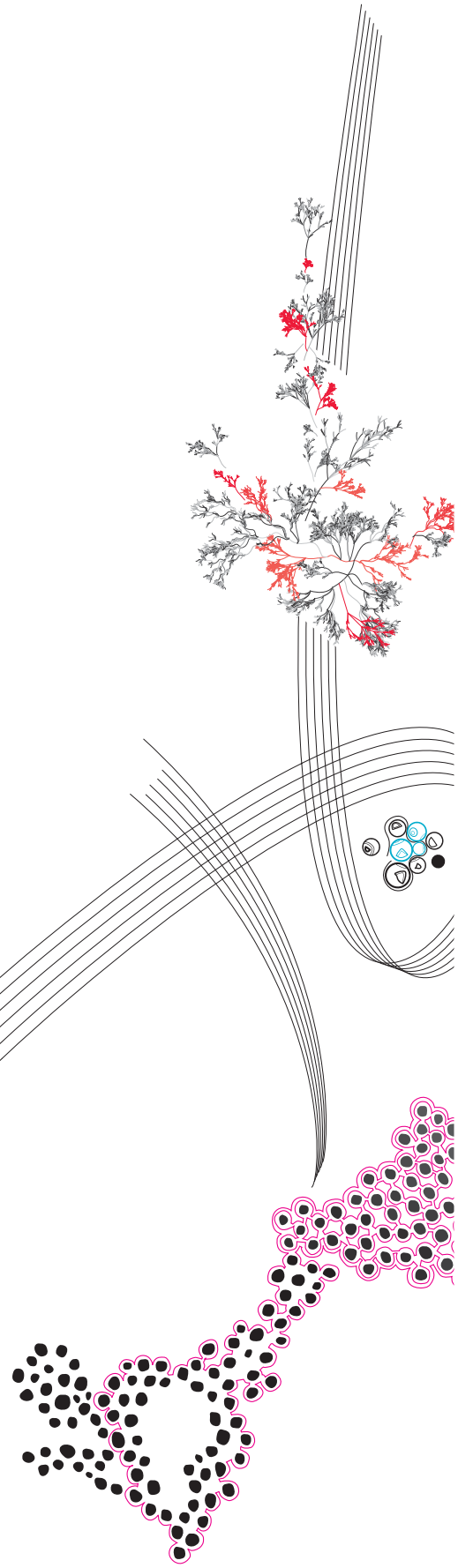

MSc Computer Science
Final Project

Towards Grammatical Inference of Legacy Programming Languages

Ömer Sayilir

Supervisor: Dr. Vadim Zaytsev

May , 2024

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

Grammatical inference is a technique that can help with the (re)implementation of language tools for abandoned or nearly abandoned languages. This thesis delves into the grammatical inference of fourth generation languages (4GLs). Languages in this language class were widely popular and hyped throughout the 1980s and were used to build parts of business-critical systems that organisations still rely on today. Methods of obtaining positive and negative language samples required for the inference process have been laid out, and a prototype genetic algorithm for grammatical inference of legacy languages has been implemented.

Keywords: grammatical inference, fourth-generation languages, software evolution, evolutionary algorithms

Chapter 1

Introduction

During the latter half of the twentieth century, many business-critical software systems of large organisations from the public and private were written. Around 22%–32% [72] of these legacy systems that are in use today were, partially or completely, developed using fourth generation languages (4GLs). Languages from this language class were popular throughout the 1980s and are on a higher level of abstraction than general-purpose programming languages (also referred to as general-purpose programming languages, GPLs, third generation languages or 3GLs). 4GLs were intended to be easier to use alternative to 3GL with often a wider audience than just software engineers [96, 97].

Many of the promises of 4GLs did not stand the test of time. Recent criticisms of 4GLs include them being regarded as badly designed domain-specific languages [166] and their vendor support getting more expensive and minimal as time goes on [171]. Because of factors such as the high maintenance costs and an ever-shrinking pool of developers who know how to use these languages, organisations relying on 4GLs look for solutions in the form of migration from or even the reimplementation of 4GLs in a way that is compatible with modern hardware and software. Both of these options require a considerable amount of resources and manpower to realise. The usage of (semi-)automatic tools could accelerate these processes [32, 167].

This thesis revolves around the inference of grammars of fourth generation languages. Grammatical inference is the process of synthesising a grammar for a language with the help of positive-, and often also negative language samples. The ideal inferred grammar recognises all positive samples and rejects all negative ones. Using grammatical inference on legacy 4GLs can reduce the overall time required to create parsers, and by extension, compilers or other language tools for these languages.

The following contributions are made in this thesis:

- The syntactical analysis and categorisation of a large number of fourth generation languages.
- Structured methodology to generate the required input datasets for use in grammatical inference solutions, given a language's grammar. These generated sets allow for the development and validation of new grammatical inference solutions.
- A prototype solution for grammatical inference of fourth generation languages based on a genetic algorithm approach.

The document is structured as follows; in chapter 2, language generations are explained and examples of second-, third- and fourth generation languages are covered. The chapter concludes with a matrix that syntactically categorises the languages that were encountered based on their main inspiration and their syntactic rigidity, and with an overview of selected languages that will be used in the rest of the project. In chapter 3, the formal definition of context-free grammars and the theoretical background of language inference are covered. Besides the theory, the chapter goes over practical examples of grammatical inference by highlighting case studies where the grammars of one or more languages are inferred. In chapter 4, the methods of obtaining positive and negative samples by means of gathering from existing sources and generation will be discussed. The implementation of a prototype genetic algorithm for grammatical inference will be discussed in chapter 5. Finally, in chapter 6 of the thesis the results of the prototype algorithm are analysed, the research questions are answered and future work is discussed.

1.1 Research Questions

The research conducted during this project will answer the following three research questions and their sub-questions. Research questions 1 and 2 are about the acquisition of samples for the inference process, these questions are largely answered in chapter 4. The third research question is about the development of a prototype genetic algorithm for grammatical inference and is addressed in chapter 5.

1. ***In what way could positive samples of a language be sourced for inferring 4GLs?***
   (a) ***How reliable are public repositories for acquiring positive samples?***
   (b) ***How can positive samples of a programming language be generated?***
2. ***In what way could negative samples of a language be sourced for inferring 4GLs?***
   (a) ***How effective is using samples from a syntactically different language from the same language family?***
   (b) ***How can negative samples of a programming language be generated?***
3. ***How can an evolutionary algorithm be utilized to infer grammars of legacy languages by using positive and negative samples of these languages?***
   (a) ***What evolutionary algorithms could be used to solve the inference problem and what are their pros and cons?***
   (b) ***What objectives should the fitness function of the algorithm consider, and how should the objectives be weighed?***
   (c) ***How should the evolutionary operators be defined in the algorithm?***

Chapter 2

Legacy Languages and Their Characteristics

Legacy software languages can be split up into four categories [37, 72, 96, 97]:

- first generation languages (1GL): binary instructions
- second generation languages (2GL): mainframe assembly languages
- third generation languages (3GL): higher level machine-independent languages
- fourth generation languages (4GL): often non-procedural, end-user oriented languages

Most legacy languages were originally part of a mainframe ecosystem. Due to the rising maintenance costs of these mainframes, companies are looking to port their systems to more modern and readily available hardware [168]. The growing demand for this modernisation is why these languages are considered "legacy" languages in the present. This part of the document will cover examples of languages from the 2GL, 3GL and 4GL categories. In section 2.1 and section 2.2 popular legacy 2GLs and 3GLs found in the TIOBE Index [148] are covered. In section 2.3 4GLs mentioned in the second and third volume of the book series "Fourth-generation languages" by James Martin and Joe Leben [96, 97] and 4GLs encountered through other sources will be covered. The last subsection will categorise languages from all three categories based on their syntactical similarities and structural rigidity.

2.1 Legacy Second Generation Languages

Second Generation Languages usually refer to assembly languages. These languages are called "second generation" since they are at a higher level of abstraction than plain binary instructions (also referred to as "first generation languages"). These languages are machine- or processor-specific since the available instructions can vary per environment [111].

A prominent legacy assembly language is the High Level Assembler (also known as Assembler or HLASM), a 2GL for IBM mainframes. Listing 2.1 shows a part of a program written in HLASM [61]:

```
 NOTREL   AIF   ('&TYPE' NE 'ABS').ERR1                              @H1A 00220000
          SLR   0,0                     INDICATES NOTE MACRO       @H1A 00230000
          LA    15,32                   ROUTER CODE                @H1A 00240000
          SVC   109                     SUPERVISOR CALL            @H1A 00250000
          MEXIT                                                    @H1A 00260000
.ERR1     MNOTE 8,'INVALID PARAMETER FOR TYPE'              @H1A       00270000
```

LISTING 2.1: A Snippet of a HLASM Program

HLASM is a rigid, position-based language. Table 2.1 illustrates how a typical line of code is structured in HLASM [60].

Type	Positions
Statement Field	1-71
Continuation-Indicator Field	72
Identification-Sequence Field	73-80

TABLE 2.1: The Structure of a HLASM Line

2.2 Legacy Third Generation Languages

Third Generation Languages are at the same level of abstraction as the general-purpose programming languages that are widely in use today [111]. This subsection will cover the most prominent 3GLs that exist in the legacy domain [72, 148].

2.2.1 COBOL

COBOL was initially created by the "short-range committee", one of three committees put together by the United States Department of Defence to create a common business programming language [122]. Over the years COBOL has been widely adopted, with many vendors creating their own compilers. This wide adoption is also the cause for there being many, slightly different, dialects of COBOL [81].

IBM COBOL is one of the major dialects that exists for COBOL today. Listing 2.2 shows a snippet of a program written in this dialect [64].

```
      ***************************************************************
      **           I D          D I V I S I O N                     ***
      ***************************************************************
       Identification Division.
       Program-id.    AWIXMP.
      ***************************************************************
```

```
      **           D A T A      D I V I S I O N                     ***
      ****************************************************************
       Data Division.
       Working-Storage Section.
      ****************************************************************
      **  Declarations for the local date/time service.
      ****************************************************************
       01   Feedback.
       COPY CEEIGZCT
        02   Fb-severity      PIC 9(4) Binary.
        02   Fb-detail        PIC X(10).
       77   Dest-output       PIC S9(9) Binary.
       77   Lildate           PIC S9(9) Binary.
       77   Lilsecs           COMP-2.
       77   Greg              PIC X(17).
      ****************************************************************
      **  Declarations for messages and pattern for date formatting.
      ****************************************************************
       01   Pattern.
        02                    PIC 9(4) Binary Value 45.
        02                    PIC X(45) Value
            "Today is Wwwwwwwwwwz, Mmmmmmmmmz ZD, YYYY.".

       77   Start-Msg         PIC X(80) Value
            "Callable Service example starting.".

       77   Ending-Msg        PIC X(80) Value
            "Callable Service example ending.".

       01 Msg.
         02 Stringlen         PIC S9(4) Binary.
         02 Str               .
          03                  PIC X Occurs 1 to 80 times
                                 Depending on Stringlen.
      ****************************************************************
      **           P R O C      D I V I S I O N                    ***
      ****************************************************************
       Procedure Division.
       000-Main-Logic.
           Perform 100-Say-Hello.
           Perform 200-Get-Date.
           Perform 300-Say-Goodbye.
           Stop Run.
      **
      ** Setup initial values and say we are starting.
      **
       100-Say-Hello.
           Move 80 to Stringlen.
           Move 02 to Dest-output.
           Move Start-Msg to Str.
           CALL "CEEMOUT" Using Msg   Dest-output Feedback.
           Move Spaces to Str.        CALL "CEEMOUT" Using Msg Dest-output Feedback
    .
      **
      ** Get the local date and time and display it.
      **
       200-Get-Date.
           CALL "CEELOCT" Using Lildate Lilsecs     Greg      Feedback.
           CALL "CEEDATE" Using Lildate Pattern     Str       Feedback.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
           Move Spaces to Str.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
      **
      ** Say Goodbye.
      **
       300-Say-Goodbye.
           Move Ending-Msg to Str.
           CALL "CEEMOUT" Using Msg     Dest-output Feedback.
       End program AWIXMP.
```

LISTING 2.2: A Sample COBOL Program

A COBOL file is split into three divisions:

- The `Identification Division`: contains identifying information for the file.
- The `Data Division`: contains the definition of the data used in the file.
- The `Procedure Division`: contains the procedures defined in the file.

Looking at the language reference [67] and the example in the listing above it can be observed that COBOL is a sentence-based language with some position-based elements. Table 2.2 illustrates the structure of a line of code written in COBOL.

Type	**Positions**	**Description**
Sequence number area	1-6	Used to label the source line, can be any character.
Continuation-Indicator Field	7	Indicator area, used to indicate line continuation from a previous line, a comment line or a debugging line.
Area A	8-11	Used for division headers, section headers level, paragraph headers, paragraph names, level indicators, level numbers, `DECLARATIVES`, `END DECLARATIVES` and `END` markers.
Area B	11-72	Entries, sentences, statements, clauses and continuation lines.

TABLE 2.2: The Structure of a COBOL Line

2.2.2 PL/I

PL/I is a programming language developed by IBM for use on their mainframe systems. It was originally developed in the 1960s with its purpose being a programming language which could be used by both commercial and scientific users, who, prior to the existence of PL/I, would use languages such as COBOL and FORTRAN respectively [110]. PL/I is maintained to this day by IBM.
Listing 2.3 shows a code snippet written in PL/I from the language reference [71]:

```
    X: proc options(main);
    dcl (A,B) char(1) init('1');
    call Y;
    return;

    Y: proc;
        dcl 1 C,
            3 A char(1) init('2');
        put data(A,B);
        return;
    end Y;
end X;
```

LISTING 2.3: A Sample PL/I Program

As seen in Listing 2.3, PL/I procedures are built using `PROC` or `PROCEDURE` statements. These statements can contain sub-procedures which can be called in the main procedure. Data can be declared as single variables as shown in procedure `X` and as "major structures" as shown in procedure `Y`. PL/I code can be written in a mix of uppercase and lowercase notation. Furthermore, keywords in PL/I are not reserved, a programmer could use the same name as a keyword for a variable name. From the language reference [71] it can be concluded that PL/I does not have strict position-based restrictions, it is a procedural language and is built up of blocks which contain statements.

2.2.3 FORTRAN

FORTRAN is a programming language first developed in the 1950s by a team led by John Backus at IBM. The main motivation behind the creation of FORTRAN was to create a programming language that was suitable for scientific computing which is easier to work with than using an assembly language [13]. FORTRAN has seen many revisions throughout the years with vendors other than IBM, such as Intel [73] and Oracle [104], making their own implementations of the language.
FORTRAN code can be written in the "fixed source form" or in the "free source form". Listing 2.4 and Listing 2.5 illustrate what the two forms of FORTRAN code would look like:

```
!IBM* SOURCEFORM (FIXED)
      CHARACTER CHARSTR ; LOGICAL X          ! 2 statements on 1 line
      DO 10 I=1,10
        PRINT *,'this is the index',I  ! with an inline comment
10    CONTINUE
C
       CHARSTR="THIS IS A CONTINUED
     X CHARACTER STRING"
       ! There will be 38 blanks in the string between "CONTINUED"
       ! and "CHARACTER". You cannot have an inline comment on
       ! the initial line because it would be interpreted as part
       ! of CHARSTR (character context).
  100 PRINT *, IERROR
! The following debug lines are compiled as source lines if
! you use -qdlines
D     IF (I.EQ.IDEBUG.AND.
D    +    J.EQ.IDEBUG)     WRITE(6,*) IERROR
D     IF (I.EQ.
D    +  IDEBUG )
D    +  WRITE(6,*) INFO
       END
```

LISTING 2.4: A Sample FORTRAN Program in Fixed Source Form

```
!IBM* SOURCEFORM (FREE(F90))
!
! Column Numbers:
!        1         2         3         4         5         6         7
!234567890123456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,20
   PRINT *,'this statement&
   & is continued' ; IF (I.LT.5) PRINT *, I

 ENDDO
 EN&
         &D              ! A lexical token can be continued
```

LISTING 2.5: A Sample FORTRAN Program in Free Source Form

As can be seen from the listings, while the two source forms look similar, they have different levels of strictness. As an example, the fixed source form has position-based rules for declaring comments and debugging lines. A notable property of FORTRAN's free source form is white-space insignificance. As originally illustrated by Zaytsev [169], the feature that was mainly meant to improve the readability of code by, for example, allowing for spaces in variable names, can lead to situations such as the one shown in Listing 2.6 and Listing 2.7:

```
    DOI=1,10
```

LISTING 2.6: FORTRAN Code That Iterates the Next Statement 10 Times

```
    DOI=1.10
```

LISTING 2.7: FORTRAN Code That Assigns 1.10 to a Variable DOI

2.3 Legacy Fourth Generation Languages

4GLs are a class of programming languages that are on an even higher level of abstraction than 3GLs. The majority of the 4GLs covered in this section are described in "*Volume II, representative 4GLs*" and "*Volume III, 4GLs from IBM*" of the "*Fourth generation Languages*" books by James Martin and Joe Leben [96, 97].

The books describe 4GLs as high-productivity languages which allow users to define applications with means other than just textual sequential statements such as screens and menus. The "high-productivity" characteristic of 4GLs is described as needing less code for the same functionality compared to 3GLs such as COBOL or PL/I.

The 4GLs that do have a textual component are split into two categories; procedural and non-procedural. In the context of the books, procedural means that the code instructs the computer to perform a task by executing steps in a specific order and non-procedural means that the code describes the desired end result and the programmer does not specify how this result should be obtained. Non-procedural programming can be considered a form of "constraint programming" as described by Van Roy [152].

Many of the promises of 4GLs did not stand the test of time. In fact, some would even consider 4GLs to be badly designed domain-specific languages (DSLs) [166]. Some argue that 4GLs can be viewed as technical debt since with time, the rightsholders offer less support and charge more fees [171].

As mentioned before, 4GLs allow users to specify a program using their facilities, this can take many forms such as text-based or screen/dialog-based format or a combination of the two. This specification is then translated into something the mainframe can execute. An example of such a translation could be the translation from 4GL code to 3GL code which is in turn compiled by the compiler present on the mainframe into executable machine code.

4GLs can greatly vary in their intended purpose as will be illustrated later in the chapter.

2.3.1 CICS

CICS (*Customer Information Control System*) is a 4GL from IBM that acts as middleware between z/OS and applications on IBM mainframes [63]. The main functionality of CICS includes services for transaction management, data management, communication, and application development [68]. Many of the 4GLs in this section [96, 97], 3GLs in section 2.2 [69, 70], and HLASM [90] (section 2.1) utilise CICS in one way or another. CICS has been actively maintained throughout the years and newer versions of CICS allow for the implementation of more modern functionality such as RESTful web services. The general structure of a CICS command is shown in Listing 2.8 and a concrete example is shown in Listing 2.9.

```
EXEC CICS command option(arg)....
```

LISTING 2.8: The General Stucture of a CICS Command [65]

```
EXEC CICS READ
          FILE('FILEA')
          INTO(FILEA)
          RIDFLD(KEYNUM)
          UPDATE
```

LISTING 2.9: A Concrete Example of a CICS Command [65]

2.3.2 ADS/ONLINE

ADS/ONLINE (also referred to as ADS/O, ADSO or ADS) is a 4GL originally developed by Cullinet Software and is currently owned by Broadcom [18]. It is an extensive 4GL that provides menu-driven interactive non-procedural methods for application development and a procedural very high-level COBOL-like programming language. Dialogs created with ADS/ONLINE are defined by the 4 components in Table 2.3.

Component	Description
Maps	Format of a dialog screen
Processes	Processes written in ADS/ONLINE code that are associated with the dialog
Subschemas	View of the database
Record definitions	Describe the data accessed by the dialog, this data description describes both the structure of data in the database and the structure of local data

TABLE 2.3: The Components of ADS/ONLINE

A snippet of ADS/ONLINE code can be seen in Listing 2.10.

```
OBTAIN OWNER WITHIN DEPT-EMPLOYEE.
IF DB-ANY-ERROR THEN
    DISPLAY MESSAGE TEXT 'COULD NOT FIND DEPARTMENT'.
    OBTAIN OWNER WITHIN OFFICE-EMPLOYEE.

IF DB-ANY-ERROR THEN
    DISPLAY MESSAGE TEXT 'COULD NOT FIND OFFICE'.

EXECUTE NEXT FUNCTION.
```

LISTING 2.10: A Snippet of ADS/ONLINE Code

The first thing that comes to mind when looking at this process code is that, structurally, It has a very strong resemblance to code found in the procedure division of a COBOL source file. The statements in this piece of code are structured like sentences like in COBOL where each statement is ended with a period. The main difference here is that the level of abstraction is a bit higher, while statements like `OBTAIN` do exist in COBOL, the syntax here is more simplified than the `OBTAIN` statement in COBOL where it would require the programmer to be more specific to retrieve the desired data from the database.

2.3.3 APPLICATION FACTORY

The earliest accounts [145] of this system can be traced to a utility called INFORM, developed by Standard Information Systems (SIS) in 1972, ported to PDP-11 as INFORM-11 in 1977 with Cortex Corporation created soon afterwards to market this product. With repository handling added in 1984, it was renamed APPLICATION FACTORY, with its PROCESS language simultaneously re-baptised as BUILDER. In 1986 the name "APPLICATION FACTORY" was dropped in favour of CorVision, which saw its last release (V5.11) in 1994. In 2005–2015 there have been commercially successful products compiling legacy BUILDER code to .NET [146].

CorVision largely has the same features and serves the same purpose as ADS/ONLINE (subsection 2.3.2). It was designed to replace 3GLs such as COBOL, allowing users to generate both simple and complex applications. CorVision consists of a collection of tools such as a data dictionary, screen generator, report generator, menu builder, a very high-level programming language and an "action diagram" editor. The application data definition is done with the data dictionary, where the user can specify the data within the application and their attributes such as length, type, input pattern and output format. Simple applications can be built by only defining data, menus, screens and reports. When a user wants to define complex response processing they could use the high-level programming language BUILDER and the action diagram editor. An action diagram in CorVision is essentially the BUILDER source code with lines and boxes drawn over the text to illustrate where the database is accessed and where the scope of a statement starts and ends.

Applications built with CorVision can be run in two modes: interpreted and compiled. The interpreted mode is meant to be used during development to give the user an impression of the end product without having to build the entire application. The compiled mode is used to compile the BUILDER code to machine code which boosts the efficiency of the program to a degree where it can be used in a production environment.

An example of BUILDER source code can be seen in Listing 2.11.

```
IF FINANCE.CHARGES:*CU = "YES"
    DAYS.OVERDUE=TODAY-(INVOICE.DATE:*IZ+TERMS:*CU)
    CONDITIONAL
    DAYS.OVERDUE GE 90
        BAL.OVER.90 = BAL.OVER.90+BALANCE.DUE:*IZ
        FINANCE.PCT = FINANCE.PCT.90:*CU
    DAYS.OVERDUE GE 60
        BAL.OVER.60 = BAL.OVER.60+BALANCE.DUE:*IZ
        FINANCE.PCT = FINANCE.PCT.60:*CU
    DAYS.OVERDUE GE 30
        BAL.OVER.30 = BAL.OVER.30+BALANCE.DUE:*IZ
        FINANCE.PCT = FINANCE.PCT.30:*CU
    ENDCONDITIONAL
    IF DAYS.OVERDUE GE 30
        ADD_FINANCE_CHARGE
    ENDIF
ENDIF
```

LISTING 2.11: A Snippet of BUILDER Code

As can be seen in the snippet above BUILDER does have some COBOL-like elements: the lines do resemble sentences, the `IF-THEN-ELSE` structure is largely similar and the language has

EVALUATE-like functionality named CONDITIONAL. The statements are also structured in the sense that most have a clear beginning and end.

