Assignment of bachelor's thesis

Title:	Dataflow analysis in Azure Data Factory
Student:	Jan Chybík
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of winter semester 2022/2023

Instructions

Study the ETL tool Azure Data Factory. Study the syntax and semantics of source codes that describe data transformations in Azure Data Factory.
Familiarize with the Manta project and its way of representing dataflows.
Describe the syntax and analysis of source codes that describe data transformations in Azure Data Factory and describe a way to analyze and represent them.
Design analysis of source codes that describe data transformations in Azure Data Factory capable of detection of its internal dataflows and dataflows from and to tools that integrate with Azure Data Factory.
Implement a proof-of-concept tool that is able to extract dataflows from Azure Data Factory source codes located in a git repository into the Manta system.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 13 September 2021 in Prague.

Bachelor's thesis

DATAFLOW ANALYSIS IN AZURE DATA FACTORY

Jan Chybík

Faculty of Information Technology
Computer science
Supervisor: Ing. Jan Trávníček, Ph.D.
May 10, 2022

Contents

List of Figures

List of Tables

I would like to thank my supervisor Ing. Jan Trávníček, Ph.D. for his leadership and great help, manta reporting team for their shared domain expertise, my family for the opportunity on this world, and finally my girlfriend for all the emotional support. When the times were dire you were all there for me. Thank you.

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 May 10, 2022 . . . .

Abstrakt

Tato práce se zaměřuje na analýzu datových toků v nástroji Azure data factory. Zkoumá reprezentaci datových toků projektu Manta. Prozkoumává a popisuje zdrojové soubory Azure data factory a následně navrhuje způsob, jak analyzovat tyto zdrojové kódy. Všechna zjištění pak využívá při implementaci prototypu skeneru, který bude součástí projektu Manta.

Klíčová slova Azure data factory, analýza datových toků, parsování data flow skriptu, data lineage, Manta Tools, s.r.o.

Abstract

This work focuses on data flow analysis of Azure data factory. It examines data flow representation in the Manta project. It explores and describes azure data factory source codes and designs a solution on how to analyze those source codes. In the end, it utilizes all findings to implement a proof-of-concept scanner, which will be part of the Manta project

Keywords Azure data factory, dataflow analysis, data lineage, data flow script parsing, Manta Tools, s.r.o.

Abreviations

ADF	Azure data factory
AST	Abstract syntax tree
DFR	Data flow resource
DFS	Data flow script
EBNF	Extended Backus–Naur form
ETL	Extract, transform, load
IR	Intermediate representation
NPDA	Non-deterministic pushdown automaton
PDA	Pushdown automaton
PDS	Push-down store

Chapter 1
Goals

This bachelor thesis aims to analyze the ETL tool Azure data factory (ADF) and implement a new proof-of-concept scanner for the Manta project, capable of detecting internal dataflows and dataflows from and to tools that integrate with Azure Data Factory.

The theoretical part of the thesis will focus on analyzing Manta and its dataflow representation, analyzing ADF its syntax and semantics of source codes that describe data transformations.

The practical part of the thesis will present a way to analyze and represent ADF data transformations. It will describe the implementation of a proof-of-concept scanner capable of extracting dataflows from ADF source codes located in git.

The first step will be to analyze the Manta project and its representation of dataflows. Then the thesis will focus on the study of ADF and research the syntax and semantics of source codes that describe data transformations. It will explain the syntax and analysis of source codes that define data transformations in Azure Data Factory and design a way to analyze source codes mentioned above, capable of detecting its internal dataflows and dataflows from and to tools that integrate with Azure Data Factory. In the end, it will show the implementation of a proof-of-concept tool that can extract dataflows from ADF located in a git repository into the Manta system.

The final result and benefit of the thesis will be a new scanner for Manta. Users of Manta will be able to use the scanner to analyze their ADF data flows and get the end-to-end lineage of their data environment.

Chapter 2

Introduction

Today's world is governed by data. It comes in all kinds of shapes and forms. Banks keep information about bank accounts, social networking sites keep information about their users, a spreadsheet can be created in order to manage to manage one's home finances. Everybody is dealing with data on day-to-day bases. But there is one big difference in how ordinary people and companies regard data. The volume.

Data represents some information someone wants to store, something important. We as humans can understand that. There is some object, and we have information about the object. It could be a name. It could be a purpose. But companies don't store three or five pieces of information. They need dozens and hundreds of specific information. And lots of it depends on other, like in a profile on social media, age is calculated from the date of birth, or a simple yes or no meaning whether a bank will give you a loan originates from a lots of variables. Data dependencies and transformations are very common. Companies should not be lost in it.

Organization's data is a strategic asset. Just like finances and customer relationships, it needs proper management. However even in industries like finance, where there's a need to comply with various regulatory requirements, organizations tend to let data governance slip through the cracks. Because of that, errors build up. When critical data is disorganized, organizations can face penalties for not complying with regulations, rising costs for storing and managing duplicate data, and other expenses. Moreover, they cannot be sure their business decisions are based on correct information. To minimize those risks, organizations need proper data governance. [1].

Software developed by Manta helps with those problems. It provides data lineage of the company's entire data environment. *Data lineage represents a detailed map of all direct and indirect dependencies between data entities in the environment.* [2, p. 3] Manta supports over 40 technologies across Modeling, Data integration, programming languages, databases, reporting, and analysis tools. It helps to understand complex data hierarchies and dependencies.

One of such tools, Manta does not support yet is the Azure data factory (ADF). *ADF is Azure's cloud ETL service for data integration and data transformation.* [3] *ETL stands for Extract, Transform and Load. It is a three-step process that extracts data from data sources, transforms it into a format satisfying the operational and analytical requirements of the business, and loads it to a target destination.* [4] ADF is a popular cloud alternative to on-premise solutions, mainly in companies already using azure services. That is why Manta should support ADF to provide even better data lineage.

Chapter 3

Definitions

This part of the work will clarify the following chapter's terms and notions. It will explain general concepts in data lineage and language processing.

3.1 Directed graph

A directed graph (shortly digraph) is an ordered pair (V, E)*, where* V *is a nonempty finite set of vertices (or nodes), and* E *is a set of directed edges (or also arcs). A directed edge* $(u, v \in E)$ *is an ordered pair of distinct vertices* $u, v \in V$*. Thus* $E \subseteq V \times V$*. We say that* u *is a predecessor of* v*, and* v *is a successor of* u*.* [5] An example of a directed graph is in figure 3.1.

3.2 Data flow graph

One can imagine data flow as a directed graph where each vertex is an endpoint or some sort of transformation step data has to move through. *A data flow is a path for data to move from one part of the information system to another. A data-flow may represent a single data element such as the Customer ID or it can represent a set of data elements (or a data structure).* [6] All data flows joined into one big picture are called *data lineage*.

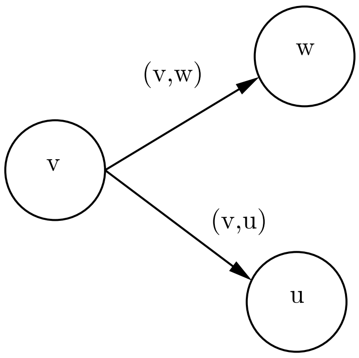


Figure 3.1 Example of directed graph

3.3 Git

Git is an open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. [7] It allows to save separate versions of files clearly and efficiently. Git is used heavily in the programming world because it was developed primarily as a way to version code. However, it can be used on other mainly text-based resources. One could wonder if the git allows sharing files with other people. Nevertheless, the git is not a platform. It is a *system*, a way to do version control. It does not directly offer data sharing. And because it is distributed system, there is no central storage of files. To share files in a centralized way, one has to install git on a server. The server becomes the main source of git repositories. Nowadays, having own server is not always the best solution. There are cloud services that offer to host one's git repositories.

Github is one of those services. ADF uses Github to store source codes of its transformations. ADF user has to provide a repository to store source codes and authorize the ADF application. After this, the user can save and version everything they do in ADF.

3.4 Basic constructs

In order to better understand more complex concepts of language processing later in this work, basic constructs have to be defined first.

3.4.1 Alphabet

An alphabet is a finite set of symbols. Usually, symbols Σ or T represent the alphabet. An example of an alphabet is binary code or the English alphabet. However, an element of an alphabet does not need to be only one letter or number. Alphabet can consist of arbitrary objects. An example of such an alphabet is alphabet $\Sigma = \{yes, no\}$.

3.4.2 String over an alphabet

String is a finite sequence of symbols of an alphabet[8]. An empty sequence is indicated by ε. Example of a string over and alphabet $\Sigma = \{0, 1\}$ is *101010*.

Because writing always "*String over and alphabet*" is tedious and not effective, it can be expressed more efficiently by utilizing special symbols $*$ and $+$. Set of all strings over an alphabet Σ is Σ^*. Set of all non empty strings over the alphabet Σ is represented by Σ^+. Based on the previous definitions it is true that $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

3.4.3 Language

Formal language L (a set of strings) over an alphabet Σ is defined as $L \subseteq \Sigma^*$. [8] For instance, the language of english words without special characters is a formal language over an alphabet $\{a, b, c \ldots, z\}$. Another example of a formal language is a set of binary numbers that end with zero. Both languages are infinite because one can put together any number of strings. Although the first language would not be correct according to English grammar, it would still be considered a formal language. In order to better work with languages there are few operations:

- *concatenation - $L = L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$ (L is defined over alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$).*
- *n-th power of language $L : L^n = L \cdot L_{n-1}, L^0 = \{\varepsilon\}$.*
- *Kleene star L^* of language $L : L^* = \bigcup_{n=0}^{\infty} L^n$.*

- *Kleene plus L^+ of language $L : L^+ = L \cdot L^*$.*

[8, p. 3]

3.5 Grammar

Grammar generates a language. It is defined as quadruple $G = (N, \Sigma, P, S)$, where

- *N is a finite set of non-terminal symbols. "They are integral to the grammar and allow expressing the internal structure of generated sentences*" [9].
- *Σ is a finite set of terminal symbols . ($N \cap \Sigma = \emptyset$, denoted also by T).*
- *P is a set of production rules. It is a finite subset of $(N \cup \Sigma)^* \cdot N \cdot (N \cup \Sigma)^* \times (N \cup \Sigma)^*$, (element (α, β) of P is written as $\alpha \to \beta$ and called a rule).* The dot symbol $\cdot$ expresses string concatenation.
- *$S \in N$ is the start symbol of the grammar.*

[8, p. 12]

An element of generated language is called a *sentence*. Following grammar G is an example of a grammar that generates language with sentences *abba* and *aca*.

$$
\begin{aligned}
&G = (\{S, B, C\}, \{a, b, c\}, P, S) \\
&P = \{ \\
&\quad S \to aBa, \\
&\quad S \to aCa, \\
&\quad B \to bb, \\
&\quad C \to c \\
&\}
\end{aligned}
$$

In order to define what *generating* language means, there is a couple of additional definitions that need to be clarified.

The first is *derivation in one step*. "*$G = (\{S, B, C\}, \{a, b, c\}, P, S), x, y \in (N \cup \Sigma)^*$ We say that x derives y in one step ($x \Rightarrow y$) if there exists $(\alpha, \beta) \in P$ and $\gamma, \delta \in (N \cup \Sigma)^*$ such that $x = \gamma\alpha\delta, y = \gamma\beta\delta$* (i.e., *$\gamma\alpha\delta \Rightarrow \gamma\beta\delta$*)."[8, p. 15] In other words, derivation in one step is an application of a production rule. If the rule is applied to an $x \in (N \cup \Sigma)^*$ then an appearance of the left-hand side of the rule in x will be replaced by the rule's right-hand side. Example of derivation in one step:

$$
\begin{aligned}
&G = (\{S, B\}, \{a, b\}, \{S \to baBa, B \to abBa, B \to \varepsilon\}, S) \\
&baBa \Rightarrow baabBaa \quad (baBa \text{ derives } baabBaa \text{ in one step})
\end{aligned}
$$

Derivation in one step is helpful, but the notation would be pretty extensive if it would require to derive only one step at a time. The second type of derivation extends the first to do multiple steps. "*$\alpha \Rightarrow^k \beta$ if there exists a sequence $\alpha_1, \alpha_2, \ldots, \alpha_k$ for $k \geq 0$ of $k+1$ strings such that $\alpha = \alpha_0, \alpha_{i-1} \Rightarrow \alpha_i$ for $1 \geq i \geq k$, and $\alpha_k = \beta$. This sequence is called derivation of string β from string α that has lenght k in grammar G.* "[8, p. 15] We can say the string α derives β in k steps.

Similar to strings symbols $*$ and $+$ can be utilized to capture any number of lenghts. It is called transitive and reflexive closure. The $+$ symbol, transitive closure, is used for $k \geq 1$. *$\alpha \Rightarrow^+ \beta$ if $\alpha \Rightarrow^i \beta$ for some $i \geq 1$. "We read '$\Rightarrow^+$' as 'derives in nonzero number of steps*". [8, p. 16] The $*$ symbol, transitive and reflexive closure, is used for $k \geq 0$. *$\alpha \Rightarrow^* \beta$ if $\alpha \Rightarrow^i \beta$*

for some $i \geq 0$. "We read '$\Rightarrow^*$' as 'derives in any number of steps'". [8, p. 16] Using the same grammar G from previous example the following is true:

$$S \Rightarrow^* baabBaa$$
$$(S \text{ derives } baabBaa \text{ in any number of steps})$$

Last crucial step is defining *sentential form*. *"String $\alpha \in (N \cup \Sigma)^*$ is called a sentential form if $S \Rightarrow^* \alpha$"[8, p. 17]*. In words, a sentential form is a sequence of the terminal and non-terminal symbols that can be derived in any number of steps from the start symbol. A particular case of a sentential form consisting only of terminal symbols is called a *sentence generated by a grammar G*.

The set of all sentences generated by the grammar G is called the *language generated by the grammar G*. Formally: *$L(G) = \{\omega : \omega \in \Sigma^*, \exists S \Rightarrow^* \omega\}$ is the language generated by grammar $G = (N, \Sigma, P, S)$*. [8, p. 18]

3.5.1 Grammar clasification

Grammars are classified into four main types.

0. *Unrestricted (Type 0) - It satisfies general grammar definition*
1. *Context-sensitive (Type 1) - Every rule from P is of the form $\gamma A \delta \to \gamma \alpha \delta$, where $\gamma, \delta \in (N \cup \Sigma)^*, \alpha \in (N \cup \Sigma)^+, A \in N$, or the form $S \to \varepsilon$ in case that S is not present in the right-hand side of any rule*
2. *Context-free (type 2) - Every rule is of the form $A \to \alpha$, where $A \in N, \alpha \in (N \cup \Sigma)^*$*
3. *Regular (type 3) - Every rule is of the form $A \to aB$ or $A \to a$, where $A, B \in N, a \in \Sigma$, or the form $S \to \varepsilon$ in case that S is not present in the right-hand side of any rule*

[8, p. 5] Languages generated by grammars are classified accordingly. We say language is:

0. *Recursively enumerable (Type 0) - if exists unrestricted grammar which generates it. This class of languages is accepted by the Turing machine*
1. *Context-sensitive (Type 1) - if $\exists$ context-sensitive grammar which generates it. This class of languages is accepted by linear bounded turing machine.*
2. *Context-free (type 2) - If $\exists$ context-free grammar which generates it. This class of languages is accepted by nondeterministic pushdown automaton.*
3. *Regular (type 3) - if $\exists$ regular grammar which generates it. This class of languages is accepted by finite automaton.*

[8, p. 6] Formal languages follow hierarchy shown in figure 3.2.

3.5.2 Regular grammar

Regular grammars are the simplest subset of formal grammars. For each regular grammar exists a finite automaton accepting the language generated by the grammar. [8, p. 6] Actually, regular grammars and finite automatons are equivalent because, for each finite automaton, there also is a regular grammar generating the same language. Nevertheless, there is one more equivalent formalism. It is called regular expression, and it offers another method of, this time, describing language.

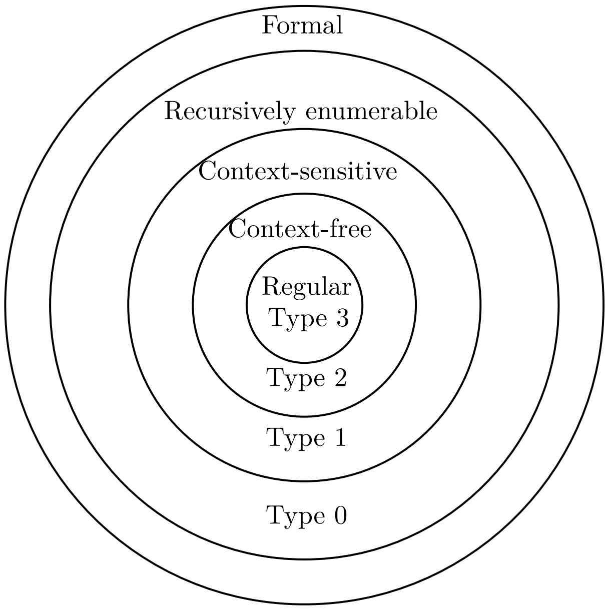


Figure 3.2 Hierarchy of formal languages [8]

3.5.3 Regular expression

Regular expression V over and alphabet Σ is defined as follows:

1. *$\emptyset, \varepsilon, a$ are regular expressoins for all $a \in \Sigma$.*
2. *If x, y are regular expressions over Σ, then:*
   - *$(x + y)$ (union, alternation),*
   - *$(x \cdot y)$ (concatenation), and*
   - *$(x)^*$ (Kleene star)*

   are regular expressions

Value $h(x)$ of regular expression x is defined as follows:

1. $h(\emptyset) = \emptyset, h(\varepsilon) - \{\varepsilon\}, h(a) - \{a\}, a \in \Sigma$
2. $h(x + y) = h(x) \cup h(y)$
3. $h(x \cdot y) = h(x) \cdot h(y)$
4. $h(x^*) = (h(x))^*$ *, were x, y are regular expressions*

[10]
Regular expressions are useful in text processing because they are a quick and short way of defining language. Regular expression generating language of binary numbers ending with zero looks like this: $(0 + 1)^*0$. In this example, it even generates sequences that start with zeroes. Regular expression has to be modified to limit the language to sequences with one at the beginning, like so: $1(0 + 1)^*0$.

3.5.4 Context-free grammar

Context-free grammars generate most of the syntactic structures of programming languages. Another benefit of context-free grammars is that there are known efficient algorithms for the analysis of sentences of context-free languages. [11] For both of those reasons, they are very practical. Each context-free language is accepted by a non-deterministic pushdown automaton. The automaton can be found by methods this work will focus on in chapter 4.

3.6 Parse

Parse of a sentential form α in a grammar G is the sequence of rule numbers used in derivation $S \Rightarrow^* \alpha$. Where $G = (N, \Sigma, P, S)$ and rules of set P are numbered from 1 to $|P|$. [12, p. 5 translated by me] In other words it is a sequence of applied rules from the start symbol during the generation of a sentential form and ultimately the sentence. When rules of the grammar G are numbered starting from one:

$$\begin{aligned}
&G = (\{S, B, C\}, \{a, b, c\}, P, S)\\
&P = \{\\
&\quad 1\,|S \rightarrow aBa,\\
&\quad 2\,|S \rightarrow aCa,\\
&\quad 3\,|B \rightarrow bb,\\
&\quad 4\,|C \rightarrow c\\
&\}
\end{aligned}$$

then the parse for each sentential form of grammar G can be seen in table 3.1. One sentential form can have multiple parses. This is not visible in the grammar G. However next grammar $G1$, even it is simpler than the previous grammar G, has multiple parses for the sentence ab.

$$\begin{aligned}
&G1 = (\{S, A, B\}, \{a, b\}, P, S)\\
&P = \{\\
&\quad 1\,|S \rightarrow AB,\\
&\quad 2\,|A \rightarrow a,\\
&\quad 3\,|B \rightarrow b\\
&\}
\end{aligned}$$

The sentence ab has two parses: (1,2,3) and (1,3,2). However, the sentence has only one left parse and one right parse.

3.6.1 Left parse

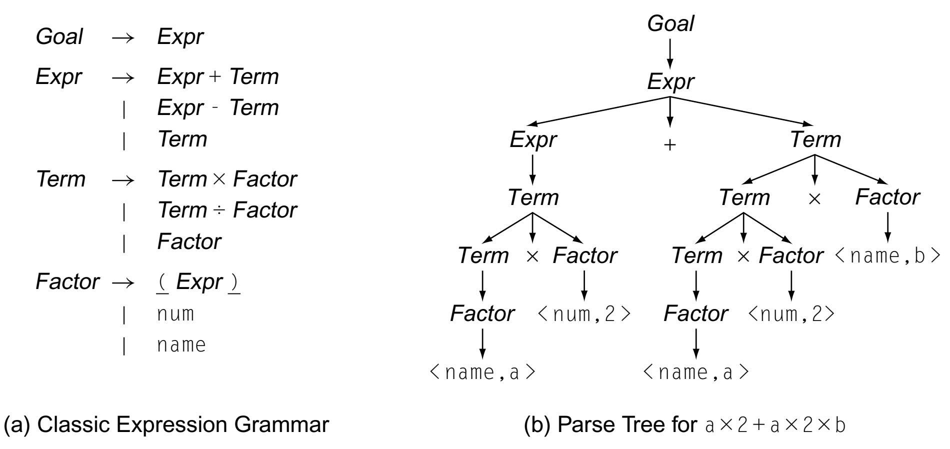
Left parse is a parse, with only the leftmost non-terminal symbol's rewrite rule used in each derivation step. It means the left parse of the sentence ab from the last example of the previous section 3.6 is (1,2,3). Derivations were done in the following order: $S \Rightarrow AB \Rightarrow aB \Rightarrow ab$. A had to be rewritten before B because it is more to the left.

