FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Dataflow extraction tool for Cobol programming language
Student:	Bc. Andrej Taňkoš
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Study the syntax and semantics of Cobol programming language with a focus on IBM Cobol dialect.
Familiarise yourself with Manta project and its metadata and dataflow representation data structures.
Design a metadata representation scheme for Cobol programming language, design an internal representation of Cobol programming language suitable for a followup dataflow analysis.
Design a dataflow analyzer of Cobol programming language intended to detect dataflows among variables in Cobol programs.
Implement a prototype of a dataflow extraction tool capable of processing Cobol programs with Manta project.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 4, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Dataflow extraction tool for Cobol programming language

Bc. Andrej Taňkoš

Department of Theoretical Computer Science
Supervisor: Ing. Jan Trávníček Ph.D.

January 7, 2021

Acknowledgements

I would like to thank the supervisor of this work, Ing. Jan Trávníček Ph.D., for his help, time, and valuable advice during the work. I would like to thank members of Manta project, namely Mgr. Jiří Toušek for his helpful advice.

Declaration

I hereby declare that I have authored this thesis independently, and that all sources used are declared in accordance with the "Metodický pokyn o etické přípravě vysokoškolských závěrečných prací".

I acknowledge that my thesis (work) is subject to the rights and obligations arising from Act No. 121/2000 Coll., on Copyright and Rights Related to Copyright and on Amendments to Certain Laws (the Copyright Act), as amended, (hereinafter as the "Copyright Act"), in particular § 35, and § 60 of the Copyright Act governing the school work.

With respect to the computer programs that are part of my thesis (work) and with respect to all documentation related to the computer programs ("software"), in accordance with Article 2373 of the Act No. 89/2012 Coll., the Civil Code, I hereby grant a nonexclusive and irrevocable authorisation (license) to use this software, to any and all persons that wish to use the software. Such persons are entitled to use the software in any way without any limitations (including use for-profit purposes). This license is not limited in terms of time, location and quantity, is granted free of charge, and also covers the right to alter or modify the software, combine it with another work, and/or include the software in a collective work.

In Prague on January 7, 2021 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Taňkoš, Andrej. *Dataflow extraction tool for Cobol programming language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

Táto práca sa zaoberá analýzou dátových tokov programovacieho jazyka COBOL, konkrétne IBM COBOL dialektu. Práca najprv skúma rôzne prístupy k analýze dátových tokov, ich reprezentáciu a vizualizáciu v projekte Manta. Následne je jazyk IBM COBOL analyzovaný s cieľom identifikovať dôležité konstrukty jazyka, v ktorých prebieha presun a používanie dát. Práca obsahuje rešerš existujúcich riešení, ktoré by pomohli pri syntaktickej analýze jazyka COBOL. Na základe existujúcich riešení a výsledku analýzy je extraktor dátových tokov navrhnútý a implementovaný pre projekt Manta. Funkčnosť výsledného riešenia je ukázaná na sade testov a príkladov.

Klíčová slova Manta, COBOL, IBM COBOL, extrakcia dátových tokov, analýza dátových tokov, syntaktická analýza

Abstract

This work deals with the data flow analysis of COBOL programming language, specifically IBM COBOL dialect. The work first examines various approaches to data flow analysis, data flow representation and visualization in Manta project. Subsequently, IBM COBOL is analyzed to identify important segments of the language in which data is transferred and used. The work contains research of existing solutions that could help with the syntax analysis of COBOL programming language. Based on existing solutions and results of the analysis, an extraction tool is designed and implemented for the Manta project. The functionality of the resulting solution is shown in a set of tests and examples.

Keywords Manta, COBOL, IBM COBOL, data flow extraction, data flow analysis, parsing

Contents

Introduction **1**
The Goal . . . 2
The Structure of the Thesis . . . 2

1 Background **3**
1.1 Data Flow Analysis . . . 3
1.1.1 Data Flow Graph . . . 4
1.2 MANTA Flow . . . 5
1.3 Static Data Flow Analysis . . . 6
1.3.1 Terminology . . . 7
1.3.1.1 Formal Languages . . . 7
1.3.1.2 Grammars . . . 8
1.3.2 Lexical Analysis . . . 8
1.3.3 Syntax Analysis . . . 9
1.3.4 Semantic Analysis . . . 10
1.3.5 Parser Generators . . . 12
1.3.5.1 ANTLR . . . 12

2 Analysis **15**
2.1 COBOL . . . 15
2.1.1 COBOL Standards, Compilers and Dialects . . . 16
2.2 IBM COBOL . . . 17
2.2.1 Program Structure . . . 18
2.2.2 Identification Division . . . 18
2.2.3 Environment Division . . . 18
2.2.4 Data Division . . . 20
2.2.4.1 Working-Storage, Local-Storage and Linkage Section . . . 20
2.2.4.2 Data Description Entry . . . 21

2.2.4.3 Condition-name (Level 88) . . . 22
2.2.4.4 Data Names . . . 23
2.2.4.5 Data Description Entry's Clauses . . . 23
2.2.4.6 Value Clause . . . 24
2.2.4.7 Data Categories, Data Types, Usage Clause and Picture Clause . . . 24
2.2.4.8 Redefines Clause . . . 30
2.2.4.9 Occurs Clause . . . 31
2.2.4.10 Renames Clause (Level 66) . . . 32
2.2.5 Procedure Division . . . 32
2.2.5.1 Add Statement . . . 36
2.2.5.2 Subtract Statement . . . 37
2.2.5.3 Multiply Statement . . . 38
2.2.5.4 Divide Statement . . . 39
2.2.5.5 Compute Statement . . . 40
2.2.5.6 Move Statement . . . 40
2.2.6 Other COBOL Features . . . 41
2.2.6.1 Subprograms . . . 42
2.2.6.2 Separators . . . 42
2.2.6.3 Identifiers and Qualification . . . 42
2.2.6.4 Subscripting . . . 43
2.2.6.5 Literals . . . 44
2.2.6.6 Source Code Formats . . . 44
2.2.6.7 Copy Statement . . . 46
2.3 Requirements . . . 47
2.4 Existing solutions . . . 47
2.4.1 IBM's VS COBOL II Grammar . . . 47
2.4.2 GnuCOBOL . . . 48
2.4.3 Java Cobol Lexer . . . 48
2.4.4 RES - An Open Cobol To Java Translator . . . 48
2.4.5 TypeCobol . . . 48
2.4.6 ProLeap ANTLR4-based parser for COBOL . . . 48
2.5 ProLeap ANTLR4 COBOL Parser . . . 49
2.5.1 Arithmetic Expressions . . . 49
2.5.2 Ambiguities in Identifiers . . . 49
2.5.3 Nongreedy Subrules . . . 49

3 Design **51**
3.1 Technologies . . . 51
3.1.1 Java . . . 51
3.1.2 Spring Framework . . . 51
3.1.3 Apache Maven . . . 52
3.1.4 JUnit . . . 52
3.1.5 ANTLR . . . 52

3.2 Modules . . . 52
3.2.1 Connector Modules . . . 53
3.2.2 Dataflow Generator Module . . . 54
3.3 Data Entities . . . 55
3.4 Data Types . . . 56
3.5 Redefines and Renames . . . 57

4 Implementation **59**
4.1 Connector Resolver . . . 59
4.1.1 CobolParserServiceImpl . . . 59
4.1.2 CobolPreprocessorImpl . . . 61
4.1.3 IBMDataItemAnalyzer . . . 62
4.1.4 ResScope . . . 62
4.1.5 CobolDataDictionary . . . 62
4.1.6 DataDescriptioItemEntryImpl . . . 63
4.1.7 QualifiedDataNameImpl and DataNameImpl . . . 63
4.2 Dataflow Generator . . . 64
4.2.1 CobolGraphHelper . . . 64
4.2.2 CobolDataFlowVisitor . . . 64

5 Testing **67**
5.1 Connector Testutils . . . 67
5.2 Connector Resolver . . . 67
5.3 Dataflow Generator . . . 68

6 Data Flow Graph Samples **71**
6.1 Simple Program . . . 71
6.2 Sieve of Eratosthenes Program . . . 73

Conclusion **75**
Summary . . . 75
Future Work . . . 76

Bibliography **77**

A Acronyms **81**

B Contents of enclosed CD **83**

List of Figures

1.1 MANTA Flow visualization 6
1.2 Phases of a compiler [1] 7
1.3 The process of a lexical analyzer 9
1.4 A parse tree (a) and an AST (b) of the same statement 10
1.5 A AST/CST combination 11

2.1 A hierarchy of record description entries [2] 22
2.2 COBOL redefines associations 30
2.3 COBOL renames examples [2] 33
2.4 COBOL fixed format [3] 45

3.1 A diagram showing dependencies among modules 53
3.2 A data flow graph for a simple ADD statement 55
3.3 An example of the data entities representation scheme 56
3.4 An example of a data flow between a redefine and its source 57

4.1 The process of the COBOL parser service 60

6.1 A data flow graph of the simple COBOL program visualized by Graphviz 72
6.2 A data flow graph of the simple COBOL program visualized in Manta Flow 73
6.3 A data flow graph in Graphviz for the Sieve of Eratosthenes program 73

List of Tables

2.1 COBOL classes and categories of elementary data items 25
2.2 COBOL usage types 26
2.3 COBOL picture characters 27
2.4 COBOL picture characters, usage types and their sizes 29
2.5 IBM COBOL statements that use data items 35
2.6 Line indicators of fixed format in COBOL 45

Introduction

As the amount of data in the world grows, it is necessary to load, process and store that data. To process the data, many organizations use programming languages which were designed for that exact purpose. COBOL is one such language. Its first definition was produced in 1960 by CODASYL Committee [4], which means it is one of the first computer programming languages. According to a survey of developer skills by HackerRank [5], Cobol is not very well known nor popular programming language among modern developers, but the following facts about Cobol [6] show that it has still huge part in technology of everyday life.

- It powers about 80% of in-person financial services transactions and 95% of ATM swipes.
- On a daily basis, it processes $3 trillion in commerce.
- There are over 220 billion lines of COBOL code and 1.5 billion are written each year.

These facts show that although COBOL is a sixty years old, not very popular programming language, it is still used programming language.

Technologies are still evolving so there will be a time when an old piece of software should be replaced with a new one. The reasons for this can be that the old software is broken or it is working inefficiently. In that case, a replacement of the software can have a huge impact of other systems and that is often not easily identifiable. Manta project with its application – Manta Flow can help with this problem.

Manta Flow is a data flow analysis and visualization tool. Currently, Manta Flow does not support the data flow analysis of COBOL programming language, which is the reason for this work.

The Goal

The main goal of this thesis is to design and implement a prototype of a data flow extraction tool capable of processing IBM COBOL programs with Manta project.

The Structure of the Thesis

Chapter 1 presents theoretical background about data flow analysis process in general, Manta Flow application and how data flow will be analyzed in this work. Chapter 2 contains the analytical part of this work. It presents the analysis of IBM COBOL language with the focus to identify important constructs of the language where data are transferred and used. This chapter also analyzes existing solutions which could help with the syntax analysis of COBOL. The chosen solution is analyzed in this chapter as well. Chapter 3 presents design concepts of this work. This chapter introduces the used technologies and core concepts of the implemented solution. Chapter 4 presents the implementation part of this work. Chapter 5 describes testing methods, which are used to verify the quality of the implemented solution. Chapter 6 shows simple COBOL programs and their data flow graphs extracted by the implemented solution.

Chapter 1

Background

This chapter presents the theoretical background about a data flow analysis and its related terms. The first section introduces the concept of data flow analysis and also describes used data flow representation – data flow graph. The second section explains what MANTA Flow is. The last section is devoted to the data flow analysis process used in this work.

1.1 Data Flow Analysis

Data flow is the sequence in which data transfer, use, and transform during the execution of a computer program [7]. Data flow analysis is a case of program analysis that computes information about the flow of data (i.e., uses and definitions of data) in the analyzed program [8]. Data flow analysis, depending on when it is performed, can be classified into two categories. These two categories are static data flow analysis and dynamic data flow analysis.

Static data flow analysis analyzes the data flow of a program without its execution. Its advantage is that the program does not have to run, and therefore with static analysis it is possible to analyze not only correct programs, but also programs which are faulty, incomplete, or programs which cannot run due to legal reasons. On the other hand, the disadvantage of static analysis is that the analyzed data flow is just an approximation of the behavior of runtime, and therefore less precise.

Dynamic data flow analysis, which is performed while the program is running, does not have this disadvantage. Dynamic data flow analysis is more precise but it adds additional runtime overhead, and it needs the program to run which is not always possible. Both types of analyses have their unique advantages and disadvantages, and therefore they are used for different purposes.

In this work, the data flow is analyzed statically for reasons described later. The other properties which characterize data flow analysis are scope of analysis (global, local, basic block, ...), approach (model checking, abstract

interpretations, ...), flow and context sensitivity, granularity, application (semantic validity, understanding the behaviour, transformation, ...), program representations (ASTs, CFGs, DAGs, ...) and data flow information representation (sets, graphs, trees, ...) [8]. All these properties influence each other, the whole process of data flow analysis as well as the result.

In this work, I'm interested in data flow analysis on the statement level. This is the way Manta project analyzes data flow of programs or scripts and this is also the way of this work. Data flow analysis on statement level analyzes data flow in the context of statement scope (i.e. data flow among variables, function calls, expressions, and so on, in statements). This has a consequence. Control flow of analyzed programs is mostly ignored, and therefore static data flow analysis is more suitable than dynamic analysis. Static data flow analysis is properly explained in Section 1.3.

1.1.1 Data Flow Graph

Data flow graph is a representation of data flow information used in Manta project. Data flow graph is an oriented graph, in which nodes represent data (variables, intermediate results, expressions, ...), operations (assignments, inserts, ...) or structural elements (blocks, statements, ...) and edges represent data flow. Data flow graph is not just a simple graph. Its edges are classified into two categories:

- Direct Flow
- Filter Flow (also called Indirect Flow)

A data flow from source data to target data is called direct if there is a direct change of the target data based on the source data. Direct flow is present in Listing 1 within an assignment statement. In the assignment statement, there is direct data flow going from variables **a** and **b** to variable **c** because **c** is directly changed by values of **a** and **b**.

```
int a, b, c;
c = a + b;
```

Listing 1: A sample C code snippet

As opposite to the direct flow, the indirect flow is present when target data is indirectly influenced by source data. This is shown in Listing 2 where the indirect flow is going from boolean variable **cond** to variable **a**. In the context of the if statement, variable **cond** influences the value of **a** indirectly with the control flow. Indirect flow is commonly called filter flow because it is present in filter clauses or filter statements of programming languages. For example, in WHERE clauses of SELECT statements in SQL dialects.

```
int a;
bool cond = true;

if (cond) a = 1; else a = 0;
```

Listing 2: A sample C code snippet (2)

It is possible to have data flow of both types between two nodes. In that case, the direct flow, as the more influencing flow, overrides the filter flow.

Structural nodes of the data flow graph have a tree-like structure. The tree-like structure is used for better separation between data elements as well as for a preservation of original hierarchy of an analyzed program. Parent nodes of data elements represent structural elements such as block scopes, program scopes, statement identifiers, and so on. It is important to note that data flow is present only in leaf nodes (data) of the tree structure and never in parent nodes (structural elements).

Another term closely related to data flow is data lineage. Data lineage is a description of the pathway from the data source to their current location and the alterations made to the data along the pathway [9]. Data lineage is also exactly what can be seen in chosen data flow representation - data flow graph. By merging the data flows of simple operations into a single representation, it is possible to create data lineages which depict pathways of data alternations from data sources to target data locations. Data lineages are commonly used in applications in the sector of business intelligence (BI) and data governance to make conclusions from data movement. They can also help with the analysis how information is used and to track key bits of information that serve a particular purpose [10].

1.2 MANTA Flow

MANTA Flow [11] is a data lineage analysis and visualization tool provided by Manta project. It supports data lineage analysis and metadata extraction for various programming languages and data technologies. The following list shows some of the currently supported technologies: [12]

- Snowflake database
- Hive database
- Oracle database
- IBM DB2 database
- Java programming language
- Apache Pig
- Oracle ODI

Data lineage is visualized in the form of data flow graph, which is the same graph representation introduced in Section 1.1.1. Figure 1.1 shows an example of the data lineage visualization in MANTA Flow.

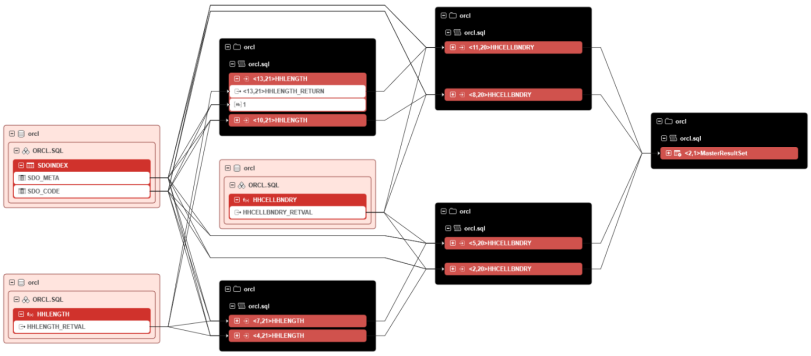

Figure 1.1: MANTA Flow visualization

1.3 Static Data Flow Analysis

Static data flow analysis is a type of data flow analysis where data flow of a program is analyzed statically (i.e. without the program's execution). Static data flow analysis can be done in many ways, but in the end it mostly depends on what kind of data flow should be analyzed and what program representation should be used. In this work, the aim of data flow analysis is to analyze data flow on the statement scope from COBOL programs, precisely from a source code of COBOL programs. Data flow is commonly extracted from internal program representations. This work is using abstract syntax tree (AST) as the internal program representation. Data flow analysis by using AST as the internal program representation has proven as very good solution, which is confirmed by many supported technologies in MANTA Flow application mentioned in Section 1.2.

To create an AST as the internal program representation, I use common techniques from compiler theory. A compiler is a program that reads a program in one language - the source language - and translate it into an equivalent program in another language - the target language [1]. From data flow analysis perspective, I'm not interested in a compiler as a translator from a source language to a target language but rather in some of its individual phases. Figure 1.2 shows the common phases of the compiler. To get program's internal representation, the first three phases are sufficient enough. These three phases are discussed in the following sections. After these phases, the program representation suitable for the data flow analysis and data flow extraction is created. This is discussed in design and implementation chapters of this work.

