University of Amsterdam
Programming Research Group

```
REQUIRED-KEYWORD file-name [IS NOT OPTIONAL] { REQUIRED-CHOICE-1 } ...
                                             { REQUIRED-CHOICE-2 }
```

```
"REQUIRED-KEYWORD" FILE-NAME OPTIONAL CHOICE+ → CONSTRUCT
[A-Z]+                                        → FILE-NAME
OPT-IS "NOT" OPT-OPTIONAL                     → OPTIONAL
                                              → OPT-IS
"IS"                                          → OPT-IS
                                              → OPT-OPTIONAL
"OPTIONAL"                                    → OPT-OPTIONAL
"REQUIRED-CHOICE-1"                           → CHOICE
"REQUIRED-CHOICE-2"                           → CHOICE
```

ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool

Peter Manuel

Report P9617 November 1996

University of Amsterdam

Department of Computer Science

Programming Research Group

ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool

Peter Manuel

Report P9617 November 1996

P.I. Manuel

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7584
e-mail: manuel@fwi.uva.nl

Universiteit van Amsterdam, 1996

ANSI COBOL III in SDF
+
an ASF definition of a Y2K tool

Peter Ivan Manuel
University of Amsterdam
Faculty of Mathematics, Computer Science,
Physics & Astronomy
Thesis supervisor: Paul Klint
Supervisor: Mark van den Brand

November 19, 1996

Abstract

COBOL programs are heavily used in business applications. With the arrival of the year 2000, companies using large software systems, often based on (some dialect of) COBOL, are facing serious problems. A formal specification of the COBOL language can perhaps assist in solving these problems. This thesis gives an overview of the ANSI standard definition of COBOL III, presents an algebraic specification of the syntax of COBOL III, and describes its specification process. Furthermore, it introduces a year 2000 reverse engineering tool based on `PICTURE` clauses. Finally, it discusses related work (including a different approach to specifying COBOL), and gives suggestions for future work.

Contents

Chapter 1

Introduction

1.1 The problem

The COBOL programming language does not provide a predefined date-variable or date-type in its language definition; in fact, it provides almost no typing mechanism at all. There has been no standard set of rules on how to denote dates either: programmers were free to choose any representation. As a result every two pieces of COBOL code in a large program (or system of programs) can have different notations for date fields. Some programmers through time may have used two digits to represent the year, others four; separators between day, month, and year may have been periods, slashes, dashes, or nothing at all; Julian dates may have been used; and so on.

With the approach of the year 2000 this has become a serious problem for the information technology industry. It will often be impossible to work with '00' as representation for '2000' because, for example, all kinds of interest calculations and sorting operations get messed up. If in need of money one could try making a phone call starting slightly before and lasting until slightly after 01-01-2000. There is some reason to expect a credit note for a phone call lasting for nearly -100 years (be sure to make it long-distance)[1].

No need to wait for the turn of the century however, the problem is here today. A lighthearted example is the case where in 1992 a 104 year-old Minnesota woman received an invitation to attend kindergarten (due to the inability of the software system to discern that '88' meant 1888, rather than 1988) [21]. Without a doubt many more serious problems have occurred and will continue to occur before the year 2000.

In short, the year 2000 problems are caused by a misinterpretation: the software considers '01-01-00' to be the first day of 1900, not 2000[2].

The obvious starting point for a solution is to track down those parts of the code that directly or indirectly modify date fields. Although a simple text scan will find the positions in the source program where possible dates are defined, it soon becomes hard to detect more subtle data-flow dependencies. Moreover, linearly scanning for potentially relevant strings like 'year' will produce innocent occurrences as well, including but not limited to trivial comment entries. As the number of 'false alarms' increases, the efficiency (and therefore usefulness) of a tool is reduced.

To effectively analyze millions of lines of decades-old code a tool will need syntactic and semantic 'knowledge' of the source language. An algebraic formalism can

[1] Many more examples can be found on the internet, for example on the IBM year 2000 pages [1] and the Peritus year 2000 pages [2]. See Section 5.3 for a more elaborate discussion of the problem.

[2] Note that the problem has more to do with a change of the century than with a change of the millenium.

express this knowledge, and tools based on the knowledge can be built within the formalism.

Some other important problems in this context are the Euro-problem, caused by the planned introduction of the 'Euro' currency in the year 2002, and the update-problem many companies have migrating from old COBOL versions to newer, possibly object oriented ones. Here too, formal specifications and tools can be helpful.

1.2 The approach

In this thesis the algebraic specification formalism ASF+SDF [14] is used to specify the syntax of COBOL III[3], the 1985 version of the ANSI[4] COBOL standard [9].

We used a top-down approach, starting out with ANSI's huge description (referred to as 'the standard' in this thesis) and developing the specification from there. In [15] a bottom-up approach is used, taking example programs and building the specification according to the requirements set by these programs. Because most examples used in [15] were written in COBOL 1974, the two specifications differ in a more fundamental way as well. This last difference could perhaps, in the long run, result in tools that assist in upgrading COBOL programs from the 1974 version to the 1985 version. Our approach, in contrast with the bottom-up approach, does not assume that the COBOL programs under analysis are syntactically correct. That is, we implemented every syntactic check we could find rules for in the standard, and time for in our project.

After writing the SDF definition of COBOL, we developed a tool for analyzing COBOL programs and detecting year 2000 relevant variables. Because the format of each variable in a COBOL program is declared in a `PICTURE` string, it is essential to find those strings and to determine the format. The ANSI [10] and ISO [6] standards of date formats can be starting points for tools that track down possible date fields in old COBOL code. To assist in the construction of our reverse engineering tool the semantics of the `PICTURE` clause has been specified.

Because COBOL is supposed to be very strict on formatting rules, a pretty printer might (in a future project) be generated using the pretty printer generator described in [23]. When our tool does some program transformation on a piece of COBOL code, the pretty printer can be used to format the output in such a way that the COBOL compiler (where the original version of the program was compiled) will be able to once again process it.

1.3 Structure of this thesis

This thesis first gives an overview of ANSI COBOL III (Chapter 2), focussing on features that will be of importance throughout the thesis. In Chapter 3 ASF+SDF is introduced, followed by a description of the syntax specification process in Chapter 4: the problems encountered and the choices made. In Chapter 5 the specification of the syntax and semantics of the `PICTURE` string is presented, together with a reverse engineering year 2000 tool based on this specification (including recommendations for future extensions). Then, in Chapter 6, related work is discussed, including a comparison of our approach with the bottom-up approach. Finally, suggestions for (a lot of) future work are presented in Chapter 7. The ASF+SDF code for the syntax specification and for our year 2000 tool are included as appendices.

In this thesis many references to the ANSI standard are made. For this, we use the following notation: [page VI-III] indicates page III from section VI of [9]. A

[3]This notation style is introduced by us and is 'not standard'.

[4]American National Standards Institute.

similar notation is used for references to more than one page, and for references to sections.

1.4 Acknowledgements

Mark van den Brand and Paul Klint suggested the subject of this thesis to me. Apparently, COBOL had not been given too much attention until then, leaving me with a lot of possible approaches. Mark van den Brand supervised the writing of this thesis, read several draft versions, and provided useful ideas at many occasions. Leon Moonen introduced me to the idea of the data-flow propagation technique that I used for our year 2000 tool. Maarten van der Graaf showed me how to reduce the number of equations using an optional construct in the ASF-part of a specification. I had some highly instructive conversations with Alex Sellink and Chris Verhoef regarding COBOL and the year 2000 problem. Eelco Visser helped me out with several LaTeX-related problems.

Chapter 2

Overview of COBOL III

2.1 History

Born on may 28 and 29 of 1959 [page XVII-1], COBOL (COmmon Business Oriented Language) is still an important programming language. Perhaps not because of its technical qualities, for in spite of updates and revisions it is hardly a state-of-the-art language at this point, but because over the years COBOL (living up to its name) has grown to become the most commonly used programming language for business data processing applications. It is estimated that currently about thirty million man years of COBOL code is running worldwide [3].

The first ANSI COBOL standard appeared in 1968 [7], the second, 'COBOL II', in 1974 [8]. Our specification is based on the third version: COBOL III, published in 1985 [page VII-13], a major revision of the 1974 version. Several updates on this version took place in 1989. Corrections of the 1985 and 1989 versions were published in 1993; these corrections and the 1989 extensions have since been included in the 1985 standard.

Many compilers implementing some dialect of COBOL have appeared; some of these are described in Section 4.3. In 1995 IBM launched a new object oriented COBOL version[1], based on the proposed object oriented ANSI COBOL standard planned for publication in 1998. However, most of today's COBOL code is still written in dialects based on either COBOL II or COBOL III. It is expected that in the (near) future many companies will try to convert COBOL II and COBOL III - based programs to object-oriented COBOL[2].

2.2 Language structure

A COBOL program can consist of four *divisions* and is optionally terminated with an 'end-program-header'. The four divisions [page VI-3] may appear in this fixed order only:

1. **Identification division:**
   contains global information about the program.

2. **Environment division:**
   contains information about the target computer the program will run on.

[1] *VisualAge for COBOL*.
[2] This too will require tools for analyzing and converting COBOL code.

3. **Data division:**
   contains the definitions of records, variables, etc. for use in the procedure division.

4. **Procedure division:**
   contains the procedural code of the COBOL program.

The identification division is used for administrative remarks about the program only, such as program name, author, last compilation date, etc. All these information fields are obsolete, see Section 2.5.4, except for the program name field. The identification division is the only required division in a COBOL program; the other three are optional.

The environment division is used to define the hardware platform. Or, as the standard puts it [page IV-31]:

> "The environment division specifies a standard method of expressing those aspects of a data processing system that are dependent upon the physical characteristics of a specific computer."

In the environment division a programmer can specify the computer's identity and its memory sizes, but (s)he can also choose a representation for the currency sign, choose a comma for the decimal point, etc. A change of hardware-platform should, according to the standard, require modification of this module only.

The data division defines the data that is to be processed by the procedure division. Record structures can be defined here, and it is possible to initialize variables. Several other features are available to the programmer; for example, the possibility to assign file variables to physical filenames.

The environment division and the data division contain the following functional sections:

Environment division:

- **configuration section**
  deals with the characteristics of the source computer (where the program is compiled) and the object computer (where the program will be executed).

- **input-output section**
  deals with information needed for transmission and handling of data between external media and the program.

Data division:

- **file section**
  defines the structure of data files and corresponding records.

- **working-storage section**
  defines records that are strictly used for internal processing, not for external I-O.

- **linkage section**
  occurs in the called program in inter-program communication, describes data items that are to be referred to by the calling program, and by the called program(s).

- **communication section**
  defines the data items in the source program that serve as an interface while

communicating with the *message control system* (an external piece of software handling the processing of messages). Observe the difference with the linkage section: the linkage section occurs only in a 'called' program, and represents a much more direct way of data-sharing. Using the linkage section might be compared to applying the 'shared-memory' approach in parallel computing, while using the communication section corresponds to applying the 'message-passing' approach.

- **report section**
  describes the contents and the format of reports. Reports are pages of output containing COBOL code represented in one of several available formats, see [page II-8] for an overview of the standard's *Report Writer* features.

Each of these sections in both divisions is optional, and each can contain (pre-defined) *entries* that take care of different parts of the functionality of the section. As an example, the file section of the data division can contain the following three entries [page V-8].

- File description entry
- Sort-merge file description entry
- Report file description entry

For example, the sort-merge file description entry describes the information related to a sort-merge file used in the source program. Each of these three entries has to conform to a different set of syntax rules (every file-type has its specific syntactical requirements).

2.3 COBOL statements

Order of execution The procedure division contains the statements of a COBOL program. Many different statements in many different forms exist (see [pages iii..x] for a list). Execution starts at the first statement encountered in the procedure division, then the next etc., unless some other order is indicated.

Structuring mechanisms An important structuring concept in the procedure division is the *sentence*: a procedure division contains user-defined sections, sections contain (user-defined) paragraphs, and paragraphs contains sentences. A sentence is a sequence of statements terminated by a period. At first glance sentences might appear very much like Pascal blocks. A major difference between the two is that sentences do not represent any form of program-control: they are mainly used for readability and disambiguation, see Section 4.2. In Figure 2.1 the structure of the procedure division is presented (adapted from the standard [page IV-36]). The format used in Figure 2.1 is the standard's way of writing syntax definitions. It is explained in section 2.5.1.

```
PROCEDURE DIVISION [ USING { data-name-1 } ...  ].
{ section-name SECTION.
 [ paragraph-name.
   [ sentence ] ...  ]  ...  } ...
```

Figure 2.1: COBOL structuring concepts

Control flow In COBOL there are no separate repeat, while, and for statements. Only one such statement exists: the `PERFORM` statement. The `PERFORM` statement can, in a different form, be used for procedure calls as well. See Figure 2.2 for examples of both forms of the `PERFORM` statement. Another form of program control within a source program is the `GO TO` statement. The argument of the `GO TO` statement is a section name or a paragraph name, which are program 'blocks' within the procedure division. Our test programs show that this construct is hardly used in real-life COBOL programs anymore. Standard COBOL provides a conditional construct with the `IF` statement. The `PERFORM` statement uses the conditional construct as well, as do some other COBOL statements (usually through a `WHEN` phrase). Finally, several statements are available for inter-program communication (`CALL`, `CANCEL`, `STOP`, etc.).

COPY, REPLACE, USE Most COBOL statements can occur only at certain positions in the procedure division (anywhere a sentence is required, see Figure 2.1). The `COPY` and `REPLACE` statements (that can appear anywhere in the program) and the `USE` statement (that can appear on only one fixed position in the program) are exceptions to the general statement-format in COBOL. For this reason these statements had to be given special treatment in our specification[3]. These three statements are all compiler directives. The `USE` statement specifies which procedures to use for I-O-error handling. It is defined for all allowed file-types (see [page IV-38]). The `COPY` and `REPLACE` statements are discussed in the next section, and in Section 4.2 and Section 4.3.1.

Imperative statements Many COBOL constructs allow the use of an *imperative statement* as an argument. An imperative statement is a verb[4], and can be a single imperative statement or a sequence of imperative statements [page IV-39]. An imperative statement usually is identical to its 'non-imperative' equivalent except for the optional constructs of that statement. Consider for example the imperative `ADD` statement: it is defined as the non-imperative `ADD` statement without the optional `ON SIZE ERROR` and `NOT ON SIZE ERROR` phrases. Not all statements have an imperative equivalent. See [page IV-39] for a list of all imperative statements.

Finally, assignment is done in COBOL through the `MOVE` statement. This statement is of importance for our year 2000 tool, see Section 5.3.

2.4 Example COBOL program

There are several observations to be made from the example COBOL program given in Figure 2.2:

- Officially, the first six character positions in a source program are reserved for labeling purposes. This feature for line numbering is a leftover from older versions of the standard, and is no longer required. Entering line numbers is still allowed but has no functionality. During testing we therefore used a pre-processor to remove any characters in the first six positions of a COBOL source program.
- In the `CURRENCY SIGN` phrase the `SIGN` and `IS` words are optional. This 'feature' reoccurs throughout the language and is one of the reasons why our syntax specification turned out to be so large.

[3]The same holds for the `IF` statement, which needed some special treatment for disambiguation purposes. See Chapter 4 for a discussion.

[4]The standard uses the term *verb* to indicate both COBOL statements and imperative statements.

```
* This is a COBOL comment entry.
*
      IDENTIFICATION DIVISION.
      PROGRAM-ID. EXAMPLE-PROGRAM.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      SPECIAL-NAMES.
            CURRENCY SIGN IS #.
      INPUT-OUTPUT SECTION.
      FILE-CONTROL.
            SELECT CONTROL-FILE ASSIGN TO DISK
            ORGANIZATION IS SEQUENTIAL.
      DATA DIVISION.
      FILE SECTION.
      FD CONTROL-FILE
            VALUE OF FILE-ID IS "DATES.COB".
      01 DATE-RECORD                      PIC 99/99/99.
      WORKING-STORAGE SECTION.
      01 EMPLOYEE-RECORD.
            05 EMPLOYEE-NAME.
                  10 EMPLOYEE-NAME-CHAR1  PIC X.
                  10 FILLER               PIC X(29).
            05 EMPLOYEE-CITY              PIC X(25).
      01 DANGER                           PIC X(5).
      PROCEDURE DIVISION.
      MAIN-SECTION.
      MAIN-PARAGRAPH.
            PERFORM SOME-PROCEDURE
            OPEN INPUT CONTROL-FILE
            MOVE "FALSE" TO DANGER
            PERFORM UNTIL DANGER = "TRUE"
                  READ DATE-RECORD
                  IF DATE-RECORD = "01/01/00"
                        MOVE "TRUE" TO DANGER
                  END-IF
            END-PERFORM
            COPY SOME-CODE.
      SOME-PROCEDURE.
            DISPLAY "This program makes no sense.".
```

Figure 2.2: Example COBOL program

- `FILE-CONTROL` is a required keyword that directly follows the `INPUT-OUTPUT SECTION` declaration. This appears to be a redundant construct.
- The '`FD`' (file description) is a pre-defined *level number* that, in the user-defined (numeric) form, serves as the index for 'multi-layered' data records. `FD` indicates that the subsequent information describes a file used by the source program.
- An example of multi-layered data records is shown by the definition of the `EMPLOYEE-RECORD`: the level numbers act as labels only; it is common COBOL practice to leave some numbers open (1-5-10) for later additions..

- The `PICTURE` string defines the layout structure of variables, and will be of great importance for the year 2000 tool described in this thesis, see Chapter 5.
- The `PERFORM` statement, as briefly mentioned before, has two occurrences, both present in this example: first as a procedure call, then as a 'while'-construct.
- The `COPY` statement is a low-level statement stating that another piece of code should be inserted at that position in the source program. `COPY` can be considered as a macro expansion mechanism: a piece of auxiliary COBOL code is inserted.
- In our specification lowercase characters are allowed inside user strings only, see Section 4.2.

2.5 The standard

2.5.1 Formalism

The ANSI standard uses a formalism called *general format* to define the syntax of COBOL. The notations and rules for this formalism are described in [pages IV-1..3] and are briefly summarized below.

- Uppercase words are keywords. Underlined uppercase words are required for the corresponding language construct in the source program, uppercase words without underlining are optional for the programmer and serve readability only.
- Lowercase words represent identifiers, literals, `PICTURE` character strings, or a syntactical entry that must be supplied by the user. An example of the last type of lowercase words is the `procedure-division-content` [page V-39] that is mentioned in the definition of a COBOL program, see Figure 2.3. Lowercase words are always required.
- Square brackets ([..][5]) indicate optional choice: one of the options contained may be specified, or all may be omitted.
- Braces ({..}) indicate required choice: one of the options must be present when using the construct in a COBOL source program.
- Repetition of a part of the format is indicated by '...'. The scope of this repetition is the logically first block when scanning from right to left starting at the repetition indicator, where a 'block' is formed by logically matching square brackets or braces.

```
(..)
[ PROCEDURE DIVISION.  procedure-division-content ]
(..)
```

Figure 2.3: Example of one type of lowercase word in formal definition

$$\underline{\texttt{REQUIRED-KEYWORD}}\ \texttt{file-name [IS}\ \underline{\texttt{NOT}}\ \texttt{OPTIONAL]} \left\{ \begin{array}{l} \texttt{REQUIRED-CHOICE-1} \\ \texttt{REQUIRED-CHOICE-2} \end{array} \right\} \ldots$$

Figure 2.4: Example of formal definition notations

2.5.2 Example

Figure 2.4 illustrates the use of the main constructs. Note the following.

- Both `REQUIRED-KEYWORD` and `filename` are required; `filename` can be any allowed user-defined identifier.
- The square bracket construct may be left out completely, or the `NOT` may be used, optionally accompanied by the `IS` and/or the `OPTIONAL` words.
- Either `REQUIRED-CHOICE-1` or `REQUIRED-CHOICE-2` has to be specified. Any number of repetitions of the brace-construct may follow.

2.5.3 Subsets

The standard defines several *subsets* that constitute independent COBOL 'sublanguages'. We have chosen not to limit ourselves to any of these subsets, but to specify the entire syntax. Our motives together with an introduction to the standard subsets are presented in Section 7.2.1.

2.5.4 Obsolete features

The standard defines language elements as obsolete when they are to be deleted from standard COBOL [page XVII-81]. To limit damage for users, every language element deleted from a version of the COBOL standard must have been obsolete in the previous version[6].

The standard lists the following rules with respect to obsolete features.

- Obsolete language elements are neither enhanced, modified, nor maintained.
- The interaction between obsolete language elements and other language elements is undefined unless otherwise specified in Standard COBOL.
- A conforming implementation of Standard COBOL is required to support obsolete language elements (also in subsets supporting the obsolete language elements).

A list of obsolete features is included as Appendix A.

2.5.5 Problems

We have had some problems using the standard. On the one hand it is too big to read from begin to end (or, if you read it all, it is too much to remember), on the other hand it is not very useful as a reference manual.

Although there is a large section (V in [9]) devoted to the general format (containing formal definitions of the syntax), of which they say [page V-1]:

[5] Because the standard uses '...' to indicate repetition, we use '..' to indicate an open entry.

[6] This implies that when an 'unexpected' modification of an existing language element of COBOL is required shortly after the release of a standard version, this modification can not take place until two releases down the line. In ANSI terms, that would mean a delay of nearly thirty years.

> "It is intended to display complete and syntactically correct formats."

many of these constructs are incomplete. When reading the rest of the standard it becomes clear that the general format allows syntactical constructs that are not allowed by standard COBOL as a whole. They announce this limitation of their general format in section IV: *Overall language considerations* (hardly the introduction of the standard), along with a description of the functionality of the *syntax rules* section and *general rules* section [page IV-3] that are used throughout the standard as supplements to the general format:

> "Syntax rules may also either impose restrictions on individual words or elements or relax restrictions implied by words or elements."

That is, in the syntax rules section syntactical properties may be added to the definition in the general format. The general rules section is supposed to describe semantics.

A striking example of the effects of this policy is the fact that the order of optional constructs ('[..]') often is defined in words in the syntax rules instead of formally in the general format. In many of the sequences of optional constructs listed in the standard (such as the one in Figure 2.5) the items can occur in arbitrary order. This is (often) not the case when the sequence is part of a COBOL statement. We believe it should not have been too difficult to come up with an extra symbol to indicate 'arbitrary order'. An example of a case where arbitrary order is allowed for the optional constructs is the communication description entry shown in Figure 2.5.

```
CD cd-name-1  FOR  OUTPUT
   [ DESTINATION COUNT IS data-name-1 ]
   [ TEXT  LENGTH  IS  data-name-3 ]
   [ STATUS  KEY  IS  data-name-3 ]
   [ DESTINATION  TABLE  OCCURS  integer-1  TIMES
      [  INDEXED  BY  {  index-name-1  }  ...   ]   ]
   [  ERROR  KEY  IS  data-name-4  ]
   [  SYMBOLIC  DESTINATION  IS  data-name-5  ].
```

Figure 2.5: Communication Description Entry

Summarizing, strictly using the section of formal definitions of the standard will not be enough to be certain of the exact properties the elements described in that formal definition are supposed to have.

A second major problem with the standard (as a reference manual) comes from the fact that the syntax rules section in the standard frequently and clearly describes (static) semantics and the general rules section sometimes describes syntax. An illustration of the inconsistent way the standard distinguishes between syntax and semantics is given below through two quotes [pages VI-110,111]. For some reason the first quote is listed in the syntax rules section while the second quote occurs in the general rules section.

> "If an index-name is specified in the `VARYING` or `AFTER` phrase, then: (..) The literal in the associated `BY` phrase must be a nonzero integer."

> "If an index-name is specified in the `VARYING` or `AFTER` phrase, and an identifier is specified in the associated `FROM` phrase, the data item referenced by the identifier must have a positive value."

Some more general format-related problems are listed below.

- The USE statement is listed in an inconsistent way, see Section 4.2.
- Entities like condition-name are defined [page IV-7], but not with reference to the general format. Conditions exist, but the exact meaning of a condition name can not be derived from the general format (where conditions are defined). From the glossary we learn that a condition name is a user-defined word that assigns a name to a subset of values that a conditional variable may assume.
- Nowhere in the formalism is it mentioned that an imperative statement can be a sequence of statements, without optional constructs. Consider the READ statement in Figure 2.6 (from [page V-28]): the occurrences of imperative--statement-3 and the imperative-statement-4 can be sequences of imperative statements.
- Literals are defined as either numeric or non-numeric. But, for example, when the standard uses the word literal in the SELECT statement (see Figure 2.7) on [page V-5] it is mentioned on [page VII-7] that this literal may only be numeric. This is also an illustration of the inevitablility of our specification being at least in part 'bottom-up', see Chapter 6 for a discussion.

```
READ file-name-1 RECORD [ INTO identifier-1 ]
    [ KEY IS data-name-1 ]
    [ INVALID KEY imperative-statement-3 ]
    [ NOT INVALID KEY imperative-statement-4 ]
    [ END-READ ]
```

Figure 2.6: Example of imperative statements

```
ASSIGN TO { implementor-name-1 }
          { literal-1          }
```

Figure 2.7: Part of SELECT statement

The standard as a whole (hopefully) is unambiguous and describes COBOL as the authors intended, but it is clear that these problems reduce its usefulness as a reference manual. Several other examples will be presented throughout this paper.

Chapter 3

ASF+SDF

In this chapter the ASF+SDF formalism is introduced. See [17] for a complete description of the SDF formalism, [14] for a complete description of the ASF formalism, and [20] for a complete description of the ASF+SDF Meta-enviroment.

3.1 Algebraic specifications

An algebraic specification consists of a signature, a set of variables, and a set of equations. The signature describes sorts and functions over these sorts. In the equations variables and functions are used to construct terms and to define (possibly conditional) equations between those terms. Priorities can be attributed to functions in order to settle ambiguities. For every lexical sort a *lexical constructor function* is created to provide access to the individual characters of that sort.

Suppose, for example, that we want to specify the classic Boolean data-structure. The signature in this case would define one sort `BOOL`; two constants of sort `BOOL`, `true` $\rightarrow$ `BOOL` and `false` $\rightarrow$ `BOOL`; one unary function `not: BOOL` $\rightarrow$ `BOOL`; and finally two binary functions, `and: BOOL # BOOL` $\rightarrow$ `BOOL` and `or: BOOL # BOOL` $\rightarrow$ `BOOL`. Suppose the variable *Bool* exists, then we can syntactically derive the following terms: `true`, `false`, *Bool*, `not(true)`, `not(`*Bool*`)`, `and(true, false)`, `or(not(false), and(true, false))`, and so on. These terms can be evaluated if the functions have semantics: the semantics of `not`, `and`, and `or` should be defined. The semantics of these functions can be defined by means of equations. An example equation that partially defines the semantics for the `and` function is `and(true,` *Bool*`) =` *Bool*[1].

This algebraic specification can be transformed into a term rewriting system by interpreting the equations as rewrite rules from left to right. As a result, the semantics specification of the booleans can be used to compute the value of any (syntactically correct) term over the signature and to produce its normal form, either `true` or `false`, as answer.

3.2 The ASF+SDF formalism

The ASF+SDF formalism started out as ASF (*Algebraic Specification Formalism*) alone. SDF (*Syntax Definition Formalism*) was added shortly after to provide the specifier with more syntactical freedom: lexical, context-free, and abstract syntax are supported and incremental parsers can be generated from the syntax.

[1]This equation will end up in the 'ASF-part' of our specification of the booleans. The specification of the booleans will be used for the year 2000 tool, see Appendix C.