2.3.4 DATATRIEVE

DATATRIEVE is a 4GL with the main purpose of being a data management and retrieval system originally developed by Digital Equipment Corporation and is currently owned and maintained by VMS Software. One of the main components of DATATRIEVE is its domains, these are a construct which contain the definition of the data (a record) and the location where the entries tied to this definition are stored. An example of a record definition can be seen in Listing 2.12.

```
DEFINE RECORD EMPLOYEES_REC

01 EMPLOYEES_REC

    05 EMPLOYEE_ID          PIC X(5)
                            QUERY_NAME IS ID
                            QUERY_HEADER IS ID
    05 EMPLOYEE_NAME        QUERY_NAME IS NAME
        10 LAST_NAME        PIC X(14)
                            QUERY_NAME IS L_NAME
                            QUERY_HEADER IS "LAST NAME"

        10 FIRST_NAME       PIC X(10)
                            QUERY_NAME IS F_NAME
                            QUERY_HEADER IS "FIRST NAME"
        10 MIDDLE_INITIAL   PIC X
                            QUERY_NAME IS INIT
                            QUERY_HEADER IS "INIT"
    05 EMPLOYEE_ADDRESS     QUERY_NAME IS ADDRESS
        10 ADDRESS_DATA     PIC X(20)
        10 STREET           PIC X(25)
        10 TOWN             PIC X(20)
        10 STATE            PIC X(2)
        10 ZIP              PIC X(5)
    05 SEX                  PIC X
                            VALID IF SEX = "M" OR SEX = "F"
    05 SOCIAL_SECURITY      PIC X(9)
                            EDIT_STRING IS XXXBXXBXXX
    05 BIRTHDAY             USAGE DATE
                            EDIT_STRING IS NN/DD/YY
```

LISTING 2.12: Example Record Definition in DATATRIEVE

The records in DATATRIEVE have a striking resemblance to those in COBOL, with the only difference being the ability to define more attributes of a field such as an alias.

Since DATATRIEVE is meant for data management, the textual component is geared towards the retrieval and modification of data. Listing 2.13 shows a snippet written in the textual component of DATATRIEVE.

```
FOR EMPLOYEES WITH EMPLOYEE_ID BETWEEN 1000 AND 5000
    BEGIN
        LIST EMPLOYEE_ID, EMPLOYEE_NAME, EMPLOYEE_ADDRESS
        MODIFY EMPLOYEE_NAME, EMPLOYEE_ ADDRESS
        LIST EMPLOYEE_ID, EMPLOYEE_NAME, EMPLOYEE_ADDRESS
    END
```

LISTING 2.13: A Snippet of DATATRIEVE Code

Statements in this language can be saved as procedures which can be invoked from the command line. The language itself looks similarly structured as COBOL but has differing keywords.

2.3.5 FOCUS

FOCUS is a 4GL that was originally developed by Information Builders Inc. and is currently still actively maintained by its current owner TIBCO under the name TIBCO FOCUS [147]. FOCUS supports a wide range of functionality such as report generation, screen generation and general application development. Data entities in FOCUS are defined using database descriptions. An example is shown in the Listing 2.14.

```
FILENAME=STAFF,SUFFIX=FOC,                                    $
    SEGNAME=EMPLOYEE, SEGTYPE=S,                              $
        FIELD=EMP-NO,        USAGE=A6,                        $
        FIELD=L-NAME,        USAGE=A15,                       $
        FIELD=F-NAME,        USAGE=A15,                       $
        FIELD=JOB,           USAGE=A1,                        $
        FIELD=SALARY,        USAGE=P6,                        $
        FIELD=AGE,           USAGE=Z2,                        $
        FIELD=SEX,           USAGE=A1,                        $
        FIELD=STREET,        USAGE=A15,                       $
        FIELD=CITY,          USAGE=A15,                       $
        FIELD=STATE,         USAGE=A2,                        $
        FIELD=ZIP,           USAGE=A5,                        $
    SEGNAME=PROJRATE,PARENT=EMPLOYEE, SEGTYPE=S,              $
        FIELD=PROJ-NO,       USAGE=A6,                        $
        FIELD=RATING,        USAGE=A1,                        $
    SEGNAME=JOBHIST, PARENT=EMPLOYEE, SEGTYPE=S,              $
        FIELD=JOB-CODE,      USAGE=A2,                        $
        FIELD=JOB-START,     USAGE=A6,                        $
        FIELD=JOB-END,       USAGE=A6,                        $
```

LISTING 2.14: An Example of a FOCUS Database Definition

In this format, each attribute of a line is separated by a comma and the line is terminated by a dollar sign. The usage of a field defines the type and length of the field. These are the same symbols found in the data definition section in COBOL, so A15 means 15 alphabetic characters and Z2 means 2 digits with zero suppression. Thus, this "usage" is somewhat similar in syntax and semantics to COBOL's `PICTURE` clauses.

Report generation is done by using the very high-level programming language included with FOCUS. Listing 2.15 shows an example of report generation using the language.

```
TABLE FILE PRODUCT
    SUM AMOUNT AS 'TOTAL,AMOUNT' BY MONTH
    SUM AMOUNT AND UNITS BY MONTH BY CUSTOMER
    PRINT ESTIMATE BY MONTH BY CUSTOMER BY PRODUCT
        IF AREA IS 'EAST'
END
```

LISTING 2.15: An Example of FOCUS Report Generation Code

The language also allows for the modification of data. Listing 2.16 shows how this can be done.

```
MODIFY FILE STAFF
VALIDATE
SAL-TEST = SALARY LT 50000;
FIXFORM EMP-NO/6 SALARY/6
MATCH EMP-NO
    ON MATCH UPDATE SALARY
    ON NOMATCH TYPE "NO MATCHING EMPLOYEE NUMBER!"
    ON NOMATCH REJECT
DATA ON SALARY FILE
END
```

LISTING 2.16: An Example of FOCUS Code Where Data Is Modified

From these examples, it can be observed that the structure of these languages is somewhere between a COBOL-like language and a SQL-like language. The language is made up of commands such as `TABLE` and `MODIFY` and sub-commands such as `SUM` and `MATCH`.

2.3.6 IDEAL

IDEAL is a 4GL which was initially developed by Applied Data Research and is currently owned by Broadcom under the name CA IDEAL [19]. The main purpose of this 4GL is application development. IDEAL is designed as a replacement for 3GLs such as COBOL and PL/I and is meant to make calling external procedures written in those languages unnecessary.

IDEAL was designed to be closely integrated with other software by Applied Data Research, including its database ADR/DATACOM/DB and data definition software ADR/DATA DICTIONARY. The IDEAL procedural language allows users to define procedures by using a structured COBOL-like language which aims to be closer to natural English than COBOL. An example program can be seen in Listing 2.17.

```
<<MAIN>> PROCEDURE
    FOR EACH PAYROLL
        WHERE (ACTIVITY-CODE = 'A'
        AND ACTIVITY-STATUS = 'S')
        OR CURRENT-RATE > 14.00
        DO CALC
        LIST NUMBER W-YTD-NET
    WHEN NONE
        LIST 'NO RECORDS FOUND'
    ENDFOR
ENDPROCEDURE

<<CALC>> PROCEDURE
    SET W-YTD-NET = (YTD-WAGES + YTD-COMMISSION) - YTD-TAXES
    SET W-DUES = CURRENT-RATE * .10
ENDPROCEDURE
```

LISTING 2.17: Example of an IDEAL Program

As can be observed in Listing 2.17, the constructs in the language are well structured, giving a clear start and end to where each construct exists in the code.

2.3.7 INTELLECT

INTELLECT is a 4GL which was originally developed by Artificial Intelligence Corporation and is designed to be a query language and report generator that is queryable using natural English. It has many names since it was marketed under different brands by various companies: Cullinet Software used "OnLine English", Management Decision Systems used "ELI: English Language Interface", InSci used "GRS EXEC", Dartmouth College partners referred to it as "ROBOT" [50], in GCOS 64 it was called"IQS: Interactive Query System", etc. This 4GL works by matching words to a lexicon component of a system which is used to keep track of words and synonyms. This lexicon is customisable and is usually managed by a system administrator of the organisation.

When INTELLECT is queried, it first responds with the translation of the input in a more formal standard form so that the user can verify if INTELLECT interpreted the question as they intended. As an example, when given the input in listing Listing 2.18, INTELLECT would first respond with its standard form translation shown in Listing 2.19 and would then give the user the result of the query.

```
GIVE ME THE NAME, AGE, AND CITY OF OUR NEW JERSEY ADMINISTRATORS.
```

LISTING 2.18: An Example Query for INTELLECT

```
PRINT THE NAME, AGE, AND CITY OF ALL EMPLOYEES WITH
    JOB = ADMINISTRATOR
    AND STATE = NEW JERSEY.
```

LISTING 2.19: The Example Query in Listing 2.18 Translated to Standard Form

2.3.8 MANTIS

MANTIS is a 4GL developed and actively maintained by Cincom [25]. It is a 4GL similar to ADS/ONLINE (covered in subsection 2.3.2) and APPLICATION FACTORY (covered in subsection 2.3.3) that is mainly geared towards application prototyping and development. It has non-procedural menu-based functionality such as a screen- and a data record designer. Besides this, it also provides users with a very high-level programming language which allows users to define procedures.

Instead of whitespace, the MANTIS language uses periods to indent code. An example of this can be seen in Listing 2.20.

```
 10 ENTRY MENU_PROGRAM
 20 .SCREEN MAP ("MENU_SCREEN")
 30 .CONVERSE MAP
 40 .WHILE MAP<>"CANCEL"
 50 ..WHEN ACTION=1 OR MAP="PF1"
 60 ...CHAIN"CUST_ENTRY"
 70 ..WHEN ACTION=2 OR MAP="PF2"
 80 ...CHAIN"CUST_REPORT"
 90 ..END
100 ..CLEAR MAP
110 ..CONVERSE MAP
120 .END
130 .STOP
140 EXIT
```

LISTING 2.20: An Example of MANTIS Code

One thing that is highlighted about the language is its ability to process arrays and text. Users can define arrays in the language by using either the `BIG` or `SMALL` keywords as shown in Listing 2.21.

```
10 BIG SUM (2,3)
```

LISTING 2.21: A 2D Array Definition in MANTIS

This example defines a two-dimensional array named `SUM` with two rows and three columns. The locations in this array can be accessed by using `SUM(X,Y)` where `X` should contain the index of the row and `Y` should contain the index of the column.

Text processing is handled in a similar way using either `TEXT(X)` or `TEXT(X,Y)`. If a variable is defined using only `X`, then the variable can store one line of text of `X` characters. If both `X` and `Y` are present in the definition then the variable can store `X` lines of `Y` characters.

The procedural language in MANTIS can be considered a COBOL-like language with its main deviation from the structure being the periods being used as indentation.

2.3.9 MIMER

MIMER is a data management-oriented 4GL. It consists of several tools that all interface with a central relational database management system (RDBMS). One of these tools is called MIMER/PG, a program generation tool which allows users to define new applications using a high-level programming language. Listing 2.22 shows how a simple program can be defined using the language.

```
SPECIFY AVG(X) = <R:SUM(X)/COUNT();
      VALUE R>;

SPECIFY AVGDEP(DNAME, SAL) =
   <INIT 10; "DEPARTMENT";35;"AVERAGE SALERY";
   SUMMARY DNAME:<10;DNAME;38;AVG(SAL)>>;

PROGRAM P_AVGDEP =
   AVGDEP GET DEPT.DNAME, EMP.SAL
      WHERE EMP.DEPT EQ DEPT.DEPTNO;
```

LISTING 2.22: An Example of a MIMER Program

MIMER allows programs like the one shown above to be translated into a lower level 3GL like COBOL or PL/I which, the result of this translation can be deployed on a mainframe. The syntax for the program definition in MIMER is quite unique compared to the other languages shown earlier in this chapter.

2.3.10 NATURAL

NATURAL is a 4GL developed and maintained by Software AG. NATURAL is a database-oriented 4GL which is both appropriate for ad-hoc querying and production-level application development. Software AG has recently pledged to support NATURAL and its accompanying database system ADABAS at least until 2050 [6].

Users of NATURAL can use its high-level programming language to develop their applications. An example of code in this language can be found in the snippet in Listing 2.23.

```
0010 FIND EMPLOYEE WITH STATE = 'NY'
0020                AND CITY = 'ITHACA' OR = 'NEW YORK'
0030    ACCEPT IF SEX = 'F'
0040    SORTED BY CITY L-NAME
0050    DISPLAY L-NAME F-NAME CITY SALARY
0060    AT BREAK OF CITY DO
0070            SKIP1
0080            WRITE 'TOTAL SALARIES:' OLD (CITY)
0090                 T*SALARY SUM (SALARY)
0100            SKIP1
0110        DOEND
0120    AT END OF DATA DO
0130            SKIP1
0140            WRITE 'AVERAGE SALARY: ' AVER (SALARY)
0150            WRITE 'MINIMUM SALARY: ' MIN (SALARY)
0160            WRITE 'MAXIMUM SALARY: ' MAX (SALARY)
0170        DOEND
0180 END
```

LISTING 2.23: A Snippet of NATURAL Code

The language seems to have a similar structure to COBOL except with higher-level constructs that are characteristic of 4GLs. Besides this, it also supports COBOL-like edit masks for the data as seen in Listing 2.24.

```
0120        DISPLAY 'EMPLOYEE' #HOLD-NAME
0130              'PRESENT/SALARY' SALARY (EM= $ZZZ, ZZ9)
0140                'SALARY/FOLLOWING/7% RAISE' #NEW-SALARY
0150                                (EM = $ZZZ,ZZ9)
```

LISTING 2.24: A Snippet of NATURAL Code Which Displays the Edit Mask Feature

2.3.11 NOMAD2

NOMAD2 is a 4GL which was developed by D&B Computing Services, its current vendor is Select Business Solutions [135]. This 4GL is mainly geared toward developing data processing applications. NOMAD2 provides the user with a schema description language which allows for the definition of tables and columns in the database. An example of a schema description can be seen in Listing 2.25.

```
SCHEMA;

MASTER DEPARTMENT INSERT KEYED(DEPTNO);
    ITEM DEPTNO AS A4;
    ITEM DEPTNAME AS A12 ALIAS = DNAME;

MASTER VENDORS INSERT KEYED (VENDID);
    ITEM VENDID AS A3 HEADING = 'VENDOR:CODE';
    ITEM VENDNAME AS A20 ALIAS = 'VNAME'

MASTER SUPPLIES INSERT KEYED(INVNUM):
    ITEM DEPTNUM AS MEMBER 'DEPARTMENT';
    ITEM VENDNUM AS MEMBER 'VENDORS';
    ITEM INVNUM AS A6 MASK AAX999;
    ITEM PURCHDATE DATE 'MM/DD/YY'
        HEADING = 'DATE OF PURCHASE';
    ITEM COST AS $99,999.99 LIMITS (0:20000);
    ITEM QTY AS 9999 LIMITS (0:400);
    DEFINE TCOST AS $999,999.99
           HEADING = 'TOTAL COST' EXPR = COST * QTY;
```

LISTING 2.25: A NOMAD2 Schema Description

As can be observed in the example above, the type definition here seems similar to what is found in a COBOL data division with some deviations such as the definition of a decimal point, which is defined by a period symbol in this notation. The language of NOMAD2 consists of two types of statements: non-procedural (such as `LIST` and `PLOT`) and procedural (such as `IF-THEN-ELSE` and `FOR EACH`). NOMAD2 supports a conversational mode and a procedural mode. In the conversational mode, the user interacts with NOMAD2 directly as a query language and is able to execute commands such as the one in Listing 2.26.

```
LIST BY DEPT ID EMPLOYEE SALARY POSITION
```

LISTING 2.26: A Query in NOMAD2's Conversational Mode

In the procedural mode, users are able to define procedures and store them using a mix of procedural and non-procedural statements. Listing 2.27 shows an example of what procedures might look like.

```
FOR EACH SKILL DO;
    IF SKILL LEVEL GT 8 THEN DO;
    PRINT 'EXPERT IN' SKILL NAME;
    CHANGE EXPERT_CNT = EXPERT_CNT + 1;
END;
```

LISTING 2.27: A Snippet of NOMAD2 Code

From the listings above it can be concluded that the language in NOMAD2 syntactically borrows from both SQL and COBOL for its non-procedural component and procedural component respectively.

2.3.12 RAMIS II

RAMIS II was initially developed by Mathematica Products Group and is currently maintained by Broadcom under the name CA RAMIS [20].

RAMIS II is primarily a data processing 4GL and has the tools that are commonly found in this category of 4GLs such as a query language, report generator, a menu-driven screen and a natural English interface (RAMIS II ENGLISH) which works in a similar fashion as INTELLECT (subsection 2.3.7).

Besides the English and menu-driven ways of using RAMIS II, it also provides a more structured language. Users can use a long version of the syntax or a short version. The two snippets in Listing 2.28 would have the same result.

```
TABLE
NOTE TOP
'SALES STATISTICS'
FILE SALESDATA
SUM UNITS AND MAXIMUM UNITS
WITHIN YEAR
BY YEAR BY MONTH
END

TABLE
NOTE AT THE TOP OF THE PAGE
'SALES STATISTICS'
FILE SALESDATA
SUM THE UNITS AND THE MAXIMUM OF THE UNITS
WITHIN THE YEAR
BY YEAR BY MONTH
END
```

LISTING 2.28: Two Snippets of RAMIS II Code Which Produce the Same Result

The language also allows for the user to define subroutines within routines as shown in Listing 2.29.

```
RELATE
PROJECT EMPLOYEE BY EMP-NO IF JOB IS 3
PROJECT PROJRATE BY EMP-NO IF AVERAGE RATNG GT 6
SAVE THE INTERSECTION
END
COMPUTE INCR1000
FILE EMPLOYEE
SALARY = SALARY + 1000;
END
MAINTAIN FILE EMPLOYEE
TRANSACTION SOURCE RAMSAVE
TRANSACTION LAYOUT EMP-NO
WHEN MATCH PERFORM INCR1000
           UPDATE SALARY
END
```

LISTING 2.29: Example of a RAMIS II Routine

It can be concluded from the listing above that the structured language in RAMIS II is relatively high-level and sometimes even borders natural language. It has some slight similarities to languages like COBOL and SQL but these similarities are much less prominent than 4GLs that are discussed in the rest of this section.

2.3.13 SYSTEM W

SYSTEM W is a so-called "decision support system", a subclass of 4GLs used for data analysis tasks such as planning and statistical forecasting. It was developed by COMSHARE Inc. Listing 2.30 shows a report definition written in the SYSTEM W language.

```
ACROSS PERIODS, *DESCRIPTION, P1-P3
TEXT CENTER 'FOUR YEAR ANALYSIS' WITH * BENEATH
SKIP 2
END
DECIMALS 0
COLUMN WIDTH 20
HEADINGS DESCRIPTION
SKIP
PRINT PRICE
PRINT U.SOLD
UNDERLINE DATA '-'
SKIP
PRINT REVENUE
UNDERLINE DATA '='
SKIP 5
```

LISTING 2.30: An Example Report Definition in SYSTEM W

Something that can be noticed immediately is the structure and relative closeness to natural language that was also seen in the RAMIS II syntax.

2.3.14 USE-IT

USE-IT is a 4GL developed by Higher Order Software (HOS) following a methodology of the same name. It uses a unique approach to application development, rarely observed in other languages: USE-IT relies on a graphically oriented specification language instead of a text-based one. It is claimed that code generated from a USE-IT specification is always mathematically correct and provably error-free "by ignoring existing programming languages" because "only in this way can USE-IT's provably correct constructs be built" [96]. However, USE-IT could generate code in conventional languages like FORTRAN, COBOL, C or Pascal.

The Higher Order Software methodology was developed by Margaret Hamilton and Saydean Zeldin [47] — the same Margaret Hamilton who is credited for coining the term "software engineering" during her pioneering work on the Apollo project. This was one of the first ambitious applications of methods of formal verification to software engineering problems, relating design with verification [48]. The HOS methodology received a lot of attention both in practice and in academic research [5, 92], and had significant impact on modern software languages like SysML [46]. Shortcomings of USE-IT and HOS in general, like "solving" the verification problem by syntactically disallowing incorrect programs, led others to develop many other languages, approaches and tools, the most notable being Statecharts [49].

The flow of the USE-IT application is defined using the graphical interface, which allows to decompose the software design into essentially a tree of functions, following a set of six well-formedness axioms [47]. When the user finishes the specification they can generate the application, USE-IT then only used proven constructs to generate mathematically correct 3GL code.

2.3.15 PACBASE

PACBASE is a 4GL that is mainly used for generating COBOL applications. It was initially developed by CGI which was later acquired by IBM. PACBASE was maintained by IBM until it's discontinuation in 2015 [32]. It features a strict position-based language for the definition of data and procedures [62]. At the centre of PACBASE is the concept of the "entity". Entities can be one of elements, segments, programs, reports or macros. While programs, reports and macros speak

for themselves, elements and segments can be defined as data definitions in a similar format to those found in a COBOL data division, where elements are the equivalent of fields and segments are the equivalent of records. Entities are meant to be reused across multiple programs [143]. An example of a procedure definition in PACBASE can be seen in Listing 2.31.

```
--------------------------------------------------------------------------------
|                   PURCHASING MANAGEMENT SYSTEM         SG000008.LILI.CIV.1583 |
|                                                                              |
| PROCEDURAL CODE       P P00001 VENDOR REPORTS                FUNCTION: 02    |
|                                                                              |
|                                                                              |
|                                                                              |
| A SF LIN OPE OPERANDS                            LVTY CONDITION              |
|   AA      N   GET CURRENT DATE                   10BL                        |
|   AA   10 ADT DATOR                                                          |
| - -- --- --- ---------------------------------- ---- -------------------------|
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
| *** END ***                                                                  |
| 0: C1 CH: -P                                                                 |
--------------------------------------------------------------------------------
```

LISTING 2.31: A Procedure Definition Screen in PACBASE [62]

2.3.16 AppBuilder

AppBuilder[1] is a 4GL which is mainly used for enterprise application development [94]. This 4GL was developed by Level 8 Systems under the name Geneva AppBuilder [85]. In 2001, Level 8 Systems sold this 4GL to Liraz Systems which brought AppBuilder under one of its subsidiaries called BluePhoenix Solutions [15, 86]. In 2011, BluePhoenix sold AppBuilder to Magic Software Enterprises which remains the owner to this day [23].

Development of AppBuilder applications happens in its IDE [93]. With this IDE users can develop their applications and generate code in a target language. All artefacts of the development process, besides the aforementioned generated code, could be categorised in 5 notations, these are described in Table 2.4. An example of one of these notations, the HpsBindfile, is shown in Listing 2.32.

