The Koopa Cobol Parser Generator

Kris De Schutter

Abstract

Koopa is a Cobol parser generator with a plan for growth. It is able to handle Cobol source files in isolation (no preprocessing required) and accepts CICS/SQL fragments. Due to its design it is easily extensible in a way which limits the impact on the overall project. It achieves this by means of a custom DSL for specifying Cobol island grammars in a concise way, and through a unit testing framework for such grammars which aids in rapid and accurate fault detection. This is complemented by support for both ad-hoc and serial handling of the generated syntax trees in a structure-shy manner.

1 Some background

The Koopa project was born out of a need to process industrial Cobol source code for a variety of purposes. One was the verification of industrial software with respect to an architectural requirements document [1]. Another was the extraction of interdependencies between programs, subprograms and copy books in such systems.

Due to my research into legacy systems I had some prior experience with the creation of Cobol parsers. In fact, the Koopa system, as it is now on Sourcefourge, would be the fourth or fifth attempt. One was based on cobc (now OpenCobol[1]). Another two were built using the grammar tools from the Vrije Universiteit Amsterdam, on which they had already built the VS-Cobol-II parser (see [3]). All these versions had one major drawback: they tried to be *complete* Cobol parsers. This proved to be too much of a problem with the Cobol code we had to process.

Our input consisted of complex source code making use of SQL and CICS[2] extensions. In addition, we only had access to the source code for part of the system, not all of it. We couldn't really count on having a complete compilable (or even preprocessable) module. What we really needed was a parser with the following properties:

- **One file at a time.** The parser should be able to process Cobol source files in isolation. In particular it should not assume that the sources have been preprocessed. This helps us deal with incomplete code bases.
- **Accept "foreign" (CICS, SQL,...) fragments.** While the parser does not need to understand the contents of these fragments, it must be able to handle and track their presence. This is needed, for instance, in control flow analysis as CICS fragments may hold calls to subprograms.
- **No spurious development effort.** The time spent developing the parser should be in proportion to the subset of statements holding the required information, not

[1] `http://www.opencobol.org/`

[2] CICS (Customer Information Control System) is a transaction manager for online processing.

the entire Cobol language. Put differently: we do not want to have to specify a complete Cobol grammar unless we need every single piece of information that's in the source code.

- **Extensible.** The parser should be extensible when the need for extracting new kinds of information establishes itself. The effort required should again be in proportion to the subset of statements holding the information we need, not the entire Cobol language.

These requirements drove the development of Koopa. It is the reason we based it on a very different technology: *island grammars*. Island grammars allow partial definition of grammars where most of the input is simply skipped. This property already solves most of the problems we had, as we will now see.

2 Island grammars

Of the requirements set forth in the previous section, the most difficult one to deal with is that of effort: how do we get a parser going in such a way that we only have to specify those parts we're interested in? As we said, the solution was found in *island grammars*. Moonen [4] defines these as follows:

> An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the *islands*) and liberal productions that catch the remainder (the *water*).

Figure 1 illustrates how this works in practice. On the left is a simple Cobol program (taken from [2]) with all reserved keywords displayed in bold. It should be clear even from this simple program that there are too many keywords to be handled, to consider writing a full parser. (A full Cobol parser typically has to deal with over 500 such keywords, many of which are context dependent.) In contrast, figure 2 shows the approach taken by an island-based parser where we only care about reconstructing the control flow. As can be seen, most of the code is now ignored (the *water*, shown in gray). Instead the parser scans for the `PROCEDURE DIVISION` marker, and then looks out for verbs (e.g. `OPEN`, `SET`, `MOVE`, etc.) and dots (shown underlined) from which we can deduce the control flow (the *islands*). The effort required here is reduced to but a few keywords, and a few structural rules (see figure 3, with details in the next section).

By skipping things we don't care about we can also see how to handle the one-file-at-a-time and foreign-fragments requirement: anything related to the preprocessor (in Cobol this is mainly about `COPY` statements), or to CICS/SQL, is simply put into the water. While in theory this might make the remainder of the source code badly formed (due to important parts being inside "copy books" or include files), in practice this is rarely a problem.