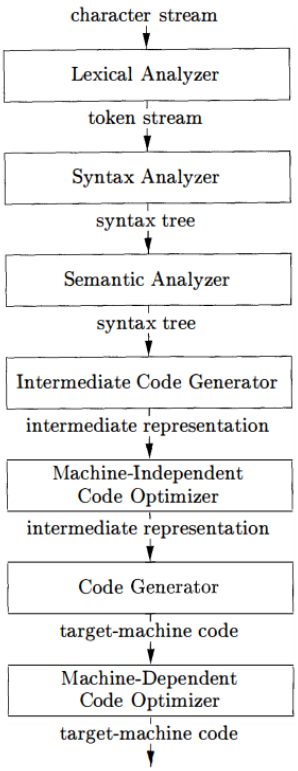

Figure 1.2: Phases of a compiler [1]

1.3.1 Terminology

Before I proceed to the explanation of individual phases, it is necessary to present formal definitions of terms from language processing theory. Definitions are extracted from textbooks on parsing and translations [1] and [13].

1.3.1.1 Formal Languages

An *alphabet* is arbitrary finite nonempty set of elements - symbols. A *string* over alphabet T is a sequence of symbols from alphabet T. The empty sequence is also a string, an *empty string*, and it is commonly denoted ε. The set of all strings over alphabet T is denoted T^*. The set of all non-empty strings over T is denoted T^+. If strings x and y are from T^* then $z = xy$ is a concatenation of string x and y. The length of a string x is denoted $|x|$.

A *formal language* L over an alphabet T is an arbitrary subset of T^*. The product of languages L_1 and L_2 is a language $L = L_1.L_2 = \{xy : x \in L_1 \wedge y \in L_2\}$. The k-th power of a language L over T is defined for each k$\geq$0 as $L^0 = \varepsilon$ and $L^k = L^{k-1}.L$ for k>0. An *iteration* of a language L is language $L^* = \bigcup_{n=0}^{\infty} L^n$. A *positive iteration* of language L is language $L^+ = \bigcup_{n=1}^{\infty} L^n$.

1.3.1.2 Grammars

A *grammar* is a fourtuple $G = (N, T, P, S)$, where N is a finite set of *nonterminal symbols* (nonterminals for short), T is a finite set of *terminal symbols*, $P \subseteq (N \cup T)^*.N.(N \cup T)^* \times (N \cup T)^*$ is a finite set of *rules* (a rule (α, β) from P is often denoted $\alpha \to \beta$), and $S \in N$ is a *starting symbol*.

A *context-free grammar* is a grammar, where rules are of the form $A \to \alpha$, $A \in N$, $\alpha \in (N \cup T)^*$. A *regular grammar* is a grammar, where rules are of the form $A \to a$, $A \to aB$, $A, B \in N$, $a \in T$. The relation $\alpha \Rightarrow \beta \in (N \cup T)^* \times (N \cup T)^*$ is a *derivation* in a grammar G if $\alpha = \gamma X \delta, \beta = \gamma\omega\delta$, $\gamma, \delta, \omega \in (N \cup T)^*, X \in N, X \to \omega \in P$. The replacement of X by ω is called an *expansion*. The k-th power, transitive closure, and reflexive and transitive closure of the derivation relation $\Rightarrow$ are denoted $\Rightarrow^k$, $\Rightarrow^+$, and $\Rightarrow^*$, respectively. String α is called a sentential form in grammar G, if $S \Rightarrow^* \alpha$, $\alpha \in (N \cup T)^*$. If α contains only terminals it is called a *sentence*.

Leftmost derivation is a derivation in which the leftmost nonterminal is expanded. *Rightmost derivation* is a derivation in which the rightmost nonterminal is expanded.

Grammar G is called ambiguous if it allows multiple leftmost derivations or multiple rightmost derivations for same sentence.

Nonterminal A is said to be *left recursive* in grammar G if there exists derivation $A \Rightarrow^+ A\alpha$, $A \in N, \alpha \in (N \cup T)^*$. Grammar G is *left recursive* if it contains at least one left recursive nonterminal.

A language L generated by a grammar G is a set $L(G) = \{x : x \in T^* \wedge S \Rightarrow^* x\}$.

1.3.2 Lexical Analysis

The first phase of a compiler is called lexical analysis or scanning. It is commonly done by a program called lexical analyzer, lexer or lexan. The lexical analyzer reads the stream of input characters (e.g. a COBOL program source code) and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces a pair called a token as output [1]. The first item of the token is the token name and the second item is its attribute value. Figure 1.3 shows the process of a lexical analyzer for a simple COBOL statement.

Whitespace characters and comments are commonly discarded by the lexer. This is done because these elements carry no syntactic purpose for the next phase, but in some cases they can be quite useful. For example, if a target tool wants to create pretty print of the original code, it is necessary to create tokens for all elements of the input stream.

Lexical analysis is generally a simple process because the lexical syntax (i.e. the syntax of tokens) of programming languages can be usually described by a regular grammar. For the regular grammar, a finite-state machine or a

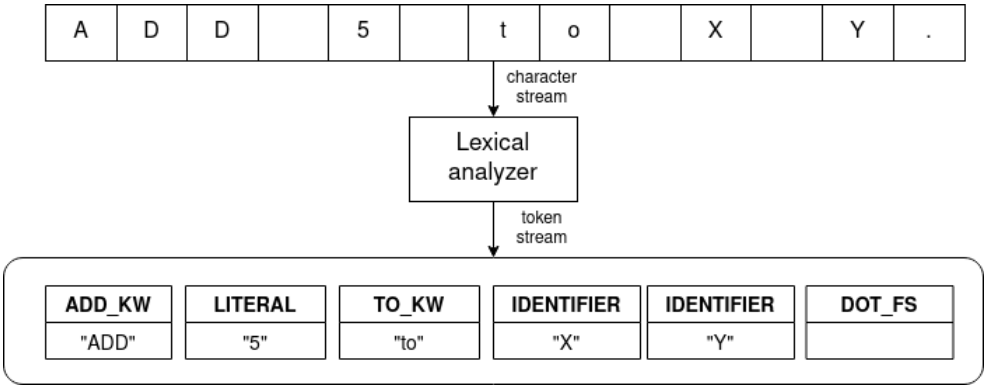

Figure 1.3: The process of a lexical analyzer

set of regular expressions can be created to easily recognize lexemes in the input character stream.

Stream of tokens produced by the lexer is then used as the input for the next compiler phase – the syntax analysis.

1.3.3 Syntax Analysis

The second phase of the compiler is called syntax analysis or parsing. It is done by a program called parser which accepts a token stream produced by the lexer and creates tree-like intermediate representation that depicts the grammatical structure of the accepted token stream [1]. A typical tree-like representation is syntax tree or parse tree.

A parse tree, also called a concrete syntax tree (CST), is created by parsing process and depicts the exact grammatical structure of input tokens according to the used language grammar. On the other hand, an abstract syntax tree (AST, or simply a syntax tree) does not depicts the exact grammatical structure. The AST represents the input syntax more abstractly, and as opposed to the concrete syntax tree it contains only important information. Abstract syntax trees do not commonly contain programming language constructs like parenthesis, commas or semicolons because these constructs are already captured by the tree representation. Figure 1.4 compares both types of trees for the same statement used in the previous section. A parse tree can be easily converted to an AST by a process of removing redundant nodes. The other way around is more complicated.

As already mentioned, this work aims at AST as the internal program representation. AST is more suitable for data flow analysis because most syntax elements of the language are not important for the process of data flow analysis. To be more precise, the representation which is used in this work is not exactly AST. AST contains only important information about the structure of the source program, but it is more difficult to process such

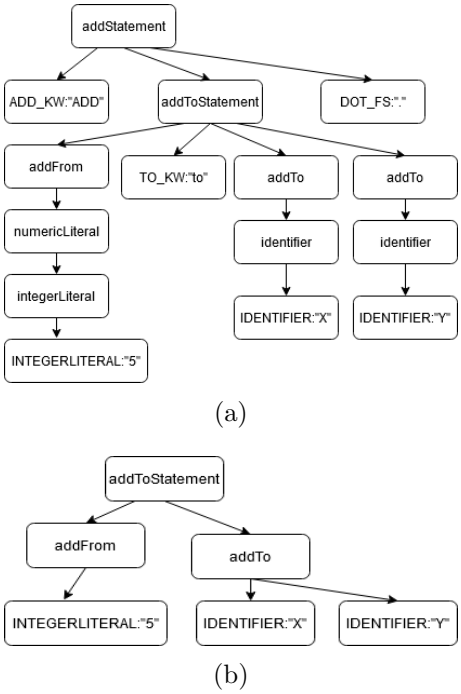

Figure 1.4: A parse tree (a) and an AST (b) of the same statement

a representation. Therefore, used representation is a combination of both – CST and AST, it is more abstract than concrete, but it contains everything important. In the thesis, I still call it an AST. Figure 1.5 depicts such a tree-like representation for the simple COBOL statement used in the previous figures.

As opposed to the lexical analysis, the syntax analysis is more difficult task. Programming languages are not generally regular languages but at least context-free, and therefore they cannot be parsed by regular expressions or finite-state machines. In the case of context-free languages, there are various parsing principles such as top down (LL) parsing or bottom up (LR) parsing. A popular method for creating a parser is to use parser generators. Parser generators are discussed in Section 1.3.5.

1.3.4 Semantic Analysis

Semantic analysis is the third phase of the compiler. This phase is represented by a program called semantic analyzer which checks the tree representation produced by syntax analysis for semantic consistency with the language definition [1]. There are many types of checks which the semantic analyzer is able to perform. A common type of problem checking is called type checking where the semantic analyzer checks each operator for a matching operand. By this type of check, the semantic analyzer is able to detect semantic problems

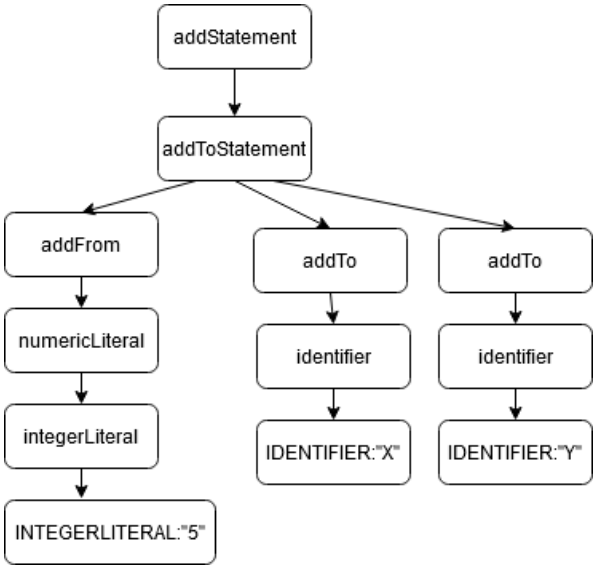

Figure 1.5: A AST/CST combination

such as assignment of a string to an integer variable or using a float number to index an array. These problems are not necessarily problems in every kind of language because some languages can interpret this semantic inconsistency and perform a correction such as casting. The following list mentions other problems which can be recognized by the semantic analyzer:

- undeclared variable
- multiple declarations of same variable in same scope
- accessing an out of scope variable

Semantic problem detection is not the only process done by the semantic analyzer. The semantic analyzer also handles process called resolving, in which it tries to resolve semantic meanings of identifiers. In many programming languages, identifiers can represent objects such as data types, variables, functions, and so on, and the purpose of resolving is to bind those identifiers with their definitions. To do this, the semantic analyzer uses a data structure called symbol table. A symbol table is a data structure which contains information about symbols, their names, locations, types, and so on [1]. Symbol table is filled with information in the whole process of compiling.

In this work, I'm interested especially in resolving because it is an important step in the data flow analysis process. Semantic error detection is not so important for the data flow analysis, because a valid input is expected, and therefore it is mostly ignored.

1.3.5 Parser Generators

A parser generator is a tool that takes a grammar description as input and generates a lexer and a parser implementation that can parse stream of characters using the grammar. Differences between various parser generators are in classes of languages which they can parse and algorithms they are using. The popular parser generators for context-free languages are:

- ANTLR [14] (uses LL(*) and ALL(*) algorithms)
- JavaCC [15] (uses LL(k) algorithm)
- Yacc [16] (uses LALR(1) algorithm)
- GNU Bison [17] (uses LALR(1), LR(1), IELR(1) and GLR algorithms)

An advantage of using a parser generator over a written parser is a time reduction. Parser generators are popular because the input grammar can be written once and a parser generator will resolve everything else. Some parser generators are also powerful enough that they can parse ambiguous grammars. In the case of ANTLR, a user can define special rules called predicates which resolve ambiguities in a deterministic way. Disadvantages of using parser generators are mostly in error detection and recovery. Error messages produced by a generated parser are very simple, and sometimes they are not very helpful. Error recovery in generated parsers is also very limited.

In this work, parser generator ANTLR is going to be used.

1.3.5.1 ANTLR

ANTLR (ANother Tool for Language Recognition) [14] is a powerful parser generator used for processing of a structured text. ANTLR accepts an input context-free grammar in extended Backus-Naur form (EBNF) and generates a lexer and a parser implementation in popular programming languages such as Java, C, C++ or Python. The input grammar can be enriched with ANTLR-specific constructs such as predicates and error handling code to extend possibilities and the scope of grammars which can be parsed by ANTLR. ANTLR is maintained in two versions - ANTLR3 and ANTLR4. In this work, I use only ANTLR4.

ANTLR4 uses an adaptive LL(*) parsing method called ALL(*). The difference from LL(*) method, which is top down (LL) parsing method with arbitrary long lookahead, is in moving grammar analysis phase to parse-time, which lets ALL(*) handle any non-left-recursive context-free grammar [18]. With ANTLR4, it is possible to process also ambiguous grammars. In ambiguous situation, the parser uses a production rule with lowest number (i.e. the first suitable rule) [18]. By this mechanism, production rules can be ordered in the favourable way to get a suitable output. The ANTLR4 grammar structure is quite simple. Lexer rules are initiated by an uppercase letter and parser rules are initiated by a lowercase letter.

ANTLR4 can process direct left-recursive rules in some situations. It uses grammar rules rewriting method to rewrite left-recursive rules to non-left-recursive versions internally [18]. A user can then write ANTLR4 grammar rules (e.g. ANTLR4 expressions grammar rules) in left-recursive form and ANTLR4 can handle it.

The output of the ANTLR4 parser is a parse tree representing the structure of the input text. The parse tree is represented by common classes from ANTLR4 framework and the same tree can be transformed into more suitable representation by a visitor pattern or listeners from ANTLR4 framework. Listing 3 shows a simple ANTLR4 grammar for parsing of arithmetic expressions composed of integers, plus and minus operators.

```
grammar example;

// parser rules
arithmeticExpression
    : INTEGERLITERAL
      ((MINUSCHAR | PLUSCHAR) INTEGERLITERAL)*
      EOF
    ;

// lexer rules
MINUSCHAR : '-';
PLUSCHAR : '+';
INTEGERLITERAL : [0-9]+;
```

Listing 3: An ANTLR4 grammar

Chapter 2

Analysis

This chapter presents the analytical part of this work. The first topic of study is COBOL programming language and one of its dialects - IBM COBOL. COBOL and IBM COBOL are analyzed in the first two sections. The next section presents requirements on the solution by Manta project. This is followed by a section about existing solutions, which could help with the syntax analysis of IBM COBOL. The chosen solution is discussed in the fourth section.

2.1 COBOL

Common Business Oriented Language, COBOL or simply Cobol is a compiled computer programming language designed for business use. Its first definition was produced in 1960 by CODASYL Committee [4], which means it is one of the first computer programming languages.

One of the key features of Cobol is its English-like syntax. Thanks to this feature, programs written in Cobol are known to be self-documenting and easy-to-read even by non-programmers. COBOL programs are structured into several types of segments (similar to books) such as sections, paragraphs or sentences, making a COBOL code to be read and written as a written English text. However, such design led Cobol to also have several disadvantages. For example, a Cobol code is very verbose, due to using whole English words for even simple operations and language constructs, making it longer than it would be in other programming languages.

Another unique feature of COBOL is its strict program structure. COBOL standards define a structure of COBOL programs precisely, where a program is divided into segments and subsegments. Each segment has its own unique name, type, purpose and also a set of rules specifying what other segments it can contain as well as their specific order, if there is any. For example, statements are placed into segment called PROCEDURE DIVISION, data

definitions are placed into DATA DIVISION, and DATA DIVISION must precede PROCEDURE DIVISION in a COBOL source code.

2.1.1 COBOL Standards, Compilers and Dialects

In its lifetime, Cobol was updated continuously and multiple Cobol specifications and standards were created and published. The most known COBOL standards have been produced in 1968, 1974 and 1985 by standardization organization ANSI, and in 2002 by ISO [4]. These Cobol standards have been introduced to resolve incompatibilities between Cobol versions, to resolve problems from older versions as well as to add new features to the language [4]. They are commonly known as COBOL 68, COBOL 74, COBOL 85, and COBOL 2002, respectively. The newest COBOL version is named COBOL 2014, but it is not very popular.

Standards have optional parts and implementers can choose if they want to support them. There are also non-standard features for COBOL. Many COBOL implementers deliver compilers supporting variations or extensions of the language, supporting standard and non-standard features. Such a flavour of the language is not exactly the same thing as the original language even though its core is the same, and thus it is often called a programming language dialect. Different COBOL dialects are often not compatible because they support different features.

To sum it up, Cobol is a successful compiled programming language which exists in various versions, supporting different standards and features. So it is natural to assume that there are multiple Cobol compilers supporting different versions and standards. During the years, many Cobol compilers were discontinued but there are still compilers which are used today. The following list mentions the major ones:

- GnuCOBOL [19]
- IBM COBOL [20]
- Micro Focus COBOL [21]
- Fujitsu NetCOBOL [22]

GnuCOBOL is known to be one of a few Cobol compilers distributed with an open-source license. According to [19], GnuCOBOL implements a substantial part of the COBOL 85, COBOL 2002 and COBOL 2014 standards as well as many extensions included in the other COBOL compilers (IBM COBOL, MicroFocus COBOL, ACUCOBOL-GT and others). This is quite handy because many Cobol compilers such as IBM COBOL or Micro Focus COBOL are propiertary products, so they cannot be used nor analyzed freely. GnuCOBOL supports popular PC operating systems such as GNU/Linux, Mac OS X and Microsoft Windows. After personal experience with this compiler, I can say that only simpler constructs of COBOL are supported and those

more advanced are not. But because this compiler is also the best open-source solution out there, I use it in analytical parts of the thesis anyway. GnuCOBOL is used in the form of OpenCobolIDE 4.7.6.

