CHALMERS

Undersökning om modernisering av äldre system med hjälp av ANTLR

Examensarbete inom högskoleingenjörsprogrammet Datateknik

Alexander Spetz

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK

CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2025
www.chalmers.se

Undersökning om modernisering av äldre system med hjälp av ANTLR

Alexander Spetz,
Chalmers tekniska högskola, Institution för data- och informationsteknik
Göteborg, Sverige 2025
www.chalmers.se

Sammanfattning

Många stora företag har något som kallas kärnsystem som funnits länge och håller på att bli föråldrade. Företaget som detta arbete görs i samarbete med, Apper Systems AB, arbetar med modernisering av applikationer som dessa kärnsystem. De vill därför testa många olika sätt att modernisera applikationer. Detta arbete har för syfte att testa programvaran ANTLR, som kan bygga Parser utifrån skapade grammatikregler, och undersöka om den kan effektivt användas för modernisering av kod i framtiden.

I arbetet har SQL-tabeller byggts upp utifrån metakod-exempel som har tillhandahållits av Apper Systems AB. För att kunna undersöka olika format så har tester gjorts med både XML och JSON. Först har det byggts upp tabeller med XML och JSON vilka fungerade utan problem. Efter det så har även en mer avancerad XML-fil som innehöll en hel call stack undersökts, men Apper var inte tillräckligt långt i sitt eget arbete för att detta skulle kunna utvärderas fullt ut. Dock så gick det i alla fall att tyda komponenterna i filen med ANTLRs grammatik.

Slutsatsen är att ANTLR funkar för att översätta enkla JSON och XML tabeller, men det är svårt att veta hur det kommer fungera för mer komplicerad kod. Därför så skulle det behöva fortsätta arbetas med ANTLR och testa mer avancerad funktionalitet som exempelvis If-satser och loopar. Det skulle också behöva utföras tester för att skapa annan kod än SQL för att definitivt avgöra användbarheten av ANTLR för modernisering av gamla system.

1 Inledning

Arbetet som genomförts har getts ut av företaget Apper Systems AB. Apper är ett företag vars huvudaffär är modernisering och förvaltning av applikationer. De jobbar därför mycket med äldre system och letar därför efter nya lösningar för att modernisera gamla system som är eller håller på att bli för gamla.

Många stora företag har något som kallas kärnsystem. Kärnsystem är centrala IT-system som oftast är grundläggande för ett företags verksamhet. Många av dessa har funnits i väldigt många år och håller därför på att bli föråldrade. Detta sker ofta då systemen eller personerna som underhåller systemen blir gamla, och personerna pensioneras, med följd av att det blir svårt att hitta nya yngre programmerare (se sektion 6.4 för mer information). Apper vill översätta de gamla systemen till modernare kod som exempelvis modern RPG och embedded SQL, och tittar även på både Java och Python. Apper verkar i många branscher som till exempel bank-, skogsbruks-, medicin- och bilindustrin.

Appers plan för att modernisera koden är att få ut metakod i modernare format utifrån dem gamla kärnsystemens programmeringsspråk. De gör detta med hjälp av Python och kan då få ut en stor del av systemet i en fil av något metakods format. Det enda de inte får ut är delar som ägarna av kärnsystemet ser som affärshemligheter, vilket kan göra arbetet svårare. Metakoden är kod som beskriver eller strukturerar annan kod. Denna metakod ska sedan användas för att bygga upp ny kod i ett modernare språk. Eftersom det finns flera olika format vill Apper undersöka om det går att använda ANTLR för att modernisera kod. ANTLR är ett verktyg som används för att skapa parser utifrån specificerade grammatiska regler (Se sektion 2.2 för mer information). Ett annat exempel på liknande programvaror som använts av Apper är det Python-baserade Lark som fungerar ungefär som ANTLR fast det är byggt primärt för Python istället för Java.

Detta uppdraget har därför skapats för att hjälpa till med moderniseringen av ett av dessa system, och testa om det är realistiskt att använda programvaran ANTLR för att uppnå detta.

1.1 Syfte

Syftet med det här uppdraget är att se om ANTLR kan vara till hjälp med att modernisera en äldre kodbas. Med hjälp av metakod som representerar delar av denna kodbas så ska det skapas mer modern kod och sedan ska det utvärderas om det är lämpligt att använda ANTLR för det här ändamålet. Detta ska skapa ett bättre underlag för vidare modernisering och underhållning av koden. Uppdraget fokuserar på att visa att det går att göra detta. Metakoden i projektet är från XML eller JSON, men meningen är att det ska gå i framtiden att översätta det från flera olika dataformat.

1.2 Mål

Moderniseringen ska ske med hjälp av ANTLR som ska användas för att generera SQL respektive Java-kodbas för en liten del av koden. Det stora målet är att kunna generera exekverbar kod med hjälp av det som genererats av ANTLR. Det har setts några delmål för att testa det som genererats av ANTLR.

1.2.1 Skapa en tabell från XML

Det första delmålet är att kunna skapa tabeller med olika fält med olika specifikationer för de fälten. Detta görs genom att ta in XML-kod given av Apper och generera en sträng med hjälp av ANTLR och Java-kod. Strängen ska sedan läggas in i en MariaDB databas. För att visa att det går så ska följande XML-kod översättas:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- This is an XML schema for defining database table structures -->
<DatabaseSchema>
  <!-- Table definition for "Customers" -->
  <Table name="Customers">
        <Field name="CustomerID" type="INT" constraint="PRIMARY KEY AUTO_INCREMENT"/>
        <Field name="FirstName" type="VARCHAR" length="50" constraint="NOT NULL"/>
        <Field name="LastName" type="VARCHAR" length="50" constraint="NOT NULL"/>
        <Field name="Email" type="VARCHAR" length="100" constraint="UNIQUE, NOT NULL"/>
        <Field name="Phone" type="VARCHAR" length="15"/>
        <Field name="Address" type="TEXT"/>
        <Field name="City" type="VARCHAR" length="50"/>
        <Field name="State" type="CHAR" length="2"/>
        <Field name="ZIPCode" type="VARCHAR" length="10"/>
        <Field name="CreatedDate" type="DATETIME" default="CURRENT_TIMESTAMP"/>
  </Table>

  <!-- Table definition for "Orders" -->
  <Table name="Orders">
        <Field name="OrderID" type="INT" constraint="PRIMARY KEY AUTO_INCREMENT"/>
        <Field name="CustomerID" type="INT" constraint="FOREIGN KEY REFERENCES Customers(CustomerID)"/>
        <Field name="OrderDate" type="DATETIME" default="CURRENT_TIMESTAMP"/>
        <Field name="TotalAmount" type="DECIMAL" length="10,2" constraint="NOT NULL"/>
        <Field name="OrderStatus" type="ENUM" values="Pending,Processing,Shipped,Delivered,Canceled"/>
        <Field name="ShipAddress" type="TEXT"/>
        <Field name="ShipCity" type="VARCHAR" length="50"/>
        <Field name="ShipState" type="CHAR" length="2"/>
        <Field name="ShipZIPCode" type="VARCHAR" length="10"/>
        <Field name="PaymentMethod" type="ENUM" values="CreditCard,Cash,PayPal,BankTransfer"/>
  </Table>
</DatabaseSchema>
```

Detta delmål ses som avklarat då, med hjälp av ANTLR så har XML-koden översatts till relevant SQL-syntax och lagts till i en databas. Det behöver inte läggas till någon data i tabellerna för att det ska ses som avklarat. Det ska även vara möjligt att enkelt modifiera för att lägga till fler sorters fält i lösningen. Poängen med det här delmålet är att se om ANTLR kan tolka XML korrekt och se om det är lätt att omvandla till SQL.

1.2.2 Skapa en tabell från JSON

Det andra delmålet är att generera en SQL-tabell baserad på JSON-kod istället för XML. Den ska fungera på samma sätt som XML-tabellen. Här är JSON koden som ska översättas:

```json
{
 "table": {
   "name": "FIALHT",
   "description": "Ärendelogg.ALH_Ärendelogg Huvud.Physical table",
   "fields": [
     {
       "sequence": 0,
       "name": "FI849F",
       "type": "A",
       "length": 50,
       "label": "ALH_Frågeställare",
       "column": "ALH_Frågeställare"
     },
     {
       "sequence": 1,
```