3 Writing island grammars

Defining an island grammar requires the definition of two things: the islands and the water. Consider again the example in figure 2. The island grammar alluded to here would look something[3] like figure 3. Assuming that the tokenizer backing this grammar

[3] The example uses an EBNF-like syntax, similar to those found in most parser generators, but not from a specific generator.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
   ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
   FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
   01 LOGFILE-RECORD PIC X(2048).
 WORKING-STORAGE SECTION.
   01 LOGFILE-STATUS     PIC 9 VALUE ZERO.
      88 LOGFILE-IS-OPEN       VALUE 1.
 LINKAGE SECTION.
   01 LOGFILE-ENTRY.
      05 LOGFILE-VERB  PIC X(12).
      05 LOGFILE-NAME  PIC X(32).
      05 LOGFILE-DATA  PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
 OPEN EXTEND LOGFILE
 SET LOGFILE-IS-OPEN TO TRUE.
 MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
 WRITE LOGFILE-RECORD.
 GOBACK.
```

Figure 1: A Cobol program (based on [2]) showing all keywords in bold.

would only return tokens visible in the grammar (i.e. `"PROCEDURE"`, `"DIVISION"`, `DOT`, etc.), then this definition would be sufficient to handle the example code. The `water` definition eats up any dot occurring before the procedure division. Once beyond that point we look for series of statements, where we recognise a statement by looking for verbs. Everything is quite straightforward (especially as we ignore nested statements in this example), with the required effort being very low.

Let us now consider the extensibility of this grammar. Assume we want to add support for `CALL` statements, where we want to know which external programs get invoked. A simple grammar rule for this statement would be:

```
call ::= "CALL" (STRING | IDENTIFIER)
```

Integrating this into the island grammar of figure 3 is not as straightforward as adding it as an extra alternative to `statement`. The reason is that the grammar will now have to deal with two new tokens: `STRING` and `IDENTIFIER`. These two, however, can occur in any number of places, which means we will have to handle more water in our grammar. Figure 4 presents a possible updated version. Note that not only did we have to extend the definition of water, we now have two different types of water depending on their location in the grammar: before the procedure division all dots should be ignored, and are therefore in the water; inside the procedure division, however, we need the dots to recognise the end of statements, and so they become part of the islands. We also had to extend existing grammar rules with the new occurrences of water, which were previously handled by the tokenizer.

The extent of the effort required for extending the grammar is now no longer based on the rule one wants to add, but on the number of existing rules. Any new rule will

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
   ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
   FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
   01 LOGFILE-RECORD PIC X(2048).
 WORKING-STORAGE SECTION.
   01 LOGFILE-STATUS     PIC 9 VALUE ZERO.
      88 LOGFILE-IS-OPEN       VALUE 1.
 LINKAGE SECTION.
   01 LOGFILE-ENTRY.
      05 LOGFILE-VERB  PIC X(12).
      05 LOGFILE-NAME  PIC X(32).
      05 LOGFILE-DATA  PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
 OPEN EXTEND LOGFILE
 SET LOGFILE-IS-OPEN TO TRUE.
 MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
 WRITE LOGFILE-RECORD.
 GOBACK.
```

Figure 2: The same Cobol program as seen through the eyes of an island grammar with a focus on control flow. Everything in grey is water and gets ignored. The bold/underlined parts are the islands used to reconstruct the control flow.

```
source ::=
  (water)*
  "PROCEDURE" "DIVISION" DOT
  (sentence)*

sentence ::= (statement)* DOT

statement ::= verb

verb ::=
 "OPEN" | "SET" |
 "MOVE" | "WRITE" | "GOBACK"

water ::= DOT
```

Figure 3: Island grammar for the view taken in figure 2.

```
source ::=
  (water-1)*
  "PROCEDURE" "DIVISION" (water-2)* DOT
  (sentence)*

sentence ::= (statement)* DOT

statement ::= call | verb (water-2)*

call ::= "CALL" (STRING | IDENTIFIER)

verb ::=
  ( "OPEN" | "SET" | "MOVE"
  | "WRITE" | "GOBACK" | "CALL" )

water-1 ::= DOT | STRING | IDENTIFIER
water-2 ::= STRING | IDENTIFIER
```

Figure 4: Extended version of the grammar in figure 3 for dealing with `CALL` statements.

have to deal with tokens established by existing rules and, worse, every existing rule will have to deal with any tokens established by the new rule. This quickly puts a cap on what can be added in a reasonable amount of time.

It is important to note that the problem we find here is not related to the concept of island grammars, but rather to the limited expressiveness offered by classic parser generators. What we need here is a way to define water such that it can evolve along with the remainder of the grammar *without further help from the developer*.