3.6.2 Right parse

Right parse is an alternative to a left parse but reversed. Derivation uses only the rightmost non-terminal symbol for a rewrite. The result is that ab also has only one right parse. It is (1,3,2) $S \Rightarrow AB \Rightarrow Ab \Rightarrow ab$.

Sentential form	Parse
aBa	1
aCa	2
abba	1,3
aca	2,4

Table 3.1 Derivations of grammar



$$\begin{array}{lcl} \textit{Goal} & \rightarrow & \textit{Expr} \\ \textit{Expr} & \rightarrow & \textit{Expr} + \textit{Term} \\ & | & \textit{Expr} - \textit{Term} \\ & | & \textit{Term} \\ \textit{Term} & \rightarrow & \textit{Term} \times \textit{Factor} \\ & | & \textit{Term} \div \textit{Factor} \\ & | & \textit{Factor} \\ \textit{Factor} & \rightarrow & \underline{(}\ \textit{Expr}\ \underline{)} \\ & | & \texttt{num} \\ & | & \texttt{name} \end{array}$$

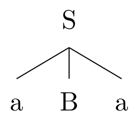
(a) Classic Expression Grammar

(b) Parse Tree for a × 2 + a × 2 × b

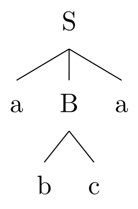
Figure 3.3 Example of a parse tree [13, p. 226 figure 5.1]

3.6.3 Parse tree

A parse tree or derivation tree shows the structure of a sentence in a context-sensitive grammar. It is a graphic representation of a parse. The root of the tree is the start symbol. Inner vertexes are non-terminal symbols, and leaves are terminal symbols or ε. Each rule application is represented as a parent vertex for the left-hand side of the rule. Its children are on the right-hand side of the rule. This means that if the rule $S \rightarrow aBa$ is applied, then the resulting tree would have S as a parent with children a, B, a. Like so:



If the B has a rule $B \rightarrow bc$, the tree is going to look like this.



This is the parse tree of sentence *abca*. This is an elementary example. More complex parse tree that could be actually used in practice is in figure 3.3.

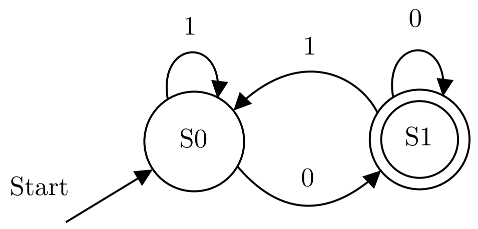


Figure 3.4 Finite automaton accepting binary numbers ending with 0 [15]

3.7 Deterministic finite automaton

Deterministic finite automaton (DFA) is a structure $M = (Q, \Sigma, \delta, q_0, F)$*, where*

- Q *is a finite set; elements of* Q *are called states;*
- Σ *is a finite set, the input alphabet;*
- $\delta : Q \times \Sigma \rightarrow Q$ *is the transition function (recall that* $Q \times \Sigma$ *is the set of ordered pairs* $((q, a)|q \in Q$ *and* $a \in \Sigma)$*. Intuitively,* δ *is a function that tells which state to move to in response to an input: if* M *is in state* q *and sees input* a*, it moves to state* $\delta(q, a)$*.*
- $q_0 \in Q$ *is the start state;*
- F *is a subset of* Q*; elements of* F *are called accept or final states.*

[14, p. 15]

A finite automaton is a simple computation model with a reading head, states, and memory that stores the current state. Figure 3.4 is a finite automaton accepting binary numbers ending with 0. Automaton is defined as:

$$
\begin{aligned}
&M = (\{S0, S1\}, \{0, 1\}, \delta, S0, \{S1\}) \\
&\delta : \\
&\quad \delta(S0, 0) = S1 \\
&\quad \delta(S0, 1) = S0 \\
&\quad \delta(S1, 0) = S1 \\
&\quad \delta(S1, 1) = S0
\end{aligned}
$$

To formally define what "to accept a language" means, the work has to define *configuration* of a finite automaton and a *move.*

Configuration of finite automaton M *is a pair* $(q, w) \in (Q, \Sigma^*)$*, where the finite automaton is* $M = (Q, \Sigma, \delta, q_0, F)$. It tells what state the automaton is in and what it has on its input. *The configuration* (q_0, w) *is called the initial configuration of an automaton M. Configuration* (q, ε) *is called accepting configuration of automaton* M *iff* $q \in Q$*.* [16, p. 1] The name *accepting configuration* is already telling, that it will have something to do with accepting. Accepting configuration is the configuration where the current state is one of the final states and nothing is on the input. *Invalid configuration* $(S, \alpha\gamma), S \in Q, \alpha \in \Sigma, \gamma \in \Sigma^*$ is a configuration for which $\delta(S, \alpha)$ is not defined.

Now, when the configuration is succesfully defined a *move* can be defined as well. " $\vdash_M$ *is relation over* $Q \times \Sigma^*$ *(i.e,. subset of* $(Q \times \Sigma^*) \times (Q \times \Sigma^*)$*) such that* $(q, w) \vdash_M (p, w')$ *iff* $w = aw'$ *and* $\delta(q, a) = p$ *for some* $a \in \Sigma, w \in \Sigma^*$*. An element of relation* $\vdash_M$ *is called a move in a automaton* M*.* ". [16, p. 1] In other words, a move is a possible transition between states for a given input. The move can be extended to express any number of back-to-back moves. It is denoted by $\vdash_M^*$ and it is called *transitive and reflexive closure* of relation $\vdash_M$. With this closure, string acceptance can be comfortably defined.

"*String* $w \in \Sigma^*$ *is accepted by a deterministic finite automaton* $M = (Q, \Sigma, \delta, q_0, F)$ *if* $\exists (q_0, w) \vdash_M^* (q, \varepsilon)$ *for some* $q \in F$"[16, p. 2]. The string is accepted if a finite number of steps from the initial configuration to some accepting configuration exists. Language L accepted by the automaton M is set of all strings over an alphabet Σ which are accepted by the automaton M. Formally written as $L(M) = \{w : w \in \Sigma^*, \exists q \in F : (q_0, w) \vdash_M^* (q, \varepsilon)\}$. [16, p. 2] String is not accepted when the automaton gets into invalid configuration.

Now when all is clear, the automaton from figure 3.4 can be revisited. The following paragraph will illustrate automaton execution on string *1000101*: At the start, the automaton will be in initial configuration $(S0, 1000101)$. The current state is S0, and the first symbol of input is 1. The next state will be the result of $\delta(S0, 1)$. $\delta(S0, 1) = S0$ so the automaton will execute move $(S0, 1000101) \vdash_M (S0, 000101)$. The current state is still the same as the previous one, but the reading head moved by one position to the right. The automaton will repeat these steps while there is something to read. $(S0, 000101) \vdash (S1, 00101) \vdash (S1, 0101) \vdash (S1, 101) \vdash (S0, 01) \vdash (S1, 1) \vdash (S0, \varepsilon)$. The automaton is now at the end of the input. The automaton will accept the input sentence if the current state is an element of F. However, in this case, $S0$ is not in the set of final states. The automaton will reject the string *1000101*, and so it does not belong to the language accepted by the automaton.

3.8 Pushdown automaton

Non-deterministic pushdown automaton is tuple with seven elements. $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$ *where*

- Q *is a finite non-empty set of states.*
- Σ *is a finite input alphabet.*
- G *is a finite non-empty alphabet of PDS symbols.*
- δ *is transition mapping from* $Q \times (\Sigma \cup \{\epsilon\} \times G^*)$ *to a set of finite subsets of a set* $Q \times G^*$.
- $q_0 \in Q$ *is the start state.*
- $Z_0 \in G$ *is the start symbol in PDS.*
- $F \subseteq Q$ *is a set of final states. [17, p. 156 translated byme]*

PDA accepts input sentence if the whole input is read and one of the following is true:

- *PDA is in a final state (If it accepts by the transition to the final state).*
- *PDA is in an arbitrary state, and its PDS is empty (If it accepts by an empty PDS).*

[17, p. 159 translated by me]

A pushdown automaton is an extension of the finite automaton with additional memory. In addition to the memory that stores the current state, it has an infinite push-down store (PDS). However, the information can be stored and retrieved only from the top of PDS. The PDA can accept more complex languages than a finite automaton. Example of pushdown automaton is in figure 3.5. The notation differs a little from the finite automaton. It has form $\alpha|\beta \to \gamma$ where $\alpha \in \Sigma, \beta, \gamma \in (\Sigma \cup \{\varepsilon\} \times G^*)$. The α is the required input. β is the required top of the PDS and γ is what will be pushed to PDS instead of β. The automaton moves to the next state if the input symbol and top of the PDS correspond to the expected α and β. Then the top of PDS β is replaced by the γ. Automaton in the figure accepts by transition to the final state and it

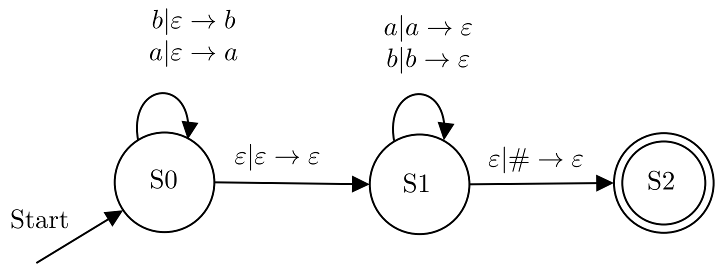


Figure 3.5 Example of pushdown automaton accepting palindromes of even length [9]

accepts palindromes of even length. To look at the automaton execution, configuration of PDA has to be defined first.

Configuration of PDA is tuple of three items (q, w, s) where $q \in Q, w \in \Sigma^*, s \in G^*$. q is the current state, w is unread input, s is PDS content with the top on the left. Initial configuration is (q_0, w, Z_0) $w \in \Sigma^*$.

- *$\delta(q, a, \alpha) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$: PDA in state q reads symbol a, moves into state $p_i, i \in \{1, 2, \ldots, m\}$, and string α on top of the PDS is replaced by string γ_i.*
- *$\delta(q, \varepsilon, \alpha) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_m, \gamma_m)\}$: transition into a new state and change of PDS content without reading of the input symbol.* [18, p. 2]

The language defined (accepted) by PDA $R = (Q, \Sigma, G, \delta, q_0, Z_0, F)$

1. by transition into a final state $L(R) = \{w : w \in \Sigma^*, \exists\gamma \in G^*, \exists q \in F, (q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma)\}$
2. by empty pushdown store $L_\varepsilon(R) = \{w : w \in \Sigma^*, \exists q \in Q, (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)\}$

With the use of a configuration, the execution of the PDA can be properly represented. Definition of the PDA from the figure 3.5:

$$\begin{aligned}
&M = (\{S0, S1, S2\}, \{a, b\}, \{a, b, \#\}, \delta, S0, \#, \{S2\})\\
&\delta :\\
&\quad \delta(S0, (a, \varepsilon)) = (S0, a)\\
&\quad \delta(S0, (b, \varepsilon)) = (S0, b)\\
&\quad \delta(S0, (\varepsilon, \varepsilon)) = (S0, \varepsilon)\\
&\quad \delta(S1, (a, a)) = (S0, \varepsilon)\\
&\quad \delta(S1, (b, b)) = (S0, \varepsilon)\\
&\quad \delta(S1, (\varepsilon, \#)) = (S2, \varepsilon)
\end{aligned}$$

For this PDA and input string *abba* the execution will look as follows. Initial configuration is $(S0, abba, \#)$. Because current state is $S0$, first symbol of input is a and the top of PDS is $\#$, the next configuration will be defined by $\delta(S0, a, \varepsilon)$. $\delta(S0, a, \varepsilon) = (S0, a)$ determines that next configuration will be $(S0, bba, a\#)$. This process repeats until the automaton has input to read: $(S0, bba, a\#) \vdash (S0, ba, ba\#) \vdash (S1, ba, ba\#) \vdash (S1, a, a\#) \vdash (S1, \varepsilon, \#) \vdash (S2, \varepsilon, \varepsilon)$. The PDA M accepts by a transition to a final state. State $S2$ is final so M accepts the string *abba*. During the PDA's run, there were moments when the next step was not clear because there were two ways how the automaton could move. This is because PDA M is not deterministic. Example of this is configuration $(S0, bba, a\#)$. The automaton can move to $(S1, bba, a\#)$ the same way it can move to $(S0, ba, ba\#)$. At this point of the work, it is not important, but in the next chapter 4, non-determinism will be a problem the work has to deal with.

Chapter 4
Parsing

Parsing, also known as syntactic analysis, is the process of analyzing and understanding the internal structure of sentences of a language. Many Manta scanners use parsers to create a language's abstract syntax tree (AST). AST is a data structure describing the internal structure of a sentence. An example of an arithmetic expression AST is in the figure. 4.1. However, building an AST is the last step of parsing. Although parsing could be done in one big step, it is usually divided into multiple stages, where each stage focuses on a different aspect. Stages are chained together, so the output from one stage is input to the other. The stages are lexical analysis, syntactic analysis, and semantic analysis. High level diagram is in figure 4.2

4.1 Lexical analysis

Lexical analysis or tokenization is the first step of language processing. It processes input text into a stream of tokens. Token represents part of input with logical meaning. In example 4.3 there is a C++ code and its resulting tokens. The code declares and initializes a variable to 8 and if the variable is greater than ten, it shows *Condition is true.* If it is not greater than ten, it shows *Condition is false.* In lexical analysis, it does not matter what the code represents. The only important thing is recognizing sequences of tokens. It is usually accomplished with a finite automaton and described with regular expressions. Tokens are then passed to the syntactic analyzer.

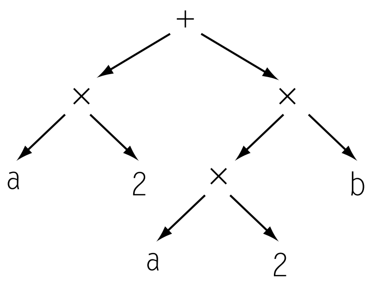


Figure 4.1 AST of arithmetic expression $a \times 2 + a \times 2 \times b$

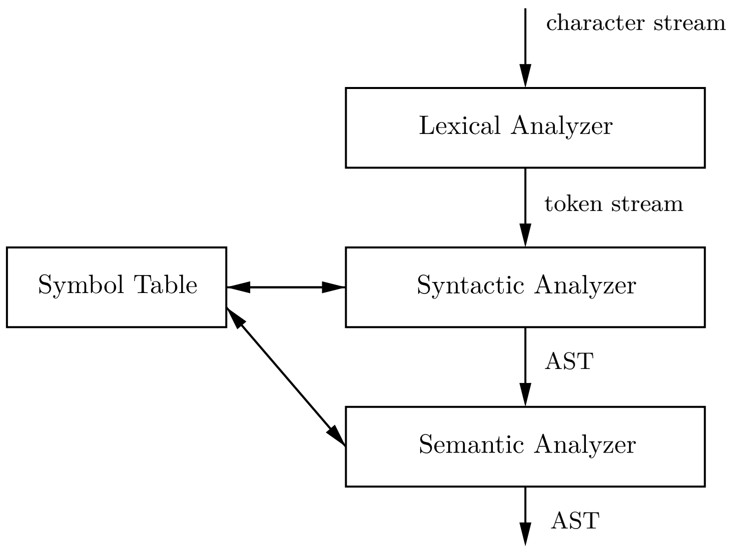


Figure 4.2 Parsing diagram

```
int id_name = 8;
if(id_name > 10)
   print("Condition is true");
else
   print("Condition is false");
```

```
Output tokens:

kw_int
identifier
equals
number
semicolon
kw_if
left_parenthesis
identifier
greater
number
right_parenthesis
identifier
left_parenthesis
string
right_parenthesis
semicolon
kw_else
identifier
left_parenthesis
string
right_parenthesis
semicolon
```

Figure 4.3 C++ tokenization

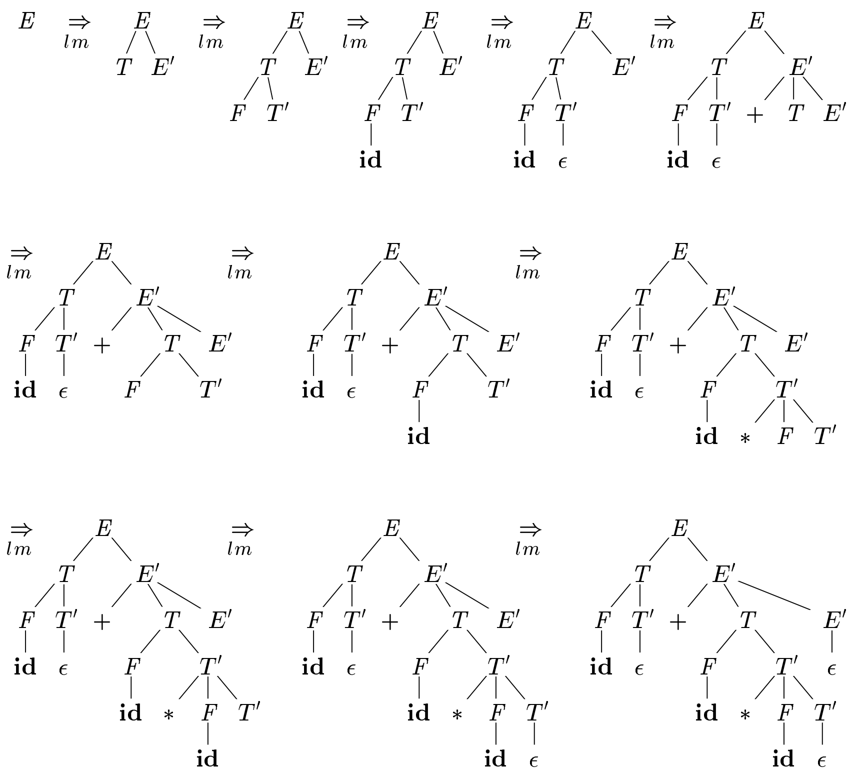


Figure 4.4 Example of building tree from top to bottom [19, p. 218 figure 4.12]

4.2 Syntactic analyzer

The syntactic analyzer, or the parser, uses tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. [19, p. 8] Output of the parser is a parse, parse tree, or AST. Parsers can work in one of two ways: Top-down or bottom-up. Both approaches are implemented with PDA.

4.2.1 Top-down

Top-down analysis (LL analysis) creates a parse tree of a sentence from top to bottom, as seen in figure 4.4, which follows the left parse. *For a given context-free grammar $G = (N, \Sigma, P, S)$ PDA R can be constructed, such that $L(G) = L_\varepsilon(R)$. Resulting PDA R is defined as like this:*

$$\begin{aligned}
&R = (q, \Sigma, N \cup \Sigma, \delta, q, S, \emptyset)\\
&\delta :\\
&1.\delta(q, \varepsilon, A) \leftarrow \{(q, \alpha) : (A \to \alpha) \in P\}, \forall A \in N, \quad \triangleright \textit{(Expansion)}\\
&\quad \delta(q, a, a) \leftarrow \{(q, \varepsilon)\}, \forall a \in \Sigma. \quad \triangleright \textit{(Comparison)}
\end{aligned}$$

[18, p. 4] Because the PDA has only one state, it can be ommited from the PDA's configuration. The configuration (q, w, s) where $w \in \Sigma^*, s \in (N \cup \Sigma)^*$ will become (w, s). And because we are interested in getting a parse, the configuration can be extended to contain the desired parse like so: (w, s, O) where O is a output parse with the most right rule being the most recently used. From this point forward this work will use this type of configuration to describe parser top-down parser.

The LL analysis depending on the PDA uses two operations: comparison and expansion. These two operations are enough to create a PDA that produces a left parse as its output. Comparison is a deletion of the same terminal symbol from input and the top of PDS. It is executed every time a terminal symbol is on the top of PDS. When the terminal symbol on the PDS does not match the first symbol of input, input is not accepted by PDA. Let the ($a\beta$, $a\alpha$, ε) be a configuration of a PDA where $a \in T, \beta \in T^*, \alpha \in (T \cup N)^*$. The comparison will be executed because of the terminal symbol on the top of the PDS. The new configuration will be (β, α, ε).