```
      "name": "RSMVF",
      "type": "A",
      "length": 50,
      "label": "ALH_Beskrivning",
      "column": "ALH_Beskrivning"
    },
    {
      "sequence": 2,
      "name": "ALHAKTIV",
      "type": "A",
      "length": 1,
      "label": "ALH_Aktiv",
      "column": "ALH_Aktiv"
    },
    {
      "sequence": 3,
      "name": "INSUSR",
      "type": "A",
      "length": 10,
      "label": "InsertUser",
      "column": "InsertUser"
    },
    {
      "sequence": 4,
      "name": "INSDT",
      "type": "L",
      "length": null,
      "label": "InsertDate",
      "column": "InsertDate"
    },
    {
      "sequence": 5,
      "name": "INSTM",
      "type": "T",
      "length": null,
      "label": "InsertTime",
      "column": "InsertTime"
    },
    {
      "sequence": 6,
      "name": "UPDUSR",
      "type": "A",
      "length": 10,
      "label": "UpdateUser",
      "column": "UpdateUser"
    },
    {
      "sequence": 7,
      "name": "UPDDT",
      "type": "L",
      "length": null,
      "label": "UpdateDate",
      "column": "UpdateDate"
    },
    {
      "sequence": 8,
      "name": "UPDTM",
      "type": "T",
      "length": null,
      "label": "UpdateTime",
      "column": "UpdateTime"
    }
  ]
 }
}
```


Värt att anmärka är att fälten har det “name” de har på grund av begränsningar med programmet de originellt skapats med, och istället för att använda de namnen så ska “label” eller “column” användas som “name” för fälten i SQL-tabellen som skapas. Detta är för att det ger ett mycket tydligare kolumnnamn då det beskriver bättre vad fältet ska innehålla. Namnet på tabellen ska inte ändras då det inte finns ett liknande fält för dem. Det finns även

andra exempel som ska översättas som fungerar på samma sätt, men har annorlunda namn på “fields” attributerna. Som tidigare ska det vara möjligt att modifiera fälten utan stora svårigheter för att lägga till mer fields-attributer. Poängen med det här delmålet är att se om ANTLR kan tolka JSON korrekt och se om det är lätt att omvandla till SQL.

1.2.3 Call stack

Det tredje delmålet var att försöka parsa och sedan översätta en call stack för en underhållning av en databastabell. För att göra detta så tilldelades en XML-fil gjord utifrån en graf som i sin tur skapats utifrån ett annat program. Här är en liten del av XML-koden:

```
<call_graph>
   <node name="*MAIN*" description="-> entry point.">
       <child>ZZINIT</child>
       <child>BAIZSF</child>
       <child>*LOOP START*</child>
   </node>
   <node name="*LOOP START*" description="">
       <child>CAEXFM</child>
       <child>ZXEXPG</child>
       <child>FBRQRL</child>
       <child>BBLDSF</child>
       <child>DAPR$$</child>
       <child>*LOOP END*</child>
   </node>
...
</call_graph>
```

Problemet med detta delmål är att Apper inte är tillräckligt långt i sitt arbete för att kunna utvärdera. Den enda delen att utvärdera var att se om det går att parsa koden med den ANTLR-genererade XML-koden. Syftet med det här delmålet är att se om ANTLR kan tolka något annorlunda XML.

1.3 Avgränsningar

Uppdraget är inte att förvandla ett gammalt kärnsystem till en ny kod, utan bara göra ett “proof of concept” för att visa att det går att ändra genom att omvandla individuella kod-bitar. Arbetet måste även utföras med hjälp av programvaran ANTLR.

Under arbetet så behövs inte alla möjliga funktionaliteter testas, utan endast de som hinns med eller går att testa. Det behövs alltså inte en applikation som kan översätta all kod från ett språk till ett annat utan endast specifika kod-delar. Det behövs inte heller skapas ett program som kan ta vilket språk som helst som input, utan olika språk delas in i olika program. Det är därför JSON och XML delmålen delas upp i olika program.

Koden som genereras med hjälp av ANTLR är Java i det här projektet, men det är möjligt att generera andra språk som till exempel Python.

2 Teknisk bakgrund

Det här kapitlet ger en överblick över några tekniska och teoretiska begrepp som är viktiga för att förstå arbetet.

2.1 Databas

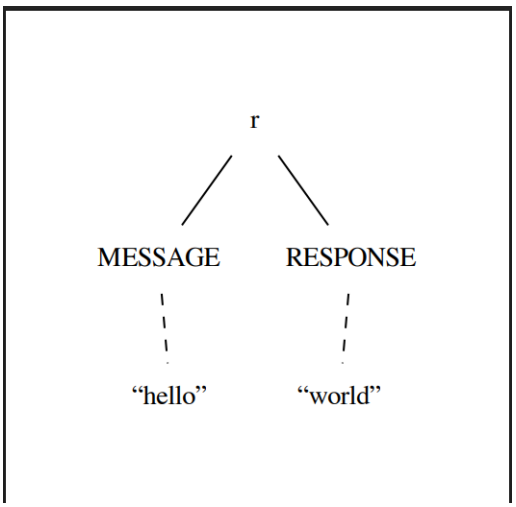
En databas är en strukturerad samling data som oftast lagras i ett databassystem. De vanligaste typerna databaser är relationsdatabaser och är också de som nämns och används under detta arbete. I relationsdatabaser delas data in i rader och kolumner i olika tabeller för att göra det enkelt att komma åt, hantera, ändra, kontrollera eller ta bort data. Relationsdatabaser använder nästan alltid programmeringsspråket SQL och databashanterare som MariaDB använder då SQL (Oracle, n.d.).

2.2 Parsers & ANTLR

En parser är något som får en textsträng som inmatning och bryter upp inmatningen i individuella komponenter, enligt en viss grammatik, exempelvis typ, objekt, värde, osv. Ofta bygger en parser ett syntaxträd utifrån som beskriver textens struktur. Att parsa är då att bryta upp något, ofta en kod, i dess baskomponenter (Karan, 2022). Många grammatiker som hanterar programmeringsspråk använder ofta något som kallas kontextfria grammatiker (CFG, context-free grammar). Den består av regler som sätter ramar för hur symbolerna i språket kan sammansättas och ett lexikon som beskriver ord och symboler i språket. Symboler i CFG kan delas in i icke-terminala och terminala. Terminala menar på symboler som representerar ord, medan icke-terminala representerar generaliseringar (Datatjej, 2019, October 7; Nystrom, 2021). I parserververktyg så hanteras ofta lexikonet av ett separat lexer och beskrivs inte i samma grammatik som reglerna. Parser är relevant för att kunna tyda metakoden som används i arbetet.

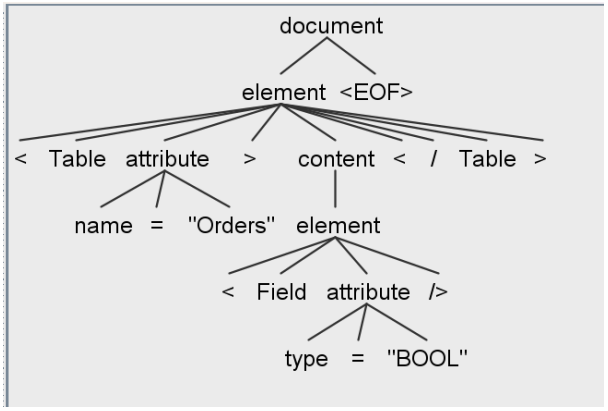
ANTLR står för “ANother Tool for Language Recognition” och är en programvara som genererar parser utifrån instruktioner i en grammatik-fil. Den här ska generera Java kod som ska hjälpa till med att översätta XML kod till annan kod utifrån instruktionerna som getts (Parr, 2012). ANTLR utgår från kontextfri grammatik (Parr & Fisher, 2011).

```
grammar Hello;

// Parser rule
r: MESSAGE RESPONSE ; // matches exactly "hello world" eller "hello friends"

// Lexer rules
MESSAGE: 'hello';
RESPONSE: 'world' | 'friends';