HpsSource is the central language within AppBuilder and it allows users to define the procedures for their AppBuilder programs [112, 167]. It supports interacting with objects, which are defined as the GUI elements declared in the HpsPanel file, using ObjectSpeak expressions. These expressions allow users to manipulate these GUI elements in a similar way to modern application development tools which provide a link between the GUI definition and the source code. It also supports embedded SQL statements within the source code, allowing the user to interact with a database within the same language as where they define their procedures.

[1] Not to be confused with *Application Builder* by James Martin Associates [96, p.478] or one of the modern low-code platforms called "App Builder": by Agora [172], ArcGIS [36], Comidor [139], Infragistics [44], Flask [39], etc.

Notation	Description
HpsSource	Language used for programming procedures
HpsBindfile	Data definition notation for data structures used within the application
MMRef	JSON-like language used for localisation, allows applications to be displayed in multiple languages
HpsPanel	Lisp-like language containing S-expressions
HpsText	Contains Metadata similar to Javadoc

TABLE 2.4: The notations found within AppBuilder

```
$$FILE 06/07/2017 23:59:59
$$FOO ABCD Y 06/07/2017 23:59:59 XYZ
A 1 00010 00 0000 Y Y N Y NAMEA NAMEB S
C 2 00015 02 0000 Y Y Y Y NAMEDDDD NAME EEE S
F 5 00030 00 0020 Y N N Y NAMEG NAMEH S
$$BAR EFGHKLMN Y 06/07/2017 23:59:59 N/A
A LONGER_NAME_FOR_ENTITY 999 10.0
A ANSWER_TO_THE_ULTIMATE_QUESTION 42 7.5
```

LISTING 2.32: An Example of the Contents of a HPSBindfile [167]

The syntax of these notations varies greatly. For example, HpsSource resembles a mainstream programming language and HpsBindFile is more of a data description language with strict position-based rules.

2.3.17 ABAP

ABAP (previously short for *Allgemeiner Berichts- (und) Aufbereitungs-Prozessor*, and later renamed to *Advanced Business (and) Application Programming* [153]) is a 4GL created by SAP in 1983 which initially was meant to be used as a report generator targeting an audience with a technical background. It has seen many developments since then, the main of which being the transition from a report generator to something that more closely resembles an object-oriented general-purpose programming language [129].

Like many other programming languages, ABAP allows users to create two types of binary files: executable binaries and binary files that resemble libraries. The executable binaries can be split into two categories, reports and module pools. The former is used for generating reports based on a query with the data in these reports being directly modifiable. The latter is used for defining complex applications with multiple screens and dialogs [124].

The structure of ABAP is very close to COBOL as is evident in Listing 2.33. ABAP does have some peculiarities when it comes to whitespace, the presence (or lack) of whitespace can make two similar-looking operations behave completely differently. For example, in Listing 2.34 the first operation sets the value of `x` to the substring of `a` starting from position `b` with a length of `c`. The second operation sets the value of `x` to the sum of `a` and the result of calling method `b` with parameter `c` [126–128, 155].

```
CLASS zcl_example_class DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    INTERFACES if_oo_adt_classrun.
  PROTECTED SECTION.
  PRIVATE SECTION.
ENDCLASS.

CLASS zcl_example_class IMPLEMENTATION.
  METHOD if_oo_adt_classrun~main.
    out→write( 'Hello World!' ).
  ENDMETHOD.
ENDCLASS.
```

LISTING 2.33: A Sample of Object-Oriented ABAP Code [125]

```
x = a+b(c)

x = a + b( c )
```

LISTING 2.34: An ABAP Sample Showcasing Syntactically Similar but Semantically Different Code [155]

2.3.18 ADF

ADF (*Application Development Facility*) is a 4GL by IBM that functions as an application generator [12]. This 4GL provides configurable pre-programmed modules that aim to speed up development. It is stated that this method of accelerating development is not as flexible as the ones found in other application generators covered in this section, requiring users of ADF to write 3GL code for features that are not available as pre-programmed modules.

ADF is also said to be syntactically less user-friendly than other languages in this section. ADF code is written in a similar format as IBM's Job Control Language [66], a sample of ADF code can be found in the snippet in Listing 2.35.

```
* APPLICATION DEFINITION STATEMENTS FOR PARTS DATA BASE
*
SYSTEM  SYSID-SAMP,DBID=PA,          RULE ID CHARS
        SOMTX=OR,                    DEFAULT SEC. OPTION
        SIGNON=YES                   SIGNON SCREEN
        POMENU=(A,B,C,D,F,H,I),      PRIMARY SCREEN
        PCBNO=1,                     PCB NUMBER
        SDBNAME='ASSEMBLY PARTS'     DEFAULT DB NAME
        SHEADING='SAMPLE PROBLEM',   GENERAL HEADING
        SFORMAT=DASH,                SCREEN FORMAT
        PGROUP=ZZ,                   PROJECT GROUP
        ASMLIST=NOLIST               GENERATE OPTIONS

*
* APPLICATION DEFINITION INPUT FOR PARTROOT SEGMENT
*
SEGMENT LEVEL=1,ID=PA,NAME=PARTROOT,LENGTH=50,
        SNAME='PART SEGMENT',SKSEG=18
FIELD   ID=KEY,LENGTH=17,POS=1,KEY=YES,NAME=PARTKEY,
        SNAME='PART NUMBER',DISP=YES,RFL=YES
FIELD   ID=DESC,LENGTH=20,POS=27,SNAME='DESCRIPTION',
        DISP=YES,REL=YES
```

LISTING 2.35: Sample ADF Code

2.3.19 ADRS II

ADRS II (*A Departmental Reporting System*) is a 4GL that mainly functions as a query language and a report generation tool. This 4GL is based on APL (covered in subsection 2.3.20), which allows users familiar with it to extend the feature set of ADRS II.

ADRS II has a handful of commands, these and their descriptions are shown in Table 2.5.

Command	Description
`CALCULATE`	Calculates and stores results
`COMPUTE`	Executes a program
`DATA`	Enters, modifies and displays data
`IF`	Formulates selection statements
`LIST`	Prepare one-time report
`PLOT`	Prints a graph
`PROGRAM`	Writes and stores a program
`REPORT`	Prints a report
`SETREPORTS`	Define a report that will be used frequently
`SETUP`	Initializes columns for storage of data
`SORT`	Sorts a file

TABLE 2.5: Commands Available in ADRS II

Data definition is done using the `SETUP` command, the user is prompted to give the column definitions of their data. The expected input format for this prompt has two parts separated by "`/`". The first part signifies how many characters a column should have, one can define decimal values here by putting two values separated by a space where the first defines the digits before- and the second defines the number of digits after the decimal separator. The second part contains the name of the column, where a comma indicates a line break. This is illustrated in Listing 2.36.

```
?COL1?:10/Name
?COL2?:2/AGE
?COL3?:6 2/GPAY
?COL4?:10/DEPT
?COL5?:8 2/PAY,YTD
```

LISTING 2.36: Data Definition in ADRS II

When retrieving data in ADRS II one can specify the criteria of the desired data using if statements similar to those in COBOL as shown in Listing 2.37. The numbers on the left-hand side of the comparisons signify the column number where the criterion should be applied.

```
IF 5 LT 15000 AND 4 EQ 'BOTTLING'
```

LISTING 2.37: Selection Criteria in ADRS II

Programs are defined in a similar manner to the data definition, the user runs the `PROGRAM` command and is then prompted to enter the statements of the program. In Listing 2.38, each statement starts with the direction the statement should be executed in with `A` being "across" (using columns) and `D` being "down" (using rows). The left- and right-hand side of the statements signify the rows or columns (depending on the direction) on which the operation should be performed. The value after the equality sign signifies in what row or column the result should be put.

```
?STATEMENT 1? A 4 * 5 = 9
?STATEMENT 2? A 2 - 3 = 6
?STATEMENT 3? D 3 + 4 = 7
```

LISTING 2.38: A Program Definition in ADRS II

2.3.20 APL

APL does not really fit the definition of a 4GL given by Martin and Leben, which the authors do admit. They justify its inclusion in the books [96, 97] by stating that it is the most 4GL-like 3GL that existed when the books were written. This language is designed for a technical audience (such as engineers, mathematicians and scientists) for use on complex problems.

Syntactically APL is quite unique compared to other languages covered in this chapter. APL can be very simple but gets significantly more complex once its more advanced features are used. For instance, simple arithmetic operations can be done in the same way as on a calculator: with a left-hand side, an infix operator and a right-hand side. On the other hand, when computing something a bit more exciting such as the standard deviation of a given vector as shown in Listing 2.39, the notation can get confusing to a new user.

```
A←⍞
M←(+/A)÷⍴A
B←(A-M)*2
C←+/B
D←(C÷⍴A)*0.5
'MEAN=';M
'STANDARD DEVIATION=';D
∇
```

LISTING 2.39: A Sample APL Program

As can be seen in Listing 2.39, APL uses some nonstandard symbols, these require a specialised keyboard (shown in Figure 2.1) that are specifically made to write APL programs.

FIGURE 2.1: An APL Keyboard by Dyalog [35], a Provider of a Third-Party APL Implementation

APL is also used as a basis for other 4GLs such as ADRS II (covered in subsection 2.3.19) and IC/I, the latter being a repackaging of APL tools for business professionals.

2.3.21 AS

AS (*Application System*) is a 4GL by IBM mainly intended to be used by end-users for accessing, processing and manipulating data. Its main features are its ability to generate reports from the data and allow users to write their own simple applications.

AS is comprised of tools in the form of facilities and written languages. Its languages include:

- COMPOSE, a text-processing language
- GRAPH, a language for making graphics
- IMAGE, a language for creating screens
- MODEL, a language for creating financial planning applications
- PROCEDURE, a language for defining procedures
- REPORT, a language for generating tabular reports
- TABULATE, a language for data analysis applications

Most of these languages are very COBOL-like in their structure, except for the PROCEDURE language. This language looks more like an assembly-like language, albeit one with higher level constructs than the ones in actual assembly languages, as can be seen in Listing 2.40.

```
PROCEDURE repproc,repproc
screen lower
terminal newline(_)
in machine
run machrep
in parts,movements
match
run partrep
cancel
run option
run optrep
in optprice
run trend
in assets
select department=22;23
run assrep
```

LISTING 2.40: A Program Written in the PROCEDURE Language of AS

2.3.22 CSP

CSP (*Cross System Product Set*) is a 4GL by IBM that provides users with an interactive application generator similar to ADS/ONLINE (subsection 2.3.2). Besides the menu-driven facilities to define data records, CSP provides a procedural language named the *CSP/AD Procedural Language* to define procedures in a COBOL-like manner. A sample of procedural code written in this language is shown in Listing 2.41.

```
PROC 1      ***************************************************
            *               USER ENTERS CUSTOMER NAME          *
            ***************************************************
                            SET MAP1 CLEAR;
                            SET MAP2 CLEAR;
                            SET MAP3 CLEAR;
            OPTION -        CONVERS  MAP1        MAP
                            TEST EZEAID PF3 EZECLOS;
            FLOW            PROC2
PROC2       ***************************************************
            *               CSP READS RECORD FOR AMT LIMIT     *
            ***************************************************
                            MOVE MAP1.CUSNAM TO REC1.CUSNAM;
            OPTION -        INQUIRY  REC1     RECORD
                            MOVE REC1.CUSNAM TO MAP2.CUSNAM;
                            MOVE REC1.AMTLIM TO MAP2.AMTLIM;
PROC3       ***************************************************
            *               USER ENTERS DATA, CSP TOTS         *
            ***************************************************
            OPTION -        CONVERS  MAP2        MAP
                            MOVE MAP2 TO MAP3;
                            TEST EZEAID PF3 EZECLOS;
                            STGRP1;
PROC4       ***************************************************
            *               CSP DISPLAYS TO BE PAID            *
            ***************************************************
            OPTION -        CONVERS  MAP3        MAP
                            TEST EZEAID PF3 EZECLOS;
            FLOW            TEST EZEAID ENTER PROC1;
STGRP1      ***************************************************
            *               CALCULATIONS FOR FINANCE APPL      *
            ***************************************************
                            COUNT = MAP2.YEARS * 12;
                            MOVE COUNT TO COUNT1;
                            A = .01 * MAP2.INT / 12;
                            B = 1 + A;
                            MOVE B TO TOT;
                            STGRP2;
                            C = 1 - 1 / TOT;
                            D = C / A;
                            E = 1 / D;
                            PAY = MAP2.AMT * E;
                            TOTAL = PAY * COUNT;
STGRP2      ***************************************************
            *               CALCULATION OF MONTHLY RATE        *
            ***************************************************
                            WHILE COUNT1 GT 1;
                              TOT = TOT * B;
                              COUNT1 = COUNT1 - 1;
                            END;
```

LISTING 2.41: A Sample of Code Written in the Procedural Language of CSP

2.3.23 DMS

DMS (*Development Management System*) is a 4GL by IBM and is one of the earliest application generators that came to market. Its main features are creating and manipulating data, validation of data, arithmetic and dialog generation. These features can be extended by defining subroutines in COBOL.

DMS supports two modes of program definition, one on-line in a terminal and one off-line done on paper forms, which are later key-entered. The latter was the original and only method for earlier versions of DMS. An example of this mode of application definition can be seen in Figure 2.2.

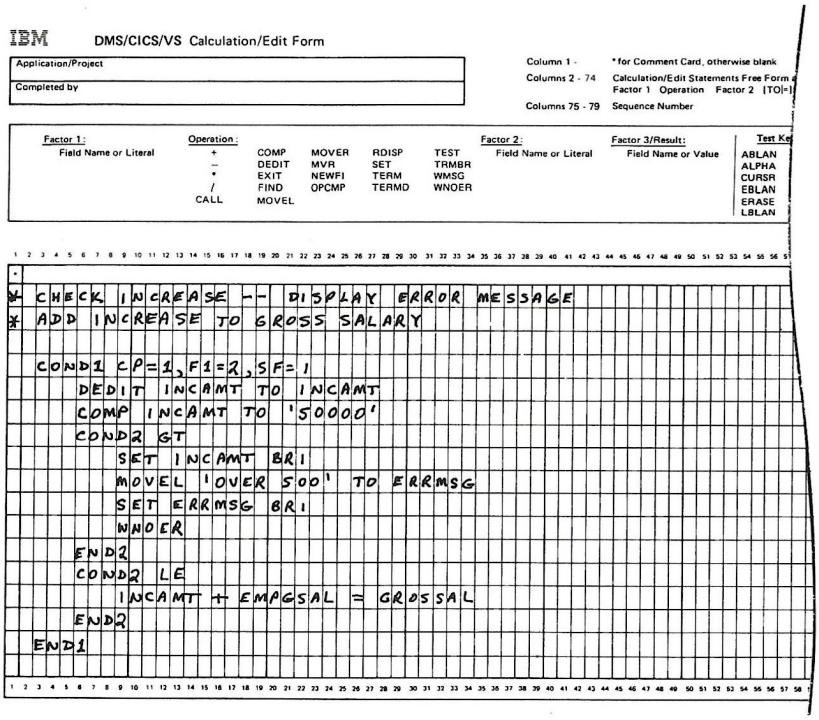

FIGURE 2.2: An Example of a Physical DMS Form

2.3.24 GIS

GIS (General Information Store, or Generalized Information System)[2] [21] is a 4GL by IBM that is meant to be used as a query language and report generator with many of the same features found in DATATRIEVE (subsection 2.3.4). It is stated to be a large language meant for data processing professionals, but it can also be used by end-users using a subset of the language. A snippet of a query written in GIS can be found in Listing 2.42. When looking at this snippet one can notice a similarity to queries written in SQL. In fact, some regard GIS as an ancestor of SQL [56].

[2]Not to be confused with ArcGIS which also has a 4GL called Informix and comes from Esri but collaborates with IBM nowadays.

```
QUERY PERSONNEL
   WHEN LOCATION EQ '173'
   AND DEPTO EQ '800', '898'
   AND (SALARY BT 18000, 30000 OR POSCODE GE '60000')
IF SALARY BT 18000, 20000
   HOLD HOLDFL1 NAME, SALARY POSCODE, DEPTO
IF SALARY GT 20000
   HOLD HOLDFL2 RECORD
SORT98 HOLDFL1 DEPTNO, DES SALARY, NAME
QUERY HOLDFL1
   LOCATE RECORD
   LIST DEPTNO, SALARY, NAME, POSCODE
SORT29 HOLDFL2 NAME
QUERY HOLDFL2
   LOCATE RECORD
   LIST RECORD
END PROCEDURE
```

LISTING 2.42: A Sample GIS Query

2.3.25 QMF, SQL and QBE

QMF (*Query Management Faciluty*) is a collection of two query 4GLs by IBM namely SQL (*Structured Query Language*) and QBE (*Query By Example*).

SQL

According to the TIOBE index [148], SQL is one of the most used query languages available. Over the years other database vendors like MariaDB [95], Microsoft [100] and Oracle [103] have developed their own dialects that remain close to the original IBM language.

QBE

QBE is a query language where users are presented with an empty table, users fill in the names of the table and columns they would like to see and, if applicable, fill in the condition the data of each column should adhere to. This method of querying has not taken off the same way as SQL has over the years. However, in recent years the philosophy of "by example" has seen a recent resurgence with projects such as Copilot, which first started as a project at Microsoft subsidiary GitHub [40] and now is being implemented throughout Microsoft software such as Windows [106] and Microsoft 365 [141].

2.3.26 TIF

TIF (*The Information Facility*) is a 4GL by IBM which is meant to simplify application development by making it possible to define entire applications using only the menu-based facility. Once the application is defined users can generate the code of their application, which will be in a COBOL-like procedural language. Advanced users of TIF can modify this code directly and bypass the entire menu-based facility. The TIF procedural language also allows users to invoke subroutines defined in another language such as Assembler, COBOL or FORTRAN. An example of a program written in the TIF procedural language can be found in Listing 2.43

```
EDIT   21  RF PRPT    0     ALT = 0    SIZE = 40           11/27/86
16:30:34
 ===>
* * * DEFINE GENERAL REPORT SPECIFICATION OPTIONS
PRODUCT -
     SORT      -
     TERMINAL  -
     WIDTH 080 -
     COPIES 1
* * * DEFINE ATTRIBUTES OF FIELDS ON THE REPORT
FIELD SECTION
  S$DATE       ALIAS D
  S$TIME LEN 5 ALIAS T
  S$PG   LEN 4 ALIAS P
  DATE         ALIAS A
  REGION       ALIAS B
  PRODUCT      ALIAS C
  REVENUE      ALIAS E
  EXPENSES     ALIAS F
  CALC#1  IN * ALIAS G
  LINETOT IN * ALIAS Z
* * * DEFINE WORK FIELD
  LINETOT N15    PACKED CLEAR NOBLANKS
  CALC#1  N15    PACKED CLEAR NOBLANKS
  CALD#1  N15.02 PACKED CLEAR NOBLANKS
* * * DEFINE PROGRAM LOGIC
PROGRAM SECTION
  MOVE 0 TO LINETOT
  ADD REVENUE TO   CALC#1  IN *
  SUBTRACT EXPENSES FROM CALD#1  IN *
  MOVE CALD#1 IN * TO  CALC#1 IN *
* * * DEFINE THE IMAGE OF THE REPORT
REPORT SECTION
+-------------------------------------------------------------------------------
-
|  D/B = PRODUCT               Sample          D________   T______  PAGE
P___
+-------------------------------------------------------------------------------
-
RP
A________ B________ C___________ _____E _____F ______G
END SECTION
* * * DEFINE REPORT TOTALS
TOTAL BY PRODUCT FOR REVENUE EXPENSES G
TOTAL EOF FOR REVENUE EXPENSES G
```

LISTING 2.43: An Example of Generated TIF Code

2.4 Syntactical Categories

This chapter has covered the global properties of a sizable set of legacy languages. This section aims to group the languages covered in this chapter into syntactical categories. This knowledge can then be used to make informed design decisions during the development of grammatical inference solutions. For example, if the language to be inferred is part of a certain category, it could be worth seeing how the inference solution would fare with other languages within the same category to determine if the solution could be generalised.

The languages are categorised in the matrix in Figure 2.3. Here the columns denote the main inspiration of the language and the rows denote the position-based rigidity of the language.

	LISP-like	ALGOL-like	COBOL-like	SQL-like	Natural Language	Data storage
Fixed Position			COBOL(-like) PIC Clause FOCUS data definition ABAP			HpsBindFile (AppBuilder) ADF
Fixed Columns and Lines		FIxedform FORTRAN APL	HLASM AS FOCUS	GIS	SYSTEM W RAMIS II	MMRef (AppBuilder) ADRS II data & program defintion
Fixed Columns			ADS/ONLINE MANTIS MIMER COBOL TIF NATURAL APPLICATION FACTORY IDEAL	FOCUS ADRS II selection criteria Embedded SQL NOMAD2 CSP Embedded CICS DMS		
No Columns	HpsPanel(AppBuilder)	PL/I HpsSource(AppBuilder) Freeform FORTRAN		CICS SQL	INTELLECT RAMIS II ENGLISH	

FIGURE 2.3: Language categorisation matrix

2.5 Languages Selected for This Project

Given the age of the legacy languages languages covered in this chapter, and their ever-decreasing relevance for the average programmer, program samples written in these languages are not easily acquirable. Because of this fact, the rest of this project will consider three languages that would fall in the most populated categories defined in the categorisation of section 2.4. The following languages have been selected for this purpose:

BabyCobol – An experimental COBOL-like programming language used as an educational tool for practising with implementing legacy programming languages [169]. This language is used in the software evolution course at the University of Twente [170], where students are tasked with implementing this language. Besides COBOL, this language also takes inspiration from other legacy languages such as AppBuilder (covered in subsection 2.3.16), FORTRAN (covered in subsection 2.2.3), and PL/I (covered in subsection 2.2.2).

Oberon/Oberon-0 – Oberon [158] is an ALGOL-like general-purpose programming language developed by Niklaus Wirth. This language is based on Modula-2 [157] and Pascal [156], which, in turn, also have roots in ALGOL. Oberon-0 is a subset of Oberon and shares a similar purpose to BabyCobol in the sense that it is used as a teaching tool for programming language implementation.

SQLite – A serverless SQL language [142]. that is widely in use, currently being in ninth place for most used database engine [140]. Despite its name, SQLite promises a fully-featured SQL implementation and can support databases of up to 281 terabytes.

Chapter 3

Inferring Context-Free Grammars from Source Code Examples

Grammatical inference (also called *grammar inference* or *grammar induction* [80,144]) is a class of problems where the goal is to infer a grammar of a target language from samples of that language. In the literature, grammatical inference has been used to both infer regular grammars and context-free grammars [118, 144]. Grammatical inference is described by Sakakibara in [121] as: "*The learning task is to identify a 'correct' grammar for the (unknown) target language, given a finite number of examples of the language*". Since the languages we are trying to learn are context-free languages (see chapter 2), this chapter will mainly revolve around the inference of context-free grammars.

In section 3.1 context-free grammars will briefly be explained. Then the learning models of context-free grammar inference will be described in section 3.2. In section 3.4, practical instances of the grammatical inference of languages are highlighted. Finally, section 3.4 closes the chapter by reporting on how the insights gained during this chapter were used to make informed decisions when implementing the prototype inference algorithm in chapter 5.