4 Koopa: island grammars using a custom DSL

In order to be able to better handle definitions of water in island grammars for Cobol we have opted for establishing a custom DSL, named Koopa, for specifying such grammars. Koopa works very much like classic EBNF-style languages, but makes some optimisations for Cobol specifications. For instance, it is not necessary to quote keywords. Every uppercase identifier is considered to be a keyword to be matched against the input. This fits nicely with the usual Cobol style, where uppercase is the default rather than the exception. Dots can also be written down freely, and will be matched against the input.

4.1 Support for skipping water

The most important contribution is the handling of water. Figure 4.1 presents a Koopa version of the grammar found in figure 4, and presented in the previous section. Every occurence of water has been replaced by a construct which looks like this:

```
[--> pattern]
```

The `-->` operator is the *skip-to* operator, and does as its name implies: it skips anything it encounters up to the provided pattern. The pattern itself is not consumed by the

```
def source =
    [--> PROCEDURE DIVISION]
    PROCEDURE DIVISION [--> .] .
    (sentence)*
end

def sentence =
    (statement)* .
end

def statement =
    ( call
    | verb [--> (verb | .)]
    )
end

def call =
    CALL (string | identifier)
end

def verb =
    ( OPEN | SET | MOVE
    | WRITE | GOBACK | CALL )
end
```

Figure 5: Koopa version of the grammar in figure 4. Uppercase identifiers are keywords to be matched. Lowercase identifiers refer to other rules. `-->` is the "skip-to" operator, as explained in section 4.

parser. The advantage here is that no matter how the grammar grows, whatever new tokens have to be parsed, as long as a good marker pattern is chosen then you won't have to change anything in the existing grammar rules.

As a proof of concept, consider adding support for the `GO TO` statement. We add the definition in figure 6 to our grammar. Despite the addition of new tokens (e.g. `DEPENDING`, `ON`) the only extra required modifications are to add this rule as an alternative in `statement` (so that it is used in the parsing), and to add `GO TO` as another verb (if it was not already listed). Nothing else needs to change. Specifically, no other rules have to be updated to handle the new tokens (e.g. `DEPENDING`, `ON`), and the `goto` rule is not bothered by anything defined by existing rules.

```
def goto =
    GO TO identifier
        [ (identifier)+
          DEPENDING [ON] identifier ]
end
```

Figure 6: Definition of `GO TO` statements. Anything between square brackets is considered optional.

```
  *> These are valid:
2 01 FILLER   PICTURE 9999   VALUE   ZERO.
  01 FILLER   VALUE   ZERO   PICTURE 9999.
4 *> This is not valid:
  01 FILLER   PICTURE 9999   PICTURE 9999.
```

Figure 7: An example of permutation of clauses in Cobol.

4.2 Support for rule permutations

Koopa also has a provision for dealing with "permutation" of a set of grammar rules. That is, in Cobol it is possible in some locations for a certain set of grammar rules to appear in any order. A typical example here is the different set of options on data definitions. Figure 7 shows this. The first two data definitions (lines 2 and 3) are basically equivalent: both define a four digit data item with initial value set to zero. The order in which these clauses appear is not important. What is important is that no clause may appear more than once. It is for this reason that the definition on line 5 fails, even though both picture clauses specify the same picture.

The obvious way of dealing with this in a classic parser generator would be to define the grammar rule along the following lines:

```
level-number data-name
  ( picture | value )*
```

This would parse the correct definitions, but would also accept definitions with duplicate clauses. A fix would be to set flags on every clause and throw an exception whenever a clause is seen again after its corresponding flag has been set. This, however, is tedious, error-prone. It also makes the specification of the grammar less readable as it gets mixed with native code to make it work.

Koopa overcomes this problem by establishing a special operator for dealing with permutations. A Koopa definition which would correctly accept the input in figure 7 is:

```
level-number data-name
  !( picture | value )
```

In Koopa, `!` is the permutation operator. It is followed by a list of alternative grammar rules. Any input where these grammar rules match, gets accepted, but only if no rule matches more than once (zero matches are also allowed). In this way our specification for data items remains concise and to the point, without the need for native code to make it work.

4.3 A "not" guard

Cobol grammars are notoriously ambiguous. Here's a telling example coming from the INSPECT statement: the TALLYING phrase[4].

```
TALLYING ( identifier-1 FOR
  ( (ALL | LEADING) (identifier-2 | literal)*
  )*
)*
```

[4] Simplified here for illustration purposes.