The second operation, expansion, replaces a non-terminal symbol on top of the PDS with its right-hand side of the rewrite rule. Expansion is executed every time a non-terminal symbol is on the top of a PDS. Let the (a, $A\alpha$, γ) be configuration of a PDA, where $a \in T^*, A \in N\, \alpha \in (T \cup N)^*, \gamma$ is sequence of rule numbers and $n|A \rightarrow \beta \in P$. Based on the top-down analysis, the expansion will be executed, and the configuration will change to (a, $\beta\alpha$). If the non-terminal on the top of the PDS has multiple rewrite rules, the top-down analysis is not deterministic. The same problem was encountered before in section 3.8.

Non-determinism in parsing is not desirable because the parser cannot be easily implemented by an algorithm. That is why there are deterministic versions of top-down parsing called LL(k) and LL(*) analysis.

4.2.2 LL(k) analysis

LL(k) analysis is the deterministic version of the top-down analysis. It is achieved by maintaining a look-ahead to characters from input that follow the already processed part of the input. The k in LL(k) stands for the number of look-ahead characters.

"To achieve deterministic LL(k) parsing, a parse table is constructed. The parse table contains the information allowing the PDA" to determine what rule should be used in the expansion. [9] The parse table for grammar $G = (N, \Sigma, P, S)$ is a mapping $M : (N \times (\Sigma^k \cup \{\varepsilon\})) \rightarrow n$, where the n is the rule number to use. The simplest is LL(1) parsing. It maintains only one look-ahead symbol. Grammar G with following rules:

$$\begin{gathered} S \rightarrow aBa \\ S \rightarrow c \\ S \rightarrow \varepsilon \\ B \rightarrow bb \end{gathered}$$

is ambiguous without LL(1) parsing because it has multiple rules with the same non-terminal symbol S on the left-hand side. However, with LL(1) analysis, it is deterministic. Parse table is in table 4.1. Based on the parse table, a parser can be executed on sentence *abba*. Process of the parser execution is in table 4.2.

In the first step S is on the top of the PDS and a is the look-ahead character. S is non-terminal. This means the parser will perform expansion. There are three rules with S on the left-hand side. Parse table shows that when S is on the top of the PDS, and a is the look-ahead, rule 1 should be used.

Parse table is constructed with the help of two functions, $FIRST(x)$ and $FOLLOW(A)$ where x is any sentential form and A is non-terminal symbol. $FIRST(x)$ *returns set of terminal symbols, including* ε*, that begin strings generated from* x[19, p. 220]. $FOLLOW(A)$ *returns a set of terminals that can appear immediately to the right of A in a sentential form that is generated by the grammar.* [19, p. 221]. $FOLLOW$ of the start symbol always contains ε symbolizing the end of input that follows it. Sets $FIRST$ and $FOLLOW$ calculated for the grammar G are in the table 4.3. The table is built in the following way

- For each rule, add its number to row with its non-terminal under each element of $FIRST$.

	a	b	c	ϵ
A	1		2	3
B		4		

Table 4.1 LL(1) parse table

- If $FIRST$ of the rule contains ε. Add the number of a rule under each terminal symbol from $FOLLOW$.

The table 4.1 has always only one rule number in each cell. That means the grammar is LL(1). But there are grammars that are not LL(1). "*Generally the grammar $G = (N, T, P, S)$ is LL(1) if for each pair of rules $A \to \alpha|\beta$ the following is true:*

- $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
- if $\varepsilon \in FIRST(\alpha)$ *then* $FOLLOW(A) \cap FIRST(\beta) = \emptyset$."

[20, p. 16 translated by me]

Assume a grammar is constructed so that there is a rule $A \to \alpha|\beta$ and $FIRST(\alpha) \cap FIRST(\beta) \neq \emptyset$. The grammar is not LL(1). An example can be grammar M with rules:

$$S \to aBa$$
$$S \to a$$
$$B \to bb$$

Its $FIRST$ and $FOLLOW$ with the parse table are in table 4.4. The $FIRST(aBa)$ and $FIRST(a)$ both contain a. This is called the first-first conflict. The first-first conflict is not the only type of conflict. Another type is first-follow conflict.

Assume a grammar with rule $A \to \alpha|\beta$ and $\varepsilon \in FIRST(\alpha)$ and $FOLLOW(A) \cap FIRST(\beta) \neq \emptyset$. The grammar is again not LL(1), because of the first-follow conflict. An example is grammar F with rules:

$$S \to aBa$$
$$S \to \epsilon$$
$$B \to bb$$
$$B \to Sa$$

its $FIRST$ and $FOLLOW$ together with parse table are in 4.5. The $FIRST(S)$ contains ε and a. $FOLLOW(S)$ also contains a. This is first-follow conflict. Both conflict types have implications.

Another restriction is that a grammar cannot contain left recursion in order to be LL(k) because it automatically means there is a first-first conflict. Some techniques can remove conflicts. However, they are not crucial for this work.

4.2.3 LL(*) analysis

LL(*) is similar to the LL(k), but "*the key idea behind LL(*) parsers is to use regular expressions rather than a fixed constant. The analysis constructs a deterministic finite automaton (DFA) for each non-terminal in the grammar to distinguish between alternative productions.* [21, p. 2] That means the k is computed dynamically based on the result of a finite automaton. The final states of the automaton dictate what rule should be used during the expansion. "*The LL(*) algorithm yields an exact DFA when the 'look-ahead language' is regular (the look-ahead language is regular*

Step	Input	PDS	Left parse
1	a b b a	S	
2	a b b a	a B a	1
3	b b a	B a	1
4	b b a	b b a	1, 4
5	b a	b a	1, 4
6	a	a	1, 4
7	ϵ	ϵ	1, 4

Table 4.2 LL(1) parser run

	Rule($A \to \alpha$)	$FIRST(\alpha)$	$FIRST(A)$	$FOLLOW(A)$
1	$S \to aBa$	a	a c ϵ	ϵ
2	$S \to c$	c		
3	$S \to \epsilon$	ϵ		
4	$B \to bb$	b	b	a

Table 4.3 First and follow

	Rule($A \to \alpha$)	$FIRST(\alpha)$	$FIRST(A)$	$FOLLOW(A)$
1	$S \to aBa$	a	a	ϵ
2	$S \to a$	a		
3	$B \to bb$	b	b	a

Parse table

	a	b	ϵ
S	1, 2		
B		3	

Table 4.4 First-first conflict

	Rule($A \to \alpha$)	$FIRST(\alpha)$	$FIRST(A)$	$FOLLOW(A)$
1	$S \to aBa$	a	a ϵ	ϵ a
2	$S \to \epsilon$	ϵ		
3	$B \to bb$	b	b a	a
4	$B \to Sa$	a		

Parse table

	a	b	ϵ
S	1, 2		2
B	4	3	

Table 4.5 First-follow conflict

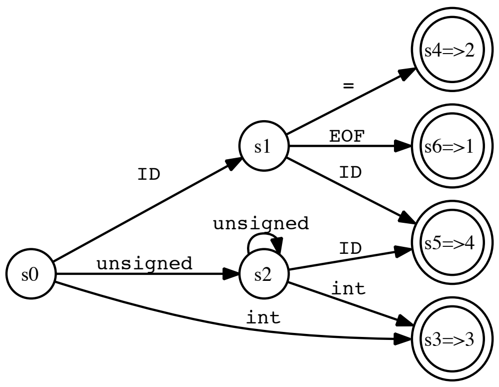


Figure 4.5 LL(*) finite automaton [21, p. 3]

unless the algorithms encounters recursion in the grammar).“[22] *When the DFA construction algorithm encounters recursive rule invocations in the grammar, it approximates the recursion with cycles in the DFA. Nevertheless the recursion still cannot be a left recursion.* [22] An example can be following grammar that is not LL(k) and contains a recursive rule:

$$\begin{aligned}
&S \rightarrow ID\\
&S \rightarrow ID = A\\
&S \rightarrow U\ int\ ID\\
&S \rightarrow U\ ID\ ID\\
&U \rightarrow unsigned\ U\\
&U \rightarrow \varepsilon
\end{aligned}$$

The finite automaton which gets the correct k is in figure 4.5. Based on the input sentence, the finite automaton can find the correct k needed for the sentence. This would not be possible if the k was fixed. Because if the k was fixed substring *unsigned* could repeat $k + 1$ times, and the parsing would result in an error.

4.3 Bottom-up

Bottom-up parsing (LR parsing) is the alternative to Top-Down parsing. It constructs the parse tree from bottom to top. Example of such construction is in figure 4.6. Its output is the reversed right parse of a sentence. Similar to top-down analysis *for a given context-free grammar* $G = (N, \Sigma, P, S)$ *PDA* R *can be constructed, such that* $L(G) = L(R)$*. The following operations determine it :*

$$\begin{aligned}
&R = (\{q, r\}, \Sigma, N \cup \Sigma \cup \{\#\}, \delta, q, \#, \{r\})\\
&\delta :\\
&\quad \delta(q, a, \varepsilon) \leftarrow \{(q, a)\}, \forall a, a \in \Sigma, \quad \triangleright\text{(shift)}\\
&\quad \delta(q, \varepsilon, \alpha) \leftarrow \{(q, A) : (A \rightarrow \alpha) \in P\}, \quad \triangleright\text{(reduce)}\\
&\quad \delta(q, \varepsilon, \#S) \leftarrow \{(r, \varepsilon)\}. \quad \text{(accept)}.
\end{aligned}$$

Oposed to the previous definition of PDA's configuration, the bottom-up PDA has the top of the PDS always on the right. [18] Similar to top-down parsing configuration, the state is always known, because only one state is used during execution and one state is used when the string is accepted. So again, the configuration can be simplified to ommit state. Output can be also

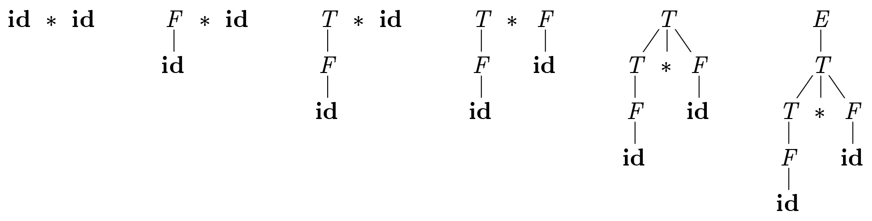


Figure 4.6 Construction of a tree from bottom to top [19, p. 243 figure 4.25]

added to contain the right-parse. From the configuration of format (S, a, α) where $S \in \{q, r\}, a \in \Sigma^*, \alpha \in N \cup \Sigma \cup \{\#\}$ is (a, α, O), where O is the output parse with the most right rule being the most recent. The configuration will be used to better explain the three operations of bottom-up analysis.

The shift is the movement of the symbol from the input to the top of the PDS. Let the $(a\beta, \alpha, \gamma)$ where $\beta \in (\Sigma)^*, a \in \Sigma, \alpha \in (N \cup \Sigma \cup \{\#\})^*, \gamma$ is sequence of rule numbers, be configuration of PDA. After shift operation the configuration will be following: $(\beta, \alpha a, \gamma)$.

The next action, reduction, replaces the rule's right-hand side on the top of PDS with its left-hand side. Let the $(\beta, \alpha aba, \gamma)$ where $\beta \in (T)^*, A \in N \cup \Sigma \cup \{\#\}, \alpha \in (N \cup \Sigma \cup \{\#\})^*, \gamma$ is sequence of rule numbers, be configuration of PDA. And let the grammar contain a rule $A \to aba$. After reduction the configuration will be following: $(\beta, \alpha A, \gamma n)$, where n is number of the rule $A \to aba$.

The last action, accept, is an acceptance of the input string. Parse constructed until acceptance is the output of the parser. Note that the bottom-up parsing, as defined until this point, is non-deterministic, and there are deterministic versions too. Those are, for instance, SLR(k), LALR(k), and LR(k).

4.4 Semantic analysis

Until now, lexical and syntactic analyzers decided whether a sentence is correct grammatically. A semantic analyzer expands on the checking of the setence in a way that it decides whether the sentence makes logical sense in its domain. In the context of programming languages, *the semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table for subsequent use during intermediate-code generation.* [19, p. 8] *Symbol table contains information about the entire program.* [19, p. 4-5] it can contain, for example, all variable names and their types.

4.5 compiler-compiler

Because designing lexical and syntactic analyzers is a repetitive process, and they are based entirely on context-free grammars or regular expressions, and these can be transformed into a finite automaton or PDA, there are tools that can generate lexical and syntactic analyzers automatically from their description. They are called compiler-compilers. Most well known is YACC, generating LALR parser, a variant of bottom-up parsing, Bison, which also utilizes LR parsing, and ANTLR, which uses LL(*) parsing.

This work will further focus only on ANTLR because it is used to implement a proof of concept scanner.

4.5.1 ANTLR

ANTLR is LL(*) parser generator. It generates both lexical and syntactic analyzer. It can be paired with many languages, like Java or C++. The term *host language* will be used from now on for them. Grammar is written in Extended Backus–Naur form (EBNF). It is very similar to grammar used until now. Example of EBNF:

```
A : 'a' B | 'b';
B : 'aba';
```

is equivalent to:

```
A -> aB
A -> b
B -> aba
```

EBNF notation can be translated into the context-free grammar. EBNF rules are denoted by comma and separated by semicolon. Valid right-hand side of an EBNF rule is defined as follows:

1. Terminal symbol (encapsulated inside commas) and non-terminal symbol (without commas) are valid. Empty string is also valid.
2. If `a` and `b` are valid, then: `ab`, `a|b`, `a*`, `a+`, `a?`, `(a)`, are all valid.

Let the function $L(f)$ be defined as all left-hand sides of the set of context-free grammar rules f. Then the resulting context-free rules $C(r)$ of valid EBNF rule r are defined as follows:

- C(`A: 's';`) $= \{A \rightarrow s\}$, where s is a terminal symbol.
- C(`A: s;`) $= \{A \rightarrow s\}$, where s is non-terminal symbol.
- C(`A: ;`) $= \{A \rightarrow \varepsilon\}$
- C(`A: ab;`) $= \{A \rightarrow A'B, A' \rightarrow L(C(a)),\ B \rightarrow L(C(b))\} \cup C(a) \cup C(b)$
- C(`A: a|b`) $= \{A \rightarrow A', A \rightarrow B, A' \rightarrow L(C(a)), B \rightarrow L(C(b))\} \cup C(a), \cup C(b)$
- C(`A: a*`) $= \{A \rightarrow A'A, A \rightarrow \varepsilon, A' \rightarrow L(C(a))\} \cup C(a)$
- C(`A: a+`) $= \{A \rightarrow A'A, A \rightarrow A', A' \rightarrow L(C(a))\} \cup C(a)$
- C(`A: a?`) $= \{A \rightarrow A', A \rightarrow \varepsilon, A' \rightarrow L(C(a))\} \cup C(a)$
- C(`A: (a)`) $= \{A \rightarrow A', A' \rightarrow L(C(a))\} \cup C(a)$

Where `a,b` are valid right-hand sides of an EBNF rule and each set of generated rules $C(r)$ has different non-terminal symbols on the left-hand side, $\bigcap_{r \in R} L(C(r)) = \emptyset$, where R is set of EBNF rules.

The translation can be done even the other way around because the EBNF is just another way to express grammar. It is also used more frequently because it utilizes similar operations as regular expressions: $*, +, ?$. However, unlike regular expressions, the operations can be applied even to non-terminal symbols. The usability and popularity are probably reasons ANTLR chose EBNF as its notation.

ANTLR uses LL(*) analysis. It works as previously described in section 4.2.3, with some additional features. ANTRL does not use only basic grammar. It uses predicated grammar. *A predicated grammar G = (N, T, P, S, Π,M) has elements:*

- *N is the set of non-terminals (rule names)*
- *T is the set of terminals (tokens)*

- *P is the set of productions*
- *$S \in N$ is the start symbol*
- *Π is a set of side-effect-free semantic predicates*
- *M is a set of actions (mutators)*

[21, p. 3].

It also uses additional notation:

- *$A \in N$: Non-terminal*
- *$\alpha \in (N \cup T)^*$: Sentential form*
- *$\pi \in \Pi$: Predicate in host language*
- *$\mu \in M$: Action in host language*

Production rules:

- *$A \to \alpha_i$: i^{th} context-free production of A*
- *$A \to (A'_i) \Rightarrow \alpha_i$: i^{th} production predicated on syntax A'_i*
- *$A \to \{\pi_i\}?\alpha_i$: i^{th} production predicated on semantics*
- *$A \to \{\mu_i\}$: i^{th} production with mutator*

[21, p. 3] Predicated grammar introduced new production rules. Every new rule can also be added to EBNF notation.

- $A \to (A'_i) \Rightarrow \alpha_i$ is rewritten into $A : (A'_i) => \alpha_i$.
- $A \to \{\pi_i\}?\alpha_i$ is rewritten into $A : \{\pi_i\}?\alpha_i$
- $A \to \{\mu_i\}$ is rewritten into $A : \{\mu_i\}$

[23]

The first production rule type is a classic production rule as previously defined in a grammar. Rule $A \to (A'_i) \Rightarrow \alpha_i$ Introduces syntactic predicate. *Syntactic predicates allow arbitrary lookahead. They are given as a grammar fragment that must match the following input in order to parse α_i.* [21, p. 2] This means, that the rewrite rule α_i will be used only if the parser will match A'_i. *Following grammar will differentiate between multiple-assignment statements and simple lists such as:*

```
(a,b) = (3,4);
(apple, orange);
```

The grammar could look like this

```
stat: list '=' list ';'
    | list ';'
    ;
```

where list is a rule, defined elsewhere, that recognizes an arbitrarily long list of expressions. The grammar is not LL(k) for any finite k, unfortunately, due to the common left-factor. [24, p. 12] When the parser sees input such as *(apple, orange, ...* it does not know which rule to use. “*Left-factoring would resolve this problem, but would result in a less readable grammar.*“[24, p. 12] This can be dealt with by using syntactic predicate

```
stat: (list '=') => list '=' list ';'
    | list ';'
    ;
```

The predicate specifies that the first production is only valid if "list "= .... is consistent with (matches) an arbitrarily large portion of the infinite look-ahead buffer. [24, p. 12]

Another new rule type is $A \rightarrow \{\pi_i\}?\alpha_i$. This is the semantic predicate. *Semantic predicates allow the state constructed up to the point of a predicate to direct the parse. Semantic predicates are given as arbitrary Boolean-valued code in the host language of the parser. Actions are written in the host language of the parser and have access to the current state.* [21, p. 2] Following grammar shows how the semantic predicate function.

```
A : {isB(getCurrentToken())}? 'b'
  | {isA(getCurrentToken())}? 'a'
  ;
```

Although the example is pretty straightforward, it can illustrate how the semantic predicate works. Parser will execute code `isB(getCurrentToken())`. If the code returns true, it will use rewrite 'b'. But if the `isB(getCurrentToken())` returns false, the next rule is checked. It again starts with semantic predicate. If `isA(getCurrentToken())` returns true, the 'a' is used. Nevertheless, instead of one simple function call, user can define more functions and execute complex logic, as they would in a host language.

Last rule type is $A \rightarrow \{\mu_i\}$. This is similar to previous rule type, because it also utilizes host language. Mutator μ_i is some arbitrary function written in host language. Example of mutator is in following grammar.

```
A : a {println("matched a");}
  | b {println("matched b");}
  ;
```

The grammar writes to console *matched a* if the first rule is used and *matched b* if the second rule is used.

Both semantic predicates and mutators have access to the parser state. That means they can access previously parsed input, or it can access look-ahead.

The output of an ANTLR parser is AST. By default, each non-terminal creates an AST node. The node has its right-hand side as its children. However, ANTLR3, which is the version this work uses, offers the additional possibility to define AST directly in grammar. ANTRL3 extends its grammar by several AST-making symbols it is the right arrow (->), caret symbol (^), and exclamation mark (!). -> says, *following will be the output*, ^ stands for *create AST node*, and ! means *do not include this in AST*. Example of custom node creation:

```
A : 'a'
  -> ^(A_NODE ^(CHILD_NODE 'a'))
  ;
```

The grammar above will create an AST node called A_NODE with one child node called CHILD_NODE. The CHILD_NODE contains one child 'a'.

The same way as semantic predicates can be used in rules, they can also be used here.

```
A : 'a'
  -> {1+1==3}? ^(A_NODE 'child' 'a')
  -> {1.equals(1)}? ^(A_NODE_WITHOUT_CHILDREN)
  ;
```

This grammar will produce an AST node called A_NODE_WITHOUT_CHILDREN, and it will not have children. Even other non-terminals can be used.

The following grammar will show how to use AST node generated in other non-terminals.