3.1 Context-Free Grammars

Context-free grammars are often defined as a tuple $G = (N, T, P, S)$ with N being the nonterminals, T being the terminals, P being the production rules, and S being the starting nonterminal.

The formal properties of the elements of this tuple are as follows [59, 113]:

- There is no overlap between the nonterminals and terminals, so $N \cap T = \varnothing$.
- The set P is finite and consists of production rules in the form of $A ::= \alpha$, where $A \in N$ and α being a string of nonterminals and terminals such that $\alpha \in (N \cup T)^*$. Note that α can be the empty string ϵ, or $A ::= \epsilon$
- $S \in N$ is the starting nonterminal, which is used to start each sentence of the language.
- If $\alpha \in (N \cup T)^*$ can be derived from a nonterminal $A \in N$ then $A \Rightarrow^* \alpha$
- If $\alpha \in T^*$ and $S \Rightarrow^* \alpha$ then α is considered a sentence in the language
- The language of G is defined as the set $L(G) = \{w \mid S \Rightarrow^* w, w \in T^*\}$

3.2 Learning Models for Context-Free Grammar Inference

Throughout the years many theoretical learning models for grammatical inference have been defined. This section aims to bring to light a couple of the learning models described in a survey by Stevenson et al. [144].

3.2.1 Identification in the Limit

In 1967 Gold started off the field of grammatical inference in his paper "Language Identification in the Limit" [41]. The main idea behind the model proposed in this paper is inferring the language grammar from a set or sequence of samples (also called "presentation" or "information"). When this contains only positive samples it is called "positive" and if it contains both positive and negative samples it is called "complete". The need for negative samples is highlighted by Gold by showing that a regular language could not be identified in the limit by using only positive samples from that language despite being a member of the simplest language class of the Chomsky hierarchy. The main problem of identifying a language in the limit using only positive samples is overgeneralisation, in other words; inferring a language that accepts more sentences than the target language. Negative samples help steer the inferred language closer to the target language when using identification in the limit. However, negative samples of a language are harder to come by than positive samples. In the survey by Stevenson et al. [144], it is mentioned that there are other ways to combat overgeneralisation besides including negative samples. One method was proposed by Angluin [7], where an inference algorithm gets an additional input of a set with "tell-tale" strings of the language. This set contains strings which are not found in any other language of the same family. If elements of this set are present in the positive samples seen so far by the algorithm, Angluin proves that the algorithm will not overgeneralise its current guess of the language. Besides this method, there are other ways of combating overgeneralisation shown in the algorithms featured in the survey. These, among others, will be discussed in section 3.4.

3.2.2 Oracles and Queries

The main idea behind an oracle-based inference algorithm is the following: during its runtime, the algorithm asks an oracle (also called a teacher in some of the literature [26, 164]) queries about the concept it is trying to learn. Usually, a human expert takes the role of an oracle. Angluin defines six query types an algorithm can ask an oracle [9], these and their expected responses are listed below:

- **Membership**. *Yes* if element x from the input is in the target concept. *No* otherwise.
- **Equivalence**. *Yes* if the current hypothesis is the target concept, *No* and a counterexample otherwise.
- **Subset**. *Yes* if the current hypothesis is smaller than the target concept. *No* and a counterexample otherwise.
- **Superset**. *Yes* if the current hypothesis is larger than the target concept. *No* and a counterexample otherwise.
- **Disjointness**. *Yes* if the current hypothesis is disjoint from the target concept. *No* and a counterexample otherwise.
- **Exhaustiveness**. *Yes* if all inputs are either in the hypothesis or the target concept. *No* and a counterexample otherwise.

A minimally adequate teacher is an oracle that can only answer membership and equivalence queries. In another paper, Angluin shows that a minimally adequate teacher is enough to be able to learn the deterministic finite automata of regular languages in polynomial time [8]. Later, Sakakibara adapted the method of using a minimally adequate teacher to make an algorithm that can learn context-free grammars in polynomial time [120].

In practice using an oracle that can answer the six query types Angluin defined, or even a minimally adequate teacher for that matter, proves to be difficult. One would need to have access to either a functional compiler or a human expert of the target language. It should also be noted that even if a compiler for the target language is available, it could only be used for membership queries.

3.2.3 PAC (Probably Approximately Correct) Learning

Probably approximately correct learning (or PAC learning for short) is a general learning method that can be applied to grammatical inference problems. This method of learning was first introduced by Valiant in 1984 [151]. The main aspect of what makes it stand out from other methods in this section is that this method does not guarantee to find the exact target concept but an approximation of it. The measure of correctness is user-defined in this case by two parameters: $\epsilon > 0$ defines the accuracy, and $\delta < 1$ defines the confidence. The input for PAC learning is a set of examples and a probabilistic distribution D over these examples. The metric "distance" in PAC learning is defined as the sum of all probabilities $D(w)$ of all w in the symmetric difference of the hypothesis and the target concept. The goal of this learning method is to make the distance as small as possible. The learning is considered a success if the probability of the guess is higher than $1 - \delta$ and the distance is less than ϵ.

Valiant has demonstrated this learning method by showcasing two cases where his implementation was able to approximate bounded conjunctive normal form and disjunctive normal form expressions in polynomial time [151]. The major restriction of this method is that the algorithm is expected to learn polynomially for all distributions, something that is often not possible, even for seemingly simple concepts. Li and Vitanyi propose a change to PAC learning where only simple distributions are considered [88], with a simple distribution being a distribution that returns simple samples with a higher probability than complex ones.

3.2.4 Neural Networks

Artificial neural networks have been very popular in recent years for solving problems such as image recognition and other classification problems [4]. Neural networks accomplish their goal by approximating an unknown function that solves a given problem. This is done by training the network on a dataset with inputs and their expected outputs. For the problem of grammatical inference, the dataset would consist of sentences and a boolean value indicating if a particular sentence is in the language or not. With a sufficient amount of data and training a neural network would be able to recognise new sentences that are in the language.

Recurrent neural networks seem to be a popular variant of this learning method for grammatical inference [14, 27, 163]. This is because recurrent neural networks offer an internal state that acts as a short-term memory that can aid the learning process.

One concern with this learning method is extracting the context-free grammar from the trained neural network. For the case of deterministic finite automaton inference, A survey by Stevenson and Cordy [144] mentions the work of Cleerman et. al. [27], where the authors recover the automaton from the hidden layers of the network. In recent years there have been developments in recovering the grammars for the case of context-free languages [14, 163].

Another concern is the availability of samples. As with most machine learning frameworks, neural networks are more accurate and less biased when trained with a larger dataset. If the samples from the language to be learned are, for example, artefacts from a past software development process, the number of samples is fixed to whatever number of artefacts were left from that process and acquiring more samples could be challenging. One potential solution to this challenge includes employing techniques for learning effectively from small datasets [17, 89, 108]. Another possible solution is using similar methods for program generation found in test suite augmentation [43, 123, 160], a part of regression testing where test suites are enlarged by automatically creating tests for parts of a program that were not tested yet.

3.3 Practical Applications of Grammatical Inference

This section aims to shed light on some attempts at solving specific inference problems, their efficacy at solving the problem at hand and their applicability to general context-free grammar inference. Many of the methods in this section were found in existing surveys on the topic [118, 144], while some of the more recent examples were found elsewhere.

3.3.1 MAGIc: Memetic Algorithm for Grammatical Inference

In a paper [59] Hrnčič et al. introduced an evolutionary algorithm named MAGIc (Memetic Algorithm for Grammatical Inference). This algorithm is designed to infer domain-specific languages using a set of positive sample samples as input. It uses a so-called "memetic algorithm" which combines a regular evolutionary algorithm with local search and domain-specific knowledge [3].

The algorithm takes the positive samples and regular definitions of the language as input. For each sample, a context-free grammar that can recognise the sample is generated using the SEQUITUR algorithm [102]. Then, the algorithm runs for a given amount of generations. During a generation, the algorithm goes through phases for improvement, mutation generalisation and selection. A global description of these steps is listed below:

- During the improvement phase a grammar and accompanying true positive sample and false negative samples are picked. Using these as inputs, a local search step is performed where a grammar is being constructed that can parse both the true positive sample and the false negative sample. This is done by identifying the differences between the samples using the Linux `diff` command and the runtime information of an LR(1) parser generated for the input grammar. The `diff` tool produces outputs that indicate whether something needs to be added, deleted or replaced from one input to get the other input. Using this and the parser state when parsing the false negative sample, MAGIc can formulate a grammar which can parse both samples. The new grammar is evaluated by generating a parser and running it on all positive samples, the fitness of the grammar is determined by the ratio of successfully parsed to total samples. If the fitness is above a certain threshold, the grammar is added to the population. this is repeated for all grammars in the initial population.

- During the mutation phase the algorithm loops through the expanded population. For each grammar, there is a random value generated for each of the symbols on the right-hand sight of the production rules. If the generated value is lower than the given probability of mutation, the algorithm mutates that symbol. This is done using domain knowledge of context-free grammars, which are often written in Extended Backus-Naur Form (EBNF) which allows for making symbols optional or iterative. When a symbol is chosen to be mutated, the algorithm again makes a random choice to make the symbol optional, one-or-more iterative (iteration$^+$)

or zero-or-more iterative (iteration*). The grammars are again evaluated on their fitness, if their fitness is sufficient the mutated grammars are added to the population.

- Then the generalisation phase takes place, in which the algorithm generalises and simplifies the grammars by properly encoding concepts like recursive structures and removing redundant production rules, among other improvements. The generalised grammars are again tested for their fitness and if they show a sufficient score, they get added to the population.
- Finally, the selection phase takes place. In this phase, the algorithm picks a set of the fittest from the population. The population size is reduced to an equal number of grammars to the input samples.

These steps are repeated for the given number of generations. After the final generation, the algorithm yields a set of the fittest grammars that were encountered. The cardinality of this set is equal to the number of samples provided as input for the algorithm.

The algorithm was later improved [57] by allowing the optional inclusion of negative samples to combat overgeneralisation and several improvements to the local search used in the algorithm. This work has been further expanded upon by the authors in the form of an extension where grammar inference is applied to infer embedded DSL languages [58], and a similar approach for the semantic inference of DSLs [76].

3.3.2 Interactive Parser Synthesis by Example

Leung et al. introduce a visual tool called Parsify [84] (shown in Figure 3.1) to guide a human-led inference process using a programming by example approach. In this paradigm, the input and expected output of the desired program are provided, with which the algorithm infers the desired program. The tool described in this paper provides a user interface with the file to be parsed, an overview of the nonterminals defined so far, and a parse tree of the input made using the current grammar. To build the grammar, the user tells the tool what production rule parts of the input belong to. During the process of building the grammar, the rules can become ambiguous. For example, take the rule $expr \rightarrow expr + expr$. This can be interpreted to be both left-associative or right-associative. The user is expected to specify the correct associativity by using the user interface. Once the user has made a choice, the algorithm creates a predicate for the parse trees that is used to reject parse trees with the wrong associativity from the parser's output set. A similar approach is used for determining the precedence of infix operators. These predicates on the parse trees are called disambiguating filters.

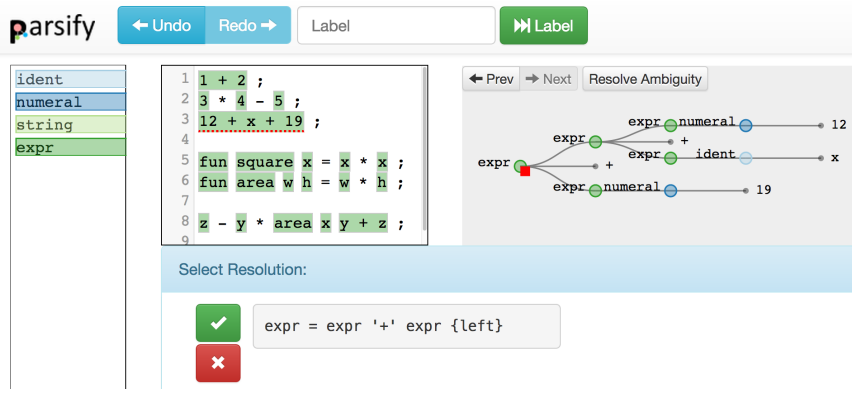

FIGURE 3.1: The User Interface of Parsify

The paper demonstrates the tool with an example where the grammar for a small input file is built. It also provides an in-depth explanation of the five components that make up the tool:

- **draw**, compute a new tree coloring. A color of a node is used to indicate to which label it belongs.
- **annotate**, accept a label (production rule).
- **generalize**, generalise an existing production rule.
- **negate**, reject an existing label.
- **resolve**, synthesize a new disambiguating filter.

The authors evaluated the tool by using it to infer the grammar Verilog (a hardware description language), Tiger (an imperative language with functional idioms), Apache (logs from an Apache web server) and SQL (a popular query language described in section 2.3.25). The input samples were divided into a breadth set and a depth set. The breadth set is described to contain anywhere from hundreds to thousands of lines of lines of code per language and contains samples of all four languages. The depth set contains hundreds of thousands up to millions of lines of code and contains only samples of Apache and SQL. The tests ran on the breadth set were all successful, managing to infer grammars that were able to parse all samples from this set. For the depth set, 97% of the Apache samples and 86% SQL samples were able to be parsed using the grammar from the breadth test. After learning a language construct that was not present in the breadth set the percentage of parsed SQL samples was increased to 97%.

According to the authors, the completion time of the tool was somewhere between 6-8 hours. To decrease this figure the authors define a set of best practices. When following these best practices, the completion time was reduced to a range between 30 minutes and 2 hours.

3.3.3 Parser Generation by Example for Legacy Pattern Languages

In this paper, Zaytsev describes the process of generating a parser for a position-based sub-language of AppBuilder (described in subsection 2.3.16) [167]. The parser generation was done in a restrictive industrial setting where only a large collection of development artefacts and human programmers with experience in that language were available. The algorithm used in this paper uses a PAC-learning-based approach.

The paper first describes a specification that is meant to express the patterns encountered in the language. This specification is made up of three parts: *patterns*, *commitments* and *bindings*. *Patterns* specify the start and end of a specific pattern, *commitments* specify the internal structure of a pattern, and *bindings* specify how to extract and put back information from a pattern.

The method used in this paper is based on the method described by Arimura, Shinohara and Otsuki [11], with changes that relax some limitations and enforce others to make the algorithm better at inferring the grammar of the target language. There are heuristics used where if more than half of the Latin alphabet is encountered in the same position, the algorithm will promote the specification for that position to be the entire alphabet. This is similarly done for numeric values: if only the digits 0–9 are encountered in a position, the specification of that position is set to be an integer.

With the help of the algorithm, an initial version of the language specification was obtained. This version of the specification needed to be refined as some aspects of it were underspecified. This was done using conventional grammar engineering principles to modify the specification such that it can parse the entire codebase. After the codebase was able to be parsed, workshops were

held with developers versed in the language to be able to shed more light on the peculiarities of the language. From these workshops, the language specification was further improved, with one of the most important improvements being the ability to define the correct bindings for the language specification.

The parser generated using the specification obtained from the algorithm is a custom solution which utilises a nested finite state machine to do the parsing. The parser goes through the input line-by-line. Because the parser knows exactly where to expect what, no scanning needs to be done. Substrings at expected index ranges are compared to the expected content. For string literals, a substring with equal is compared to the expected literal. For patterns, the substring is checked to see if it adheres to the commitment, after which the binding takes place.

The author mentions that this approach was employed successfully for creating a parser for the language, which later became part of TIALAA [112], a reimplementation of AppBuilder.

3.3.4 A Toolbox for Context-Sensitive Grammar Induction by Genetic Search

In this paper, Wieczorek et al. introduce a grammatical inference library for context-sensitive languages that can infer grammars in polynomial time. The library is written for the .NET Framework and is written in F# [154]. Even though this chapter is mostly concerned with the inference of context-free languages, this paper is still relevant as context-free languages can be expressed as context-sensitive languages.

The algorithm used is a steady-state genetic algorithm and learns from both positive and negative samples. The biggest aspects that differentiate steady-state genetic algorithms from classical genetic algorithms are their selection policy and their deletion policy. During selection, a tournament selection method is used on a random subset of the population. Two of the fittest individuals of this subset become parents to a new entry of the population. During deletion, the least fit individual is replaced by the newly created individual. Algorithm 1 describes the general structure that is shared across steady-state genetic algorithms.

Algorithm 1 The general structure for a steady-state genetic algorithm

Input: n_i – number of iterations
n_p – the size of the population
p_m – probability of mutation
Output: highest fitness value individual in Population P

1: generate random initial population P of size n_p
2: **for** $i \leftarrow 1$ to n_i **do**
3: select $T \subseteq P$ individuals to the tournament
4: select $p_1, p_2 \in T$ with highest fitness
5: create offspring p_n based on p_1, p_2 crossover
6: execute mutation function on p_n with probability p_m
7: calculate fitness for p_n
8: replace lowest fitness $p_r \in T$ with p_n in P
9: **end for**
10: **return** an individual from P with highest fitness

The tool described in this paper takes as input a sample with positive and negative samples of the language. The goal is to return a grammar in Kuroda normal form [79] that recognises as many of the positive samples and as few of the negative samples as possible.

In the steady-state genetic algorithm shown in this paper individuals of the population are defined as integer arrays that encode grammars of the Kuroda normal form. As an example of this encoding, take a grammar with two nonterminals (S, A) and two terminals (a, b), each possible

production rule that can be constructed using these elements is associated with a number. $S \to a$ is associated with 0, $S \to b$ with 1, $A \to a$ with 2, $A \to b$ with 3 and $S \to S$ with 4. For this example, there are 31 possible production rules, with the production rule associated with the integer 31 being $AA \to AA$. Each integer in the arrays is less than $n = |N|$ and $t = |T|$.

The initial population is built randomly during the initialisation of the algorithm. This is done with the help of the input of the algorithm, which provides the algorithm with information about the expected size of the grammar, the terminal symbols, and the cardinality of the nonterminal symbols. The individuals of the population have a fixed length of k, and each entry of the array has an integer value that was uniformly sampled from the range $[0, n(t + n + n^2) + n^4]$. The nonterminals of the grammar have an index starting from 0. When the grammar gets decoded, the nonterminal with the smallest index present on the left-hand side of the production rules of the grammar becomes the starting symbol of the language.

The crossover operator in this implementation is designed as a recursive procedure. The procedure takes the two parents as input. On these parents the array operations `head` and `tail` can be performed, these operations return the first element of the array and the remainder of the array respectively. This procedure has four cases, these are illustrated using two parent arrays `x` and `y`:

- Both `x` and `y` are non-empty arrays. If `head(x)` and `head(y)` are equal, the value gets added to the result and the procedure gets called again with `tail(x)` and `tail(y)`. If `head(x)` < `head(y)` then there is a 50% chance that `head(x)` gets added to the result and the procedure gets called again with `tail(x)` and `y`. If `head(y)` < `head(x)` then there is a 50% chance that `head(y)` gets added to the result and the procedure gets called again with `x` and `tail(y)`.
- If `x` is non-empty and `y` is empty, every element of `x` has a 50% chance to be added to the result.
- If `y` is non-empty and `x` is empty, every element of `y` has a 50% chance to be added to the result.
- If both `x` and `y` are empty, an empty array is returned.

Mutation in this instance is implemented in a straightforward manner. When mutation occurs, a random value in the array is chosen and modified to an integer not yet present in the array. The fitness function is the balanced accuracy of the grammar. This is calculated using Equation 3.1, with S_+ being positive samples, S_- being the negative samples and L(G) being the language of grammar G.

$$f(G) = \frac{1}{2} \cdot \left(\frac{|\{w \in S_+ : w \in L(G)\}|}{|S_+|} + \frac{|\{w \in S_- : w \notin L(G)\}|}{|S_-|} \right) \tag{3.1}$$

In the paper, the authors show how to use the published library in several .NET languages, testing it on simple context-sensitive grammars.

3.3.5 Gramin: A System for Incremental Learning of Programming Language Grammars

Saha and Narula introduce Gramin [119], a grammatical inference algorithm that allows for the inference of grammars of languages using positive samples only. Some knowledge of the language is required as Gramin takes not only the source code as input but also an initial version of the grammar. Gramin consists of the main Gramin procedure, and two sub-procedures called Focus and Gramin-one. If the machinery gets stuck anywhere in the process, the algorithm allows for

backtracking until it can resume the inference. The paper does not describe any countermeasures to avoid overgeneralisation in the process of inferring the grammar, apart from a mention in the future work section about how an existing compiler could help with detecting overgeneralised rules introduced by the algorithm.

The iterative process Gramin uses starts with breaking up the input in statements using a lexer in a way where the frequency of occurrence of each statement is kept track of. Then the algorithm attempts to parse the statements using the initial grammar, statements that could not be parsed are collected in the so-called positive sample set. Then the algorithm learns the structure of each type of statement. This is done by ranking each variation of the statement encountered in the input based on frequency of appearance. The idea behind ordering the input of the learning algorithm like this is that this allows the algorithm to learn the basic structure of the statement first and variations of it later, as the basic form is expected to be encountered more frequently. For a statement that cannot be parsed, the Focus procedure is used to determine a set of nonterminal and substring pairs in which change can occur. This set is ordered in terms of the length of the substrings. The Gramin-one procedure takes a nonterminal, a substring and the current grammar and uses these to generate a set of potential rules that can parse the unparsable statement. These are ranked based on so-called goodness criteria with the highest-ranking rules incorporated in the grammar, with the rest stored for potential later use during backtracking.

The authors tested their solution by inferring the grammar for 30 ABAP statements (a 4GL covered in subsection 2.3.17). In the paper, the authors go into detail on how an original grammar, which can only parse the most basic and most frequently occurring variation of a statement, is gradually updated to parse other variations encountered in the input.

3.3.6 Towards the Automatic Evolution of Reengineering Tools

This paper by Di Penta et al. proposes the adoption of genetic algorithms for the evolution of existing grammars for a base language to a dialect grammar or language variant [33]. The authors show the effectiveness of this approach by demonstrating the inference of a C variant from a base grammar.

The semi-automatic inference process is made up of five steps: Firstly, the main differences between the target and starting language are identified. This is done by examining language references, source code and errors given by the parsing target language samples using the starting language parser. Then, for larger grammars, the authors recommend splitting the grammars up into sections. This helps with reducing the search space. Now the genetic algorithm can be run. This algorithm has a rather straightforward setup, as illustrated in Figure 3.2. After that, if the grammar was split up into sections, it should be merged back into a single grammar. Lastly, the grammar must be thoroughly tested by parsing a large test set of files from the target language.

For the grammar to be able to evolve, it must be encoded in a genome format where on which crossover and mutation can be applied. The authors suggest using a widely used approach [82] where the grammar is encoded as a matrix in which each production rule is encoded into an array of integers. The first element in this array denotes the left-hand side, and the rest of the array denotes the right-hand side. An example of this encoding can be seen in Figure 3.3.

The method proposed uses three genetic operators: selection, crossover and mutation, with crossover and mutation explained in detail. For the crossover operator, the authors use a two-point crossover, meaning the genomes of the parents are split into three chunks, with a crossover probability of 0.7. The probability of mutation is $1/m$ where m is the number of elements in the genome. Mutation can occur on elements of the rules of the genome or the genome itself. The first option modifies a production rule from the genome by switching a random element with another random one, where the first element of the array can only be switched to another nonterminal. The second option can add or remove entire production rules from the genome.