So, having seen (for instance) a `TALLYING A FOR ALL B`, how would a following `C` be matched ? Well, Koopa's repetition is greedy so it would match up this `C` with `identifier-2`. But that's not always correct:

```
TALLYING A FOR ALL B
         C FOR ALL D
```

In this case `C` should have been matched to `identifier-1`.

We can solve the ambiguity in the TALLYING phrase if we say that `identifier-2` can't be followed by a `FOR`. When an identifier is followed by a `FOR` we know that we should match it to `identifier-1` instead. Koopa can solve this problem through a "not" guard:

```
TALLYING ( identifier-1 FOR
  ( (ALL | LEADING) (identifier-2 -FOR | literal)*
  )*
)*
```

The dash preceding the `FOR` in the above grammar rule is the "not" guard. It specifies that whatever token directly follows this guard should not appear at that point in the token stream. If it does then the match fails for that alternative. If it doesn't then we can continue matching. Note that the "not" guard does not consume the token it's testing! That token is still available for matching by the remainder of the grammar rule.

5 Tokenising Cobol

In order to keep grammar rule interactions to a minimum Koopa tries not to distinguish different types of tokens until the very last moment[5], and then only does so based on the grammar rule. Consider again the definitions in figure 4.1. The tokenizer in Koopa will not distinguish the `CALL` verb from generic identifiers. Only when the grammar needs to know "Is the next token a `CALL` keyword ?" will the tokenizer check if it is. The advantage here is that keywords are always local to grammar rules, and will therefore never interfere with other rules.

To make this work Koopa provides a specialised tokenizer which returns a stream made up of basic tokens with some metadata in the form of tags. It is only in a later phase (by request of the parser) that the differentiation of keywords versus identifiers is made. An added advantage of this approach is that the tokenizer does not have to co-evolve with the grammar, which again reduces the amount of effort for developing and extending the Cobol grammar.

6 Unit testing Koopa grammars

In order to detect problems as early as possible, thereby keeping the required effort for extending grammar definitions further in check, Koopa provides support for unit testing grammar rules. In Koopa, any grammar rule may be accessed as if it were a full parser. It can then be used to parse any piece of relevant input; it need not be a full source file.

Consider again the `goto` rule from figure 6. Given this definition we can now set up a set of tests, as is shown in figure 8. Line 1 of figure 8 declares which grammar rule

[5] Notable exception are literals (strings and numbers), which are never context dependent.

```
target goto;

+[ GO TO SUB-A ]
+[ GO TO SUB-A SUB-B DEPENDING ON A-VALUE ]
+[ GO TO SUB-A SUB-B DEPENDING A-VALUE ]
```

Figure 8: Unit test for the grammar rule in figure 6.

```
target addStatement;

+[ ADD A TO B # . ]
+[ ADD A TO B GIVING C # . ]
```

Figure 9: Unit test for a partially defined Cobol construct.

will be tested. Lines 3, 4 and 5 each define a single test. Everything between square brackets is used as tokens to be parsed, in free format[6] Cobol. The leading + (or –) flags whether the parser should accept this input (or not).

Unit testing is not limited to grammar rules for which you have a complete definition. You can test rules which rely on skipping tokens to do their work. The problem is that this skipping is not greedy, but the unit tests do expect you to consume all tokens. If there are tokens left, this is considered an error. You can overcome this by giving only a partial input, but it is nicer to go for the option of marking the end of what gets consumed by your parser. Figure 9 shows an example. The "sharp" sign marks the exit point. So in this case there should always be one token left after each test.

These unit tests, together with the grammar, should quickly tell a developer whether or not he or she is breaking any existing feature of the grammar and, more importantly, where exactly he or she is breaking it. Being able to pinpoint any problems as fast as possible is another key factor in keeping development overhead in check.

7 Grammar development cycle

The development cycle for Cobol island grammars in Koopa follows an iterative waterfall model, with a limited number of steps:

1. Define and add the new grammar rule. You can do this based on existing documentation, code examples, or even on existing definitions such as those found in [3].

2. Add unit tests, and run them. This includes running all other unit tests! Experience shows that most problems here are with respect to the newly added rule, which is ideal. If problems occur in existing rules, then any fix should include a new test case which tests the problem encountered.

3. Run the full parser on a representative body of code. Ideally this is the code for whatever project you're working on. This will help you establish support for that code as fast as possible. In addition, we also run tests on a Cobol 85 test suite[7].