// Ignore whitespace
WS: [ \t\r\n]+ -> skip;
```



Det här är ett exempel på en grammatik som skapar strängar “hello world” och “hello friends”, samt ett syntaxträd för strängen “hello world”. Parser-regler matchar ihop ord av rätt kategorier: en MESSAGE följt av RESPONSE, medan lexer-regeln definierar vad som är token för MESSAGE (“hello”) och RESPONSE (“world” eller “friends”).

Följande är ett exempel på ett annat syntaxträd skapat med en Parser- och Lexer-grammatik-fil för en liten XML-kod.



Detta syntaxträd beskriver en XMLkod som består av ett element Table som innehåller ett element Field, där elementet Field är ett content för elementet Table. Table har attributet name som tilldelas namnet “Orders” och Field har attributet type som tilldelas namnet BOOL. Parser-filen används för att visa hur ett attribut och ett element ser ut, medan Lexer beskriver hur name, type, Table, Field, “BOOL” och “Orders” kan se ut (se mer om XMLLexer och XMLParser på sektion 4.1).

2.3 Andra verktyg

Java är ett objektorienterat programmeringsspråket som används för att skapa de här verktygen och är formatet på koden som skapas av ANTLR. I princip all kod som skrivs kommer att vara gjord i Java (*Vad är Java? Nybörjarguide Till Java*, n.d.).

XML är ett märkspråk som används för att utväxla data mellan informationssystem. Detta är formatet av de kodbitarna som ska genereras av ANTLR. Det kommer därför behövas grundläggande kunskap inom språket för att kunna förstå grammatiken för det samt förstå hur koden ska översättas till andra språk (*XML För Nybörjare*, n.d.).

SQL är ett programmeringsspråk som används för att hämta och modifiera data i en relationsdatabas (*What Is SQL? - Structured Query Language (SQL) Explained*, n.d.).

MariaDB är en open source-databas som liknar MySQL då den från början gjordes som en möjlig ersättning till MySQL av dess utvecklare. Den är gjord för att lagra strukturerad data och använder sig av SQL (*About MariaDB Server - MariaDB.org*, n.d.). Den används av Apper, och är det är därför dess syntax som alla SQL outputs måste passa .

Beekeeper Studio är en SQL klient som hjälper till med att visualisera databaser. Den kan användas för att enkelt kunna visualisera MariaDB databasen som används för test.

JSON är ett format som används för att representera strukturerad data som en sträng. Det används ofta i webbapplikationer då strängbaserat dataformat är enkelt att skicka över nätverk. JSON syntax är baserat på JavaScripts (*Working With JSON - Learn Web Development | MDN*, 2025).

2.4 Nyckelord

Här är några ord som används men inte förklaras i rapporten.

- Call stack: En call stack är en datastruktur i minnet för ett program, som håller koll på alla funktioner som har anropats och i vilken ordning de gjort det så att programmet vet när en viss funktion är klar och därför när den ska fortsätta.
- Visitor: Är ett designmönster som tillåter definiering av nya operationer på objekt utan att modifiera klassen (*Visitor Design Pattern*, n.d.). En Visitor-klass kan genereras av ANTLR.
- XMLToSQLVisitor: Skriven kod som använder ANTLR-genererad visitor-klass för att kolla igenom och tolka XML-kod och göra om den till SQL. Ärver från en generad Visitor-klass.
- Metakod: Kod som beskriver strukturen eller beteendet av en annan kod. I det här arbetet används JSON och XML som metakod. Språk som definierar metakod kan kallas metaspråk.
- Kärnsystem: Det centrala IT-system som hanterar en organisations data och processer.

3 Metod

Den här delen menar på hur arbetet ska genomföras för att målen ska nås samt hur kunskap och information ska samlas. Verktygen som används kommer också att förklaras.

3.1 Arbetsprocess

Arbetet genomförs med hjälp av en iterativ process. Detta betyder att delen som arbetas på ska bli klar innan nästa del av arbetet startas. Vattenfallsmodellen är en arbetsmodell som liknar det här arbetssättet. Ett Ganttschema används också för att skapa en tidsplan för arbetet. Då det är svårt att förutspå hur svårt vissa delar av arbetet kommer att vara och vad som behöver göras vid olika delar av projektet så kommer den här endast visa ungefär hur lång tid kommer spenderas på olika delar.

I praktiken betyder detta att en del tid varje vecka kommer gå till att försöka översätta en slags input med hjälp av ANTLR, och nästa kommer inte att påbörjas förrän den är klar. Under tiden så kommer även rapporten att jobbas med några gånger i veckan. Det kommer även vara ett möte med Chalmers handledare och ett med handledaren på Apper minst en gång per två veckor.

3.2 Informationsinsamling

För att bättre behärska de programvaror och programmeringsspråk som är relevanta för att genomföra arbetet så kommer en del av tiden i början av projektet gå till att testa och få studera dem. Detta görs genom att gå igenom internet genomgångar samt läsa relevant dokumentation. AI-motorer som chat GPT kommer också användas för ställa frågor om det är något oklart om projektet.

Mer teoretisk och rapport-specifik information kommer att hittas genom att undersöka internet och källor på Chalmers bibliotek.

3.3 Användning av AI

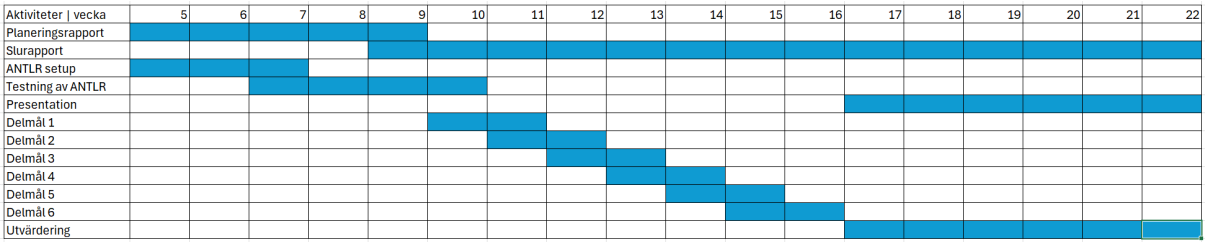
AI kommer att användas mycket i projektet, specifikt så kommer ChatGPT användas. Främst så kommer det användas för att komma igång med att använda ANTLR då projekt genomföraren inte använt det innan. Det kommer också användas för att hjälpa till med felsökning av felmeddelanden i koden.

3.4 Utvärdering

För att utvärdera projektet så kommer exempelprogram användas och gå igenom programmet. De ska testas en och en och inkludera exempel från delmålen som tidigare specificerades.

3.5 Tidsplan

Arbetet påbörjades den 2 februari 2025. Därför så påbörjas Gantt-schemat från och med vecka 5. Då ordning av vilka mål som ska avklaras är okänt då Gantt schemat skapats så kommer delmålen refereras som “delmål 1”, “delmål 2”, etc. Det läggs ungefär 2 veckor per delmål, men det är möjligt att det kommer ta längre tid. För att klargöra så förväntas inte arbetet vara klart vecka 16, utan vissa delar kan ta längre tid och det är möjligt att extra delmål kan läggas till om de som är nedskrivna avklarats. Exempelinsamling för utvärdering sker under delmålen.



Aktiviteter | vecka	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Planeringsrapport																		
Slurapport																		
ANTLR setup																		
Testning av ANTLR																		
Presentation																		
Delmål 1																		
Delmål 2																		
Delmål 3																		
Delmål 4																		
Delmål 5																		
Delmål 6																		
Utvärdering																		

4 Genomförande

4.1 ANTLR & XML

För att kunna genomföra arbetet så behövdes en ANTLR-grammatik skapas som kunde generera klasser som kan tolka XML-kod. Första försöket att hantera XML-filer med grammatik gjordes med skräddarsydda filer, men dessa saknade väldigt ofta något och fungerade inte som de skulle. Efter det så användes istället ANTLRs färdigskapta grammatiker som delade upp dig i två filer, XMLLexer och XMLParser (Parr & ANTLR Project Contributors, n.d.).

XMLLexer används för att definiera hur man känner igen ord eller tecken i XML, till exempel Text, <, >, String, etc. Medan XMLParser lägger till syntaxregler till de identifierade delarna för att bestämma strukturen av programmet och vad den faktiskt gör.
Här är några exempel på regler definierade i XMLLexer grammatiken:

```
1.  OPEN         : '<'       -> pushMode(INSIDE);
2.  TEXT: ~[<&]+; // match any 16 bit char other than < and &
3.  STRING       : '"' ~["]* '"' | '\'' ~[']* '\'';
```

Den första raden (1) är en förklaring på vad som definierar en öppning av ett element, där den använder “pushmode(INSIDE)” för att förklara att den är innanför ett element. Text (2) och String (3) använder ANTLR Lexer Rule för att definiera vad de får vara. Text får vara alla karaktärer förutom “<” och “&”, och “+” betyder att det ska finnas ett eller flera karaktärer för att räknas som en Text. String visar att allt inom två dubbla citattecken eller två enkla citattecken räknas som en String och att en String som använder dubbla inte får ha “ “ ” eller “<” i dess String medan enkla inte får ha “<” eller “ ‘ “ och “ * “ betyder att det kan vara 0 eller fler karaktärer inne en String (för mer information och exempel se sektion 2.2).

Här är några exempel på regler definierade i XMLParser grammatiken och även en rad med XML-kod som representerar ett element som innehåller flera attributer:

```
attribute
        : Name '=' STRING
        ; // Our STRING is AttValue in spec
element
        : '<' Name attribute* '>' content '<' '/' Name '>'
        | '<' Name attribute* '/>'
        ;
```

