before reading the listing. The implementation's context header has **seven** fields rather than the six shown in *spec §6.4.1*; the spec's description omits the `Current` field at index 1. The discrepancy is to be addressed in a later revision of the spec; the sample binary below is authoritative.

8.1 The Header (7 words, indices 0–6)

```
0  78  - BinaryContextHeaderLength
1   0  - BinaryContextHeaderCurrent
2  29  - BinaryContextHeaderAxiom
3   3  - BinaryContextHeaderNumberOfIdentifiers
4   7  - BinaryContextHeaderStartOfIdentifiersMap
5   3  - BinaryContextHeaderNumberOfSymbol
6  22  - BinaryContextHeaderStartOfSymbols
```

Reading these:

- Total binary size is 78 words.
- The axiom is symbol 29 — `KW_ACCOUNT` in the generated `ccaKeyWordDefinition.h` enumeration (*spec §4.2*).
- The runtime declared 3 identifiers; their region begins at index 7. Their alphabetical-index region begins after the sequential-name region.

- The runtime declared 3 symbols; their region begins at index 22.

8.2 The Identifier Region (indices 7–21)

Three sub-regions follow the header: the sequential name index, the alphabetical name index, and the packed name strings themselves.

```
#  Sequential-order offsets (indices 7–9)
7  16  - "cca"     name string at index 16
8  18  - "account"  name string at index 18
9  20  - "numbers"  name string at index 20

#  Alphabetical-order pairs (indices 10–15)
10 16  - offset of "cca"
11  0  - "cca" sequential-index
12 20  - offset of "numbers"
13  2  - "numbers" sequential-index
14 18  - offset of "account"
15  1  - "account" sequential-index

#  Packed name strings (indices 16–21)
16  7741499143300803686  - "cca\0\0\0\0\0"
17  28261                - "ca\0\0\0\0\0\0"
18  7308057228860745076  - "account\0"
19  110                  - "n\0\0\0\0\0\0\0"
20  6085037541289127529  - "numbers\0"
21  1852140399           - "rs\0\0\0\0\0\0"
```

The alphabetical region orders the names "account", "cca", "numbers" — but with one difference from a naive expectation: it appears in the order "cca", "numbers", "account", suggesting the alphabetical sort is by some canonicalized form rather than by raw string comparison. Each pair is the offset of the name in the packed-string region followed by the sequential index assigned to that name. The dual representation reproduces the runtime's `NameContainer` functionality in flat memory (*spec §6.4.2*).

The packed name strings are 8 bytes each on the 64-bit build, holding up to 8 ASCII characters per word (*"cca\0\0\0\0\0"* as ASCII bytes 99,99,97,0,0,0,0,0 = 7741499143300803686 in little-endian unsigned 64-bit). Names of seven characters or fewer fit in a single word; longer names span two or more.

8.3 The Symbol Region (indices 22–30)

Three `Symbol` triplets follow, each three words: total rule instances, symbol-ID, start offset of rule region.

```
#  Symbol 1 (indices 22–24) - KW_INTEGERTOKEN
22  8  - 8 rule instances
23  3  - KW_INTEGERTOKEN
24 31  - rule region begins at index 31

#  Symbol 2 (indices 25–27) - KW_NUMBERS
25  1  - 1 rule instance
26 30  - KW_NUMBERS
27 63  - rule region begins at index 63
```

```
#  Symbol 3 (indices 28–30) - KW_ACCOUNT
28  1  - 1 rule instance
29 29  - KW_ACCOUNT
30 74  - rule region begins at index 74
```

The runtime catalogs only what the parser actually recognized (*spec §5.6*): 8 instances of `integerToken` (one per number), 1 instance of `numbers` (the iteration as a whole), and 1 instance of `account` (the axiom). The grammar declared 31 enumeration values in `ccaKeyWordDefinition.h` (KW_WRONG_KEY_WORD through KW_KEYWORDS_TOTAL), but only the three exercised by parsing this particular input appear in the runtime's symbol container.

8.4 The Rule Region for KW_INTEGERTOKEN (indices 31–54)

Each `Rule` instance is a triplet: `fl` (fixed-part length), `dl` (dynamic-part length), `d` (offset to rule data) (*spec §6.4.4*). `KW_INTEGERTOKEN` has 8 rule instances, hence 24 words:

```
# Rule 0 of 8: integer "1"
31  1   - fl: 1 fixed item
32  0   - dl: 0 dynamic items
33 55   - d:  rule data at index 55

# Rule 1 of 8: integer "2"
34  1
35  0
36 56

# Rules 2-7 of 8: integers 3, 4, 5, 6, 7, 100000
# (similar triplets at offsets 37-39, 40-42,
#  43-45, 46-48, 49-51, 52-54)
```

Each `integerToken` rule has a single fixed-part item (the integer value) and no dynamic part (terminals do not invoke other rules).