Furthermore, how to distinguish between two same non-terminals. Let us have grammar.

```
1 A : 'a' B C
2   ;
3 B : 'b'
4   ;
5 C : 'c'
```

Add to it manual AST creation

```
1 A : 'a' B C
2   -> ^(A 'a' B C)
3   ;
4 B : 'b'
5   -> ^(B 'b')
6   ;
7 C : 'c'
```

Until now, everything was clear. But if the same terminal or non-terminal is used multiple times. There is a problem.

```
1 A : 'a' B B C
2   -> ^(A 'a' B B C)
3   ;
4 B : 'b'
5   -> ^(B 'b')
6   ;
7 C : 'c'
```

Now the parser cannot tell which B it should use in AST creation on line 2. Labels deal with this problem. Labels are set with equal sign (=) and they are accessed with dolar sign ($).

```
1 A : 'a' first_b=B second_b=B C
2   -> ^(A 'a' $first_b $second_b C)
3   ;
4 B : 'b'
5   -> ^(B 'b')
6   ;
7 C : 'c'
```

Chapter 5
Manta

Manta is a software project providing data lineage of complex data environments. At this point, it supports over 40 technologies across Modeling, reporting, ETL, data analysis tools, databases, and programming languages. Manta scans every part of the data environment and creates connected end-to-end lineage across all environments.

Knowing the data lineage has many advantages. The organization that knows how data is moving across their environment can track why certain decisions were made, or they can better migrate to a different platform. Everybody can simply look at data lineage and see what influenced the current data state. An example of Manta's data flow visualization is in figure 5.1.

5.1 Ways of generating lineage

There are many ways how to generate data lineage. Every way has its positives and drawbacks. Manta uses decoded lineage, which will be described later in this chapter. Its biggest problem is that it needs to know every environment very well. That is part of the reason Manta has to support so many technologies and cannot get any result from environments it does not know.

5.1.1 Manual

One way to create lineage is to do it manually. *This approach usually starts by mapping and documenting the knowledge in people's heads. Talking to application owners, data stewards, data*

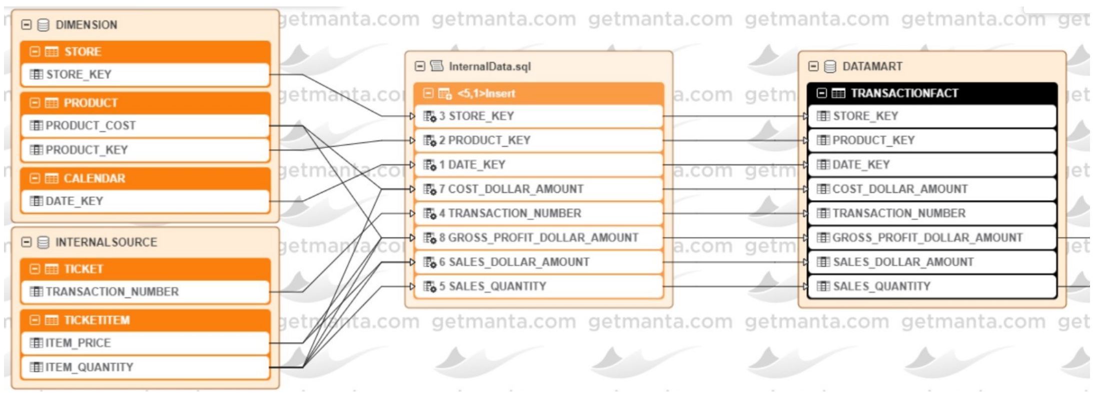


Figure 5.1 Manta data lineage [25]

integration specialist gives adequate but often contradictory information about data movements in the organization. [26] It is not efficient but it is simple. It can be enough for smaller companies or a project, but the more company grows, the more data it produces, and the need for a better lineage solution arises.

5.1.2 Data taging

Next, a more automated approach is data tagging. *The whole idea is that each piece of data that is being moved or transformed is tagged/labeled by a transformation engine which then tracks the label all its way from start to the end.* [26] Issue of this method is security. The transformation engine needs to have access to data that is being moved. Another problem is that transformation changes are not visible without being executed first, and changes outside the transformation engine are not visible at all.

5.1.3 Data transformation tool

Previous methods were done outside the data transformation tool. However, why not have everything in one package. Some solutions combine data transformations with data lineage. ADF is such an example. Nevertheless, its lineage is minimal. This approach deals with security problems. Because it already works with the data, it is unnecessary to protect it from itself. The biggest problem here is that it does not see anything outside the tool.

5.1.4 Decoded lineage

Decoded lineage can solve the visibility and security problems. It does not look at data directly. It decodes metadata. The scanner extracts information and analyzes how the data is flowing. In order to analyze the whole data environment, the metadata scanner has to understand every tool that the user is using. This way of analysis is the way Manta uses.

5.2 Data flow representation

Manta represents data flows as an oriented graph. Each appearance of data in some data processing tool or a language is represented as a node. This naturally creates duplication of one piece of data. That is correct, and it is the base for data lineage. The duplicates are tracked across the graph and connected by an oriented edge. This edge is the data flow. For example, an ETL tool that loads information about bank accounts. Filters account, so only accounts with' Smith' surname are left. Then it filters once again, so only the account number is left, and saves it to some database. This would result in a graph in 5.2. These data nodes are then structured into tables, schemas, transformations, and more tool-specific constructions to create logical groupings for the data flow graph to make sense. Some of these groups can be seen in figure 5.1. These groups are also represented as nodes. The relation *node belongs to group* is also an oriented graph. The node representing a group is connected to the node which belongs to the group. The edge leads from the parent group to the child node. This means manta's data flow graph is two merged graphs. One graph dictates data groups, and the second graph represents data flows. Typical node grouping is *Transformation tool* $\rightarrow$ *conrete transformation* $\rightarrow$ *table* $\rightarrow$ *column*. The column is the individual piece of data. That is why the term *column* will be used to reference individual information from this point forward.

Logical groups can be filtered and provide different levels of detail. One level of detail represents each column on the lowest level, and another level represents only tools working together, without the information about specific columns. The high-level lineage of example 5.2 is in 5.3.

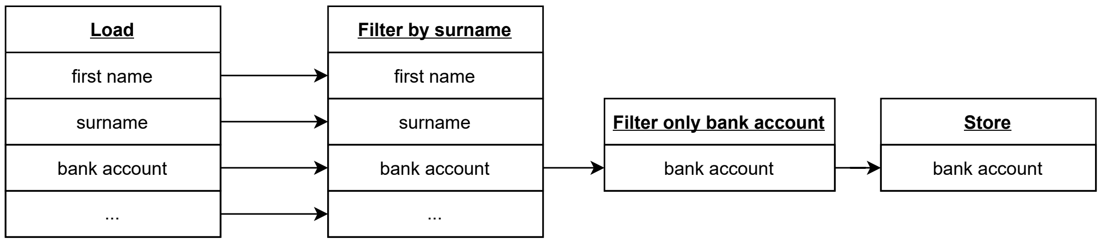


Figure 5.2 Example of ETL data flow

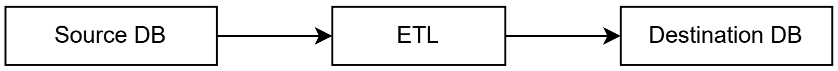


Figure 5.3 Example of high-level lineage

Each node can have attributes that store additional information about the data or the transformation. A transformation that sorts the data would not impact the lineage because the data is still the same. It is only delivered in a different order. This is nice example where node *Sort transformation* would have *sort by* attribute. The attribute would store based on what the transformation is sorting data.

5.2.1 Indirect data flows

Besides ordinary data flows, Manta also has the notion of *indirect data flows*. Indirect data flow is not data flow per se. However, it indicates that some data is affecting others. Indirect data can be seen in a filter. In the lineage from figure 5.2 is *Filter by surname* transformation which filters data, so only data with some specific surname is left. The surname is affecting all data from this point on. This is the indirect lineage. Edited figure 5.2 with added indirect flows is in figure 5.4.

5.3 Architecture

The architecture of manta is split into multiple components. This work is concerting only to Manta Flow CLI in figure 5.5. This part aims to extract metadata of tools and resources, analyze them and create data lineage in a graph form. It is formed from two parts. It is an extractor and analyzer.

This work will skip the extractor part because ADF resources will be extracted manually from git. However, there are ways how to extract resources automatically via REST API. Section 6.4 will focus more on this topic. Nevertheless, the analyzer is responsible for all the analyses, so it will not hinder any usefulness of the work. The chapter 7 will show the design of the analyzer.

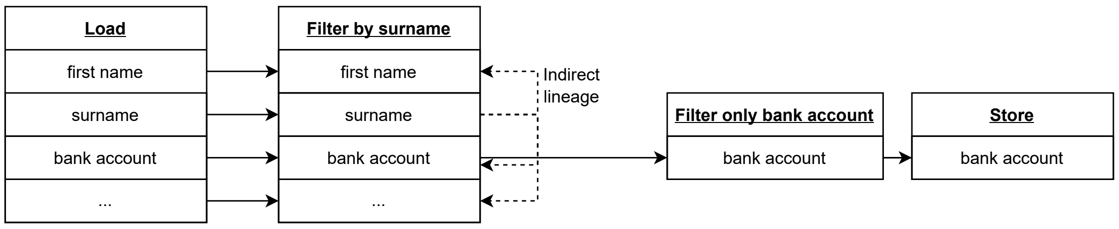


Figure 5.4 Example of indirect data flow

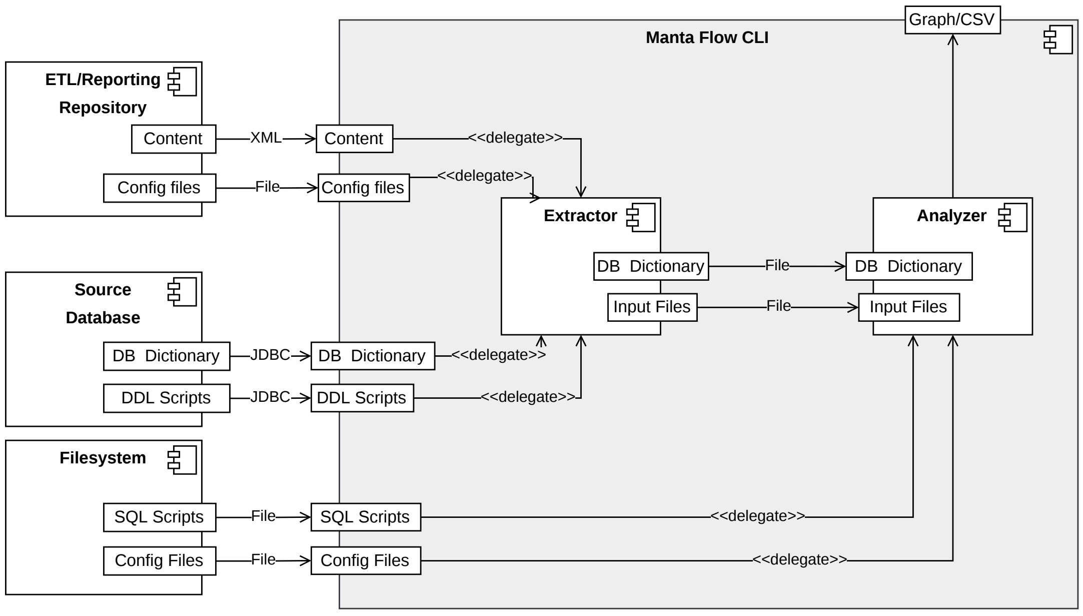


Figure 5.5 Architecture of Manta Flow CLI

The covered part of the architecture looks more like in figure 5.6. The input files are extracted manually from git and used directly from an input folder on the filesystem with installed manta. Finally, DB Dictionary is entirely removed because it is crucial mainly for database analyzers.

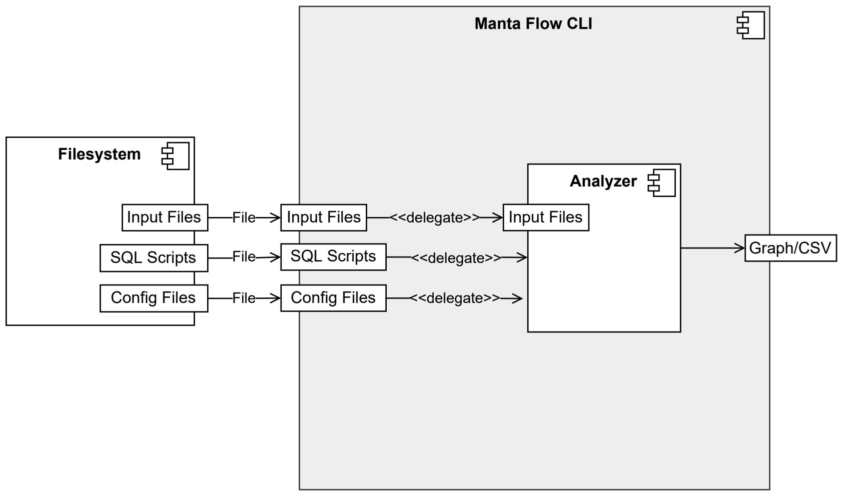


Figure 5.6 Covered part of Manta Flow CLI

Chapter 6
ADF

ADF is cloud ETL tool from Microsoft. *It offers scale-out serverless data integration and data transformation.* [3] It has two versions, ADF version 1 and current version. This thesis works only with the current version, so version 1 will be disregarded.

6.1 Resource

ADF works with a couple of different resources. Each resource serves some purpose. It is stand-alone component and can be edited independently of other resources. However resources can use and reference other resources, which is in fact very common. Not every resource can reference every resource. Resources create hieharachy which can be seen in figure 6.1. Each edge which has *inline dataset* note, means that dataset is created directly in the resource it uses it. Otherwise it is only referenced.

Each resource can have parameters. Parameter can be used inside the resource instead of constant value. When resource is referenced from another resource, the referencing resource has to specify values for each parameter. Exception for this rule is data flow resource (DFR)[1]. When

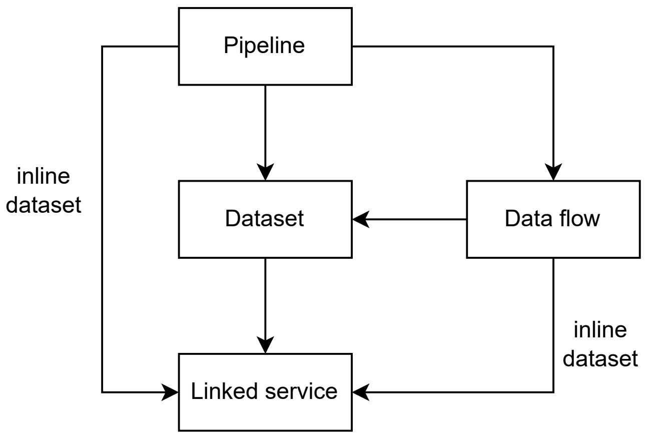


Figure 6.1 ADF resource hierarchy

[1]The resource is called *data flow.* Because it could be missinterpreted as data flow as in data lineage data flow, the thesis will use DFR until now on.

DFR references dataset or linked service it cannot specify values for their parameters. These values are specified when DFR is referenced by a pipeline.

6.1.1 Pipeline

A pipeline is a logical grouping of activities that together perform a task[27]. It serves as an orchestration unit. The task could be to load information about customers, backup it into file, analyze their spending and save the results into database. Step of the task is called activity and it is the basic block of a pipeline. Each pipeline contains multiple activities, which together complete desired task.

An activity takes zero or more input datasets and produce one or more output datasets. [27] Dataset is resource representing some data. Example of dataset could be table or delimited file. More about datasets will be in section 6.1.2. Input of copy activity is dataset which is coppied into output. Output is temporary dataset which can be used from other activites by specifing `@activity"(<name of activity">).output`.

Activites are divided into three categories: Data movement, data transformation and data control. Data movement category contains only *copy activity*. It serves to copy data from a source to a sink (destination). Source and sink are specified as datasets or inline linked services. Both will be explained in next section 6.1.3. Data transformation activities are *Mapping data flow*, which references DFR and executes its logic, and *Wrangling data flow* which utilizes *power query* to prepare data for future use. *Power Query is the data connectivity and data preparation technology that enables to import and reshape data from a wide range of Microsoft products.* [28] The last group, data control, contains activites such as *foreach, execute pipeline, filter, set variable* and more.

6.1.2 Dataset

Dataset, as already mentioned in previous section, is a resource representing some data. To be more specific *it is a named view of data that points or references the data somebody wants to use in activities as inputs and outputs.* [29] They exist because pipelines are general orchestration units. They don't care where the data is coming from. They just need to know what format it uses and what operations they can do with it. On top of that *datasets identify data within different data stores, such as tables, files, folders, and documents.* [29] Each type of data store data will have separate dataset. Example of datasets are: Excel, delimited text, Azure storage, Azure postgresql and many more.

Each dataset needs to have reference to linked service. Linked service specifies concrete implementation of the dataset and it will be described in next section 6.1.3. But it does not need to have defined schema. The schema can be deduced during run time. The operation of deducing is called column drift and columns are called drifted columns. This makes sense, if data transformations are not on column level. However even if they are on column level there are specific aproaches how to retrieve the drifted column data. In DFR you can get column data by expression `byPosition` or `byName` where you specify position of column or its supposed name.

6.1.3 Linked service

Linked service is the simplest form of resource. Because pipelines and datasets are still general concepts and they do not specify, where exactly is the data stored, there needs to be concept that stores data about concrete data source. Realization of this concept is linked service.

Linked service can be of many types. It mirrors dataset types. There are database linked services, that define hostname, port and dabase name. There are many file storage types like Azure blob storage, google drive or azure datalake, that specifies path to the folder or file.

Linked services are usually used together with datasets. Typical use is that DFR or pipeline use load component, which loads some dataset. The dataset has underlying linked service and based on the linked service real data is retrieved from the data source. However linked service can be used without dataset. When DFR or pipeline would reference dataset it can reference linked service instead and provide dataset information together with the reference. It is called *inline linked service.*

6.1.4 Data flow resource (DFR)

There are two types of DFR. Mapping DFR and wrangling DFR. Wrangling DFR is outside the scope of this work. Every time DFR is mentioned it is used as alias for Mapping DFR.

It is the main tool for data transformation. Pipelines also have some data transformation capability, but the DFR is the primary way of doing transformations on column level. Pipelines are not suited to do that.

Similar to pipelines DFR is split into smaller chunks. They are called transformations. Each transformation has specific purpose. Based on the purpose of transformation they can be divided into groups: Schema modifier, row modifier, formatters and multiple inputs/outputs. Two transformations which are not included in any of the groups are Source transformation and Sink transformation. These two transformations are special and at least one of each need to be in each DFR.

Source transformation is the entry point of data in DFR. There can be many source transformations, but at least one needs to be in DFR. It uses datasets or inline linked service to define concrete data source. The transformation copies the schema of dataset the first time DFR is executed. When dataset does not contain schema, the schema can be generated (drifted). Drifting can be disabled and columns won't be used. Sink transformation is very similar to source transformation. It utilizes datasets or inline linked service to specify data destination. There needs to be at least one, but there can be many.

6.2 Data representation

Manta scanners create decoded lineage. In order to create decoded lineage the scanner needs to analyze metadata of analyzed tool. It is often saved in JSON or XML format. ADF scanner is not different and it will follow the same pattern. ADF itself uses JSON format for its resources. However in DFR's JSON file, there is a field called *scriptlines* which contains definition of transformations. More on *Data flow script* in the next section 6.2.1. Example of dataset file:

```
"name": "<name of dataset>",
"properties": {
    "type": "<type of dataset: DelimitedText, AzureSqlTable etc...>",
    "linkedServiceName": {
            "referenceName": "<name of linked service>",
            "type": "LinkedServiceReference",
    },
    "schema":[

    ],
    "typeProperties": {
        "<type specific property>": "<value>",
        "<type specific property 2>": "<value 2>",
    }
}
```

Example of DFR file:

```

"name": "filter",
"properties": {
    "type": "MappingDataFlow",
    "typeProperties": {
        "sources": [
            {
                "dataset": {
                    "referenceName": "AzurePostgreSqlTable1",
                    "type": "DatasetReference"
                },
                "name": "source1"
            }
        ],
        "sinks": [
            {
                "dataset": {
                    "referenceName": "DelimitedText1",
                    "type": "DatasetReference"
                },
                "name": "sink1"
            }
        ],
        "transformations": [
            {
                "name": "filter1"
            }
        ],
        "scriptLines": "source(output(\n\t\ttrue as string\n\t),\n\tallowSchemaDrift:
            true,\n\tvalidateSchema: false,\n\tisolationLevel: 'READ_UNCOMMITTED',\n\
            tquery: 'select col_name as \"true\" from table_name;',\n\tformat: 'query')
            ~> source1\nsource1 filter(true == '\"Column name is 42\"') ~> filter1\
            nfilter1 sink(allowSchemaDrift: true,\n\tvalidateSchema: false,\n\
            tskipDuplicateMapInputs: true,\n\tskipDuplicateMapOutputs: true) ~> sink1"
    }
}
```

There are similarities accross all resources, mainly names and how resources are referencing other resources. Otherwise each resource has distinct attributes. Some attributes are important some are not. Some are basic strings and some are complex data types, like objects.

Most attribute values are written as expressions. This means every value has to be processed before it gives its actual value. Exception to this are names, which are constant. More on expressions in section 6.3.