Another COBOL compiler, which seems to be the most popular COBOL compiler, is IBM COBOL. IBM COBOL is a proprietary product which targets enterprise systems (IBM mainframes). To be more precise, this compiler is not just one compiler but a family of Cobol compilers produced by IBM. IBM has introduced multiple Cobol compilers implementing different versions and standards during the years [23] [24]. IBM has also many COBOL dialects with the most recent one called Enterprise COBOL [23].

The last two COBOL compilers – Micro Focus COBOL and Fujitsu NetCOBOL also belong to the category of proprietary COBOL compilers. They are not beneficial to purposes of this work so I will not discuss them here.

Due to multiple specifications, multiple versions of language coming in various flavours, it is not an easy task to cover all variants of language, even though the most of them support the same core of the language. Therefore, this work is focused only on the most popular Cobol family - IBM COBOL and only one of its dialects - Enterprise COBOL for z/OS Version 6 which is also the newest one from IBM COBOL family. As the following text is focused only on this language dialect, I will refer to it by IBM COBOL or simply just COBOL unless otherwise specified.

2.2 IBM COBOL

IBM industry specification of Enterprise COBOL for z/OS Version 6.3 [25] precisely defines what specifications and standards are supported by this language as well as what non-standard features this language includes. It seems like IBM COBOL is very good at standards compliance and there is just a one small restriction related to common standards. IBM Cobol targets at support of COBOL 85 with partial support of COBOL 2002 and COBOL 2014 standards. It also includes additional modules for intrinsic functions and 7-bit coded character set. These standards, mainly COBOL 85, are commonly supported by other major COBOL implementers [19] [26] [27], so it is very convenient to aim at them.

The next parts of this section are devoted to the documentation of features of IBM COBOL. The most information is extracted from one single source - Enterprise COBOL for z/OS Version 6.3 Language Reference [2]. It is important to note that not every part of the language is analyzed. Language documentation contains about 730 pages so it is not convenient to process everything from the language. Instead, this work is focused only on general concepts of COBOL and also on parts of the language which are the most important for data flow analysis and extraction.

2.2.1 Program Structure

COBOL programs are known for their unique structure which is strictly defined by the language. A sample COBOL program used in Listing 4 demonstrates this unique structure. COBOL describes various types of divisions, paragraphs and sections which can be used in a program code. Those divisions, paragraphs and sections are used for many things such as a configuration of environment, setting program properties, definitions of data items, statements usages, and so on. COBOL describes their exact order, if there is any, and where the corresponding elements can appear. In the following text, I will follow the order of these elements while I will be explaining them, but I will not address their exact order everywhere as it is mostly irrelevant for this work. For an interested reader, the structure and the order of these elements is nicely described in the language reference.

Before I proceed, it is important to mention that COBOL keywords and identifiers are case-insensitive, even in user-defined words, so **IDENTIFICATION** and **identification** represent same keyword. However, all of them are commonly written in uppercase letters so I will write them like that too.

COBOL programs are composed of building blocks called divisions. There are four divisions which can be defined within a COBOL program:

1. IDENTIFICATION DIVISION
2. ENVIRONMENT DIVISION
3. DATA DIVISION
4. PROCEDURE DIVISION

2.2.2 Identification Division

The IDENTIFICATION DIVISION is the only single division which is mandatory for every COBOL program. It has to contain mandatory paragraph named PROGRAM-ID in which a programmer specifies a name of the COBOL program and optionally, program attributes. The other paragraphs of this division only serve as fields for documentation.

2.2.3 Environment Division

The ENVIRONMENT DIVISION is the second division of COBOL programs. It serves as a place for describing computer environment and input-output resources of COBOL programs. This division is a place for two sections - the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION.

The first section is not important for data flow analysis so I skip it entirely. The second section is related to input-output properties of COBOL programs. These properties are somewhat important for data flow analysis but because COBOL is an extensive language and the target of this work is on the data flow analysis between variables, I will not cover them in this work.

```
       Identification Division.
       Program-id.    AWIXMP.
       Data Division.
       Working-Storage Section.
       01   Feedback.
        02   Fb-severity      PIC 9(4) Binary.
        02   Fb-detail        PIC X(10).
       77   Dest-output       PIC S9(9) Binary.
       77   Lildate           PIC S9(9) Binary.
       77   Lilsecs           COMP-2.
       77   Greg              PIC X(17).
       01   Pattern.
        02                    PIC 9(4) Binary Value 45.
        02                    PIC X(45) Value
            "Today is Wwwwwwwwwwwz, Mmmmmmmmmz ZD, YYYY.".
       77   Start-Msg         PIC X(80) Value
            "Callable Service example starting.".
       77   Ending-Msg        PIC X(80) Value
            "Callable Service example ending.".
       01 Msg.
         02 Stringlen         PIC S9(4) Binary.
         02 Str               .
          03                  PIC X Occurs 1 to 80 times
                                    Depending on Stringlen.
       Procedure Division.
       000-Main-Logic-Paragraph.
           Perform 100-Say-Hello-Paragraph.
           Perform 300-Say-Goodbye-Paragraph.
           Stop Run.
       100-Say-Hello-Paragraph.
           Move 80 to Stringlen.
           Move 02 to Dest-output.
           Move Start-Msg to Str.
           CALL "CEEMOUT" Using Msg   Dest-output Feedback.
           Move Spaces to Str.        CALL "CEEMOUT"
                Using Msg Dest-output Feedback.
       300-Say-Goodbye-Paragraph.
           Move Ending-Msg to Str.
           CALL "CEEMOUT" Using Msg Dest-output Feedback.
       End program AWIXMP.
```

Listing 4: A sample COBOL program demonstrating program structure [28]

2.2.4 Data Division

The DATA DIVISION is a segment of COBOL programs where data definitions belong. This division can contain four sections:

1. FILE SECTION
2. WORKING-STORAGE SECTION
3. LOCAL-STORAGE SECTION
4. LINKAGE SECTION

These sections are a place for variable definitions which are called description entries. There are two types of description entry:

- file description entry
- data description entry

COBOL uses an uncommon terminology where variables are not called variables but items. In the case of a variable representing a reference to a file it is called a file item or simply a file, and in case of a plain variable it is a data item. I will use both terminologies in this work.

The first section is the FILE SECTION and it is used as the place for definitions of working structures of files. I will not cover files in this work so I skip this section entirely.

```
       Working-Storage Section.
       01 PERSON-INFO.                *> data description
          02 FULL-NAME.                *> data description
               03 NAME     PIC X(10). *> data description
               03 SURNAME  PIC X(10). *> data description
          02 AGE           PIC X.     *> data description
          02 ADDRESS-INFO.             *> data description
               03 STREET   PIC X(10). *> data description
               03 CITY     PIC X(10). *> data description
       01 COMPANY-INFO     PIC X(80). *> data description
```

Listing 5: A COBOL code snippet demonstrating description entries

2.2.4.1 Working-Storage, Local-Storage and Linkage Section

The WORKING-STORAGE SECTION and the LOCAL-STORAGE SECTION are sections of COBOL programs where data definitions, in form of data description entries, are stored. Difference between those two sections is that variables defined in the working-storage section stay persistent across the program calls whereas variables defined in the local-storage section are allocated and freed on a program per-invocation basis. Data definitions stored

in these sections are very similar to what are in other languages called global variables definitions. Data items (variables) defined in these sections can be used in various types of statements such as arithmetic statements or data movement statements as a source and/or a destination of a computation.

The last section of the Data Division is called the LINKAGE SECTION. This section defines data items which are available from another program, meaning that the storage of data is not reserved within the actual program but elsewhere. The typical case for using data items defined by entries within the LINKAGE SECTION is as program parameters.

The following text explains data description entries, their structures and clauses. I will use Listing 5 for an easier grasp of them, where every data description entry is marked by the floating comment $*>$. Although data description entries are quite similar in different sections, there are still some differences. For example, the EXTERNAL clause cannot be used in entries in the local-storage section nor entries in the linkage section. Because these differences are not important for the thesis, I will not address them unless it is necessary.

2.2.4.2 Data Description Entry

A data description entry is a definition of a data item in COBOL. Its first part is called a level number. Level numbers are used to define a hierarchy of variables, where the level number 01 is the signalization of a top-level parent, and every entry with bigger level number defined below it has some sort of relationship with it. Data items create a tree-like structure, in which subordinate entries are indicated by increased level number. This is demonstrated in Listing 5, where PERSON-INFO, as a top-level entry, has subordinate (child) entries FULL-NAME, AGE and ADDRESS-INFO. FULL-NAME entry is also subdivided and it has subordinate entries NAME and SURNAME. It is important to note that an indentation of a COBOL program does not mean anything for this structure and its purpose is purely for better readability. Level numbers can range from 01 to 49 and there is no rule which states they must be incremented by 1, though, it is typical to increment levels by 5. Level numbers 66, 77 and 88 are used for special purposes.

Figure 2.1 shows more visually how entries are subdivided within a hierarchy. Important thing to mention is that only bottom entries (leaves) represent actual variables holding data. Variables defined by those entries are called elementary data items or elementary items, and they are organized sequentially in the memory according to definition order. Because data is organized sequentially, it is possible to reference multiple entries together.

Non-leaf data items are commonly called groups, group items or records, and they define mostly structural properties of the hierarchy. Data description entries describing records are commonly called record description entries. A

record description entry can also contain clauses, which are then reflected on its subordinate data description entries.

The level number 77 identifies a data item as a pure elementary data item so this is the way to explicitly specify that this data item is a standalone variable.

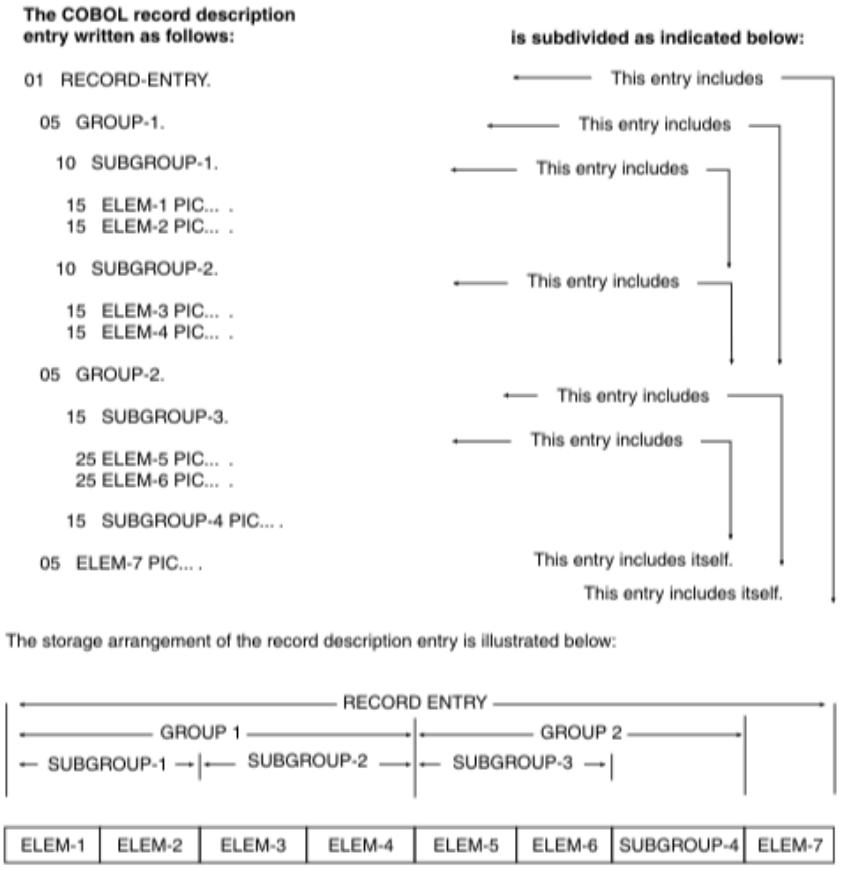

Figure 2.1: A hierarchy of record description entries [2]

2.2.4.3 Condition-name (Level 88)

Condition-names are a special case of data description entries with level number 88. A condition-name associates a value, a set of values, or a range of values with variable named conditional variable and signalizes a state of that variable. Condition-names always belong to some conditional variable, defined by a data description entry, and thus they must be defined immediately after that entry. The value clause is mandatory for condition-names, and it is only possible clause for condition-names.

Listing 6 illustrates syntax and usage of condition-names. In the example, there is a conditional variable named **AGE**. **AGE** has 4 condition-names which, depending on the value of **AGE**, determine true or false state. They can be seen as a some kind of boolean variables which depend on the value of another variable. Statements, such as if statements, can then use those condition-names in their condition expressions to influence control flow of program. It is important to note that condition-names are special items which don't hold actual data, so they are not taken as leaf entries.

```
01  AGE                PIC  99.
    88  INFANT         VALUE 0.
    88  BABY           VALUE 1, 2.
    88  CHILD          VALUE 3 THROUGH  12.
    88  TEENAGER       VALUE 13 THRU  17.

IF INFANT   DISPLAY 'INFANT'.
IF BABY     DISPLAY 'BABY'.
IF CHILD    DISPLAY 'CHILD'.
IF TEENAGER DISPLAY 'TEENAGER'.
```

Listing 6: A COBOL code snippet demonstrating condition-names

2.2.4.4 Data Names

Data names represent identifiers (user-defined words) for variables defined by data description entries. Names are placed after an entry's level number. In COBOL, it is possible to define a variable without a user-defined data name. Those variables are called fillers and they are identified either with keyword FILLER in the place of a data name or by omitting the data name at all. Variables defined like this are still present within data hierarchy but there is no explicit reference to them. However, they can be used in conjunction with their parent data names or by techniques such as redefinition. Data names are case-insensitive.

2.2.4.5 Data Description Entry's Clauses

Data description entry's clauses are clauses which can be present in a data description entry. Some clauses are deprecated, some are unused, some are irrelevant for the thesis but there are still some clauses which are important. These clauses are:

- Redefines clause
- Renames clause

- Occurs clause
- Picture clause
- Usage clause
- Value clause

2.2.4.6 Value Clause

The value clause specifies the initial contents of a data item. In a case of condition-names this clause specifies a value, a set of values, or a range of values which are associated with the condition-name. Only literals can be specified as initial values in COBOL. If the value clause is specified for a group item then the value clause is applied correspondingly on subordinate data items. The value clause can also be applied on data description entries representing arrays. In that case, all elements of an array are initialized to given initial value.

2.2.4.7 Data Categories, Data Types, Usage Clause and Picture Clause

The two clauses which are used to describe the data type of a data item are the usage clause and the picture clause. To be more precise, COBOL does not have traditional data types which are known in modern programming languages, such as integers or strings. Instead, COBOL assigns data items into data classes and data categories. I will demonstrate classes and categories by their enumeration which is present in Table 2.1.

The category of a data item is established by its attributes. For example, a data item which represents alphanumeric string belongs to category alphanumeric. Among all categories listed in the table, most of them are pretty self-explanatory. DBCS stands for double byte character set and describes string in which every character has 2 bytes. National category describes data items representing data in extended encoding. For IBM COBOL, it is UTF-16. Data items belonging to categories ending with edited have special properties. And the inner representation of a floating-point data item decides if it is internal or external. On the other hand, classes are simpler groupings of data categories. For a data item with class numeric, it does not matter if it is a floating-point number or a decimal number. It is important to note that all these classes and categories apply only to elementary data items.

Group items are divided into less categories (alphanumeric, national and UTF-8), and implicitly a group item belongs to the alphanumeric category. Literals and functions, depending on their return values, also belong to some categories.

Classes and categories are very important for COBOL because many statements operate differently depending on the categories of their arguments. Categories of elementary data items can be seen as data types and therefore, I

Class	Category	Usage Type
Alphabetic	Alphabetic	DISPLAY
Alphanumeric	Alphanumeric	DISPLAY
	Alphanumeric-edited	DISPLAY
	Numeric-edited	DISPLAY
DBCS	DBCS	DISPLAY-1
National	National	NATIONAL
	National-edited	NATIONAL
	Numeric-edited	NATIONAL
UTF-8	UTF-8	UTF-8
Numeric	Numeric	DISPLAY (zoned decimal)
		NATIONAL (national dec.)
		COMP-3 (internal decimal)
		PACKED-DECIMAL (inter.)
		BINARY
		COMP
		COMP-4
		COMP-5
Numeric	Internal floating-point	COMP-1
	Internal floating-point	COMP-2
	External floating-point	DISPLAY
	External floating-point	NATIONAL

Table 2.1: COBOL classes and categories of elementary data items

will recognize them as such. In the case of groups, literals and functions, categories are not so important so I will ignore them.

Table 2.1 has shown that multiple usage types are assigned to same classes and categories. Usage types are used in the usage clause, and they specify the format in which data are represented in storage. Usage types are listed in Table 2.2, which also contains a simple description of each type. Usage clause is commonly specified only for elementary data items but it is possible to specify it also for group items. In that case, the group item usage clause is applied to elementary data items in that group but there is a rule which states that usage clauses of elementary data items cannot contradict the group usage clause. COBOL documentation does not specify if a group usage clause is applied also on data items which are more deeply in the hierarchy. In GnuCOBOL, the usage clause is applied on every subordinate data item until redefined by another usage clause. This is demonstrated in Listing 7, and I will assume it works same for IBM COBOL.