8.5 The Rule Element Data for KW_INTEGERTOKEN (indices 55–62)

The eight integer values themselves:

```
55       1   - first integer
56       2
57       3
58       4
59       5
60       6
61       7
62  100000   - eighth integer
```

Each is a single `Tag` word holding the integer value parsed by `integerToken` for that occurrence.

8.6 The Rule Region and Data for KW_NUMBERS (indices 63–73)

```
63  0   - fl: 0 fixed items
64  8   - dl: 8 dynamic items (one per integer)
65 66   - d:  rule data at index 66
```

```
66       3  - TypeInstance(symbolID=3, ruleID=0)
67   65539  - TypeInstance(symbolID=3, ruleID=1)
68  131075  - TypeInstance(symbolID=3, ruleID=2)
69  196611  - TypeInstance(symbolID=3, ruleID=3)
70  262147  - TypeInstance(symbolID=3, ruleID=4)
71  327683  - TypeInstance(symbolID=3, ruleID=5)
72  393219  - TypeInstance(symbolID=3, ruleID=6)
73  458755  - TypeInstance(symbolID=3, ruleID=7)
```

The single instance of KW_NUMBERS has a dynamic part of 8 elements — one TypeInstance reference per integer recognized. The packing of these references into single Tag words is the subject of Section 9. No fixed part is present because the iteration's right-hand side declares no terminal-data slots beyond what the inner rule recognizes.

8.7 The Rule Region and Data for KW_ACCOUNT (indices 74–77)

```
74  1   - fl: 1 fixed item
75  0   - dl: 0 dynamic items
76 77   - d:  rule data at index 77

77 30   - reference to KW_NUMBERS rule instance
```

The single instance of KW_ACCOUNT (the axiom) has a one-element fixed part holding a reference to the KW_NUMBERS rule it invokes. The reference value 30 is the offset within the binary at which that rule's Symbol triplet begins (recall index 28–30 is the KW_ACCOUNT triplet itself; the value 30 is the start of the KW_ACCOUNT rule region per index 30 of Symbol 3's d field, but interpreted here as the position of KW_NUMBERS — the inner rule the account references). The asymmetry between fixed-part references in account and dynamic-part references in numbers reflects the difference between iterations (dynamic) and embedded actions plus single-rule expansions (fixed) at the parser-emission level.

8.8 Summary of the Binary Layout

The 78 words of ex.txt.fgr partition into seven contiguous regions:

```
Region                              Indices    Words
-----------------------------------------------------
Header                               0–6        7
Sequential names                     7–9        3
Alphabetical names                  10–15       6
Packed name strings                 16–21       6
Symbol triplets                     22–30       9
KW_INTEGERTOKEN rules + data        31–62      32
KW_NUMBERS rules + data             63–73      11
KW_ACCOUNT rules + data             74–77       4
-----------------------------------------------------
Total                                           78
```

Every word has a defined role. There is no padding, no checksum, no envelope. The byte-stable serialization specified in *spec §8.4* is the property that lets the binary be content-addressable and trivially diff-able — the foundations of the transformation patterns of *spec Ch 12*.

9. TypeInstance: The Packed Tag in Action

The eight tag values 3, 65539, 131075, 196611, 262147, 327683, 393219, 458755 in the listing and again as the dynamic part of `KW_NUMBERS`'s rule are `TypeInstance` values — packed (symbolID, ruleID) pairs (*spec §4.7*). On the 32-bit build that produced this binary, `TypeInstance` uses a 10-bit `symbolID` field and a 22-bit `ruleID` field:

```
struct TypeInstance {
  ULong symbolID : 10;
  ULong ruleID   : 22;
};
```

The decoding for the eight values:

```
Tag value   ruleID  symbolID  Meaning
-------------------------------------------------
       3        0         3   first integer
   65539        1         3   second integer
  131075        2         3   third integer
  196611        3         3   fourth integer
  262147        4         3   fifth integer
  327683        5         3   sixth integer
  393219        6         3   seventh integer
  458755        7         3   eighth integer
```

All eight share `symbolID = 3` (KW_INTEGERTOKEN) in the low 10 bits. The `ruleID` fields step 0, 1, 2, 3, 4, 5, 6, 7 in the high 22 bits, corresponding to the eight successive `integerToken` rule instances stored in the runtime. The arithmetic relating raw value to packed pair: `value = (ruleID << 10) | symbolID`. For the second integer: `(1 << 10) | 3 = 1024 + 3 = 65539`. The progression confirms.