6.2.1 Data flow script

Data flow script is a language which is used to represent DFR transformations. It is located in DFR JSON definition file under *scriptLines* attribute as seen in the DFR file example in section 6.2. The language uses its custom syntax with data flow expressions. More on expressions will be in next section 6.3. Transformations in data flow script have always, except for one exception, the same format.

```
    <input stream> <transformation name>(attributes) ~> <output stream>
```

Every transformation has *output stream*. And every transformation except source transformation has *input stream*. The output stream is usually name of the transformation. If transformation has multiple outputs they are distinguished by *at* symbol (@) like so:

```
    <input stream> <transformation name>(attributes) ~> <output stream>@(out1,out2)
```

Outputs are separated by comma. Input stream is always an output stream of some other transformation. That is the reason source does not have an input stream, because it is the only transformation without an input. Example of simple script with source and one transformation:

```
    source(attributes) ~> source1
    source1 transformation1(attributes) ~> transformation_name
```

Each transformation has different attributes. Source transformation needs to have information about query, or specific file. Filter transformation needs information about the filter condition. Nevertheles some attributes are shared. Shared attributes are most often describing optimization to perform better in a certain context. Attributes tend to have following syntax.

```
    <attribute name> : <attribute value>
```

Attributes are separated by comma. The *<attribute value>* is almost always data flow expression. Sometimes there are exceptions, but those have the same syntax as expressions so it is not a problem for the parser. The other type of attribute has following syntax:

```
    <attribute name>(specific attribute information)
```

This syntax is used mainly in transformation specific attributes, that need to be specified further. The attribute specification is inserted inside the parenthesis. It has varied syntax based on the attribute. Some of these attributes are however shared between certain transformations. Example of such attribute is `each`. Which is used by transformations that map input columns to output columns. The `each` attribute tells that *each* column that matches certain condition will be mapped and in what way.

```
MoviesYear derive(
    Rating = toInteger(Rating),
    each(
        match(startsWith(name,'movies')),
        'movie' = 'movie_' + toString($$)
    )
) ~> CleanData
```

The above example defines a derived column named CleanData that takes an incoming stream MoviesYear and creates two derived columns. The first derived column replaces column Rating with Rating's value as an integer type. The second derived column is a pattern that matches each column whose name starts with 'movies'. For each matched column, it creates a column movie that is equal to the value of the matched column prefixed with 'movie_'. [30]

6.3 Expression language

Data flow expressions are functional expression language used to pass dynamic values to resources. There are two types of expressions languages: Pipeline expressions and Data flow expressions. The name is already telling that both can be used only on specific places.

6.3.1 Pipeline expressions

pipeline expressions is expression language used in pipelines, datasets and linked services. *Expressions can appear anywhere in a JSON string value and always result in another JSON value. If a JSON value is an expression, the body of the expression is extracted by removing the at-sign*

(@). If a literal string is needed that starts with @, it must be escaped by using @@. [31] An example of pipeline expression:

```
{
  "type": "@if(equals(1, 2), 'Blob', 'Table' )",
  "name": "@toUpper('myData')"
}
```

The first expression, which is used for type, compares two numbers one and two, and based on the result of the compare it will return either *Blob* or *Table*. The second expression that calculates name converts string *myData* to upper case. As already mentioned, the expression is denoted by at symbol (@). The same is true for datasets and linked services. Expressions can also appear inside strings, using a feature called string interpolation where expressions are wrapped in *@{ ... }*. For example:

```
"name" : "'first-@{toUpper('John')}, last-@{concat('Smi','th')}'"
```

Pipeline expressions offer a variety of different functions divided into six categories. The categories are: Date functions, String functions, Collection functions, Logical functions, Conversion functions and Math functions. All categories could be found in some form in all programming languages, however there are more specific constructs which can be used in pipeline expressions.

Pipelines can use *global parameters*. *Global parameters are constants across a data factory that can be consumed by a pipeline in any expression*[32]. They are accessed by

```
pipeline().globalParameters.<parameterName>
```

However there are not only global parameters. Pipelines, datasets and linked services can access their own parameters by.

```
@pipeline().parameters.<parameterName>
@dataset().<parameterName>
@linkedService().<parameterName>
```

In this piece of code one could see that datasets and linked services do not use the `.parameters.` inside its syntax unlike pipelines. That is because pipelines have more special variables than just parameters. Datasets and linked services don't.

The variables, are called system variables. They contain, for instance, name of data factory and name of the pipeline.

```
@pipeline().DataFactory
@pipeline().Pipeline
@pipeline().TriggerTime
```

They return ordinary string as a result. *Trigger-related date/time system variables (in both pipeline and trigger scopes) return UTC dates in ISO 8601 format, for example,* 2017-06-01T22:20:00.4061448Z [33]. However pipeline expressions can use more types than just strings.

All possible types are: String, Integer, Float, Boolean, Array, Dictionary, Object, XML, JSON native type. Return types of functions have to be ultimately compatible with JSON, because it will be put into a JSON file. However, some types are not visibly compatible with JSON. Those types are most of the time not used alone. They are used together with other functions that extract data from them and the final value is used inside JSON as string. Neverheles if someone uses for example XML, it will be converted to string.

6.3.2 Data flow expressions

Data flow expressions are alternative to pipeline expressions. They are used mainly in DFR but they can be used in pipelines when passing parameters to DFR activity. In contrast to pipeline expressions, data flow expressions are not used in JSON, they are used in data flow script as a way to work with columns and to dynamically decide attributes of transformation. Expression functions are separated into multiple categories. Some expressions are allowed to be used only in certain DFR transformations. The category list is following: aggregate functions, array functions, cached lookup functions, conversion functions, date and time functions, expression functions, map functions, metafunctions, window functions. From this list agregate functions can be used only in transformations: aggregate, pivot, unpivot, and window transformations. This restrictions seems a little unnecesary, because it means that ,for instance, filter transformation cannot use function which calculates average of a column. This could be a problem for expression evaluation later in implementation of the scanner. However because function names are not overloaded, this won't cause a problem and implementation could evaluate the function no matter where it is used.

Functions are however overloaded based on type and number of arguments. A great example of this is function `add`. It *adds a pair of strings or numbers. Adds a date to a number of days. Adds a duration to a timestamp. Appends one array of similar type to another. Same as the + operator.*

```
add(10, 20) -> 30

10 + 20 -> 30

add('ice', 'cream') -> 'icecream'

'ice' + 'cream' + ' cone' -> 'icecream cone'

add(toDate('2012-12-12'), 3) -> toDate('2012-12-15')

toDate('2012-12-12') + 3 -> toDate('2012-12-15')

[10, 20] + [30, 40] -> [10, 20, 30, 40]

toTimestamp('2019-02-03 05:19:28.871', 'yyyy-MM-dd HH:mm:ss.SSS') + (days(1) + hours(2)
     - seconds(10)) -> toTimestamp('2019-02-04 07:19:18.871', 'yyyy-MM-dd HH:mm:ss.SSS'
    )
```

[34] From the example above it is clear it accepts many different types and also returns many types. In comparison to pipeline, the data flow expressions have more complex types. Nevertheles data flow expression do not contain only functions.

Expressions need a way to reference columns the transformation is working with. Column names are not restricted in any way. *Simple, simple name with spaces, very = complex{{{ + _ name with special symbols* are all valid column names. If a column name has any special symbol including space it has to be encapsulated inside braces {...}. This is the reason only symbol that is not allowed is closing brace }, because it is messing with the internal representation of DFR and the whole DFR wil be deleted if someone puts the } symbol in the name of a column. Here is example of why this happens:

```
source1 derive(name = 'value') ~> derivedColumn1
```

Code above defines derive transformation which can create new columns. The part `name = 'value'` has name of the column on the left and its value on the right. When the name changes to *na}me* the definition changes to:

```
source1 derive({na}me} = 'value') ~> derivedColumn1
```

Everything looks correct from the ADF web application. Validation of DFR is also not signaling any errors. But when the DFR is saved and loaded again, the data flow script is loaded with *{na}me}*. The internal parser recognizes *na* to be the name and it should be followed by equals. But because the *me}* is next, the DFR does not know how to create all transformations and the file loading fails. It either deletes one transformation, whole DFR or the DFR is considered corrupted and cannot be opened.

Columns in expressions are referenced by their name. If the column name is complex, the expression has to contain enclosing braces too. Following expressions are all valid.

```
column == 'expected name'
toUpper({column name})
concat({very = complex{{{ + _ name with special symbols}, '.txt')
```

This is not the only problem with unrestricted column names, because there are expressions without parenthesis that are not columns. They are called literals. To be specific two literals in dataflow expressions could cause problems. They are **`true`** and **`false`**. Meaning of these literals changes based on a context. When there are no columns with names *true* or *false*, it is considered boolean literal. When there are columns named *true* or *false*, it is considered column name. In the second case boolean values have to be expressed as **`true`**`()` or **`false`**`()`.

6.3.3 Similarities in languages

Both pipeline and data flow languages are very similar. They even contain functions with same name. Example of such function is addDays. The function does the same thing in each language, but it accepts different data types and number of parameters. Pipeline expression:

```
addDays(<timestamp> : String, <days> : Integer, <format>? : String) => String
```

Data flow expression:

```
addDays(<date/timestamp> : datetime, <days to add> : integral) => datetime
```

An interesting thing to note is different naming conventions for types. Pipeline expresions use big first letter, meanwhile data flow expressions use small.

Even though pipeline and data flow expressions use same functions, which could cause problems in parsing later on, they are not conflicting in any syntactic structures, so it will not be a problem to parse them.

6.4 Extraction

Source codes of data transformations can be retrieved in two ways. First is manual, and the second one is via REST API. Although REST API is the automatic way, focus of this work is on the analysis of data from git repository, which is the manual way of extraction.

Manual extraction is done by downloading whole git repository. It contains all important files the scanner needs to properly analyze a data factory. The strucutre of repository is following:

```
Repository root
 ├─dataflow
 ├─dataset
 ├─factory
 ├─linkedService
 └─templates
```

In each folder is corresponding resource (dataflow folder - DFR, pipeline - pipeline resource, ...). Factory folder contains one JSON file, which holds all the information about the data factory environment. Templates folder stores templates, which are used to aid DFR creation. The JSON files in each folder are almost the same as if the files would be extracted via REST API.

Chapter 7

Scanner design

Scanners at Manta are usually divided into two parts: connector and data flow generator. The connector processes input files and creates the model. Model classes are then passed to the data flow generator, which creates a data flow graph. Both are further divided into more packages.

7.1 Connector

The connector consists of three parts: Reader, Model, Resolver that are shown in figure 7.1. The reader reads the input and creates the model. The package called model represents the interfaces of the model. Their implementation is located inside the resolver package.

7.1.1 Reader

Reader's purpose is to read ADF's JSON files from the downloaded git repository. It traverses its folder structure and creates model classes for pipelines and DFRs. After the DFR model is created, it contains references to datasets and linked services. The reader then loads corresponding datasets and linked services from the input. The output of the reader is class `ADF` that is passed to the data flow generator to create a data flow graph.

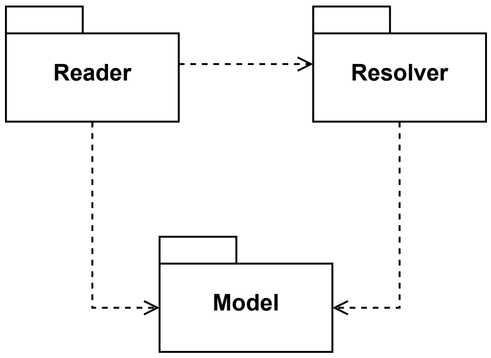
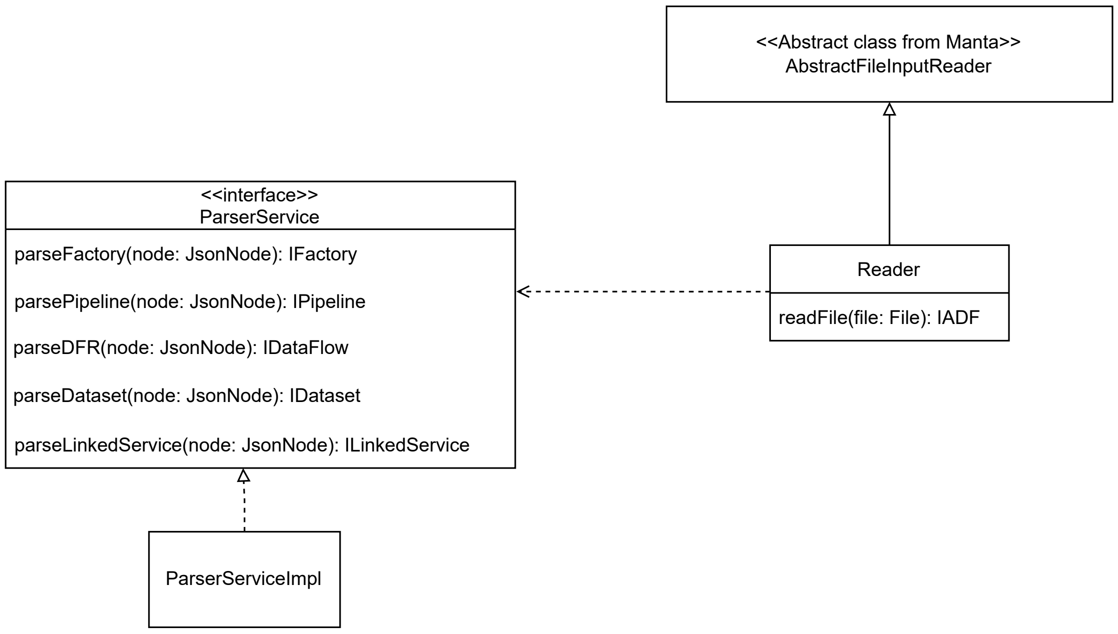


Figure 7.1 Connector packages



<<Abstract class from Manta>>
AbstractFileInputReader
<<interface>>
ParserService
parseFactory(node: JsonNode): IFactory
parsePipeline(node: JsonNode): IPipeline
parseDFR(node: JsonNode): IDataFlow
parseDataset(node: JsonNode): IDataset
parseLinkedService(node: JsonNode): ILinkedService
Reader
readFile(file: File): IADF
ParserServiceImpl

Figure 7.2 Class diagram of a reader

To process each resource, the reader uses parser service, which is part of resolver package. Parser service has methods to process each type of resource. This is shown in figure 7.2. After the service creates model for resource it is added to the output class. Sequence diagram for reader is in figure 7.3

7.1.2 Parser service

Parser service is integral for the reader. It is a facade that the reader can easily use. The parser service delegates JSON processing to specialized classes because each ADF's resource has a slightly different structure. Even the same resource has a varied format based on the specific resource type. They parse JSON and create corresponding classes. The class structure of parser service is in figure 7.4. The function of all parsers is pretty similar. They accept a JSON node on the input and they output model created from the JSON. A pipeline expression parser parses each field that could contain a pipeline expression. A slight exception is the DFR parser because its JSON contains a data flow script. Hierarchy of datasets and linked service parsers was designed by Kyrylo Bulat, who was working on the scanner before this bachelor thesis began.

Data flow script, pipeline expressions, and data flow expressions are parsed by a parser generated by ANTLR. Because data flow expressions are an integral part of the script and pipeline expressions use very similar syntactic constructs, they are all in one grammar.

7.1.3 Model

The whole ADF input is represented by one interface `IADF`. It provides parsed pipeline or DFR with datasets, linked services and optional information about the factory. Hierarchy can be seen in figure 7.5. In the hierarchy is interface `IADF` with relation to all interfaces that represent a resource. Similar to the design of datasets and linked service parsers, model for datasets and linked services was designed by Kyrylo Bulat.

Because input files are not always complete and can contain errors, there is a possibility that some resources will be missing. This is why all resources are not required. The only necessary

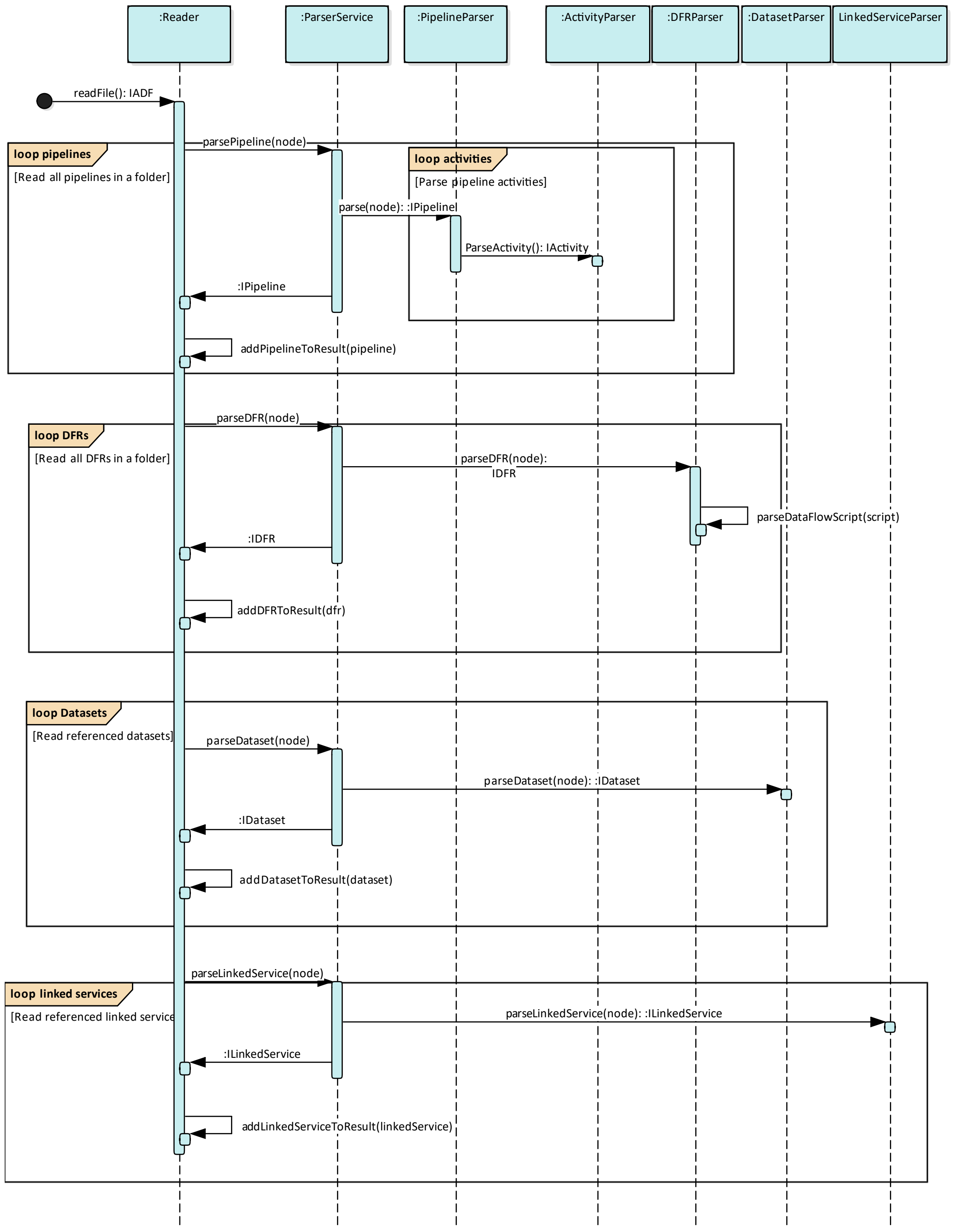

:Reader
:ParserService
:PipelineParser
:ActivityParser
:DFRParser
:DatasetParser
LinkedServiceParser
readFile(): IADF
loop pipelines
[Read all pipelines in a folder]
parsePipeline(node)
loop activities
[Parse pipeline activities]
parse(node): :IPipeline
ParseActivity(): IActivity
:IPipeline
addPipelineToResult(pipeline)
loop DFRs
[Read all DFRs in a folder]
parseDFR(node)
parseDFR(node): IDFR
parseDataFlowScript(script)
:IDFR
addDFRToResult(dfr)
loop Datasets
[Read referenced datasets]
parseDataset(node)
parseDataset(node): :IDataset
:IDataset
addDatasetToResult(dataset)
loop linked services
[Read referenced linked service
parseLinkedService(node)
parseLinkedService(node): :ILinkedService
:ILinkedService
addLinkedServiceToResult(linkedService)