```
<Field name="TotalAmount" type="DECIMAL" length="10,2" constraint="NOT NULL"/>
```

Här är elementnamnet Field och det är ett exempel på syntaxen som definieras `'<' Name attribute* '/>'`. Där ett element börjar med “<”, sedan ett namn för elementet, efter det så är det 0 eller fler attribut och den slutar på “/>”. Attributerna är definierade som ett namn på sortens attribut och sedan en sträng med namnet på den specifika attributen. I en tabell med `name="TotalAmount"` så skulle “name” vara sorten och “TotalAmount” vara kolumnnamnet (för mer exempel på XMLParser se sektion 2.2).

När kommandot “antlr4 XML*.g4” körs så genereras Java-filer som kan kompileras till klasserna som används. Attribute har name och string så attribute context klassen får ett name och en string-metod visar attribute metoderna som ANTLR genererar. Metoden Name() ger namnet för sorten på en attribut medan metoden STRING() ger strängen som definierar namnet. I det tidigare exemplet så skulle “name”, “type”, “length” och “constraint”

ges av Name(), medan STRING() ger “TotalAmount”, etc. ElementContext fungerar på ungefär samma sätt men används för att kolla in i ett element istället för ett attribut.

4.2 XML-tabell

Första försöket för att kunna genera tabeller utifrån XML-kod gjordes genom att generera kod utifrån XMLLexer och XMLParser och sedan skapa en XMLToSQLVisitor som skulle använda de genererade Visitor-filerna för att gå igenom XML-koden och generera en string i SQL-format, men det blev olika problem, delvis med syntax och att det var svårt att få rätt information på rätt plats.

Den andra metoden som användes var att försöka använda ett Java-verktyg som är gjort för att hantera XML istället för ANTLR för att hantera datan. Verktyget som valdes var JAXB(Java Architecture for XML Binding) som gör det enkelt att benämna XML-delar som Java-variabler. Den följande kodsnutten är ett exempel från koden där den benämner XML-attributen name som Java-strängen med samma namn:

```
@XmlAttribute(name = "name")
private String name;
```

Den här koden är en del av Field-filen som användes för att gå igenom och bygga upp ett fält av en tabell. Sedan finns även en Table-fil som kollar igenom alla Field-delar i filen och skapar en lista med fält och en DatabaseSchema-fil som bygger upp en lista med Table. På det här sättet kan den gå igenom hela XML-filen och i filerna finns det getter- och setter-metoder som ser till att strängen byggs upp på ett sätt så den passar MariaDB-syntaxen. Slutgiltligt så skapades en Main-metod för att kolla igenom alla tables och Fields på följande sätt:

```
public static void main(String[] args) {
   try {
       JAXBContext context = JAXBContext.newInstance(DatabaseSchema.class);
       Unmarshaller unmarshaller = context.createUnmarshaller();

       DatabaseSchema schema = (DatabaseSchema) unmarshaller.unmarshal(new File("schema.xml"));

       for (Table table : schema.getTables()) {
           System.out.println("Create Table " + table.getName() + "(");
           List<String> output = new ArrayList<>();
           for (Field field : table.getFields()) {
               output.add(field.getName() + " " + field.getType() + field.getSQLConst());
           }
           System.out.println(String.join(",\n", output) );
           System.out.println(");");
       }
   } catch (JAXBException e) {
       e.printStackTrace();
   }
}
```

Den här metoden fungerade men då den inte använde ANTLR som en del i den så räckte den inte som en lösning. Men detta test gav en idé för hur den slutgiltiga lösningen skulle byggas upp.

Den slutgiltliga metoden som användes för att konvertera XML-tabellen till SQL var att skapa DatabaseSchema, Table och Field-filer och liknande som de var strukturerade i JAXB-försöket, men istället för att använda JAXB-exklusiva delar för att se till att rätt attribut var tilldelade rätt variabel så skapades en konstruktor för filerna som tog en ElementContext

från XMLParser som variabel. Det är så här vi vill omvandla ett Field från XML till en sträng enligt MariaDBs SQL-syntax:

```
<Field name="TotalAmount" type="DECIMAL" length="10,2" constraint="NOT NULL"/>
```

```
TotalAmount DECIMAL(10,2) NOT NULL
```

Vi vill då skapa ett Field-objekt som använder en konstruktor för att bestämma vilka attributer som används i ett Field. Det följande är konstruktorn i Field:

```
public Field(XMLParser.ElementContext ctx) {
   for (XMLParser.AttributeContext attr : ctx.attribute()) {
       String name = attr.Name().getText();
       String val = attr.STRING().getText();
       val = val.replaceAll("\"","");
       if (name.equals("name")) {
           setName(val);
       }
       if (name.equals("type")) {
           setType(val);
       }
       if (name.equals("constraint")) {
           setConstraint(val);
       }
       if (name.equals("default")) {
           setDefaultValue(val);
       }
       if (name.equals("length")) {
           setLength(val);
       }
       if (name.equals("values")) {
           setValues(val);
       }
   }
}
```

Här går man igenom alla attributen i ctx och kollar om de matchar variabel-namnen som finns med de attributen som hittas genom att kolla om de matchar strängen som man får av getText(). Om de existerar så används setter-funktioner för att deklarera de som ska användas. På “TotalAmount”-exemplet så körs anropen setName(), setType(), setConstraint() och setLength().

Sedan så finns även en metod som kallas getSQLConst() som bygger upp strängen med getter-metoderna som finns och ser till att den är uppbyggd med rätt syntax:

```
public String getSQLConst() {
   String result = "";
   if (length != null) {
       result += "("+getLength()+")";
   }
   if (constraint != null) {
       result += " " + getConstraint();
   }
   if (defaultValue != null) {
       result += " DEFAULT " + getDefaultValue();
   }
   if (values != null) {
       result += "(" + getValues() + ")";
   }
   return result;
}
```

I exempelfältet för “TotalAmount” så är det length och constraint som specificeras av getSQLConst. Standard getter-metoder används för “name” och “type” då alla möjliga fält har båda dessa. Alltså returnerar den getSQLConst() följande del: `(10,2) NOT NULL`

Konstruktorn för Table tar också en ElementContext som variabel och den har i princip samma funktion som konstruktorn i Field fast istället för att bygga upp ett Field så bygger den upp ett Table med alla Field. Detta gör den genom att skapa en lista av Field som man kan se nedanför:

```
public Table(XMLParser.ElementContext ctx){
   for (XMLParser.AttributeContext attr : ctx.attribute()) {
       String name = attr.Name().getText();
       String val = attr.STRING().getText();
       val = val.replaceAll("\"","");
       if(name.equals("name")) {
           setName(val);
       }
   }
   List<Field> fields = new ArrayList<>();
   for(XMLParser.ElementContext elem : ctx.content().element()) {
       Field field = new Field(elem);
       fields.add(field);
   }
   setFields(fields);
}
```

DatabaseSchema är den slutgiltiga klassen som behöver användas för att kunna bygga upp en SQL-tabell utifrån XML-filen. Dess konstruktor tar också ett ElementContext och gör samma sak som de tidigare fast på ännu en högre nivå, genom att identifiera alla Table i XML-filen och lägga dem i en lista. Här behövs något extra läggas till som i de tidigare klasserna, vilket kan ses nedanför:

```
public DatabaseSchema(XMLParser.ElementContext ctx) {
   List<Table> tables = new ArrayList<>();
   for(XMLParser.ElementContext elem : ctx.content().element()) {
       Table table = new Table(elem);
       tables.add(table);
   }
   setTables(tables);
}
```

Main används för att starta upp programmet och se till att strängen byggs upp korrekt och läggs in i MariaDB-databasen. Den använder ANTLR-genererade metoder och klasser för att parsa koden som visas nedan:

```
public static void main(String[] args) throws Exception {

   String xml = Files.readString(Path.of("Test.xml"));

   XMLLexer lexer = new XMLLexer(CharStreams.fromString(xml));
   CommonTokenStream tokens = new CommonTokenStream(lexer);
   XMLParser parser = new XMLParser(tokens);

   DatabaseSchema schema = new DatabaseSchema(parser.document().element());
   String url = "jdbc:mariadb://localhost:$PORT/$DBNAME";
   String user = "$DBUSERNAME";
   String password = "$DBPASSWORD";

   try (Connection conn = DriverManager.getConnection(url, user, password);
        Statement stmt = conn.createStatement()) {

       for (Table table : schema.getTables()) {
           StringBuilder createSQL = new StringBuilder();
           createSQL.append("CREATE TABLE IF NOT EXISTS ")
                   .append(table.getName())
                   .append(" (\n");

           List<String> fields = new ArrayList<>();
```

```
            for (Field field : table.getFields()) {
                fields.add("  " + field.getName() + " " + field.getType() + field.getSQLConst());
            }

            createSQL.append(String.join(",\n", fields));
            createSQL.append("\n);");

            System.out.println("Executing SQL:\n" + createSQL);
            stmt.execute(createSQL.toString());
        }

        System.out.println("Tables created successfully!");

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Efter det så tar den ElementContext av den parsade koden och tar den som variabel i ett DatabaseSchema objekt. Efter det så startas databasen. DatabaseSchema objekt används för att hitta alla Tables i det med hjälp av getTables() och så loopas de igenom. I varje loop så finns en till loop som går igenom alla fält i varje Table med hjälp av getFields() och i den loopen används getName(), getType() och getSQLConst() för att bygga upp strängen som blir dess fält och som efteråt läggs in i databasen. Det är följande del: `fields.add("  " + field.getName() + " " + field.getType() + field.getSQLConst())` som sätter upp “TotalAmount”-fältet där getName() lägger till `TotalAmount` (“name”) på rätt sätt, getType() lägger till `DECIMAL` (“type”) och getSQLConst() ser att “length” och “constraint” finns och lägger då till `(10,2) NOT NULL`.