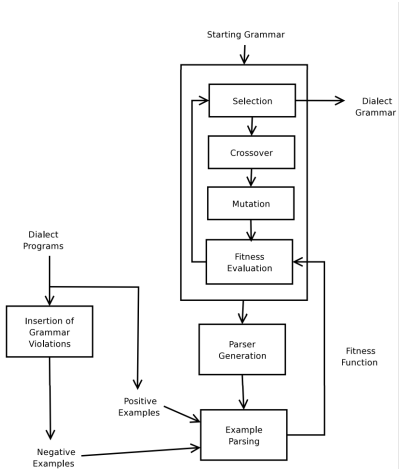

FIGURE 3.2: The General Setup of the Genetic Algorithm

Terminals	
0	a
1	b
2	c

Non-Terminals	
3	A
4	B
5	ε

	LHS	RHS		
A → bA	3	1	3	5
B → ABa	4	3	4	0
B → c	5	1	5	5

FIGURE 3.3: An Example Grammar Encoded in a Genome

Since the goal of the evolutionary process is to infer a similar language to the starting language, the fitness function is designed in a fashion where it discourages excessive deviation from the initial population (in this case, the starting grammar). To this end, the authors used the function described in Equation 3.2.

$$F = w_1 * (w_2 * cor\_positive + p\_total * w_3) * cor\_negative + w_4 * sim \tag{3.2}$$

This function aims to output a higher value if the grammar parses a sufficient percentage of positive samples($cor\_positive$) and rejects a sufficient percentage of negative samples($cor\_negative$), and when the deviation is low(sim). The output value is also raised if a grammar can parse most lines of each sample, even if it cannot parse a single complete file(p_total). The weights in this function should be chosen in a manner where the positive attributes of the grammar are rewarded proportionally. In the case studies in this paper, the authors set $w_{1...4}$ to 0.75, 0.75, 0.25, and 0.25 respectively.

For the positive samples of the input, the authors suggest picking a set of samples where the majority of the files include the language concept that is to be learned. This should increase the speed at which the algorithm is able to learn the target language.

For the selection of the negative samples of the input, the authors highlight two cases. The first case is when the grammar is a backwards-compatible extension of the source language. In this case, the negative samples could for example consist of source files with syntax errors introduced. The second case is when the two are distinct languages, in which case the negative samples could consist of programs in the source language.

The authors tested their approach in two case studies, one where a C dialect was learned from

a smaller C grammar and one where the grammar for the programming language PHP was learned using the same C grammar.

In a later work, the authors expanded on this work by introducing improvements to the genetic operators and the fitness function used in the algorithm [34]. This paper also sheds more light on the initialisation process of the algorithm, which involves applying random mutations on copies of the starting grammar to increase genetic diversity. Besides this, it further specifies that the selection operator used is specified to be the *roulette wheel* selection operator [99].

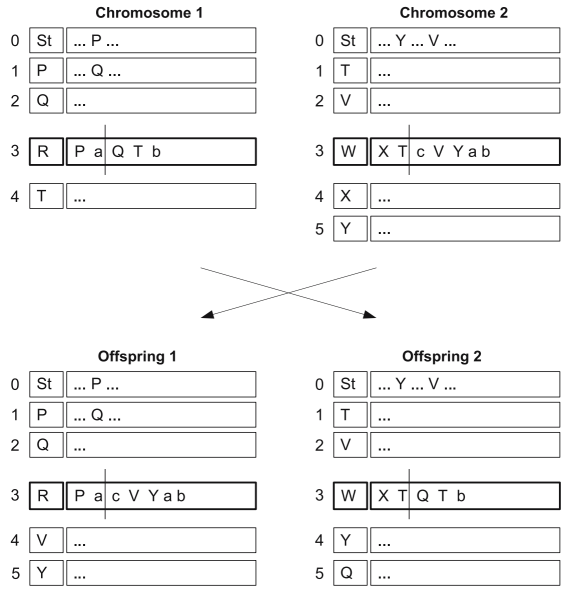

FIGURE 3.4: The Modified Crossover Operator

The two-point crossover from the first paper has been changed to a process where one rule is selected for both parents, a cut point is chosen for the selected rules and crossover is applied using this cut point as shown in Figure 3.4.

The mutation operator had been expanded and modified. Where previously rule addition/deletion and the mutation on indices of the rules in a genome were possible, this paper keeps the rule mutation on the indices and rule addition but does not allow for the deletion of entire rules. Besides these modifications, the paper introduces symbol addition and symbol deletion on the rules of a genome, which allows the insertion and deletion of random symbols within the rules of the genome.

Finally, the fitness function was modified. This newer paper no longer considers the similarity and only has two weights. These modifications to the fitness function are shown in Equation 3.3.

$$F = (w_1 * cor\_positive + p\_total * w_2) * cor\_negative \tag{3.3}$$

These changes made the algorithm more scalable than the initial version, allowing for the evolution of larger grammars with up to 70 rules while previously only being able to handle grammars up to 35 rules.

3.4 Applying the Insights Gained on the Design of the Prototype Algorithm

This chapter grants the insight required to make informed design decisions for the prototype algorithm implemented during this project.

For the general setup of the algorithm the approach of the MAGIc algorithm (described in subsection 3.3.1) is used; The initial population consists of grammars that can each parse one of the samples of the positive sample sets which were generated using the SEQUITUR algorithm [102].

The representation is largely based on the one found in the work of Di Penta et al. (described in subsection 3.3.6), where grammars in the population are encoded in a matrix representation.

The genetic operators of the prototype are mostly inspired by the ones found in the genetic algorithms discussed in this chapter, such as the single-point crossover discussed in subsection 3.3.6 and the mutation operators discussed in subsection 3.3.1.

The objectives to determine the fitness of an individual grammar in the population are inspired by all the solutions in that use an evolutionary approach. This includes the obvious objectives such as the ratio of positive samples parsed, but also the ratio of negatives rejected. The idea of weighting the objectives (which is used in the solutions described in subsection 3.3.4 and in subsection 3.3.6) has also been applied.

The exact details of the implementation can be found in chapter 5.

Chapter 4

Obtaining Positive- and Negative Samples for the Selected Languages

As was previously discussed in subsection 3.2.1, all grammatical inference methods should have two sets of input: one with positive samples that should be recognised by the language and one with negative samples that should be rejected by the language. Having both sets available during the inference helps to mitigate the inference of a grammar that is larger than the target grammar [41]. The positive sample set should contain samples that are successfully parsed by a recogniser. The negative sample set is less straightforward. This set ideally contains files that do not fail to parse outright at the first token of the token stream but are partially parsable.

4.1 Obtaining Samples from Public Code Repositories

To measure the performance of a grammatical inference algorithm in an accurate manner, it would be best to use as many real-world code samples as possible. This way a real-world environment can be replicated as best as possible while performing grammatical inference.

For the three selected languages of this project, the GitHub API [1] was used to obtain relevant samples. For the language SQLite, the API was used to scrape SQL samples which were filtered through a recogniser to separate samples that were not compatible with the exact dialect of SQLite, these samples are kept for the negative sample set. Since Oberon-0 and BabyCobol are not general-purpose programming languages but rather languages that are meant to practice compiler construction with, gathering a large set of real-world samples in these languages is less simple than with SQLite.

The GitHub API does not allow searching for Oberon-0 and BabyCobol code directly since these languages are relatively obscure and are not used for everyday programming. The API does however allow for searching samples of related languages that are more commonplace. While samples of related languages might not all be recognised by the parser, they can be good candidates for negative samples as the recogniser might parse at least a portion of the file due to the similarities between the languages. For Oberon-0 the samples of the languages Modula-2 [157] and Modula-3 [24] were collected, which are closely related languages to Oberon-0. For BabyCobol, samples of COBOL were collected, since this is the main inspiration of BabyCobol.

While there are no production-grade codebases available for Oberon-0 and BabyCobol, there are, however, multiple implementations of both languages that include test cases. The compiler test cases of Oberon-0 were sourced from implementations from GitHub [16, 75, 115, 117, 134, 150] and the test case bundled with the book Compiler Construction [159] by from Niklaus Wirth, the author of Oberon-0. The compiler tests of BabyCobol were sourced from student implementations of the language for the course Software Evolution at the University of Twente [170].

Besides the compiler test cases, there are also samples available on Rosetta Code [2], a website that catalogues solutions to programming tasks written in different languages. There are a couple of programs written in BabyCobol available on Rosetta Code and there are no programs written in Oberon-0 available on the website. There are however programs available written in the related languages. Besides Modula-2 and Modula-3 there are also samples available in Oberon-2, which is another closely related language to Oberon-0. While only the simplest of these programs are compatible with Oberon-0, they still make an interesting addition to the set of negative samples.

For the three selected languages, gathering a large corpus of real or human written code proves to be challenging. This was to be expected for Oberon-0 and BabyCobol, due to the nature of these languages. For SQLite and gathering samples of languages related to Oberon-0 and BabyCobol, there exists a bottleneck of the GitHub API. The GitHub API only grants access to the first 1000 results of a search query, thus limiting the results that can be obtained this way. The next section will go over how to generate additional samples to expand the dataset in cases where a grammar and a parser are available.

4.2 Sample Generation

When faced with the problem of having smaller than ideal positive- and negative sample sets, a solution could be to complement the sets with synthetic samples. This section discusses the ways investigated and used during this project to obtain more positive- and negative samples.

4.2.1 Positive sample generation

One way to increase the size of the positive sample set is to generate new positive samples. This can be done with the help of fuzzers. Fuzzing (or fuzz testing) [101] is a method of automatic software testing where random inputs are given to a program to find bugs and vulnerabilities. For fuzzing with compilers, parsers and other programs that require structured input, it is often not useful to use completely random inputs as this most likely won't expose anything other than the most surface-level bugs. Fuzzers made for compilers and other language tools produce inputs that are in the language the compiler implements. This subsection will look at such fuzzers and report how they were utilised to expand the positive sample set.

Semantically Correct Fuzzers

Semantically correct fuzzers are tools that generate programs within the language that the system-under-test accepts and are also compilable. They are often used for practices such as differential testing [98]. This is a method of testing where two or more implementations of the same language are given the same input, and their outputs are analysed on differences they might produce.

One of the earliest of such tools, and consequently one of the most influential, is Csmith, a fuzzer for the C programming language [161,162]. It has successfully been used to find hundreds of bugs in multiple implementations of C, including the two most popular implementations GCC and Clang/LLVM. This work has inspired multiple similar projects such as Verismith [53] for Verilog, YARPgen [91] for C and C++, SQLsmith [136] for various dialects of SQL and Xsmith [51], a framework made to accelerate the implementation of fuzzers.

SQLsmith [136, 137] is of particular interest for this project. It was directly inspired by Csmith and developed as a fuzzing tool for database engines that have a SQL dialect as their query language. It has built-in support for PostgreSQL, SQLite and MonetDB and it has successfully been used to find bugs in these database engines. SQLSmith has been successfully used to expand the positive sample set for SQLite. The exact process of how these samples were generated will be described in subsection 4.3.3.

Xsmith [51, 52] is a Racket library that facilitates mechanisms and reusable infrastructure for implementing fuzzers more rapidly. Xsmith lets users define aspects of the language such as syntax, types, and semantics using a DSL defined within the Racket language to guide the generation. The user-defined input gets converted to a RACR attribute grammar definition [22], which is used as input for the generation process. The authors have implemented several fuzzers for Racket, Dafny, Standard ML and WebAssembly and have found bugs in implementations of each of these languages. It was originally planned to use Xsmith to implement fuzzers for both Oberon-0 and BabyCobol. The outcome of this attempt will be discussed in subsection subsection 4.3.3.

Grammar Based Fuzzers

Grammar-based fuzzers refer to fuzzers that use the rules defined in the context-free grammar of a language to generate abstract syntax trees within the language. While these samples may be recognizable by a parser, they most likely will not compile since the generation process does not consider the semantics of the language. For example, consider the following grammar for a simple calculator.

```
1 expression = term  { ("+" | "-") term} .
2 term       = constant  { ("*"|"/") constant} .
3 constant   = digit {digit} .
4 digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

LISTING 4.1: The Grammar for a Simple Calculator

When generating purely from these rules, one possible output could be `"99/0"`, which is not something that the calculator could solve.

This method of sentence generation has been around since the 1970s, where Purdom gives an algorithm that generates a small set of sentences such that each production rule is used at least once [109]. Here, it is aimed to discuss more recent works that were encountered and used during this project that utilise an (E)BNF or similar input for sample generation.

FuzzPEG [77] is a grammar-based fuzzer that accepts parsing expression grammars (or PEGs) as its input. PEGs are a type of grammar that aim to be an alternative to context-free grammars, specifically defined for expressing machine languages [38]. One of the major differences between PEGs and context-free grammars is that PEGs do not allow for ambiguity. FuzzPEG uses a PEG implementation in Java called j-PEG [78], both were developed by the same author. While j-PEG and FuzzPeg look promising, the source code of these projects has not been maintained since 2021.

Yagg [28, 29] is a tool that can generate structured inputs for programs when given a grammar. It borrows heavily from Lex [83] for its terminal definition and from Yacc [74] for its grammar definition. While the examples in the paper and on the git repository are not fully fledged programming languages, programming languages defined in Yacc and Lex could be modified so they conform to the input that Yagg accepts. Yagg was first introduced in 2004 and has been last maintained in 2018.

Grammarinator [54, 55] is a grammar based fuzzer that accepts ANTLRv4 [107] grammars as its input. This tool is highly configurable, for example, Grammarinator allows both samples to be generated from scratch and to be generated from existing samples. When generating from scratch, Grammarinator uses the grammar to generate a valid abstract syntax tree (AST). When generating using existing samples, Grammarinator uses the genetic operators recombination (also

called crossover) and mutation. With recombination, branches of the AST get replaced with a valid branch originating from another sample. With mutation, a branch of the AST gets removed and replaced with a generated branch. Besides this, Grammarinator allows for the configuration of the maximum depth of the ASTs that are generated. Of the tools encountered, Grammarinator seems to be the best maintained and documented, with its latest commit being from December 2023.

4.2.2 Negative sample generation

The negative sample set can also be increased through generation. This process is less straightforward than expanding the positive sample set since, in that case, fuzzers to generate samples that fall in the target language could be used. For the generation of negative samples such projects are less available.

There are several points to consider while making a tool for the generation of negative samples for a given grammar.

The input must not be completely random. While completely random inputs might almost always fail to parse, they do not provide sufficient information about the language to be inferred. They would most likely fail on the first token, giving the algorithm little to work with when inferring what is and is not allowed in the rest of the language.

Meaningful negative samples are samples that fail somewhere down the line of the parsing process. Preferably, a negative sample set covers as much of the language as possible in this way.

Modifying the Grammar Operators

One way to expand the negative sample set is to modify the grammar and use a grammar-based fuzzer to generate new samples using the modified grammar. This can be done by modifying the operators within the grammar definition. For this method, the terminology introduced by Zaytsev [165] will be considered.

With the original grammar G,a modified grammar G' can be created by preforming the following modifications:

- Widening (changing occurrences of x^+ to $x^?$)
- Widening (changing occurrences of $x^?$ to x^*)
- Narrowing (changing occurrences of x^* to x^+)

A downside of generating negative samples in this manner is the fact that not 100% of the generated samples are guaranteed to be negative. Another fact that challenges this method of negative sample generation is that the samples will fail only on positions where the operators are modified since the rest of the G' is identical to G.

Word Mutation

A recent paper by Raselimo, Taljaard and Fischer [114] explores the mutation-based generation of programs with guaranteed syntax errors. This paper showcases two methods with which negative samples can be reliably obtained, namely word mutation and rule mutation. This subsection specifically focuses on word mutation. Before this method can be laid out in detail, the concept of poisoned pairs PP(G) from the paper will be explained.

For a grammar $G = (N, T, P, S)$ the precede and follow sets are defined as precede$(X) = \{Y | S \Rightarrow^* \alpha Y X \beta\}$ and follow$(X) = \{Y | S \Rightarrow^* \alpha X Y \beta\}$ for a symbol X. (X, Y) are considered a *poisoned pair* iff $X \notin$ precede(Y) or iff $Y \notin$ follow(X). PP(G) denotes all the poisoned pairs found in G.

Using these poisoned pairs, the authors make the following proposition, which contains the same cases used to calculate the Damerau-Levenshtein distance between two strings [30, 87].

Propisition 1 *(Damerau-Levenshtein mutations). Let G be a grammar. Then:*

1. *(token deletion) If* $uabcv \in \mathcal{L}(G)$ *and* $(a, c) \in PP(G)$*, then* $uacv \notin \mathcal{L}(G)$.
2. *(token insertion) If* $uacv \in \mathcal{L}(G)$*,* $b \in T$*, and either* $(a, b) \in PP(G)$ *or* $(b, c) \in PP(G)$*, then* $uabcv \notin \mathcal{L}(G)$.
3. *(token substitution) If* $uabcv \in \mathcal{L}(G)$*,* $d \in T$*, and either* $(a, d) \in PP(G)$ *or* $(d, c) \in PP(G)$*, then* $uadcv \notin \mathcal{L}(G)$.
4. *(token transposition) If* $uabcdv \in \mathcal{L}(G)$ *and either,* $(a, c) \in PP(G)$ *or* $(c, b) \in PP(G)$ *or* $(b, d) \in PP(G)$*, then* $uacbdv \notin \mathcal{L}(G)$.

With the mutation operators defined in this predicate, a negative sample set can be computed given a positive sample set, in other words, the Damerau-Levenshtein mutants set of the positive sample set. The exact implementation and use of the word mutation strategy will be detailed in subsection 4.3.2.

4.3 The Resulting Sample Sets

This section aims to highlight how the sample sets were gathered, generated, and sorted. For each of the three languages, an ANTLRv4-based recognizer was created to help distinguish positive samples from negative ones. The grammar for Oberon-0 was sourced from one of the implementations that were found earlier [115], the grammar for SQLite was sourced from a repository [10] that keeps a collection of ANTLR grammars, and the BabyCobol grammar was created by combining pieces of the student submissions for Software Evolution such that the resulting grammar covers as much of the language as possible within the ANTLR specification.

First, the process is shown of how the found positive samples were filtered and how the positive sample sets were expanded. Then the process of how the negative sample sets were expanded using the modified grammars, and how a tool for word mutation was implemented and utilised for generating negative samples, will be described.

4.3.1 The Positive Sample Set

The positive sample sets were constructed using both real-world samples and generated ones. The real-world samples were gathered as described in section 4.1. After this, the sample sets for each language were filtered through the recognizers to distinguish between the positive samples and the negative ones. This process was performed on the found samples of each language.

Then the positive sample sets were supplemented with generated samples. It was initially decided to use semantically correct fuzzers to generate additional positive samples to expand the positive sample sets.

For SQLite, the SQLite generator of SQLsmith was used. Three databases were used as input during the generation; one that models a car company [149], one that models a video rental

store [42] and one that models the employee system of a company [116]. In total 333.333 queries per dataset were generated, adding up to a total of 999.999 queries generated with SQLsmith.

For Oberon-0 and BabyCobol no existing semantically correct fuzzers were available so it was attempted to develop fuzzers for these languages using Xsmith. This attempt started with creating a fuzzer for Oberon-0. Many of the resources Xsmith offers were utilised. This includes using sample fuzzers as a base to work from, and the so-called canned components, which implement reusable generic structures and concepts across many programming languages such as loops and datatypes. This effort was eventually abandoned after several weeks, as it became increasingly evident that the infrastructure that Xsmith provides is not optimal for 4GL(-like languages). Most of the aspects of Xsmith that aim to accelerate the development of fuzzers are more appropriate for functional languages like Haskell, Scheme-like languages like Racket, and procedural languages like C, JavaScript, and Python.

After the attempt to create semantically correct fuzzers with Xsmith, a pivot was made towards using Grammarinator. This choice was made since Grammarinator seems to be the most well-maintained grammar-based fuzzer encountered during this project. Grammarinator accepting ANTLRv4 grammars as input meant this switch was relatively painless.

By default, Grammarinator will write the tokens of the generated AST as a single line with no spaces between the tokens. There are several ways to gain better-formatted outputs. One would be to either use the default serializer that puts a space between each token or write a custom one. The other option is to modify the grammar in a way that makes the positions of whitespaces explicit. For Oberon-0 and BabyCobol, the second option was chosen.

Then the fuzzers were generated using the modified grammars as input. These Grammarinator fuzzers were then used to generate the positive samples. For both languages, 1.000.000 samples with a maximum AST depth of 10, and 1.000.000 samples with a maximum AST depth of 20 were generated. Besides these samples, another 500.000 were generated for both languages using the recombination method of generation, where the ASTs of the compiler tests were used as input.

4.3.2 The Negative Sample Set

As with the positive sample sets, the negative sample sets were constructed using both real-world samples and generated ones. The real-world samples that failed to parse using the recognizer were carried over to the negative sample set. After this, the negative sample set was expanded using generation. Both methods of negative sample generation discussed in subsection 4.2.2 were utilised to expand the negative sample sets. In the first method described in section 4.2.2, the following operations were discussed.

- Widening (changing occurrences of x^+ to $x^?$)
- Widening (changing occurrences of $x^?$ to x^*)
- Narrowing (changing occurrences of x^* to x^+)

These were applied to all grammars of the languages and Grammarinator was used to generate 100.000 samples using these modified grammars. Not all of these were guaranteed to fail because of the nature of the operations, therefore the passing tests were filtered out using the recognizer.

Besides the samples generated from the modified grammars, a tool was built to generate negative samples in a more systematic manner, based on the word mutation method described in section 4.2.2. For this, the large positive sample sets generated earlier were used. The tool takes the positive sample set of a language and its lexer as input. Then it goes through the positive samples and keeps track of which type of tokens can come before and after each token type. With these (estimated) precede- and follow sets the (estimated) poisoned pairs for each token type can be determined.

The process of generating negative samples as defined in proposition 1 on the entire positive sample sets would lead to an explosive number of tests since for each token in each file multiple mutants can be generated. To mitigate this, a process of selecting the smallest subset from the positive samples has been defined. Using the precede- and follow sets of each token type, the tool goes over the positive sample set again in ascending order based on file size. The preceding token type of each token is checked and, if it is encountered for the first time, the token type is removed from the precede set and the current file is selected. This is done similarly for the following token and the follow set. This process is repeated until all precede- and follow sets are emptied, resulting in a selection of samples that contains all instances of tokens with a specific type of preceding or following token at least once.

Then the word mutation process is performed with the poisoned pairs and the subset of positive samples. This process yields a set of samples that are presumed to be negative. This presumption being true however completely relies on the coverage of the positive sample set. If it is sufficiently covered the output should only contain files that fail to parse. In practice, it was encountered that the output contained a subset that succeeded parsing. This was an indication that the set of poisoned pairs was too broad, caused by gaps in the coverage in the positive sample set. These false negatives were carried over to the positive sample set and the process was repeated. After these additions to the positive sample set, the samples that were put out were confirmed to be all negative.

Figure 4.1 gives an overview of the entire process.

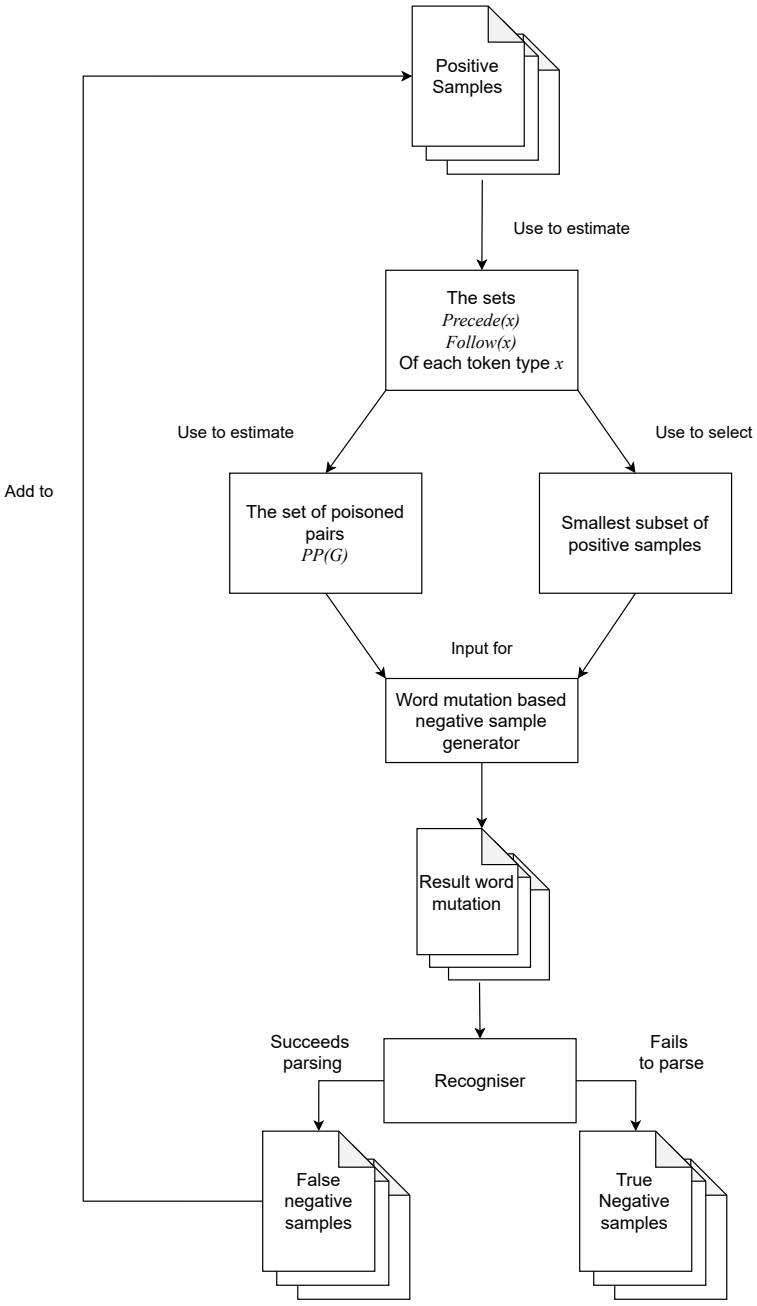

FIGURE 4.1: Overview of the Negative Sample Generation Process

4.3.3 Overview Total Dataset

Language	Unique Positive Samples	Unique Negative Samples
Oberon-0	2.014.809	396.016
BabyCobol	1.737.416	10.114.144
SQLite	1.961.471	73.020.229[1]

TABLE 4.1: The Cardinality of the Positive- and Negative Sample Sets

The final dataset obtained through all the methods discussed in this chapter can be found in Table 4.1. A couple of observations were made while gathering these sample sets.

Since SQLsmith is developed to find bugs in implementations of SQL, it was not entirely surprising that not each input was deemed parsable by the recogniser. For this project, only samples that fit the dialect of the SQLite grammar used to generate the parser are of interest, so samples that failed to parse were separated from the samples that succeeded to parse. This left 907.237 of the 999.999 generated samples in the positive sample set for SQLite. To fill this gap of lost samples, Grammarinator was utilised again to generate an additional 100.000 samples.

For all three languages, a large portion of the samples that were obtained through the recombination feature of Grammarinator were filtered. This was done because since the input set of ASTs from the real-world samples is rather small, the generation algorithm of Grammarinator sometimes puts out the same sample multiple times.

Besides these duplicates, a sizeable portion of the negative samples generated through word mutation were also filtered. These duplicates were occurring due to the nature of the rules in proposition 1, causing the word mutation tool to generate the same sample multiple times as a consequence of overlap between the rules in the predicate. In the case of BabyCobol, the samples generated from word mutation were reduced by nearly a third, from around 15 million samples to 10 million samples.

[1] This is a projected result from a run that exhausted the available disk space before it could terminate. Around 75% of the word mutation input was consumed and yielded 82.147.758 negative samples. When accounting for the remaining 25% of the input data it can be estimated that a completed run would yield 109.530.344 samples. Then when considering that around 2/3 of this yield is unique, the final estimate becomes 73.020.229 unique negative samples.

Chapter 5

Prototype Implementation of a Grammatical Inference Algorithm

This chapter contains the description of the prototype grammatical inference algorithm that was implemented during this project. The overall design and goal of the algorithm are described in section 5.1, the representation of the chromosomes in the population is described in section 5.2, the objectives of the algorithm are described in section 5.3, the genetic operators that were used in this implementation explained in section 5.4, and finally, the results of running the algorithm to infer grammars for Oberon-0 and DESK will be laid out in section 5.5.

5.1 Setup of the Implementation

This prototype was implemented using the MOEA (multiobjective evolutionary algorithms) framework [45]. This framework was chosen because it offers an extensible API, has generic implementations of many popular evolutionary algorithms, and supports both single objective and multiobjective problems.

In this experiment, the inference of grammars is approached both as a single objective problem (where all objectives are encoded in a weighted fitness function) and as a multiobjective Pareto-optimisation problem (where every objective has the same weight).

For the single objective approach, an implementation of a simple genetic algorithm was used. The multiobjective approach uses an implementation of the NSGA-II algorithm [31], a popular multiobjective algorithm that returns a set of Pareto-optimal solutions to a multiobjective problem.

The algorithm takes similar input as the MAGIc algorithm described in subsection 3.3.1. The input samples need to be modified such that instances of non-keyword terminals have been replaced by their terminal symbols. For example, an instance of an integer becomes `INT`, and an instance of a string becomes a `STR`.

To obtain samples in this form, the tools from chapter 4 were utilised once again with a modified grammar to generate positive- and negative samples. The input grammars were modified such that the terminals that were defined by a regular expression were changed to a string literal of their name. For example `ID : [a-z]+;` becomes `ID : 'ID';`, and `NUM :[0-9]+;` becomes `NUM : 'NUM';`.

There are two languages that were used for this experiment; DESK [105] and Oberon-0 [159]. DESK is a small language that allows for easier debugging of the algorithm during its development. DESK was also used by the authors of MAGIc for this purpose. Oberon-0 is one of the selected languages from section 2.5.

Since parser generation and parsing can be some relatively computationally heavy tasks depending on the size of the grammar, it is not wise to use the entire positive dataset. Therefore a

subset of the positive sample set is needed. The input of the word mutation process described in section 4.2.2 is used for this purpose since this set of samples has reasonably broad coverage of the language due to the way these samples are selected. To construct the input subset of positive samples, 50 of the closest files based on the Damerau-Levenshtein [30, 87] distance are selected from the entire positive sample for each file in the word mutation input. Table Table 5.1 shows the size of the sample sets generated for use with the prototype.

Language	Subset Positive Samples	Unique Negative Samples
DESK	400	496
Oberon-0	8000	142.434

TABLE 5.1: The Cardinality of the Positive- and Negative Sample Sets That Are Used with the Prototype

5.2 The Chromosome Representation

Initially, a grammar is created for each of the samples from the positive input. This is done using a SEQUITUR implementation [138] which generates an instance of the Java type `TreeMap<String, String>` for an input. In this `TreeMap` the key represents the rule name and the value represents the rule definition.

This implementation has been modified in such a way the generated output can be easily converted into an ANTLRv4 [107] grammar. This grammar can be used to generate a parser instance which can be used to evaluate the grammar.

As an illustration of this process, consider the following DESK sample:

```
print NUM + ID + NUM where ID = NUM ; ID = NUM
```

When using this as an input for the modified SEQUITUR implementation, the following `TreeMap` is returned:

Key	Value
r0	print NUM + ID + NUM where r1 ; r1
r1	ID = NUM

This representation can then be converted to an ANTLR grammar:

```
grammar desk;
r0: 'print' 'NUM' '+' 'ID' '+' 'NUM' 'where' r1 ';' r1 EOF;
r1: 'ID' '=' 'NUM';
```

5.3 Evaluating the Chromosomes

The fitness of the chromosomes is based on five objectives, which can be found in Table 5.2.

Objective	Goal
Ratio parsed positive samples	Maximise
Ratio semi-parsed positive samples	Maximise
Ratio parsed negative samples	Minimise
Average length production rules	Minimise
Number of production rules in grammar	Minimise

TABLE 5.2: The Objectives of the Algorithm

When tackling the inference as a multiobjective Pareto-optimisation problem, all objectives have an equal weight. This is the case when NSGA-II is used as the algorithm.

Some of these objectives could arguably be viewed as more important than others. For example, a high ratio of semi-parsed positive samples, where a number of the tokens of an input were not recognised but the parser managed to create a valid parse tree with the tokens it had recognised, is less impactful for the viability of an individual grammar than a high ratio of parsed positive samples. When taking this into consideration, a weighted fitness function is desired.

To define such a fitness function function, the objectives that are not a ratio value need to be normalised. To do this, the largest grammar of the ANTLR in the "grammars-v4" repository [10] was analysed to estimate a maximum for the number of rules and a maximum for the length of the rules in a grammar. From this analysis, it was observed that the maximum number of rules equals 1293, and the maximum length of these rules equals 1750. With this information, the following normalisation functions can be defined:

$$normalised\_avg\_length\_rules = \frac{avg\_length\_rules - 1}{1750} \tag{5.1}$$

$$normalised\_number\_of\_rules = \frac{number\_of\_rules - 1}{1293} \tag{5.2}$$

With this normalised value, the following fitness function is defined:

$$\begin{aligned} Fitness &= w_1 * parsed\_positive\_samples + w_2 * semi\_parsed\_positive\_samples \\ &+ w_3 * (1 - parsed\_negative\_samples) + w_4 * (1 - normalised\_avg\_length\_rules) \\ &+ w_5 * (1 - normalised\_number\_of\_rules) \end{aligned} \tag{5.3}$$

Where $w_1 + w_2 + w_3 + w_4 + w_5 = 1$.

The values used in the prototype are laid out in Table 5.3. These values are chosen based on the relative importance of each objective to the fitness of a grammar.

Weight	Value
w_1	0.5
w_2	0.15
w_3	0.25
w_4	0.05
w_5	0.05

TABLE 5.3: The Weights for the Fitness Function

5.4 Genetic Operators

This section goes over the implemented genetic operators and how they process their input when they are invoked.

5.4.1 Crossover

The crossover operator implemented in this project is a single-point crossover. This operator is similar to the crossover operator defined by Di Penta et al. [33, 34], shown in Figure 3.4 in subsection 3.3.6.

Consider the following two DESK grammars that will be used to demonstrate this operator:

```
r0 -> 'print' r1 '+' r1
r1 -> 'ID' '+' 'NUM'
```

```
r0 -> 'print' r1 r1 'ID'
r1 -> 'NUM' '+'
```

For both input grammars a random rule is chosen cross over with each other. In this example `r0` of both grammars is chosen:

```
r0 -> 'print' r1 '+' r1
```

```
r0 -> 'print' r1 r1 'ID'
```

After choosing the rules to cross over, a cut point is defined. This is determined by picking a random point in the shortest of the two rules. For this example, the cut point is defined after the second symbol of the rules. This is illustrated by the characters "<|>".

```
r0 -> 'print' r1 <|> '+' r1
```

```
r0 -> 'print' r1 <|> r1 'ID'
```

When the rules are crossed over the corresponding rules of any non-terminals that present after the cut point are also carried over. This gives the following output grammars:

```
r0 -> 'print' r1 r2 'ID'
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

```
r0 -> 'print' r1 '+' r2
r1 -> 'NUM' '+'
r2 -> 'ID' '+' 'NUM'
```

5.4.2 Mutation (Introduction of EBNF Operators)

This mutation operator of the prototype is based on the one found in the MAGIc algorithm [57,59]. This operator adds the EBNF operators `?` (0 or 1 time), `*` (0 or more times) and `+` (1 or more times) to an input grammar. The position where these operators are inserted is determined by so-called near-misses, or positive samples that fail to parse and have exactly one parser error and exactly zero lexer errors. The parser output is examined to determine where the parser error has occurred and, based on this, what the last legal token of the input is.

The mutation operator then creates three mutant grammars, one with each of the EBNF operators injected next to the last legal token. This process has two cases; the first case is when the last legal token is a terminal symbol of the root rule `r0`. In this case, the EBNF operators are added to this terminal in the rule. The second case is when the last legal token is part of any other rule in the grammar. In this case, the EBNF operators get added next to the corresponding non-terminal token of the parent rule in the mutant grammars. In the case that a terminal or non-terminal symbol is matched more times by the parser than they appear in the rule, the mutation operator falls back on the last occurrence of that symbol.

As an instance of the first case, consider the following grammar:

```
r0 -> 'print' r1 r2 'ID'
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

Assuming a near miss that considers `'ID'` in `r0` to the last legal token, the mutation operator would produce the following three mutant grammars:

```
r0 -> 'print' r1 r2 ('ID')?
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

```
r0 -> 'print' r1 r2 ('ID')*
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

```
r0 -> 'print' r1 r2 ('ID')+
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

As an illustration of the second case, consider the same input grammar but with a near-miss that considers the `'+'` of `r2` to be the last legal token. With this input grammar and near-miss, the mutation operator would produce the following three mutant grammars:

```
r0 -> 'print' r1 (r2)? 'ID'
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

```
r0 -> 'print' r1 (r2)* 'ID'
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

```
r0 -> 'print' r1 (r2)+ 'ID'
r1 -> 'ID' '+' 'NUM'
r2 -> 'NUM' '+'
```

5.4.3 Mutation (Using Repair)

This mutation operator aims to mutate grammars by generating grammar rules for an unrecognised substring of a failing positive sample and inserting these rules into the grammar, resulting in a repaired grammar. The operator determines the first- and last syntax errors in a failing sample to determine a substring. This substring is then fed into the same modified SEQUITUR implementation used during the initialisation of the population of the algorithm. The resulting rules are then placed between the legal tokens before and after the substring.

The following example illustrates this process, consider the following grammar and failing positive sample:

```
r0 -> 'print' r1
r1 -> 'ID' r2
r2 -> '+' 'ID'
```

```
print ID + NUM where NUM = ID
```

First, the operator would determine the failing substring. In this case, the failing substring is:

```
NUM where NUM =
```

Generating a grammar for this substring would yield:

```
r0 -> 'NUM' 'where' 'NUM' '='
```

Then this grammar is merged into input grammar. Before merging the two grammars the rules in the substring grammars are renamed to avoid overwriting existing rules. This results in the following repaired grammar:

```
r0 -> 'print' r1
r1 -> 'ID' r2
r2 -> '+' (r3)? 'ID'
r3 -> 'NUM' 'where' 'NUM' '='
```

Note that the start rule of the substring grammar is placed between the legal tokens before and after the substring in the original grammar. This rule is also made optional, this allows the grammar to not reject any samples that were previously recognised.

5.5 Results

This section contains the results that were obtained when using the prototype algorithm to infer a DESK grammar and an Oberon-0 grammar. In both cases the algorithm was run for 10 generations with a population size of 400 grammars, both as a Pareto-optimisation problem using NSGA-II and as a single objective problem using a weighted fitness function, using different combinations of operators. In the case of DESK, the entire dataset shown in Table 5.1 was used for the run. For Oberon-0, a subset of 400 positive samples and 1400 negative samples were randomly selected from the sample sets in Table 5.1. This was done because using a population of 8000 proved to be too large for the prototype to terminate within a reasonable time.

5.5.1 DESK

Multiobjective Approach

The following tables contain the Pareto-optimal solutions after ten generations using NSGA-II to infer a DESK grammar with different combinations of genetic operators. The operators used are listed in the table captions. In the tables, higher values in the columns “Ratio Positive Parsed” and “Ratio Positive Semi-Parsed” indicate a better solution. In the columns “Ratio Negative Parsed”, “Average Length of Rules” and "No. of Rules" lower values indicate a better solution. The columns “Ratio Positive Parsed” and “Ratio Negative Parsed” are the best indicators for the feasibility of a grammar, therefore these values have been emphasized in bold.

In some cases, the returned Pareto fronts exceeded ten elements. In these cases, the Pareto front was truncated to the ten most feasible grammars in the output.

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.162500**	0.162500	**0.000000**	4.000000	2.000000
2	**0.157500**	0.157500	**0.000000**	3.000000	3.000000
3	**0.085000**	0.085000	**0.000000**	3.000000	2.000000
4	**0.070000**	0.200000	**0.000000**	3.000000	2.000000
5	**0.070000**	0.270000	**0.012097**	3.000000	2.000000
6	**0.055000**	0.055000	**0.000000**	2.666667	3.000000
7	**0.042500**	0.062500	**0.026210**	4.000000	1.000000
8	**0.042500**	0.042500	**0.012097**	4.000000	1.000000
9	**0.030000**	0.030000	**0.004032**	8.000000	1.000000
10	**0.030000**	0.050000	**0.014113**	4.000000	1.000000

TABLE 5.4: The Resulting Pareto-Front When Using the Mutation (EBNF Operators) Operator to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.020000**	0.020000	**0.000000**	6.000000	1.000000
2	**0.020000**	0.020000	**0.000000**	3.250000	4.000000
3	**0.020000**	0.020000	**0.000000**	3.500000	2.000000
4	**0.017500**	0.017500	**0.000000**	2.666667	3.000000
5	**0.017500**	0.035000	**0.000000**	4.000000	1.000000
6	**0.017500**	0.025000	**0.000000**	3.000000	2.000000
7	**0.012500**	0.292500	**0.000000**	2.000000	1.000000

TABLE 5.5: The Resulting Pareto-Front When Using the Mutation (Repair) Operator to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.020000**	0.020000	**0.000000**	6.000000	1.000000
2	**0.020000**	0.020000	**0.000000**	2.500000	4.000000
3	**0.020000**	0.020000	**0.000000**	2.200000	10.000000
4	**0.020000**	0.020000	**0.000000**	3.500000	2.000000
5	**0.020000**	0.020000	**0.000000**	2.111111	18.000000
6	**0.020000**	0.020000	**0.000000**	3.000000	3.000000
7	**0.020000**	0.020000	**0.000000**	2.400000	5.000000
8	**0.020000**	0.020000	**0.000000**	2.333333	6.000000
9	**0.020000**	0.020000	**0.000000**	2.285714	7.000000
10	**0.017500**	0.017500	**0.000000**	2.051282	39.000000

TABLE 5.6: The Resulting Pareto-Front When Using the Crossover Operator to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.232500**	0.232500	**0.032258**	3.500000	4.000000
2	**0.210000**	0.210000	**0.002016**	3.000000	5.000000
3	**0.162500**	0.162500	**0.002016**	3.500000	4.000000
4	**0.122500**	0.122500	**0.000000**	3.000000	5.000000
5	**0.115000**	0.115000	**0.000000**	3.250000	4.000000
6	**0.085000**	0.085000	**0.010081**	3.000000	4.000000
7	**0.082500**	0.082500	**0.018145**	4.000000	3.000000
8	**0.075000**	0.075000	**0.006048**	3.000000	4.000000
9	**0.072500**	0.072500	**0.002016**	3.333333	3.000000
10	**0.072500**	0.072500	**0.004032**	3.000000	4.000000

TABLE 5.7: The Resulting Pareto-Front When Using the Mutation (Repair) and Mutation (EBNF Operators) Operators to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.100000**	0.100000	**0.000000**	2.200000	10.000000
2	**0.100000**	0.100000	**0.000000**	2.600000	5.000000
3	**0.100000**	0.100000	**0.000000**	2.444444	9.000000
4	**0.095000**	0.095000	**0.000000**	2.166667	12.000000
5	**0.095000**	0.095000	**0.000000**	2.133333	15.000000
6	**0.095000**	0.095000	**0.000000**	2.500000	4.000000
7	**0.095000**	0.095000	**0.000000**	2.250000	8.000000
8	**0.095000**	0.095000	**0.000000**	2.222222	9.000000
9	**0.082500**	0.082500	**0.000000**	2.066667	30.000000
10	**0.082500**	0.082500	**0.000000**	2.024691	81.000000

TABLE 5.8: The Resulting Pareto-Front When Using the Crossover and Mutation (EBNF Operators) Operators to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.035000**	0.035000	**0.000000**	4.500000	2.000000
2	**0.032500**	0.032500	**0.000000**	3.090909	11.000000
3	**0.020000**	0.020000	**0.000000**	6.000000	1.000000
4	**0.020000**	0.020000	**0.008065**	2.333333	6.000000
5	**0.020000**	0.020000	**0.002016**	3.333333	3.000000
6	**0.020000**	0.020000	**0.000000**	2.000000	17.000000
7	**0.020000**	0.020000	**0.000000**	2.625000	8.000000
8	**0.020000**	0.020000	**0.000000**	3.500000	2.000000
9	**0.020000**	0.020000	**0.000000**	3.000000	4.000000
10	**0.020000**	0.020000	**0.000000**	2.600000	10.000000

TABLE 5.9: The Resulting Pareto-Front When Using the Crossover and Mutation (Repair) Operators to Infer DESK Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.100000**	0.100000	**0.000000**	2.090909	44.000000
2	**0.075000**	0.075000	**0.000000**	2.363636	11.000000
3	**0.075000**	0.075000	**0.000000**	2.076923	13.000000
4	**0.072500**	0.072500	**0.010081**	1.777778	45.000000
5	**0.072500**	0.072500	**0.008065**	1.781250	32.000000
6	**0.072500**	0.072500	**0.004032**	1.964286	28.000000
7	**0.072500**	0.072500	**0.008065**	2.222222	9.000000
8	**0.072500**	0.072500	**0.002016**	2.250000	8.000000
9	**0.072500**	0.072500	**0.006048**	1.900000	10.000000
10	**0.072500**	0.072500	**0.004032**	2.052632	19.000000

TABLE 5.10: The Resulting Pareto-Front When Using the Crossover, Mutation (EBNF Operators) and Mutation (Repair) Operators to Infer DESK Grammars

Single Objective Approach

The following table shows the fitness of optimal solutions after ten generations using a weighted fitness function. Higher fitness values indicate better solutions, with 1 indicating a perfect solution and 0 indicating the least feasible solution. The operator combination yielding the highest fitness solution for DESK is emphasized in bold.

Operators Used	Fitness Optimal Solution
Mutation (EBNF operators)	0.484963
Mutation (Repair)	0.400096
Crossover	0.400096
Mutation (Repair) and Mutation (EBNF operators)	**0.544711**
Crossover and Mutation (EBNF operators)	0.400345
Crossover and Mutation (Repair)	0.400096
Crossover, Mutation (Repair) and Mutation (EBNF operators)	0.415695

TABLE 5.11: Fitness of Optimal Solutions After Ten Generations Using a Weighted Fitness Function to Infer a DESK Grammar.