Figure 7.3 Sequence diagram of a reader

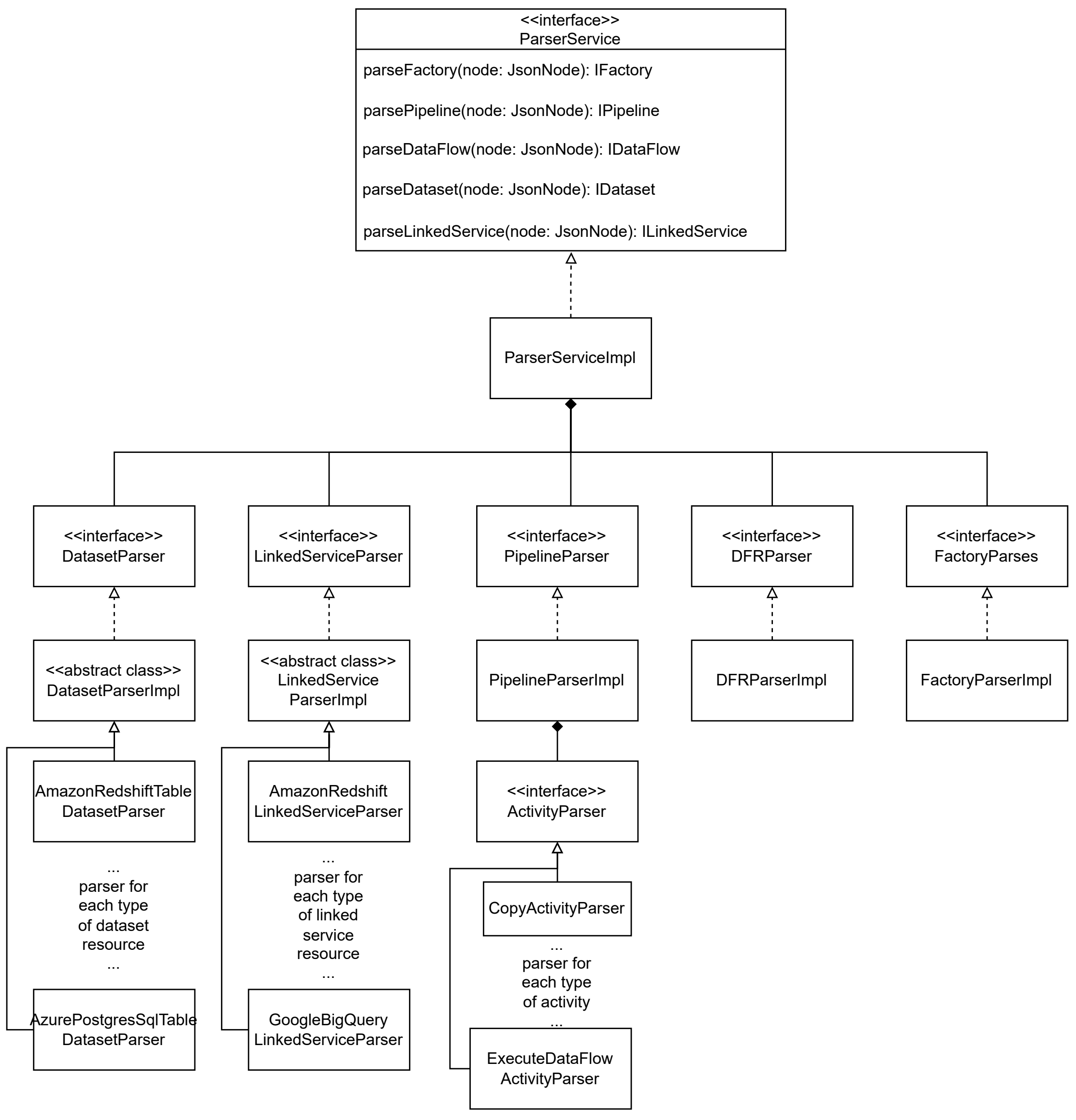
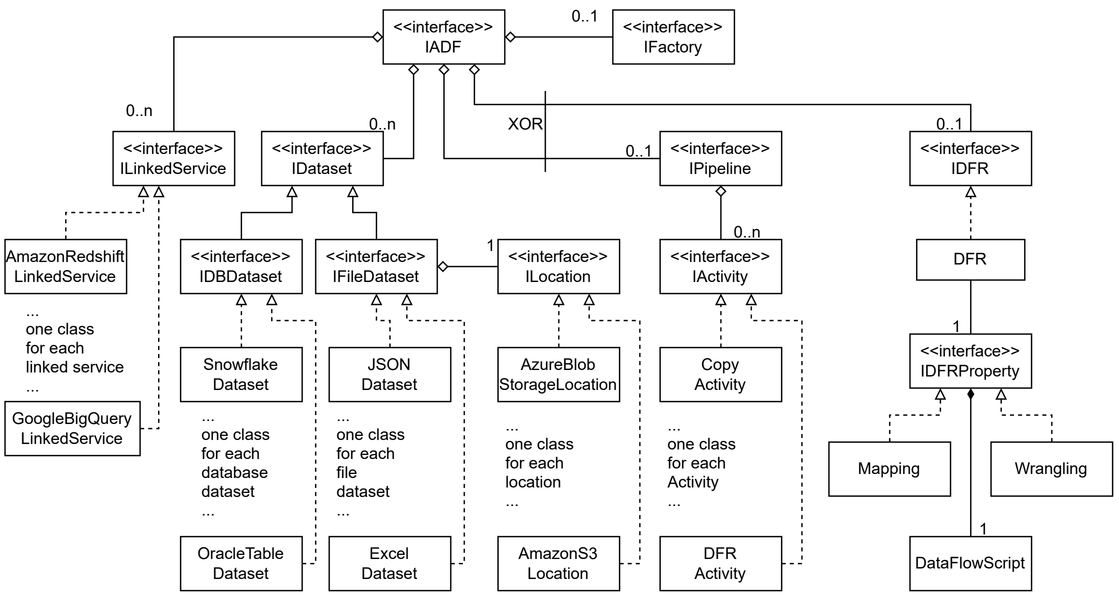


Figure 7.4 Parser service



<<interface>> IADF
<<interface>> IFactory
0..1
0..n
0..n
XOR
0..1
0..1
<<interface>> ILinkedService
<<interface>> IDataset
<<interface>> IPipeline
<<interface>> IDFR
AmazonRedshift LinkedService
... one class for each linked service ...
GoogleBigQuery LinkedService
<<interface>> IDBDataset
<<interface>> IFileDataset
1
<<interface>> ILocation
0..n
<<interface>> IActivity
DFR
1
<<interface>> IDFRProperty
Snowflake Dataset
... one class for each database dataset ...
OracleTable Dataset
JSON Dataset
... one class for each file dataset ...
Excel Dataset
AzureBlob StorageLocation
... one class for each location ...
AmazonS3 Location
Copy Activity
... one class for each Activity ...
DFR Activity
Mapping
Wrangling
1
DataFlowScript

Figure 7.5 Model

resource is Pipeline or DFR. However, there cannot be both. This fact is denoted in the hierarchy by *XOR* between relations from `IADF` to `IPipeline` and `IDFR`.

`IPipeline` provides all activities in a pipeline. Because the pipeline can be empty, its activities are not required. `IDFR` on the other hand, provides the required `property`. The `Property` can be of two types wrangling and mapping. Nevertheless, the most important thing the `property` has is `DataFlowScript`. The `DataFlowScript` is the representation of the data flow script, and it is crucial for the analysis of DFR. More about `DataFlowScript` in the next section 7.1.4.

The next resource which requires a closer look is the dataset. Unlike linked services, where there is no common ancestor, datasets are split into two groups: File datasets and database datasets, because each has something in common. For instance, database datasets have a schema, and a table and file datasets have a location. The location stores information about the specific folder and a file. It is separated into multiple classes because each type of location has a few different attributes. It can be URL, bucket name, file system, and other information based entirely on the type of location.

7.1.4 Data flow script processing

The data flow script, which was described in section 6.2.1 of the previous chapter, is not generally used file format. This means there is no already created third-party parser that could be used. The parser has to be built from scratch. Luckily there are technologies that can generate parsers from the grammar of the parsed language. These technologies were described in section 4.5. The parser generator used for this work is called ANTRL, and its syntax and inner workings were also in the section 4.5. More specifically, ANTLR3 because it offers to define AST directly in the grammar. Processing of data flow script is split into two steps: Creating AST and creating a model from the AST.

AST is created by the parser. Each AST node implements one interface that defines `resolve` method. The method takes `IDataFlowScript` as its parameter and returns updated version of it. Design of AST is in figure 7.6. The *concreteTransformation* node represents concrete transformation. Nodes visible in the tree under the concrete transformation node are common to all the transformations. However, each transformation will be slightly different. It will add

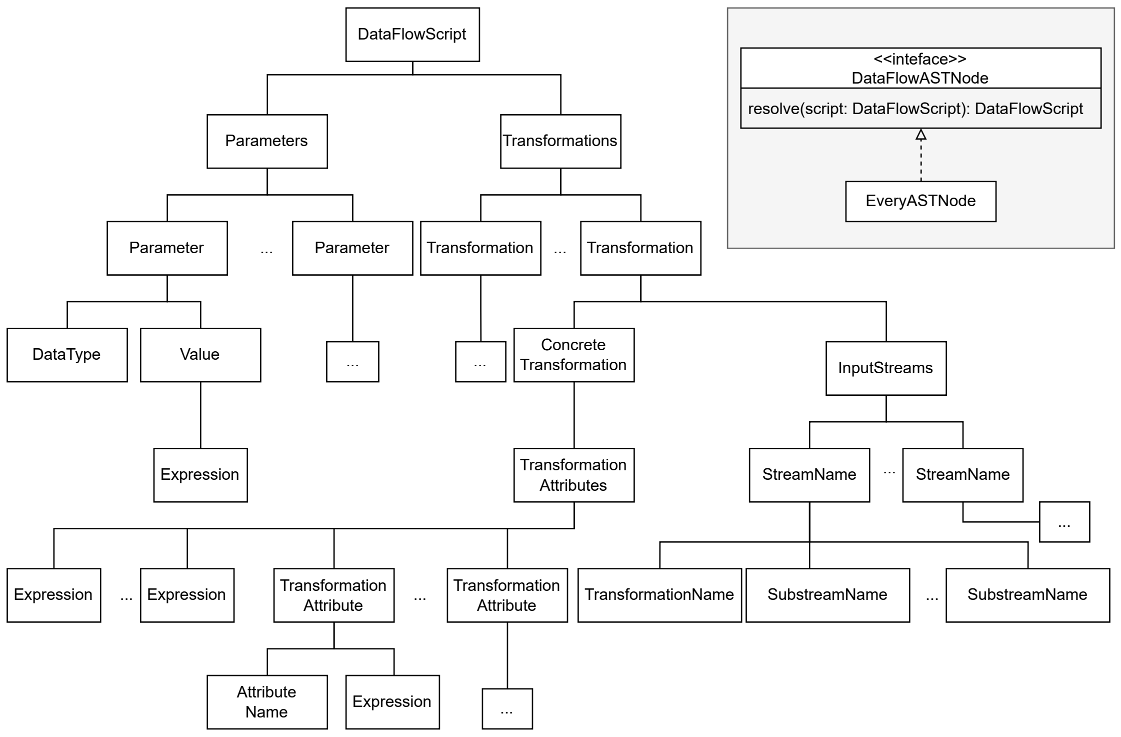


Figure 7.6 AST of data flow script

more types of attributes. The output attribute, which defines columns for the whole DFR, is in figure 7.7. Output transformation defines columns, their data type, and format for data types.

When the AST is created it needs to be processed into the model by the `resolve` method. Sequence diagram of this is in figure 7.8. Each node implements `resolve` differently. However it always ends with the creation of model class and addition of the class to the `DataFlowScript`. Extension of the model from figure 7.5 is in figure 7.9. Extension includes `DataFlowScript`. The data flow script contains parameters and a transformation table. The table can deliver transformation based on its name. This is used later in the data flow generator to find transformations.

Each transformation has information about its name and previous and following transformations. It also contains a table of its columns. Columns are divided into three types: Column definition, a column reference, and column matching. *Column definition* is the type that source transformation uses. *Column reference* is used by every transformation that explicitly specifies columns from the previous transformation. By default, transformations reuse all columns from the previous transformations. Nevertheless, this is not always true. When the transformation uses only specific columns, it will use the column reference. *Column matching* is the most complicated type. It is used by the same transformations as the *Column reference*. However, because transformations can use conditions and pattern matchings instead of ordinary references, this type must exist.

7.2 Data flow generator

The data flow generator is the second part of the analyzer. Its purpose is to create a graph of the ADF from the input model `IADF`. This work will focus only on the data flow of DFR because pipelines do not describe data transformations in a such big way as DFRs do, so they are considered out of the scope of the work.

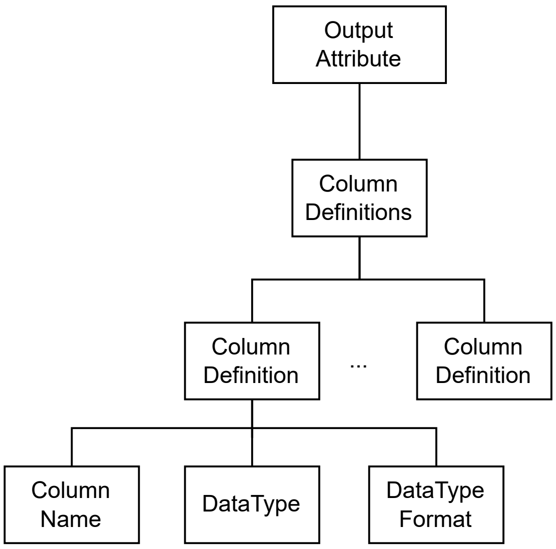


Figure 7.7 AST of output attribute

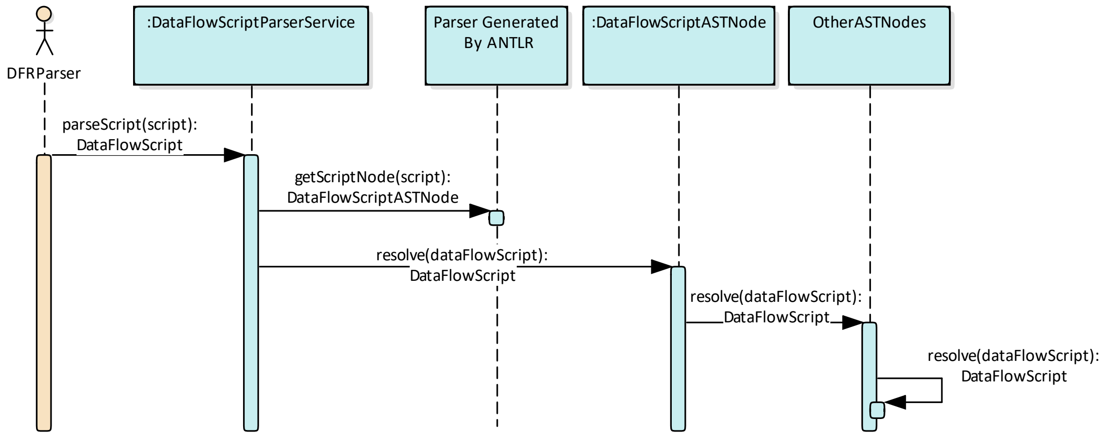

DFRParser
:DataFlowScriptParserService
Parser Generated By ANTLR
:DataFlowScriptASTNode
OtherASTNodes
parseScript(script): DataFlowScript
getScriptNode(script): DataFlowScriptASTNode
resolve(dataFlowScript): DataFlowScript
resolve(dataFlowScript): DataFlowScript
resolve(dataFlowScript): DataFlowScript

Figure 7.8 Processing of data flow script

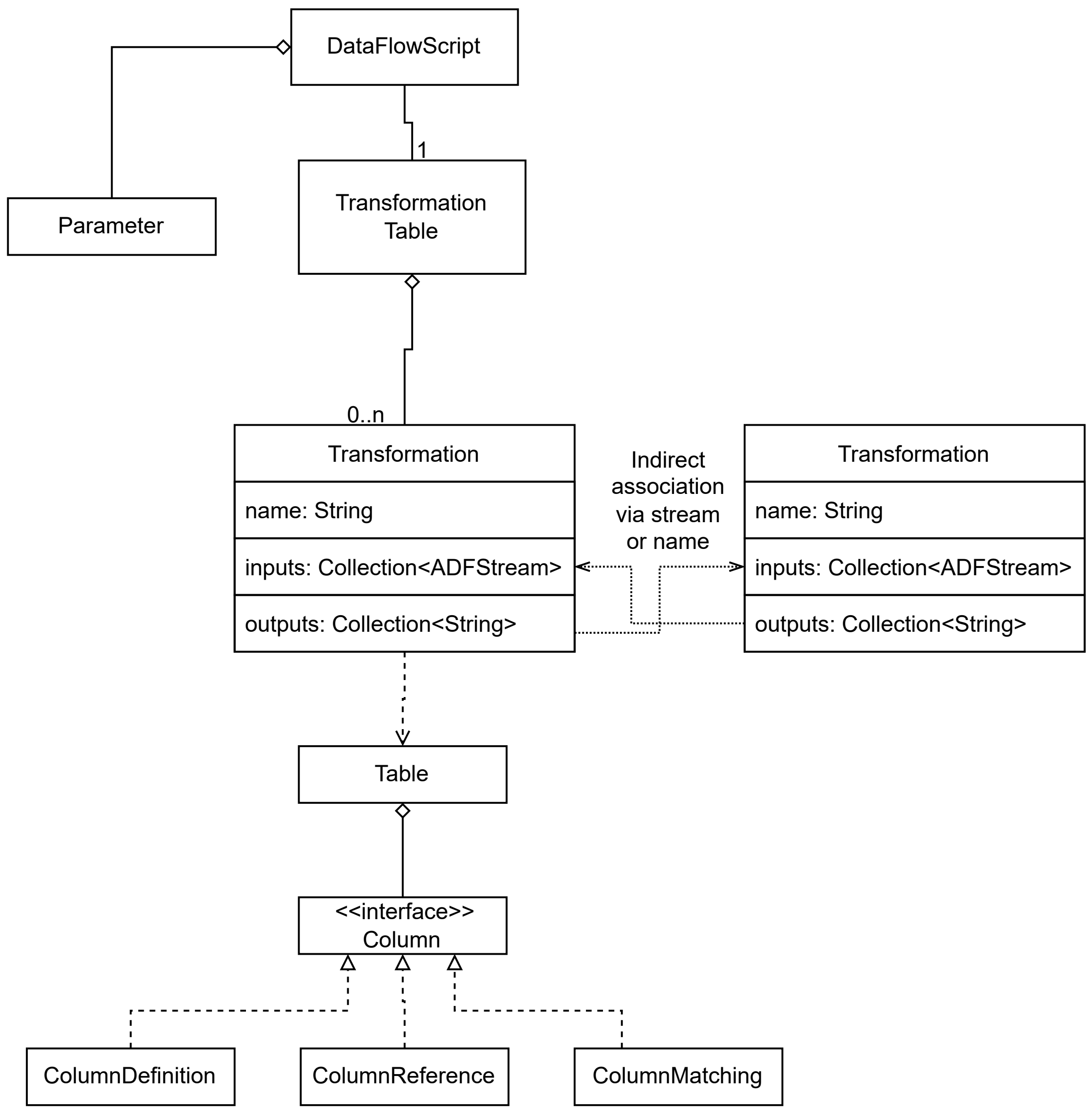


Figure 7.9 Data flow script model

DFR transformations are analyzed one by one. Each transformation has its analyzer class that creates graph nodes of the transformation. The output of the analyzer is column nodes of the analyzed transformation. Column nodes are passed to the analyzer of the following transformation so it can connect its new columns to previous ones. Because of this, transformations have to be analyzed in the correct order. The order is established at the beginning of the data flow generator by topologically sorting transformations.

Which columns are created and how the columns are connected is determined by the category of columns (see figure 7.9). If the transformation does not contain any table, the columns from the previous transformation are copied and connected to the original. If there is `ColumnDefinition`, a new column is created, and it is not connected to anything. `ColumnReference` copies referenced column and connects the new column to it. This is very similar to the transformation without any columns. The difference is that the new column can have a different name than the referenced one. `ColumnMatching` creates columns defined in the matching and connects to it based on some condition. All this is achieved by visitor pattern. Each type of column has a visit method, and the visitor creates and connects columns. This ensures that future column types and connections can be easily added to the scanner without changing any functionality or interface.

7.2.1 Connection outside ADF

Besides transformation that transforms data. DFR contains sources and sinks. Both have to be connected to the corresponding endpoint. The endpoint could be a database, filesystem, or service that can deliver and accept data. Datasets and linked services define endpoints.

Each source and sink transformation has a dataset or linked service reference. This reference can be used to retrieve a real dataset or linked service from the analyzer context. Analyzer context is a class that holds data necessary for the analysis. Datasets and linked services are types of data saved inside the analyzer context. Analyzer context is passed to every transformation analyzer. So even source and sink analyzers have access to it.

Source and sink analyzer also have another dependency. It is `EndpointConnector`. Endpoint connector is interface that provides two methods: `ConnectToDataset` and `ConnectToLinkedService`. There are two implementations of `EndpointConnector` one connects columns to the input endpoint, and one connects them to the sink endpoint. Both are realized by a visitor because every dataset and linked service has a different way of connecting to their respective technology. There are some similarities which are also taken into account. The visitor was chosen because there will be more technologies that will be added in the future. The visitor also ensures type safety which is beneficial. Class diagram of data flow generator is in figure 7.10. Sequence diagram of data flow generation in 7.11.