Lexical syntax is the definition of a low level structure of sentences (sequences of characters) in terms of lexical tokens. A lexical token is a pair consisting of a *sort name* and a *lexeme*. The former is used to distinguish classes of lexical tokens, the latter is the actual text of the token. For example

```
[A-ZO-9] → NAME-ELT
```

defines `NAME-ELT` to be either a character or a digit.

List functions and list variables are an essential part of the ASF+SDF formalism. The list function uses two constructs, `*` and `+`, indicating zero or more repetition and one or more repetition, respectively.

Suppose we want to define the set. In the context-free syntax we would define the following list function:

```
"{" {ELEMENT ","}* "}" → SET
```

The comma is used as a separator between `ELEMENTS`. A set therefore is defined here as consisting of `ELEMENTS`, separated by commas and delimited by { and }. The non-terminal, the separator, and the delimiters can be chosen freely for list functions. The repetition constructs `*` and `+` can be used in a different context as well: as 'attributes' to a sort, as in

```
STATEMENT+ "." → SENTENCE
```

indicating that a sentence constists of one or more statements (and is terminated by a period).

An ASF+SDF specification is organized in *modules*. Each module defines syntax rules as well as semantic rules. The semantic (ASF) part uses the signature in the syntax (SDF) part[2]. A module can import other modules in the specification. All visible entities of the imported module become available in the importing module. It is possible to hide entities in an imported module by using the *hiddens* keyword. Its inverse is the *exports* keyword. Below we present the keywords allowed in an ASF+SDF module:

- **imports**
  The names of imported modules are listed in this section.

- **exports**
  The items exported by the module are listed in this section. These items can be used by modules that import the module.

- **hiddens**
  The items that are hidden in the module are listed here. These items can not be used by importing modules.

- **sorts**
  Is followed by the sort names used in the lexical and context-free syntax sections.

- **lexical syntax**
  Followed by definitions of layout conventions and lexical tokens.

- **context-free syntax**
  Followed by context-free function declarations.

[2]Sometimes parts of the syntax are, for efficiency reasons, specified in the ASF part, see Chapter 4.

- **priorities**
  Priorities between context-free functions are listed here.
- **variables**
  Followed by declarations of variables used in the equations.
- **equations**
  Followed by the (conditional) equations of the module.

Sort declarations, lexical functions, context-free syntax, and variables must either be part of an `export` section or be declared as `hiddens`.

3.3 Example

In Figure 3.1 we present the Booleans example as it is used in this thesis in the year 2000 tool.

```
exports
   sorts BOOL

   context-free syntax
      true                 → BOOL
      false                → BOOL
      BOOL or BOOL         → BOOL {left}
      BOOL and BOOL        → BOOL {left}
      not "(" BOOL ")"     → BOOL
      "(" BOOL ")"         → BOOL {bracket}

   variables
      Bool[0-9']*          → BOOL

   priorities
      or < and


equations
   [1] true or Bool = true
   [2] false or Bool = Bool

   [3] true and Bool = Bool
   [4] false and Bool = false

   [5] not(false) = true
   [6] not(true) = false
```

Figure 3.1: Booleans in ASF+SDF

Some observations:

- The version in the Appendix C.2 includes some lexical syntax defining layout that is to be ignored by the parser.
- The term attributes {`left`} and {`right`} are used for disambiguation; it defines a function to be either left or right associative. For instance, the attribute

{left} in BOOL and BOOL → BOOL indicates that the term true and false and true should be interpreted as (true and false) and true rather than as true and (false and true). In this case it is just for disambiguation: the normal forms of any term using the and function would be identical for either left or right. This is not the case however in the example 2^{3^4}, where clearly $2^{(3^4)}$ has a meaning different from $(2^3)^4$.

- The attribute {bracket} is added for grouping and disambiguation only, context-free rules with this attribute are not included in the abstract syntax tree.

- Declarations of variables are treated as declarations of lexical syntax. The [0-9']* notation indicates that all variables Bool immediately followed by any number of digits and/or quotes are now declared.

- The 'and' binds stronger than the 'or', since its priority is higher. This means that the term true and false or true should be parsed as (true and false) or true, rather than as true and (false or true), which reduces to a different normal form.

- The numbers in front of the equations are labels and can be of arbitrary value. They are useful when debugging the specification.

3.4 The ASF+SDF Meta-environment

The ASF+SDF Meta-environment [20] is a programming environment for writing algebraic specifications in ASF+SDF by means of syntax directed editors. Given a syntax and semantics specification of a (programming) language, an environment can be generated using the ASF+SDF Meta-environment consisting of a syntax directed editor, a parser, and a term rewrite system. The system is able to perform static semantic checks on specifications and supports the testing and debugging of specifications.

Chapter 4

The syntax specification

In this chapter our main specification effort is described: the specification of the complete COBOL III syntax in SDF. First, we present the method we used for translating constructs from the standard's general format to the SDF notation, together with an example. Next, we present a discussion of the problems we encountered and subsequent solutions. The chapter ends with a discussion of non-standard COBOL versions and their effects on the specification process.

4.1 Translation method

For translating the standard's general format into equivalent SDF code we used a few general rules:

- Optional keywords (not underlined) are represented in SDF by a sort and two context-free syntax rules. For example, translating `CONTAINS` results in the sort `OPT-CONTAINS` with the rules shown in Figure 4.1.

	→ OPT-CONTAINS
"CONTAINS"	→ OPT-CONTAINS

Figure 4.1: SDF rules for `OPT-CONTAINS` sort

- 'Blocks' formed in the standard's general format by [..] or {..} constructs are, when they include a choice between syntactical alternatives, represented by a separate sortname. Examples are the `RECORD-DESCR` sort and the `DATA-DESCR` sort in Figure 4.3, corresponding to the block following the underlined `RECORD` and the underlined `DATA` in Figure 4.2, respectively.

- Because the standard does not use nonterminal names in its descriptions in the general format, we had to come up with sortnames ourselves. Considering the large number of different types of structures used in the standard it was hard to find a uniform naming-scheme. Some general principles we followed are:

  - `OPT` and `OPTION` are used frequently as extensions of sortnames to indicate that it describes an optional block in the general format. As much as possible our sortnames are identical to the names provided in the general format: the optional block belonging to the `CLOSE` statement will be named `CLOSE-OPTION` etc.

- `HEAD`, `BODY`, and `TAIL` are used often for first, middle, and last constructs belonging to a COBOL language element.
- Sometimes several COBOL elements are combined into one sort, as in `ID-OR-LIT`, indicating that an identifier or a literal can occur at that position.

As a small translation example we have chosen the definition of the sort-merge file description entry [page V-12]), see Figure 4.2.

```
SD file-name-1

[        { CONTAINS integer-1 CHARACTERS                                                }  ]
[ RECORD { IS VARYING IN SIZE [ [ FROM integer-2 ] [ TO integer-3 ] CHARACTERS ]        }  ]
[        {     [ DEPENDING ON data-name-1 ]                                             }  ]
[        { CONTAINS integer-4 TO integer-5 CHARACTERS                                   }  ]

[DATA { RECORD IS   } data-name-2 ...] .
[     { RECORDS ARE }                ]
```

Figure 4.2: COBOL standard definition of the sort-merge file

Observe that the numbers in integer-1, integer-2, ... do not indicate that these numbers have to be unique but are included for reference in the explanations throughout the standard.

The translation of this construct results in the SDF code (taken from our ASF+SDF `File-description-entry` module) in Figure 4.3. Here, `ID-NAME` is defined lexically as a string of allowed characters and `INTEGER` is defined in the usual way. The `OPT-` constructs are defined as either empty or the associated word.

Note that since the sort `ID-NAME` does not have an 'empty rule' (it can not be left out) we have effectively implemented a way of counting the number of arguments here: the `RECORD IS` phrase can only be followed by exactly one `ID-NAME`, and the `RECORDS ARE` phrase must be followed by at least two instances of `ID-NAME`. This choice may seem logical, but the standard does not provide any information on the use of this aspect of these constructs, leaving us with a formal definition that clearly states that any number of `ID-NAME` constructs can occur with both `RECORD IS` and `RECORDS ARE`.

4.2 Problems and solutions

In this section we will discuss some of the problems we encountered while specifying the COBOL syntax. Most of the time the choices we made in order to solve a problem implied a deviation from the ANSI standard. The subsections are, where possible, listed from 'abstract' to 'concrete'. Several smaller issues are treated as comments in the syntax specification itself, see Appendix B.2.

4.2.1 Functional modules

Although the standard defines several conceptual functional modules in which statements (or rather language elements) can occur, we have chosen to rearrange them a little. Below is a short list of differences between our division and the one in the standard. The differences are purely 'organizational'.

- Instead of three different I-O modules, one for each allowed COBOL file type (see the paragraph on file types in this section), we have only one (`Io-verbs`).

"SD" ID-NAME RECORD DATA "."	→ SORT-MERGE-ENTRY
	→ RECORD
"RECORD" RECORD-DESCR	→ RECORD
OPT-CONTAINS INTEGER OPT-CHARACTERS	→ RECORD-DESCR
OPT-IS "VARYING" OPT-IN OPT-SIZE FROM-TO DEPENDING	→ RECORD-DESCR
OPT-CONTAINS INTEGER "TO" INTEGER OPT-CHARACTERS	→ RECORD-DESCR
FROM-INT TO OPT-CHARACTERS	→ FROM-TO
	→ FROM-INT
"FROM" INTEGER	→ FROM-INT
	→ TO
"TO" INTEGER	→ TO
	→ DEPENDING
"DEPENDING" OPT-ON ID-NAME	→ DEPENDING
	→ DATA
"DATA" DATA-DESCR	→ DATA
"RECORD" OPT-IS ID-NAME	→ DATA-DESCR
"RECORDS" OPT-ARE ID-NAME ID-NAME+	→ DATA-DESCR

Figure 4.3: Resulting SDF specification

- The standard defines several conceptual modules like *source text manipulation module*, that contain the `COPY` and `REPLACE` statements. We give those statements special treatment (instead of devoting an entire module to them). See also Section 4.3.1.
- We made a modular division between arithmetic verbs, program control verbs, common verbs, and I-O verbs. The standard uses a different classification of verbs. The standard is subdivided into several functional modules (sequential I-O module, source text manipulation module, etc.); for each module a subset of the COBOL statements is used and defined. (For example, the sequential I-O module will use all file-I-O statements, and so on.)

Our conceptual division can be found in Figure B.1; the standard's choice of modules is listed in [page I-10..39].

4.2.2 Syntax versus semantics

We have tried to syntactically solve syntax problems as much as possible. An example to the contrary is the (syntactical) restriction that most user-defined COBOL words can at most be of length 30. Suppose one wants to specify this syntactically in SDF: thirty alternatives for some (lexical) sort would then be required, one for each allowed length. For this reason, and because it is a relatively unimportant problem, we have not implemented this syntactically. The restriction can be specified much more efficiently in semantics (which we have not specified): it is possible to use the lexical constructor function to 'count' the number of characters in a string.

Another example of a syntax/semantics conflict is the `PICTURE`-syntax rule 4 as listed in the standard (that states that `PICTURE` strings can be of length 30 at most as well) [page VI-29], which, because five types of `PICTURE` strings exist, would have led to thirty times five extra lines of ASF+SDF code. A situation where we *did* implement the 'counting' is the case of fixed number of arguments of a statement. An example of this is the communication description entry, where either six or eleven data-names are allowed as arguments, depending on the format (three formats for communication description entries exist).

During the early stages of the specification process we specified the `DATA RECORD` clause as shown in Figure 4.2 and Figure 4.3, unaware of the fact that this clause was obsolete. Later we discovered that our (intuitively correct) conclusion, that the `RECORD IS` phrase should be followed by exactly one data-name and the `RECORDS ARE` should be followed by at least two data-names was wrong. The standard has no restrictions on the number of data-names in either case, except that in both cases at least one data-name is required. This is of course exactly what the formal definition of the `DATA RECORD` clause implies, but it is not what one would expect, especially considering the other exceptions and additions to the formalism occurring throughout the standard (changing the apparent meaning of the general format's description). This might be one of the reasons why the clause is obsolete. Because the construct is obsolete we decided to leave the specification as it is.

Often, the standard is not very clear about what it intends to be syntax or semantics. The `PERFORM` statement, for example, has four occurrences in the formal descriptions; in each of these occurrences it has an optional beginning and an optional end, see Figure 4.4. It is defined in words in the standard [page VI-110] that these two constructs are to be used in a mutually exclusive way. Is this a syntax or a semantics constraint? The standard mentions this in the syntax rules section and that seems to be the most logical, but they do not indicate it in their formal description [pages V-26,27]. Here too, we decided to postpone this to the semantic phase, and not to specify it syntactically.

```
PERFORM [ procedure-name-1 [ { THROUGH } procedure-name-2 ] ]
                             { THRU    }

    (..)

    [ imperative-statement-1 END-PERFORM ]
```

Figure 4.4: Skeleton of the `PERFORM` statement

Another example is the `GREATER` construct of (among others) the `START` statement, partially shown in Figure 4.5.

```
IS GREATER THAN
```

Figure 4.5: `GREATER` construct

Is it allowed to leave out both optional parts (`IS` and `THAN`) as one likes, or only either both or none? If the latter guess is right, then should this be specified syntactically or semantically? We could not find any justification for either guess. We believe that the former guess is correct and therefore decided to allow any combination of the optional parts in our specification.

4.2.3 Order of optional constructs

As mentioned in Section 2.5.5 it is sometimes allowed to put optional language constructs in arbitrary order. In ASF+SDF this creates a problem. In the example of Figure 2.5 [pages V-16, XIV-4] the order of all optional constructs is arbitrary.

The only way to exactly specify this in ASF+SDF is to create an 'optional' sort for each optional construct, and make a context-free rule for every allowed combination (which in this case would imply 6! = 720 context-free rules). The 'second best' specification would be to let all of the new sorts unite in one new optional sort, say E, and then make one context-free rule, for example

```
E  E  E  E  E  E  →    COMM-DESCR-ENTRY
```

The disadvantage of this specification is that all of the optional constructs can occur more than once. The shortest possible way, which would allow the most non-standard side-effects, would be simply E*. We have chosen to specify this as if there was no freedom in the order of the optional constructs: the order in which they are listed in the standard is the only allowed order in COBOL programs. We have encountered some problems with this while testing real-life COBOL programs, but not too many.

Another approach would be to solve this problem at the semantic level. It is relatively easy to define equations which checks whether each optional construct occurs exactly once.

4.2.4 Obsolete features

Obsolete features are marked in the standard; they will not reappear in the new object oriented version of COBOL. The obsolete features have not been updated during the preparation of the 1985 standard and are sometimes undefined in interaction with other statements. In spite of this, the standard states that an implementation of (a subset of) the standard *must* include the obsolete features and that warnings must be given when a user tries to compile a program with obsolete language elements. We have chosen in general not to specify obsolete features (see Appendix A for a list).

4.2.5 File types

Three file types are allowed: *sequential* (where the only access possibilities are a successor and predecessor field in each record, and the order of the records in the file depends on the order in which they are written), *relative* (where each logical record can be accessed by an assigned number), and *indexed* (with facilities for indexed access). These file types have an ambiguous definition in the standard. Figures 4.6 and 4.7 shows the definitions of the `REWRITE` statement for the sequential and the indexed file [page V-28].

```
REWRITE record-name-1 [ FROM identifier-1 ]
```

Figure 4.6: The `REWRITE` statement for sequential files

```
REWRITE record-name-1 [ FROM identifier-1 ]
    [ INVALID KEY imperative-statement-1 ]
    [ NOT INVALID KEY imperative-statement-2 ]
    [ END-REWRITE ]
```

Figure 4.7: The `REWRITE` statement for indexed files

Observe that if in some source program the optional constructs are left out in both definitions, it is impossible to determine which one to take.

The standard seems to indicate that only one of the three possible types is allowed at one time in a COBOL program. This of course causes problems when specifying the file types formally: the only solution was to choose one file type and to forget the other two. During testing we chose to specify the sequential access file type, because in COBOL source programs this type of file is almost always used. In the semantics definition of the COBOL language it might be possible to specify more than one file type without ambiguities, for example by checking the statements that operate on a certain file. For now, the final version of our specification includes specifications of I-O statements for all three allowed file types, and is therefore ambiguous. If the specification is to be used for COBOL programs with file I-O, two of the three allowed file types should be excluded from the specification or, as mentioned, some semantic disambiguation should take place.

4.2.6 Scope of statements/ambiguities

Statements in the procedure division at first glance appear prone to ambiguities in the cases where an imperative statement is allowed as the argument or substructure of a statement. Consider for example the definition of the `RECEIVE` statement [page V-28] in Figure 4.8.

```
RECEIVE cd-name-1 { MESSAGE } INTO identifier-1
                  { SEGMENT }
   [ NO DATA imperative-statement-1 ]
   [ WITH DATA imperative-statement-2 ]
   [ END-RECEIVE ]
```

Figure 4.8: COBOL standard definition of the `RECEIVE` statement

Because an imperative statement can consist of multiple statements the risk of ambiguity arises when:

- The `RECEIVE` statement is part of a sequence of statements in a sentence.
- It is not the last statement in that sentence.
- It is used with the optional `imperative-statement-2` construct.
- The optional `END-RECEIVE` construct is not used.

It then seems impossible to determine whether the next statement in the program belongs to the `RECEIVE` statement or whether it is the next statement in the sentence[1]. Fortunately, it turns out there is no risk for ambiguity when fully implementing the standard's rules: it then becomes clear that an imperative statement has to be terminated by:

- A period terminating all statements in the imperative statement.
- Or, a phrase belonging to the statement containing that imperative statement (e.g. `WHEN`, `NOT AT END`, `END-RECEIVE`, etc.).

These two 'characteristics' of imperative statements are mentioned somewhat out-of-context in the standard[2] [page IV-39].

[1]Note that when a COBOL program uses the `imperative-statement-1` construct disambiguation will automatically take place when the program uses the `imperative-statement-2` construct as well (see further on). However, the problem then moves to the `imperative-statement-2` construct.

[2]Again, the standard's usefulness as a reference manual decreases.

If the imperative statement is ended by a period it can not be followed by another construct of the statement in which the imperative statement occurred. Consider for example Figure 4.8: if the imperative-statement-1 is followed by a period in some source program it is not allowed to follow it up with the `WITH DATA` phrase. A new statement has to follow the period. This is our interpretation of the somewhat 'ambiguous' remark in the standard [page IV-39]:

> "(..) sequence of consecutive imperative statements that must be ended by a period or by any phrase associated with a statement containing that 'imperative-statement'."

We read this 'or' as an exclusive one (either a period or a related phrase but not both) but for efficiency reasons have specified it as an inclusive one. Although we allow too much in our current specification by defining an imperative statement as a sequence of statements followed by a period and/or any other construct of the containing statement (the `RECEIVE` statement in our previous example), it does not lead to ambiguities and every syntactically correct program is still parsed correctly.

Initially our specification did not allow a period as the terminator of an imperative statement. This resulted in the following interesting situation. If an imperative statement is not ended by another construct of the containing statement, then this imperative statement has to be the last construct of that statement. This imperative statement then has to be directly followed by a period. In our old specification this period accidentally indicated the end of the imperative statement but also, and more importantly, the end of the sentence. Since a paragraph contains a potentially endless and unstructured[3] sequence of sentences the next statement that may have been intended as part of the previous sentence is now considered the first statement of a new sentence, and no parse errors will occur. Of course, the program structure is mixed up because of this and pretty printing will produce a mess. (As mentioned above, our specification allows periods now, so we do not have this problem.)

Another problem arises from the fact that the standard states that the statements inside an imperative statement may each be terminated by a separator [page IV-39]. The period being a separator too, this again appears to be a guarantee for ambiguities. It is mentioned elsewhere in the standard however [page IV-4] that the separator period may only be used as the indicator for the end of a sentence (and although an imperative statement is a sequence of statements as well, it is not a sentence) or as shown in the formats (see for example Figure 4.2)[4].

4.2.7 Lower/uppercase

Our specification can only deal with uppercase code. To ensure 'uppercase conformance' we have written a simple C-program that converts every source-program into its uppercase equivalent. Because almost every COBOL feature is case-insensitive [pages IV-5, 6], this can be done safely. The only case-sensitive example we could find are COBOL literals. Our choice not to allow mixed-case in our specification stems from the same type of problem as described in the paragraph on syntax versus semantics: allowing mixed-case would result in an explosion of the number of lines of ASF+SDF. Here, we have one of the most dramatic examples of that problem: every COBOL keyword should be specified with each of its constituent characters in upper- and lowercase, and this for every possible combination. Of course this should be done for every COBOL construct, not just for keywords.. Fortunately there is a way to considerably reduce the number of SDF-rules, as the following example shows. Consider the `MOVE` statement (assignment): instead of writing down every combination of upper- and lowercase characters it is sufficient to simply write

[3] Since the `PERFORM` statement also operates on imperative statements, not on statements.

[4] This is another clear example of the failure of the standard as a reference manual.

```
[Mm][Oo][Vv][Ee] → MOVE-KEYWORD
```

Now, instead of using the rule

```
"MOVE" → STATEMENT
```

the rule

```
MOVE-KEYWORD → STATEMENT
```

can be used, incorporating all possible permutations of upper- and lowercase characters inside `"MOVE"`. Although this construct strongly reduces the number of SDF-rules required, it is still a lot of work.

4.2.8 COBOL words

There are many different types of COBOL *words* [page IV-6]. A COBOL word can be a user-defined word (such as the identifier), a system-name (for example computer-name), or a reserved word (for example `ZERO`). Note that the reserved words do not include the COBOL keywords. Many types of the user-defined words exist (e.g. condition-name, record-name, data-name, file-name, etc.). We have chosen to identify these and have called them `ID-NAME` in the specification. This `ID-NAME` is also the basis for the identifier. The identifier has several extra options [page IV-23] in comparison to the COBOL word. *Not* a COBOL word, for example, is the *literal*, which has been defined separately.

4.2.9 The `CALL` statement

The `CALL` statement as listed in the formal definition of the ANSI/ISO standard is ambiguous, see Figure 4.9 and Figure 4.10 [pages V-20, 21][5].

$$\underline{\texttt{CALL}} \left\{ \begin{array}{l} \texttt{identifier-1} \\ \texttt{literal-1} \end{array} \right\} \left[\underline{\texttt{USING}} \left\{ \begin{array}{l} [\ \texttt{BY}\ \underline{\texttt{REFERENCE}}\ ]\ \{\texttt{identifier-2}\}... \\ \texttt{BY}\ \underline{\texttt{CONTENT}}\ \{\texttt{identifier-2}\}\ ... \end{array} \right\} ... \right]$$
$$[\ \texttt{ON}\ \underline{\texttt{OVERFLOW}}\ \texttt{imperative-statement-1}\ ]$$
$$[\ \underline{\texttt{END-CALL}}\ ]$$

Figure 4.9: The `CALL` statement: format 1

$$\underline{\texttt{CALL}} \left\{ \begin{array}{l} \texttt{identifier-1} \\ \texttt{literal-1} \end{array} \right\} \left[\underline{\texttt{USING}} \left\{ \begin{array}{l} [\ \texttt{BY}\ \underline{\texttt{REFERENCE}}\ ]\ \{\texttt{identifier-2}\}... \\ \texttt{BY}\ \underline{\texttt{CONTENT}}\ \{\texttt{identifier-2}\}\ ... \end{array} \right\} ... \right]$$
$$[\ \texttt{ON}\ \underline{\texttt{EXCEPTION}}\ \texttt{imperative-statement-1}\ ]$$
$$[\ \underline{\texttt{NOT}}\ \texttt{ON}\ \underline{\texttt{EXCEPTION}}\ \texttt{imperative-statement-2}\ ]$$
$$[\ \underline{\texttt{END-CALL}}\ ]$$

Figure 4.10: The `CALL` statement: format 2

That is, if in some COBOL program the optional `OVERFLOW` and `EXCEPTION` parts are omitted in a `CALL` statement, it is impossible to determine which format applies. The standard does not have a problem with this type of ambiguity because it does not have to provide an implementation. This may sound trivial but in fact they might as well have provided a definition of the `CALL` statement with all three mentioned optional contructs in one format, with an explanation of how to use it somewhere down the line in the standard (as they do with many other syntax/semantics issues). The decision to split it up here appears to be very much ad-hoc.

[5] See appendix B.2.14 for our SDF-solution of this ambiguity problem.

4.2.10 The `COPY` statement

The `COPY` statement is a direct 'input' of source code from another file into the current file. Since no recursion is allowed inside a `COPY` statement, we were able to write a pre-processor that includes the required files at the right places in the source program (see also Section 4.3.1). Such a macro mechanism is impossible to specify in ASF+SDF unless the source file is available as well.

4.2.11 The `IF` statement

The `IF` statement [page VI-90] (see Figure 4.11 [page V-24]) needed some special treatment. The reason for this is that in order to disambiguate the statement it is sometimes necessary to ensure that it is directly followed by the period ending the sentence in which it occurs. This is the case when:

- the `ELSE/END-IF` part of the `IF` statement is not ended with `END-IF` after the statement-part is chosen; this is allowed since the `END-IF` is optional there. In that case the scope has to be terminated by a period, or by an `ELSE` phrase that belongs to a higher-level `IF` statement.
- the `ELSE NEXT SENTENCE` phrase is omitted; this is allowed when it is immediately followed by the period ending the sentence [page VI-90].

```
IF condition-1 THEN { { statement-1 } ...  }
                    { NEXT SENTENCE         }

   { ELSE { statement-2 } ...   [ END-IF ] }
   { ELSE NEXT SENTENCE                    }
   { END-IF                                }
```

Figure 4.11: The `IF` statement

4.2.12 The `USE` statement

The definition of the procedure division has some notational ambiguities in the `DECLARATIVES` section [page V-19] shown in Figure 4.12: the `USE statement` remark should, according to the standard's explanation of the formalism [pages IV-1..3], define an optional word '`USE`' followed by any of the statements.

```
[ DECLARATIVES.
{ section-name SECTION [ segment-number ].
   USE statement.
 [ paragraph-name.
   [ sentence ] ...  ]  ...  } ...
 END DECLARATIVES.  ]
```

Figure 4.12: `DECLARATIVES` section of procedure division

From test programs and from the standard [pages IV-35, VII-50] it is clear that this is not what is intended here[6]. We believe the standard means to say that only the `USE`-statement is allowed at this point. The fact that they do not use an

[6] Also, a `USE` followed by a random statement would not make much sense.

appended number or letter (for reference in the standard's comments) here [page IV-2] like they do elsewhere in the formal definition with imperative statements (etc.) does not help a lot because they use these extensions inconsistently in the definition anyway: sometimes a letter or number is added to an item when there is only one such item [e.g. page V-20 etc.], sometimes it is not [e.g. page V-19]; sometimes '`paragraph-name`' gets an extension [page V-37], sometimes it does not [page V-19]. In the case of the imperative-statement they *do* use clear notation [e.g. page V-21]. Inconsistencies like this again reduce the usefulness of the standard as a reference manual.

4.2.13 77-level-description-entry

The 77-level-description-entry is used to group items that bear no hierarchical relationship to one another [page VI-18]. To distinguish such 'data-groups' from ordinary records the number 77 is inserted at the position of the level-number. We have chosen to syntactically identify this 77-level-description-entry, listed in [9] under the working storage section and the linkage section, with the 'ordinary' record-description-entry [section V]. See the `Data-division` module.