4.3 ANTLR & JSON

För att kunna klara av nästa mål så behövdes en ANTLR-fil skapas som kan hantera JSON-format. Som med XML så fanns det en standardiserad version som skapats av ANTLR vilket enkelt kunde användas för att läsa JSON-filerna den behövde. Till skillnad från XML räckte det med en fil för att kunna generera relevanta metoder (Parr & ANTLR Project Contributors, n.d.). Några av de viktigaste delarna var hur Obj och Value hanterades.

```
obj
   : '{' pair (',' pair)* '}'
   | '{' '}'
   ;

pair
   : STRING ':' value
   ;

value
   : STRING
   | NUMBER
   | obj
   | arr
   | 'true'
   | 'false'
   | 'null'
   ;
```

Utifrån den övre delen så kan `"sequence": 0,` brytas ner genom att följa “value” och “pair”. Pair är uppbyggd med hjälp av en String som bestämmer variabelnamnet och en “value” som kan vara något av de alternativ som står över. I det här fallet så är namnet “sequence” och värdet 0, vilket är ett “NUMBER”. För att se hur “obj” funkar så kan man se hela objektet den här är en del av:

```
{
 "sequence": 0,
 "name": "FI849F",
```

```
 "type": "A",
 "length": 50,
 "label": "ALH_Frågeställare",
 "column": "ALH_Frågeställare"
},
```

Här ser vi att ett objekt är uppbyggt av flera olika “pair” inom en “{}”. Varje del i det här exemplet representerar en specificering av ett fält i en tabell.

Generering av Java-klasser fungerar på samma sätt som för XML, med enda skillnaden är att kommandot är “antlr4 JSON.g4” istället. ObjectContext fungerar på ungefär samma sätt och används på samma sätt som ElementContext medan ValueContext fungerar på ungefär samma sätt som AttrubteContext.

4.4 JSON-tabell

För att enklare skilja på XML och JSON-filerna om man vill använda dem i samma mapp så döptes Field till FieldJSON, Table till TableJSON och DatabaseSchema till DatabaseSchemaJSON. Deras struktur var ungefär samma som XML versionerna med några skillnader för att fungera för JSON-syntax istället.

FieldJSON konstruktorn ser ut så här:

```
public FieldJSON(JSONParser.ObjContext ctx) {
   Map<String, JSONParser.ValueContext> pairs = new LinkedHashMap<>();
   for (JSONParser.PairContext pair : ctx.pair()) {
       String key = stripQuotes(pair.STRING().getText());
       pairs.put(key, pair.value());
   }

   if (pairs.containsKey("label")) {
       this.name = stripQuotes(pairs.get("label").getText());
   } else if (pairs.containsKey("name")) {
       this.name = stripQuotes(pairs.get("name").getText());
   } else if (pairs.containsKey("FieldDescription")) {
       this.name = stripQuotes(pairs.get("FieldDescription").getText());
   } else if (pairs.containsKey("FieldName")) {
       this.name = stripQuotes(pairs.get("FieldName").getText());
   }

   String rawType = null;
   if (pairs.containsKey("type")) {
       rawType = stripQuotes(pairs.get("type").getText());
   } else if (pairs.containsKey("FieldType")) {
       rawType = stripQuotes(pairs.get("FieldType").getText());
   }

   Integer length = null;
   String strLength = null;
   if (pairs.containsKey("length")) {
       strLength = stripQuotes(pairs.get("length").getText());
   } else if (pairs.containsKey("FieldLength")) {
       strLength = stripQuotes(pairs.get("FieldLength").getText());
   }

   try {
       length = Integer.parseInt(strLength);
   } catch (NumberFormatException e) {
       length = null;
   }
   this.type = mapType(rawType, length);

   if (primaryKey && type.startsWith("TEXT")) {
       System.out.println("WARNING: Primary key field '" + name + "' has type TEXT. Consider using VARCHAR
```

```
instead.");
   }
   // Detect primary key
   if (pairs.containsKey("PrimaryKeySequence")) {
       this.primaryKey = true;
   }
}
```

Den är ganska mycket större än XML-versionen och detta beror på att JSON-tabellerna som ska testas i den har annorlunda namn på samma variabler. Exempelvis så är detta skillnaden mellan första Field på tabellen FIALHT (från exempelfil Ärendeloghuvud) och FIALKT (från exempelfil Ärendelogkategori):

```
{
 "sequence": 0,
 "name": "FI849F",
 "type": "A",
 "length": 50,
 "label": "ALH_Frågeställare",
 "column": "ALH_Frågeställare"
},
```

```
{
 "FieldSequence": 1,
 "FieldName": "ALKID",
 "FieldLength": 9,
 "FieldType": "P",
 "FieldDescription": "ALK_ID",
 "ColumnHeading": "ALK_ID",
 "PrimaryKeySequence": 1
},
```

Denna skillnad beror egentligen bara på att Apper hade döpt de olika, och de kan standardiseras senare i arbetet och då förenkla konstruktorn så den bara bryr sig om de relevanta namnen. Det skapade väldigt långa if-satser i vissa fall och gjorde koden lite mer komplicerad än var den annars behövt vara.

FieldJSON tar en ObjectContext som variabel vilket sen används för att skapa en Map med par för varje ValueContext, dvs värde av variabeln, och själva variabelnamnet med hjälp av en for-loop som går igenom hela ObjectContext. Exempelvis för `"sequence": 0` så är den här delen `String key = stripQuotes(pair.STRING().getText())` samma som `"sequence"` och den här delen `pair.value()` samma som `0`. En stor skillnad mellan XML-koden och JSON-koden är att vi byggde upp en lista istället för en map med XML.

Efter det så ska namnet sättas på fältet. Detta görs med hjälp av en if-sats:

```
if (pairs.containsKey("label")) {
   this.name = stripQuotes(pairs.get("label").getText());
} else if (pairs.containsKey("name")) {
   this.name = stripQuotes(pairs.get("name").getText());
} else if (pairs.containsKey("FieldDescription")) {
   this.name = stripQuotes(pairs.get("FieldDescription").getText());
} else if (pairs.containsKey("FieldName")) {
   this.name = stripQuotes(pairs.get("FieldName").getText());
}
```

Detta görs då name-variablerna som existerar är svåra att förstå och endast dessa namn på grund av en begränsning av programvaran denna JSON-tabell skapats från. Därför används hellre label (eller column, eller FieldDescription/FieldColumn, i det här fallet användes label eller FieldDescription) för SQL-tabellen som skapades. Alltså skulle den första Field se ut så här `ALH_Frågeställare VARCHAR(50)` istället för att ha namnet FI894F. I koden användes också

metoden stripQoutes() för att se till att citationstecken inte följer med när strängen byggs upp.

Typen på Field bestäms ungefär på samma sätt som name, förutom att det fanns färre alternativ för hur en typ kan heta så if-satsen är endast till för att fungera på båda sorters exempel som gavs av Apper. Efter att stripquotes() använts så läggs den in i en string som sedan matchas med en längd om det finns. Den gör det med en egenskapad metod kallad mapType() som tar en typ och en längd och beroende på vad de är så bygger den upp strängen för fältets typ med korrekt MariaDB-syntax. Till exempel, om den får typ L från givna JSON-exemplen som representerar datum så returnerar den stringen “DATE”.