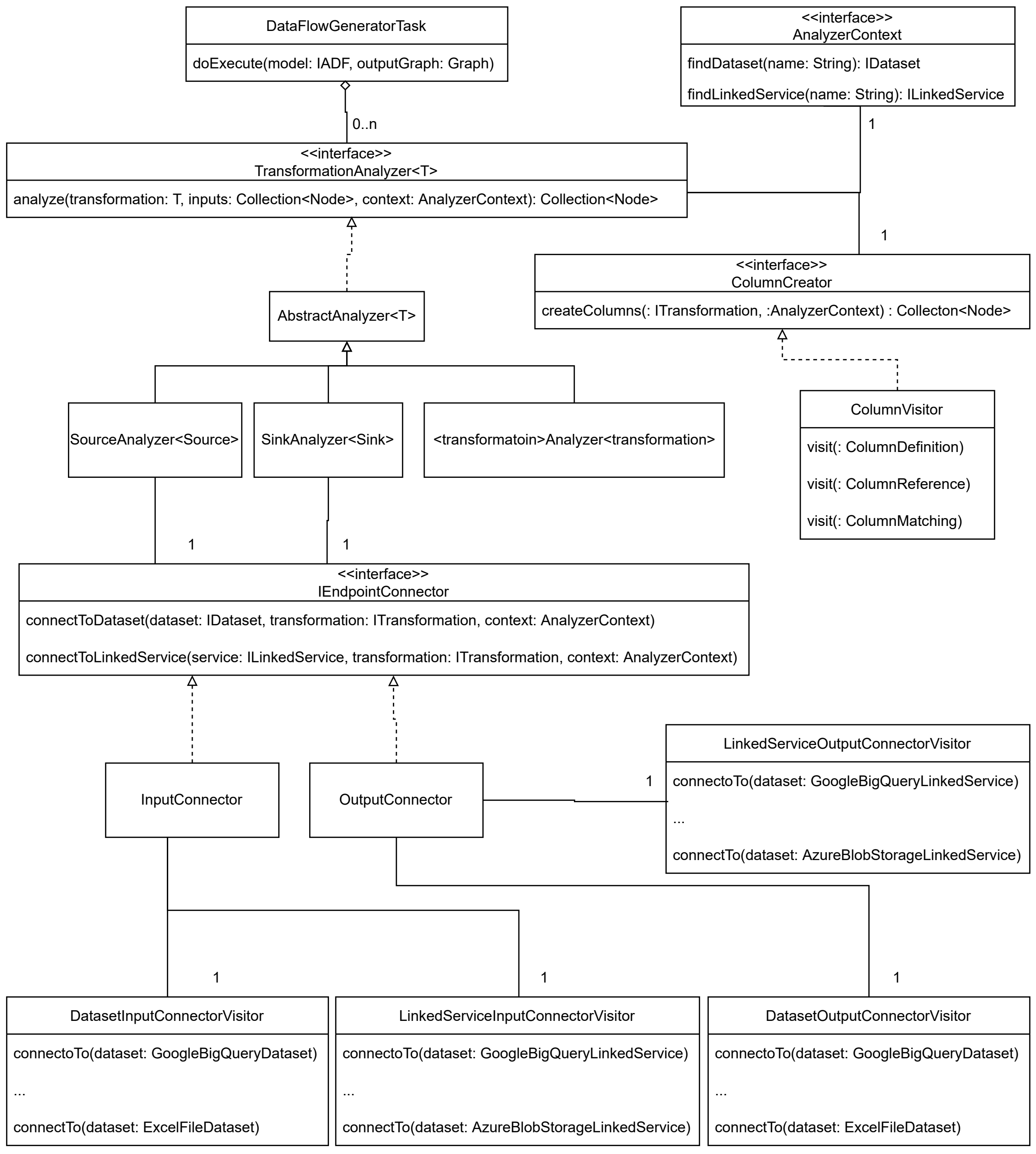
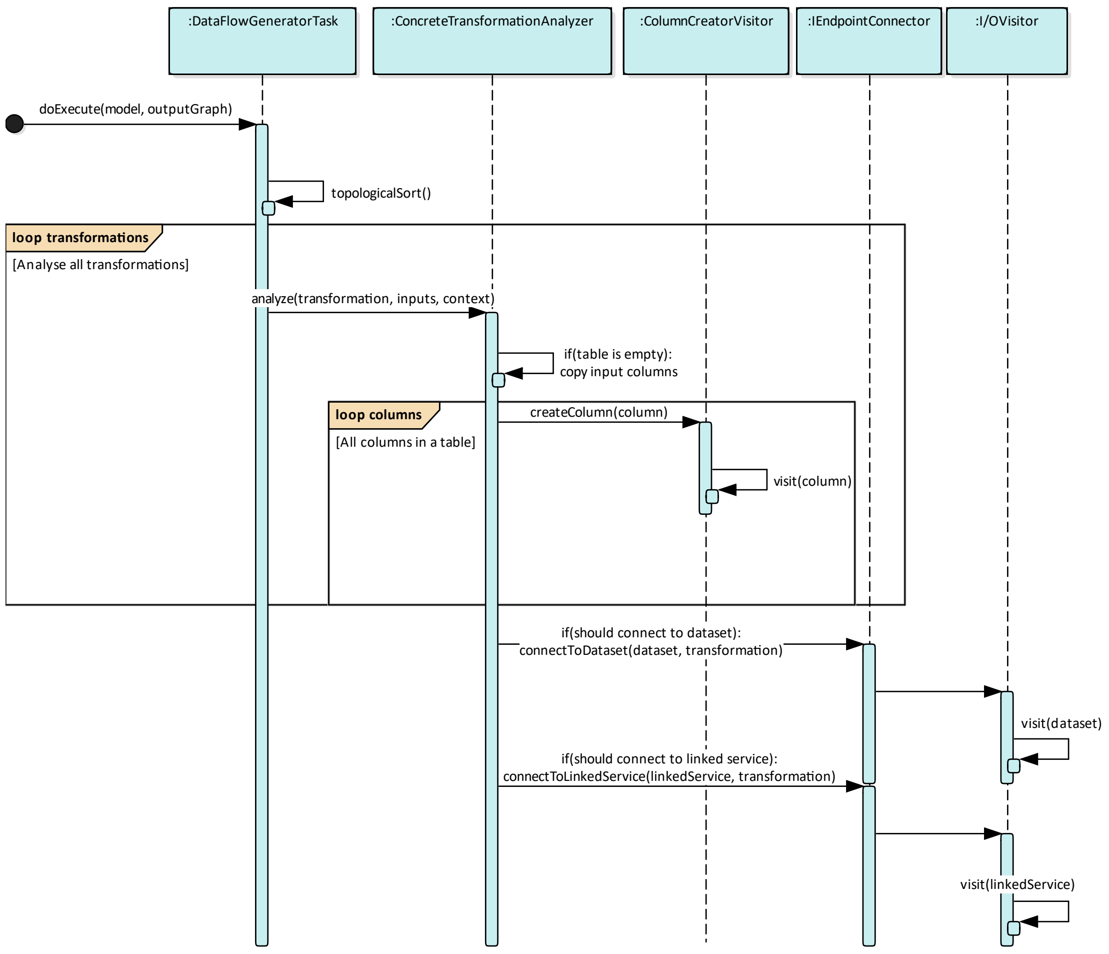

DataFlowGeneratorTask
doExecute(model: IADF, outputGraph: Graph)
<<interface>>
AnalyzerContext
findDataset(name: String): IDataset
findLinkedService(name: String): ILinkedService
0..n
1
<<interface>>
TransformationAnalyzer<T>
analyze(transformation: T, inputs: Collection<Node>, context: AnalyzerContext): Collection<Node>
1
<<interface>>
ColumnCreator
createColumns(: ITransformation, :AnalyzerContext) : Collecton<Node>
AbstractAnalyzer<T>
SourceAnalyzer<Source>
SinkAnalyzer<Sink>
<transformatoin>Analyzer<transformation>
ColumnVisitor
visit(: ColumnDefinition)
visit(: ColumnReference)
visit(: ColumnMatching)
1
1
<<interface>>
IEndpointConnector
connectToDataset(dataset: IDataset, transformation: ITransformation, context: AnalyzerContext)
connectToLinkedService(service: ILinkedService, transformation: ITransformation, context: AnalyzerContext)
LinkedServiceOutputConnectorVisitor
connectoTo(dataset: GoogleBigQueryLinkedService)
...
connectTo(dataset: AzureBlobStorageLinkedService)
1
InputConnector
OutputConnector
1
1
1
DatasetInputConnectorVisitor
connectoTo(dataset: GoogleBigQueryDataset)
...
connectTo(dataset: ExcelFileDataset)
LinkedServiceInputConnectorVisitor
connectoTo(dataset: GoogleBigQueryLinkedService)
...
connectTo(dataset: AzureBlobStorageLinkedService)
DatasetOutputConnectorVisitor
connectoTo(dataset: GoogleBigQueryDataset)
...
connectTo(dataset: ExcelFileDataset)

Figure 7.10 Classes of data flow generator


:DataFlowGeneratorTask
:ConcreteTransformationAnalyzer
:ColumnCreatorVisitor
:IEndpointConnector
:I/OVisitor
doExecute(model, outputGraph)
topologicalSort()
loop transformations
[Analyse all transformations]
analyze(transformation, inputs, context)
if(table is empty):
copy input columns
loop columns
[All columns in a table]
createColumn(column)
visit(column)
if(should connect to dataset):
connectToDataset(dataset, transformation)
visit(dataset)
if(should connect to linked service):
connectToLinkedService(linkedService, transformation)
visit(linkedService)

■ **Figure 7.11** Sequence diagram of data flow generator

Chapter 8
Scanner implementation

This chapter will focus on the implementation of the proof-of-concept scanner. It will present code examples and provide essential details about the implementation which were not considered in the design. Not everything designed was needed for the proof-of-concept scanner. Another aspect this section will present is self-reflection about what could be done better or differently.

8.1 Connector

The connector is separated into three packages as stated in the design of connector 7.1. The original design is to put the parser service into the *resolver* package. However, the better solution would be to put it inside the package together with *reader* because only the reader calls parser service, and it handles only JSON data which is directly connected to the reader.

8.1.1 Reader

ADF reader extends abstract reader from Manta. It works in Manta's standard fashion. File is passed into `readFile` method. Which file is chosen is determined by the `setInputfile` method.

```java
@Override
public void setInputFile(File inputFile) {

    // Filters so only important resources are scanned
    List<IOFileFilter> resourceFilters = Arrays.asList(
          new ParentIsFilter(PIPELINE_FOLDER),
          new ParentIsFilter(DATAFLOW_FOLDER));

    super.setInputFile(inputFile);

    // Set all the filters
    setFilter(
          new OrFileFilter(
                new AndFileFilter(
                   new SuffixFileFilter(".json"),
                   new OrFileFilter(
                         resourceFilters)),
                DirectoryFileFilter.DIRECTORY)}
```

Filters are set so only JSON files in folders `PIPELINE_FOLDER` and `DATAFLOW_FOLDER` are scanned. These constants stand for pipeline and dataflow folders. Example of `readFile` method is below.

```java
@Override
protected IADF readFile(File file) throws IOException {
    LOGGER.info("Reading file: " + file.getPath());

    IADF adf = new ADF();
    try{
         // Setup reader
        ObjectMapper mapper = new ObjectMapper();
        JsonNode rootNode = mapper.readTree(file);

        // Check what kind of resource we are dealing with
        if(parentIs(file, PIPELINE_FOLDER)) {
            return readPipeline(rootNode);
        }
        else if(parentIs(file, DATAFLOW_FOLDER)){
            return readDataFlow(rootNode);
        }
        else {
            LOGGER.log(Categories.inputStructureErrors().unsupportedResourceType().file(
                Paths.get(file.getAbsolutePath()).normalize().toString()));
            return adf;
        }
    } catch (IOException | NullPointerException e) {
        LOGGER.log(Categories.inputStructureErrors().parsingJson().file(Paths.get(file.
            getAbsolutePath()).normalize().toString()).catching(e));
        return adf;
    } catch (UnsupportedOperationException e) {
        LOGGER.log(Categories.inputStructureErrors().unsupportedResourceType().file(
            Paths.get(file.getAbsolutePath()).normalize().toString()).catching(e));
        return adf;
    } catch (Exception e) {
        LOGGER.log(Categories.inputStructureErrors().parsingJson().file(Paths.get(file.
            getAbsolutePath()).normalize().toString()).catching(e));
        return adf;
    }
}
```

`readFile` is split into two additional methods, which read pipeline or dataflow, respectively. `readPipeline` throws `UnsupportedOperationException` for now, because pipelines are not required to analyze transformations inside DFR. `readDataFlow` calls the parser service to process DFR and after that calls the parser service again to process dataset and linked service references. The output of all these methods is the `IADF` which represents the scanned DFR or pipeline with datasets and linked services.

In the future, the DFR could be read only when the pipeline references it. This would lead to the edit of input filters and `readFile` method. It would be treated similarly to datasets and linked services in how they are read. Datasets and linked services could also be cached when they are read. The current implementation always rereads them when multiple DFR reference them. This would be naturally extended to DFRs.

8.1.2 Parser service

The parser service is a facade for all the resource parsing needed. In reality, there are many small specialized parsers for each JSON structure. The implementation of the dataset and linked service parsers was done partly by Kyrylo Bulat. An example of an abstract dataset parser, the parent class for all the dataset parsers, is in the code below.

```java
public final T parseDataset(JsonNode datasetNode) {
   T dataset = null;
   if (JsonUtil.isNotNullAndNotMissing(datasetNode)) {
      String name = JsonUtil.pathAsText(datasetNode, "name");

      JsonNode propertiesNode = datasetNode.path("properties");

      String type = JsonUtil.pathAsText(propertiesNode, "type");

      IReference lsName = referenceParser.parseReference(propertiesNode.path("
          linkedServiceName"), ReferenceType.LINKED_SERVICE);

      if(lsName == null){
         LOGGER.log(Categories.parsingErrors().missingLinkedServiceReferenceOfDataset
             ().datasetName(name));}

      IParameters parameters = null;
      if (parametersParser != null) {
         parameters = parametersParser.parseParameters(propertiesNode.path("
             parameters"));}
      dataset = parseDatasetSpecificFields(
            name,
            type,
            lsName,
            parameters,
            parseSchema(propertiesNode.path("schema")),
            propertiesNode.path("structure"),
            propertiesNode.path("typeProperties")
      );
   }
   return dataset;
}
```

Although having specialized parsers can be great if the parsing is done in a lot different places and classes are reused, it proved to be unnecessary in the case of ADF's proof-of-concept scanner. Every parser is saved in a map with the key being *type of the resource* the parser processes. This results in many steps in the configuration. Example of the configuration file for the parser service:

```xml
<util:map id="datasetParserMap"
     map-class="java.util.HashMap"
     key-type="eu.profinit.manta.connector.datafactory.model.dataset.DatasetType"
     value-type="eu.profinit.manta.connector.datafactory.resolver.service.dataset.
         DatasetParser">
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    ORACLE_TABLE }"
      value-ref="oracleTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    AZURE_SQL_DW_TABLE }"
      value-ref="azureSqlDWTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    AZURE_POSTGRESQL_TABLE }"
      value-ref="azurePostgreSqlTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    NETEZZA_TABLE }"
      value-ref="netezzaTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    AMAZON_REDSHIFT_TABLE }"
```

```
      value-ref="amazonRedshiftTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    DB2_TABLE }"
      value-ref="db2TableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    GOOGLE_BIGQUERY_OBJECT }"
      value-ref="googleBigQueryObjectDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    GREENPLUM_TABLE }"
      value-ref="greenplumTableDatasetParser"/>
<entry key="#{ T(eu.profinit.manta.connector.datafactory.model.dataset.DatasetType).
    HIVE_OBJECT }"
      value-ref="hiveObjectDatasetParser"/>
```

A similar idea of using the map to hold specialized classes was used in the data flow generator for transformation analyzers. However, the key to the map was replaced by a more automatic way, without the explicit declaration. More in the section 8.2.

8.1.3 Data flow script processing

The data flow script, as described in the design in section 7.1.4, is realized by the ANTLR3 parser. The parser is split into three grammars: Lexer, parser, and reserved keywords. The lexer grammar generates a lexical analyzer. Example of the grammar:

```
ARRAY_DECIMAL : 'DECIMAL[]';
ARRAY_STRING : 'STRING[]';
DECIMAL_TYPE
    : KW_DECIMAL LEFT_PARENT (DIGIT)* COMMA (DIGIT)* RIGHT_PARENT
    ;

AT_SIGN :   '@';
DOLLAR_SIGN: '$';
LEFT_PARENT :  '(';
RIGHT_PARENT :  ')';
LEFT_BRACKET :  '[';
RIGHT_BRACKET :  ']';
LEFT_BRACE :  '{';
```

It is a simple grammar capable of creating tokens for the parser. Nevertheless, this does not mean every grammar rule is straightforward. Here is an example of a more complex rule recognizing floating-point literals:

```
INTEGER_LITERAL
    : (MINUS_SIGN | PLUS_SIGN)? DIGIT+
    ;

FLOATING_POINT_LITERAL
    : (MINUS_SIGN | PLUS_SIGN)? DIGIT* PERIOD INTEGER_LITERAL ( ('e' | 'E')
        INTEGER_LITERAL)?
    | (MINUS_SIGN | PLUS_SIGN)? DIGIT* ('e' | 'E') INTEGER_LITERAL
    ;
```

The rule does not represent the syntactically correct ADF floating-point literal completely. The problem comes with the string `-12.-34e-56`. The rule allows such string, and the ADF does not. However, this rule is good enough because the scanner can recognize a superset of a language.

Tokens created by the lexer are passed to the parser. The parser grammar contains all syntactic rules of the data flow script. The proof-of-concept scanner does not need every expression or construct to be processed. That is the reason not everything is implemented. However, the grammar has to contain everything necessary. The grammar rule for the transformation looks like this:

```
transformations
    : (tran+=transformation OUTPUT_STREAM sn+=stream_name)+
       -> ^(AST_TRANSFORMATIONS ^(AST_TRANSFORMATION<AstTransformation>[contextState]
           $tran OUTPUT_STREAM $sn)+ )
    ;
    transformation
    : st=source_transformation
       -> $st
    | is=inputStreams twi=transformation_with_inputs
       -> $is $twi
    ;

source_transformation
    : KW_SOURCE ta=transformation_attributes
       -> ^(AST_CONCRETE_TRANSFORMATION<AstSourceTransformation>[contextState]
           KW_SOURCE $ta)
    ;

transformation_with_inputs
    : ft=filter_transformation
       -> $ft
    | jt=join_transformation
       -> $jt
    | st=sink_transformation
       -> $st
    | gt=general_transformation
       -> $gt
    ;
```

The example describes the following syntactic structure below.

```
source(attributes)~><outputSource>
<input> transformation(attributes) ~> <output>
...
<input2> transformation2(attributes) ~> <output2>
```

The grammar is not yet complete. However, all transformations are parsed because of the rule `general_transformation`. It allows transformations to be partially parsed and get their name and attributes. General transformation grammar can be seen below.

```
general_transformation
    : SIMPLE_ID ta=transformation_attributes
       -> ^(AST_CONCRETE_TRANSFORMATION<AstGeneralTransformation>[contextState]
           SIMPLE_ID $ta)
    ;
transformation_attributes
    : LEFT_PARENT cta=common_transformation_attributes? RIGHT_PARENT
       -> LEFT_PARENT $cta? RIGHT_PARENT
    ;

common_transformation_attributes
    : ta1=transformation_attribute (COMMA ta2+=transformation_attribute)*
       -> ^(AST_TRANSFORMATION_ATTRIBUTES<AstTransformationAttributes>[contextState]
           $ta1 (COMMA $ta2)*)
    ;
```

```

transformation_attribute
    : oa=output_attribute
       -> $oa
    | id=all_identifier COLON de=df_expression
       -> ^(AST_TRANSFORMATION_ATTRIBUTE<AstTransformationAttribute>[contextState] ^(
            AST_ATTRIBUTE_NAME<AstIdentifier>[contextState] $id) COLON $de)
    | de=df_expression
       -> $de
    ;
```

Every attribute is a data flow expression. Expressions are not fully implemented yet. However, the current implementation deals with basic constructs such as literals, arithmetic, boolean expressions, and function calls. Implementation of complex expressions was not necessary for the proof-of-concept scanner.