4.2.14 COBOL peculiarities

Most problems and solutions in this section clearly originated from shortcomings of COBOL, or from limitations of ASF+SDF. Many features of COBOL were encountered however that did not directly cause a problem but appeared very strange indeed. We present an example of this, consider a small part of the report description entry in Figure 4.13 [page V-17][7]. The phrase `LIMIT` or `LIMITS` can be left out while the `IS` or `ARE` can remain, leaving a sentence with no intuitive information for a programmer or software engineer to read.

```
      [ LIMIT IS   ]
PAGE  [ LIMITS ARE ]  integer-1
```

Figure 4.13: A strange COBOL feature

4.3 Non-standard COBOL

During testing of the specification several different COBOL-versions/dialects were encountered. In this section we list two of them, along with their deviations from the standard (as far as we could obtain from our testprograms).

First, here are some remarks on the pre-processing of COBOL programs.

4.3.1 Pre-processing

In order to get our grammar to recognize certain (sometimes non-standard) COBOL programs some pre-processing was necessary. We have written simple C-program that takes care of several pre-processing requirements. In all of these cases the modification to the source code has been done in a 'safe' way (i.e. the functionality of the COBOL programs was not changed):

[7] [pages XIII-17 – 19] (where the syntax- and general rules sections for the `PAGE` construct are listed) provide a good example of the standard's extensive use of exceptions.

- In COBOL uppercase and lowercase notations are equivalent except for the trivially safe quote-delimited strings. All other language elements are case-insensitive, making it safe for the pre-processor to convert a source program to its, in our case uppercase, equivalent.
- The `COPY` statement directly imports source code from a different file. The pre-processor copies the contents of this second file into the source program at the position of the `COPY` statement while removing the statement itself. The `COPY` statement can occur everywhere in the program where some COBOL construct is required (that is, it is not allowed for example to use the `COPY` statement halfway in a keyword etc.). Without pre-processing the source program in this way an analysis of the program would not cover the entire code, which reduces its usefulness.
- The `EXEC`..`END-EXEC` construct is a COBOL extension, probably introduced by Telon (a tool for generating COBOL code), that is not part of the COBOL language and is therefore eliminated by our pre-processor. The pre-processor places comment-indicators '*' before such blocks of mostly SQL (-like) statements. These blocks *are* important for the year 2000 problem, but can not be handled by our ASF+SDF specification yet (this could be dealt with in a future project).

4.3.2 IBM VS COBOL for OS/VS

This COBOL dialect explained a number of errors that occurred while testing the specification. The dialect was published in 1986, after the COBOL III publication, but is nevertheless based on the 1974 ANSI standard. Because many of our test programs were based on this dialect and because these problems did not occur due to updates in the 1985 standard, we have included the dialect here.

Non-standard extensions are listed throughout the IBM document [18]. Here we present the IBM extensions our test programs provided us with.

- On page 107 [18] 'Computational Options' such as `COMP-4` are, except for the one form `COMP`, not standard.
- The single quote instead of the standard double quote as string delimiter is an IBM extension, see page 457 [18]. Our pre-processor changes this notation to the standard double quotes. This can be done without risk because both symbols function as string-delimiters only in COBOL.
- The `EJECT` and `SKIP1, 2, 3` statements are extensions intended for the formatting of the source text. The pre-processor inserts comment-indicators before occurrences of these statements. We assume that these statements can occur on an empty line only, so no other statements or constructs will be removed by our pre-processor.
- Apparently the first section name and paragraph name of the procedure division can be left out. This is not standard.

4.3.3 ANSI-COBOL 6.50

This DOS-oriented COBOL implementation [4] is called 'ANSI-standard', but it does not conform to the standard. It is based on the 1974 standard [8], of which it implements the 'low intermediate' level subset (see Section 7.2.1). Apart from the fact that it is under development the main deviation from the standard is the use of the `SCREEN SECTION`. This section describes all kinds of screen layout features not defined in the standard. (Telon also seems to offer screen design facilities; we have not encountered these in our test programs so far.) A second major deviation is the fact that paragraph names can be left out altogether, while in standard COBOL they are required.

Chapter 5

Tools

Starting out with our SDF definition of the syntax of ANSI COBOL III it is possible to build tools for reverse engineering COBOL programs in ASF. In this chapter we present and discuss an example of such a tool. First, we have to explain the PICTURE clause in further detail.

The PICTURE clause [pages VI-28..37] determines the layout format of user-defined items. Because the structure of date-fields has to be defined through a PICTURE clause, this field becomes essential when building tools that assist in solving year 2000 problem. Many other tools based on the PICTURE string could be developed. Observe that the PICTURE string is in fact the only (and rather crude) typing mechanism COBOL provides.

Our PICTURE specification is included in the Data-library module (see Appendix B.2.15). This is the only COBOL construct for which we specified semantics.

First the syntax and semantics of the PICTURE string are discussed, followed by a description of our year 2000 tool.

5.1 PICTURE syntax

The strings "PIC" and "PICTURE" are equivalent and interchangeable. A picture character string consists of certain allowable combinations of characters from the COBOL character set [page IV-4]. Lowercase letters are equivalent to their uppercase representations for the following symbols: A, B, P, S, V, X, Z, CR (credit) and DB (debit) [page VI-29]. Although this seems to imply that other symbols are not equivalent to their uppercase representations, we decided that this does not matter since no other characters are allowed in a PICTURE string (except for symbols like +, -, etc. that have no lower/uppercase representations).

The currency symbol can be used in a PICTURE string and must be the standard currency sign ($), except when specified in the SPECIAL-NAMES section of the environment division [page VI-17]. The specification of the currency symbol can be found in the lexical part of the Data-library module.

Several rules define the allowed number of symbols in a PICTURE string (rules (3)a and (6)a.1 [page VI-30], for example). As discussed in Chapter 4.2 these rules are not implemented syntactically. The following symbols are allowed only one occurrence in the PICTURE clause: S, V, period, +, -, CR, and DB. These symbols occur only in the 'strictly numeric' PICTURE strings. The last four are mutually exclusive. At most one currency symbol may be used. In the Numeric Edited PICTURE string the P and V may not occur in combination with the period.

Several more rules for the PICTURE string exist. We have not implemented every rule because it turned out that this was not necessary for building tools based on the

`PICTURE` string: we make an exception to our policy (of specifying every constraint we encounter) and assume that the `PICTURE` strings are correct in the COBOL programs under analysis.

5.2 PICTURE semantics

The meaning of the allowed symbols in a `PICTURE` string is provided 'in words' in the standard [pages VI-31, 32]. We briefly summarize them here.

- `A` – alphabetic character
- `B` – space character
- `P` – decimal point indicator ('scaling')
- `S` – indicates presence of operational sign
- `V` – decimal point indicator
- `X` – any allowed character
- `Z` – space when its value is zero in numeric string
- `9` – numeric character
- `0` and `/` and `,` – itself
- `.` – decimal point (or comma)
- `+` and `-` – sign control symbols
- `*` – itself when its position has value zero in numeric string

Furthermore, the currency sign may be used: either the standard ($) or the sign defined in the `SPECIAL-NAMES` paragraph of the environment division.

5.3 A year 2000 tool

5.3.1 The tool

"How could this happen?" In the 1960s and 1970s programmers tried to reduce record-size as much as possible because computer memory and storage were still very expensive [1]. Perhaps unexpectedly, many of these programs are still in use, and many of these programs are written in COBOL. The problem has been denied for many years, partially because people did not realize it existed and partially because it is a tremendous undertaking to fix it: the Gartner Group estimates that a medium sized company with about 8000 programs will spend about 4 million dollars to repair the software [1]. Clearly, contrary to some popular belief, the year 2000 problem is mainly a software problem.

ANSI apparently has published a separate extension to the standard for the year 2000 [11][1], but we have not been able to track this down yet. We did find some IBM documentation on the issue however; it describes three possible solutions for the year 2000 problem [5]:

- Conversion to full four-digit year format.

[1]It will probably contain guidelines for date-handling and perhaps some special statements related to the problem.

- Windowing techniques.
- Encoding/compression scheme.

Of these three, the latter two are not very reliable according to IBM: the windowing technique in short implies the use of a 100-year period in which dates with two-digit year notation can occur. If this period crosses a century boundary it is still possible, based on the starting value of the interval, to determine to which century any date within the period belongs. For example, if the interval is 1970 – 2070 then '55' should be read as 2055 and '85' should be read as 1985. This method is not very useful when the information processed can contain dates that span more than 100 years however. An example of this situation is a system that keeps track of people's birth dates and uses the current date for calculations: people turn 100 nowadays. Also, with this technique one knows for sure that sooner or later a problem will occur (although this part of the problem can perhaps be prevented by using the *sliding window* variant, see [5]).

The encoding/compression scheme is the least desirable of the three because it requires major changes to both the data and the source code of a program. Any kind of encoding/compression scheme, satisfying the goal of unifying the notation for the pre- and post-2000 dates, is imaginable.

The conversion to full four-digit year format is the most reliable method. Intuitively one would perhaps not be interested in two-digit date formats that occur in report headers, comments, and so on. A danger to this approach is that occurrences might be labeled 'cosmetic' too quickly. The IBM document gives the following examples:

- Today's reports might be written to a data set tomorrow.
- Display-only dates of today may prove useful as a collating value when archiving that output tomorrow to meet a new business or government standard.
- Even when viewed by a human, 2-digit dates can prove ambiguous if the data spans more than 100 years: '02' for example can indicate 1902 and 2002; if the context does not disambiguate this then humans too (who read the reports or comments etc.) will have problems dealing with these dates.

In spite of these warnings we have decided not to track down every occurrence of a possible date reference (for example in the case of comment entries and in the case of the `Report-group-description-entry`, because that entry is (probably) used merely as layout). Intuitively safe locations in a COBOL source program are not analyzed. The tool consists of the following functions that each take a COBOL source program as an argument.

- **Check**
  Generates a list of all variables together with possible `PICTURE` declarations and indicates which of those are possible date-descriptors.
- **Danger**
  Indicates if at least one possible date field is present in the program.
- **Show-dates**
  Gives a list of possible dates.
- **Density**
  Gives the number of possible date variables in the program. This is called the 'density' of the program.

Three qualifications for variables are possible: NO-DATE — most likely not a date variable; SAFE-DATE — most likely a date variable but no calculations take place on this variable or on any of its dependents (see the discussion on the MOVE statement below); RISK-DATE — most likely a date variable, and calculations take place on this variable or on one of its dependents. For now, the three qualifications are represented by '1', '2', and '3', respectively. The final three functions all depend on the check function. The check function determines if a variable is 'dangerous' using the following criteria.

- Does a declaration of a PICTURE string contain possible date-structures?[2]
- Does a variable *name* contain a possible date-reference? An answer to this question could be based on a list of short strings provided by IBM and Unibol (see Appendix C.1.1 and C.1.2). As an example we specified a rule to check if the variable name is 'CENT'.
- Is a 'clean' variable infected somewhere in the program by being assigned to a possible date variable through a MOVE statement. Here we used a simple data-flow technique: the transitive closure (or 'fixed point') is calculated by keeping track of a copy of the type environment containing all variables plus 'possible date'-flags; as soon as the type environment remains the same after a scan of the source code, every direct and indirect MOVE influence has been taken care of. This is a conservative approach, since if the source were checked dynamically, some MOVE's that are now labeled as dangerous might then be labeled as safe. It is our estimation that in practice (with COBOL programs) this will not be a major influence. The MOVE statement can infect variables in both 'directions', i.e. a dangerous variable in either the first or the second argument of the MOVE statement results in the infection of the other.
- Is a date variable used in a calculation (through the ADD statement; the SUBTRACT and COMPUTE statements can be handled in a similar way and are left for future work) anywhere in the source code? If somewhere in the source code a variable moves from being a '1' (probably not a date) or a '2' (possible date without operations) to a '3' (possible date with operations), then all variables (indirectly) linked to this variable through either MOVE statements or any of the calculation statements are changed into '3' as well. Again, this is not an optimal solution, but the tool is limited to static checks.

5.3.2 Example session

Figure 5.1 shows an example COBOL program under analysis of our year 2000 tool in its current form. We only show an example of the check function because all other functions are trivial extensions of it. Notice that the COBOL program does not execute the interesting statements in the procedure division, since they are included as comments (*) in the source. For our first test, only the trivial STOP RUN is 'executed'.
Analyzing this program results in the list of variables in Figure 5.2, with the following structure: variable name, ':', picture string, if any, ':', flag. The flags indicate (as mentioned): '1' for safe, '2' for date but probably a safe one, or '3' for risky date. Entries are separated by a # symbol.
Now, we start adding the statements that were listed as comments in Figure 5.1. Figure 5.3 shows the results of analyzing the program after the first ADD statement is executed. Analyzing the program with the first two or the first three ADD statements

[2] As mentioned, this should be analyzed with [6] and [10] as starting points.

```
check(
      IDENTIFICATION DIVISION.
      PROGRAM-ID. TEST-PROGRAM.
      DATA DIVISION.
      FILE SECTION.
            FD THE-FILE IS GLOBAL
            VALUE OF NUMBERS IS MANY
            CODE-SET IS PETER.

            01 FILE-VAR       PIC 99.99.99.

      WORKING-STORAGE SECTION.

            01 DIRTY          PIC 99.99.99.
            01 CLEAN          PIC 999.
                  03 OKAY     PIC AAAA.
            01 THIRD          PIC X.

      LINKAGE SECTION.

            01 LINKAGE-VAR.
            01 NO-DANGER.
                  03 VARIABLE PIC 99/99/99.

      COMMUNICATION SECTION.

            CD THE-CD-NAME FOR INITIAL INPUT
                  SYMBOLIC QUEUE IS TESTER
                  MESSAGE COUNT IS TEN

            01 COMM-VAR-SAFE.

            01 CENT           PIC XXX.

      PROCEDURE DIVISION.

      TEST-PARAGRAPH.

            * ADD OKAY TO VARIABLE.
            * ADD COMM-VAR-SAFE TO NO-DANGER.
            * ADD THIRD TO CLEAN ON SIZE ERROR STOP RUN
            * MOVE CLEAN TO DIRTY

            STOP RUN
)
```

Figure 5.1: A COBOL program under analysis

gives the same result: all three sequences of statements lead to the same result. Figure 5.4 shows the result when all four statements are executed.
Note that the order in which the statements are added to the procedure division is not important. The final `MOVE` statement could be added just as well as the first statement of the procedure division, before the first `ADD`. The propagation technique

```
[
 FILE-VAR : 99.99.99 :  2#
 DIRTY : 99.99.99 :  2#
 CLEAN : 9 9 9 :  1#
 OKAY : A A A A : 1#
 THIRD : X : 1#
 LINKAGE-VAR : :  1#
 NO-DANGER : :  1#
 VARIABLE : 99/99/99 :  2#
 COMM-VAR-SAFE : :  1#
 CENT : XXX : 2 ]
```

Figure 5.2: First test result

```
[
 FILE-VAR : 99.99.99 :  2#
 DIRTY : 99.99.99 :  2#
 CLEAN : 9 9 9 :  1#
 OKAY : A A A A : 3#
 THIRD : X : 1#
 LINKAGE-VAR : :  1#
 NO-DANGER : :  1#
 VARIABLE : 99/99/99 :  3#
 COMM-VAR-SAFE : :  1#
 CENT : XXX : 2 ]
```

Figure 5.3: Second, third, and fourth test result

```
[
 FILE-VAR : 99.99.99 :  2#
 DIRTY : 99.99.99 :  3#
 CLEAN : 9 9 9 :  3#
 OKAY : A A A A : 3#
 THIRD : X : 3#
 LINKAGE-VAR : :  1#
 NO-DANGER : :  1#
 VARIABLE : 99/99/99 :  3#
 COMM-VAR-SAFE : :  1#
 CENT : XXX : 2 ]
```

Figure 5.4: Fifth test result

we used ensures the proper handling of the variables involved, and identical results will be obtained.

5.3.3 Future extensions

Here are some suggestions for extensions that can be made to the tool. Most of these extensions can, starting with the current specification of our tool, be specified in a straightforward way: performance seems to be the biggest problem when extending the tool.

Non-standard restrictions A few non-standard restrictions imposed on input programs by the tool should be lifted. Most of our test programs, sometimes with minor alterations, were analyzed correctly in spite of these shortcomings. Here is, to the best of our knowledge, a complete list of the restrictions:

1. Once the words `FILE SECTION` occur in a source program, the tool requires that the body of the file section occurs as well, while the standard allows the body to be empty.
2. The 'sections' construct in the procedure division is not yet allowed. Just programs with a paragraph structure can be analyzed so far.
3. `MOVE` and `ADD` statements can not occur inside `PERFORM` statements. The tool will terminate but will probably produce incorrect results because the statements inside a `PERFORM` statement are not influencing the analysis.
4. An `ADD` statement can only have one identifier as an argument, while the standard allows for any number of arguments.
5. Nested programs are not yet allowed.
6. The `COPY` and `REPLACE` statements are not allowed in a source program: this should be taken care of by the pre-processor. (This one is not really a shortcoming of our tool.) Also, the `USE` statement is not yet allowed by our tool.
7. The `USING` construct of the procedure division is not yet allowed.

Improvements to the search results

- Not just identifiers, but integers and literals too should be checked for date-structures.
- Perhaps the report description entry should be checked for date-like occurrences as well. Although the entry appears to be cosmetic we remember IBM's warnings.
- The `SUBTRACT` and `COMPUTE` statements should be considered in determining whether calculation takes place on a date variable. So far we just specified the (most general form of the) `ADD` statement.
- Sorting operations should be indicated as high-risk operations. Perhaps another classification can be made that takes into account if a date field is used in a sorting operation.
- A check should be performed on comparisons involving the number 100 or 99. For example, the comparison `IF GREATER THAN 100` might be used in a COBOL source program to indicate that a four-digit year should be truncated to its two-digit 'equivalent'. Similarly, the tool should search for `ADD` statements where the number 1900 is involved.
- The `ACCEPT` statement should be checked separately for use of the `DATE` and `DAY` constants. The standard defines these conceptual fields in the 'dangerous' way [page VI-72]. This is the standard's way of dealing with the `DATE` field:

  > "(..) December 25, 1986, would be expressed as 861225."

  and the `DAY` field:

  > "(..) December 25, 1986, would be expressed as 86359."

It follows from the latter description that a field of five numeric positions, possibly separated by a period after two digits, can also be a date field. This requirement, together with probably many others that we have forgotten, should one day be incorporated in the search criteria.

- The tool should mark any programs called by the (infected) source program. These programs should then themselves be analyzed, keeping track of the exchange of dangerous variables.
- The `EXEC` extension should be analyzed because SQL-like statements are executed there which may very well contain infected date variables.
- Data in files on a disk should be inspected for dates as well. This might be useful as a first action to take when creating a set of dangerous variables. The tool should in one way or another be 'informed' about the dangerous information in the files.
- An important problem with year 2000 tools appears to be that the set of dangerous variables found by the tool turns out to be too big. For this, it might be possible to introduce some more categories into which variables can be divided. Instead of checking the `MOVE` statement in both directions and marking both the 'left' and the 'right' variable as dangerous, it might be a better idea not to consider the 'right-to-left infection' as dangerous as the 'left-to-right infection'. It is conceivable that a variable on the left side of the `MOVE` statement is just a number being added to a date variable. In that case the 'number variable', if it is used over and over again as a temporary variable, will lead to a tremendous increase in false alarms. Many other levels of infection can be thought of to fine-tune the tool.
- Finally, a distinction could perhaps be made between 'safe' (four-digit year) and 'dangerous' (two-digit year) dates, although a four digit variable can be dangerous too when two-digit user input is allowed and inserted in a four-digit variable.

Remarks on program transformation

- A new function could be written that performs a program transformation from an 'infected' to a 'clean' program, with hopefully identical functionality. Such a conversion function would start out with the check function as described above. It would then take the COBOL source program as input and perform updates on the definition of the possible date variables (expanding them to four-digit) and, and this is far from trivial, alter the source code to ensure the new date-format is used properly.

  In ASF+SDF the first steps for such a function would be to take the source program as done in the ASF specification of our year 2000 tool and return the identification division and the environment division unchanged. The data division would be transformed for example along the lines of the example transformation of a shortened version of the data description entry shown in Figure 5.5. Of course, many more formats for possible dates exist and should be incorporated in the `convert` function.

  A similar approach should be taken to transform the statements in the procedure division (which will turn out to be much more complicated).

- When using a COBOL `COMP-3` format a CCYYMMDD date can be packed into an existing 6-byte field [5]. This IBM extension can be helpful when

```
convert(XX.XX.XX) = XX.XX.XXXX

convert(picture) = safe-picture
====================================
transform( level-number (..)  picture (..)  )
= level-number (..)  safe-picture (..)
```

Figure 5.5: Possible structure of a conversion function for the data division

transforming an 'infected' source program to a 'clean' source program. We have not used this because we dealt primarily with standard COBOL.

Chapter 6

Related work

6.1 Bottom-up specification of COBOL II

In parallel with our project Alex Sellink, using a bottom-up approach, developed another COBOL definition in ASF+SDF [15]. This project started out with a large set of COBOL programs: given those programs and several manuals the syntax of COBOL II was specified in SDF.

6.1.1 Differences

The bottom-up approach leads to a smaller specification (approximately 400 versus 1100 lines of ASF+SDF code). We list the two main reasons for this reduction in size.

- Not all defined COBOL III constructs are used in the example programs that are used as the basis for the specification in [15].
- The bottom-up approach worked from the assumption that the code was syntactically correct at the moment of parsing. This reduces all kinds of syntax-code used for checking purposes, like the restriction that COBOL words can sometimes be no longer than 30 characters. (We never implemented this rule ourselves, but in our case it was a clear deviation from our policy.)

Because the size of the semantics is strongly influenced by the size of the syntax definition, the bottom-up approach reduces the amount of work necessary for specifying the semantics as well. Our approach could one day lead to a COBOL programming environment, but for our current goal (reverse engineering), it appears to be too detailed.

A closer inspection of our approach reveals that it has not been too different from the bottom-up approach. Because there are so many syntactical constraints and exceptions it has turned out to be almost impossible to specify it all in the time-frame set for this thesis. Therefore, we too often included only those syntactical features that were needed to succesfully parse example programs. Of course, the difference remains that we added only those features that the standard allows, while in the bottom-up approach any feature encountered was added.

Another, more important difference remains. We specified the COBOL III syntax while [15], using mainly programs with a 1974 basis, specified (a dialect based on) COBOL II.

6.1.2 Performance tests

During testing it turned out that our definition is slower than the definition resulting from the bottom-up approach. When we try to load our syntax definition in the ASF+SDF Meta-environment it turns out that it takes about thirty times longer instead of three times longer than [15] (fifteen minutes versus thirty seconds). Also, reducing a COBOL program of about a hundred lines takes about a minute with our specification, versus a few seconds with the bottom-up. Unfortunately we were not able to compare the performance of our tools because our tool is more elaborate than the tool described in [15], and our tool is not easy to split up in smaller parts. It is our strong impression however that here too our specification is slower. In terms of memory requirements as well, our specification has some severe problems. Running our tool on a program of more than some 300 lines will result in a fatal memory shortage, while the tool in [15] has no problems up to that point (and beyond).

Because this kind of performance tests and especially the factors that cause this behavior have never been studied thoroughly for ASF+SDF, we will try to make a start with that. Perhaps a future project can extend these remarks somewhat and come up with conclusions. The following factors could theoretically play an important role in the performance of the ASF+SDF system:

1. The number of modules. Because [15] uses a much larger number of modules this factor does not seem very important.
2. The size of the modules. Because we have two modules of size 13.000 bytes or over that are used by almost all other modules, and [15] has no modules of size more than 3.000 bytes, this might be a factor.
3. Import structure of modules. Our import structure is 'clean' in the sense that no 'lower' module needs some 'higher' module to be correct. Any substructure of modules can be loaded independently, therefore this is probably not a factor.
4. The average number of alternatives per sort. This could be an interesting factor. At first glance there does not seem to be a tremendous difference in this respect between the two specifications though.
5. The overall size of the specification. Perhaps there is a 'maximum size' above which performance collapses. This is most likely an important factor but it does not seem to explain a thirty-fold increase in time versus a three-fold increase in size.
6. Cycles in specification. Our specification had cycles in its parse tree; this may cause delays when parsing.

6.2 Other specifications

Here we briefly list some more related specification projects.

- **Semantics of COBOL control constructs**.
  In [12] a formal (denotational) description of the semantics of control constructs of COBOL II is presented. The control constructs under consideration are, as expected, the concept of paragraphs and sections, the `GO TO` statement, and the `PERFORM` statement. A large part of an implementation of this semantics is provided in this paper. This is done in the functional programming language *Standard ML*.

  We recognized ourselves in the final remark of the paper:

> "To conclude, we remark that in our project we use a slightly extended version of the semantics in order to analyze real world COBOL 74 programs."

Related to this is an article by the same authors [13]. Here semantics-based reverse engineering is introduced and advocated. COBOL II is again taken as an example, and the *AEMES* project is announced: reportedly suitable for reverse engineering of COBOL II programs.

- **COBOL and interest products.**
  In [22] interest products are modelled in the Risla[1] language. Then, Risla product specifications are translated into COBOL code. To do this, the syntax of COBOL was required: an SDF definition of a small subset of COBOL is presented in the paper.

- **Specification of programming languages.**
  An example of a full specification of a progamming language is the Pico specification [14]. This Pascal-like language is specified both syntactically and semantically, and can be executed. Many useful ideas for specifying COBOL semantics might be found in this specification. In [16] specifications of several other programming languages can be found.

- **Pretty printer for COBOL.**
  In [23] the generation of formatters for context-free languages is explained. As an example pretty print session the development of a COBOL formatter is presented.

[1] *Rente Informatie Systeem Language* (Interest Information System Language)

Chapter 7

Future work

A *lot* of future work can be done in the context of this thesis. First of all, in Section 4.2 choices were made in order to solve certain problems. In many cases these choices may not have been optimal, or even correct: they might very well be improved in a future project. The other chapters also introduced questions and suggestions for improvements to our work (especially our year 2000 tool, Section 5.3.3). All of these could be studied and worked on in future projects.

In Section 7.2 we discuss some possible reasons and techniques for reducing the size of the syntax specification. The current specification describes the entire standard, and in practice turns out to be too large. There might be other, more efficient ways of specifying COBOL.

First, here are some other suggestions for future work.

- The reverse engineering tool described in this thesis can be extended in many ways, as described in Section 5.3.3. One of the main problems of designing this tool was finding a detailed set of 'dangerous' COBOL constructs. Such a set is hard to find. The more restrictive, yet still correct, this set becomes, the more effective the tool will be.

- The COBOL syntax can be extended (hopefully conservatively) to recognize certain dialects of COBOL. When considering real-life application of the year 2000 tool this becomes essential. Currently, our pre-processor eliminates any deviation from the standard in a source program. After the conversion by the tool it will be impossible to obtain the non-standard features of the original program.

- Translating COBOL dialects into other dialects could be the subject of a new tool. The syntax definition described in this thesis can be a starting point for the specification of the source and the target languages (the two dialects), and a tool can be built along the lines of the `convert` function in Figure 5.5. Such a tool can be of help for companies trying to migrate from old to newer versions of COBOL as well. (In the case of upgrading from standard COBOL II to standard COBOL III [19] might be useful, since they have a section describing all changes from the '74 to '85 standard.)

- Sequential I-O alone is probably not enough for most software, some research to the conservativeness of extending a subset (or our specification) with the relative or indexed I-O functional modules might be an option. The standard only mentions that it is safe to add any of four optional (non-file-I-O) modules to any subset, so this is probably not safe with the other modules, including the file-I-O modules.

- The syntax specification might not be 100% error-free. We started our specification effort with a list of COBOL structures that appeared very much standard. However, once we obtained the standard itself this list turned out to contain quite some mistakes/deviations. Some errors due to these differences may still reside in the code. Moreover, as described earlier, there are so many rules in the standard (that affect syntax as well) that we are sure to have forgotten to implement some of them. An exceptionally clear example of this is our `Cobol-data` module, where, because of the excessive number of rules the standard prescribes, we have only implemented a limited number of features/restrictions that apply to COBOL data-items.