If an elementary data item nor its parent have usage type, the elementary data item has implicitly defined usage type. Which usage type it is depends

Usage Type	Description
BINARY	Two's complement binary number.
COMP/COMPUTATIONAL	Same as binary type.
COMP-1/COMPUTATIONAL-1	Single precision internal floating-point number.
COMP-2/COMPUTATIONAL-2	Double precision internal floating-point number.
COMP-3/COMPUTATIONAL-3	Number in packed-decimal format.
COMP-4/COMPUTATIONAL-4	Same as binary type.
COMP-5/COMPUTATIONAL-5	Native binary number.
DISPLAY	String with 1 byte per character.
DISPLAY-1	Double-byte character set (DBCS) string.
INDEX	Index to tables (arrays).
NATIONAL	UTF-16 string.
UTF-8	UTF-8 string.
OBJECT REFERENCE	Reference to object.
PACKED-DECIMAL	Number in packed-decimal format.
POINTER	32-bit or 64-bit data pointer. (platform-dependent)
POINTER-32	Strictly 32-bit data pointer.
PROCEDURE-POINTER	Strictly 32-bit procedure pointer.
FUNCTION-POINTER	32-bit or 64-bit procedure pointer. (platform-dependent)

Table 2.2: COBOL usage types

```
       01 X USAGE POINTER.
           05 Y.
               10 Z. <* pointer

       01 A USAGE POINTER.
           05 B USAGE DISPLAY.
               10 C PICTURE X. <* display
           05 B2. <* pointer
```

Listing 7: A COBOL code snippet demonstrating usage type inheritance

on its picture clause. Categories and usage types are closely related but there is an exception. Not every data item with usage type has a category. This can be seen in previously mentioned tables, in usage types such as INDEX or POINTER which are not assigned to any category nor class. Data items defined with usage types as these are not categorized by COBOL, and in most cases they are used only with special statements or clauses such as a SET statement or a CALL statement.

Character	**Description**
A	A position for a letter of the Latin alphabet or a space.
E	Marks the starting point of the exponent.
G	A DBCS character position.
N	A DBCS or a national character position.
P	An assumed decimal scaling position.
S	An indicator of the presence of a operation sign.
U	A UTF-8 character position.
V	An indicator of the assumed decimal point location.
X	A location of alphanumeric character.
9	A position for numeric character.
.	A position for decimal point.

Table 2.3: COBOL picture characters

Picture clause is clause initiated by keyword PIC or PICTURE, and followed by so-called picture string. A picture string is a string of picture characters describing general characteristics of data items. The picture clause can be specified only for elementary data items. Table 2.3 presents possible picture characters and their meanings. This table does not contain editing picture characters which are used as positions of special characters, giving a better representation of data item content. If a picture string has any editing character then the category of data item has edited suffix.

Picture strings are demonstrated in Listing 8 and explained by the following list:

1. The first picture string can be used to define a data item holding numeric value with up to 3 digits. The data item can then hold values such as 123 or 0.
2. This picture string can be used to define a data item holding a signed numeric value with up to 4 digits.
3. The third picture string is the same as the second one. However, this picture string is written in a shortcut notation. X(n) is shortcut for n-times repetition of character X, so 9(4) is equivalent to 9999.
4. The fourth picture string describes an alphanumeric data item. Its content can be digits, characters as well as a combination of both.

```
1  PIC 999       VALUE 123.      <* numeric
2  PIC S9999     VALUE +7890.    <* numeric
3  PIC S9(4)     VALUE -4560.    <* numeric
4  PIC XX        VALUE "1B".     <* alphanumeric
5  PIC -9.99     VALUE 3.14.     <* external FP
6  PIC A(5)      VALUE "COBOL". <* alphabetic
7  PIC N         VALUE "X".      <* DBCS or national
8  PIC 00099     VALUE 42.       <* numeric-edited; 00042
```

Listing 8: Examples of COBOL picture strings.

5. The fifth example describes a representation of an external floating-point data item.
6. The sixth example shows a representation of a string with alphabetic characters. As opposed to the alphanumeric data item, a data item described with this picture string can contain only alphabetic characters.
7. This example is a system-dependent. Symbol N represents a position for either a DBCS character or a national character. Which one is it depends on the usage clause, compiler options and also on the environment. Fortunately, both characters are 2 bytes long with difference only in character encoding, so in the end it does not matter which one it is.
8. The last case shows the usage of editing characters (zeroes). The special meaning of zeroes is that they are always present at given positions. Editing characters are irrelevant for data flow analysis, so I will not discuss them here.

There are more things to write about picture characters and usage clauses. If an usage clause is not defined for a data description entry and a picture clause is, the usage type of data item is implied from picture characters of the picture clause. Presence of N character in the picture string implies it has national usage type. If the picture string contains character G, it has DBCS type. Otherwise, it has display usage type.

There are also rules stating which usage type cannot have a picture string, or rules stating what picture characters can be present within a data item with given usage type. To have it more simplified, I will not discuss these rules but I will use Table 2.4 for better recognition of categories, picture characters and usage types. The table describes possible combinations of categories, usage types and picture characters for elementary data items in COBOL. The last column of the table describes the size of data item with present picture characters and given usage type.

Usage types (category)	**Picture characters**	**Size (in bytes)**
BINARY/COMP COMP-4/COMP-5 (numeric)	optional symbols S P V; 1-4 digits (9)	2
BINARY/COMP COMP-4/COMP-5 (numeric)	optional symbols S P V; 5-9 digits (9)	4
BINARY/COMP COMP-4/COMP-5 (numeric)	optional symbols S P V; 10-18 digits (9)	8
PACKED-DECIMAL/COMP-3 (numeric)	optional symbols S P V; digits (9)	(count of digits) / 2
COMP-1 (internal floating-point)	no picture clause	4
COMP-2 (internal floating-point)	no picture clause	8
FUNCTION-POINTER INDEX/POINTER OBJECT-REFERENCE	no picture clause	4 or 8 (platform-dependent)
POINTER-32	no picture clause	4
PROCEDURE-POINTER	no picture clause	8
DISPLAY (alphabetic)	only symbols A	1 byte for each symbol
DISPLAY (alphanumeric)	symbols 9 A X, where X is at least once	1 byte for each symbol
DISPLAY (alphanumeric-edited)	symbols 9 A X B 0 \, where X or A is at least once and B, 0 or \ is at least once	1 byte for each symbol
DISPLAY-1 (DBCS)	symbols G and B	2 bytes each symbol
DISPLAY (external floating-point)	symbols 9 + - E . V	1 byte each, V is not counted
NATIONAL (external FP)	symbols 9 + - E . V	2 bytes each, V is not counted
NATIONAL (national)	symbol N	2 bytes each
NATIONAL (national-edited)	symbols N B 0 \, where N is at least once and B, 0 or \ is at least once	2 bytes each
DISPLAY (numeric)	symbols 9 S P V	1 byte each
NATIONAL (numeric)	symbols 9 S P V	2 bytes each digit
DISPLAY (numeric-edited)	symbols B P V Z 9 0 \, . + - CR DB * $	1 byte each; P V not counted
NATIONAL (numeric-edited)	symbols B P V Z 9 0 \, . + - CR DB * $	2 bytes each, P V not counted
UTF-8 (utf-8)	symbols U	4 bytes each

Table 2.4: COBOL picture characters, usage types and their sizes

2.2.4.8 Redefines Clause

The redefines clause allows to use different data description entries to describe same computer storage area. Redefines will be explained by description entries present in Listing 9.

```
05  A PICTURE X(6).
05  B REDEFINES A.
    10 B-1          PICTURE X(2).
    10 B-2          PICTURE 9(4).
05  C REDEFINES B PICTURE 99V999.
```

Listing 9: Data description entries demonstrating redefines

In the example, the first data description entry defines variable **A** representing an alphanumeric data item. The next three data description entries defines a record with redefines clause. It means that this record is a redefinition of another data storage, the storage of variable **A**. The last description entry redefines a storage of **B** but **B** is the redefinition of the storage **A**, so in the end, all data description entries share the very same storage. In the case of **B**, the data item is structured into smaller data items so corresponding elementary items use only parts of that storage. How these data items share storage is illustrated by Figure 2.2.

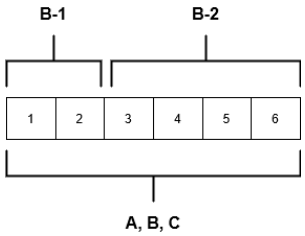

Figure 2.2: COBOL redefines associations

With redefines, it is possible to redefine a group to another group, a group to an elementary data item, an elementary item to a group, or an elementary item to an elementary item. Redefines can also redefine only a part of the source storage.

IBM COBOL allows a redefinition of storage which is smaller than storage specified with the picture and/or usage clause. However, the documentation does not specify what exactly happens in such case. In case of GnuCOBOL, this is not allowed so I will not handle this case in this work.

2.2.4.9 Occurs Clause

Data items described with the occurs clause represent arrays. Arrays are commonly called tables because their structure simulates tables. There are two versions of arrays in COBOL - fixed-length tables and variable-length tables.

Listing 10 shows definitions of arrays in COBOL. In the example, WS-

```
01 WS-TABLE.
  05 WS-A OCCURS 10 TIMES.  <* fixed-length
    10 WS-B.
      15 WS-C PIC X(6) OCCURS 5 TIMES
          INDEXED BY I J. <* fixed-length

01 VAR1 PIC 99 VALUE 4.
01 VAR2 PIC 9999 VALUE 1000.

01 WS-VTABLE OCCURS 1 TO 5
      DEPENDING ON VAR1. <* variable-length
01 WS-VTABLE2 OCCURS 5 TO UNBOUNDED TIMES
      DEPENDING ON VAR2. <* variable-length
```

Listing 10: Data description entries demonstrating the occurs clause

TABLE represents a two-dimensional array where WS-A are rows and WS-C are columns. Each occurrence of WS-A has subordinate WS-B and WC-C, and each occurrence of WS-C is an alphanumeric string of length six. In COBOL, it is possible to define an array up to seven levels in a one structure.

The clause INDEXED BY specifies indices which can be used to index or to subscript the array. Indices are not part of data hierarchy, and therefore they don't represent leaf data items. In IBM COBOL, indices are regarded as private special registers for the use of defining program only. It is not clear if indices can be used only with a table in which they are defined. In GnuCOBOL, it is possible to use indices also with other tables in the same program.

Variable-length arrays are defined with an upper and a lower bound, where actual size depends (the DEPENDING ON clause) on some other variable. It is possible to use keyword UNBOUNDED to specify an array to have unlimited size. However, unbounded arrays can only be used in restricted conditions. IBM documentation does not clearly specify how variable-length arrays are aligned to other data items defined in same structure. It seems like there are multiple cases of the alignment, and the documentation also mentions variably located items which are items following variable-length arrays. In the case of

GnuCOBOL, it seems that variable-length arrays have the maximal defined size (UNBOUNDED is not working) and other items are aligned accordingly. I will handle variable-length arrays in the same way. Unbounded arrays will not be supported.

2.2.4.10 Renames Clause (Level 66)

The RENAMES clause specifies alternative and possibly overlapping groupings of elementary data items. To use the renames clause, it is necessary to describe an elementary data item with special level number 66. The usage of renames is illustrated in Figure 2.3. In the figure, it can be seen that with renames it is possible to create new data items which refer to storage of other data items. Renames are similar to redefines with this feature, but with renames is possible to refer to multiple groupings together.

Renames are structurally bounded to a previous data item with level 01 (i.e. it is its parent), and thus it is not possible to rename entries with level 01. In the example, the parent of **DN-6** is **RECORD-I**. It is also not possible to rename entries with level 66, 77 or 88, but it is possible to define more than one rename for a record description entry. Renames can be used in a version without THROUGH keyword, and in such case, they rename only one specific group/elementary data item to which they refer.

2.2.5 Procedure Division

The PROCEDURE division is a division where a program logic resides. This division starts by PROCEDURE DIVISION keywords, which are followed by a procedure division header and a procedure division body. The procedure division header represents a definition of program input and output parameters and a return value of the program. In this work, I'm working with COBOL programs as separate units so I skip the procedure header entirely.

In the procedure division body, it is possible to define COBOL procedures. A procedure consists of a section or a group of sections, and a paragraph or group of paragraphs. The purpose of these elements is to structure the program into logical segments. This is shown in Listing 11.

Sections are identified with a section name, which is followed by SECTION keyword. Paragraphs are represented only with an user-defined paragraph name. These names don't have to be unique if there is a possibility to qualify these segments uniquely. In the case of sections, this is not possible, but for paragraphs it is.

Sections and paragraphs in the procedure division are optional, so the procedure division can start right away with sentences. Specific COBOL statements such as GO TO or PERFORM can change the control flow of the program by jumping or performing only chosen set of segments. The COBOL program typically starts with the first statement in the procedure division. It

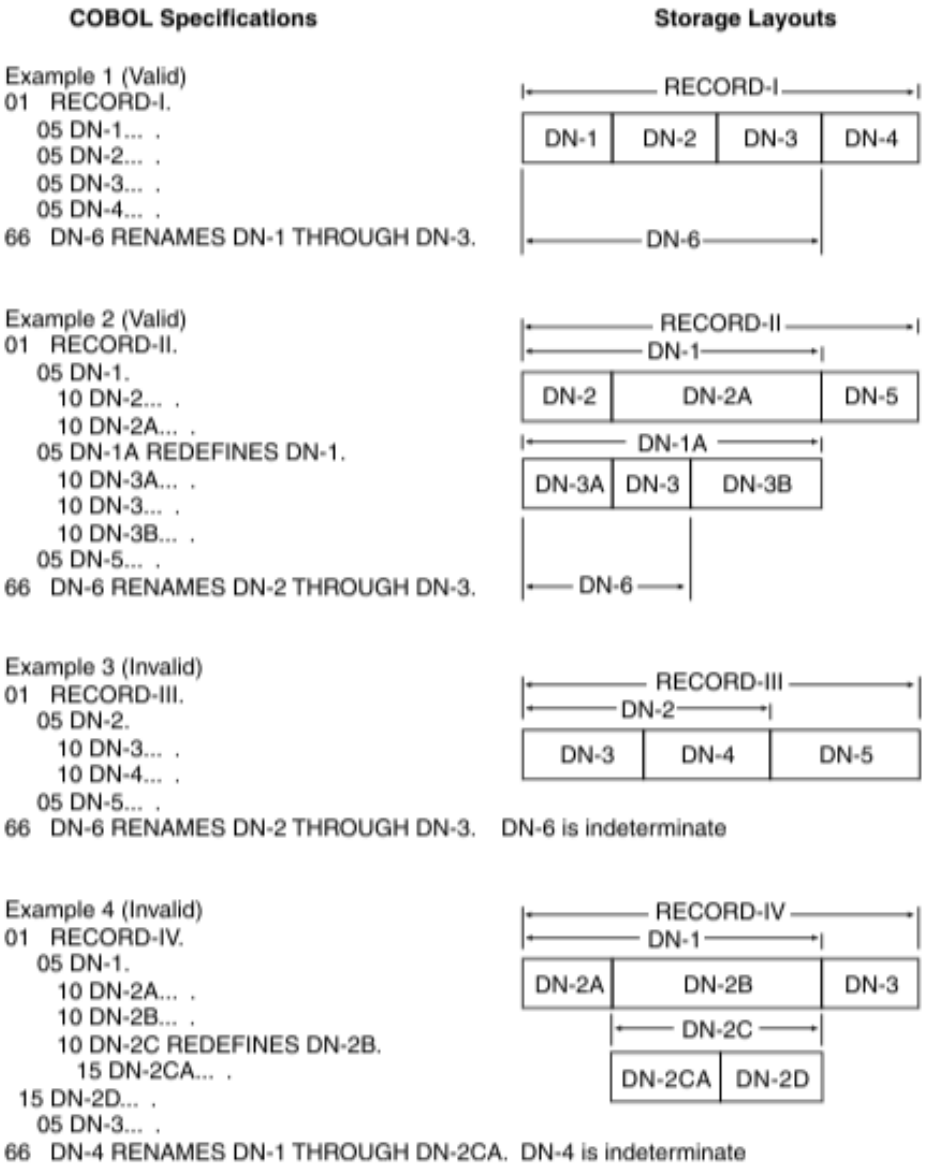

Figure 2.3: COBOL renames examples [2]

```
    PROCEDURE DIVISION. <* procedure division
        P.              <* paragraph
      CBL-SEC SECTION.  <* section
        PARA.           <* paragraph
        PARA2.          <* paragraph
          ADD 5 TO A.   <* sentence
          DISPLAY 5.    <* sentence
          ADD 1 TO X
          DISPLAY X.    <* sentence
      SEC2 SECTION.     <* section
          MOVE B TO C.  <* sentence
      SEC3 SECTION.     <* section
        PARA.           <* paragraph
```

Listing 11: A COBOL code snippet demonstrating procedure division

is important to note that an indentation of a code does not mean anything for the structure of these elements.

Sentences group one or more statements into a single unit. This is demonstrated in Listing 11, where COBOL code contains a few sentences. Statements in these sentences contain reserved keywords so they can be easily identified. Sentences are always ended with separator period.

COBOL supports a wide range of statements. Table 2.5 lists statements of COBOL, which use data items, and therefore data flow is present in them. However, due to the extent of the language, this work targets only to the most used ones:

- Arithmetic
  - ADD
  - SUBTRACT
  - MULTIPLY
  - DIVIDE
  - COMPUTE
- Data movement
  - MOVE

The chosen statements are explained in the subsequent sections. Other statements will not be discussed in this work, however, the same principles of the data flow analysis can still be applied to them, and the implemented solution can be extended with the support for them. There are two additional constructs which are related to statements.

The first construct is called explicit scope terminator. As already noted,

Statement	Description
ACCEPT	Transfers an input data into a variable.
ADD	Adds numeric items.
CALL	Used for calling other programs.
COMPUTE	Computes arithmetic expressions.
DIVIDE	Divides numeric items.
EVALUATE	Performs a switch statement.
IF	Performs an if statement.
INITIALIZE	Sets variables to predetermined values.
INSPECT	Examines characters in a data item.
INVOKE	Invokes a COBOL or Java class.
JSON PARSE	Parses data from JSON format.
JSON GENERATE	Converts data into JSON format.
MOVE	Moves data between data items.
MULTIPLY	Multiplies numeric items.
SET	Performs a set operation.
SEARCH	Searches a table for a specific element.
SORT	Arranges files or tables in a user-specified sequence.
STRING	Strings multiple items together.
SUBTRACT	Subtracts numeric items.
XML GENERATE	Converts data into XML format.
XML PARSE	Parses data from XML format.
UNSTRING	Separates a data item into multiple fields.

Table 2.5: IBM COBOL statements that use data items

COBOL sentences are ended with an explicit terminator period. But in the case of statements, there was no explicit terminator. Consider Listing 12. In

```
    IF NUMC = 1
      DISPLAY '1'
      IF NUMD = 2
          DISPLAY '2'
      ELSE            <* which if has this else branch?
          DISPLAY '3'.
```

Listing 12: A COBOL code snippet demonstrating sentence ambiguity

the example, there are two if statements with only one else branch. Because this code does not use an explicit scope terminator, the code is ambiguous.