5.5.2 Oberon-0

Multiobjective Approach

The following tables contain the Pareto-optimal solutions after ten generations using NSGA-II to infer an Oberon-0 grammar with different combinations of genetic operators. The operators used are listed in the table captions. In the tables, higher values in the columns “Ratio Positive Parsed” and “Ratio Positive Semi-Parsed” indicate a better solution. For the columns “Ratio Negative Parsed”, “Average Length of Rules” and "No. of Rules" lower values indicate a better solution. The columns “Ratio Positive Parsed” and “Ratio Negative Parsed” are the best indicators for the feasibility of a grammar, therefore these values have been emphasized in bold.

In some cases, the returned Pareto fronts exceeded ten elements. In these cases, the Pareto front was truncated to the ten most feasible grammars in the output.

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.030000**	0.097500	**0.002092**	13.000000	1.000000
2	**0.025000**	0.132500	**0.001395**	15.000000	1.000000
3	**0.020000**	0.065000	**0.000000**	15.000000	1.000000
4	**0.020000**	0.042500	**0.000697**	13.000000	1.000000
5	**0.020000**	0.050000	**0.001395**	13.000000	1.000000
6	**0.015000**	0.182500	**0.005579**	11.000000	1.000000
7	**0.015000**	0.152500	**0.002092**	11.000000	1.000000
8	**0.015000**	0.170000	**0.004184**	11.000000	1.000000
9	**0.015000**	0.165000	**0.003487**	11.000000	1.000000
10	**0.015000**	0.132500	**0.000000**	11.000000	1.000000

TABLE 5.12: The Resulting Pareto-Front When Using the Mutation (EBNF Operators) Operator to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.022500**	0.022500	**0.000000**	3.478261	23.000000
2	**0.017500**	0.017500	**0.000000**	15.000000	1.000000
3	**0.017500**	0.017500	**0.000000**	4.333333	6.000000
4	**0.017500**	0.017500	**0.000000**	8.500000	2.000000
5	**0.017500**	0.017500	**0.000000**	3.933333	15.000000
6	**0.017500**	0.017500	**0.000000**	3.588235	17.000000
7	**0.017500**	0.017500	**0.000000**	3.192308	26.000000
8	**0.017500**	0.017500	**0.000000**	3.454545	22.000000
9	**0.017500**	0.017500	**0.000000**	3.526316	19.000000
10	**0.017500**	0.017500	**0.000000**	3.523810	21.000000

TABLE 5.13: The Resulting Pareto-Front When Using the Mutation (Repair) Operator to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.017500**	0.017500	**0.000000**	15.000000	1.000000
2	**0.017500**	0.017500	**0.000000**	6.333333	3.000000
3	**0.017500**	0.017500	**0.000000**	8.500000	2.000000
4	**0.012500**	0.012500	**0.000000**	13.000000	1.000000
5	**0.012500**	0.015000	**0.000000**	4.142857	7.000000
6	**0.012500**	0.012500	**0.000000**	7.500000	2.000000
7	**0.012500**	0.015000	**0.000000**	4.166667	6.000000
8	**0.010000**	0.040000	**0.000000**	12.000000	1.000000
9	**0.010000**	0.015000	**0.000000**	7.500000	2.000000
10	**0.007500**	0.050000	**0.000000**	12.000000	1.000000

TABLE 5.14: The Resulting Pareto-Front When Using the Crossover Operator to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.037500**	0.037500	**0.016039**	4.000000	17.000000
2	**0.037500**	0.037500	**0.016039**	3.888889	18.000000
3	**0.037500**	0.037500	**0.018131**	3.789474	19.000000
4	**0.037500**	0.037500	**0.018828**	3.722222	18.000000
5	**0.035000**	0.035000	**0.011855**	4.125000	16.000000
6	**0.035000**	0.035000	**0.011855**	3.888889	18.000000
7	**0.035000**	0.035000	**0.013947**	3.823529	17.000000
8	**0.035000**	0.035000	**0.013947**	3.631579	19.000000
9	**0.035000**	0.035000	**0.013947**	3.722222	18.000000
10	**0.035000**	0.035000	**0.011855**	4.000000	17.000000

TABLE 5.15: The Resulting Pareto-Front When Using the Mutation (Repair) and Mutation (EBNF Operators) Operators to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.017500**	0.017500	**0.000000**	15.000000	1.000000
2	**0.017500**	0.017500	**0.000000**	2.866667	15.000000
3	**0.017500**	0.017500	**0.000000**	8.500000	2.000000
4	**0.017500**	0.017500	**0.000000**	6.333333	3.000000
5	**0.017500**	0.017500	**0.000000**	5.250000	4.000000
6	**0.017500**	0.017500	**0.000000**	4.166667	6.000000
7	**0.017500**	0.017500	**0.000000**	4.600000	5.000000
8	**0.017500**	0.017500	**0.000000**	3.857143	7.000000
9	**0.017500**	0.017500	**0.000000**	3.625000	8.000000
10	**0.017500**	0.017500	**0.000000**	3.181818	11.000000

TABLE 5.16: The Resulting Pareto-Front When Using the Crossover and Mutation (EBNF Operators) Operators to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.017500**	0.017500	**0.000000**	15.000000	1.000000
2	**0.017500**	0.017500	**0.000000**	5.750000	4.000000
3	**0.017500**	0.017500	**0.000000**	5.000000	7.000000
4	**0.012500**	0.012500	**0.000000**	13.000000	1.000000
5	**0.012500**	0.015000	**0.000000**	6.000000	3.000000
6	**0.012500**	0.012500	**0.000000**	9.500000	2.000000
7	**0.010000**	0.040000	**0.000000**	7.500000	2.000000
8	**0.010000**	0.040000	**0.000000**	12.000000	1.000000
9	**0.010000**	0.010000	**0.000000**	4.571429	7.000000
10	**0.007500**	0.037500	**0.000000**	11.000000	1.000000

TABLE 5.17: The Resulting Pareto-Front When Using the Crossover and Mutation (Repair) Operator to Infer Oberon-0 Grammars

Solution No.	**Ratio Positive Parsed**	Ratio Positive Semi-Parsed	**Ratio Negative Parsed**	Average Length of Rules	No. of Rules
1	**0.017500**	0.017500	**0.000000**	15.000000	1.000000
2	**0.017500**	0.017500	**0.000000**	2.666667	21.000000
3	**0.017500**	0.017500	**0.000000**	9.000000	2.000000
4	**0.017500**	0.017500	**0.000000**	2.916667	12.000000
5	**0.017500**	0.017500	**0.000000**	5.750000	4.000000
6	**0.017500**	0.017500	**0.000000**	4.800000	5.000000
7	**0.017500**	0.017500	**0.000000**	3.888889	9.000000
8	**0.012500**	0.012500	**0.000000**	2.615385	52.000000
9	**0.012500**	0.012500	**0.000000**	3.800000	10.000000
10	**0.012500**	0.012500	**0.000000**	13.000000	1.000000

TABLE 5.18: The Resulting Pareto-Front When Using the Crossover, Mutation (EBNF Operators) and Mutation (Repair) Operators to Infer Oberon-0 Grammars

Single Objective Approach

The following table shows the fitness of optimal solutions after ten generations using a weighted fitness function. Higher fitness values indicate better solutions, with 1 indicating a perfect solution and 0 indicating the least feasible solution. The operator combination yielding the highest fitness solution for Oberon-0 is emphasized in bold.

Operators Used	Fitness Optimal Solution
Mutation (EBNF operators)	0.410012
Mutation (Repair)	0.372743
Crossover	0.372743
Mutation (Repair) and Mutation (EBNF operators)	0.384206
Crossover and Mutation (EBNF operators)	**0.419484**
Crossover and Mutation (Repair)	0.372895
Crossover, Mutation (Repair) and Mutation (EBNF operators)	0.372743

TABLE 5.19: Fitness of Optimal Solutions After Ten Generations Using a Weighted Fitness Function to Infer an Oberon-0 Grammar

Chapter 6

Concluding Remarks

This chapter contains the concluding remarks for this thesis. First the results of the previous chapter will be discussed in section 6.1, then the research questions posed in section 1.1 in section 6.2, and finally, future work will be discussed in section 6.3 and the repositories of the software written during this project will be listed in section 6.4.

6.1 Discussion

When analysing the results from the previous chapter, a number of observations can be made:

The first observation is that in the case of DESK, where the entire dataset is used during the runs, the mutation operators perform better than crossover when considering the ratio of positive samples parsed to be the most important objective when using the multiobjective approach. This is less true in the case of Oberon-0, which could be caused by the fact that a subset of the entire dataset was used.

The use of multiple operators in the algorithm seems to have mixed results; in some cases running two or more genetic operators in succession during a generation of the algorithm produces results that are better than with a single one, in other cases, it has no effect in improving the results. With both languages and approaches, it can be observed that the fittest grammars were produced when multiple genetic operators were used in succession.

Another observation is that treating the problem as a single objective one, by encoding the objectives in a weighted fitness function, seems to produce better results than treating it as a multiobjective Pareto-optimisation problem. This is likely because, with a weighted fitness function, the objectives that are more important for the viability of a grammar can be given priority over the other objectives. This allows the algorithm to pick solutions that are performing well on objectives that have a high weight value assigned to them during the selection step, without too much concern for objectives with a lower weight.

Finally, it can be observed that the overall fitness across all outputs is less than optimal when comparing the results obtained with the prototype implementation with the algorithms discussed in chapter 3. This could likely be improved by introducing additional operators and by modifying existing operators by eliminating random elements within them and having them only promote aspects that result in a fitter grammar. These possible improvements will be further discussed in section 6.3.

6.2 Conclusion

This section aims to answer the following research questions which were posed at the beginning of this thesis:

1. ***In what way could positive samples of a language be sourced for inferring 4GLs?***
   (a) ***How reliable are public repositories for acquiring positive samples?***
   (b) ***How can positive samples of a programming language be generated?***
2. ***In what way could negative samples of a language be sourced for inferring 4GLs?***
   (a) ***How effective is using samples from a syntactically different language from the same language family?***
   (b) ***How can negative samples of a programming language be generated?***
3. ***How can an evolutionary algorithm be utilized to infer grammars of legacy languages by using positive and negative samples of these languages?***
   (a) ***What evolutionary algorithms could be used to solve the inference problem and what are their pros and cons?***
   (b) ***What objectives should the fitness function of the algorithm consider, and how should the objectives be weighed?***
   (c) ***How should the evolutionary operators be defined in the algorithm?***

6.2.1 In what way could positive samples of a language be sourced for inferring 4GLs?

In chapter 4 the viability of utilising public repositories for acquiring positive samples was examined and it was determined that using these repositories could be a valuable resource in cases where the 4GL in question is currently widely in use or has been widely in use for some time. This was shown in the case of the SQL dialect SQLite, where samples of SQL could be gathered from public repositories.

In cases of more obscure languages like BabyCobol and Oberon-0, it was determined that using public repositories was less viable. Some samples could be obtained from compiler tests from existing implementations of the languages, and from repositories that contain code written in a related language, of which most will be invalid samples due to differences in the languages.

The possibility of generating positive samples was also analysed in the same chapter. This way of obtaining positive samples would mostly be for creating datasets for developing inference methods since either a grammar or a good understanding of the language is a requirement to utilise the tools that were discussed. During this project Grammarinator [54] was utilised to generate a majority of the positive samples and SQLsmith [136] was used for SQLite in particular.

The resulting datasets that were constructed by utilising these methods can be found in the column "Unique Positive Samples" of Table 4.1 in subsection 4.3.3.

6.2.2 In what way could negative samples of a language be sourced for inferring 4GLs?

Similarly with the positive samples, chapter 4 also looks into the possibility of using public repositories for sourcing negative samples. It is discussed that some of the samples of dialects and similar languages could be passed through a recognizer, and if they fail to pass, be added to the negative

samples. For this project recognizers in the form of ANTLR parsers are used, but in a real-world application a recognizer could also take the form of an existing implementation of the language that is to be replaced.

A generation technique could also be used to pad out the negative sample set. During this project, two methods were used, both of which require an ANTLR grammar of the language. One includes the process of changing the EBNF operator in the grammars by narrowing and widening them, this process is described in section 4.2.2. A downside of this method of negative sample generation is that it could generate positive samples that would need to be filtered out.

The other generation technique is word mutation. This is a systematic way of negative sample generation inspired by the original method described by Raselimo, Taljaard and Fischer [114]. As illustrated in Figure 4.1, this implementation works with an estimate of which token-type sequences are illegal in the language and, if the input of positive samples has gaps in its coverage, might generate positive samples. Once the positive samples are supplemented with these additional samples, the algorithm will produce the corresponding negative sample set of the input.

The resulting datasets that were constructed by utilising these methods can be found in the column "Unique Negative Samples" of Table 4.1 in subsection 4.3.3.

6.2.3 How can an evolutionary algorithm be utilized to infer grammars of legacy languages by using positive and negative samples of these languages?

In chapter 5, the process of implementing a prototype inference algorithm is described. For the two variants of this prototype, two algorithms are used as a base, one is based on the multiobjective algorithm NSGA-II and the other is based on a generic genetic algorithm.

When using NSGA-II, the problem is treated as a Pareto-optimisation problem and all objectives weigh the same. When using the generic genetic algorithm as a base, the objectives can be encoded into a single value in the fitness function. This also allows for adding weights to the different objectives based on their importance.

The objectives chosen for this algorithm are the ratio of parsed positive samples, the ratio of semi-parsed positive samples, the ratio of parsed negative samples and the average number of production rules in a grammar.

When using NSGA-II as the base algorithm, all these objectives weigh the same, which is mainly meant for optimisation problems where all objectives are of equal import. In the case of grammatical inference, it can be argued that some objectives such as parsing a high number of positive samples, are more important than balancing the number of production rules.

To weigh the objectives based on their importance the following fitness function was created:

$$\begin{aligned} Fitness &= w_1 * parsed\_positive\_samples + w_2 * semi\_parsed\_positive\_samples \\ &+ w_3 * (1 - parsed\_negative\_samples) + w_4 * (1 - normalised\_avg\_length\_rules) \\ &+ w_5 * (1 - normalised\_number\_of\_rules) \end{aligned} \tag{6.1}$$

The following weights were assigned to the objectives:

Weight	Value
w_1	0.5
w_2	0.15
w_3	0.25
w_4	0.05
w_5	0.05

TABLE 6.1: The Weights for the Fitness Function

These weights are based on the importance of each objective to get a desirable result.

Both variations of the algorithm have the same operators, crossover (described in subsection 5.4.1), mutation by introducing EBNF operators (described in subsection 5.4.2) and mutation by repairing the grammars (described in subsection 5.4.3). In section 5.5 it can be seen that some operators have more success in raising the fitness of the population in comparison to others. Specifically, while there are exceptions, the way the crossover operator is currently implemented seems to have a detrimental effect on the fitness of the population whenever it is applied compared to the other operators.

Overall, this prototype shows promise but is ultimately lacking when compared to previous works that use evolutionary algorithms for grammatical inference. The prototype would need further refinements that improve the fitness of the output to make it a viable way of inferring grammars of legacy languages.

6.3 Future work

The prototype algorithm implemented in chapter 5 needs further refinements and additions before it could become a viable way of inferring grammars of legacy languages.

One way of improving the results of the algorithm could be to implement all genetic operators found in the MAGIc algorithm [57,59] discussed in subsection 3.3.1. These operators have shown to produce relatively fit grammars and they provide a baseline implementation that could be analysed on its strengths and weaknesses. The results of this analysis could be used to make informed decisions on how to further improve the algorithm.

Besides this, the existing genetic operators could be improved. One example is the crossover operator, the current implementation randomly decides which rules to cross over and where the crossover point should be. This could produce offspring that is fitter than the parents but also offspring that is less fit. An improvement for the crossover operator could be that given a parent and a positive sample that could not be entirely parsed, a second parent is selected based on its ability to parse the failing parts of the positive sample. Then, an offspring is created that includes the grammar definitions from both parents that allow it to parse the positive sample entirely. This way, the crossover operator aims to create offspring that is fitter than both parents.

From the results in section 5.5, it can be observed that mutation operators do not seem to have a beneficial effect in every circumstance. An improvement here could be to have different genetic operators for use in different circumstances. By having targeted mutation operators, a grammar could be analysed to identify its problem and a mutation operator that is known to solve similar problems could be applied. This way, the application of mutation operators that have a detrimental effect on the fitness of a grammar with a specific type of problem can be avoided.

Another interesting point to further research is how the generated datasets could be applied in other methods of grammatical inference besides evolutionary algorithms. For example, with the recent advancements in artificial neural networks [14, 163], an inference process based on this approach could be attempted with the gathered datasets. Another approach could be a human-guided process where a human oracle answers questions posed to them about the language and helps build a grammar incrementally (similar to Parsify [84] discussed in subsection 3.3.2).

6.4 Software Written During This Project

The following table lists the repositories of the software that was written during this project.

Name repository	Description	Licence
GramInfGenAlg [131]	Prototype implementation of a genetic algorithm for grammatical inference built using the MOEA Framework	GPL-2.0
OberonRecognizer [132]	Contains an ANTLR-based Oberon-0 recognizer and the tools necessary for Oberon-0 negative sample generation	GPL-2.0
BabyCobolRecognizer [130]	Contains an ANTLR-based BabyCobol recognizer and the tools necessary for BabyCobol negative sample generation	GPL-2.0
SQLiteRecognizer [133]	Contains an ANTLR-based SQLite recognizer and the tools necessary for SQLite negative sample generation	GPL-2.0

TABLE 6.2: The Repositories of the Software Written During This Project

Bibliography

[1] GitHub REST API documentation - GitHub Docs — docs.github.com. https://docs.github.com/en/rest?apiVersion=2022-11-28.

[2] Rosetta Code — rosettacode.org. https://rosettacode.org/wiki/Rosetta_Code.

[3] Hybrid Evolutionary Algorithms. 75, 2007. URL: http://link.springer.com/10.1007/978-3-540-73297-6, doi:10.1007/978-3-540-73297-6.

[4] Oludare Isaac Abiodun, Aman Jantan, Abiodun Esther Omolara, Kemi Victoria Dada, Nachaat AbdElatif Mohamed, and Humaira Arshad. State-of-the-art in artificial neural network applications: A survey. *Heliyon*, 4(11):e00938, November 2018. doi:10.1016/j.heliyon.2018.e00938.

[5] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Comput. Surv.*, 14(2):159–192, jun 1982. doi:10.1145/356876.356879.

[6] Software AG. Adabas & Natural for Application Modernization | Software AG — softwareag.com. https://www.softwareag.com/en_corporate/platform/adabas-natural.html, 2023.

[7] Dana Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980. URL: https://www.sciencedirect.com/science/article/pii/S0019995880902855, doi:10.1016/S0019-9958(80/90285-5.

[8] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526, doi:10.1016/0890-5401(87/90052-6.

[9] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, Apr 1988. doi:10.1023/A:1022821128753.

[10] ANTLR Project. antlr/grammars-v4, 2024. URL: https://github.com/antlr/grammars-v4.

[11] Hiroki Arimura, Takeshi Shinohara, and Setsuko Otsuki. Finding minimal generalizations for unions of pattern languages and its application to inductive inference from positive data. volume 775, 05 1997. doi:10.1007/3-540-57785-8_178.

[12] Elizabeth A. Babb. A Look at the IBM Application Development Facility (ADF-II). *Journal of Information Systems Management*, 1(3):39–49, 1984. arXiv:https://doi.org/10.1080/07399019408963044, doi:10.1080/07399019408963044.

[13] John Backus. *The History of Fortran I, II, and III*, page 25–74. Association for Computing Machinery, New York, NY, USA, 1978. URL: https://doi.org/10.1145/800025.1198345.

[14] Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Igor Khmelnitsky, Martin Leucker, Daniel Neider, Rajarshi Roy, and Lina Ye. Extracting Context-Free Grammars from Recurrent Neural Networks using Tree-Automata Learning and A* Search. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 113–129. PMLR, 23–27 Aug 2021. URL: https://proceedings.mlr.press/v153/barbot21a.html.

[15] BluePhoenix Solutions. BLUEPHOENIX SOLUTIONS, 2002. URL: http://web.archive.org/web/20020121233342/http:/www.bluephoenixsolutions.com:80/.

[16] Luca Boasso. GitHub - lboasso/oberon0: Adapted source code of Niklaus Wirth's "Compiler Construction" book, 2023. URL: https://github.com/lboasso/oberon0.

[17] Lorenzo Brigato and Luca Iocchi. A close look at deep learning with small data. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 2490–2497, 2021. doi:10.1109/ICPR48806.2021.9412492.

[18] Broadcom. IDMS™ Reference 19.0. https://techdocs.broadcom.com/us/en/ca-mainframe-software/database-management/ca-idms-reference/19-0.html, 2005.

[19] Broadcom. CA Ideal. https://techdocs.broadcom.com/us/en/ca-miscellaneous/legacy_bookshelves_and_pdfs/bookshelves_and_pdfs/bookshelves/ca-ideal.html, 2015.

[20] Broadcom. CA Ramis. https://techdocs.broadcom.com/us/en/ca-miscellaneous/legacy_bookshelves_and_pdfs/bookshelves_and_pdfs/bookshelves/ca-ramis.html, 2023.

[21] J. H. Bryant and Parlan Semple. Gis and file management. In *Proceedings of the 1966 21st National Conference*, ACM '66, page 97–107, New York, NY, USA, 1966. Association for Computing Machinery. doi:10.1145/800256.810686.

[22] Christoff Bürger. Reference attribute grammar controlled graph rewriting: motivation and overview. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, page 89–100, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2814251.2814257.

[23] Business Wire. BluePhoenix to Sell AppBuilder Business to Magic Software, 2011. URL: https://www.businesswire.com/news/home/20111031005584/en/BluePhoenix-to-Sell-AppBuilder-Business-to-Magic-Software.

[24] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, aug 1992. doi:10.1145/142137.142141.

[25] Cincom. Upgrade to the latest release of mantis for mobile applications. https://mantis.cincom.com, 2023.

[26] Alexander Clark, Rémi Eyraud, and Amaury Habrard. A Polynomial Algorithm for the Inference of Context Free Languages. In Alexander Clark, François Coste, and Laurent Miclet, editors, *Grammatical Inference: Algorithms and Applications*, pages 29–42, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[27] Axel Cleeremans, David Servan-Schreiber, and James L. McClelland. Finite State Automata and Simple Recurrent Networks. *Neural Computation*, 1(3):372–381, 09 1989. arXiv:https://direct.mit.edu/neco/article-pdf/1/3/372/811867/neco.1989.1.3.372.pdf, doi:10.1162/neco.1989.1.3.372.

[28] David Coppit and Jiexin Lian. yagg: an easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 356–359, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1101908.1101969.

[29] David Coppit and Jiexin Lian. coppit/yagg, 2018. URL: https://github.com/coppit/yagg.

[30] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, mar 1964. doi:10.1145/363958.363994.

[31] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. doi:10.1109/4235.996017.

[32] Céline Deknop, Johan Fabry, Kim Mens, and Vadim Zaytsev. Improving a Software Modernisation Process by Differencing Migration Logs. In Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka, editors, *Proceedings of the 21st International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 270–286. Springer, 2020. doi:10.1007/978-3-030-64148-1_17.