7.1 COBOL layout rules/pretty printing

Good documentation is essential for software engineering. In order to use pieces of COBOL source programs in a document, it is advisable to automatically convert COBOL code to the document's format. A pretty printer can do that job. A second reason for using a pretty printer for COBOL is that COBOL is character-oriented. For example, the first six positions of a source program are for labeling use only. Using a pretty printer as a pre-processor can eliminate many formatting errors. These errors in the source program may have been introduced by the year 2000 tool, after performing some program transformation. The methodology from [23] could be used to automatically generate a pretty printer for COBOL based on our specification.

The famous strict layout rules that COBOL uses in many cases appear no longer to be essential for disambiguation. These rules are mainly leftovers from (mechanical computer) history. Of the rules that are still in place but appear obsolete we *have* used a feature for disambiguation: the asterisk that is used as comment indicator may occur only at the seventh position in a program, directly following the optional line numbers. We need this because the asterisk also occurs in the `COMPUTE` statement, as multiplication operator. Because layout is recognized 'first' in ASF+SDF, this caused serious problems.

Another COBOL layout rule *is* still important: it defines the required indentation for data description entries. At the time our specification approached its final version we discovered that data description entries were parsed incorrectly. If we consider the example COBOL program in Figure 2.2, we see the definition of the `EMPLOYEE-RECORD` in the working-storage section. The fields that belong to this record definition are indented, and for a good reason: the level numbers that preceed the fields can be of arbitrary value. That means that some other mechanism is required for disambiguation and, of course, this turned out to be indentation. Our specification considered the definition of the `EMPLOYEE-RECORD` as a sequence of five equivalent data description entries at the highest level of structure, followed by another same-level data description entry (`01 DANGER`). Perhaps some semantic way of disambiguation can be worked out in a later project. A pre-processor could organize the working-storage section into `BEGIN` .. `END` blocks of some sort, enabling a correct parse.

A related problem that we discovered at the very end of writing this thesis as well (and that therefore ended up in the 'future work' section instead of in the 'problems and solutions' section) is the following. A record can, for example, be defined as shown in Figure 7.1.
Here we see that a year 2000 tool that scans for variable names will track down `DATE-OUT` but not `OUT`, which is the actual date variable. In the source code the `OUT` field will be referenced by an identifier. This identifier does not have to include qualification of the `OUT` field if this `OUT` field is unique in the program [page IV-23].

```
01 RENTAL-ADMINISTRATION
      05 ARTICLE-NUMBER
      05 DATE-OUT
            10 CLIENT NUMBER  PIC X(10)
            10 OUT            PIC X(8)
      05 DATE-IN
            10 IN
```

Figure 7.1: Date recognition problem

That means there is no immediate way to link the (risky) `OUT` field to the (risky but not considered as such) `DATE-OUT` field. Similarly, there is no immediate way to identify the `RENTAL-ADMINISTRATION` record as risky. Observe that our testprogram from Figure 5.1 in fact suffers from the same problem: the variable `NO-DANGER` remains safe in the fifth test result while it contains a date variable. Future work would therefore have to include a scan of the record definitions and the storing of all information directly or indirectly relevant to fields[1]. This is a simplified example of the problems that can occur due to this phenomenon because usually a year 2000 tool would scan for more than just variable names and would soon discover the dangerous `PIC X(8)` definition. Serious problems could still occur however when date fields are split up in day, month, and year sections. These sections would probably be represented by `XX`, which might be impossible to track down in a large COBOL source program because of 'noise' caused by other (safe) variables with an identical `PICTURE` definition.

Regardless of the necessity of the COBOL layout rules it is useful to pretty print the output of the year 2000 tool according to those rules for the fact that many COBOL compilers still require the traditional COBOL structure in their input source programs. (Again, the problem with wrongly parsed data description entries should be addressed first or pretty printing will produce incorrect results.)

7.2 Size reduction

Specifying the static semantics of COBOL was one of the original goals for this thesis, but we soon discovered that this would not fit into the time period set for the project: the standard defines syntax and semantics in words, using hundreds of descriptions of rules and exceptions to these rules. There are several advantages to reducing of the size of our ASF+SDF specification. First of all, considering the approximately twenty minutes that it takes our tool (see Chapter 5.3) to analyse a COBOL program of some 500 lines, performance should be taken into account. Smaller specifications result in faster execution. Secondly, a reduction of the sometimes redundant syntax will make specifying the static, and perhaps even dynamic, semantics a more realistic goal. Thirdly, building tools — for example for reverse engineering and data-flow analysis — will become easier as the specification becomes smaller. For reducing the size of a specification several options are available. The 'subset-approach' is discussed in the next section; here we list two other possibilities.

- Many redundant syntax constructs can be left out completely when one assumes that the source program is syntactically correct. We estimate that this, in combination with the bottom-up approach mentioned in Chapter 6, can lead to a reduction of nearly two-thirds. Also, the test set of programs

[1]Of course, the previous problem of wrongly parsed data description entries should be solved before such an extension becomes feasible.

used for the bottom-up specification will probably never use all features of the COBOL language which leads to a smaller syntax (for example, many of the obsolete features have a low probability of occuring in recent real-life programs).

- Another approach with results (probably) comparable to the bottom-up approach could be to translate COBOL into some target language with a much smaller syntax definition, but with the same expressivity. Then data-flow analysis etc. can be performed on this target language.
- It would be possible to considerably reduce the ASF+SDF code if 'options' were available. Each of the many optional words in COBOL requires an extra sort, as do many longer optional constructs in COBOL. Moreover, the optionality of these entities requires an extra line of ASF+SDF code. Our estimation is that the specification can be reduced by at least one third. The ASF+SDF 'optional' feature will be available in the near future [24].

7.2.1 Subsets

In order to reduce the size of the specification one might decide to choose a COBOL III standard subset. For this thesis we considered doing this, but eventually (after several tries) decided not to, for the following reasons:

- The chance that a company uses standard COBOL is very small, the chance that their software conforms to a COBOL subset is a lot smaller.
- Starting from the full syntax specification it is a non-trivial task to specify a subset. Apart from removing items, functionalities change when going from 'full-COBOL' to 'subset-COBOL'.
- Even for the smallest standard subset specifying the semantics was impossible in the time frame set for this thesis.

In spite of these practical problems, it might very well be an interesting project to specify a subset, including semantics. A short introduction to standard COBOL subsets follows.

Standard subsets. COBOL III supports three standards subsets; the *minimal*, the *intermediate*, and the *high* subset. The subset mechanism works as follows. Every statement can occur at a certain level. There are three possible levels, 0, 1, and 2, although not all statements have a level 0 and a level 2 occurrence. Level 0 means the statement is not implemented, level 1 means the basic functionality of the statement is implemented (but e.g. several optional constructs may not be defined), and level 2 means all functionality is implemented.

Then, every subset is defined via a table. For each functional unit (of which eleven exist, but four are optional and are not required for the subsets) it is mentioned at what level statements, or rather all language elements, most occur if you want to implement the chosen subset in a standard way. For example, the high subset has to support all highest levels in all units for all statements. The minimal subset on the other hand has to define its statements only at level 1, and only for three units. Not all units define every level, so this reduces the possible requirements for a subset for those units.

Chapter 8

Conclusions

Problems like the year 2000 problem can, in theory, be handled in a powerful way using the approaches presented in this thesis. But COBOL is a big language, and many even bigger dialects exist. The performance of tools based on the syntax and semantics of COBOL therefore becomes an important factor. The importance of the performance of the ASF+SDF Meta-environment is reduced when the specification is used merely as a design mechanism for implementations of tools written in more efficient languages.

On a more philosophical level, one might wonder if the ASF+SDF formalism is sufficiently equiped to specify 'oldfashioned' languages like COBOL. Trying to (algebraically) process hundreds of pages of exceptions to rules appears unnatural indeed. The situation gets worse when one realizes that often the rules are not quite unambiguous themselves.

While writing this thesis it became clear to us that, although it is sometimes hard to translate constructs from standard COBOL into constructs written in SDF, it is not an impossible task to do so. We believe that, because ASF+SDF provides many useful features, creating reverse engineering tools like the year 2000 tool described in this thesis will be much more difficult in an imperative language.

Of course, for solving problems of huge software systems, tools such as the one described in this thesis will never give a 100% good solution. However, the same is probably true for any other conceivable tool. To perform an analysis, rules describing the format and use of the language elements are required. Any definition of a set of such rules will, if the code under scrutiny is sufficiently large and complex, always be an approximation.

Considering the extensive list of future work one can come up with related to this subject, and considering the fact that the problem has not been recognized for such a long time, there is serious reason to doubt if every currently 'infected' software system will be cured and running safely by the turn of the millenium.

Appendix A

Obsolete language elements

The following language elements are mentioned as obsolete in the COBOL III standard[1] [pages XVII-81..86]:

- `ALL literal` and `numeric` or `numeric edited item`. The figurative constant `ALL` is obsolete.
- `AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY` paragraphs.
- `MEMORY SIZE` clause.
- `RERUN` clause.
- `MULTIPLE FILE TAPE` clause.
- `LABEL RECORDS` clause.
- `VALUE OF` clause.
- `DATA RECORDS` clause.
- `ALTER` statement.
- `KEY` phrase of the `DISABLE` and `ENABLE` statement.
- `ENTER` statement.
- `REVERSED` phrase of the `OPEN` statement.
- `STOP literal` statement.
- Segmentation module.
- Debug module.

[1] This is not a complete list, only features of (some) importance to the thesis and the specification are included.

Appendix B

Syntax of COBOL III

In this appendix the ASF+SDF specification of the full syntax of COBOL III is presented. Additional remarks are made throughout the specification. Section B.1 first gives a graphical representation of the import structure used by the ASF+SDF modules.

B.1 Import relations

Figure B.1 shows our division of functional modules plus the corresponding import structure. Somewhat 'non-standard' here is the use of the arrows: an arrow from A to B means that B is imported ('used') by A. Explanations of the shorthands for the division-names are listed below the graph, together with references to the sections in the appendix where the specifications of the modules can be found. References to the corresponding pages in the standard are provided as the opening line of the source code of each ASF+SDF module.

B.2 Specification

B.2.1 Cobol-Syntax

COBOL III syntax specification [pages V-39, 40].

imports Identification-division$^{B.2.2}$ Environment-division$^{B.2.3}$ Data-division$^{B.2.4}$
Procedure-division$^{B.2.5}$ Cobol-data$^{B.2.17}$
exports
sorts PROGRAM END-PROGRAM OPT-LOOP NESTED-PROGRAM

A COBOL program, possibly containing nested program(s), as defined on [page VI-3] and on [page V-39]:

context-free syntax

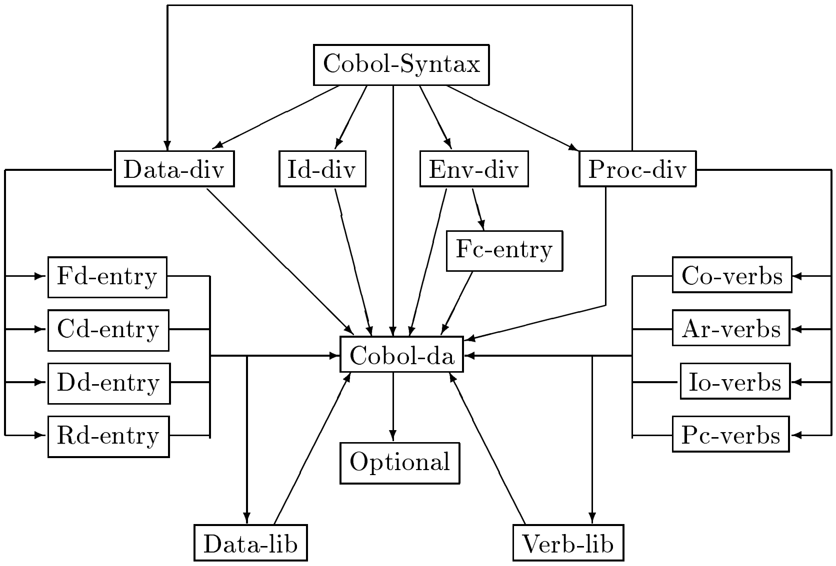
ID-DIV ENV-DIV DATA-DIV PROC-DIV OPT-LOOP → PROGRAM

→ OPT-LOOP
NESTED-PROGRAM∗ END-PROGRAM → OPT-LOOP

"END" "PROGRAM" ID-NAME "." → END-PROGRAM

A sequence of (mainly nested) source programs [page V-40]:

context-free syntax



Ar-verbs	Arithmetic Verbs (B.2.12)
Cd-entry	Communication Description Entry (B.2.8)
Co-verbs	Common Verbs (B.2.11)
Cobol-da	Cobol Data, Expressions, Conditions (B.2.17)
Cobol-Syntax	Main file (B.2.1)
Data-div	Data Division (B.2.4)
Data-lib	Data Division Library (B.2.15)
Dd-entry	Data Description Entry (B.2.9)
Env-div	Environment Division (B.2.3)
Fc-entry	File Control Entry (B.2.6)
Fd-entry	File Description Entry (B.2.7)
Id-div	Identification Division (B.2.2)
Io-verbs	Input-Output Verbs (B.2.13)
Optional	Optionals (B.2.18)
Pc-verbs	Program Control Verbs (B.2.14)
Proc-div	Procedure Division (B.2.5)
Rd-entry	Report (Group) Description Entry (B.2.10)
Verb-lib	Verbs Library (B.2.16)

Figure B.1: Import relations Cobol III syntax specification

NESTED-PROGRAM+ PROGRAM → PROGRAM

A nested source program [page V-39]:

context-free syntax
ID-DIV ENV-DIV DATA-DIV PROC-DIV
NESTED-PROGRAM∗ END-PROGRAM → NESTED-PROGRAM

B.2.2 Identification-division

Identification Division [page V-2].

```
imports Cobol-data^B.2.17
exports
  sorts ID-DIV PROGRAM-ID INFORMATION COMMENT-ENTRY
        INFORMATION1 INFORMATION2 INFORMATION3
        INFORMATION4 INFORMATION5
  context-free syntax
    "IDENTIFICATION" "DIVISION." PROGRAM-ID "."
    INFORMATION                                        → ID-DIV
```

A program that uses the COMMON clause is to be included in some other program [page X-12]. We do not check for this requirement in our specification.

```
  context-free syntax
    "PROGRAM-ID." ID-NAME                                         → PROGRAM-ID
    "PROGRAM-ID." ID-NAME OPT-IS "COMMON" "PROGRAM"               → PROGRAM-ID
    "PROGRAM-ID." ID-NAME OPT-IS "INITIAL" "PROGRAM"              → PROGRAM-ID

    INFORMATION1 INFORMATION2 INFORMATION3 INFORMATION4
    INFORMATION5                                                  → INFORMATION

                                                                  → INFORMATION1
    "AUTHOR." COMMENT-ENTRY*                                      → INFORMATION1
                                                                  → INFORMATION2
    "INSTALLATION." COMMENT-ENTRY*                                → INFORMATION2
                                                                  → INFORMATION3
    "DATE-WRITTEN." COMMENT-ENTRY*                                → INFORMATION3
                                                                  → INFORMATION4
    "DATE-COMPILED." COMMENT-ENTRY*                               → INFORMATION4
                                                                  → INFORMATION5
    "SECURITY." COMMENT-ENTRY*                                    → INFORMATION5

    STRING-ELT                                                    → COMMENT-ENTRY
```

B.2.3 Environment-division

Environment Division [page V-3].

```
imports Cobol-data^B.2.17 File-control-entry^B.2.6
exports
  sorts ENV-DIV CONF-SECTION IO-SECTION SOURCE-COMP
        OBJ-COMP SPECIAL-NAMES ENV-DEBUG-OPTION
        SEQ-OPTION SEGM-OPTION STATUS ON-OFF ON OFF
        ALPH-TYPE ALPH-SUBTYPE ALPH-SUBSUBTYPE
        SYMBOLICS CLASS CLASS-TYPE CLASS-SUBTYPE
        CURR-SIGN DEC-POINT IO-CONTROL SAME
        OPT-IO-CONTROL MULTIPLE RECORD-OR-SORT POSITION
        OPT-POSITION SYMB-NAME ENV-IMPL-NAME ALPH-NAME
        MEM-OPTION W-C-M
  context-free syntax
                                                          → ENV-DIV
    "ENVIRONMENT" "DIVISION." CONF-SECTION IO-SECTION → ENV-DIV
```

Configuration section:

```
  context-free syntax
                                                                  → CONF-SECTION
```

"CONFIGURATION" "SECTION." SOURCE-COMP OBJ-COMP SPECIAL-NAMES	→ CONF-SECTION
	→ SOURCE-COMP
"SOURCE-COMPUTER."	→ SOURCE-COMP
"SOURCE-COMPUTER." ID-NAME ENV-DEBUG-OPTION "."	→ SOURCE-COMP
	→ ENV-DEBUG-OPTION
OPT-WITH "DEBUGGING" "MODE."	→ ENV-DEBUG-OPTION
	→ OBJ-COMP
"OBJECT-COMPUTER."	→ OBJ-COMP
"OBJECT-COMPUTER." ID-NAME MEM-OPTION SEQ-OPTION SEGM-OPTION "."	→ OBJ-COMP
	→ MEM-OPTION
"MEMORY" OPT-SIZE INTEGER W-C-M	→ MEM-OPTION
"WORDS"	→ W-C-M
"CHARACTERS"	→ W-C-M
"MODULES"	→ W-C-M
	→ SEQ-OPTION
OPT-PROGRAM OPT-COLLATING "SEQUENCE" OPT-IS ID-NAME	→ SEQ-OPTION
	→ SEGM-OPTION
"SEGMENT-LIMIT" OPT-IS ID-NAME	→ SEGM-OPTION
	→ SPECIAL-NAMES
"SPECIAL-NAMES."	→ SPECIAL-NAMES
"SPECIAL-NAMES." ENV-IMPL-NAME∗ ALPH-NAME∗ SYMB-NAME∗ CLASS∗ CURR-SIGN DEC-POINT "."	→ SPECIAL-NAMES
ID-NAME STATUS	→ ENV-IMPL-NAME
OPT-IS ID-NAME ON-OFF	→ STATUS
OPT-IS ID-NAME	→ STATUS
ON-OFF	→ STATUS
"ON" OPT-STATUS OPT-IS ID-NAME OFF	→ ON-OFF
"OFF" OPT-STATUS OPT-IS ID-NAME ON	→ ON-OFF
	→ OFF
"OFF" OPT-STATUS OPT-IS ID-NAME	→ OFF
	→ ON
"ON" OPT-STATUS OPT-IS ID-NAME	→ ON
"ALPHABET" ID-NAME OPT-IS ALPH-TYPE	→ ALPH-NAME
"STANDARD-1"	→ ALPH-TYPE
"STANDARD-2"	→ ALPH-TYPE
"NATIVE"	→ ALPH-TYPE
ID-NAME	→ ALPH-TYPE
CBL-LITERAL ALPH-SUBTYPE∗	→ ALPH-TYPE
CBL-LITERAL THROUGH CBL-LITERAL	→ ALPH-SUBTYPE
ALPH-SUBSUBTYPE∗	→ ALPH-SUBTYPE
"ALSO" CBL-LITERAL	→ ALPH-SUBSUBTYPE
"SYMBOLIC" "CHARACTERS" SYMBOLICS+ IN ID-NAME	→ SYMB-NAME
"SYMBOLIC" "CHARACTERS" SYMBOLICS+	→ SYMB-NAME

CHARACTER OPT-IS INTEGER	→ SYMBOLICS
CHARACTERS OPT-ARE INTEGER INTEGER+	→ SYMBOLICS
"CLASS" ID-NAME OPT-IS CLASS-TYPE	→ CLASS
CBL-LITERAL CLASS-SUBTYPE	→ CLASS-TYPE
	→ CLASS-SUBTYPE
THROUGH CBL-LITERAL	→ CLASS-SUBTYPE
	→ CURR-SIGN
"CURRENCY" OPT-SIGN OPT-IS CURRENCY-SYMBOL	→ CURR-SIGN
	→ DEC-POINT
"DECIMAL-POINT" OPT-IS "COMMA"	→ DEC-POINT

Input-output section:

context-free syntax

	→ IO-SECTION
"INPUT-OUTPUT" "SECTION." "FILE-CONTROL." FILE-CONTR-ENTRY∗ IO-CONTROL	→ IO-SECTION
	→ IO-CONTROL
"IO-CONTROL." OPT-IO-CONTROL	→ IO-CONTROL
SAME∗ MULTIPLE∗ "."	→ OPT-IO-CONTROL
"SAME" RECORD-OR-SORT OPT-AREA OPT-FOR ID-NAME ID-NAME+	→ SAME
	→ RECORD-OR-SORT
"RECORD"	→ RECORD-OR-SORT
"SORT"	→ RECORD-OR-SORT
"SORT-MERGE"	→ RECORD-OR-SORT
"MULTIPLE" "FILE" OPT-TAPE OPT-CONTAINS POSITION+	→ MULTIPLE
ID-NAME OPT-POSITION	→ POSITION
	→ OPT-POSITION
"POSITION" INTEGER	→ OPT-POSITION

B.2.4 Data-division

Data Division [page V-8].

imports Cobol-data$^{B.2.17}$ File-description-entry$^{B.2.7}$ Data-description-entry$^{B.2.9}$ Report-description-entry$^{B.2.10}$ Communication-description-entry$^{B.2.8}$
exports
sorts DATA-DIV FILE-SECTION WORKING-STORAGE-SECTION LINKAGE-SECTION COMM-SECTION REPORT-SECTION FILE-SECTION-BODY REPORT-SECTION-BODY COMM-SECTION-BODY
context-free syntax

	→ DATA-DIV
"DATA" "DIVISION." FILE-SECTION WORKING-STORAGE-SECTION LINKAGE-SECTION COMM-SECTION REPORT-SECTION	→ DATA-DIV
	→ FILE-SECTION
"FILE" "SECTION." FILE-SECTION-BODY∗	→ FILE-SECTION

```
S-R-I-FD-ENTRY RECORD-DESCR-ENTRY+                          → FILE-SECTION-BODY

SORT-MERGE-ENTRY RECORD-DESCR-ENTRY+                        → FILE-SECTION-BODY

REP-DESCR-ENTRY                                             → FILE-SECTION-BODY

                                                            → WORKING-STORAGE-SECTION
"WORKING-STORAGE" "SECTION." RECORD-DESCR-ENTRY*            → WORKING-STORAGE-SECTION

                                                            → LINKAGE-SECTION
"LINKAGE" "SECTION." RECORD-DESCR-ENTRY*                    → LINKAGE-SECTION

                                                            → COMM-SECTION
"COMMUNICATION" "SECTION." COMM-SECTION-BODY*               → COMM-SECTION

COMM-DESCR-ENTRY RECORD-DESCR-ENTRY*                        → COMM-SECTION-BODY

                                                            → REPORT-SECTION
"REPORT" "SECTION." REPORT-SECTION-BODY*                    → REPORT-SECTION

REPORT-DESCR-ENTRY REPORT-GROUP-DESCR-ENTRY*                → REPORT-SECTION-BODY
```

B.2.5 Procedure-division

Procedure division [page V-19].

imports Data-division$^{B.2.4}$ Cobol-data$^{B.2.17}$ Common-verbs$^{B.2.11}$
Input-output-verbs$^{B.2.13}$ Program-control-verbs$^{B.2.14}$ Arithmetic-verbs$^{B.2.12}$
exports
sorts PROC-DIV PARAGRAPH SENTENCE DECLARATIVES
PROC-USING PROC-HEADER DECL-SECTION SECTION NR
STATEMENT2
context-free syntax

```
                                                            → PROC-DIV
PROC-HEADER DECLARATIVES SECTION+                           → PROC-DIV
PROC-HEADER PARAGRAPH+                                      → PROC-DIV

"PROCEDURE" "DIVISION" PROC-USING "."                       → PROC-HEADER

                                                            → PROC-USING
"USING" ID-NAME+                                            → PROC-USING

                                                            → DECLARATIVES
"DECLARATIVES." DECL-SECTION+ "END" "DECLARATIVES."         → DECLARATIVES

ID-NAME "SECTION" NR "." USE-STATEMENT "." PARAGRAPH*       → DECL-SECTION

ID-NAME "SECTION" NR "." PARAGRAPH*                         → SECTION

                                                            → NR
INTEGER                                                     → NR
```

The following definition of a sentence and a paragraph can be found in the standard [page IV-35]. The `IF` statement is given special treatment here (see remarks in text).
Following the period in the `PARAGRAPH` specification line there should be a space. We have not implemented this layout-feature.

context-free syntax

```
ID-NAME "." SENTENCE*                       → PARAGRAPH

STATEMENT* IF-STATEMENT-NO-ENDIF            → STATEMENT2
STATEMENT* IF-STATEMENT-NO-ELSE             → STATEMENT2
STATEMENT                                   → STATEMENT2
```

STATEMENT2+ "."	→ SENTENCE
STATEMENT+	→ IMPERATIVE-STATEMENT
STATEMENT+ "."	→ IMPERATIVE-STATEMENT

B.2.6 File-control-entry

File Control Entry, part of Environment Division [page V-5].
imports Cobol-data[B.2.17]
exports
sorts FILE-CONTR-ENTRY SEQUENTIAL-FILE RELATIVE-FILE INDEXED-FILE SORT-OR-MERGE-FILE REPORT-FILE FILE-HEADER RESERVE SEQ-ORG PADDING DELIMITER SEQ-ACCESS-MODE FILE-STATUS REL-ORG KEY IND-ORG IND-ACCESS-MODE SELECT ASSIGN AREA ST-OR-ID-NAME REL-ACCESS-MODE-2 REL-ACCESS-MODE OPT-REL-KEY IND-ACCESS-MODE-2 OPT-ALTERNATE OPT-DUPLICATES ID-NAME-OR-LIT ST-OR-ID

Note that this specification of file types is ambiguous (see discussion in Chapter 7). For our testing we only used the sequential file definition.