IBM COBOL does not explicitly state how these types of sentences are resolved but in the case of GnuCOBOL, the ELSE branch is connected to the closest IF statement. It can be stated that the GnuCOBOL's parser has greedy behaviour. I will expect that IBM COBOL handles this case in the same way. Explicit scope terminators are represented by the keyword END, a hyphen and an operation. For example, explicit scope terminator for IF is END-IF and it can be used to explicitly terminate an IF statement. Other explicit terminators are END-ADD, END-SUBTRACT, END-MOVE, and so on.

The second construct is a conditional statement. A statement is called conditional statement if it contains at least one condition phrase (condition). This is demonstrated in Listing 13. There are many types of conditions in COBOL, such as ON OVERFLOW, ON EXCEPTION, INVALID KEY, and so on. Conditions are used to perform particular actions in response to the result of the associated statement. Conditions are composed of condition keywords and a list of statements. Conditional statements present a similar behaviour as the one mentioned with explicit scope terminators. A condition can contain multiple statements so everything following condition keywords belong to the condition until an explicit terminator, such as period or END keyword, occurs. Statements which are not conditional are commonly called

```
ADD 1000 TO NUM                      <* (cond.) statement
  ON SIZE ERROR DISPLAY 'ERROR'      <* condition
  NOT ON SIZE ERROR                  <* condition
      DISPLAY 'NOT ERROR'
END-ADD
DISPLAY 'NOT IN ADD'.
```

Listing 13: A COBOL code snippet demonstrating conditional statements

imperative statements. In this work, I will not distinguish between these types of statements and I will process them in the same way. If a statement has also conditions, I will take them as additional phrases of that statement.

2.2.5.1 Add Statement

The ADD statement sums two or more numeric operands and stores the result. The ADD statement can be written in the three formats. These formats are shown with examples in Listing 14. The first format is used to sum literals and/or numeric data items preceding keyword TO, and to add the sum to every data item following keyword TO. The ADD statement in second format sums literals and/or numeric data items preceding keyword TO, this sum adds to a numeric data item or a literal following keyword TO, and finally stores

```
ADD 5, NUM1 TO NUM2, NUM3.       <* NUM2 += 5 + NUM1;
                                 <* NUM3 += 5 + NUM1
ADD NUM1 5 TO NUM2 GIVING NUM3. <* NUM3 = NUM2 + NUM1 + 5
ADD CORRESPONDING GROUP1 TO GROUP2.
```

Listing 14: A COBOL code snippet demonstrating the ADD statement

the result in every numeric data item following keyword GIVING. The third format uses CORRESPONDING phrase.

The CORRESPONDING phrase is a phrase used in ADD, SUBTRACT and MOVE statements. When this phrase is specified, an operation between a source and a target group is done over their subordinate data items. The operation is done between two data items (one from the source group and one from the target group) if the following conditions are fulfilled:

- Both data items are elementary numeric data items.
- Both data items have the same name and the same qualifiers up to but not including them.
- None of them is filler.
- None of them is described as level 66, level 77, or level 88 item.
- None of them is described with USAGE INDEX, USAGE POINTER, USAGE FUNCTION-POINTER, USAGE PROCEDURE-POINTER, or USAGE OBJECT REFERENCE.
- None of them include a REDEFINES, RENAMES, or OCCURS clause.

Listing 15 demonstrates the CORRESPONDING phrase. Only ITEM-A, ITEM-B and ITEM-E items are added from the one group to the other. Both ITEM-C items are not included because they are not numeric. ITEM-D from ITEM-1 contains the REDEFINES clause so they are not included either. Both ITEM-F items are defined with USAGE INDEX, so they are not included as well.

The operation of the CORRESPONDING clause is applied recursively if two subordinate data items with same name are groups.

2.2.5.2 Subtract Statement

The SUBTRACT statement subtracts one numeric item, or the sum of two or more numeric items, from one or more numeric items, and stores the result. The SUBTRACT statement can be written in the three formats. These formats are shown with examples in Listing 16. The first format is used to sum literals and/or numeric data items preceding keyword FROM, and to subtract this sum from every data item following keyword FROM. The SUBTRACT

```
05 ITEM-1 OCCURS 6.
  10  ITEM-A PIC S9(3).
  10  ITEM-B PIC +99.9.
  10  ITEM-C PIC X(4).
  10  ITEM-D REDEFINES ITEM-C PIC 9(4).
  10  ITEM-E USAGE COMP-1.
  10  ITEM-F USAGE INDEX.
05 ITEM-2.
  10  ITEM-A PIC 99.
  10  ITEM-B PIC +9V9.
  10  ITEM-C PIC A(4).
  10  ITEM-D PIC 9(4).
  10  ITEM-E PIC 9(9) USAGE COMP.
  10  ITEM-F USAGE INDEX.

ADD CORRESPONDING ITEM-2 TO ITEM-1(X).
```

Listing 15: A COBOL code snippet demonstrating the ADD CORRESPONDING statement

```
SUBTRACT 5, NUM1 FROM NUM2, NUM3. <* NUM2 -= (5 + NUM1);
                                  <* NUM3 -= (5 + NUM1)
SUBTRACT NUM1 5 FROM NUM2
      GIVING NUM3.          <* NUM3 = NUM2 - (NUM1 + 5)
SUBTRACT CORRESPONDING GROUP1 TO GROUP2.
```

Listing 16: A COBOL code snippet demonstrating the SUBTRACT statement

statement in the second format sums literals and/or numeric data items preceding keyword FROM, this sum subtracts from a numeric data item or a literal following keyword FROM, and finally stores the result in every numeric data item following keyword GIVING. The third format uses CORRESPONDING phrase which is the same phrase introduced in Section 2.2.5.1. In the case of the SUBTRACT statement, the operation is subtracting, not adding.

2.2.5.3 Multiply Statement

The MULTIPLY statement multiplies numeric items and sets the values of data items equal to the results. The MULTIPLY statement can be written in

the two formats. These formats are shown with examples in Listing 17. The

```
MULTIPLY 5 BY NUM2, NUM3.   <* NUM2 *= 5;
                            <* NUM3 *= 5
MULTIPLY NUM1 BY 5
     GIVING NUM2, NUM3.     <* NUM2 = NUM3 = NUM1 * 5
```

Listing 17: A COBOL code snippet demonstrating the MULTIPLY statement

first format is used to multiply every data item following keyword BY by an item preceding keyword BY. The MULTIPLY statement in the second format multiplies items preceding and following keyword BY together, and stores the result in every data item following keyword GIVING.

2.2.5.4 Divide Statement

The DIVIDE statement divides one numeric data item into or by others and sets the values of data items equal to the quotient and remainder. The DIVIDE statement can be written in the five formats. These formats are shown with examples in Listing 18. The first format is used to divide every data item

```
DIVIDE 5 INTO NUM1 NUM2.   <* NUM1 /= 5;
                           <* NUM2 /= 5
DIVIDE 5 INTO 10
      GIVING NUM1 NUM2.    <* NUM1 = NUM2 = 10 / 5
DIVIDE 5 BY 10
      GIVING NUM1 NUM2.    <* NUM1 = NUM2 = 5 / 10
DIVIDE 5 INTO 10
      GIVING NUM1 NUM2     <* NUM1 = NUM2 = 10 / 5;
      REMAINDER NUM4.      <* NUM4 = 10 % 5
DIVIDE 5 BY 10
      GIVING NUM1 NUM2     <* NUM1 = NUM2 = 5 / 10;
      REMAINDER NUM4.      <* NUM4 = 5 % 10
```

Listing 18: A COBOL code snippet demonstrating the DIVIDE statement

following keyword INTO by an item preceding keyword INTO. The DIVIDE statement in the second format divides an item following keyword INTO by an item preceding keyword INTO. The result is stored in every data item after keyword GIVING. The third format does same thing as the second format but the order of division operands is reversed. The fourth and the fifth formats

are same as the second and the third formats, respectively, but these formats contain also remainder phrase which stores remainder of the division in target data items.

2.2.5.5 Compute Statement

The COMPUTE statement is a COBOL statement which allows a computation of variables in the more modern form by using arithmetic expressions. This statement can be used as the replacement of the other arithmetic statements. It is demonstrated in Listing 19.

```
COMPUTE A, B = 42 / 13.    <* A = B = 42 / 13
```

Listing 19: A COBOL code snippet demonstrating the COMPUTE statement

COBOL arithmetic expressions can contain standard arithmetic operators with the following operator precedence (highest to lowest):

1. Unary + or -
2. Exponentiation (**)
3. Multiplication (*) and division (/)
4. Addition (+) and subtraction (-)

Operands of arithmetic expressions can be other arithmetic expressions, numeric literals, numeric functions or numeric elementary items. It is also possible to change the order of evaluation of operators with parentheses. The order of evaluation of operators with same precedence is from left to right.

2.2.5.6 Move Statement

The MOVE statement transfers data from one area of storage to one or more other areas. The MOVE statement can be written in the two formats. These formats are shown in examples in Listing 20. The first format is used to move

```
MOVE A TO B C.
MOVE CORRESPONDING GROUP1 TO GROUP2.
```

Listing 20: A COBOL code snippet demonstrating the MOVE statement

data from an item preceding keyword TO to data items following keyword TO. The second format uses the same CORRESPONDING phrase introduced with the ADD statement in Section 2.2.5.1. In the case of the MOVE CORRESPONDING, it is not necessary to have both subordinate data items described

as elementary numeric items, but at least one of them must be elementary item.

Moves between elementary data items are quite simple. If data items have different categories or they are described with different usage types, COBOL automatically converts data to have the correct format. So, in the case of moving data from a numeric data item represented by a string to a numeric data item represented by a binary number, COBOL automatically converts data from the string format to the binary format.

A group move is a move in which at least one operand is a group item. The group move is treated as though it were an alphanumeric-to-alphanumeric elementary move, except there is no conversion of data from one form of representation to another. COBOL group moving is similar to copying raw memory from one place to another. To find the exact location where a subordinate data item is moved, I will use Table 2.4. By calculating individual sizes and offsets of elementary data items, it can be decided where exactly elementary data items are moved. This is demonstrated in Listing 21. The example

```
01 S1.
  05 NUM1 PIC 9(5) USAGE IS BINARY VALUE 65. <* size 4
  05 NUM2 PIC 9(9) VALUE 10.                 <* size 9

01 S2.
  05 NUMA PIC X(4).                          <* size 4
  05 NUMB PIC 9(9) VALUE 15.                 <* size 9

MOVE S1 to S2.
DISPLAY NUMA " " NUMB. <* output: A 000000010
```

Listing 21: A COBOL code snippet demonstrating group move

shows that data of **NUM1** are moved to the location of **NUMA** perfectly aligned. This happened because these elementary data items have the same size, so their alignment from the start is also the same. After the move, data of **NUMA** represents the string A instead of the original numeric value 65. Data between **NUM2** and **NUMB** are moved similarly.

2.2.6 Other COBOL Features

This section presents other COBOL features which are important for data flow analysis.

2.2.6.1 Subprograms

Program divisions are typically followed by an optional END PROGRAM command to indicate that the code of the current program is ended. However, COBOL supports nested programs, also called subprograms, to be defined within a program. After divisions and before the END PROGRAM command, a definition of another COBOL program can appear. Subprograms are almost identical to a main, top-level COBOL program. It is possible for a program to communicate with an inner program but I will not discuss it here. All programs, even nested programs, will be processed as independent units in this work.

2.2.6.2 Separators

Separators are used to indicate a separation of COBOL segments or tokens. A typical separator is a space. A space indicates a separation of two tokens. A period is another separator. As it was already seen, periods are used as an ending character of various COBOL constructs. Sentences are ended with periods, division keywords are ended with periods as well as many other keywords are ended with periods. A period should be followed with at least one space.

Two other separators commonly used in COBOL are a comma and a semicolon. A comma and a semicolon should be followed by at least one space, and they represent the same separator as a space. They are used to delimit tokens, with possibly better readability. Therefore, **ADD 5, 5 TO X** is the same statement as **ADD 5 5 TO X**.

2.2.6.3 Identifiers and Qualification

An identifier is an user-defined word specifying the name of an entity. In COBOL, it is possible to name variables, paragraphs, sections, programs, and so on. A name can contain the following characters:

- Latin uppercase letters A-Z
- Latin lowercase letters a-z
- digits 0-9
- - (hyphen)
- _ (underscore)

The hyphen cannot appear as the first nor the last character in user-defined words. The underscore cannot appear as the first character in user-defined words. In some cases, a user-defined word must have at least one alphabetic character. For example, data names cannot be named **12-34** but sections can.

Names don't need to be unique if there is a possibility to qualify them uniquely. Consider Listing 22. In the example, there are multiple entities with same names. These definitions are correct because it is possible to uniquely

```
01 A.
    05 B USAGE DISPLAY.
        10 C PICTURE X.
01 A2 USAGE POINTER.
    05 B USAGE DISPLAY.
        10 C PICTURE X.

DISPLAY C of A.
DISPLAY C OF B of A.
```

Listing 22: A COBOL code snippet demonstrating a qualification of names

qualify **B** and **C** through its parent **A**. To do this, COBOL uses keywords **OF** or **IN** in a qualification. For example, to qualify the first **C**, a user can write **C of A**. The names in the qualification are written in the order from the inner item to the outer item. The qualification of intermediate entities is not necessary if names can be uniquely qualified (i.e. it is not necessary to write **C of B of A**). The qualification works on all items in the hierarchy system, so also on condition-names, indices, renames, redefines, and so on.

2.2.6.4 Subscripting

Subscripting is a method of providing table references through the use of subscripts. A subscript is a positive integer whose value specifies the occurrence number of a table element. Subscripts are written in the parentheses after the whole qualified name is specified. This is shown in Listing 23.

```
01  TABLE-THREE.
  05  ELEMENT-ONE OCCURS 3 TIMES INDEXED BY I.
    10  ELEMENT-TWO OCCURS 3 TIMES.
      15  ELEMENT-THREE PIC X(8).

DISPLAY ELEMENT-THREE OF TABLE-THREE (2 1).
```

Listing 23: A COBOL subscripting example

Subscripts must be specified in the order from the outermost to the innermost. Arrays are started from the index one, and the number of subscripts must be exactly the same as the number of dimensions in the hierarchy of the data entity. It is possible to use literals, data items or indices in subscripting. Subscripts can be absolute (a single item) or relative (a single item +/- an integer).

Since COBOL 2002 standard [29], it is possible to use arithmetic expressions in subscripting. However, such combination gave COBOL some problems. There is no specific delimiter which specifies a delimitation of two adjacent subscripts, and therefore some situations are ambiguous. For example, **A(1 + 1 + 1 + 1)** can have two representations. One with two subscripts **(1 + 1)** and **(+ 1 + 1)**, and the second one with a single subscript (the whole expression). From experiments with GnuCOBOL, it seems that GnuCOBOL goes the greedy way and it tries to parse everything into the one expression until it cannot continue and then it starts another arithmetic expression. In this work, I will go on this in the same way. It is important to note that IBM COBOL does not support subscripting with arithmetic expressions, but because it is an additive used in other COBOL implementations, I have decided to include it.

2.2.6.5 Literals

IBM COBOL supports various types of literals. These types are:

- Basic alphanumeric literals (enclosed in quotation marks or apostrophes)
- Hexadecimal alphanumeric literals (prefixed with X)
- Null-terminated alphanumeric literals (prefixed with Z)
- DBCS literals (prefixed with G or N)
- UTF-8 literals (prefixed with U)
- UTF-8 hexadecimal literals (prefixed with UX)
- National literals (prefixed with N)
- National hexadecimal literals (prefixed with NX)
- Integer literals (sign, digits)
- Floating-point literals (sign, digits, decimal point is dot character)
- Floating-point literals (scientific format - sign, mantisa, symbol E, sign, exponent)
- Figurative constants

A literal can be prefixed with symbols X, Z, G, and so on. For example, X"abcd" represents a hexadecimal literal. All literals except numeric literals and figurative constants should be enclosed in quotation marks or apostrophes.

In the case of floating-point literals, a typical delimiter is a period. It is possible to change decimal point character to a comma character with DECIMAL-POINT IS COMMA command, but it has some consequences. I will not support this command in this work. Figurative constants are constants with special properties. For example, the figurative constant HIGH-VALUE can be used to assign a highest possible value to given data item.

2.2.6.6 Source Code Formats

Source code formats define the structure of a COBOL source code. There are three main source code formats:

- Fixed format
- Variable format
- Free format

Fixed format is the oldest COBOL source code format and it has a very strict structure. The first six positions on every line have a special purpose. They are used as the place for a sequence number. This number is not used nowadays but fixed format still demands it. The seventh position on each line has also a special purpose. This position is the place for a line indicator, which states the purpose of line. The possible choices are listed in Table 2.6. From eighth to seventy-second position, it is possible to write a COBOL code.

Line indicator	**Meaning**
* or /	line is commentary
-	continuation of the previous line
$	line with compiler directive
D or d	debugging line (ignored if program is not in the debug mode)
(space)	code line

Table 2.6: Line indicators of fixed format in COBOL

After this, there is segment of length eight, which is used as a programmer documentation. Every source code line in fixed format has exactly eighty positions. Figure 2.4 describes this structure more visually. In earlier versions of COBOL, statements had to start from twelfth position but in COBOL 2002 this restriction was lifted. Fixed format is the default format for IBM COBOL programs.