Nästa del av konstruktorn hanterar Primary Key:

```
if (primaryKey && type.startsWith("TEXT")) {
   System.out.println("WARNING: Primary key field '" + name + "' has type TEXT. Consider using VARCHAR instead.");
}
// Detect primary key
if (pairs.containsKey("PrimaryKeySequence")) {
   this.primaryKey = true;
}
```

Den övre delen kollar om det finns en primaryKey i Field och om den i så fall också har typen TEXT, som en primary key inte får ha i MariaDB, så printar den ut en varning i konsolen. Den nedre kollar endast om det finns en primary key i Field och i sådana fall sätter den variabeln som hanterar den till true. Det finns även en getSQLConst()-metod i Field-klassen som fungerar på samma sätt som den som finns i XML-versionen av klassen.

```
public TableJSON(JSONParser.ObjContext ctx) {
   for (JSONParser.PairContext pair : ctx.pair()) {
       String key = stripQuotes(pair.STRING().getText());
       JSONParser.ValueContext value = pair.value();

       if (key.equals("name") || key.equals("TableName")) {
           this.name = stripQuotes(value.getText());
       } else if ((key.equals("fields") || key.equals("attributes")) && value.arr() != null) {
           for (JSONParser.ValueContext valCtx : value.arr().value()) {
               if (valCtx.obj() != null) {
                   fields.add(new FieldJSON(valCtx.obj()));
               }
           }
       }
   }
}
```

Table är väldigt mycket mindre och har egentligen bara syftet att loopa igenom alla Field och lägga in dem i en string med korrekt MariaDB-syntax. DatabaseSchema-klassen gör samma sak som Table, fast den loopar igenom alla Table i en JSON-fil.

Main-delen kör programmet genom att ta en JSON-fil och konvertera den till en sträng med MariaDB-syntax och sedan lägga in den i en MariaDB-databas.

```
public class Main {
   public static void main(String[] args) throws Exception {
       // Read JSON File
       String json = Files.readString(Path.of("../ÄrendelogKategori.json"));

       // Create Lexer & Parser
       JSONLexer lexer = new JSONLexer(CharStreams.fromString(json));
       JSONParser parser = new JSONParser(new CommonTokenStream(lexer));
```

```
        JSONParser.JsonContext tree = parser.json();

        // Extract root object
        JSONParser.ObjContext rootObj = tree.value().obj();

        // Build schema
        DatabaseSchemaJSON schema = new DatabaseSchemaJSON(rootObj);
        // DB variables
        String url = "jdbc:mariadb://localhost:$PORT/$DBNAME";
        String user = "$DBUSERNAME";
        String password = "$DBPASSWORD";


        try (Connection conn = DriverManager.getConnection(url, user, password);
             Statement stmt = conn.createStatement()) {
            // SQL output
            for (TableJSON table : schema.getTables()) {
                StringBuilder createSQL = new StringBuilder();
                createSQL.append("CREATE TABLE IF NOT EXISTS ")
                        .append(table.getName())
                        .append(" (\n");

                List<String> cols = new ArrayList<>();
                List<String> primaryKeys = new ArrayList<>();

                for (FieldJSON field : table.getFields()) {
                    cols.add("  " + field.getName() + " " + field.getType());
                    if (field.isPrimaryKey()) {
                        primaryKeys.add(field.getName());
                    }
                }

                if (!primaryKeys.isEmpty()) {
                    cols.add("  PRIMARY KEY (" + String.join(", ", primaryKeys) + ")");
                }

                createSQL.append(String.join(",\n", cols)).append("\n);");

                System.out.println(createSQL);
                stmt.execute(createSQL.toString());
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

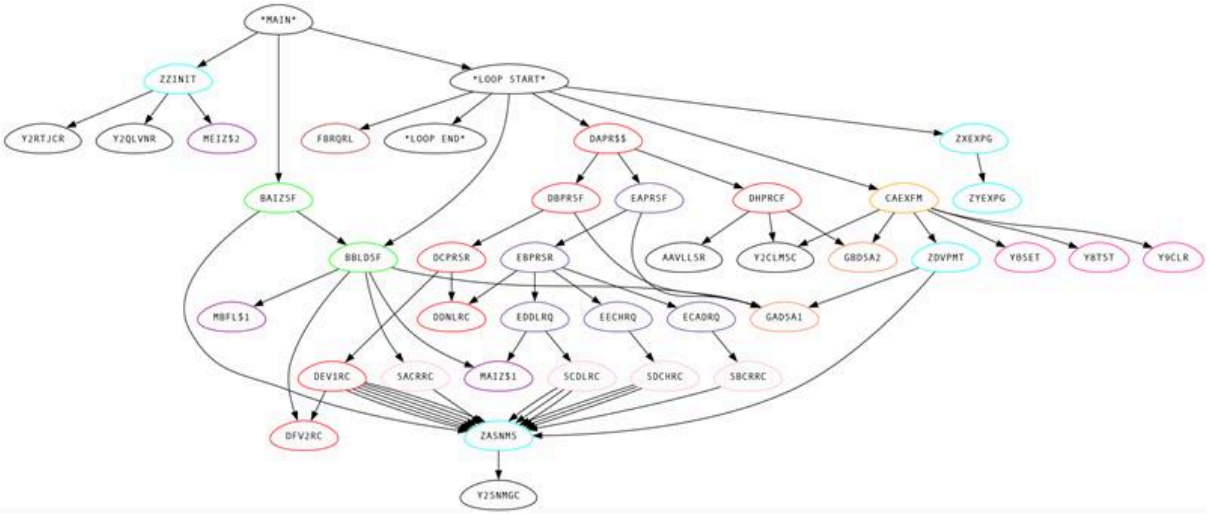
Den fungerar alltså på exakt samma sätt som XML-main-klassen, förutom att den lägger till primary key i slutet av den uppbyggda tabellen istället för i raden som definierar primary key.

Som tidigare gjordes även ett försök att översätta tabellen med JSONToSQLVisitor, vilket fungerade på ett liknande sätt som det första försöket för XML-tabellerna. Till skillnad från XMLToSQLVisitor så fungerade det här, och en sträng lyckades genereras med korrekt SQL-syntax. Men mer gjordes inte med den här då det verkade bättre att försöka använda samma struktur med Field, Table och DatabaseSchema som användes för XML. Detta var delvis för att vara konsekvent, men också för lättare kunna förstå och ändra koden när det behövs. Detta betydde att det aldrig testades om det var lätt att ändra så att namnen på Field gavs av “label” istället.

4.5 Call Stack

Nästa del var att försöka översätta en call stack vars innehåll var funktioner och subrutiner som har syftet att underhålla en databastabell. Det följande är en graf som visar hur denna

call stack ser ut:



Den här grafen hade sedan konverterats till ett XML-program som gick att parsa med den XML parser som genererats med hjälp av ANTLR.

Tyvärr är Apper inte så långt framme i processen att arbetet kan utvärderas, men för att enklare förstå hur Call stacken fungerar så gjordes en fil som gjorde en pseudokod som förtydligade hur grafen är uppsatt och hur call stacken ska fungera. Den här pseudokoden gjordes med hjälp av en skapad CallGraphVisitor som var baserad på XMLParserVisitor och en main fil som printade allting.

CallGraphVisitor är uppbyggd för att ta in XML-data och bygga upp en pseudokod som enklare visar hur den är ihopkopplad. Först skapades klassen NodeData som innehöll klassens namn, beskrivning samt en lista med alla barn.

```
public static class NodeData {
   String name;
   String description;
   List<String> children = new ArrayList<>();

   public String toString() {
       return name + " (" + description + ") -> " + children;
   }
}

Map<String, NodeData> graph = new LinkedHashMap<>();
```

Sedan användes VisitElement för att fylla en map genom att gå igenom XML-filen och sätta in alla barn i listan och beskrivningsdelen i NodeData.

```
public Void visitElement(XMLParser.ElementContext ctx) {
   String tagName = ctx.Name(0).getText();

   if ("node".equals(tagName)) {
       NodeData node = new NodeData();

       for (XMLParser.AttributeContext attr : ctx.attribute()) {
           String attrName = attr.Name().getText();
           String attrValue = stripQuotes(attr.STRING().getText());
           if ("name".equals(attrName)) node.name = attrValue;
           if ("description".equals(attrName)) node.description = attrValue;
       }

       XMLParser.ContentContext content = ctx.content();
       if (content != null) {
           for (XMLParser.ElementContext child : content.element()) {
```

```
                String childTag = child.Name(0).getText();
                if ("child".equals(childTag)) {
                    String childName = child.content().getText().trim();
                    node.children.add(childName);
                }
            }
        }

        if (node.name != null) {
            graph.put(node.name, node);
        }
    }

    return super.visitElement(ctx);
}
```

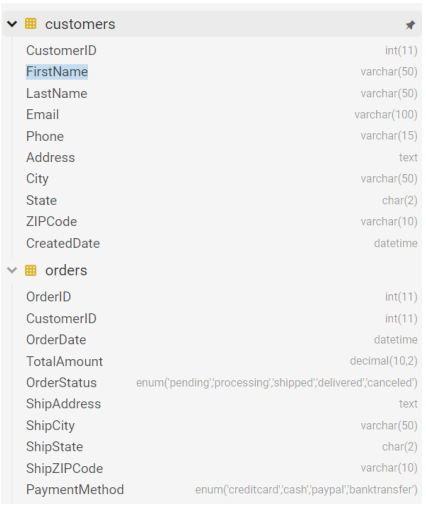
Sedan användes en Main-fill som kallade på CallGraphVisitor och printade den i terminalen. Tanken var att översätta denna till ett objektorienterat språk som Java eller liknande, men detta gick inte då Apper inte kommit tillräckligt långt i arbetet för att kunna utvärdera detta. För att utvärdera det så behöver alla subrutiner som call stacken är uppbyggd av också bli översatta till språket som call stacken skulle översättas till.

5 Resultat

5.1 XML-tabell

Enligt delmålsdefinitionen så har det första målet avklarats. Metoden som användes för att skapa en databas fungerade. Nedanför kan vi se hur strängen som läggs i MariaDB databasen ser ut samt hur tabellerna ser ut i Beekeeper studios:

```
Executing SQL:
CREATE TABLE IF NOT EXISTS Customers (
  CustomerID INT PRIMARY KEY AUTO_INCREMENT,
  FirstName VARCHAR(50) NOT NULL,
  LastName VARCHAR(50) NOT NULL,
  Email VARCHAR(100) UNIQUE NOT NULL,
  Phone VARCHAR(15),
  Address TEXT,
  City VARCHAR(50),
  State CHAR(2),
  ZIPCode VARCHAR(10),
  CreatedDate DATETIME DEFAULT CURRENT_TIMESTAMP
);
Executing SQL:
CREATE TABLE IF NOT EXISTS Orders (
  OrderID INT PRIMARY KEY AUTO_INCREMENT,
  CustomerID INT REFERENCES Customers(CustomerID),
  OrderDate DATETIME DEFAULT CURRENT_TIMESTAMP,
  TotalAmount DECIMAL(10,2) NOT NULL,
  OrderStatus ENUM('Pending','Processing','Shipped','Delivered','Canceled'),
  ShipAddress TEXT,
  ShipCity VARCHAR(50),
  ShipState CHAR(2),
  ShipZIPCode VARCHAR(10),
  PaymentMethod ENUM('CreditCard','Cash','PayPal','BankTransfer')
);
Tables created successfully!
```



Lösningen klarar av alla möjliga sorters Field som representeras i XML-filen. Det går även att lägga till flera sorters fält genom att lägga till get- och set-metoder för den nya sorten och ändra i getSQLConst() metoden.

5.2 JSON-tabell

Enligt delmålsdefinitionen för JSON-tabeller så har delmålet avklarats då man kan använda olika namn för attribut, vilket kan visas med två olika exempel. Först exemplet som visades i delmålet printar följande sträng som läggs in i MariaDB:

```
CREATE TABLE IF NOT EXISTS FIALHT (
  ALH_Frågeställare VARCHAR(50),
  ALH_Beskrivning VARCHAR(50),
  ALH_Aktiv VARCHAR(1),
  InsertUser VARCHAR(10),
  InsertDate DATE,
  InsertTime TIME,
  UpdateUser VARCHAR(10),
  UpdateDate DATE,
  UpdateTime TIME
);
```

Andra exemplet som används för att utvärdera koden är följande:

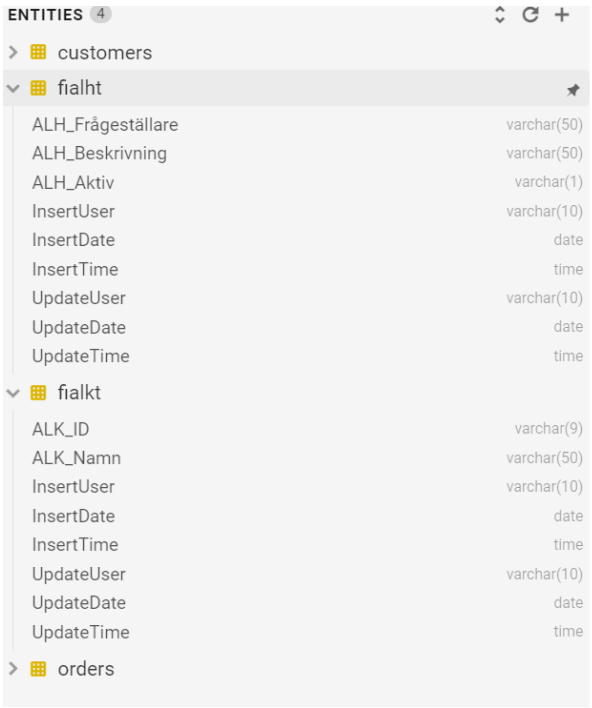
```
{
 "table": {
   "TableName": "FIALKT",
```

```
  "Description": "Ärendelogg.ALK_Ärendelogg Kategori.Physical table",
  "Module": "LAH",
  "attributes": [
    {
      "FieldSequence": 1,
      "FieldName": "ALKID",
      "FieldLength": 9,
      "FieldType": "P",
      "FieldDescription": "ALK_ID",
      "ColumnHeading": "ALK_ID",
      "PrimaryKeySequence": 1
    },
    {
      "FieldSequence": 2,
      "FieldName": "ALKNAMN",
      "FieldLength": 50,
      "FieldType": "A",
      "FieldDescription": "ALK_Namn",
      "ColumnHeading": "ALK_Namn"
    },
    {
      "FieldSequence": 3,
      "FieldName": "INSUSR",
      "FieldLength": 10,
      "FieldType": "A",
      "FieldDescription": "InsertUser",
      "ColumnHeading": "InsertUser",
      "AutoGenType": "*USER_INSERT"
    },
    {
      "FieldSequence": 4,
      "FieldName": "INSDT",
      "FieldType": "L",
      "FieldDescription": "InsertDate",
      "ColumnHeading": "InsertDate",
      "AutoGenType": "*DATE_INSERT"
    },
    {
      "FieldSequence": 5,
      "FieldName": "INSTM",
      "FieldType": "T",
      "FieldDescription": "InsertTime",
      "ColumnHeading": "InsertTime",
      "AutoGenType": "*TIME_INSERT"
    },
    {
      "FieldSequence": 6,
      "FieldName": "UPDUSR",
      "FieldLength": 10,
      "FieldType": "A",
      "FieldDescription": "UpdateUser",
      "ColumnHeading": "UpdateUser",
      "AutoGenType": "*USER_UPDATE"
    },
    {
      "FieldSequence": 7,
      "FieldName": "UPDDT",
      "FieldType": "L",
      "FieldDescription": "UpdateDate",
      "ColumnHeading": "UpdateDate",
      "AutoGenType": "*DATE_UPDATE"
    },
    {
      "FieldSequence": 8,
      "FieldName": "UPDTM",
      "FieldType": "T",
      "FieldDescription": "UpdateTime",
      "ColumnHeading": "UpdateTime",
      "AutoGenType": "*TIME_UPDATE"
    }
  ]
 }
}
```


Som får följande sträng ner Main kör med den:

```
CREATE TABLE IF NOT EXISTS FIALKT (
  ALK_ID VARCHAR(9),
  ALK_Namn VARCHAR(50),
  InsertUser VARCHAR(10),
  InsertDate DATE,
  InsertTime TIME,
  UpdateUser VARCHAR(10),
  UpdateDate DATE,
  UpdateTime TIME,
  PRIMARY KEY (ALK_ID)
);
```

Båda dessa har korrekt syntax och går att lägga in i databasen utan problem. Vi kan se både tabellerna i databasen genom beekeeper här:



Eftersom den använder samma struktur som XML så är det också enkelt att lägga till mer möjliga sorters fält ifall det skulle behövas.

5.3 Call stack

Detta genomfördes för att testa om grafen som byggs upp i filen läses in korrekt genom att köra ett program som ska koppla varje “Node” till dess “children” och printa ut det.
Följande är en liten del av hur resultatet såg ut.

MAIN (-> entry point.) -> [ZZINIT, BAIZSF, *LOOP START*]
LOOP START () -> [CAEXFM, ZXEXPG, FBRQRL, BBLDSF, DAPR$$, *LOOP END*]
BBLDSF (* Load subfile page (write empty page if add mode)) -> [MAIZ$1, MBFL$1, DFV2RC, SACRRC, GADSA1]
BAIZSF (* Initialise & load subfile page) -> [BBLDSF, ZASNMS]
…

Apper såg detta som tillräckligt avklarat då det gick att läsa av datan och översätta den till en datastruktur. Eftersom de inte kunde utvärdera något mer så försökte inte en översättning av koden till Java göras. Detta visade ändå att det gick att använda ANTLR genererade kod för att tyda call stacken korrekt.

6 Diskussion

6.1 ANTLR

När Apper föreslog den här uppgiften så var det för att de ville testa om de kunde använda ANTLR i sitt arbete. Om deras enda syfte skulle vara att använda XML-filer för att generera Java-filer så skulle något som JAXB fungera mycket bättre, men med tanke på att flera olika format kan användas som input så fungerar det inte. I detta arbete har också endast Java-filer genererats, men andra programmeringsspråk som Python används också av Apper så JAXB som endast fungerar till Java skulle därför inte fungera. Det finns andra ekvivalenta program till ANTLR som kan hantera flera olika programmeringsspråk, men Apper ville att detta skulle undersökas i den här uppgiften. Till exempel har de själva jobbat med Lark medan jag testat ANTLR.

JSON och XML användes för att de är några av de vanligaste sätten att representera data, och det var då enklare för Apper att ge exempel för utvärdering av delmålen. Att de här fungerade relativt enkelt tyder bra då väldigt mycket metakod görs i något av dem.

Utifrån resultaten så kan vi se att det finns flera sätt att använda ANTLR för att bygga upp en SQL-tabell med metakod. För XML så undersöktes 1 sätt och för JSON så hittades 2, då användning av genererade visitor även fungerade. Troligtvis så skulle visitor-metoden även fungera med XML, men lyckades inte för att den gjordes i början av arbetet innan jag var van vid ANTLR. Detta är bra då det betyder att det finns flera sätt att använda ANTLR för att tyda och sedan använda information från en kod med ANTLR. Det var även enkelt att gå igenom och parsa Call stack XML-filen trots att den var något mer komplicerad vilket även är bra.

Det är svårt att avgöra om det går att använda ANTLR för mer komplicerade saker, som if-satser, loopar, etc. För att veta det bör man fortsätta med arbetet och testa dessa funktioner. Men troligtvis går det i alla fall att tyda vilken XML- eller JSON-fil som helst med de grammatik-filer som jag använt.

6.2 Vidareutveckling av nuvarande kod

Det finns flera möjligheter att förbättra det nuvarande arbetet. En bit som XML klarar, men inte JSON är att en JSON-fil kan endast ha en tabell i sig, medan XML kan ha flera. Detta bör vara en ganska enkel lösning, men sågs som onödig för att utvärderingen av JSON skulle klaras. Man skulle också kunna göra möjligheten att man loopar igenom flera olika filer i Main-filen istället för att man lägger alla tabellerna i en.

En annan förbättring man kan göra är att man uppdaterar programmet så att den identifierar om man kör en JSON, XML eller annan fil och från det använder den relevanta Field-, Table- och DatabaseSchema-klasserna.

Som förbättring skulle man kunna undersöka om man kan göra JSON- och XML-klasserna mer lika, kanske till och med göra en generell klass för Field, Table och DatabaseSchema och sedan lägga in relevant ANTLR genererad kod beroende på vilket språk den ska använda. Detta kan även hjälpa integrationen av andra format på metakoden.

Om man vill ha ett mer interaktivt program så skulle man kunna bygga upp det så man får en “pop-up” och lägger in en fil samt väljer vilket format den skall va i. Men detta är ganska långt utanför uppgiftens omfattning.

6.3 Framtida arbete

Från början så var tanken med det här arbetet att undersöka många fler olika sorters funktionaliteter än bara skapa tabeller. Några exempel på möjliga funktionaliteter var if-satser, loopar och input/output. Detta blev inte av på grund av tidsbrist, och för att det var svårt för Apper att hitta kodstycken som skulle kunna utvärderas av mig. I framtiden så är det något som kan vara viktigt att utforska innan man bestämmer sig för att använda ANTLR i större delar av arbetet.

En annan möjlig förbättring skulle vara att testa med andra format som input istället för JSON och XML, samt använda något annat som output som PostgreSQL. Detta beror mest på hur många format som Apper har metakod. Om nästan alla kod är i JSON eller XML så är det nog viktigare att fokusera på att undersöka alla möjliga funktionaliteter.

Som tidigare sagt så gick det inte att utvärdera call stacken då Apper inte kommit tillräckligt långt i sitt arbete, detta var för att de inte hade kommit tillräckligt långt i arbetet. Det de skulle behöva bli klara med för att jag skulle kunna utvärdera denna är de subrutiner som call stacken håller reda på. Det skulle kanske vara möjligt för mig att omvandla alla subrutiner, men detta skulle vara väldigt svårt och ta väldigt lång tid, så antagligen bör det genomföras av flera personer med mer erfarenhet av ANTLR eller liknande program som Lark. Om detta skulle ha gjorts så hade jag försökt omvandla call stacken till Java-kod med alla subrutiner.

6.4 Hållbar utveckling

Många äldre system kan innehålla väldigt mycket information som är viktiga för olika verksamheter och deras kunder. Om de systemen är gjorda med en äldre programvara eller på ett sätt som inte lärs ut längre så finns det en stor risk att framtidens programmerare inte har kunskapen för att kunna underhålla och uppdatera dessa system. Detta kan skapa problem exempelvis om ny logik läggs till eller om systemet inte gjordes för att hantera så mycket information som det behöver i framtiden. Om verksamheten är samhällsviktig som exempelvis ett sjukhus eller en bank så kan det även skapa problem om viktig information går förlorad.

En kodbas byggs ofta upp över lång tid och av många olika programmerare så det finns en stor risk att det inte finns någon som förstår hela logiken bakom koden. Detta gör att om man skapar ett helt nytt system så är det en stor risk att man missar någon funktionalitet som man inte skulle missat om man moderniserade det direkt med hjälp av den äldre koden.

7 Slutsats

Arbetet kan ses som lyckat då syftet var främst att undersöka om det gick att använda generad ANTLR kod för att skapa exekverbar kod vilket avklarades. Om det skulle funnits mer tid för arbetet, eller om genomföraren var mer erfaren med att använda programmen och språken som använts så skulle det nog varit möjligt att ta det längre och gå igenom mer exempel än bara databastabeller. Det finns även mycket man kan göra för att förbättra det som redan gjorts, samt att arbeta vidare. Om det sker en fortsättning av arbetet så skulle det nog funka mycket bättre om den som gör det har tillgång till koden den ska översätta, då det inte blir lika många pauser i arbetet, och mer grejer kan undersökas lättare.

Avslutningsvis så har arbetet genomförts till en tillfredsställande nivå, men det skulle helst ingå fler element i det för att få en bättre överblick för hur bra det funkar att använda ANTLR för det här syftet.

Referenser

1. (n.d.). Introduction to Beekeeper Studio. Retrieved May 22, 2025, from https://docs.beekeeperstudio.io/
2. *About MariaDB Server - MariaDB.org*. (n.d.). MariaDB Foundation. Retrieved May 22, 2025, from https://mariadb.org/about/
3. Datatjej. (2019, October 7). *Generativ grammatik och Chomsky-hierarki*. https://datatjej.github.io/Generativ-grammatik-och-Chomsky-hierarki/
4. Karan, R. (2022, July 7). *What is a Parser? Definition, Types and Examples*. TechTarget. Retrieved May 27, 2025, from https://www.techtarget.com/searchapparchitecture/definition/parser
5. Oracle. (n.d.). *What is a database?* Oracle. https://www.oracle.com/se/database/what-is-database/
6. Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning.
7. Parr, T. (2012). *Definitive ANTLR 4 Reference*. Pragmatic Bookshelf.
8. Parr, T., & ANTLR Project Contributors. (n.d.). *JSON.g4* [ANTLR grammar file]. GitHub. https://github.com/antlr/grammars-v4/blob/master/json/JSON.g4
9. Parr, T., & ANTLR Project Contributors. (n.d.). *XML grammar for ANTLR v4* [Source code]. GitHub. https://github.com/antlr/grammars-v4/tree/master/xml
10. Parr, T., & Fisher, K. (2011). *LL()*: The Foundation of the ANTLR Parser Generator*. *ACM SIGPLAN Notices*, 46(6), 425–436. https://www.antlr.org/papers/LL-star-PLDI11.pdf
11. *Vad är Java? Nybörjarguide till Java*. (n.d.). Microsoft Azure. Retrieved May 22, 2025, from https://azure.microsoft.com/sv-se/resources/cloud-computing-dictionary/what-is-java-programming-language
12. *Visitor Design Pattern*. (n.d.). Refactoring.Guru. Retrieved June 16, 2025, from https://refactoring.guru/design-patterns/visitor
13. *What is SQL? - Structured Query Language (SQL) Explained*. (n.d.). AWS. Retrieved May 22, 2025, from https://aws.amazon.com/what-is/sql/
14. *Working with JSON - Learn web development | MDN*. (2025, April 28). MDN Web Docs. Retrieved May 22, 2025, from https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/JSON

15. *XML för nybörjare*. (n.d.). Microsoft Support. Retrieved May 22, 2025, from https://support.microsoft.com/sv-se/office/xml-f%C3%B6r-nyb%C3%B6rjare-a87d234d-4c2e-4409-9cbc-45e4eb857d44

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2025
www.chalmers.se

CHALMERS