context-free syntax

SEQUENTIAL-FILE	→ FILE-CONTR-ENTRY
RELATIVE-FILE	→ FILE-CONTR-ENTRY
INDEXED-FILE	→ FILE-CONTR-ENTRY
SORT-OR-MERGE-FILE	→ FILE-CONTR-ENTRY
REPORT-FILE	→ FILE-CONTR-ENTRY
SEQUENTIAL-FILE	→ REPORT-FILE
FILE-HEADER RESERVE SEQ-ORG PADDING DELIMITER SEQ-ACCESS-MODE FILE-STATUS "."	→ SEQUENTIAL-FILE
FILE-HEADER RESERVE REL-ORG REL-ACCESS-MODE FILE-STATUS "."	→ RELATIVE-FILE
FILE-HEADER RESERVE IND-ORG IND-ACCESS-MODE KEY FILE-STATUS "."	→ INDEXED-FILE
"SELECT" ID-NAME "ASSIGN" OPT-TO ID-NAME-OR-LIT+ "."	→ SORT-OR-MERGE-FILE
SELECT ASSIGN	→ FILE-HEADER
"SELECT" "OPTIONAL" ID-NAME	→ SELECT
"SELECT" ID-NAME	→ SELECT
"ASSIGN" OPT-TO ID-NAME-OR-LIT+	→ ASSIGN
	→ RESERVE
"RESERVE" INTEGER AREA	→ RESERVE
OPT-AREA	→ AREA
OPT-AREAS	→ AREA
	→ SEQ-ORG
"ORGANIZATION" OPT-IS "SEQUENTIAL"	→ SEQ-ORG
"ORGANIZATION" OPT-IS "RELATIVE"	→ REL-ORG

```
“ORGANIZATION” OPT-IS “INDEXED”                          → IND-ORG

                                                         → PADDING
“PADDING” OPT-CHARACTER OPT-IS ID-NAME-OR-LIT            → PADDING

                                                         → DELIMITER
“RECORD” “DELIMITER” OPT-IS ST-OR-ID-NAME                → DELIMITER

“STANDARD-1”                                             → ST-OR-ID
ID-NAME                                                  → ST-OR-ID

                                                         → SEQ-ACCESS-MODE
“ACCESS” OPT-MODE OPT-IS “SEQUENTIAL”                    → SEQ-ACCESS-MODE

                                                         → FILE-STATUS
OPT-FILE “STATUS” OPT-IS ID-NAME                         → FILE-STATUS

                                                         → REL-ACCESS-MODE
“ACCESS” OPT-MODE OPT-IS REL-ACCESS-MODE-2               → REL-ACCESS-MODE

“SEQUENTIAL” OPT-REL-KEY                                 → REL-ACCESS-MODE-2
“RANDOM” “RELATIVE” KEY-OPTION                           → REL-ACCESS-MODE-2
“DYNAMIC” “RELATIVE” KEY-OPTION                          → REL-ACCESS-MODE-2

“RELATIVE” OPT-KEY OPT-IS ID-NAME                        → OPT-REL-KEY

                                                         → IND-ACCESS-MODE
“ACCESS” OPT-MODE OPT-IS IND-ACCESS-MODE-2               → IND-ACCESS-MODE

“SEQUENTIAL”                                             → IND-ACCESS-MODE-2
“RANDOM”                                                 → IND-ACCESS-MODE-2
“DYNAMIC”                                                → IND-ACCESS-MODE-2

“RECORD” OPT-KEY OPT-IS ID-NAME OPT-ALTERNATE            → KEY

                                                         → OPT-ALTERNATE
“ALTERNATE” “RECORD” OPT-KEY OPT-IS ID-NAME
OPT-DUPLICATES                                           → OPT-ALTERNATE

                                                         → OPT-DUPLICATES
OPT-WITH “DUPLICATES”                                    → OPT-DUPLICATES

ID-NAME                                                  → ID-NAME-OR-LIT
CBL-LITERAL                                              → ID-NAME-OR-LIT
```

B.2.7 File-description-entry

File Description Entry, part of Data Division [page V-9].

imports Cobol-data$^{B.2.17}$ Data-library$^{B.2.15}$
exports
sorts S-R-I-DESCR-ENTRY LINAGE CODE-SET S-R-I-FD-ENTRY
SORT-MERGE-ENTRY REP-DESCR-ENTRY BLOCK RECORD
LABEL VALUE DATA START-INT RECORD-DESCR
STANDARD-OMITTED FOOTING TOP BOTTOM REPORTS
FROM-TO DEPENDING FROM-INT TO IMPL-NAME
REC-OR-CHARS DATA-DESCR

Note: `SD` means Sort-merge Description entry and `FD` means File Description entry

context-free syntax

```
S-R-I-DESCR-ENTRY LINAGE CODE-SET “.”                    → S-R-I-FD-ENTRY
```

"SD" ID-NAME RECORD DATA "."	→ SORT-MERGE-ENTRY
"FD" ID-NAME EXTERNAL GLOBAL BLOCK RECORD LABEL VALUE CODE-SET REPORTS	→ REP-DESCR-ENTRY
"FD" ID-NAME EXTERNAL GLOBAL BLOCK RECORD LABEL VALUE DATA	→ S-R-I-DESCR-ENTRY
	→ BLOCK
"BLOCK" OPT-CONTAINS START-INT INTEGER REC-OR-CHARS	→ BLOCK
"RECORDS"	→ REC-OR-CHARS
OPT-CHARACTERS	→ REC-OR-CHARS
	→ RECORD
"RECORD" RECORD-DESCR	→ RECORD
OPT-CONTAINS INTEGER OPT-CHARACTERS	→ RECORD-DESCR
OPT-IS "VARYING" OPT-IN OPT-SIZE FROM-TO DEPENDING	→ RECORD-DESCR
OPT-CONTAINS INTEGER "TO" INTEGER OPT-CHARACTERS	→ RECORD-DESCR
FROM-INT TO OPT-CHARACTERS	→ FROM-TO
	→ FROM-INT
OPT-FROM INTEGER	→ FROM-INT
	→ TO
"TO" INTEGER	→ TO
	→ DEPENDING
"DEPENDING" OPT-ON ID-NAME	→ DEPENDING
	→ START-INT
INTEGER "TO"	→ START-INT
	→ LABEL
"LABEL" "RECORD" OPT-IS STANDARD-OMITTED	→ LABEL
"LABEL" "RECORDS" OPT-ARE STANDARD-OMITTED	→ LABEL
"STANDARD"	→ STANDARD-OMITTED
"OMITTED"	→ STANDARD-OMITTED
	→ VALUE
"VALUE" OF IMPL-NAME+	→ VALUE
ID-NAME OPT-IS ID-OR-LIT	→ IMPL-NAME

Here is an example of how we (partially) syntactically check the number of arguments: the third `DATA` line requires at least two instances of `ID-NAME`, where the second line only allows for one. The counting here (and in other places in the specification) is justified purely by common sense, not by the standard (see also the discussion in Section 4.1).

context-free syntax

	→ DATA
"DATA" DATA-DESCR	→ DATA
"RECORD" OPT-IS ID-NAME	→ DATA-DESCR
"RECORDS" OPT-ARE ID-NAME ID-NAME+	→ DATA-DESCR
	→ LINAGE
"LINAGE" OPT-IS ID-OR-INT OPT-LINES FOOTING TOP BOTTOM	→ LINAGE
	→ FOOTING

```
    OPT-WITH “FOOTING” OPT-AT ID-OR-INT                          → FOOTING

                                                                 → TOP
    OPT-LINES OPT-AT “TOP” ID-OR-INT                             → TOP

                                                                 → BOTTOM
    OPT-LINES OPT-AT “BOTTOM” ID-OR-INT                          → BOTTOM

                                                                 → CODE-SET
    “CODE-SET” OPT-IS ID-NAME                                    → CODE-SET
  context-free syntax
    “REPORT” OPT-IS ID-NAME                   → REPORTS
    “REPORTS” OPT-ARE ID-NAME ID-NAME+        → REPORTS
```

B.2.8 Communication-description-entry

Communication Description Entry [pages V-15, 16].

```
imports Cobol-data^{B.2.17}
exports
  sorts COMM-DESCR-ENTRY SYMB-QUEUE SYMB-SUB-QUEUE1
        SYMB-SUB-QUEUE2 SYMB-SUB-QUEUE3 MESSAGE-DATE
        MESSAGE-TIME SYMB-SOURCE TEXT-LENGTH END-KEY
        STATUS-KEY MESSAGE-COUNT ID-LIST1 ID-LIST2
        DESTINATION-COUNT DESTINATION-TABLE ERROR-KEY
        SYMB-DESTINATION SYMB-TERMINAL INPUT-OPTIONS
  context-free syntax
    “CD” ID-NAME OPT-FOR OPT-INITIAL “INPUT” INPUT-OPTIONS → COMM-DESCR-ENTRY
    “CD” ID-NAME OPT-FOR “OUTPUT” DESTINATION-COUNT
    TEXT-LENGTH STATUS-KEY DESTINATION-TABLE ERROR-KEY
    SYMB-DESTINATION “.”                                   → COMM-DESCR-ENTRY
    “CD” ID-NAME OPT-FOR OPT-INITIAL “I-O” MESSAGE-DATE
    MESSAGE-TIME SYMB-TERMINAL TEXT-LENGTH END-KEY
    STATUS-KEY ID-LIST2                                    → COMM-DESCR-ENTRY

    SYMB-QUEUE SYMB-SUB-QUEUE1 SYMB-SUB-QUEUE2
    SYMB-SUB-QUEUE3 MESSAGE-DATE MESSAGE-TIME
    SYMB-SOURCE TEXT-LENGTH END-KEY STATUS-KEY
    MESSAGE-COUNT ID-LIST1                                 → INPUT-OPTIONS

                                                           → SYMB-QUEUE
    OPT-SYMBOLIC “QUEUE” OPT-IS ID-NAME                    → SYMB-QUEUE

                                                           → SYMB-SUB-QUEUE1
    OPT-SYMBOLIC “SUB-QUEUE-1” OPT-IS ID-NAME              → SYMB-SUB-QUEUE1

                                                           → SYMB-SUB-QUEUE2
    OPT-SYMBOLIC “SUB-QUEUE-2” OPT-IS ID-NAME              → SYMB-SUB-QUEUE2

                                                           → SYMB-SUB-QUEUE3
    OPT-SYMBOLIC “SUB-QUEUE-3” OPT-IS ID-NAME              → SYMB-SUB-QUEUE3

                                                           → MESSAGE-DATE
    “MESSAGE” “DATE” OPT-IS ID-NAME                        → MESSAGE-DATE

                                                           → MESSAGE-TIME
    “MESSAGE” “TIME” OPT-IS ID-NAME                        → MESSAGE-TIME

                                                           → SYMB-SOURCE
    OPT-SYMBOLIC “SOURCE” OPT-IS ID-NAME                   → SYMB-SOURCE

                                                           → TEXT-LENGTH
```

“TEXT” “LENGTH” OPT-IS ID-NAME → TEXT-LENGTH

→ END-KEY
“END” “KEY” OPT-IS ID-NAME → END-KEY

→ STATUS-KEY
“STATUS” “KEY” OPT-IS ID-NAME → STATUS-KEY

→ MESSAGE-COUNT
OPT-MESSAGE “COUNT” OPT-IS ID-NAME → MESSAGE-COUNT

→ ID-LIST1
ID-NAME “,” ID-NAME “,” ID-NAME “,” ID-NAME “,” ID-NAME “,”
ID-NAME “,” ID-NAME “,” ID-NAME “,” ID-NAME “,”
ID-NAME “,” ID-NAME → ID-LIST1

→ ID-LIST2
ID-NAME “,” ID-NAME “,” ID-NAME “,” ID-NAME “,”
ID-NAME “,” ID-NAME → ID-LIST2

→ DESTINATION-COUNT
“DESTINATION” “COUNT” OPT-IS ID-NAME → DESTINATION-COUNT

→ DESTINATION-TABLE
“DESTINATION” “TABLE” “OCCURS” INTEGER OPT-TIMES
INDEXED → DESTINATION-TABLE

→ ERROR-KEY
“ERROR” “KEY” OPT-IS ID-NAME → ERROR-KEY

→ SYMB-DESTINATION
OPT-SYMBOLIC “DESTINATION” OPT-IS ID-NAME → SYMB-DESTINATION

→ SYMB-TERMINAL
OPT-SYMBOLIC “TERMINAL” OPT-IS ID-NAME → SYMB-TERMINAL

B.2.9 Data-description-entry

Data Description Entry [pages V-13, 14].

imports Cobol-data$^{B.2.17}$ Data-library$^{B.2.15}$
exports
sorts RECORD-DESCR-ENTRY DATA-DESCR-ENTRY
ID-NAME-OR-FILLER REDEFINES USAGE-BODY USAGE
DD-VALUE SYNCHRONIZE OCCURS-A-OR-D OCCURS
LEFT-OR-RIGHT DD-VALUES ARG-88 ARG-88-OPTION
DD-THROUGH

A Record Description Entry is the total set of Data Description Entries required for the definition of a record [page III-19]. (This is not clear from the BNF-like definition of the ANSI/ISO-standard alone.)

context-free syntax
DATA-DESCR-ENTRY+ → RECORD-DESCR-ENTRY

The level-numbers of format 1 must be either be between 1 and 50 (excluding 50), or 77. Level numbers for format 2 and 3 are 66 and 88 respectively. See also `Cobol-data.syn`.

context-free syntax
LEVEL-NUMBER ID-NAME-OR-FILLER REDEFINES EXTERNAL GLOBAL
OPT-PICTURE USAGE LEAD-OR-TRAIL OCCURS

SYNCHRONIZE JUSTIFY BLANK DD-VALUE "."	→ DATA-DESCR-ENTRY
"66" ID-NAME "RENAMES" ID-NAME DD-THROUGH "."	→ DATA-DESCR-ENTRY
"88" ID-NAME DD-VALUES ARG-88+ "."	→ DATA-DESCR-ENTRY
	→ DD-VALUE
"VALUE" OPT-IS CBL-LITERAL	→ DD-VALUE
"VALUE" OPT-IS	→ DD-VALUES
"VALUES" OPT-ARE	→ DD-VALUES
	→ DD-THROUGH
THROUGH ID	→ DD-THROUGH
CBL-LITERAL ARG-88-OPTION	→ ARG-88
	→ ARG-88-OPTION
THROUGH CBL-LITERAL	→ ARG-88-OPTION
	→ ID-NAME-OR-FILLER
ID-NAME	→ ID-NAME-OR-FILLER
"FILLER"	→ ID-NAME-OR-FILLER
	→ REDEFINES
"REDEFINES" ID-NAME	→ REDEFINES
	→ USAGE
USAGE-HEAD USAGE-BODY	→ USAGE
"BINARY"	→ USAGE-BODY
"COMPUTATIONAL"	→ USAGE-BODY
"COMP"	→ USAGE-BODY
"DISPLAY"	→ USAGE-BODY
"INDEX"	→ USAGE-BODY
"PACKED-DECIMAL"	→ USAGE-BODY
	→ OCCURS
"OCCURS" INTEGER OPT-TIMES OCCURS-A-OR-D* INDEXED	→ OCCURS
"OCCURS" INTEGER "TO" INTEGER OPT-TIMES "DEPENDING" OPT-ON ID-NAME OCCURS-A-OR-D* INDEXED	→ OCCURS
A-OR-D OPT-KEY OPT-IS ID-NAME+	→ OCCURS-A-OR-D
	→ SYNCHRONIZE
"SYNCHRONIZED" LEFT-OR-RIGHT	→ SYNCHRONIZE
"SYNC" LEFT-OR-RIGHT	→ SYNCHRONIZE
	→ LEFT-OR-RIGHT
"LEFT"	→ LEFT-OR-RIGHT
"RIGHT"	→ LEFT-OR-RIGHT

B.2.10 Report-description-entry

Report Description Entry and Report Group Description Entry [pages V-17, 18].

imports Cobol-data$^{B.2.17}$ Data-library$^{B.2.15}$
exports
sorts REPORT-DESCR-ENTRY REPORT-GROUP-DESCR-ENTRY FIRST-DETAIL LAST-DETAIL PAGE-FOOTING PAGE CODE CONTROLS CONTROLS-ARG HEADING LIMITS COLUMN RGD-USAGE REPORT-GD-TYPE NEXT-GROUP LINE-NUMBER SOURCE-OR-VALUE GROUP NEXT-PAGE NEXT-GROUP-PLUS LINE-NUMBER-PLUS DATA-NAME-OR-FINAL UPON-IDS SUM RESET STD-LEVEL-NUMBER

lexical syntax

```
"01" → STD-LEVEL-NUMBER
```

Report description entry:

context-free syntax

```
"RD" ID-NAME GLOBAL CODE CONTROLS PAGE "." → REPORT-DESCR-ENTRY

                                           → CODE
"CODE" CBL-LITERAL                         → CODE

                                           → CONTROLS
"CONTROL" OPT-IS CONTROLS-ARG              → CONTROLS
"CONTROLS" OPT-ARE CONTROLS-ARG            → CONTROLS

ID-NAME+                                   → CONTROLS-ARG
"FINAL" ID-NAME*                           → CONTROLS-ARG

"PAGE" LIMITS INTEGER LINE-OR-LINES HEADING
FIRST-DETAIL LAST-DETAIL PAGE-FOOTING      → PAGE

                                           → LIMITS
"LIMIT" OPT-IS                             → LIMITS
"LIMITS" OPT-ARE                           → LIMITS

                                           → FIRST-DETAIL
"FIRST" "DETAIL" INTEGER                   → FIRST-DETAIL

                                           → LAST-DETAIL
"LAST" "DETAIL" INTEGER                    → LAST-DETAIL

                                           → PAGE-FOOTING
"FOOTING" INTEGER                          → PAGE-FOOTING
```

Report group description entry:

context-free syntax

```
STD-LEVEL-NUMBER OPT-ID-NAME LINE-NUMBER
NEXT-GROUP "TYPE" OPT-IS REPORT-GD-TYPE
RGD-USAGE "."                                → REPORT-GROUP-DESCR-ENTRY
LEVEL-NUMBER OPT-ID-NAME LINE-NUMBER
RGD-USAGE "."                                → REPORT-GROUP-DESCR-ENTRY
LEVEL-NUMBER OPT-ID-NAME PICTURE RGD-USAGE
LEAD-OR-TRAIL JUSTIFY BLANK LINE-NUMBER
COLUMN SOURCE-OR-VALUE GROUP "."             → REPORT-GROUP-DESCR-ENTRY

                                             → LINE-NUMBER
"LINE" OPT-NUMBER OPT-IS LINE-NUMBER-PLUS    → LINE-NUMBER

INTEGER NEXT-PAGE                            → LINE-NUMBER-PLUS
"PLUS" INTEGER                               → LINE-NUMBER-PLUS

                                             → NEXT-PAGE
OPT-ON "NEXT" "PAGE"                         → NEXT-PAGE

                                             → NEXT-GROUP
"NEXT" "GROUP" OPT-IS NEXT-GROUP-PLUS        → NEXT-GROUP

INTEGER                                      → NEXT-GROUP-PLUS
"PLUS" INTEGER                               → NEXT-GROUP-PLUS
"NEXT" "PAGE"                                → NEXT-GROUP-PLUS
```

"REPORT" "HEADING"	→ REPORT-GD-TYPE
"RH"	→ REPORT-GD-TYPE
"PAGE" "HEADING"	→ REPORT-GD-TYPE
"PH"	→ REPORT-GD-TYPE
"CONTROL" "HEADING" DATA-NAME-OR-FINAL	→ REPORT-GD-TYPE
"CH" DATA-NAME-OR-FINAL	→ REPORT-GD-TYPE
"DETAIL"	→ REPORT-GD-TYPE
"DE"	→ REPORT-GD-TYPE
"CONTROL" "FOOTING" DATA-NAME-OR-FINAL	→ REPORT-GD-TYPE
"CF"	→ REPORT-GD-TYPE
"PAGE" "FOOTING"	→ REPORT-GD-TYPE
"PF"	→ REPORT-GD-TYPE
"REPORT" "FOOTING"	→ REPORT-GD-TYPE
"RF"	→ REPORT-GD-TYPE
ID-NAME	→ DATA-NAME-OR-FINAL
"FINAL"	→ DATA-NAME-OR-FINAL
	→ RGD-USAGE
USAGE-HEAD "DISPLAY"	→ RGD-USAGE
	→ COLUMN
"COLUMN" OPT-NUMBER OPT-IS INTEGER	→ COLUMN
"SOURCE" OPT-IS ID	→ SOURCE-OR-VALUE
"VALUE" OPT-IS CBL-LITERAL	→ SOURCE-OR-VALUE
SUM+ RESET	→ SOURCE-OR-VALUE
"SUM" ID+ UPON-IDS	→ SUM
	→ UPON-IDS
"UPON" ID-NAME+	→ UPON-IDS
	→ RESET
"RESET" OPT-ON DATA-NAME-OR-FINAL	→ RESET
	→ GROUP
"GROUP" OPT-INDICATE	→ GROUP

B.2.11 Common-verbs

Common verbs [section V]. See also Arithmetic-verbs, Program-control-verbs, Input-output-verbs; used in Procedure Division.

ALTER, CONTINUE, COPY, DELETE, DISABLE, ENABLE, EVALUATE, GENERATE, IF, INITIALIZE, INITIATE, INSPECT, MOVE, PURGE, RECEIVE, SEARCH, SEND, SET, STRING, SUPPRESS, TERMINATE, UNSTRING, USE

imports Cobol-data$^{B.2.17}$ Verbs-library$^{B.2.16}$

exports
sorts PROCEED ALTER-LOOP END-DELETE EVAL
EVAL-CONDITION EVAL-SUBCOND WHEN-COND OTHER
THROUGH-COND END-EVALUATE EVAL-ARGUMENT
IF-SENTENCE IF-ELSE INIT-REPLACING REPL-BODY
REPL-OPTION TALLYING TALLY REPLACING REP-LOOP
A-L-F CHAR-OPTION ALL-OR-LEADING TAL-LOOP
TAL-SUBLOOP MESS-OR-SEGM NO-DATA WITH-DATA
END-RECEIVE VARYING WHEN END-SEARCH VARY
EVAL-WHEN SEARCH-AND SEARCH-COND STAT-OR-NEXT IS
SEND-WITH IO-ADV SEND-OPTION UP-OR-DOWN SET-LOOP
SET-ON-OFF END-UNSTRING UNSTRING-TALLY COUNT
UNSTRING-LOOP DELIM-LOOP STR-DELIMITER DELIMITED
EXCEPT-OR-ERROR FILE-TYPE DEBUG-OPTION
REFERENCES POINTER END-STRING SIZE
EVAL-ARGUMENT-2 ALSO REPLACE SEND-REPLACING
STRING-LOOP IF-STATEMENT-NO-ENDIF IF-ELSE2 BOTH-IF
IF-ELSE3 IF-STATEMENT-NO-ELSE D-E-ABLE-KEY
COPY-STATEMENT USE-STATEMENT ID-OR-ID-NAME-OR-INT
REPLACE-OPTION MOVE-STATEMENT

The `COPY` statement is an exception to the general use of COBOL statements, as is the `USE` statement. The `COPY` statement may be used whereever a character string or a separator is required in a COBOL program [page XII-2]. Our attempt has been to specify it as layout, but that did not work properly. As indicated in the paper we advise to have a preprocessor handle this statement. The `COPY` statement indicates the incorporation of text into the program at the position of the `COPY` statement.
Pseudo-text should be allowed inside a `COPY` statement as well. We have not implemented this. Here is the specification the ANSI standard gives for the `COPY` statement (this result in an 'error in regular expression' warning by the Meta-environment, that is why it is listed here as commentary):

% % "COPY" ID-NAME IN-OR-OF ID-NAME
% % "REPLACING" ID-OR-ID-NAME-OR-LIT "BY" ID-OR-ID-NAME-OR-LIT -> LAYOUT
% % "COPY" ID-NAME IN-OR-OF ID-NAME -> LAYOUT
% % "COPY" ID-NAME "REPLACING" ID-OR-ID-NAME-OR-LIT "BY" ID-OR-ID-NAME-OR-LIT -> LAYOUT
% % "COPY" ID-NAME -> LAYOUT

context-free syntax
"ALTER" ALTER-LOOP+ → STATEMENT

ID-NAME "TO" PROCEED ID-NAME → ALTER-LOOP

→ PROCEED
"PROCEED" "TO" → PROCEED

context-free syntax
"CONTINUE" → STATEMENT

context-free syntax
"DELETE" ID-NAME OPT-RECORD INVALID NOT-INVALID
END-DELETE → STATEMENT

→ END-DELETE
"END-DELETE" → END-DELETE

context-free syntax
"DISABLE" "INPUT" OPT-TERMINAL ID-NAME D-E-ABLE-KEY → STATEMENT
"DISABLE" "I-O" "TERMINAL" ID-NAME D-E-ABLE-KEY → STATEMENT
"DISABLE" "OUTPUT" ID-NAME D-E-ABLE-KEY → STATEMENT

context-free syntax

"ENABLE" "INPUT" OPT-TERMINAL ID-NAME D-E-ABLE-KEY	→ STATEMENT
"ENABLE" "I-O" "TERMINAL" ID-NAME D-E-ABLE-KEY	→ STATEMENT
"ENABLE" "OUTPUT" ID-NAME D-E-ABLE-KEY	→ STATEMENT
	→ D-E-ABLE-KEY
OPT-WITH "KEY" ID-OR-LIT	→ D-E-ABLE-KEY

context-free syntax

"ENTER" ID-NAME	→ STATEMENT
"ENTER" ID-NAME ID-NAME	→ STATEMENT

context-free syntax

"EVALUATE" EVAL-ARGUMENT EVAL-ARGUMENT-2*	
EVAL-WHEN+ OTHER END-EVALUATE	→ STATEMENT
ID-OR-LIT	→ EVAL-ARGUMENT
ARITH-EXP	→ EVAL-ARGUMENT
"TRUE"	→ EVAL-ARGUMENT
"FALSE"	→ EVAL-ARGUMENT
"ALSO" EVAL-ARGUMENT	→ EVAL-ARGUMENT-2
EVAL-CONDITION+ IMPERATIVE-STATEMENT	→ EVAL-WHEN
"WHEN" EVAL-SUBCOND ALSO*	→ EVAL-CONDITION
"ALSO" EVAL-SUBCOND	→ ALSO
"ANY"	→ EVAL-SUBCOND
CONDITION	→ EVAL-SUBCOND
"TRUE"	→ EVAL-SUBCOND
"FALSE"	→ EVAL-SUBCOND
OPT-NOT WHEN-COND THROUGH-COND	→ EVAL-SUBCOND
ID-OR-LIT	→ WHEN-COND
ARITH-EXP	→ WHEN-COND
	→ THROUGH-COND
THROUGH WHEN-COND	→ THROUGH-COND
	→ OTHER
"WHEN" "OTHER" IMPERATIVE-STATEMENT	→ OTHER
	→ END-EVALUATE
"END-EVALUATE"	→ END-EVALUATE

context-free syntax

"GENERATE" ID-NAME	→ STATEMENT

context-free syntax

"IF" CONDITION OPT-THEN IF-SENTENCE IF-ELSE	→ STATEMENT
"IF" CONDITION OPT-THEN IF-SENTENCE IF-ELSE2	→ IF-STATEMENT-NO-ENDIF
"IF" CONDITION OPT-THEN IF-SENTENCE	→ IF-STATEMENT-NO-ELSE
STATEMENT	→ BOTH-IF
IF-STATEMENT-NO-ENDIF	→ BOTH-IF
IF-STATEMENT-NO-ELSE	→ BOTH-IF
STATEMENT+	→ IF-SENTENCE
"NEXT" "SENTENCE"	→ IF-SENTENCE
"ELSE" BOTH-IF+	→ IF-ELSE2
"ELSE" BOTH-IF+ "END-IF"	→ IF-ELSE
"ELSE" "NEXT" "SENTENCE"	→ IF-ELSE3
"END-IF"	→ IF-ELSE

IF-ELSE3 → IF-ELSE

context-free syntax

“INITIALIZE” ID+ INIT-REPLACING → STATEMENT

→ INIT-REPLACING
“REPLACING” REPL-BODY+ → INIT-REPLACING

REPL-OPTION OPT-DATA “BY” ID-OR-LIT → REPL-BODY

“ALPHABETIC” → REPL-OPTION
“ALPHANUMERIC” → REPL-OPTION
“NUMERIC” → REPL-OPTION
“ALPHANUMERIC-EDITED” → REPL-OPTION
“NUMERIC-EDITED” → REPL-OPTION

context-free syntax

“INITIATE” ID-NAME+ → STATEMENT

context-free syntax

“INSPECT” ID TALLYING → STATEMENT
“INSPECT” ID REPLACING → STATEMENT
“INSPECT” ID TALLYING REPLACING → STATEMENT
“INSPECT” ID “CONVERTING” ID-OR-LIT “TO”
ID-OR-LIT CHAR-OPTION∗ → STATEMENT