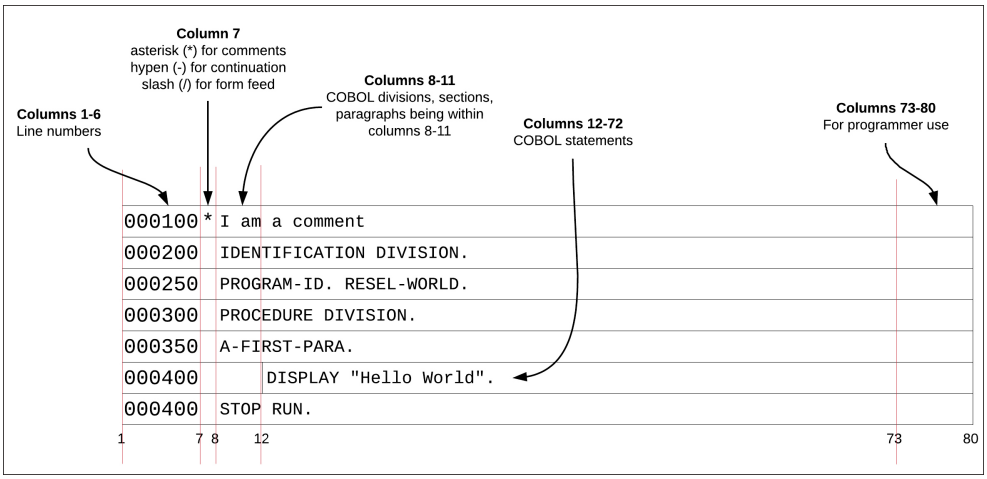

Figure 2.4: COBOL fixed format [3]

Variable format is similar to fixed format. It divides source code line into same segments except the two last ones. The COBOL code can have unlimited positions and there is no programmer documentation area. For practical purposes, a COBOL code in this format is commonly limited to 250 characters per line.

The last format is free format. As its name suggest, this format is the least restrictive. In this format, line indicator (same as in fixed format) is placed at the first position on each line. In free format, there are no continuation lines, and COBOL code can start even at first position if it starts with the other characters than indicator characters.

2.2.6.7 Copy Statement

The COPY statement is a library statement that places a prewritten text in a COBOL program source code. This statement is used to place a part of source code that is stored elsewhere into the actual source code unit. Its simplified syntax is explained by Listing 24. Using such copy statement in a COBOL source code will result in placing the content of the file named **copybook** at the place of the copy statement, replacing the copy statement. In COBOL world, copybooks are files sharing data definitions between multiple programs.

It is possible to share an arbitrary COBOL code between files, not only data definitions. Files which are not sharing definitions are not commonly called copybooks but I will call them copybooks anyway, for simplicity.

The REPLACING clause of the copy statement is used to replace the first string with the second string while copying the content of the source file. In this case, it would replace all words **word** for the word **word2**. The replacing clause is an optional clause and it can contain an unlimited number of replacing parts.

```
COPY 'copybook' REPLACING ==word== BY ==word2==.
```

Listing 24: A COPY statement example

A text used between two assignments is special for the copy statement. This text is called pseudo-text and it can contain an arbitrary string (except one with ==). With pseudo-text, it is possible to replace not only single words by also groups of words. Other operands which can be used in the REPLACING clause are:

- word (also keyword)
- identifier
- literal

Copy statements are evaluated in the preprocessing phase, so all copy statements are replaced before the program is compiled.

2.3 Requirements

The solution produced by this work will be used in Manta project with other data flow analysis tools, so it is important to state the requirements on the solution by Manta project.

The **functional** requirements are:

- The tool will parse a correct COBOL source code to AST internal representation.
- The tool will produce data flow information in the form of the data flow graph.

The **nonfunctional** requirements are:

- The tool will use parser generator ANTLR.
- The solution will be implemented in Java programming language and it will use common codebase and technologies from Manta project.
- Data flow analysis should have a reasonable execution time (i.e. in a matter of seconds).
- The quality of the solution should be verifiable by tests.

There are other suggestions from Manta project which should be considered in the course of the implementation. These are:

- It is expected that an input to the analyzer is correct, so the tool does not need to check syntax validity of the input entirely. Checking syntax validity entirely only makes process much harder with no added benefit. For example, the order of clauses in the grammar can be arbitrary which allows the grammar to be simpler.
- The tool does not need to check semantic validity of the input entirely. Checking semantic validity entirely only makes process much harder with no added benefit. For example, it is allowed to have a string literal in a place where only numeric literals are allowed.

2.4 Existing solutions

COBOL is an extensive language, so it is not very convenient to start from scratch. Therefore, this section is devoted to the analysis of existing solutions which could help with parsing of COBOL.

2.4.1 IBM's VS COBOL II Grammar

IBM's VS COBOL II grammar is a COBOL grammar for an older version of IBM's COBOL. This grammar is written in ENBF and contains the syntax of many COBOL constructs. VS COBOL II seems to be very similar to Enterprise COBOL for z/OS, so this grammar is very suitable for this work. Its disadvantage is that lexical rules are not very specific and many of them

contain only generic regular expressions. This version of COBOL is also a bit older so constructs such as intrinsic functions are not present in the grammar. Even though this grammar is a decent one, I will look for a more complete description of COBOL syntax.

2.4.2 GnuCOBOL

GnuCOBOL [19] is an open-source COBOL compiler. This compiler is a stable and also well known. Unfortunately, it is written in C programming language and uses bison and flex for parsing. It would be necessary to rewrite its grammar, so this solution is not very suitable.

2.4.3 Java Cobol Lexer

Java Cobol Lexer [30] is an implementation of COBOL lexer in Java. This program meets requirements but it is not very well supported (last update was in 2013). Also, lexical analysis is easier than parsing so unless there is a COBOL parser that does not have a lexer I will not use this solution.

2.4.4 RES - An Open Cobol To Java Translator

RES [31] is an open-source translator of a VS COBOL code to a Java code. This tool contains a COBOL lexer and also a parser. Unfortunately, the parser and lexer are generated by JavaCC parser generator, and therefore it does not meet requirements. The project contains a source grammar for COBOL parser which can be used, however, the JavaCC grammar syntax is different to the ANTLR grammar syntax, so it would be necessary to rewrite the grammar.

2.4.5 TypeCobol

TypeCobol [32] is an open-source Cobol 85 incremental parser and an extension of Cobol 85 language named TypeCobol. This project uses an ANTLR4 parser to parse an input COBOL code, and the whole project is created in C#. Its grammar seems to be very robust and covers COBOL concepts extensively but sadly, the project's license does not allow the use of this parser in Manta project.

2.4.6 ProLeap ANTLR4-based parser for COBOL

ProLeap COBOL parser [33] is an ANTLR4 parser for a COBOL code. This parser is not targeted at any particular COBOL dialect but supports a wide range of COBOL dialects and extensions. The project contains a COBOL parser, a COBOL grammar, and also a codebase to use the parser and to generate AST program representation. The project converts AST representation to another representation which the author calls ASG (Abstract Semantic

Graph). An ASG is a graph representation of an AST on which some phases of semantic analysis were done.

This solution is very much suitable because it contains not only the grammar but also the codebase to preprocess a COBOL code and many COBOL testing examples for parsing. The project is managed under MIT license, so the license is also suitable. Because this solution seems to be best out there, I will use this solution as the background for the design and implementation.

2.5 ProLeap ANTLR4 COBOL Parser

Even though the ProLeap ANTLR4 COBOL parser seems to be the best open-source COBOL parser out there, there are some things which are done incorrectly, or could be improved. In this section, I want to mention some things with their possible fixes. I will not mention everything as the grammar has a lot of problems with ambiguities and incorrect/missing keywords. Some grammar ambiguities can be easily fixed but others arise from the nature of COBOL syntax, and therefore cannot be easily fixed. I will tackle these things in the implementation part of this work.

2.5.1 Arithmetic Expressions

The grammar rules for parsing arithmetic expressions is done by the typical method of top down parsing by priorities. However, ANTLR4 parser generator allows to use a direct left-recursion in a source grammar, so original arithmetic expressions rules can be rewritten to the simpler, more natural form. The same principle can be applied on other expression rules in COBOL, such as condition expression rules.

2.5.2 Ambiguities in Identifiers

Listing 25 shows grammar rules for parsing of qualified data names in the ProLeap ANTLR4 grammar. The problem is that the rules for parsing are almost the same for all formats, and therefore parsing algorithm will always go with **format1**. It is not possible to distinguish between a data name and a paragraph name on the lexical nor syntax level. The solution for this is to use specific names only in segments where they are the only option, and in a general case let the semantic analysis process to figure out what given name represents.

2.5.3 Nongreedy Subrules

ANTLR4 allows to use nongreedy versions of lexer and parser subrules in grammars [34]. To define a subrule with the nongreedy behaviour, it is necessary to add suffix **?** (the question mark) to given subrule. Consider Listing

```
qualifiedDataName
   : qualifiedDataNameFormat1
   | qualifiedDataNameFormat2
   | qualifiedDataNameFormat3
   ;

qualifiedDataNameFormat1
   : (dataName | conditionName)
     (qualifiedInData+ inFile? | inFile)?
   ;

qualifiedDataNameFormat2
   : paragraphName inSection
   ;

qualifiedDataNameFormat3
   : textName inLibrary
   ;
```

Listing 25: ProLeap ANTLR4 parser grammar rules for parsing qualified names

26. This way, pseudotexts can be parsed on the lexer level instead of the

```
PSEUDO_TEXT : DOUBLEEQUALCHAR .*? DOUBLEEQUALCHAR;
```

Listing 26: A ANTLR4 lexer grammar rule for parsing pseudotext

parser level. Nongreedy subrules can also be used in other lexer rules where it is important to match an input until the first occurrence of a given symbol. The typical case is parsing of strings.

Chapter 3

Design

This chapter presents design concepts of the target solution. The first section presents technologies which are used for the implementation of the target solution. The second section describes a decomposition of the solution into modules and their responsibilities. The third section describes how data entities will be represented in the analyzer. The fourth section describes how data types will be assigned, and the fifth section tells about data flow in renames and redefines.

3.1 Technologies

This section presents technologies that are commonly used in Manta project. These technologies will be also used in this work.

3.1.1 Java

Java is a modern general-purpose programming language. Manta project uses Java for most of its codebase, and the usage of Java is also one of the requirements for the target solution. In this work, I will use the version which is common for other modules in Manta project and that is Java SE 8.

3.1.2 Spring Framework

The Spring Framework [35] is an application framework for building Java applications. This framework is very extensive and delivers multiple modules for various purposes. The Spring framework is an easy-to-use framework, in which it is possible to define a series of XML configurations which set up everything necessary for easy prototyping. I will use Spring for a configuration of the target solution in testing phase.

3.1.3 Apache Maven

Apache Maven [36] is a software project management tool used primarily for Java. With Maven, it is possible to decouple a project into multiple modules (artifacts) and manage dependencies between them.

Maven has a very simple configuration file called POM file, in which a developer states information about the module and all its dependencies. Maven then handles everything - resolving dependencies, downloading dependencies and building of the module. Maven will be used in this work to decouple the solution into multiple independent units, and also to resolve dependencies to other Manta project modules and external libraries.

3.1.4 JUnit

JUnit [37] is a unit testing framework for Java. With JUnit, it is possible to create Java unit tests simply by appending Java annotations on corresponding test classes and methods. JUnit also supports parametrized tests where it is possible to maintain test suites with supplied parameter values.

This library is very much standard for Java unit testing and it will be used also in this work. I will use JUnit for verification of quality of created parser as well as for the verification of results of the data flow analysis.

3.1.5 ANTLR

ANTLR is a parser generator which was introduced in Section 1.3.5.1. This parser generator will be used for the creation of a parser and a lexer for COBOL. Other Manta analyzers are commonly using ANTLR3 but I will use ANTLR4 due to the existence of the suitable ANTLR4 COBOL grammar.

There was an attempt to convert the ANTLR4 COBOL grammar to an ANTLR3 COBOL grammar but it was not successful. ANTLR3 solves ambiguities at compile-time with predicates whereas ANTLR4 solves them at runtime by using production rules with the lowest numbers. Due to this, it would be necessary to rewrite many rules of the grammar to get it working.

It is important to note that the Manta codebase somewhat expects the usage of ANTLR3 AST so it is necessary to make some adjustments and convert an output of ANTLR4 (a parse tree) to an ANTLR3 AST.

3.2 Modules

The solution will be decoupled into a few modules based on similarities with other modules in Manta project. These modules are:

- manta-connector-cobol-model (**Connector Model**)
- manta-connector-cobol-resolver (**Connector Resolver**)
- manta-connector-cobol-testutils (**Connector Testutils**)

- manta-dataflow-generator-cobol (**Dataflow Generator**)

Figure 3.1 shows a diagram of dependencies among these modules.

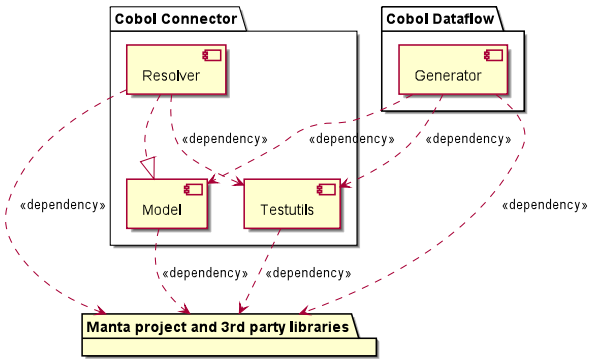

Figure 3.1: A diagram showing dependencies among modules

3.2.1 Connector Modules

Connector modules are responsible for reading, parsing, processing, and creation of the internal representation (AST) of an input program. These modules also handle semantic analysis (resolving) of the created internal representation.

Model module describes Java interfaces to other connector modules. This module is used as a communication resource between connector modules, dataflow modules and also other Manta project modules.

Testutils module contains base classes for testing. Classes of this module are shared within multiple modules.

Resolver module is a module which has the most responsibilities. The following list describes its individual phases:

1. Reading of an input source code.
2. Preprocessing of the input source code to a single code representation.
3. Parsing of the single code representation to a parse tree.
4. Transformation of the parse tree to an AST.
5. Semantic analysis of the AST.

The first phase is responsible for reading of an input COBOL source code into string. It should be possible to read an input source code in various input formats such as a string or a file. This phase should also accept additional information about the input such as used encoding, copybooks locations and the source code format. The input source code and additional information are passed to the next phase.

The second phase is responsible for preprocessing of the input. As already noted, there are many COBOL source code formats. This phase should

handle converting of the input source code to an unified source code which is stripped from sequence numbers, the last segment (commentary segment) and the line identification field. The preprocessing phase is also devoted to finding copybooks and replacing COPY statements for a content of copybooks. It is necessary to process replacing clauses of COPY statements too.

The third phase is devoted to parsing of the unified source code representation. A parser should be generated by the improved ProLeap ANTLR4 grammar. An output of the ANTLR4 parser is a parse tree representing the structure of the input program.

The fourth phase is done over the parse tree produced by the previous phase. The purpose of this phase is to transform the parse tree to an AST with an enhanced functionality. In the process, the transformation should remove unimportant nodes which are not needed. The AST representation should be based on the common codebase used in Manta project.

The last phase accepts the AST representing the input program. In this phase, the module will resolve all identifiers to their real representations. If an identifier cannot be resolved, the module should try to guess its representation. The output of this phase is the program representation suitable to the following data flow extraction.

3.2.2 Dataflow Generator Module

Dataflow Generator module is a module responsible for a generation of a data flow graph from the input AST, which is passed to this module from Connector modules. The role of this module will be to traverse the resolved AST representing the COBOL program, and for each statement generate corresponding data flow graph nodes and edges. For a traversal of the tree, the module should implement a visitor pattern.

Statements of COBOL programs should be processed as follows:

1. Create nodes for every operand and data item used in a given statement. Create also their corresponding structures according to defined hierarchy.
2. Create nodes representing the given statement with its corresponding hierarchy parents (sections, paragraphs, programs, ...).
3. Create intermediate result nodes (called **ColumnFlows**) which represents changes in te given statement and attach them under the statement node. This way these nodes will be present as leaf nodes.
4. According to semantics of given statement, connect created nodes correspondingly. Edges should be only between leaf nodes.

These steps don't have to be executed in given order. In some cases, it can be easier to create statement nodes before operand nodes. Figure 3.2 presents a possible output for a simple ADD statement. Edges marked with **D** represent direct flows as discussed in Section in 1.1.1.

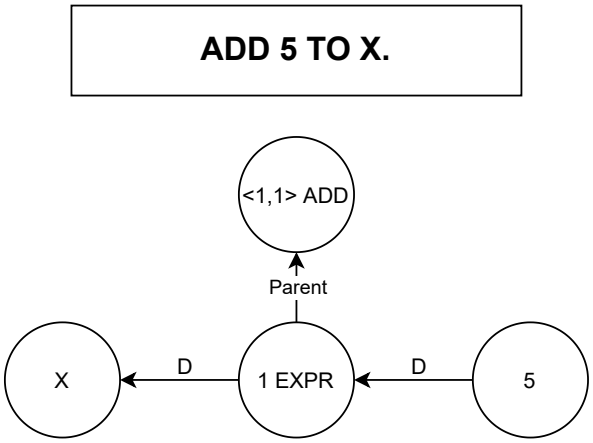

Figure 3.2: A data flow graph for a simple ADD statement

3.3 Data Entities

The process of semantic analysis should create data entities representing variables, data types and other related constructs, and save them to a symbol table. In this work, I will use data representation scheme of Manta project.

For each data description entry from Data Division, I will create an object of **IResObject** type which will represent data of the entry. **IResObject** objects should create same hierarchy scheme as it is present in leveling system of COBOL description entries. Each object should have assigned a corresponding data type object represented by **IResDataType** class. Data types will be assigned according to the usage clause and/or picture clause of the corresponding data description entry. In the case of variables which represent special objects such as data items with level 66, redefines, renames or condition-names (level 88), these variables shouldn't be a part of direct hierarchy of **IResObject** objects. Special objects will be called pseudo attributes, and they will be saved to and retrieved from the hierarchy with special methods.

An example of the data representation scheme is shown in Figure 3.3. In the example, it can be seen that the hierarchy contains parent/child relations between data types (RECORDs) and objects (ELEM-ITEM, ELEM-ARRAY, ELEM-ARRAY-ITEM). Typically, an object has a datatype and if the data type is a record, inner objects are associated with the data type. The same pattern is used here. In the case of pseudo attributes, they are associated directly with corresponding objects. In this hierarchy, arrays are exceptions. Because an array is not a simple object, it is represented with an **IResArray** object, and a duo - a key and a content. The object ELEM-ARRAY represents the whole array. Its key is not so important and it is here purely for compatibility with the codebase. The content, however, is important and it represents the content (the data) of the particular array.