After the parser is generated, it is used inside `DFSParserService`, which executes the parser to get the AST. Part of `DFSParserService` is in the following code snippet.

```
public IDataFlowScript processScript(String script) {

    DFSParser dfsParser = parseDFS(script);
    DFSParser.dfs_return dfsReturn = null;

    try {
       dfsReturn = dfsParser.dfs();
    } catch (RecognitionException e) {
       LOGGER.log(Categories.parsingErrors().DFSParsingFailure()
              .input(script)
              .catching(e));
       return createFailedDFS();}

    Object astRoot = dfsReturn.getTree();

    if (!(astRoot instanceof IAstDFS)) {

       LOGGER.log(Categories.parsingErrors().malformedAst()
              .message("Parsing output was not a DFS"));

       return createFailedDFS();}
    else{
       IAstDFS rootNode = (IAstDFS) astRoot;

       IDataFlowScript dfs = rootNode.resolve(new DataFlowScript(Collections.emptyList
            (), Collections.emptyMap()));

       if (dfsParser.getNumberOfSyntaxErrors() > 0) {
          dfs = dfs.updatedParsedWithoutErrors(false);
       }

       return dfs;}
}
```

After the return from the function `processScript`, the AST is processed via its `resolve` function. The function transforms it to the `IADF` representation. Example of the `resolve` function of root node.

```java
public IDataFlowScript resolve(IDataFlowScript currentDFS) {

    // Fill DFS with parameters
    List<IAstParameter> parameters = findParameters();
    for(IAstParameter parameter : parameters){
        currentDFS = parameter.resolve(currentDFS);
    }

    // Fill DFS with transformations
    List<IAstTransformation> transformations = findTransformations();
    for(IAstTransformation transformation : transformations){
        currentDFS = transformation.resolve(currentDFS);
    }

    return currentDFS;
}
```

In the code above, someone could spot how the AST is splitting into two branches, the parameters branch, and the transformation branch. This is according to the design of the AST that was in figure 7.6.

8.2 Dataflow generator

Data flow generator starts from the `dataFlowTask` class, where `IADF` input model is recieved together with the output parameter graph.

```java
 protected void doExecute(IADF input, Graph outputGraph) {

    if(input == null || !input.isValid()){
       LOGGER.log(Categories.dataFlowErrors().dataFactoryInputIsInvalid());
       return;
    }

    ADFGraphHelper graphHelper = new ADFGraphHelper(
          outputGraph,
          getScriptResource(),
          getScriptResource(),
          NodeType.COLUMN_FLOW,
          locationNodeCreator,
          nodeCreator);

    // Create context
    AnalyzerContext context = new AnalyzerContextImpl();

    analyzer.analyze(input, context, null, graphHelper);

}
```

First, the input is tested whether it is correct or not. Then graph helper and analyzer context are created. Graph helper helps to create the output graph. It contains methods like `addTransformationNode` and `connectNodesByName`, ... Analyzer context was already described a little in the previous chapter about design. It contains everything that could be needed during analysis. Example of analyzer context code:

```java
@Override
public void setDatasets(Collection<IDataset> datasets) {
   for(IDataset dataset : datasets){
      this.datasets.put(dataset.getName(), dataset);
   }
}

@Override
public void setLinkedServices(Collection<ILinkedService> linkedServices) {
   for(ILinkedService linkedService : linkedServices){
      this.linkedServices.put(linkedService.getName(), linkedService);
   }
}
```

These methods add datasets and linked services. They are used only at the beginning of the analysis to set everything needed.

After the context is configured, the DFR can be analyzed. Mapping DFR is analyzed in the following method.

```java
private void analyzeMappingDF(IMappingDataFlowProperties properties, AnalyzerContext
    context, Node parentNode, ADFGraphHelper helper){
   LOGGER.info("Analyzing mapping data flow");

   //Analyze Sources and Sinks
   Map<String, IDataFlowEndpoint> sources = properties.getSources();
   Map<String, IDataFlowEndpoint> sinks = properties.getSinks();

   Map<String, IDataFlowEndpoint> endpoints = new HashMap<>(sources);
   endpoints.putAll(sinks);
   context.setDataflowEndpoints(endpoints);

   List<String> transformations = new ArrayList<>(properties.getTransformationNames())
       ;
   transformations.addAll(endpoints.keySet());
   List<ITransformation> sortedTransformations = sortTransformations(transformations,
       properties.getScript().getTransformations());

   for(ITransformation transformation : sortedTransformations){
      analyzerDispatcher.analyzeTransformation(transformation, context, parentNode,
          helper);}
}
```

In the beginning, sources and sinks are retrieved from the model of mapping DFR. After that, endpoint datasets and linked service names are registered in the context. Finally, the names of transformations are transformed into transformation models. The correct analyzer is chosen, and all transformations are analyzed in a for loop.

How the correct analyzer is chosen was already mentioned in the section about parser service. The analyzer is chosen by a class called `AnalyzerDispatcher`. The `AnalyzerDispatcher` contains a map of analyzers. The map is set in the following way:

```java
public void setAnalyzers(Collection<TransformationAnalyzer<?>> analyzersToAdd) {
   for (TransformationAnalyzer<?> analyzer : analyzersToAdd) {
      analyzers.put(analyzer.getAnalyzedType(), analyzer);
   }
}
```

The map is configured in a much simpler way than the dataset and linked service map from the section 8.1.2. Example of the configuration:

```
<property name="analyzers">
   <list>
      <ref bean="datafactorySourceAnalyzer" />
      <ref bean="datafactorySinkAnalyzer" />
      <ref bean="datafactoryFilterAnalyzer" />
      <ref bean="datafactoryJoinAnalyzer" />
      <ref bean="datafactoryUnknownTransformationAnalyzer" />
```

The disadvantage of this approach is that one has to define what kind of class the analyzer analyzes in the constructor of the analyzer, like so:

```
class AbstractAnalyzer<T extends ITransformation>{
   protected TransformationAnalyzer(Class<T> analyzedType) {
      this.analyzedType = analyzedType;
   }
}

class FilterAnalyzer extends AbstractAnalyzer<IFilterTransformation>{
   public FilterAnalyzer() {
      super(IFilterTransformation.class);
   }
}
```

Nevertheless, this is not a big problem because it also has an enormous advantage, unlike the method used with datasets and linked services. The type of analyzed model will always match with the type analyzer analyzes. The previous method would result in a runtime exception if this would ever happen.

Analyzers analyze transformations one by one. Transformations are topologically sorted. When the transformation is analyzed, all its preceding transformations are already analyzed. The `analyze` function of `UnkownTransformationAnalyzer` can be seen below.

```
public Map<IADFStream, ProcessedStream> analyze(IUnknownTransformation toAnalyze,
    AnalyzerContext context, Node parentNode, ADFGraphHelper helper) {

   LOGGER.info("Analyzing unknown transformation");
   Node transformationNode = helper.addNode(toAnalyze.getName(), NodeType.
       DATAFACTORY_TRANSFORMATION, parentNode);

   Map<IADFStream, ProcessedStream> processedStreams = createStreams(toAnalyze,
       context, transformationNode, helper);

   return processedStreams;

}
```

It creates a node for the transformation. After that, it calls the `createStreams` method, which creates columns of the transformation and connects it to the previous transformation. The `ProcessedStream` represents the output stream of the transformation. The next transformation that will be analyzed can connect to this `ProcessedStream`. `createStreams` method is in the following code snippet.

```
protected Map<IADFStream, ProcessedStream> createStreams(T transformation,
    AnalyzerContext context, Node transformationNode, ADFGraphHelper helper){

      Collection<IADFStream> streams = transformation.getStreams();

      Map<IADFStream, ProcessedStream> processedStreams = new HashMap<>();
```

```

        for(IADFStream stream : streams){

            String substreamName = stream.getSubstreamName();
            if(substreamName.isEmpty())
                substreamName = DEFAULT_STREAM_NAME;

            Node streamNode = helper.addSchemaNode(substreamName, transformationNode);

            List<Node> columnNodes = createColumns(transformation, context, streamNode, helper);
            processedStreams.put(stream, new ProcessedStream(streamNode, columnNodes));

        }
        return processedStreams;
    }
```

It creates a node for each output of the transformation. Inside the node are columns. The node is generated for each stream. In the future, it would be enough if only one node was created for all streams. The multiple streams would reference the one node and its columns.

Columns are created in the following way. First, column names are retrieved by column visitor. The visitor visits column definition, column reference, or column matching as described in the design. In the future, this will create columns directly. In the current implementation, the visitor only retrieves names and creates columns based on the name. After columns are created, the analyzer connects columns to available streams. The important thing to note is that available input streams are not passed as a parameter to the analyzer but are saved in the context. This differs from the design. The decision was made because having it in the context makes sense, and it gets rid of one parameter from the `analyze` method, which already has a few complex parameters.

8.2.1 Connection outside ADF

Connection outside ADF is realized by `IEndPointConnector`. Both transformations responsible for the input and output outside ADF use this interface. The interface uses visitors to connect to datasets and linked services. An example of usage from the `SourceEndpointConnector` can be seen in the following code.

```
    Optional<IDataset> datasetOpt = context.getDataset(endpointReference);
    if(datasetOpt.isPresent())
        return datasetOpt.get().accept(datasetConnector);
    else{
        LOGGER.log(Categories.dataFlowErrors().datasetNotFound().name(endpointReference.getReferenceName()));
        return Optional.empty();
    }
```

Return value of this code is `ProcessedSchema` of the transformation. `ProcessedSchema` exists because transformations can have a dynamic schema, which is determined during the runtime of ADF. The schema cannot be retrieved before the actual run from ADF's metadata. The only way to retrieve the schema is to use a query service, Manta's way of evaluating queries. The visitor `datasetConnector` calls the query service. Here is an example of the creation of columns from the query:

```java
protected Optional<ProcessedTransformation> connectByQuery(Connection connection,
    String query){

   DataflowQueryResult queryResult = queryService.getDataFlow(processedTransformation.
       getTransformationNode(), "QUERY", query, connection);

   // Updated processed streams
   Map<IADFStream, ProcessedStream> newProcessedStreams = new HashMap<>(
       processedTransformation.getProcessedStreams());

   // If schema drift is allowed generate additional columns
   if(allowSchemaDrift){
      newProcessedStreams = generateDriftedStreamsFromResultset(queryResult,
          processedTransformation.getProcessedStreams());
   }

   // Connect result with transformation by name
   for(ProcessedStream stream : newProcessedStreams.values()){
      queryResult.connectAllResultsetsTo(stream.getColumnNodes(), DataflowQueryResult.
          MatchStrategy.MATCH_BY_NAME_CI);
   }

   Node merged = queryResult.mergeTo(graphHelper.Graph());

   processedTransformation.getTransformationNode().addAttribute("QUERY", query);

   return (merged != null)
         ? Optional.of(new ProcessedTransformation(processedTransformation.
             getTransformationNode(), newProcessedStreams))
         : Optional.empty();
}
```

The code above calls the query service. Behind the query service is another analyzer similar to this work's analyzer. However, the analyzer behind the query service analyzes the database or Kafka. After processing the query, the query service returns a resultset. The resultset contains a graph of the query. If the transformation allows it, new columns are generated from the resulset, and at the end, the resultset graph is merged into ADF's graph. An updated current transformation node is returned in case new columns were generated.

Chapter 9

Testing

The code was tested by unit and integration tests. The whole implementation was also tested on real-world data.

9.1 Connector

The connector is covered only by unit tests. The reader was tested on the three data flows, eight different datasets, ten linked services, and five pipelines. Example of dataset:

```
{
  "name": "DelimitedText1",
  "properties": {
    "linkedServiceName": {
      "referenceName": "AzureBlobStorage1",
      "type": "LinkedServiceReference"
    },
    "annotations": [],
    "type": "DelimitedText",
    "typeProperties": {
      "location": {
        "type": "AzureBlobStorageLocation",
        "container": "testcontainer"
      },
      "columnDelimiter": {
        "value": "@variables('comma')",
        "type": "Expression"
      },
      "escapeChar": "\\",
      "quoteChar": "\""
    },
    "schema": []
  },
  "type": "Microsoft.DataFactory/factories/datasets"
}
```

Unit tests also covered all essential classes from the resolver package. One of the most critical parts of this thesis, the data flow script parser, was tested on the real-world data used in the industry. It was shown to be very effective in the initial stages of the development, and many bugs were fixed because of the real-world samples. However, even artificial samples had to be made to cover every aspect of the language. The parser is tested on eight data flow scripts. Example of data flow script testing sample:

```
parameters{
  parameter1 as string ('Default value'),
  parameter2 as date
}
source(output(
         normal as integer '$###',
           {as with __ {{ and $)!*%)@&)!#* more} as timestamp 'yyyy-MM-dd\'T\'HH:mm:ss
                \'Z\'',
           {} as decimal(13,3) '000,000,000.000',
           second_normal as ()
       ),
       allowSchemaDrift: true,
       validateSchema: false,
       ignoreNoFilesFound: false,
     wildcardPaths:[($parameter1),'second_value',('third_value')]) ~> Source1@(
          streamone, streamTwo, streamThree)
```

This sample aims to test the parsing of the column names.

9.1.1 Data flow generator

The data flow generator is covered by unit and integration tests. Unit tests are used for the analyzer context. However, integration tests that cover the whole implementation and test output graph with the expected graph are more interesting. Inputs of the scanner are in the test resource folder. The inputs have the same structure as the actual input would have. The only difference is that testing folders contain a file with the name <resourceName>_expected.txt with the expected graph for the resource. Example of the implemented proof-of-concept scanner inside manta is in figure 9.1.

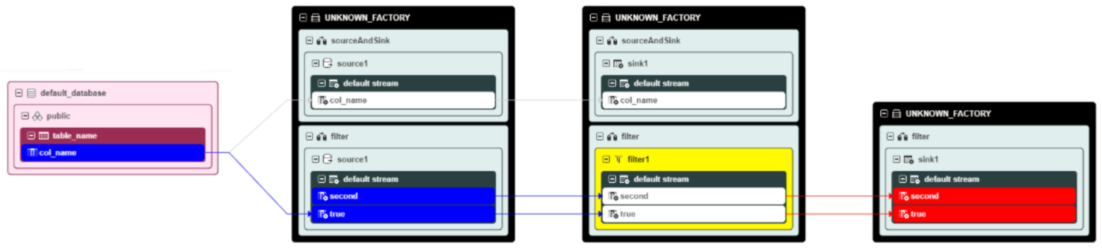


Figure 9.1 Simple DFR data flow

Chapter 10
Summary

The goal of this thesis was to analyze the Manta project and its representation of dataflows , study ADF and research the syntax and semantics of source codes that describe data transformations. Design a way to analyze them so the resulting scanner would be able to detect ADF's internal dataflows and dataflows from and to tools that integrate with it. The last goal was to implement the proof-of-concept scanner, so it is able to extract ADF's dataflows from git repository into the Manta system.

Every goal of the work was acomplished. Manta and ADF constructs were successfuly analyzed. A way to extract dataflows from transformation source codes, located in git, was designed. And it was later implemented into a proof-of-concept scanner integrated that will be later integrated with Manta.

The proof-of-concept scanner will be maintained from now on as part of Manta. The implementation will be later extended to analyze pipelines and even more transformations. Parts of the scanner could also be used for other tools such as Azure synapse analytics, because it shares the same language for data transformations and for it's internal expressions.

Bibliography

1. TIERNEY, Mike. *What Is Data Governance: Definition, Advantages and Process Flow* [online] [visited on 2022-04-11]. Available from: `https://blog.netwrix.com/2019/09/19/data-governance/`.
2. MANTA. *The Ultimate Guide to Data Lineage* [online] [visited on 2022-04-12]. Available from: `https://getmanta.com/library/documents/`.
3. *Azure Data Factory documentation* [online] [visited on 2022-04-12]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/`.
4. NAEEM, Tehreem. *What is an ETL Tool: Definition, Uses, and Use-Cases* [online] [visited on 2022-04-12]. Available from: `https://www.astera.com/type/blog/what-is-etl-tool/`.
5. SAUMELL, Maria. *Connectivity, Graph Traversal, Digraphs, Trees* [online] [visited on 2021-05-09]. Available from: `https://courses.fit.cvut.cz/BIE-AG1/media/lectures/BIE-AG1-lec2-handout.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
6. *What is Data Flow Diagram?* [Online] [visited on 2022-04-15]. Available from: `https://www.visual-paradigm.com/guide/data-flow-diagram/what-is-data-flow-diagram/`.
7. *Git* [online] [visited on 2022-04-26]. Available from: `https://git-scm.com/`.
8. HOLUB, Jan. *Basic notions* [online] [visited on 2021-04-27]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-01-zakladni_pojmy-4.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
9. TRÁVNÍČEK, Jan [personal communication]. 2022.
10. HOLUB, Jan. *Regular expressions* [online] [visited on 2021-04-28]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-04-regularni_vyrazy-4.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
11. HOLUB, Jan. *Context-free Grammars* [online] [visited on 2021-04-28]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-07-bezkontextove_gramatiky-4.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
12. JANOUŠEK, Jan. *LL SYNTAKTICKÁ ANALÝZA* [online] [visited on 2021-04-27]. Available from: `https://courses.fit.cvut.cz/BI-PJP/lectures/files/pjplecture2_2020.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.

13. COOPER, Keith D.; TORCZON, Linda. *Engineering a Compiler*. Second. 2012. ISBN 978-0-12-088478-0.
14. KOZEN, Dexter C. *Automata and computability*. Springer, 1997. ISBN 0387949070.
15. GEEKSFORGEEKS. *Introduction of Finite Automata* [online] [visited on 2022-04-19]. Available from: `https://www.geeksforgeeks.org/introduction-of-finite-automata/`.
16. HOLUB, Jan. *Deterministic and nondeterm. finite automata* [online] [visited on 2021-04-28]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-02-konecne_automaty-4.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
17. ŠESTÁKOVÁ, Eliška. *Automaty a gramatiky Sbírka řešených úloh*. Harlow: ČVUT, 2020. ISBN 978-80-01-06306-4.
18. HOLUB, Jan. *Pushdown automata* [online] [visited on 2021-04-28]. Available from: `https://courses.fit.cvut.cz/BIE-AAG/lectures/bie-aag-08-zasobnikove_automaty-4.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
19. AHO, Alfred V. et al. *Compilers: Principles, Techniques, and Tools*. Second. Greg Tobin, 2007. ISBN 978-0321486813.
20. JANOUŠEK, Jan. *LL SYNTAKTICKÁ ANALÝZA - IMPLEMENTACE, VLASTNOSTI LL GRAMATIK* [online] [visited on 2021-04-29]. Available from: `https://courses.fit.cvut.cz/BI-PJP/lectures/files/pjplecture3_2020.pdf`. the file is accesible after loging to CTU network - copy of the file is saved on the attached SD card.
21. PARR, Terence; FISHER, Kathleen S. *LL(*): The Foundation of the ANTLR Parser Generator* [online] [visited on 2022-04-20]. Available from: `https://www.antlr.org/papers/LL-star-PLDI11.pdf`.
22. PARR, Terence. *LL(*) grammar analysis* [online] [visited on 2022-04-29]. Available from: `https://theantlrguy.atlassian.net/wiki/spaces/~admin/pages/524294/LL+grammar+analysis`.
23. PARR, Terence. *Grammars* [online] [visited on 2022-04-30]. Available from: `https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687027/Grammars`.
24. PARR, Terence J.; QUONG, Russell W. *Adding Semantic and Syntactic Predicates To LL(k): pred-LL(k)* [online] [visited on 2022-04-22]. Available from: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.427&rep=rep1&type=pdf`.
25. *Manta live* [online] [visited on 2022-04-25]. Available from: `https://service.getmanta.com/manta-dataflow-server-gallery/viewer/dataflow?revision=1&olderRevision=0&object=&selectedItems=216&level=BOTTOM&direction=BOTH&filter=2&depth=3&hint=column-data-lineage`.
26. KRÁTKÝ, Tomáš. *Different Approaches To Data Lineage* [online] [visited on 2022-04-15]. Available from: `https://getmanta.com/blog/different-approaches-to-data-lineage/`.
27. *Pipelines and activities in Azure Data Factory and Azure Synapse Analytics* [online] [visited on 2022-04-25]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/concepts-pipelines-activities`.
28. *Power Query documentation* [online] [visited on 2022-04-25]. Available from: `https://docs.microsoft.com/en-us/power-query/`.
29. *Datasets in Azure Data Factory and Azure Synapse Analytics* [online] [visited on 2022-04-25]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/concepts-datasets-linked-services`.

30. *Derived column transformation in mapping data flow* [online] [visited on 2022-04-25]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/data-flow-derived-column`.

31. *Expressions and functions in Azure Data Factory and Azure Synapse Analytics* [online] [visited on 2022-05-02]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/control-flow-expression-language-functions`.

32. *Global parameters in Azure Data Factory* [online] [visited on 2022-05-02]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/author-global-parameters`.

33. *System variables supported by Azure Data Factory and Azure Synapse Analytics* [online] [visited on 2022-05-02]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/control-flow-system-variables`.

34. *Data transformation expression usage in mapping data flow* [online] [visited on 2022-05-03]. Available from: `https://docs.microsoft.com/en-us/azure/data-factory/data-flow-expressions-usage`.

Contents of the attached SD card

```
readme.txt ............................ Brief description of the content of the card
src
  implementation ........................................ Implementation source codes
  thesis .................................... Source codes of the thesis in LaTeX format
text ............................................................... Text of the thesis
  lectures ........................ Lectures from FIT CTU, that were used in the thesis
  thesis.pdf ........................................... Text of the thesis PDF format
```