“TALLYING” TALLY+ → TALLYING

ID “FOR” TAL-LOOP+ → TALLY

“REPLACING” REPLACE+ → REPLACING

“CHARACTERS” CHAR-OPTION∗ → TAL-LOOP
ALL-OR-LEADING TAL-SUBLOOP+ → TAL-LOOP

“ALL” → ALL-OR-LEADING
“LEADING” → ALL-OR-LEADING

ID-OR-LIT CHAR-OPTION∗ → TAL-SUBLOOP

BEFORE-OR-AFTER OPT-INITIAL ID-OR-LIT → CHAR-OPTION

“CHARACTERS” “BY” ID-OR-LIT CHAR-OPTION∗ → REPLACE
A-L-F REP-LOOP+ → REPLACE

ALL-OR-LEADING → A-L-F
“FIRST” → A-L-F

ID-OR-LIT “BY” ID-OR-LIT CHAR-OPTION∗ → REP-LOOP

context-free syntax

“MOVE” ID-OR-LIT “TO” ID+ → MOVE-STATEMENT
“MOVE” CORR ID “TO” ID → MOVE-STATEMENT
MOVE-STATEMENT → STATEMENT

context-free syntax

“PURGE” ID-NAME → STATEMENT

context-free syntax

“RECEIVE” ID-NAME MESS-OR-SEGM “INTO” ID NO-DATA
WITH-DATA END-RECEIVE → STATEMENT

“MESSAGE” → MESS-OR-SEGM
“SEGMENT” → MESS-OR-SEGM

→ NO-DATA
“NO” “DATA” IMPERATIVE-STATEMENT → NO-DATA

→ WITH-DATA

OPT-WITH "DATA" IMPERATIVE-STATEMENT	→ WITH-DATA

	→ END-RECEIVE
"END-RECEIVE"	→ END-RECEIVE

context-free syntax

"REPLACE" REPLACE-OPTION+	→ STATEMENT

PSEUDO "BY" PSEUDO	→ REPLACE-OPTION

context-free syntax

"SEARCH" ID VARY YES-AT-END WHEN+ END-SEARCH	→ STATEMENT
"SEARCH" "ALL" ID YES-AT-END "WHEN" SEARCH-COND SEARCH-AND* STAT-OR-NEXT END-SEARCH	→ STATEMENT

"WHEN" CONDITION STAT-OR-NEXT	→ WHEN

"AND" SEARCH-COND	→ SEARCH-AND

ID	→ SEARCH-COND
ID-NAME IS WHEN-COND	→ SEARCH-COND

OPT-IS "EQUAL" OPT-TO	→ IS
OPT-IS ALG-EQUAL	→ IS

IMPERATIVE-STATEMENT	→ STAT-OR-NEXT
"NEXT" "SENTENCE"	→ STAT-OR-NEXT

	→ END-SEARCH
"END-SEARCH"	→ END-SEARCH

	→ VARY
"VARYING" ID-NAME	→ VARY

context-free syntax

"SEND" ID-NAME "FROM" ID	→ STATEMENT
"SEND" ID-NAME FROM SEND-WITH IO-ADV SEND-REPLACING	→ STATEMENT

OPT-WITH ID	→ SEND-WITH
OPT-WITH "ESI"	→ SEND-WITH
OPT-WITH "EMI"	→ SEND-WITH
OPT-WITH "EGI"	→ SEND-WITH

	→ IO-ADV
BEFORE-OR-AFTER OPT-ADVANCING SEND-OPTION	→ IO-ADV

ID-OR-INT LINE-OR-LINES	→ SEND-OPTION
ID-NAME	→ SEND-OPTION
"PAGE"	→ SEND-OPTION

context-free syntax

"SET" ID-OR-ID-NAME+ "TO" ID-OR-ID-NAME-OR-INT	→ STATEMENT
"SET" ID-NAME+ UP-OR-DOWN ID-OR-INT	→ STATEMENT
"SET" SET-LOOP+	→ STATEMENT
"SET" ID+ "TO" "TRUE"	→ STATEMENT

ID-OR-ID-NAME	→ ID-OR-ID-NAME-OR-INT
INTEGER	→ ID-OR-ID-NAME-OR-INT

"UP" "BY"	→ UP-OR-DOWN
"DOWN" "BY"	→ UP-OR-DOWN

ID-NAME+ "TO" SET-ON-OFF	→ SET-LOOP

"ON"	→ SET-ON-OFF
"OFF"	→ SET-ON-OFF

context-free syntax

"STRING" STRING-LOOP+ "INTO" ID POINTER OVERFLOW
END-STRING → STATEMENT

ID-OR-LIT+ "DELIMITED" OPT-BY SIZE → STRING-LOOP

→ POINTER
OPT-WITH "POINTER" ID → POINTER

ID-OR-LIT → SIZE
"SIZE" → SIZE

→ END-STRING
"END-STRING" → END-STRING

context-free syntax
"SUPPRESS" OPT-PRINTING → STATEMENT

context-free syntax
"TERMINATE" ID-NAME+ → STATEMENT

context-free syntax
"UNSTRING" ID DELIMITED "INTO" UNSTRING-LOOP+
POINTER UNSTRING-TALLY OVERFLOW
END-UNSTRING → STATEMENT

→ DELIMITED
"DELIMITED" OPT-BY OPT-ALL ID-OR-LIT DELIM-LOOP* → DELIMITED

"OR" OPT-ALL ID-OR-LIT → DELIM-LOOP

→ STR-DELIMITER
"DELIMITER" OPT-IN ID → STR-DELIMITER

ID STR-DELIMITER COUNT → UNSTRING-LOOP

→ COUNT
"COUNT" OPT-IN ID → COUNT

→ UNSTRING-TALLY
"TALLYING" OPT-IN ID → UNSTRING-TALLY

→ END-UNSTRING
"END-UNSTRING" → END-UNSTRING

The `USE` statement may only be used in the header of the declaratives section of the `PROCEDURE DIVISION` [page VII-50], therefore it is not a '`STATEMENT`' but a '`USE-STATEMENT`'. Please note: the first `USE` specifies the first two of the definition [page V-32].

context-free syntax
"USE" OPT-GLOBAL "AFTER" OPT-STANDARD
EXCEPT-OR-ERROR "PROCEDURE" OPT-ON FILE-TYPE → USE-STATEMENT
"USE" OPT-GLOBAL "BEFORE" "REPORTING" ID → USE-STATEMENT
"USE" OPT-FOR "DEBUGGING" OPT-ON DEBUG-OPTION+ → USE-STATEMENT

"EXCEPTION" → EXCEPT-OR-ERROR
"ERROR" → EXCEPT-OR-ERROR

ID-NAME+ → FILE-TYPE
"INPUT" → FILE-TYPE
"OUTPUT" → FILE-TYPE
"I-O" → FILE-TYPE
"EXTEND" → FILE-TYPE

ID-NAME → DEBUG-OPTION
REFERENCES ID → DEBUG-OPTION

```
   "ALL" "PROCEDURES"                                  → DEBUG-OPTION

                                                       → REFERENCES
   "ALL" OPT-REFERENCES OPT-OF                         → REFERENCES
```

B.2.12 Arithmetic-verbs

Arithmetic Verbs [pages V-20, V-21, V-22, V-31]. See also Co-verbs, Pc-verbs, Io-verbs; used in Procedure Division.

ADD, COMPUTE, DIVIDE, MULTIPLY, SUBTRACT

imports Cobol-data$^{B.2.17}$ Verbs-library$^{B.2.16}$
exports
  sorts ROUNDED END-ADD END-SUBTRACT END-MULTIPLY
        END-DIVIDE END-COMPUTE ADD-STATEMENT
        COMPUTE-STATEMENT SUBTRACT-STATEMENT

```
 context-free syntax
   "ADD" ID-OR-LIT+ "TO" ROUNDED+ ERROR END-ADD        → ADD-STATEMENT
   "ADD" ID-OR-LIT+ OPT-TO ID-OR-LIT "GIVING" ROUNDED+
   ERROR END-ADD                                       → ADD-STATEMENT
   "ADD" CORR ID "TO" ID OPT-ROUNDED ERROR END-ADD     → ADD-STATEMENT

   ADD-STATEMENT                                       → STATEMENT

   ID OPT-ROUNDED                                      → ROUNDED

                                                       → END-ADD
   "END-ADD"                                           → END-ADD
 context-free syntax
   "COMPUTE" ROUNDED+ ALG-EQUAL ARITH-EXP ERROR
   END-COMPUTE                                         → COMPUTE-STATEMENT

   COMPUTE-STATEMENT                                   → STATEMENT

                                                       → END-COMPUTE
   "END-COMPUTE"                                       → END-COMPUTE
 context-free syntax
   "DIVIDE" ID-OR-LIT "INTO" ROUNDED+ ERROR END-DIVIDE → STATEMENT
   "DIVIDE" ID-OR-LIT "INTO" ID-OR-LIT "GIVING"
   ROUNDED+ ERROR END-DIVIDE                           → STATEMENT
   "DIVIDE" ID-OR-LIT "BY" ID-OR-LIT "GIVING"
   ROUNDED+ ERROR END-DIVIDE                           → STATEMENT
   "DIVIDE" ID-OR-LIT "BY" ID-OR-LIT "GIVING" ROUNDED
   "REMAINDER" ID ERROR END-DIVIDE                     → STATEMENT

                                                       → END-DIVIDE
   "END-DIVIDE"                                        → END-DIVIDE
 context-free syntax
   "MULTIPLY" ID-OR-LIT "BY" ROUNDED+ ERROR
   END-MULTIPLY                                        → STATEMENT

   "MULTIPLY" ID-OR-LIT "BY" ID-OR-LIT "GIVING"
   ROUNDED+ ERROR END-MULTIPLY                         → STATEMENT

                                                       → END-MULTIPLY
   "END-MULTIPLY"                                      → END-MULTIPLY
 context-free syntax
   "SUBTRACT" ID-OR-LIT+ "FROM" ROUNDED+ ERROR
   END-SUBTRACT                                        → SUBTRACT-STATEMENT
   "SUBTRACT" ID-OR-LIT+ "FROM" ID-OR-LIT "GIVING"
```

```
ROUNDED+ ERROR END-SUBTRACT                        -> SUBTRACT-STATEMENT
"SUBTRACT" CORR ID-OR-LIT+ "FROM" ID OPT-ROUNDED
ERROR END-SUBTRACT                                 -> SUBTRACT-STATEMENT

SUBTRACT-STATEMENT                                 -> STATEMENT

                                                   -> END-SUBTRACT
"END-SUBTRACT"                                     -> END-SUBTRACT
```

B.2.13 Input-output-verbs

Input/Output Verbs [section V]. See also Co-verbs, Pc-verbs, Ar-verbs; used in Procedure Division.

ACCEPT, CLOSE, DISPLAY, MERGE, OPEN, READ, RELEASE, RETURN, REWRITE, SORT, START, WRITE

imports Cobol-data$^{B.2.17}$ Verbs-library$^{B.2.16}$
exports
sorts SUB-CLOSE CLOSE-OPTION REEL-OR-UNIT NOREWIND-OR-LOCK LOCK LOCK-OPTION FILE-TYPE-1 FILE-TYPE-2 FILE-TYPE-3 NO-REWIND READ-TAIL END-READ END-REWRITE DAY-TIME UPON-ID ADVANCING END-OF-PAGE NOT-END-OF-PAGE END-WRITE DUPLICATES RESULT2 SORT-OPTION RESULT COLLATING ASC-DESC INTO END-RETURN ACCEPT-FROM START-OPTION END-START IS-OPTIONS
context-free syntax

```
"ACCEPT" ID ACCEPT-FROM                   -> STATEMENT
"ACCEPT" ID "FROM" DAY-TIME               -> STATEMENT
"ACCEPT" ID-NAME OPT-MESSAGE "COUNT"      -> STATEMENT
```

The `ACCEPT FROM` is different from the standard, because here only the identifier name is allowed instead of the identifier itself.

context-free syntax

```
                    -> ACCEPT-FROM
"FROM" ID-NAME      -> ACCEPT-FROM

"DATE"              -> DAY-TIME
"DAY"               -> DAY-TIME
"DAY-OF-WEEK"       -> DAY-TIME
"TIME"              -> DAY-TIME
```

context-free syntax

```
"CLOSE" SUB-CLOSE+                    -> STATEMENT
"CLOSE" LOCK+                         -> STATEMENT

ID-NAME CLOSE-OPTION                  -> SUB-CLOSE

                                      -> CLOSE-OPTION
REEL-OR-UNIT OPT-FOR "REMOVAL"        -> CLOSE-OPTION
REEL-OR-UNIT                          -> CLOSE-OPTION
OPT-WITH NOREWIND-OR-LOCK             -> CLOSE-OPTION

"REEL"                                -> REEL-OR-UNIT
"UNIT"                                -> REEL-OR-UNIT

"NO" "REWIND"                         -> NOREWIND-OR-LOCK
"LOCK"                                -> NOREWIND-OR-LOCK

ID-NAME LOCK-OPTION                   -> LOCK
```

OPT-WITH “LOCK” → LOCK-OPTION

context-free syntax

“DISPLAY” ID-OR-LIT+ UPON-ID ADVANCING → STATEMENT

→ UPON-ID
“UPON” ID-NAME → UPON-ID

→ ADVANCING
OPT-WITH “NO” “ADVANCING” → ADVANCING

context-free syntax

“MERGE” ID-NAME ASC-DESC+ COLLATING “USING”
ID-NAME ID-NAME+ RESULT → STATEMENT

OPT-ON A-OR-D OPT-KEY ID-NAME+ → ASC-DESC

→ COLLATING
OPT-COLLATING “SEQUENCE” OPT-IS ID-NAME → COLLATING

“OUTPUT” “PROCEDURE” OPT-IS ID-NAME THRU-PROCEDURE → RESULT
“GIVING” ID-NAME+ → RESULT

Again, in order to disambiguate the specification one of the following three file types should be selected. The same goes for most input-output COBOL verbs.

context-free syntax

“OPEN” FILE-TYPE-1+ → STATEMENT
“OPEN” FILE-TYPE-2+ → STATEMENT
“OPEN” FILE-TYPE-3+ → STATEMENT

context-free syntax

“INPUT” ID-NAME NO-REWIND → FILE-TYPE-1
“OUTPUT” ID-NAME NO-REWIND → FILE-TYPE-1
“I-O” ID-NAME+ → FILE-TYPE-1
“EXTEND” ID-NAME+ → FILE-TYPE-1

“INPUT” ID-NAME → FILE-TYPE-2
“OUTPUT” ID-NAME → FILE-TYPE-2
“I-O” ID-NAME+ → FILE-TYPE-2
“EXTEND” ID-NAME+ → FILE-TYPE-2

“OUTPUT” ID-NAME NO-REWIND → FILE-TYPE-3
“EXTEND” ID-NAME+ → FILE-TYPE-3

→ NO-REWIND
OPT-WITH “NO” “REWIND” → NO-REWIND

context-free syntax

“READ” ID-NAME OPT-NEXT OPT-RECORD INTO AT-END
END-READ → STATEMENT
“READ” ID-NAME OPT-RECORD INTO READ-TAIL → STATEMENT
“READ” ID-NAME OPT-RECORD INTO KEY-OPTION READ-TAIL → STATEMENT

INVALID NOT-INVALID END-READ → READ-TAIL

→ END-READ
“END-READ” → END-READ

context-free syntax

“RELEASE” ID-NAME FROM → STATEMENT

context-free syntax

“RETURN” ID-NAME OPT-RECORD INTO OPT-AT “END” STATEMENT
NO-AT-END END-RETURN → STATEMENT

→ END-RETURN
“END-RETURN” → END-RETURN

```
                                                    → INTO
  "INTO" ID                                         → INTO
context-free syntax
  "REWRITE" ID FROM INVALID NOT-INVALID END-REWRITE → STATEMENT

                                                    → END-REWRITE
  "END-REWRITE"                                     → END-REWRITE
context-free syntax
  "SORT" ID-NAME SORT-OPTION+ DUPLICATES COLLATING RESULT2
  RESULT                                            → STATEMENT

                                                    → DUPLICATES
  OPT-WITH "DUPLICATES" OPT-IN OPT-ORDER            → DUPLICATES

  "INPUT" "PROCEDURE" OPT-IS ID-NAME THRU-PROCEDURE → RESULT2
  "USING" ID-NAME+                                  → RESULT2

  OPT-ON A-OR-D OPT-KEY ID-NAME+                    → SORT-OPTION
context-free syntax
  "START" ID-NAME START-OPTION INVALID NOT-INVALID
  END-START                                         → STATEMENT

                                                    → START-OPTION
  "KEY" OPT-IS IS-OPTIONS ID-NAME                   → START-OPTION

                                                    → END-START
  "END-START"                                       → END-START

  "EQUAL" OPT-TO                                    → IS-OPTIONS
  RELATION3                                         → IS-OPTIONS
  "GREATER" OPT-THAN                                → IS-OPTIONS
  "NOT␣LESS" OPT-THAN                               → IS-OPTIONS
  "NOT␣<"                                           → IS-OPTIONS
  "GREATER" OPT-THAN OPT-OR "EQUAL" OPT-TO          → IS-OPTIONS
  ALG-LEQ                                           → IS-OPTIONS
context-free syntax
  "WRITE" ID-NAME FROM IO-ADV END-OF-PAGE NOT-END-OF-PAGE
  END-WRITE                                         → STATEMENT
  "WRITE" ID-NAME FROM INVALID NOT-INVALID END-WRITE → STATEMENT

                                                    → END-OF-PAGE
  OPT-AT "EOP" IMPERATIVE-STATEMENT                 → END-OF-PAGE

                                                    → NOT-END-OF-PAGE
  "NOT" END-OF-PAGE                                 → NOT-END-OF-PAGE

                                                    → END-WRITE
  "END-WRITE"                                       → END-WRITE
```

B.2.14 Program-control-verbs

Program Control Verbs [section V]. See also Ar-verbs, Co-verbs, Io-verbs; used in Procedure Division.
Note that the import of Verbs-Library is not required yet for the constructs listed in this module right now. For consistency and perhaps future use we have included the import here.

`CALL, CANCEL, EXIT, EXIT-PROGRAM, GO-TO, PERFORM, STOP`

imports Cobol-data B.2.17 Verbs-library B.2.16

exports
sorts CALL-HEAD USING REFERENCE CALL-OVERFLOW
END-CALL ON-EXCEPTION NOT-ON-EXCEPTION
PERFORM-VAR PERFORM-AFTER PERFORM-WITH
PERFORM-OPTION PERFORM-TAIL PROC-OPTION

The `CALL` statement from page [page X-27] is ambiguous. This is the way we dealt with it, the `OVERFLOW` part is required, the `EXCEPTION`-parts are optional:

```
context-free syntax
  "CALL" CALL-HEAD CALL-OVERFLOW END-CALL          → STATEMENT
  "CALL" CALL-HEAD ON-EXCEPTION NOT-ON-EXCEPTION
  END-CALL                                         → STATEMENT

  ID-OR-LIT USING*                                 → CALL-HEAD

  "USING" REFERENCE ID+                            → USING
  "USING" OPT-BY "CONTENT" ID+                     → USING

                                                   → REFERENCE
  OPT-BY "REFERENCE"                               → REFERENCE

  OPT-ON "OVERFLOW" IMPERATIVE-STATEMENT           → CALL-OVERFLOW

                                                   → END-CALL
  "END-CALL"                                       → END-CALL

                                                   → ON-EXCEPTION
  OPT-ON "EXCEPTION" IMPERATIVE-STATEMENT          → ON-EXCEPTION

                                                   → NOT-ON-EXCEPTION
  "NOT" ON-EXCEPTION                               → NOT-ON-EXCEPTION
context-free syntax
  "CANCEL" ID-OR-LIT+ → STATEMENT
context-free syntax
  "EXIT" → STATEMENT
context-free syntax
  "EXIT" "PROGRAM" → STATEMENT
context-free syntax
  "GO" OPT-TO PROC-OPTION                          → STATEMENT
  "GO" OPT-TO ID-NAME+ "DEPENDING" OPT-ON ID       → STATEMENT

                                                   → PROC-OPTION
  ID-NAME                                          → PROC-OPTION
```

In the `PERFORM` statement a choice *has* to be made between the procedure name (in `PERFORM-OPTION`) and the imperative statement (in `PERFORM-TAIL`), see (1) [page VI-110]:

```
context-free syntax
  "PERFORM" PERFORM-OPTION                                    → STATEMENT
  "PERFORM" PERFORM-TAIL                                      → STATEMENT

  "PERFORM" PERFORM-OPTION ID-OR-INT "TIMES"                  → STATEMENT
  "PERFORM" ID-OR-INT "TIMES" PERFORM-TAIL                    → STATEMENT

  "PERFORM" PERFORM-OPTION PERFORM-WITH "UNTIL"
  CONDITION                                                   → STATEMENT
  "PERFORM" PERFORM-WITH "UNTIL" CONDITION
  PERFORM-TAIL                                                → STATEMENT

  "PERFORM" PERFORM-OPTION PERFORM-WITH PERFORM-VAR           → STATEMENT
```

"PERFORM" PERFORM-WITH PERFORM-VAR PERFORM-TAIL	→ STATEMENT
"VARYING" ID-OR-ID-NAME "FROM" ID-OR-ID-NAME-OR-LIT "BY" ID-OR-LIT "UNTIL" CONDITION PERFORM-AFTER*	→ PERFORM-VAR
"AFTER" ID-OR-LIT "FROM" ID-OR-ID-NAME-OR-LIT "BY" ID-OR-LIT "UNTIL" CONDITION	→ PERFORM-AFTER
	→ PERFORM-WITH
OPT-WITH "TEST" BEFORE-OR-AFTER	→ PERFORM-WITH
ID-NAME THRU-PROCEDURE	→ PERFORM-OPTION

Although in words the standard says that the 'statement' here may consist of many imperative statements, this does not follow from their formal definition. We choose to follow their verbal description since from our test-programs that appears to be the correct one.

context-free syntax
IMPERATIVE-STATEMENT "END-PERFORM" → PERFORM-TAIL
context-free syntax
"STOP" "RUN" → STATEMENT
"STOP" CBL-LITERAL → STATEMENT

B.2.15 Data-library

Data Division library [section V]. The description entries of the data division show a substantial amount of overlap. These sorts are placed in this module.

imports Cobol-data$^{B.2.17}$
exports
sorts LEAD-OR-TRAIL LEAD-OR-TRAIL-HEAD
LEAD-OR-TRAIL-TAIL JUSTIFY BLANK USAGE-HEAD
OPT-PICTURE PICTURE GLOBAL EXTERNAL PIC-STRING
ALPHA-PIC-STRING NUM-PIC-STRING
ALPHANUM-PIC-STRING ALPHANUM-ED-PIC-STRING
NUM-ED-PIC-STRING ALPHA-PIC-ELT NUM-PIC-ELT
ALPHANUM-PIC-ELT ALPHANUM-ED-PIC-ELT1
ALPHANUM-ED-PIC-ELT ALPHANUM-ED-PIC-ELT2
NUM-ED-ELT NUM-ED-ELT2 PIC-A PIC-COMMA PIC-X PIC-9
PIC-Z PIC-STAR PIC-B PIC-SLANT PIC-0 PIC-PLUS PIC-MINUS
PIC-CS NUMBER-OF-OCCUR PIC-P NUM-ED-ELT3
NUM-ED-ELT4 NUM-ED-ELT5 CONTAINS-ELT2-OR-3
CONTAINS-NO-ELT2-OR-3 PIC-LIT

An unsigned integer in parentheses following any of the following characters indicates the number of occurrences of that character (this can occur within a larger `PICTURE` as well) [page VI-30].

lexical syntax

"(" NON-ZERO-INTEGER ")"	→ NUMBER-OF-OCCUR
"0"	→ PIC-0
"0" NUMBER-OF-OCCUR	→ PIC-0
"A"	→ PIC-A
"A" NUMBER-OF-OCCUR	→ PIC-A
","	→ PIC-COMMA
"," NUMBER-OF-OCCUR	→ PIC-COMMA
"X"	→ PIC-X
"X" NUMBER-OF-OCCUR	→ PIC-X
"9"	→ PIC-9

```
"9"NUMBER-OF-OCCUR                    → PIC-9
"P"                                   → PIC-P
"P"NUMBER-OF-OCCUR                    → PIC-P
"Z"                                   → PIC-Z
"Z"NUMBER-OF-OCCUR                    → PIC-Z
"*"                                   → PIC-STAR
"*"NUMBER-OF-OCCUR                    → PIC-STAR
"B"                                   → PIC-B
"B"NUMBER-OF-OCCUR                    → PIC-B
"/"                                   → PIC-SLANT
"/"NUMBER-OF-OCCUR                    → PIC-SLANT
"+"                                   → PIC-PLUS
"+"NUMBER-OF-OCCUR                    → PIC-PLUS
"−"                                   → PIC-MINUS
"−"NUMBER-OF-OCCUR                    → PIC-MINUS
CURRENCY-SYMBOL                       → PIC-CS
CURRENCY-SYMBOL NUMBER-OF-OCCUR       → PIC-CS

PIC-P                                 → NUM-ED-ELT
"V"                                   → NUM-ED-ELT
PIC-9                                 → NUM-ED-ELT
PIC-0                                 → NUM-ED-ELT2
PIC-B                                 → NUM-ED-ELT2
PIC-SLANT                             → NUM-ED-ELT2
PIC-Z                                 → NUM-ED-ELT2
"."                                   → NUM-ED-ELT3
PIC-STAR                              → NUM-ED-ELT2
PIC-COMMA                             → NUM-ED-ELT2
PIC-MINUS                             → NUM-ED-ELT2
PIC-PLUS                              → NUM-ED-ELT2
"CR"                                  → NUM-ED-ELT2
"DB"                                  → NUM-ED-ELT2
PIC-CS                                → NUM-ED-ELT2

NUM-ED-ELT                            → NUM-ED-ELT4
NUM-ED-ELT2                           → NUM-ED-ELT4
NUM-ED-ELT4                           → NUM-ED-ELT5
NUM-ED-ELT3                           → NUM-ED-ELT5
```

The period is only allowed once, except when in the SPECIAL NAMES section it is declared that DECIMAL-POINT IS COMMA. In that case the rules for comma and period are swapped [page VI-32].
In our current specification it is not allowed to use the period at the end of a `PICTURE`; this restriction is not standard [page VI-34]. A restriction we have not implemented is that there can be only one comma or period inside a `PICTURE` string. As noted in the paper, many of these restrictions exist but in the case of the `PICTURE` string we have taken a more pragmatic approach, assuming that the COBOL source code is correct.

```
lexical syntax
  NUM-ED-ELT5*NUM-ED-ELT2 NUM-ED-ELT5* → CONTAINS-ELT2-OR-3
  NUM-ED-ELT5*NUM-ED-ELT3 NUM-ED-ELT5* → CONTAINS-ELT2-OR-3

  NUM-ED-ELT+                          → CONTAINS-NO-ELT2-OR-3

  CONTAINS-ELT2-OR-3+NUM-ED-ELT4       → NUM-ED-PIC-STRING
  CONTAINS-NO-ELT2-OR-3+NUM-ED-ELT2    → NUM-ED-PIC-STRING
  NUM-ED-ELT4                          → NUM-ED-PIC-STRING

  PIC-A                                → ALPHANUM-PIC-ELT
  PIC-X                                → ALPHANUM-PIC-ELT
```

PIC-9 → ALPHANUM-PIC-ELT

lexical syntax

ALPHANUM-PIC-ELT∗PIC-X ALPHANUM-PIC-ELT∗ → ALPHANUM-PIC-STRING

Note that perhaps the other `PICTURE` definitions (listed below) should be defined lexically as well.