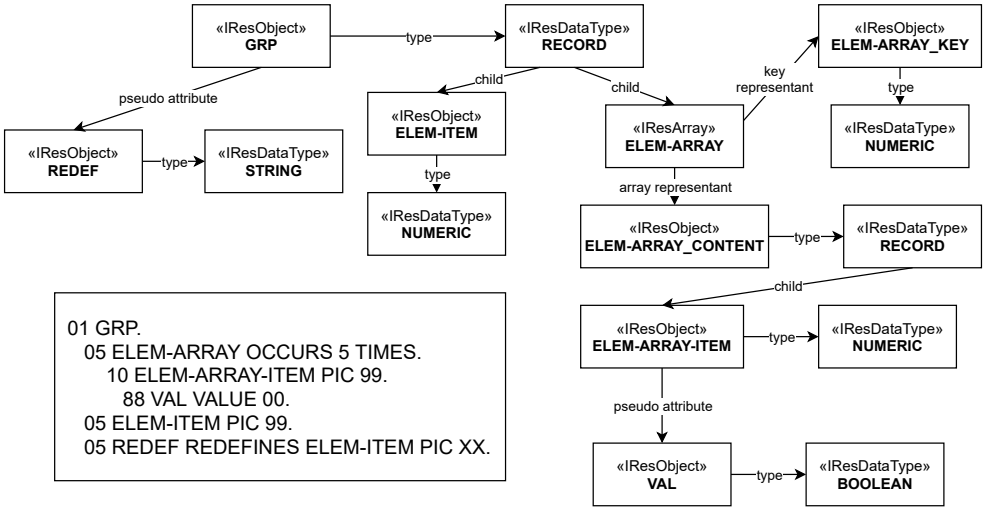

Figure 3.3: An example of the data entities representation scheme

All created objects will be saved to a symbol table which will be represented by a program scope (**IResScope** class). The program scope will maintain a list of all created objects in the program. In COBOL, all variables have the global scope of validity so one scope for every program is enough. Even though programs are processed as individual units in this work, I will create a tree-like structure of scopes according to the hierarchy of programs and inner programs. This will help in further extensions of the COBOL analyzer. It is important to note that all objects must be saved to the scope because in COBOL, it is possible to use references to data items without their fully qualified names.

3.4 Data Types

As already mentioned in Section 2.2.4.7, COBOL does not have traditional data types. In this work, I will assign common data types, used in modern programming languages, to data items. These types will be:

- record
- numeric (purely)
- string
- boolean
- data pointer
- procedure pointer
- object reference
- unknown

The data type of a data item will be assigned by the picture and the usage clause of its data description entry. A data item will have record type if it

is a record. The other data types are primitive data types and they will be recognized by Table 2.4. The table contains possible combinations of picture characters and usage types, so a series of if statements can recognize what type given data item has. In the case of condition-names, they will have assigned boolean type because their behaviour simulates booleans. A rename doesn't have data type, so its type should be unknown. If it won't be possible to recognize the type of an item, unknown type will be used.

3.5 Redefines and Renames

Redefines and renames were introduced in Sections 2.2.4.8 and 2.2.4.10, respectively. These data items are special because they share a memory with other data items, and therefore a change of data in one data item results in a change in other data items. Thus, it is important to represent data flow also between these items.

A trivial solution is to create data flow edges between corresponding data items in both directions every time, so it will ensure that data flow won't be lost. However, this solution is not very suitable because it will create many data flow edges in the data flow graph.

In this work, I will use a better approach. A direct data flow will be present between corresponding data elements only if there is an incoming edge to a source node and there is also an outgoing edge from a target node. This will handle cases when a one data item is used only for reading and a second one is used only for writing. In such cases, it is not necessary to create a data flow in both directions. Figure 3.4 illustrates that.

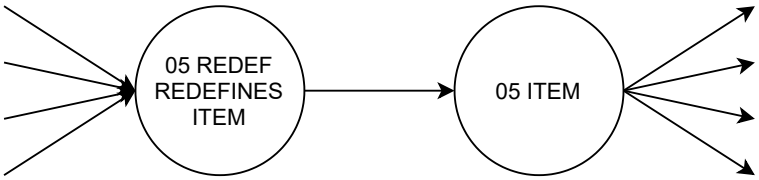

Figure 3.4: An example of a data flow between a redefine and its source

Chapter 4

Implementation

This chapter describes the implementation part of this work. Due to an extensive number of implemented classes, I have decided to describe only the most important ones. Classes described in this section comes from two modules – Connector Resolver and Dataflow Generator. The other two modules – Connector Model and Connector Testutils – contains mostly interfaces and generic classes, so they are discussed here. The common pattern for class naming is adding suffix **Impl** to the name of a interface which the class implements.

It is important to note that source codes contain also an implementation of the data flow analysis for additional features of COBOL such as embedded SQL processing, statements working with files, and the data flow analysis between COBOL subprograms. These additions were not the goal of this thesis and some of them were created as a collaboration with other members of Manta project. This part of the thesis describes source codes that were created by myself and they cover the goal of the thesis, so constructs discussed in the analysis and design part of this work.

4.1 Connector Resolver

This section describes the most important classes of Connector Resolver module.

4.1.1 CobolParserServiceImpl

CobolParserServiceImpl is the most important class of Resolver module. This class provides two public methods for parsing and resolving - *analyzeFile* and *analyzeCode*. As names imply, these methods are used for the analysis of files and the analysis of codes, respectively. Both methods process an input in corresponding formats and converts it into a string. The string of input code is then processed by a workflow illustrated in Figure 4.1.

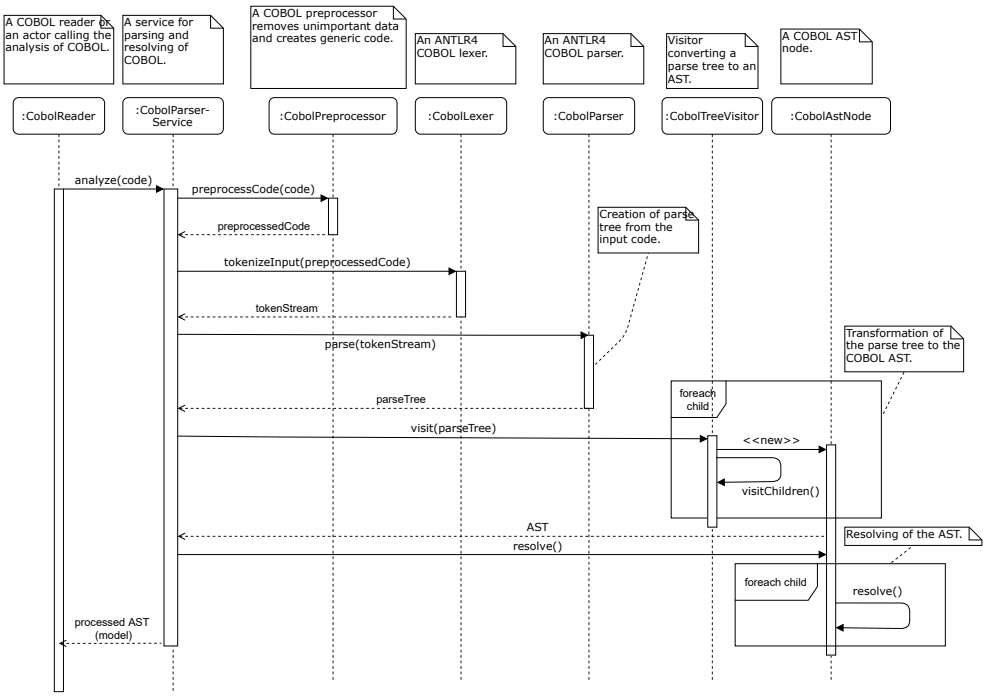

Figure 4.1: The process of the COBOL parser service

The workflow of **CobolParserServiceImpl** is the same as it was designed in Section 3.2.1.

In the first phase, a COBOL code is preprocessed by a **CobolPreprocessor**. In this phase, the original COBOL code is stripped from unimportant data, continuation lines are appended accordingly, COPY statements are replaced with copybooks, and so on.

The next two phases are tokenization and parsing by an ANTLR4 generated lexer and parser. After this, the original code is parsed in the form of a ANTLR4 parse tree representation. The ANTLR4 parse tree representation is a Java tree with information about the parsing process. The tree is composed of nodes called contexts, where each context represent an instance of used rule or subrule in the process of parsing. For example, a node of the class **StartRuleContext** represents the rule *startRule*. Additional information, which can be retrieved from contexts, are starting and ending positions of used rules, and input tokens.

The ANTLR4 parse tree is transformed to a COBOL AST represented by class **CobolAstNode**. The transformation is done by the visitor pattern, which is handled by the following classes:

- **CobolTreeBuilderVisitor**
- **CobolTreeBaseVisitor** (base for others)
- **CobolTreeIdVisitor** (visits IDENTIFICATION DIVISION)

- **CobolTreeEnvVisitor** (visits ENVIRONMENT DIVISION)
- **CobolTreeDataVisitor** (visits DATA DIVISION)
- **CobolTreeProcVisitor** (visits PROCEDURE DIVISION)

The first class from the list is used as the builder class of the other visitor objects, whereas the others are devoted to processing of specific segments. **CobolTreeBaseVisitor** is the base class for visitors, in which common methods such as *visitTerminal*, *visitChildren*, *visitIdentifier* are defined. Most transformation methods are quite simple. They convert an instance of parse tree context to an instance of **CobolAstNode** node. Important nodes such as identifiers, data names and divisions are represented by more specific classes with extended possibilities. A created **CobolAstNode** node keeps the original parse tree context for retrieval of additional information about its origin. **CobolTreeDataVisitor** is a bit more complicated class because it contains also processing of description entries. Method *processDescriptionEntriesHierarchy* transforms the original hierarchy of description entries, which is linear, to the tree hierarchy, which represents the proper structure according to leveling system of COBOL.

The last phase – semantic analysis (resolving) – is done over the COBOL AST. Resolving is started with calling of **CobolAstNode's** *resolve* method on the root of the tree. The generic form of this method is shown in Listing 27. In short, this method does nothing and commands its children to do resolving. Important nodes such as identifiers and data names have this method overridden with the corresponding logic.

```
    public IResEntity resolve() {
      for (CobolAstNode child : getChildren()) {
          child.resolve();
      }
      return null;
  }
```

Listing 27: The generic resolving method

4.1.2 CobolPreprocessorImpl

CobolPreprocessorImpl and its related classes are an implementation of preprocessing for COBOL. Most of preprocessing classes were taken from ProLeap COBOL project. Its implementation is decent and covers most aspects of preprocessing of COBOL. Even though most of the code was copied, there are two changes which improved preprocessing of COBOL in my implementation. The first change is related to COPY statements. ProLeap handles replacing clauses wrongly and does not take into account that tokens are case-insensitive.

My implementation replaces tokens in COPY REPLACING clauses regardless of the case. The second change related to preprocessing is a grammar ambiguity. I have rewritten some grammar rules to be less ambiguous, mostly in a segment of COPY statement processing.

Even though the implementation is taken from another project, I want to summarize steps which are done in the COBOL preprocessing. These steps are:

1. An input COBOL code is divided into lines represented by custom objects.
2. Lines are normalized by stripping sequence numbers and comment entries. Line indicators are interpreted and removed. Continuation lines are appended correspondingly.
3. Lines are converted into a string.
4. The string is parsed by the ANTLR4 COBOL preprocessor parser.
5. The parse tree is traversed by a tree walker. While the tree is traversed, compiler directives are discarded, a content of copybooks is replaced and included. A content of the other nodes is simply copied to the output.
6. The output of tree walking is a string with the unified code.

4.1.3 IBMDataItemAnalyzer

IBMDataItemAnalyzer is a class responsible for analysis of IBM COBOL data items. Methods of this class accept the picture string and/or the using clause of an analyzed data item, and in series of if statements, they decide its category, size and the internal data type. Decision logic of this class is based on Table 2.4. Information produced by this class is then used in other classes related to resolving.

4.1.4 ResScope

ResScope is an implementation of the program scope in the COBOL analyzer. This class maintains an associative array where **IResObject** objects are stored. The class has two important methods:

- IResObject **resolveObjects**(objectName, properties) - returns an object with matching name and fulfilled properties, or null if there is no such object
- void **registerObject**(object) - registers an object to the scope

4.1.5 CobolDataDictionary

CobolDataDictionary is a factory class devoted to the creation of dictionary (IResObject) objects. Its interface consists of the following methods:

- **createObjectAsPseudoAttribute**

- **createArrayAsPseudoAttribute**
- **createObject**
- **createArray**
- **createDataType**

All these methods accept input parameters such as object's name, properties, data type, definition source type and its defining AST node. The defining AST node is an AST node from which the object is created. This node is used in cases when it is necessary to get additional information about object's origin.

4.1.6 DataDescriptioItemEntryImpl

DataDescriptionItemEntry is a class representing data description entries in the COBOL AST. This class overrides *resolve* method, and in that method, it creates an **IResObject** object representing a data item described by the data description entry. In short, *resolve* method tries to figure out the data type and the size of the data item. This is done by passing the picture string and the usage type to a **IBMDataItemAnalyzer** object.

After analysis by **IBMDataItemAnalyzer**, name, type, properties and the defining AST node reference are passed to a particular method of **CobolDataDictionary**. **CobolDataDictionary** creates an **IResObject** instance which is then saved to the program scope. However, the job of *resolve* method is not yet done. Resolving is delegated to subordinate data description entries. It is important to note that in this method, the size of elementary data item is attached as an attribute on corresponding **IResObject** object, which is subsequently used in Dataflow Generator module for a calculation of offsets in group moves.

4.1.7 QualifiedDataNameImpl and DataNameImpl

QualifiedDataNameImpl is a class of the COBOL AST devoted to processing of qualified names. Children of an instance of **QualifiedDataNameImpl** are objects of class **DataNameImpl**. These two classes are the core of the resolving process and both classes override the generic method *resolve*.

The *resolve* method passes the **QualifiedDataNameImpl** object to a **CobolReferenceResolver** class, which tries to find references for every data name. The **CobolReferenceResolver** is an extension of a common codebase of Manta resolving. This class is quite complicated but in short, it traverse through every data name (segment) and tries to resolve it.

Resolving is done in the order from the last to the first segment because COBOL uses reversed ordering of data names in qualified names. The first segment of a qualified data name is resolved through the program scope. Resolving process will look into a list of objects which are registered in the scope and tries to find match on the name of the segment. After the successful

match, the next segment, if it is present, is going to be resolved. Resolving of the next segment looks into the result of the previous segment and tries to find the suitable object in its hierarchy. The process continues until every segment of the qualified name is resolved. After the process, the reference of the last segment is returned and that is the reference of the qualified name.

It is important to note that there are cases when resolving does not find any matching variable. In that case, resolving tries to deduce the missing variable from the context of resolving. If a variable is deduced then it is registered into the program scope, so next missing variable with same qualifiers does not have to be deduced again.

4.2 Dataflow Generator

This section describes two most important classes of Dataflow Generator module.

4.2.1 CobolGraphHelper

CobolGraphHelper is a helper class devoted to building of a data flow graph. This class extends **AbstractGraphHelper** class from Manta codebase which contains common methods for building graph nodes and edges. The following list describes most important methods of **CobolGraphHelper** class:

- **buildNode(IResEntity)** - builds graph nodes for an **IResEntity** object passed as the input parameter. **IResEntity** is a more abstract version of **IResObject**.
- **buildOpNode(CobolAstNode)** - builds an operation graph node for a COBOL AST node passed as the input parameter.
- **buildEntityLeavesNodes(IResEntity)** - processes the whole structure of an input entity, builds it, and returns its leaf nodes.
- **connectDirectFlowBySize(IResEntity source, IResEntity target, Node opNode)** - method builds nodes for a source and target entity, and connect corresponding nodes according to their sizes and offsets. This method is used to create data flow between group items and redefines, where it is necessary to calculate offsets of data items for correct data flow processing. If an operation node is supplied, data flow is created through its **ColumnFlow** nodes.

4.2.2 CobolDataFlowVisitor

CobolDataFlowVisitor is the core class of the data flow analysis in this work, and its purpose is to create a data flow graph from an input COBOL AST. Data flow graph is built by traversing of a COBOL AST with a visitor

pattern. **CobolDataFlowVisitor** implements visitor methods for most of COBOL AST nodes, which contain a statement-specific logic of the data flow analysis processing. **CobolGraphHelper** is used as its helper class.

Chapter 5

Testing

This chapter describes created unit tests used for a verification of the implemented solution. Two modules containing the program logic, which is tested, are Connector Resolver and Dataflow Generator. Both modules use common classes from Connector Testutils to extend possibilities of testing. JUnit library is used as a driver for unit testing.

5.1 Connector Testutils

Connector Testutils module is a module containing a base class for testing – **CobolTestBase**. This class supplies auxiliary attributes and methods used for testing such as printing an AST to a file or checking if resolving were done.

Method *assertAllReferencesResolved(ICobolAstNode ast, boolean allowDeduction)* is used to assert if all important references of a COBOL AST were resolved (i.e. they are not null). The method implements a visitor pattern for this task. A code snippet of the visitor pattern is shown in Listing 28.

5.2 Connector Resolver

Connector Resolver module is responsible for parsing, resolving and preprocessing of an input COBOL code. All these phases are covered by unit tests.

AstParsingTest is a JUnit parametried test class responsible for testing of the parser and lexer. This test tries to read and parse every file from input test folders. These folders contain many COBOL program examples, which were taken from ProLeap project. It can be said that these tests are very comprehensive because they are taken from COBOL NIST Test Suite [38], which is a test suite used for the verification of a COBOL compiler possibilities. My parser is able to parse most (656) examples but there are still some examples, which it is unable to parse. Those examples contain

```
@Override
    public Object process(DataName node) {
        IResEntity entity = node.getReferencedObject();
        assertResolved(entity, node);
        return super.process(node);
    }

public void assertResolved(IResEntity entity,
    ICobolAstNode node) {
    Assert.assertNotNull("Reference not resolved: "
        + node.toNormalizedString() + " " + node.logNode(),
        entity);

    if (!allowDeduction) {
        Assert.assertTrue(
            "Reference resolved using deduction: " +
            entity.getName() + ", " + node.logNode(),
            entity.getDefinitionSourceType() !=
            DefinitionSourceType.DEDUCTION);
    }
}
```

Listing 28: A code snippet of the visitor for asserting references

unknown constructs, which even GnuCOBOL is not able to recognize. They were placed into the folder *not_working*.