[33] M. Di Penta and Kunal Taneja. Towards the automatic evolution of reengineering tools. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 241–244, 2005. doi:10.1109/CSMR.2005.52.

[34] Massimiliano Di Penta, Pierpaolo Lombardi, Kunal Taneja, and Luigi Troiano. Search-based inference of dialect grammars. *Soft Computing*, 12(1):51–66, Jan 2008. doi:10.1007/s00500-007-0216-5.

[35] Dyalog Ltd. Dyalog - Dyalog Version 18.2. https://www.dyalog.com/dyalog/dyalog-versions/182.htm, March 2022.

[36] Esri. ArcGIS Web AppBuilder. https://developers.arcgis.com/web-appbuilder/, 2015.

[37] Gary E. Fisher. A functional model for fourth generation languages. Technical report, 1986. doi:10.6028/nbs.sp.500-138.

[38] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/964001.964011.

[39] Daniel Vaz Gaspar et al. Flask App Builder. http://flaskappbuilder.pythonanywhere.com, 2014.

[40] GitHub. GitHub Copilot — Your AI pair programmer, 2021. URL: https://github.com/features/copilot.

[41] E Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967. URL: https://www.sciencedirect.com/science/article/pii/S0019995867911655, doi:10.1016/S0019-9958(67/91165-5.

[42] Bradley Grant. bradleygrant/sakila-sqlite3, 2024. URL: https://github.com/bradleygrant/sakila-sqlite3.

[43] Marco Grochowski, Marcus Völker, and Stefan Kowalewski. Test Suite Augmentation for Reconfigurable PLC Software in the Internet of Production. In *Formal Methods for Industrial Critical Systems: 27th International Conference, FMICS 2022, Warsaw, Poland, September 14–15, 2022, Proceedings*, page 137–154, Berlin, Heidelberg, 2022. Springer-Verlag. doi:10.1007/978-3-031-15008-1_10.

[44] Dean Guida et al. App Builder — Software to Build complete Enterprise Apps Faster. https://www.infragistics.com/products/appbuilder, 2021.

[45] David Hadka. MOEA Framework, 2023. URL: http://moeaframework.org/index.html.

[46] Margaret H. Hamilton and William R. Hackler. A Formal Universal Systems Semantics for SysML. *INCOSE International Symposium*, 17(1):1333–1357, 2007. doi:10.1002/j.2334-5837.2007.tb02952.x.

[47] Margaret H. Hamilton and Saydean Zeldin. Higher Order Software — A Methodology for Defining Software. *IEEE Transactions on Software Engineering*, 2(1):9–32, 1976. doi:10.1109/TSE.1976.233798.

[48] Margaret H. Hamilton and Saydean Zeldin. The relationship between design and verification. *Journal of Systems and Software*, 1:29–56, 1979. doi:10.1016/0164-1212(79/90004-9.

[49] David Harel. Statecharts in the making: a personal account. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 5–1–5–43, New York, NY, USA, 2007. ACM. doi:10.1145/1238844.1238849.

[50] L. R. Harris. User oriented data base query with the ROBOT natural language query system. *International Journal of Man-Machine Studies*, 9(6):697–713, 1977. doi:10.1016/S0020-7373(77/80037-0.

[51] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. Generating Conforming Programs with Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2023, page 86–99, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3624007.3624056.

[52] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. xsmith/xsmith, 2023. URL: https://gitlab.flux.utah.edu/xsmith/xsmith/.

[53] Yann Herklotz and John Wickerson. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 277–287, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373087.3375310.

[54] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 45–48, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3278186.3278193.

[55] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer, 2023. URL: https://github.com/renatahodovan/grammarinator.

[56] HOPL. Gis (id:2733/gis001): General information store. https://hopl.info/showlanguage2.prx?exp=2733, 2023.

[57] Dejan Hrncic, Marjan Mernik, and Barrett R. Bryant. Improving Grammar Inference by a Memetic Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(5):692–703, 2012. doi:10.1109/TSMCC.2012.2186802.

[58] Dejan Hrnčič, Marjan Mernik, and Barrett R. Bryant. Embedding DSLs into GPLs: A Grammatical Inference Approach, Dec 2011. doi:10.5755/j01.itc.40.4.980.

[59] Dejan Hrnčič, Marjan Mernik, Barrett R. Bryant, and Faizan Javed. A memetic grammar inference algorithm for language learning. *Applied Soft Computing*, 12(3):1006–1020, 2012. URL: https://www.sciencedirect.com/science/article/pii/S1568494611004662, doi:10.1016/j.asoc.2011.11.024.

[60] IBM. Assembler language coding conventions - IBM Documentation. URL: https://www.ibm.com/docs/en/hla-and-tf/1.6?topic=structure-assembler-language-coding-conventions.

[61] IBM. Sample program - IBM Documentation. URL: https://www.ibm.com/docs/en/zos/2.1.0?topic=guide-sample-program.

[62] IBM. VisualAge Pacbase: Structured Code. http://www.ibm.com/support/docview.wss?rs=37Ź context=SSEP67Ź uid=swg27005477, 1983.

[63] IBM. Introduction to CICS — IBM Documentation. https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-introduction-cics, 1990.

[64] IBM. z/OS Language Environment Concepts Guide — Sample COBOL program, 1990. URL: https://www.ibm.com/docs/en/zos/2.1.0?topic=routines-sample-cobol-program.

[65] IBM. CICS API command format - IBM Documentation. https://www.ibm.com/docs/en/cics-ts/5.4?topic=development-cics-api-command-format, 2017.

[66] IBM. IBM z/OS MVS JCL Reference (SA23-1385-30), 2019. URL: https://www-40.ibm.com/servers/resourcelink/svc00100.nsf/pages/zOSV2R3sa231385?OpenDocument.

[67] IBM. Reference format, 2019. URL: https://www.ibm.com/docs/en/developer-for-zos/14.1?topic=structure-reference-format.

[68] IBM. CICS Transaction Server for z/OS 6.1 - IBM Documentation. https://www.ibm.com/docs/en/cics-ts/6.1, 2021.

[69] IBM. Coding COBOL programs to run under CICS - IBM Documentation. https://www.ibm.com/docs/en/cobol-zos/6.3?topic=cics-coding-cobol-programs-run-under, 2021.

[70] IBM. Writing CICS transactions in PL/I — IBM Documentation. https://www.ibm.com/docs/en/epfz/5.3?topic=preprocessor-writing-cics-transactions-in-pli, 2021.

[71] IBM. IBM Enterprise PL/I for z/OS 6.1, 2022. URL: https://www.ibm.com/docs/en/epfz/6.1.

[72] Feargus Illingworth et al. 2022 Mainframe Modernization Business Barometer Report. Technical report, Advanced, 2022. https://modernsystems.oneadvanced.com/en/reports/modernisation2022/.

[73] Intel Corporation. Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer Guide and Reference. URL: https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-0/overview.html.

[74] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[75] Rochus Keller. rochus-keller/Oberon, 2023. URL: https://github.com/rochus-keller/Oberon.

[76] Željko Kovačevic, Marjan Mernik, Miha Ravber, and Matej Črepinšek. From Grammar Inference to Semantic Inference—An Evolutionary Approach. *Mathematics*, 8(5), 2020. URL: https://www.mdpi.com/2227-7390/8/5/816, doi:10.3390/math8050816.

[77] Patrick Kreutzer. FAU-Inf2/FuzzPEG, 2021. URL: https://github.com/anse1/sqlsmith.

[78] Patrick Kreutzer. FAU-Inf2/j-PEG, 2021. URL: https://github.com/FAU-Inf2/j-PEG.

[79] S.-Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964. URL: https://www.sciencedirect.com/science/article/pii/S0019995864901202, doi:10.1016/S0019-9958(64/90120-2.

[80] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics. URL: https://aclanthology.org/D11-1140.

[81] R. Lammel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, 2001. doi:10.1109/52.965809.

[82] Marc Lankhorst. Iterated function systems optimization with genetic algorithms. 05 1996.

[83] Michael E Lesk and Eric Schmidt. *Lex: A lexical analyzer generator*, volume 39. Bell Laboratories Murray Hill, NJ, 1975.

[84] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 565–574, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2737924.2738002.

[85] Level 8 Systems. Level 8: Products: Geneva AppBuilder. http://web.archive.org/web/20010604014633/http:/www.level8.com:80/appbuilder/default.asp, 2001.

[86] Level 8 Systems. Level 8 Systems Subsidiary Completes $20 Million Sale Of Geneva Appbuilder Product To Liraz Systems, 2001. URL: http://web.archive.org/web/20011112063826/http:/www.level8.com/news/news_view.asp?newsId=77.

[87] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet physics. Doklady*, 10:707–710, 1965. URL: https://api.semanticscholar.org/CorpusID:60827152.

[88] M. Li and P.M.B. Vitanyi. A theory of learning simple concepts under simple distributions and average case complexity for the universal distribution. In *30th Annual Symposium on Foundations of Computer Science*, pages 34–39, 1989. doi:10.1109/SFCS.1989.63452.

[89] Zhenhui Li, Huaxiu Yao, and Fenglong Ma. Learning with Small Data. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, WSDM '20, page 884–887, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3336191.3371874.

[90] Barry Lichtenstein and Sharuff Morsa. With a Little Help from My Friends: How HLASM Interacts with LE, Binder, C, CICS and MQ. Technical report, IBM, 2013.

[91] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi:10.1145/3428264.

[92] Jochen Ludewig. Practical methods and tools for specification. In A. Kündig, R. E. Bührer, and J. Dähler, editors, *Embedded Systems: New Approaches to Their Formal Description and Design. An Advanced Course*, pages 174–207, Berlin, Heidelberg, March 1986. Springer. doi:10.1007/BFb0016352.

[93] Magic Software Enterprises. IDE, 2013. URL: http://www.appbuilder.com/products/ide.html.

[94] Magic Software Enterprises. About Us. http://www.appbuilder.com/about/about-us.html, 2023.

[95] MariaDB. SQL Statements - MariaDB Knowledge Base, 2023. URL: https://mariadb.com/kb/en/sql-statements/.

[96] James Martin and Joe Leben. *Fourth-Generation Languages*, volume II, Representative 4GLs. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[97] James Martin and Joe Leben. *Fourth-Generation Languages*, volume III, 4GLs from IBM. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[98] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10:100–107, 1998. URL: https://api.semanticscholar.org/CorpusID:14018070.

[99] Zbigniew Michalewicz and Marc Schoenauer. Evolutionary Algorithms. In Hossein Bidgoli, editor, *Encyclopedia of Information Systems*, pages 259–267. Elsevier, New York, 2003. URL: https://www.sciencedirect.com/science/article/pii/B0122272404000654, doi:10.1016/B0-12-227240-4/00065-4.

[100] Microsoft. Transact-SQL reference (Database Engine) - SQL Server | Microsoft Learn, February 2023. URL: https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16.

[101] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, dec 1990. doi:10.1145/96267.96279.

[102] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, September 1997. doi:10.1613/jair.374.

[103] Oracle. Oracle Database SQL Language Reference, 19c. https://docs.oracle.com/en/database/oracle/oracle-database/19/sqlrf/, June 2023.

[104] Oracle Corporation. Fortran Programming Guide. URL: https://docs.oracle.com/cd/E19957-01/805-4940/index.html.

[105] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, jun 1995. doi:10.1145/210376.197409.

[106] Panos Panay. Bringing the power of AI to Windows 11 - unlocking a new era of productivity for customers and developers with Windows Copilot and Dev Home - Windows Developer Blog, May 2023. URL: https://blogs.windows.com/windowsdeveloper/2023/05/23/bringing-the-power-of-ai-to-windows-11-unlocking-a-new-era-of-productivity-for-customers-and-developers-with-windows-copilot-and-dev-home/.

[107] Terence Parr. ANTLR—ANother Tool for Language Recognition, release 4.12.0, 2023. http://antlr.org.

[108] Antonello Pasini. Artificial neural networks for small dataset analysis. *Journal of Thoracic Disease*, 7(5), 2015. URL: https://jtd.amegroups.org/article/view/4418.

[109] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, Sep 1972. doi:10.1007/BF01932308.

[110] George Radin. *The Early History and Characteristics of PL/I*, page 551–575. Association for Computing Machinery, New York, NY, USA, 1978. URL: https://doi.org/10.1145/800025.1198410.

[111] Mamillapally Raghavender Sharma. A Short Communication on Computer Programming Languages in Modern Era. *International Journal of Computer Science and Mobile Computing*, 9:50–60, 09 2020. doi:10.47760/IJCSMC.2020.v09i09.006.

[112] Raincode. TIALAA, 2021. URL: http://web.archive.org/web/20210308200334/https://www.raincode.com/technical-landscape/tialaa/.

[113] Moeketsi Raselimo and Bernd Fischer. Automatic grammar repair. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2021, page 126–142, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486608.3486910.

[114] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In *Proceedings of the 12th ACM*

SIGPLAN International Conference on Software Language Engineering, SLE 2019, page 83–87, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3357766.3359542.

[115] Stephen Reindl. steven-r/Oberon0Compiler, 2023. URL: https://github.com/steven-r/Oberon0Compiler.

[116] Peter Reutemann. fracpete/employees-db-sqlite, 2024. URL: https://github.com/fracpete/employees-db-sqlite.

[117] Rodrigo Rocha. GitHub - rcorcs/OberonC: Oberon-0 Compiler: An educational compiler for a subset of the Oberon programming language, 2014. URL: https://github.com/rcorcs/OberonC.

[118] Ovidiu Roşu. Grammatical Inference from Source Code Written in an Unknown Programming Language. 2014. URL: https://grammarware.net/text/2014/rosu.pdf.

[119] Diptikalyan Saha and Vishal Narula. Gramin: A system for incremental learning of programming language grammars. In *Proceedings of the 4th India Software Engineering Conference*, ISEC '11, page 185–194, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1953355.1953380.

[120] Yasubumi Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science*, 76(2):223–242, 1990. URL: https://www.sciencedirect.com/science/article/pii/030439759090017C, doi:10.1016/0304-3975(90/90017-C.

[121] Yasubumi Sakakibara. Recent advances of grammatical inference. *Theoretical Computer Science*, 185(1):15–45, 1997. URL: https://www.sciencedirect.com/science/article/pii/S0304397597000145, doi:10.1016/S0304-3975(97/00014-5.

[122] J.E. Sammet. The real creators of Cobol. *IEEE Software*, 17(2):30–32, 2000. doi:10.1109/52.841602.

[123] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-Suite Augmentation for Evolving Software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227, 2008. doi:10.1109/ASE.2008.32.

[124] SAP SE. ABAP — Keyword Documentation. https://help.sap.com/doc/abapdocu_751_index_htm/7.51/en-US/index.htm, 2017.

[125] SAP SE. ADT Class Execution — SAP Help Portal. https://help.sap.com/docs/btp/sap-business-technology-platform/adt-class-execution, 2017.

[126] SAP SE. Arithmetic Calculations — ABAP Keyword Documentation. https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abendivisions_abexa.htm, 2017.

[127] SAP SE. Method Calls — ABAP Keyword Documentation. https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abenmethod_call_guidl.htm, 2017.

[128] SAP SE. Substrings — ABAP Keyword Documentation. https://help.sap.com/doc/abapdocu_752_index_htm/7.52/en-us/abendata_process_fields_abexa.htm, 2017.

[129] SAP SE. Do you know ABAP as a Programing Language? | SAP Blogs, 2020. URL: https://blogs.sap.com/2020/02/26/do-you-know-abap-as-a-programing-language/.

[130] Ömer Sayilir. omersayilir75/babycobolrecognizer, 2024. URL: https://github.com/omersayilir75/BabyCobolRecognizer.

[131] Ömer Sayilir. omersayilir75/GramInfGenAlg, 2024. URL: https://github.com/omersayilir75/GramInfGenAlg.

[132] Ömer Sayilir. omersayilir75/oberonrecognizer, 2024. URL: https://github.com/omersayilir75/OberonRecognizer.

[133] Ömer Sayilir. omersayilir75/sqliterecognizer, 2024. URL: https://github.com/omersayilir75/SQLiteRecognizer.

[134] Tobias Schaer. GitHub - tschaer/Oberon-0-go: Port of Niklaus Wirth's Oberon-0 compiler to Go, 2017. URL: https://github.com/tschaer/Oberon-0-go.

[135] Select Business Solutions. NOMAD, 2023. URL: https://selectbs.com/products/nomad/.

[136] Andreas Seltenreich. Bug Squashing with SQLsmith, 2018. URL: https://www.postgresql.eu/events/pgconfeu2018/sessions/session/2221/slides/145/sqlsmith-talk.pdf.

[137] Andreas Seltenreich. anse1/sqlsmith, 2023. URL: https://github.com/anse1/sqlsmith.

[138] Pavel Senin, Jessica Lin, Xing Wang, Tim Oates, Sunil Gandhi, Arnold P. Boedihardjo, Crystal Chen, Susan Frankenstein, and Manfred Lerner. GrammarViz 2.0: A Tool for Grammar-Based Pattern Discovery in Time Series. In Toon Calders, Floriana Esposito, Eyke Hüllermeier, and Rosa Meo, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 468–472, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[139] S. Skolarikis et al. Comidor Low-Code App Builder Environment. https://www.comidor.com/low-code-platform/, 2022.

[140] solid IT gmbh. SQLite System Properties, 2023. URL: https://db-engines.com/en/system/SQLite.

[141] Jared Spataro. Introducing Microsoft 365 Copilot – your copilot for work - The Official Microsoft Blog, March 2023. URL: https://blogs.microsoft.com/blog/2023/03/16/introducing-microsoft-365-copilot-your-copilot-for-work/.

[142] SQLite Consortium. SQLite Home Page, 2023. URL: https://sqlite.org/index.html.

[143] Srinimf. How To Master PACBASE For Mainframe In Only Seven Days, December 2015. URL: https://www.slideshare.net/srinipdn/pacbase-fundamentals.

[144] Andrew Stevenson and James R. Cordy. A survey of grammatical inference in software engineering. *Science of Computer Programming*, 96:444–459, 2014. Selected Papers from the Fifth International Conference on Software Language Engineering (SLE 2012). URL: https://www.sciencedirect.com/science/article/pii/S0167642314002469, doi:10.1016/j.scico.2014.05.008.

[145] Order Processing Technologies. About Us: CorVision History. http://www.cv2vb.com/CorVisionHistory.htm, 2015.

[146] Order Processing Technologies. CV2VB — cv2vb.com. http://www.cv2vb.com/, 2015.

[147] TIBCO. TIBCO FOCUS Release Notes. Release 8207.27.0. DN1001076.0721. https://ecl.informationbuilders.com/focus/index.jsp?topic=/shell_8207/foc8207relnotes.pdf, 2021.

[148] TIOBE. TIOBE Index for June 2023. https://www.tiobe.com/tiobe-index/, 2023.

[149] David Tippett. dtaivpp/car_company_database, 2024. URL: https://github.com/dtaivpp/car_company_database.

[150] UnBCIC-TP2. GitHub - UnBCIC-TP2/Oberon-Scala: An implementation of the Oberon language using Scala, 2023. URL: https://github.com/UnBCIC-TP2/Oberon-Scala.

[151] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, nov 1984. doi:10.1145/1968.1972.

[152] Peter Van Roy. Programming Paradigms for Dummies: What Every Programmer Should Know. 04 2012.

[153] VeriTreff GmbH. Die Historie der SAP-Programmiersprache ABAP, 2001. URL: http://www.4ap.de/abap/historie.

[154] Wojciech Wieczorek, Łukasz Strąk, Arkadiusz Nowakowski, and Olgierd Unold. A Toolbox for Context-Sensitive Grammar Induction by Genetic Search. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno van Zaanen, editors, *Proceedings of the Fifteenth International Conference on Grammatical Inference*, volume 153 of *Proceedings of Machine Learning Research*, pages 191–201. PMLR, 23–27 Aug 2021. URL: https://proceedings.mlr.press/v153/wieczorek21a.html.

[155] Wikipedia contributors. ABAP — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ABAP&oldid=1141146839, 2023.

[156] N. Wirth. The programming language pascal. *Acta Informatica*, 1(1):35–63, 3 1971. URL: https://link.springer.com/article/10.1007/BF00264291, doi:10.1007/BF00264291.

[157] Niklaus Wirth. MODULA-2. *ETH, Eidgenössische Technische Hochschule Zürich, Institut für Informatik*, 36, 1980. URL: https://doi.org/10.3929/ethz-a-000189918, doi:10.3929/ETHZ-A-000189918.

[158] Niklaus Wirth. From Modula to Oberon and the programming language Oberon. *ETH, Eidgenössische Technische Hochschule Zürich, Institut für Informatik, Fachgruppe Computer-Systeme*, 82, 1987. URL: https://doi.org/10.3929/ethz-a-005363226, doi:10.3929/ETHZ-A-005363226.

[159] Niklaus Wirth. *Compiler Construction*. Addison Wesley Publishing Company, 1996.

[160] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra Cohen. Directed test suite augmentation: techniques and tradeoffs. pages 257–266, 11 2010. doi:10.1145/1882291.1882330.

[161] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1993498.1993532.

[162] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. GitHub - csmith-project/csmith: Csmith, a random generator of C programs, 2011. URL: https://github.com/csmith-project/csmith.

[163] Daniel M. Yellin and Gail Weiss. Synthesizing Context-free Grammars from Recurrent Neural Networks (Extended Version), 2021. arXiv:2101.08200.

[164] Ryo Yoshinaka and Alexander Clark. Polynomial Time Learning of Some Multiple Context-Free Languages with a Minimally Adequate Teacher. In Philippe de Groote and Mark-Jan Nederhof, editors, *Formal Grammar*, pages 192–207, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[165] Vadim Zaytsev. Recovery, Convergence and Documentation of Languages, October 2010.

[166] Vadim Zaytsev. Open Challenges in Incremental Coverage of Legacy Software Languages. In Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara, editors, *Post-proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2)*, pages 1–6, 2017. doi:10.1145/3167105.

[167] Vadim Zaytsev. Parser Generation by Example for Legacy Pattern Languages. In Matthew Flatt and Sebastian Erdweg, editors, *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, pages 212–218. ACM, 2017. doi:10.1145/3136040.3136058.

[168] Vadim Zaytsev. Ecosystem Health as a Reason for Migration: The Mainframe Case. Second International Workshop on Software Health, Industrial Track (SoHeal), 2019.

[169] Vadim Zaytsev. Software Language Engineers' Worst Nightmare. In Ralf Lämmel, Laurence Tratt, and Juan De Lara, editors, *Proceedings of the 13th International Conference on Software Language Engineering (SLE)*, pages 72–85. ACM, 2020. doi:10.1145/3426425.3426933.

[170] Vadim Zaytsev. OSIRIS - Course offerings 201400225 2022 - Software Evolution, 2022. URL: https://osiris.utwente.nl/student/OnderwijsCatalogusSelect.do?selectie=cursusŹcursus=201400225Źcollegejaar=2022Źtaal=en.

[171] Vadim Zaytsev and Johan Fabry. Fourth Generation Languages are Technical Debt. International Conference on Technical Debt, Tools Track (TD-TD), 2019. Extended Abstract.

[172] Tony Zhao et al. Agora App Builder. https://appbuilder.agora.io, 2020.