context-free syntax

→ LEAD-OR-TRAIL
LEAD-OR-TRAIL-HEAD "LEADING" LEAD-OR-TRAIL-TAIL → LEAD-OR-TRAIL
LEAD-OR-TRAIL-HEAD "TRAILING" LEAD-OR-TRAIL-TAIL → LEAD-OR-TRAIL

→ LEAD-OR-TRAIL-HEAD
"SIGN" OPT-IS → LEAD-OR-TRAIL-HEAD

→ LEAD-OR-TRAIL-TAIL
"SEPARATE" OPT-CHARACTER → LEAD-OR-TRAIL-TAIL

→ JUSTIFY
"JUSTIFIED" OPT-RIGHT → JUSTIFY
"JUST" OPT-RIGHT → JUSTIFY

→ BLANK
"BLANK" OPT-WHEN "ZERO" → BLANK

→ USAGE-HEAD
"USAGE" OPT-IS → USAGE-HEAD

→ OPT-PICTURE
PICTURE → OPT-PICTURE

"PIC" → PIC-LIT
"PICTURE" → PIC-LIT

PIC-LIT OPT-IS PIC-STRING → PICTURE

ALPHA-PIC-STRING → PIC-STRING
NUM-PIC-STRING → PIC-STRING
ALPHANUM-PIC-STRING → PIC-STRING
ALPHANUM-ED-PIC-STRING → PIC-STRING
NUM-ED-PIC-STRING → PIC-STRING

PIC-A → ALPHA-PIC-ELT
ALPHA-PIC-ELT+ → ALPHA-PIC-STRING

PIC-9 → NUM-PIC-ELT
PIC-P → NUM-PIC-ELT
"S" → NUM-PIC-ELT
"V" → NUM-PIC-ELT
NUM-PIC-ELT+ → NUM-PIC-STRING

PIC-A → ALPHANUM-ED-PIC-ELT1
PIC-X → ALPHANUM-ED-PIC-ELT1
PIC-9 → ALPHANUM-ED-PIC-ELT
PIC-B → ALPHANUM-ED-PIC-ELT2
PIC-0 → ALPHANUM-ED-PIC-ELT2
PIC-SLANT → ALPHANUM-ED-PIC-ELT2

ALPHANUM-ED-PIC-ELT1 → ALPHANUM-ED-PIC-ELT
ALPHANUM-ED-PIC-ELT2 → ALPHANUM-ED-PIC-ELT

ALPHANUM-ED-PIC-ELT∗ ALPHANUM-ED-PIC-ELT1
ALPHANUM-ED-PIC-ELT∗ ALPHANUM-ED-PIC-ELT2

```
ALPHANUM-ED-PIC-ELT*                                     → ALPHANUM-ED-PIC-STRING

ALPHANUM-ED-PIC-ELT* ALPHANUM-ED-PIC-ELT2
ALPHANUM-ED-PIC-ELT* ALPHANUM-ED-PIC-ELT1
ALPHANUM-ED-PIC-ELT*                                     → ALPHANUM-ED-PIC-STRING

                                                         → EXTERNAL
OPT-IS "EXTERNAL"                                        → EXTERNAL

                                                         → GLOBAL
OPT-IS "GLOBAL"                                          → GLOBAL
```

B.2.16 Verbs-library

Verbs library [section V], used by all verb-modules. Part of Procedure Division.

imports Cobol-data[B.2.17]
exports
sorts OVERFLOW AT-END ERROR YES-AT-END NO-AT-END CORR FROM INVALID NOT-INVALID YES-OVERFLOW NO-OVERFLOW YES-ERROR NO-ERROR THRU-PROCEDURE BEFORE-OR-AFTER
context-free syntax

```
YES-OVERFLOW NO-OVERFLOW                          → OVERFLOW

                                                  → YES-OVERFLOW
OPT-ON "OVERFLOW" IMPERATIVE-STATEMENT            → YES-OVERFLOW

                                                  → NO-OVERFLOW
"NOT" YES-OVERFLOW                                → NO-OVERFLOW

YES-AT-END NO-AT-END                              → AT-END

                                                  → YES-AT-END
OPT-AT "END" IMPERATIVE-STATEMENT                 → YES-AT-END

                                                  → NO-AT-END
"NOT" YES-AT-END                                  → NO-AT-END

YES-ERROR NO-ERROR                                → ERROR

                                                  → YES-ERROR
OPT-ON "SIZE" "ERROR" IMPERATIVE-STATEMENT        → YES-ERROR

                                                  → NO-ERROR
"NOT" YES-ERROR                                   → NO-ERROR

"CORRESPONDING"                                   → CORR
"CORR"                                            → CORR

                                                  → FROM
"FROM" ID                                         → FROM

                                                  → INVALID
"INVALID" OPT-KEY IMPERATIVE-STATEMENT            → INVALID

                                                  → NOT-INVALID
"NOT" INVALID                                     → NOT-INVALID

                                                  → THRU-PROCEDURE
THROUGH ID-NAME                                   → THRU-PROCEDURE

"BEFORE"                                          → BEFORE-OR-AFTER
"AFTER"                                           → BEFORE-OR-AFTER
```

B.2.17 Cobol-data

Cobol Data [pages IV-4, IV-12, V]: Integers, Identifiers, Literals; Expressions, Conditions; Standard choices.

imports Optionals$^{B.2.18}$
exports
sorts ID-OR-LIT ID-OR-INT CHARACTER STATEMENT
CHARACTERS INTEGER NAME-ELT ID CBL-LITERAL
IMPERATIVE-STATEMENT QUALIFICATION IN-OR-OF
NUM-LITERAL NONNUM-LITERAL ARITH-EXP
ID-ID-NAME-LIT-ARITH REL-OPTIONS REL-CONDITION
COND-TYPE OPT-REL-OPTIONS CONDITION
SWITCH-CONDITION NEXT-CONDITION ABBR-REL-COND
SIGN-CONDITION NUMERIC-TYPE COND-NAME-CONDITION
CLASS-CONDITION STRING-ELT OPT-SUBSCR-OPTION
SUBSCRIPT-OPTION SUBSCRIPT LEFTMOST OPT-INT NUM
KEY-OPTION SINGLE-DIGIT FIRST-DIGIT LAST-DIGIT
LEVEL-NUMBER THROUGH A-OR-D INDEXED LINE-OR-LINES
CURRENCY-SYMBOL NON-ZERO-INTEGER OPERATOR
RELATION RELATION2 FIRST-ZERO RELATION3 ALG-LESS
ALG-EQUAL ALG-GEQ ALG-LEQ ALG-GREATER ALG-STAR
ALG-PLUS ALG-MINUS SEPARATOR FIG-CONSTANT
ID-NAME OPT-ID-NAME ID-OR-ID-NAME PSEUDO
ID-OR-ID-NAME-OR-LIT ID-TYPE
lexical syntax
“≡”STRING-ELT “≡” → PSEUDO

“77” → LEVEL-NUMBER

[0-9]∗ → INTEGER
INTEGER → NON-ZERO-INTEGER

[0-9]+“.”[0-9]+ → NUM
[0-9]+ → NUM

Numerical literals can be no longer than 18 digits [page IV-10]. Although this is a syntactical restraint this is not implemented here but postponed to the semantical phase.

lexical syntax
NUM → NUM-LITERAL
“–”NUM → NUM-LITERAL
“+”NUM → NUM-LITERAL

Nonnumeric literals are made up of string elements and can be up to 160 characters in length [page IV-9]. This has not been treated here. Two consecutive quotationmarks are considered part of the nonnumeric literal (with the meaning of one quotationmark, perhaps something for the semantic phase):

lexical syntax
“\"”~[\"]+“\"” → STRING-ELT

[A-Z0-9] → NAME-ELT

Cobol words (here: ID) can not be longer than 30 characters [page IV-5]. This again has not been dealt with at the syntactic level.

lexical syntax
NAME-ELT∗[A-Z]NAME-ELT∗ → ID-TYPE
NAME-ELT [\-A-Z0-9]∗“–”[\-A-Z0-9]∗NAME-ELT → ID-TYPE

[A-Za-z]	→ CHARACTER
CHARACTER CHARACTER+	→ CHARACTERS

The difference between slant and asterisk has no meaning for us (yet): it deals with page-breaking when printing a Cobol program [page IV-42].

lexical syntax

"*" ~[\n]* "\n"	→ LAYOUT
"/" ~[\n]* "\n"	→ LAYOUT
[␣\t\n]	→ LAYOUT
[1-4]	→ FIRST-DIGIT
[0-9]	→ LAST-DIGIT
[1-9]	→ SINGLE-DIGIT
[0]	→ FIRST-ZERO
"+"	→ ALG-PLUS
"−"	→ ALG-MINUS
"*"	→ ALG-STAR
">"	→ ALG-GREATER
"<"	→ ALG-LESS
"="	→ ALG-EQUAL
"≥"	→ ALG-GEQ
"≤"	→ ALG-LEQ

The currency symbol is used in Data-lib.syn and Env-div.syn [page VI-17]. Note that the a-z is not necessary here, because every source program is converted to its uppercase equivalent anyway. This means however that we cannot check syntactically on capitalization errors.
The standard currency symbol ($) is the only currency symbol allowed semantically, except when this is overruled in the SPECIAL-NAMES section.

lexical syntax

~[\n\t␣0-9a-zABCDPRSVXZ*+\−, .; ()" = /]	→ CURRENCY-SYMBOL
","	→ SEPARATOR

context-free syntax

ID-TYPE	→ ID-NAME
IN ID-NAME	→ IN-OR-OF
OF ID-NAME	→ IN-OR-OF
ALG-PLUS	→ OPERATOR
ALG-MINUS	→ OPERATOR
ALG-STAR	→ OPERATOR
ALG-GREATER	→ RELATION
ALG-GREATER	→ RELATION3
ALG-LESS	→ RELATION
ALG-EQUAL	→ RELATION
ALG-EQUAL	→ RELATION3
ALG-GEQ	→ RELATION2
ALG-LEQ	→ RELATION2
STRING-ELT	→ NONNUM-LITERAL

context-free syntax

OPT-ALL "ZERO"	→ FIG-CONSTANT
OPT-ALL "ZEROS"	→ FIG-CONSTANT
OPT-ALL "ZEROES"	→ FIG-CONSTANT
OPT-ALL "SPACE"	→ FIG-CONSTANT
OPT-ALL "SPACES"	→ FIG-CONSTANT

```
OPT-ALL "HIGH-VALUE"    → FIG-CONSTANT
OPT-ALL "HIGH-VALUES"   → FIG-CONSTANT
OPT-ALL "LOW-VALUE"     → FIG-CONSTANT
OPT-ALL "LOW-VALUES"    → FIG-CONSTANT
OPT-ALL "QUOTE"         → FIG-CONSTANT
OPT-ALL "QUOTES"        → FIG-CONSTANT
"ALL" NONNUM-LITERAL    → FIG-CONSTANT

NUM-LITERAL             → CBL-LITERAL
NONNUM-LITERAL          → CBL-LITERAL
FIG-CONSTANT            → CBL-LITERAL

ID                      → ID-OR-LIT
CBL-LITERAL             → ID-OR-LIT

ID-NAME                 → ID-OR-INT
INTEGER                 → ID-OR-INT
```

Level numbers 66 and 88 are excluded here, see our `Data-description-entry` for details. This problem has deliberately been dealt with on the syntactical level.

context-free syntax

```
SINGLE-DIGIT                → LEVEL-NUMBER
FIRST-ZERO SINGLE-DIGIT     → LEVEL-NUMBER
FIRST-DIGIT LAST-DIGIT      → LEVEL-NUMBER
```

Possible formats for Cobol identifiers:

Qualifications: (note that in fact all four are qualifications, not just the first one).

context-free syntax

```
ID-NAME QUALIFICATION                   → ID
"LINAGE-COUNTER" IN-OR-OF ID-NAME       → ID
"PAGE-COUNTER" IN-OR-OF ID-NAME         → ID
"LINE-COUNTER" IN-OR-OF ID-NAME         → ID
```

Identifiers [page IV-23]:

context-free syntax

```
ID-NAME IN-OR-OF* SUBSCRIPT LEFTMOST → ID

IN-OR-OF+                            → QUALIFICATION
```

Number of subscripts [pages II-14, IV-21].

context-free syntax

```
                                    → SUBSCRIPT
"(" {SUBSCRIPT-OPTION ","}+ ")"     → SUBSCRIPT

INTEGER                             → SUBSCRIPT-OPTION
ID-NAME OPT-SUBSCR-OPTION           → SUBSCRIPT-OPTION

                                    → OPT-SUBSCR-OPTION
ALG-PLUS INTEGER                    → OPT-SUBSCR-OPTION
ALG-MINUS INTEGER                   → OPT-SUBSCR-OPTION

                                    → LEFTMOST
"(" INTEGER ":" OPT-INT ")"         → LEFTMOST

                                    → OPT-INT
INTEGER                             → OPT-INT
```

Conditions:

context-free syntax

REL-CONDITION	→ CONDITION	
CLASS-CONDITION	→ CONDITION	
COND-NAME-CONDITION	→ CONDITION	
SWITCH-CONDITION	→ CONDITION	
SIGN-CONDITION	→ CONDITION	
"NOT" CONDITION	→ CONDITION	
CONDITION NEXT-CONDITION+	→ CONDITION	

Please note: the `object` occurrence on [page V-36] is not defined (not mentioned in index and not explained in [page VI-61]).

context-free syntax

REL-CONDITION ABBR-REL-COND+	→ CONDITION	
"(" CONDITION ")"	→ CONDITION	{**bracket**}
ID-ID-NAME-LIT-ARITH OPT-IS REL-OPTIONS ID-ID-NAME-LIT-ARITH	→ REL-CONDITION	
ID OPT-IS OPT-NOT COND-TYPE	→ CLASS-CONDITION	
ID-NAME	→ COND-NAME-CONDITION	
ARITH-EXP OPT-IS OPT-NOT NUMERIC-TYPE	→ SIGN-CONDITION	
OPT-NOT "GREATER" OPT-THAN	→ REL-OPTIONS	
OPT-NOT RELATION	→ REL-OPTIONS	
OPT-NOT "LESS" OPT-THAN	→ REL-OPTIONS	
OPT-NOT "EQUAL" OPT-TO	→ REL-OPTIONS	
RELATION2	→ REL-OPTIONS	
"LESS" OPT-THAN "OR" "EQUAL" OPT-TO	→ REL-OPTIONS	
ARITH-EXP	→ ID-ID-NAME-LIT-ARITH	
ID-OR-ID-NAME-OR-LIT	→ ID-ID-NAME-LIT-ARITH	
"NUMERIC"	→ COND-TYPE	
"ALPHABETIC"	→ COND-TYPE	
"ALPHABETIC-LOWER"	→ COND-TYPE	
"ALPHABETIC-UPPER"	→ COND-TYPE	
ID-NAME	→ COND-TYPE	
"POSITIVE"	→ NUMERIC-TYPE	
"NEGATIVE"	→ NUMERIC-TYPE	
"ZERO"	→ NUMERIC-TYPE	
"AND" CONDITION	→ NEXT-CONDITION	
"OR" CONDITION	→ NEXT-CONDITION	
"AND" OPT-REL-OPTIONS ID	→ ABBR-REL-COND	
"OR" OPT-REL-OPTIONS ID	→ ABBR-REL-COND	
REL-OPTIONS	→ OPT-REL-OPTIONS	
	→ OPT-REL-OPTIONS	
ID-OR-LIT	→ ARITH-EXP	
"(" ARITH-EXP ")"	→ ARITH-EXP	{**bracket**}

```
    ARITH-EXP OPERATOR ARITH-EXP     → ARITH-EXP        {left}
    ALG-MINUS ARITH-EXP              → ARITH-EXP
```

A few shared sorts:

```
  context-free syntax
    "KEY" OPT-IS ID-NAME  → KEY-OPTION

    "ASCENDING"           → A-OR-D
    "DESCENDING"          → A-OR-D

                          → INDEXED
    "INDEXED" OPT-BY ID+  → INDEXED

    "THROUGH"             → THROUGH
    "THRU"                → THROUGH

                          → OPT-ID-NAME
    ID-NAME               → OPT-ID-NAME
```

Some standard choices:

```
  context-free syntax
    OPT-LINE     → LINE-OR-LINES
    OPT-LINES    → LINE-OR-LINES

    ID           → ID-OR-ID-NAME
    ID-NAME      → ID-OR-ID-NAME

    ID           → ID-OR-ID-NAME-OR-LIT
    CBL-LITERAL  → ID-OR-ID-NAME-OR-LIT
```

B.2.18 Optionals

Optionals [section V], used by most modules via Cobol-data. In a future version of ASF+SDF (with options) this module could perhaps be left out (together with many rules in the rest of the specification).

```
exports
  sorts OPT-PROGRAM OPT-COLLATING OPT-WITH OPT-IS
        OPT-STATUS OPT-CHARACTERS OPT-ARE OPT-SIGN
        OPT-AREA OPT-FOR OPT-TAPE OPT-CONTAINS OPT-TO
        OPT-AREAS OPT-CHARACTER OPT-MODE OPT-FILE
        OPT-KEY OPT-IN OPT-SIZE OPT-ON OPT-LINES OPT-AT
        OPT-TIMES OPT-BY OPT-RIGHT OPT-WHEN OPT-SYMBOLIC
        OPT-MESSAGE OPT-LIMIT OPT-LIMITS OPT-LINE
        OPT-NUMBER OPT-INDICATE OPT-DATA OPT-INITIAL
        OPT-RECORD OPT-ADVANCING OPT-ORDER OPT-OR
        OPT-THAN OPT-PRINTING OPT-STANDARD OPT-OF
        OPT-ROUNDED OPT-GLOBAL OPT-ALL OPT-NEXT
        OPT-REFERENCES OPT-TERMINAL OPT-THEN OPT-NOT
        OPT-END-IF IN OF IN-TYPE OPT-FROM
```

Here is an example of a problem with ASF+SDF when specifying COBOL. In COBOL, the phrase `INITIAL` exists. Because `IN` also exists as a COBOL keyword, ASF+SDF will recognize `INITIAL` as `IN ITIAL` (because `IN` is defined lexically it is recognized 'first'). This is of course incorrect, that is why we created the following construct, where `IN` is recognized in conjunction with everything except a trailing `I`.
This problem occurs with many other constructs in COBOL. We have only solved the problem for this one case however; the others can be dealt with in a similar way. Observe

that ˜ (indicating 'not') followed by something like 'not a space character' would not be a sufficient solution. (This negative formulation is necessary because it is impossible to state that a space character should occur at a certain position lexically: spaces would then be recognized everywhere throughout the source program.)

lexical syntax
"IN"~[I] → IN-TYPE
context-free syntax

```
IN-TYPE       → IN

"OF"          → OF

              → OPT-PROGRAM
"PROGRAM"     → OPT-PROGRAM

              → OPT-COLLATING
"COLLATING"   → OPT-COLLATING

              → OPT-WITH
"WITH"        → OPT-WITH

              → OPT-IS
"IS"          → OPT-IS

              → OPT-STATUS
"STATUS"      → OPT-STATUS

              → OPT-CHARACTERS
"CHARACTERS"  → OPT-CHARACTERS

              → OPT-ARE
"ARE"         → OPT-ARE

              → OPT-SIGN
"SIGN"        → OPT-SIGN

              → OPT-AREA
"AREA"        → OPT-AREA

              → OPT-FOR
"FOR"         → OPT-FOR

              → OPT-TAPE
"TAPE"        → OPT-TAPE

              → OPT-CONTAINS
"CONTAINS"    → OPT-CONTAINS

              → OPT-TO
"TO"          → OPT-TO

              → OPT-AREAS
"AREAS"       → OPT-AREAS

              → OPT-CHARACTER
"CHARACTER"   → OPT-CHARACTER

              → OPT-MODE
"MODE"        → OPT-MODE

              → OPT-FILE
"FILE"        → OPT-FILE

              → OPT-KEY
```

"KEY"	→ OPT-KEY
	→ OPT-IN
IN	→ OPT-IN
	→ OPT-SIZE
"SIZE"	→ OPT-SIZE
	→ OPT-ON
"ON"	→ OPT-ON
	→ OPT-LINES
"LINES"	→ OPT-LINES
	→ OPT-AT
"AT"	→ OPT-AT
	→ OPT-TIMES
"TIMES"	→ OPT-TIMES
	→ OPT-BY
"BY"	→ OPT-BY
	→ OPT-RIGHT
"RIGHT"	→ OPT-RIGHT
	→ OPT-WHEN
"WHEN"	→ OPT-WHEN
	→ OPT-SYMBOLIC
"SYMBOLIC"	→ OPT-SYMBOLIC
	→ OPT-MESSAGE
"MESSAGE"	→ OPT-MESSAGE
	→ OPT-LIMIT
"LIMIT"	→ OPT-LIMIT
	→ OPT-LIMITS
"LIMITS"	→ OPT-LIMITS
	→ OPT-LINE
"LINE"	→ OPT-LINE
	→ OPT-NUMBER
"NUMBER"	→ OPT-NUMBER
	→ OPT-INDICATE
"INDICATE"	→ OPT-INDICATE
	→ OPT-DATA
"DATA"	→ OPT-DATA
	→ OPT-INITIAL
"INITIAL"	→ OPT-INITIAL
	→ OPT-RECORD
"RECORD"	→ OPT-RECORD
	→ OPT-ADVANCING
"ADVANCING"	→ OPT-ADVANCING
	→ OPT-ORDER
"ORDER"	→ OPT-ORDER

	→ OPT-OR
"OR"	→ OPT-OR
	→ OPT-THAN
"THAN"	→ OPT-THAN
	→ OPT-PRINTING
"PRINTING"	→ OPT-PRINTING
	→ OPT-STANDARD
"STANDARD"	→ OPT-STANDARD
	→ OPT-OF
OF	→ OPT-OF
	→ OPT-ROUNDED
"ROUNDED"	→ OPT-ROUNDED
	→ OPT-GLOBAL
"GLOBAL"	→ OPT-GLOBAL
	→ OPT-ALL
"ALL"	→ OPT-ALL
	→ OPT-NEXT
"NEXT"	→ OPT-NEXT
	→ OPT-REFERENCES
"REFERENCES"	→ OPT-REFERENCES
	→ OPT-TERMINAL
"TERMINAL"	→ OPT-TERMINAL
	→ OPT-THEN
"THEN"	→ OPT-THEN
	→ OPT-NOT
"NOT"	→ OPT-NOT
	→ OPT-END-IF
"END-IF"	→ OPT-END-IF
	→ OPT-FROM
"FROM"	→ OPT-FROM

Appendix C

Year 2000 tool

First some search criteria for year 2000 tools are presented in C.1, then the specification for our tool (based on the syntax specification listed in appendix B.2) is given.

C.1 Search criteria

First of all we list search criteria for variables and other texts in a COBOL source program. They come from IBM and Unibol respectively. Note that these criteria are based on the English language.

C.1.1 IBM strings

IBM's search hints are listed in Figure C.1.

AS OF	ASOF	BEGIN	BEG	BGN
CCYY	CYYDDD	CYYDDMM	CYYMMDD	CURR
CURRENT	DATE	DAT	DTE	DT
DAY	DA	DD	DDMMYY	DDDYY
DIFFDATE	DOB	DOH	END	EXPIRE
EXP	JULIAN	MONTH	MON	MO
MMM	MDY	MMDDYY	START	TERM
TIME	TIMESTAMP	TIME STAMP	TIMEDATE	THISDATE
TOD	T O D	WEEK	WEEKDAY	YEAR
YR	YY	YMD	YYMMDD	YYDDD

Figure C.1: IBM strings

C.1.2 Unibol strings

Unibol's search hints are listed in Figure C.2.