CopyReplaceTest is a unit test responsible for a verification of copying and replacing of copybooks. The test tries to process an input COBOL program with the COPY statement and verifies if a content of copybook is correctly replaced into the input COBOL program.

IBMDataItemAnalyzerTest is a unit test used for testing if the class **IBMDataItemAnalyzer** is working correctly. It uses a set of picture strings and usage types to verify if expected result is same as return value.

The last test class is named **ResolvingTest**. As the name suggest, this class is used for a verification of resolving process. It uses separate methods for testing of different aspects of resolving.

5.3 Dataflow Generator

Dataflow Generator uses a similar approach for testing as Connector Resolver. **ParametrizedCobolScriptTest** is a parametrized JUnit test class responsible for data flow analysis testing. This class is parametrized by an input

COBOL program and its expected data flow graph. The class performs all steps of the data flow analysis and in the end, it checks if the generated data flow graph is same as expected. By this approach, every program from the input folder *parametrized* is checked if its result is the same as expected.

Chapter 6

Data Flow Graph Samples

This chapter shows outputs of the implemented tool for a few COBOL samples. Outputs were generated by the graph rendering program GraphViz [39] and Manta Flow application. The first section shows data flow graphs for a COBOL program with simple statements. The second section presents a graph for a program implementing Sieve of Eratosthenes.

6.1 Simple Program

Listing 29 shows the program which is used for the presentation. In the program, the value 5 is moved to the variable **X**. Then, with the COMPUTE statement, the value of the variable Y is computed as two raised to the power of the variable **X**. The data flow graph of this program is shown in Figure 6.1.

```
1         Identification Division.
2         Program-ID. SAMPLE-PROGRAM1.
3         DATA DIVISION.
4         WORKING-STORAGE SECTION.
5         01 X PIC 9.
6         01 Y PIC 99.
7         Procedure Division.
8         MOVE 5 TO X.
9         COMPUTE Y = 2 ** X.
```

Listing 29: A sample COBOL program

Nodes of the graph have a specific format. Every node is initiated with name, followed with its type in brackets and with the parent name in parenthesis. For example, the first node <**8,14**>**5 [Literal] (SAMPLE-PROGRAM1)** is the node representing literal 5 at position 8:14 (8. line, 14. column) and its

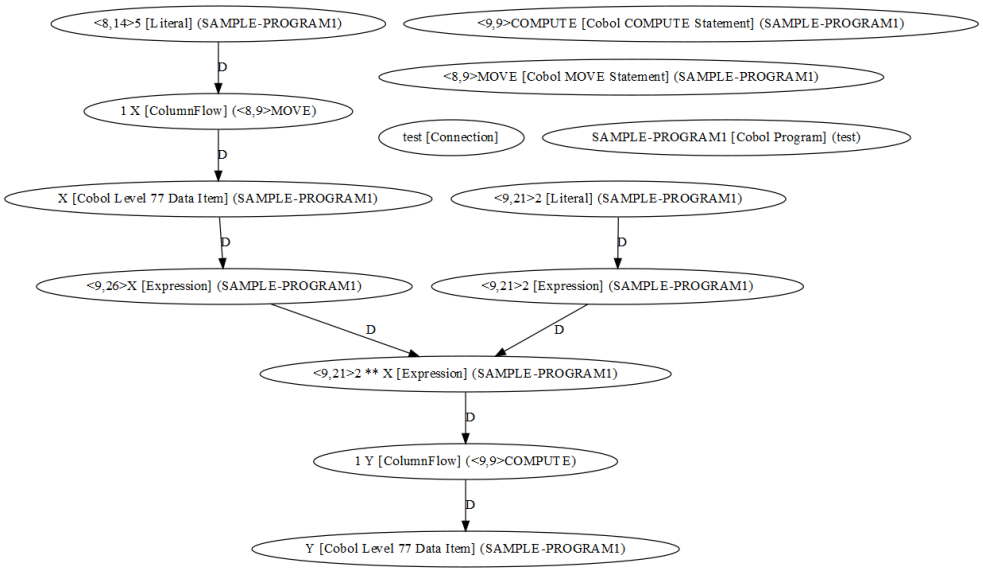

Figure 6.1: A data flow graph of the simple COBOL program visualized by Graphviz

parent is SAMPLE-PROGRAM1. Nodes of this graph are divided into the following types:

- **ColumnFlows.** ColumnFlows represent data flow within a statement. They are typically prefixed with the order to distinguish between same-named nodes.
- **Expressions.** Expression nodes represent expressions or subexpressions. The graph also shows how expressions are grouped together.
- **Literals.** Literal nodes represent literals used in the COBOL program.
- **Cobol Level 77 Data Items.** These nodes represent elementary data items. The type **Cobol Level 77 Data Item** is used for all data items which are not a part of any record.
- **Cobol Program.** The Cobol program node is a node identifying the analyzed program. Every program element is a descendant of this node in the hierarchy.
- **Statements.** Statement nodes represent used statements. Their names are typically prefixed with their location to easily distinguish between multiple usages of the same statement.
- **Connection.** Connection node is the top-level node used to distinguish between multiple runs of the data flow extraction.

Figure 6.2 shows the same program in Manta Flow visualization. The Manta Flow graph is more simplified because Manta Flow application filters out unimportant nodes such as expression and literal nodes.

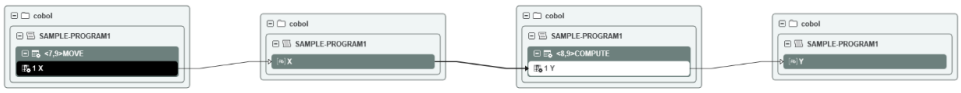

Figure 6.2: A data flow graph of the simple COBOL program visualized in Manta Flow

6.2 Sieve of Eratosthenes Program

Listing 30 shows an implementation of Sieve of Eratosthenes program in COBOL [40]. This program was analyzed by the implemented tool and its data flow graph is present in Figure 6.3. However, the figure does not show the complete graph because the original graph has contained many nodes which would not fit the page. The figure shows data flow between individual data items. This data flow graph contains similar node types as the graph described in Section 6.1. The variable **PRIME** is used as a register for primes. Literals and other variables (**I**, **J**, **K**) are used in the prime computation.

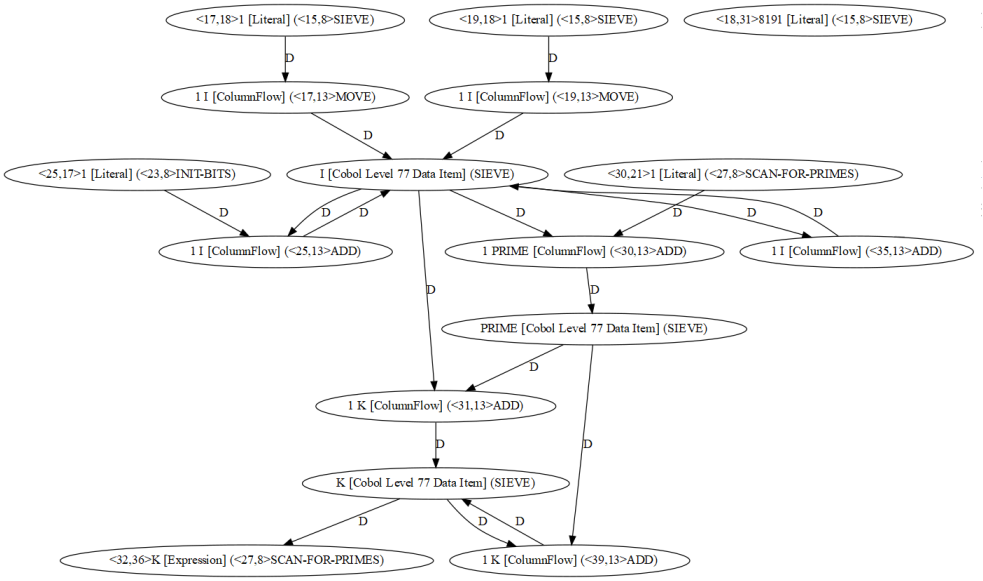

Figure 6.3: A data flow graph in Graphviz for the Sieve of Eratosthenes program

```
1        IDENTIFICATION DIVISION.
2        PROGRAM-ID. SIEVE.
3        DATA DIVISION.
4        WORKING-STORAGE SECTION.
5        77  PRIME                          PIC 9(5) COMP.
6        77  PRIME-COUNT                    PIC 9(5) COMP.
7        77  I                              PIC 9(4) COMP.
8        77  K                              PIC 9(5) COMP.
9        01  BIT-ARRAY.
10            03 FLAG OCCURS 8191 TIMES       PIC 9 COMP.
11       PROCEDURE DIVISION.
12       START-UP.
13            PERFORM SIEVE THROUGH SIEVE-END.
14            STOP RUN.
15       SIEVE.
16            MOVE ZERO TO PRIME-COUNT.
17            MOVE 1 TO I.
18            PERFORM INIT-BITS 8191 TIMES.
19            MOVE 1 TO I.
20            PERFORM SCAN-FOR-PRIMES THROUGH END-SCAN-FOR-PRIMES
21                 8191 TIMES.
22       SIEVE-END. EXIT.
23       INIT-BITS.
24            MOVE 1 TO FLAG (I).
25            ADD 1 TO I.
26       END-INIT-BITS. EXIT.
27       SCAN-FOR-PRIMES.
28            IF FLAG (I) = 0
29                 THEN GO TO NOT-PRIME.
30            ADD I I 1 GIVING PRIME.
31            ADD I PRIME GIVING K.
32            PERFORM STRIKOUT UNTIL K > 8191.
33            ADD 1 TO PRIME-COUNT.
34       NOT-PRIME.
35            ADD 1 TO I.
36       END-SCAN-FOR-PRIMES. EXIT.
37       STRIKOUT.
38            MOVE 0 TO FLAG (K).
39            ADD PRIME TO K.
40       END-PROGRAM. EXIT.
```

Listing 30: An implementation of Sieve of Eratosthenes program in COBOL

Conclusion

Summary

The goal of the thesis was to analyze the syntax and semantics of COBOL programming language with a focus on IBM COBOL dialect, and to learn about Manta project, its data flow analysis process and the data flow representation. IBM COBOL programming language was analyzed with the target to find constructs of the language which are the most important for the data flow analysis process. Manta project, its data flow analysis process and data flow representation were introduced in this work as well.

Other goals were to design a metadata representation for COBOL programming language and design an internal representation of COBOL programming language suitable for a follow-up data flow analysis. The metadata representation and the internal representation for COBOL programming language were designed, and design concepts are described in the design chapter of this work.

Next two goals of this work were to design and implement a prototype of a data flow analyzer for COBOL programs with the intention to detect data flow among variables. The solution was designed, implemented and also tested. The last chapter of this work presented outputs for sample COBOL programs.

The implemented solution contains also additional features that were not a part of the goals of this work. Some of these features were created in a collaboration with other Manta project members. Additional features of the prototype are the support of embedded SQL processing, the data flow analysis to/from files, additional statements, the processing of new COBOL source code formats, and the data flow analysis between COBOL subprograms.

The prototype was also successfully integrated into Manta framework, and it was already released in the alpha version.

Future Work

The implemented solution is a decent data flow analyzer, however, there is still work to do. The implemented solution cannot process more advanced constructs of IBM COBOL such as its object-oriented extensions where a COBOL program is communicating with Java programs. The prototype does not support control flow statements. The support of control flow statements can be an improvement of the implemented prototype, and it will result in a more accurate data flow analysis.

Bibliography

[1] Aho, A.; Lam, M.; et al. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2011, ISBN 9780133002140.

[2] Enterprise COBOL for z/OS Version 6.3. July 2020, accessed on 19.11.2020. Available from: `http://publibfp.boulder.ibm.com/epubs/pdf/igy6lr30.pdf`

[3] Best of 2018: The Beauty of the COBOL Programming Language. 2018, accessed on 21.11.2020. Available from: `https://devops.com/the-beauty-of-the-cobol-programming-language-v2/`

[4] Coughlan, M. *Beginning COBOL for Programmers*. Expert's voice in COBOL, Apress, 2014, ISBN 9781430262541, 3-4 pp.

[5] 2020 Developer Skills Report - HackerRank. 2020, accessed on 17.11.2020. Available from: `https://research.hackerrank.com/developer-skills/2020?utm_medium=content&utm_source=blog&utm_campaign=returnofcobol`

[6] Taulli, T. COBOL Language: Call It A Comeback? July 2020, accessed on 17.11.2020. Available from: `https://www.forbes.com/sites/tomtaulli/2020/07/13/cobol-language-call-it-a-comeback/?sh=2b2b39a37d0f`

[7] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990: pp. 1–84, doi:10.1109/IEEESTD.1990.101064.

[8] Khedker, U.; Sanyal, A.; et al. *Data Flow Analysis: Theory and Practice*. CRC Press, 2017, ISBN 9780849332517, 16-18 pp.

[9] Steenbeek, I. The Basics of Data Lineage. Accessed on 27.11.2020. Available from: `https://www.ewsolutions.com/the-basics-of-data-lineage/`

[10] Data Lineage. Accessed on 20.12.2020. Available from: https://www.techopedia.com/definition/28040/data-lineage

[11] MANTA. Accessed on 28.12.2020. Available from: https://getmanta.com

[12] MANTA - Supported technologies. Accessed on 19.12.2020. Available from: https://getmanta.com/technologies/databases/

[13] Melichar, B.; Janoušek, J.; et al. *Parsing and translation*. Prague: Czech Technical University, 2013, ISBN 978-80-01-05192-4.

[14] ANTLR. Accessed on 23.12.2020. Available from: https://www.antlr.org/

[15] JavaCC. Accessed on 6.1.2021. Available from: https://javacc.github.io/javacc/

[16] The Lex & Yacc Page. Accessed on 6.1.2021. Available from: http://dinosaur.compilertools.net/

[17] GNU Bison. Accessed on 6.1.2021. Available from: https://www.gnu.org/software/bison/

[18] Parr, T.; Harwell, S.; et al. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. *SIGPLAN Not.*, volume 49, no. 10, Oct. 2014: p. 579–598, ISSN 0362-1340, doi:10.1145/2714064.2660202. Available from: https://doi.org/10.1145/2714064.2660202

[19] GnuCOBOL — Sourceforge.net. Accessed on 19.11.2020. Available from: https://sourceforge.net/projects/gnucobol

[20] IBM COBOL. Accessed on 19.11.2020. Available from: https://developer.ibm.com/languages/cobol/

[21] Micro Focus COBOL. Accessed on 19.11.2020. Available from: https://www.microfocus.com/en-us/products/cobol-development/overview

[22] FUJITSU Software NetCOBOL. Accessed on 19.11.2020. Available from: https://www.fujitsu.com/global/products/software/developer-tool/netcobol/

[23] IBM COBOL compilers by name and version. Oct 2020, accessed on 18.11.2020. Available from: https://www.ibm.com/support/knowledgecenter/SS6SG3_6.3.0/migrate/igympreab2.html

[24] Enterprise COBOL for z/OS, Version 4.2, Compiler and Runtime Migration Guide. Accessed on 18.11.2020. Available from: https://www.ibm.com/support/knowledgecenter/SS6SG3_4.2.0/com.ibm.entcobol.doc_4.2/MG/igymch1001.htm

[25] Industry specifications for Enterprise COBOL for z/OS Version 6.3. Oct 2020, accessed on 19.11.2020. Available from: https://www.ibm.com/support/knowledgecenter/SS6SG3_6.3.0/lr/ref/rlind.html

[26] Chapter 1: Introduction to the COBOL Language. Accessed on 20.11.2020. Available from: https://supportline.microfocus.com/Documentation/books/sx40/lrintr.htm

[27] FUJITSU Software BS2000 COBOL85. Accessed on 20.11.2020. Available from: https://www.fujitsu.com/fts/products/computing/servers/mainframe/bs2000/software/programming/cobol85.html

[28] Sample COBOL program. Accessed on 20.11.2020. Available from: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.ceea800/cblex1.htm

[29] Subscripting. Accessed on 20.11.2020. Available from: https://www.microfocus.com/documentation/visual-cobol/vc60/DevHub/HRLHLHCLANU045F002.html

[30] Java Cobol Lexer — Sourceforge.net. Accessed on 22.12.2020. Available from: https://sourceforge.net/projects/javacobollexer/

[31] RES - An Open Cobol To Java Translator — Sourceforge.net. Accessed on 22.12.2020. Available from: https://sourceforge.net/projects/opencobol2java/

[32] Github - TypeCobol. Accessed on 22.12.2020. Available from: https://github.com/TypeCobolTeam/TypeCobol

[33] Github - ProLeap ANTLR4-based parser for COBOL. Accessed on 22.12.2020. Available from: https://github.com/uwol/proleap-cobol-parser

[34] Parr, T. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, second edition, 2013, ISBN 1934356999.

[35] Github - Spring Framework. Accessed on 23.12.2020. Available from: https://github.com/spring-projects/spring-framework

[36] Apache Maven Project. Accessed on 23.12.2020. Available from: https://maven.apache.org/

[37] JUnit 5. Accessed on 23.12.2020. Available from: https://junit.org/junit5/

[38] COBOL Test Suites. Accessed on 1.1.2021. Available from: https://www.itl.nist.gov/div897/ctg/cobol_form.htm

[39] Graphviz - Graph Visualization Software. Accessed on 4.1.2021. Available from: https://graphviz.org

[40] Sieve of Eratosthenes example COBOL program. Accessed on 4.1.2021. Available from: https://www.roug.org/retrocomputing/languages/cobol/microfocus/sieve-of-eratosthenes-cbl

Appendix A

Acronyms

ANSI American National Standards Institute

AST Abstract Syntax Tree

ANTLR ANother Tool for Language Recognition

BI Business intelligence

CFG Control Flow Graph

COBOL Common Business Oriented Language

CODASYL Conference of Data Systems Languages

CST Concrete Syntax Tree

DAG Directed Acyclic Graph

IEEE Institute of Electrical and Electronics Engineers

ISO International Organization for Standardization

EBNF Extended Backus–Naur Form

GNU GNU's Not Unix!

JSON JavaScript Object Notation

LOC Lines of code

PC Personal Computer

OS Operating system

XML Extensible Markup Language

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... source codes
  - `impl` ..... source codes of the implemented modules
    - `connector` ..... source codes of Connector modules
    - `dataflow` ..... source codes of Dataflow modules
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format