[6] Fixed format samples are also allowed. The syntax for these uses curly braces instead of square ones.

[7] `http://www.itl.nist.gov/div897/ctg/cobol_form.htm`

```
tree grammar MyAdaptiveTreeParser;

options { /* ... */ }

compilationGroup : compilationUnit* ;

compilationUnit : ^(COMPILATION_UNIT PROGRAM_NAME) ;
```

Figure 10: Island tree grammar in ANTLR.

And, finally, repeat as needed. Given enough iterations the grammar under development will approach a full Cobol grammar; that is, one where there is no more water left. This, however, is an incidental feature of our approach, not a required one.

8 Koopa back-end

The back-end provided by Koopa is left open. The default output is a tokenstream tagged with the names of grammar rules. This is easily transformed into an abstract syntax tree. In fact, Koopa already provides support for building an XML DOM tree or an ANTLR[8] syntax tree. The first can be queries with XPath expressions, the latter may be processed using ANTLR tree parsers. For convenience Koopa also knows how to generate a DTD for the generated XML, as well as a basic ANTLR tree grammar from the Cobol island grammar.

Here again we encounter a possible maintenance problem: ANTLR tree parsers have to be complete and exact. So if we want to process the abstract syntax tree we're again faced with setting up a complete parser. And while Koopa knows how to generate such a tree grammar from the Koopa grammar, it does not know how to update an existing one. So updating the Koopa grammar will likely break the tree grammar.

Koopa provides a solution for this problem by again going back to island grammars. It is possible in Koopa to define an island ANTLR tree grammar. Figure 10 shows an example of such a grammar which only looks at the compilation units defined in a source file. Koopa will analyse this grammar and generate a filter which can be applied to the abstract syntax tree. This filter filters out any tokens which the ANTLR generated tree parser would not expect. For the example this would mean anything which is not a `COMPILATION_UNIT` or a `PROGRAM_NAME`. In essence it generates a run-time view on the full abstract syntax tree so that your tree parsers see what they expect to see. The result is that changes in your Koopa grammars will mean changes in the abstract syntax trees, but will not necessarily require you (or others) to update your (or their) tree grammars.

9 Discussion/Conclusion

I realise that there are a lot of claims in these pages and I feel that I should back them up with some real data.

The generated parsers are able to process real-life industrial Cobol code, warts and all, even if you do not have a complete system. I know this because I have had the

[8] `http://www.antlr.org/`

chance to test it against several real-life examples, totalling several million lines of code. I have also had reports from several users, one of which who ran Koopa on a set of 20.000 Cobol programs and another 20.000 copybooks. So I'm not worried there.

I know that the island grammar approach helps a lot in the piecemeal definition of Cobol grammars. I know this because I have tried it both ways. I'm able to contrast it agains a non-island grammar approach, using the grammarware toolkits from the Vrije Universiteit Amsterdam, YACC and basic ANTLR. The grammars I made in these environments have proved much harder to maintain. I have also had a few users who successfully added to the Cobol grammar on their own, and even contributed some changes to the project. So I'm confident that the grammar can grow. Still, we're far from a full Cobol grammar in Koopa, and I don't know how it will cope with such a thing. But I'm optimistic.

The grammar unit testing is also a very welcome tool. It allows you to make changes to the grammar, and to verify (to some extent) that you haven't broken previous work. And if you do happen to break it you can quickly spot where it went wrong. My experiences with previous toolchains were never this nice. I would always have to run the complete parser on a complete Cobol file, and if something went wrong spend a long time trying to figure out why it went wrong. True, the quality of testing in Koopa stands and falls with the quality of the unit tests written by the developer, but it is much easier to set up unit tests which cover all eventualities in Koopa than it is by writing a full Cobol file.

The biggest question mark is the back-end. I have only had to do simple processing so far, and for that the solution using the ANTLR island tree grammars worked just fine. The XML DOM tree is also something which is being used by others. Other than that all suggestions are welcome.

References

[1] A. Kellens, K. De Schutter, T. D'Hondt, L. Jorissen, and B. Van Passel. Cognac: A framework for documenting and verifying the design of cobol systems. In A. Winter, R. Ferenc, and J. Knodel, editors, *CSMR*, pages 199–208. IEEE, 2009.

[2] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In *Aspect-Oriented Software Development (AOSD)*, pages 99–110, 2005.

[3] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[4] L. Moonen. Generating robust parsers using island grammars. In *WCRE*, page 13, 2001.