C.2 Specification

C.2.1 Tool

imports Cobol-Syntax$^{B.2.1}$ Booleans$^{C.2.2}$ Integers

19	APR	AUG	CC	CENT
CUR	DAT$	DAT	DAT(A)	DAY
DEC	DMY	DOB	DOH	DT
DY	EXPIR	FEB	GREG	JAN
JJ	JUL	JUN	LEAP	MAR
MAY	MM	MNT	MONTH	MTH
NOV	OCT	SEP	WHEN	YEAR
YER	YMD	YR	YY	(A)DD

Figure C.2: Unibol strings

exports
sorts TENV-ELM TENV TENV-ELM-LIST FILLER ID-FIL-LIT ID-FIL-LIT-STAR
context-free syntax

```
check(PROGRAM)                            → TENV
danger(PROGRAM)                           → BOOL
show-dates(PROGRAM)                       → TENV
density(PROGRAM)                          → INT

has-dates(TENV)                           → BOOL
add(ID-FIL-LIT*, PIC-STRING*, TENV)       → TENV
```

Overloading is impossible for `is-date` due to the ambiguity that arises when both are empty:

context-free syntax

```
is-date(PIC-STRING*)                      → BOOL
is-date1(PIC-STRING)                      → BOOL
is-date2(ID-FIL-LIT*)                     → BOOL
filter(TENV-ELM-LIST)                     → TENV-ELM-LIST

ID-NAME-OR-FILLER                         → ID-FIL-LIT
CBL-LITERAL                               → ID-FIL-LIT

ID-FIL-LIT ":" PIC-STRING* ":" INT        → TENV-ELM

{TENV-ELM "#"}*                           → TENV-ELM-LIST

"[" TENV-ELM-LIST "]"                     → TENV

make-env(DATA-DESCR-ENTRY*)               → TENV

make-env-proc-div(PROC-DIV, TENV)         → TENV
make-env-fs(FILE-SECTION)                 → TENV
make-env-wss(WORKING-STORAGE-SECTION)     → TENV
make-env-ls(LINKAGE-SECTION)              → TENV
make-env-cs(COMM-SECTION)                 → TENV

is-move(STATEMENT2)                       → BOOL
is-compute(STATEMENT2)                    → BOOL
is-add(STATEMENT2)                        → BOOL
is-subtract(STATEMENT2)                   → BOOL

append(TENV, TENV)                        → TENV

analyse-par(PARAGRAPH*, TENV)             → TENV
analyse-sen(SENTENCE*, TENV)              → TENV
analyse-stat(STATEMENT2+, TENV)           → TENV
```

is-date-in-tenv(ID-FIL-LIT∗, TENV)	→ INT
update(ID-FIL-LIT∗, INT, TENV)	→ TENV
equal(ID-FIL-LIT∗, ID-FIL-LIT∗)	→ BOOL
count(TENV-ELM-LIST)	→ INT
highest(INT, INT)	→ INT
"FILLER"	→ FILLER
ID-FIL-LIT∗	→ ID-FIL-LIT-STAR
get-name(ID+)	→ ID-FIL-LIT-STAR
make-ifl(ID-OR-LIT∗)	→ ID-FIL-LIT-STAR
get-id-from-rounded(ROUNDED+)	→ ID-FIL-LIT

hiddens

variables

Tenv [*0-9′*]∗	→ TENV
Tenv-elm	→ TENV-ELM
Tenv-elm-list [*0-9′*]∗	→ {TENV-ELM "#"}∗
Id-name [*1-9′*]∗	→ ID-NAME
Id-lit	→ ID-OR-LIT
Id-lits	→ ID-OR-LIT+
Program	→ PROGRAM
Opt-loop	→ OPT-LOOP
Id-div	→ ID-DIV
Env-division	→ ENV-DIV
Proc-division	→ PROC-DIV
Data-descr-entry	→ DATA-DESCR-ENTRY∗
F-section	→ FILE-SECTION
W-S-section	→ WORKING-STORAGE-SECTION
L-section	→ LINKAGE-SECTION
C-section	→ COMM-SECTION
R-section	→ REPORT-SECTION
R-D-entry	→ RECORD-DESCR-ENTRY
SRI-fd-entry	→ S-R-I-FD-ENTRY
Cd-entry	→ COMM-DESCR-ENTRY
Dd-through	→ DD-THROUGH
Dd-values	→ DD-VALUES
Arg-88	→ ARG-88∗
Picture	→ PICTURE
Pic-string	→ PIC-STRING∗
Pic-string2	→ PIC-STRING
Pic-lit	→ PIC-LIT
Opt-is	→ OPT-IS
Level-number	→ LEVEL-NUMBER
Id-name-or-filler	→ ID-NAME-OR-FILLER
Id-fil-lit [*0-9*]∗	→ ID-FIL-LIT
Id-fil-lits [*0-9*]	→ ID-FIL-LIT

Redefines	→ REDEFINES
External	→ EXTERNAL
Global	→ GLOBAL
Usage	→ USAGE
Lead-or-trail	→ LEAD-OR-TRAIL
Occurs	→ OCCURS
Synchronize	→ SYNCHRONIZE
Justify	→ JUSTIFY
Blank	→ BLANK
Dd-value	→ DD-VALUE
Proc-header	→ PROC-HEADER
Paragraph	→ PARAGRAPH∗
Sentence	→ SENTENCE∗
Statement [0-9′]∗	→ STATEMENT2+
One-statement	→ STATEMENT2
Move-statement	→ MOVE-STATEMENT
Compute-statement	→ COMPUTE-STATEMENT
Add-statement	→ ADD-STATEMENT
Subtract-statement	→ SUBTRACT-STATEMENT
Fig-constant	→ FIG-CONSTANT
Id	→ ID+
Cbl-literal [0-9′]∗	→ CBL-LITERAL
Identifier [0-9′]∗	→ ID
Corr	→ CORR
Subscript	→ SUBSCRIPT
Leftmost	→ LEFTMOST
Int	→ INT
Rounded	→ ROUNDED+
Error	→ ERROR
End-add	→ END-ADD
Opt-rounded	→ OPT-ROUNDED

equations

$$[\text{ch1}]\quad \frac{\begin{array}{rcl}\text{make-env-fs}(\textit{F-section}) & = & \textit{Tenv}_1,\\ \text{make-env-wss}(\textit{W-S-section}) & = & \textit{Tenv}_2,\\ \text{make-env-ls}(\textit{L-section}) & = & \textit{Tenv}_3,\\ \text{make-env-cs}(\textit{C-section}) & = & \textit{Tenv}_4\end{array}}{\begin{array}{l}\text{check}(\textit{Id-div Env-division}\ \text{DATA DIVISION.}\ \textit{F-section W-S-section L-section C-section R-section}\\ \qquad \textit{Proc-division Opt-loop}) =\\ \text{make-env-proc-div}(\textit{Proc-division},\\ \qquad \text{append}(\textit{Tenv}_1,\ \text{append}(\textit{Tenv}_2,\ \text{append}(\textit{Tenv}_3,\ \textit{Tenv}_4))))\end{array}}$$

$$[\text{app1}]\quad \frac{\begin{array}{l}\textit{Tenv-elm-list}_1 \neq\ ,\\ \textit{Tenv-elm-list}_2 \neq\end{array}}{\begin{array}{l}\text{append}([\textit{Tenv-elm-list}_1],\ [\textit{Tenv-elm-list}_2]) =\\ [\textit{Tenv-elm-list}_1 \#\ \textit{Tenv-elm-list}_2]\end{array}}$$

$$[\text{app2}]\quad \text{append}([],\ [\textit{Tenv-elm-list}_2]) = [\textit{Tenv-elm-list}_2]$$

$$[\text{app3}]\quad \text{append}([\textit{Tenv-elm-list}_1],\ [])$$

$= [\,Tenv\text{-}elm\text{-}list_1]$

$$[\text{mp1}]\ \frac{\begin{array}{rcl}\text{analyse-par}(Paragraph,\ Tenv_1) &=& Tenv_2,\\ \text{analyse-par}(Paragraph,\ Tenv_2) &=& Tenv_2\end{array}}{\text{make-env-proc-div}(Proc\text{-}header\ Paragraph,\ Tenv_1)\ =\ Tenv_2}$$

$$[\text{mp2}]\ \frac{\begin{array}{rcl}\text{analyse-par}(Paragraph,\ Tenv_1) &=& Tenv_2,\\ \text{analyse-par}(Paragraph,\ Tenv_2) &=& Tenv_3,\\ Tenv_2 &\neq& Tenv_3\end{array}}{\begin{array}{l}\text{make-env-proc-div}(Proc\text{-}header\ Paragraph,\ Tenv_1)\ =\\ \text{make-env-proc-div}(Proc\text{-}header\ Paragraph,\ Tenv_3)\end{array}}$$

$$[\text{ap1}]\ \frac{\begin{array}{rcl}\text{analyse-sen}(Sentence,\ Tenv) &=& Tenv_1,\\ \text{analyse-par}(Paragraph,\ Tenv_1) &=& Tenv_2\end{array}}{\text{analyse-par}(Id\text{-}name\ .\ Sentence\ Paragraph,\ Tenv)\ =\ Tenv_2}$$

$$[\text{ap2}]\ \text{analyse-par}(,\ Tenv) = Tenv$$

$$[\text{as1}]\ \frac{\begin{array}{rcl}\text{analyse-stat}(Statement,\ Tenv) &=& Tenv_1,\\ \text{analyse-sen}(Sentence,\ Tenv_1) &=& Tenv_2\end{array}}{\text{analyse-sen}(Statement\ .\ Sentence\ ,\ Tenv)\ =\ Tenv_2}$$

$$[\text{as2}]\ \text{analyse-sen}(,\ Tenv) = Tenv$$

$$\begin{array}{lll}[\text{ge1}] & \text{get-name}(Id\text{-}name\ \ Subscript\ Leftmost) & = Id\text{-}name\\ [\text{ge2}] & \text{get-name}(Id\text{-}name\ \ Subscript\ ) & = Id\text{-}name\\ [\text{ge3}] & \text{get-name}(Id\text{-}name\ \ \ Leftmost) & = Id\text{-}name\\ [\text{ge4}] & \text{get-name}(Id\text{-}name\ \ \ ) & = Id\text{-}name\end{array}$$

$$[\text{ast1}]\ \frac{\begin{array}{rcl}\text{get-name}(Identifier_1) &=& Id\text{-}fil\text{-}lit,\\ \text{get-name}(Identifier_2) &=& Id\text{-}fil\text{-}lit_2,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit,\ Tenv) &=& Int_1,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit_2,\ Tenv) &=& Int_2,\\ \text{highest}(Int_1,\ Int_2) &=& Int_3,\\ Int_3 &\neq& 1,\\ \text{update}(Id\text{-}fil\text{-}lit,\ Int_3,\ Tenv) &=& Tenv_2,\\ \text{update}(Id\text{-}fil\text{-}lit_2,\ Int_3,\ Tenv_2) &=& Tenv_3\end{array}}{\begin{array}{l}\text{analyse-stat}(\text{MOVE}\ Corr\ Identifier_1\ \text{TO}\ Identifier_2,\ Tenv)\ =\\ Tenv_3\end{array}}$$

$$[\text{ast2}]\ \frac{\begin{array}{rcl}\text{get-name}(Identifier_1) &=& Id\text{-}fil\text{-}lit,\\ \text{get-name}(Identifier_2) &=& Id\text{-}fil\text{-}lit_2,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit,\ Tenv) &=& 1,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit_2,\ Tenv) &=& 1\end{array}}{\text{analyse-stat}(\text{MOVE}\ Corr\ Identifier_1\ \text{TO}\ Identifier_2,\ Tenv)\ =\ Tenv}$$

$$[\text{ast3}]\ \frac{\begin{array}{rcl}\text{make-ifl}(Id\text{-}lit) &=& Id\text{-}fil\text{-}lit,\\ \text{get-name}(Id) &=& Id\text{-}fil\text{-}lit_2,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit,\ Tenv) &=& 1,\\ \text{is-date-in-tenv}(Id\text{-}fil\text{-}lit_2,\ Tenv) &=& 1\end{array}}{\text{analyse-stat}(\text{MOVE}\ Id\text{-}lit\ \text{TO}\ Id,\ Tenv)\ =\ Tenv}$$

$$\text{[ast4]}\ \frac{\begin{array}{rcl} \text{get-name}(\mathit{Id}) &=& \mathit{Id\text{-}fil\text{-}lit}_1, \\ \text{make-ifl}(\mathit{Id\text{-}lit}) &=& \mathit{Id\text{-}fil\text{-}lit}_2, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit}_1,\ \mathit{Tenv}) &=& \mathit{Int}_1, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit}_2,\ \mathit{Tenv}) &=& \mathit{Int}_2, \\ \text{highest}(\mathit{Int}_1,\ \mathit{Int}_2) &=& \mathit{Int}_3, \\ \mathit{Int}_3 &\neq& 1, \\ \text{update}(\mathit{Id\text{-}fil\text{-}lit}_1,\ \mathit{Int}_3,\ \mathit{Tenv}) &=& \mathit{Tenv}_2, \\ \text{update}(\mathit{Id\text{-}fil\text{-}lit}_2,\ \mathit{Int}_3,\ \mathit{Tenv}_2) &=& \mathit{Tenv}_3 \end{array}}{\text{analyse-stat}(\text{MOVE } \mathit{Id\text{-}lit} \text{ TO } \mathit{Id},\ \mathit{Tenv}) = \mathit{Tenv}_3}$$

$$\text{[hi1]}\ \frac{\text{gt}(\mathit{Nat\text{-}con}_1,\ \mathit{Nat\text{-}con}_2) = \text{true}}{\text{highest}(\mathit{Nat\text{-}con}_1,\ \mathit{Nat\text{-}con}_2) = \mathit{Nat\text{-}con}_1}$$

$$\text{[hi2]}\ \frac{\text{gt}(\mathit{Nat\text{-}con}_1,\ \mathit{Nat\text{-}con}_2) \neq \text{true}}{\text{highest}(\mathit{Nat\text{-}con}_1,\ \mathit{Nat\text{-}con}_2) = \mathit{Nat\text{-}con}_2}$$

$$\text{[ast4b]}\ \frac{\begin{array}{rcl} \text{get-id-from-rounded}(\mathit{Rounded}) &=& \mathit{Id\text{-}fil\text{-}lit}, \\ \text{make-ifl}(\mathit{Id\text{-}lits}) &=& \mathit{Id\text{-}fil\text{-}lit}_2, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit},\ \mathit{Tenv}) &=& \mathit{Int}_1, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit}_2,\ \mathit{Tenv}) &=& \mathit{Int}_2, \\ \text{highest}(\mathit{Int}_1,\ \mathit{Int}_2) &=& \mathit{Int}_3, \\ \mathit{Int}_3 &\neq& 1, \\ \text{update}(\mathit{Id\text{-}fil\text{-}lit}_2,\ 3,\ \mathit{Tenv}) &=& \mathit{Tenv}_2, \\ \text{update}(\mathit{Id\text{-}fil\text{-}lit},\ 3,\ \mathit{Tenv}_2) &=& \mathit{Tenv}_3 \end{array}}{\text{analyse-stat}(\text{ADD } \mathit{Id\text{-}lits} \text{ TO } \mathit{Rounded\ Error\ End\text{-}add},\ \mathit{Tenv}) = \mathit{Tenv}_3}$$

$$\text{[ast4c]}\ \frac{\begin{array}{rcl} \text{get-id-from-rounded}(\mathit{Rounded}) &=& \mathit{Id\text{-}fil\text{-}lit}, \\ \text{make-ifl}(\mathit{Id\text{-}lits}) &=& \mathit{Id\text{-}fil\text{-}lit}_2, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit},\ \mathit{Tenv}) &=& 1, \\ \text{is-date-in-tenv}(\mathit{Id\text{-}fil\text{-}lit}_2,\ \mathit{Tenv}) &=& 1 \end{array}}{\text{analyse-stat}(\text{ADD } \mathit{Id\text{-}lits} \text{ TO } \mathit{Rounded\ Error\ End\text{-}add},\ \mathit{Tenv}) = \mathit{Tenv}}$$

$$\text{[gi1]}\ \frac{\text{get-name}(\mathit{Identifier}) = \mathit{Id\text{-}fil\text{-}lit}}{\text{get-id-from-rounded}(\mathit{Identifier\ Opt\text{-}rounded}) = \mathit{Id\text{-}fil\text{-}lit}}$$

$$\text{[as5]}\ \frac{\begin{array}{rcl} \text{analyse-stat}(\mathit{One\text{-}statement},\ \mathit{Tenv}) &=& \mathit{Tenv}_2, \\ \text{analyse-stat}(\mathit{Statement},\ \mathit{Tenv}_2) &=& \mathit{Tenv}_3 \end{array}}{\text{analyse-stat}(\mathit{One\text{-}statement\ Statement},\ \mathit{Tenv}) = \mathit{Tenv}_3}$$

$$\text{[as6]}\ \frac{\begin{array}{rcl} \text{is-move}(\mathit{One\text{-}statement}) &\neq& \text{true}, \\ \text{is-compute}(\mathit{One\text{-}statement}) &\neq& \text{true}, \\ \text{is-add}(\mathit{One\text{-}statement}) &\neq& \text{true}, \\ \text{is-subtract}(\mathit{One\text{-}statement}) &\neq& \text{true} \end{array}}{\text{analyse-stat}(\mathit{One\text{-}statement},\ \mathit{Tenv}) = \mathit{Tenv}}$$

ifl = identifier, filler, or literal

[mi1] make-ifl(*Identifier*) = get-name(*Identifier*)

[mi2] make-ifl(*Cbl-literal*) = *Cbl-literal*

[im1] is-move(*Move-statement*) = true
[im2] is-move(*One-statement*) = false **otherwise**

[ic1] is-compute(*Compute-statement*) = true
[ic2] is-compute(*One-statement*) = false **otherwise**

[ia1] is-add(*Add-statement*) = true
[ia2] is-add(*One-statement*) = false **otherwise**

[is1] is-subtract(*Subtract-statement*) = true
[is2] is-subtract(*One-statement*) = false **otherwise**

[it0] is-date-in-tenv(*Fig-constant*, *Tenv*) = 0

[it1] is-date-in-tenv(*Cbl-literal*, *Tenv*) = 1

[it2] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{true}}{\text{is-date-in-tenv}(\textit{Id-fil-lit}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}\#\ \textit{Tenv-elm-list}])\ =\ \textit{Int}}$$

[it3] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{false}}{\text{is-date-in-tenv}(\textit{Id-fil-lit}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}\#\ \textit{Tenv-elm-list}])\ =\ \text{is-date-in-tenv}(\textit{Id-fil-lit}_1,\ [\textit{Tenv-elm-list}])}$$

[it4] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{true}}{\text{is-date-in-tenv}(\textit{Id-fil-lit}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}])\ =\ \textit{Int}}$$

[it5] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{false}}{\text{is-date-in-tenv}(\textit{Id-fil-lit}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}])\ =\ 1}$$

[it6] is-date-in-tenv(*Id-name*, []) = 1

[up1] update(, *Int*, *Tenv*) = *Tenv*

[up2] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{true}}{\text{update}(\textit{Id-fil-lit}_1,\ \textit{Int}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2\#\ \textit{Tenv-elm-list}])\ =\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_1\#\ \textit{Tenv-elm-list}]}$$

[up3] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{false}}{\text{update}(\textit{Id-fil-lit}_1,\ \textit{Int}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2\#\ \textit{Tenv-elm-list}])\ =\ \text{append}([\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2],\ \text{update}(\textit{Id-fil-lit}_1,\ \textit{Int}_1,\ [\textit{Tenv-elm-list}]))}$$

[up4] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{true}}{\text{update}(\textit{Id-fil-lit}_1,\ \textit{Int}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2])\ =\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_1]}$$

[up5] $$\frac{\text{equal}(\textit{Id-fil-lit}_1,\ \textit{Id-fil-lit}_2)\ =\ \text{false}}{\text{update}(\textit{Id-fil-lit}_1,\ \textit{Int}_1,\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2])\ =\ [\textit{Id-fil-lit}_2 : \textit{Pic-string} : \textit{Int}_2]}$$

[eq1] equal(*Id-fil-lit*, *Id-fil-lit*) = true

[eq2] $$\frac{\textit{Id-fil-lit}_1 \neq \textit{Id-fil-lit}_2}{\text{equal}(\textit{Id-fil-lit}_1, \textit{Id-fil-lit}_2) = \text{false}}$$

[mf1] make-env-fs() = []

[mf2] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env-fs(FILE SECTION. } \textit{SRI-fd-entry Data-descr-entry}) = \textit{Tenv}}$$

[mw1] make-env-wss() = []

[mw2] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env-wss(WORKING-STORAGE SECTION. } \textit{Data-descr-entry}) = \textit{Tenv}}$$

[ml1] make-env-ls() = []

[ml2] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env-ls(LINKAGE SECTION. } \textit{Data-descr-entry}) = \textit{Tenv}}$$

[mc1] make-env-cs() = []

[mc2] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env-cs(COMMUNICATION SECTION. } \textit{Cd-entry} \; \textit{Data-descr-entry}) = \textit{Tenv}}$$

[da1] $$\frac{\text{check}(\textit{Program}) = \textit{Tenv}}{\text{danger}(\textit{Program}) = \text{has-dates}(\textit{Tenv})}$$

[sh1] $$\frac{\text{check}(\textit{Program}) = [\textit{Tenv-elm-list}],\quad \text{filter}(\textit{Tenv-elm-list}) = \textit{Tenv-elm-list}'}{\text{show-dates}(\textit{Program}) = [\textit{Tenv-elm-list}']}$$

[de1] $$\frac{\text{check}(\textit{Program}) = [\textit{Tenv-elm-list}],\quad \text{count}(\textit{Tenv-elm-list}) = \textit{Int}}{\text{density}(\textit{Program}) = \textit{Int}}$$

[co1] count(*Id-fil-lit* : *Pic-string* : 1) = 0

[co2] $$\frac{\textit{Int} \neq 1}{\text{count}(\textit{Id-fil-lit} : \textit{Pic-string} : \textit{Int}) = 1}$$

[co3] count(*Id-fil-lit* : *Pic-string* : 1# *Tenv-elm-list*)
= count(*Tenv-elm-list*)

[co4] $$\frac{Int \neq 1}{\text{count}(\textit{Id-fil-lit} : \textit{Pic-string} : Int\#\ \textit{Tenv-elm-list}) = 1 + \text{count}(\textit{Tenv-elm-list})}$$

[co5] count() = 0

[fi1] filter() =

[fi2] filter(*Id-fil-lit* : *Pic-string* : 1# *Tenv-elm-list*)
= filter(*Tenv-elm-list*)

[fi3] $$\frac{\text{filter}(\textit{Tenv-elm-list}) = \textit{Tenv-elm-list}',\quad Int \neq 1}{\text{filter}(\textit{Id-fil-lit} : \textit{Pic-string} : Int\#\ \textit{Tenv-elm-list}) = \textit{Id-fil-lit} : \textit{Pic-string} : Int\#\ \textit{Tenv-elm-list}'}$$

[ma1] make-env() = []

Simultaneous occurrence of the Pic-items has already been checked syntactically:

[ma2] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env}(\textit{Level-number}\ \textit{Id-name-or-filler}\ \textit{Redefines}\ \textit{External}\ \textit{Global}\ \textit{Pic-lit}\ \textit{Opt-is}\ \textit{Pic-string}_2\ \textit{Usage}\ \textit{Lead-or-trail}\ \textit{Occurs}\ \textit{Synchronize}\ \textit{Justify}\ \textit{Blank}\ \textit{Dd-value}\ .\ \textit{Data-descr-entry}) = \text{add}(\textit{Id-name-or-filler}, \textit{Pic-string}_2, \textit{Tenv})}$$

[ma3] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv},\quad \text{add}(\textit{Id-name-or-filler}, , \textit{Tenv}) = \textit{Tenv}_1}{\text{make-env}(\textit{Level-number}\ \textit{Id-name-or-filler}\ \textit{Redefines}\ \textit{Global}\ \textit{Usage}\ \textit{Lead-or-trail}\ \textit{Occurs}\ \textit{Synchronize}\ \textit{Justify}\ \textit{Blank}\ \textit{Dd-value}\ .\ \textit{Data-descr-entry}) = \textit{Tenv}_1}$$

[ma4] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env}(66\ \textit{Id-name}\ \text{RENAMES}\ \textit{Id-name}\ \textit{Dd-through}\ .\ \textit{Data-descr-entry}) = \text{add}(\textit{Id-name}, , \textit{Tenv})}$$

[ma5] $$\frac{\text{make-env}(\textit{Data-descr-entry}) = \textit{Tenv}}{\text{make-env}(88\ \textit{Id-name}\ \textit{Dd-values}\ \textit{Arg}{-}_{88}\ .\ \textit{Data-descr-entry}) = \text{add}(\textit{Id-name}, , \textit{Tenv})}$$

[ha1] has-dates([]) = false

[ha2] has-dates([*Id-fil-lit* : *Pic-string* : 2# *Tenv-elm-list*]) = true
[ha3] has-dates([*Id-fil-lit* : *Pic-string* : 3# *Tenv-elm-list*]) = true

[ha4] has-dates([*Id-fil-lit* : *Pic-string* : 1# *Tenv-elm-list*])
= has-dates([*Tenv-elm-list*])

$$\text{[ad1]}\ \frac{\text{is-date2}(\textit{Id-fil-lit}) \mid \text{is-date}(\textit{Pic-string}) \ = \ \text{true}}{\begin{array}{l}\text{add}(\textit{Id-fil-lit},\ \textit{Pic-string},\ [\textit{Tenv-elm-list}]) \ = \\ [\textit{Id-fil-lit} : \textit{Pic-string} : 2\#\ \textit{Tenv-elm-list}]\end{array}}$$

$$\text{[ad2]}\ \frac{\text{is-date2}(\textit{Id-fil-lit}) \mid \text{is-date}(\textit{Pic-string}) \ = \ \text{false}}{\begin{array}{l}\text{add}(\textit{Id-fil-lit},\ \textit{Pic-string},\ [\textit{Tenv-elm-list}]) \ = \\ [\textit{Id-fil-lit} : \textit{Pic-string} : 1\#\ \textit{Tenv-elm-list}]\end{array}}$$

[is1] is-date($\textit{Pic-string}_2$) = is-date1($\textit{Pic-string}_2$)
[is2] is-date() = false

This is just a first try. For example, the symbol `V` also indicates a period, and `9` can be represented by `X` as well.

[is10] is-date1(`99X99X99`) = true
[is11] is-date1(`99X99X9999`) = true
[is12] is-date1(`9999X99X99`) = true
[is13] is-date1(`99/99/99`) = true
[is14] is-date1(`99/99/9999`) = true
[is15] is-date1(`9999/99/99`) = true
[is16] is-date1(`99.99.99`) = true
[is17] is-date1(`99.99.9999`) = true
[is18] is-date1(`9999.99.99`) = true
[is19] is-date1(9 9) = true
[is20] is-date1(9 9 9 9) = true
[is21] is-date1(9 9 9 9 9 9) = true
[is22] is-date1(9 9 9 9 9 9 9 9) = true
[is23] is-date1(`9(6)`) = true
[is24] is-date1(`9(8)`) = true

[is30] is-date2(`CENT`) = true
[is31] is-date2(FILLER) = false

[is50] is-date1($\textit{Pic-string}_2$) = false

$$\text{[is51]}\ \frac{\textit{Id-fil-lit} \neq \texttt{CENT}}{\text{is-date2}(\textit{Id-fil-lit}) \ = \ \text{false}}$$

C.2.2 Booleans

imports Layout
We list the booleans here since we refer to it in the thesis; the module `integers` has been left out.
exports
sorts BOOL
lexical syntax
[␣\t\n] → LAYOUT
"%%"~[\n]∗"\n" → LAYOUT

```
    "%"~[%\n]+"%"  → LAYOUT
  context-free syntax
    true                   → BOOL
    false                  → BOOL
    BOOL "|" BOOL          → BOOL  {left}
    BOOL "&" BOOL          → BOOL  {left}
    not "(" BOOL ")"       → BOOL
    "(" BOOL ")"           → BOOL  {bracket}
  variables
    Bool [0-9']* → BOOL
  priorities
    "|" < "&"

equations

[1]   true | Bool = true
[2]   false | Bool = Bool

[3]   true & Bool = Bool
[4]   false & Bool = false

[5]   not(false) = true
[6]   not(true) = false
```

Bibliography

[1] The ibm year 2000 world wide web pages. Internet. http://-www.s390.hosting.ibm.com/stories/tran2000.html.

[2] Peritus year 2000 world wide web pages. Internet. http://www.peritus.com/-1c.htm.

[3] Review of the great cobol debate. Internet. http://www.cobol.org/-welcome2.htm.

[4] *ANSI COBOL 650*, 1990. Found on the internet.

[5] *The Year 2000 and 2-digit Dates: A Guide for Planning and Implementation.* Number GC28-1251-02. IBM, third edition, 1996.

[6] Technical Committee ISO/TC 154. *Data Elements and Interchange Formats – Information Interchange – Representation of Dates and Times.* ISO, 1988.

[7] American National Standards Institute. *ANSI Programming Language - COBOL*, number ANSI X3.23-1968, New York, NY, 1968.

[8] American National Standards Institute. *ANSI Programming Language - COBOL*, number ANSI X3.23-1974, New York, NY, 1974.

[9] American National Standards Institute. *ANSI Programming Language - COBOL*, number ANSI X3.23-1985 / ISO 1989-1985, New York, NY, 1985.

[10] American National Standards Institute. *ANSI Representation for Calendar Date and Ordinal Date for Information Exchange*, number ANSI X3.30 1985, New York, NY, 1985.

[11] American National Standards Institute. *ANSI COBOL Standard Modified for the Year 2000*, 1990.

[12] P. Baumann. Beauty and the beast or a formal semantic description of the control constructs of cobol and its implementation. Technical Report 93.39, Universität Zürich, 1993.

[13] P. Baumann. Semantics-based reverse engineering. Technical report, Universität Zürich, 1994.

[14] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[15] H. Bosma and A. Sellink. A small grammar containing cobol. To be published, 1996.

[16] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Co., 1996.

[17] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. *The syntax definition formalism SDF - reference manual,* version 6 December. 1992. Earlier version in *SIGPLAN Notices*, 24(11):43-75, 1989. ftp://ftp.cwi.nl/pub/gipe/reports/SDFmanual.ps.Z.

[18] IBM. *IBM VS COBOL for OS/VS*, fifth edition, 1986.

[19] IBM. *IBM Migration Guide, release 1 and 2*, 1991, 1995.

[20] P. Klint. A meta environment for generating programming environments. In *ACM Transactions on Software Engineering and Methodology*, pages 176–210. 1993.

[21] I. Peterson. *Fatal Defect Chasing Killer Computer Bugs.* Times Book Random House, 1995.

[22] Martijn Res. A generated programming environment for risla. Master's thesis, 1995.

[23] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. In *ACM Transactions on Software Engineering and Methodology*, volume 5, pages 1–41. 1996.

[24] E. Visser. A family of syntax definition formalisms. In *ASF+SDF95*, pages 89–126, 1995.

Index

Technical Reports of the Programming Research Group

Note: These reports can be obtained using the technical reports overview on our WWW site (`http://www.fwi.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.fwi.uva.nl`, directory `pub/programming-research/reports/`.

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for μCRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*

[P9414] M.G.J. van den Brand and C. Groza. *The algebraic specification of annotated abstract syntax trees.*