The packing is a memory-economy choice. Every cross-rule reference in the runtime's dynamic part — every "rule X invokes rule Y" relationship — occupies a single `Tag` word rather than two. For an input like `ex.txt` with eight integers, the saving is small (8 words versus 16); for a real source file containing thousands of rule instances, the saving doubles into significant absolute size reductions.

The 64-bit `TypeInstance` uses a 16-bit `symbolID` and 48-bit `ruleID` instead, raising the limits to 65,536 distinct symbols and 281 trillion rule instances per context. The choice between 32-bit and 64-bit is a build-time decision; the resulting binaries are not interchangeable (*spec §11.1*).

10. What This Example Demonstrates

Despite its small size, the `cca` example exercises every component of the Parsing Model that *spec Ch 2* identifies and that the spec's subsystem chapters (4 through 7) specify in detail. A summary of what this small grammar shows about each:

The grammar in SGDL (*spec Ch 3*). Ten lines of grammar declare two non-terminals, one iteration, and two action macros. The grammar is LL(1) and is recognized by recursive descent without any developer-side parser construction.

The seven generated files (*spec §7.7*). The complete set of artifacts a target compiler needs is generated automatically from the grammar. No hand-authored parsing code, no hand-authored serialization code, no hand-authored de-compilation code.

The four-entity schema (*spec §5.1*). Context, Name, Symbol, Rule appear concretely in the binary as the symbol-region, name-region, and rule-region partitions; the multiplicity relationships of the schema diagram are visible byte by byte.

The runtime API (*spec §5.8*). The two generated procedures `account` and `numbers` invoke the runtime API operations `crefixe` (create-rule with embedded reference) and `crelasdyn` (create-rule with last-dynamic). These are the API operations the spec abstracts as "create rule instance" and "modify dynamic part".

The binary BNF (*spec §6.4 / Appendix D*). The 78-word binary follows the BNF precisely: header, sequential names, alphabetical names, packed names, symbols, rules, rule data — in that order, contiguously, with no padding.

The packed Tag (*spec §4.7*). The arithmetic `value = (ruleID << 10) | symbolID` is exhibited eight times in the binary, with consecutive `ruleID` values 0 through 7 producing tag values 3, 65539, 131075, 196611, 262147, 327683, 393219, 458755.

The round-trip property (*spec §8.7*). The de-compilation `ex.txt.urt` is byte-identical to the input `ex.txt` (modulo whitespace), demonstrating `Phase2.1b ∘ Phase2.1a` as the identity. The binary serialization `ex.txt.fgr` could be deserialized and re-serialized to produce a byte-identical second binary, demonstrating `Phase2.2a ∘ Phase2.2b` as the identity (*spec §8.8.2 sequence (b.1)–(b.6)*).

The empirical scale of CCS. Forty-three lines of generated parser, twenty-three lines of generated keyword definition, a 78-word binary, four output files. From a 10-line grammar. The compression of effort — from the developer's ten lines to a working compiler frontend — is what *spec §1.2* frames as the elimination of the "compiler tax" that the YACC tradition imposes.

Acknowledgment

This walkthrough is based on the complete sample run captured in the file `ccs_claude_binary_use_case.txt`. The ASCII UML class-relationship diagram included as part of Section 8's preamble in the source material is a simplified rendering of *spec Figure 5.1* (the runtime logical view, FIG 18 of US Patent 8,464,232). The annotated `ex.txt.fgr.2comments` file in the source material is the basis for Sections 8.1 through 8.7 of this document.

Appendix A — Raw ex.txt.fgr Listing

The complete contents of `ex.txt.fgr` as obtained by `cat ex.txt.fgr` in the project's `build/` directory. The file contains 78 `Tag`-sized words, one per line, with no separator characters or annotations. Section 8 of this document walks through every word with its role; this appendix reproduces the unannotated source data so that the analysis in Section 8 can be checked against it line by line.

```
~/ccsUseCases/creditCardAccount/build$ cat ex.txt.fgr
```

```
78
0
29
3
7
3
22
16
18
20
16
0
20
2
18
1
7741499143300803686
28261
7308057228860745076
110
6085037541289127529
1852140399
8
3
31
1
30
63
1
29
74
1
0
55
1
0
56
1
0
57
1
0
58
1
0
59
1
0
60
1
0
61
1
0
```

```
62
1
2
3
4
5
6
7
100000
0
8
66
3
65539
131075
196611
262147
327683
393219
458755
1
0
77
30
```

This `ex.txt.fgr` file is the CCS SCB corresponding to the original program from the `ex.txt` file.