CODE GENERATION: AN INTRODUCTION TO TYPED EBNF

by

Jason G. Young

A report submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

_____   _____
Dr. Vicki H. Allan   Dr. Kenneth Sundberg
Major Professor   Committee Member

_____
Dr. Curtis Dyreson
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2015

ABSTRACT

Code Generation: An Introduction to Typed EBNF

by

Jason G. Young, Master of Science

Utah State University, 2015

Major Professor: Dr. Vicki H. Allan
Department: Computer Science

Errors and inconsistencies between code components can be very costly in a software project. Efforts to reduce these costs can include the use of tools that limit human interaction with code by generating it from a description. Most of these tools only generate one of the many components found in a typical application, reducing human-introduced errors within code. This paper introduces two new works: (1) an input specification called Typed EBNF (TEBNF), and (2) a prototype tool that demonstrates how TEBNF can be used to generate code. The tool generates code for a console application as described by a TEBNF grammar. An application built from the generated code will be able to receive input data, parse it, and process it.

(60 pages)

CONTENTS

LIST OF TABLES

LIST OF LISTINGS

LIST OF FIGURES

CHAPTER 1

Introduction

Various defects in software can result from multiple developers working on the same code within a project [1]. Some errors introduced by the coding activities of developers may consist of behavioral differences within and between components and their interfaces [2–4]. Other errors stem from inadequate comprehension of project code, supported by the fact that developers spend 50% to 80% of their time understanding it [5]. Furthermore, these kinds of misunderstandings can lead to bug fixes that inject more faults into code [3].

Comprehensive requirements analysis, design, and other good software development practices can prevent many post-delivery software problems [6], but cannot completely eliminate the possibility of human error. Even well designed projects can get mired in code implementation details that hinder or prevent developers from delivering on time and within budget.

Code generation tools can improve the software development process by generating well-organized code reducing implementation errors and inconsistencies [7]. Code generation has been shown to improve the productivity of developers, guarantee correctness of syntax, and decrease errors [8]. A compiler is a type of code generation tool that generates binary executables from code written in higher-level programming languages like C and C++. Some tools generate higher-level code instead of binary given a set of rules known as a grammar. This high-level generated code is then fed to a compiler to create a binary executable.

Most high-level code generation tools are specialized and only generate one part of an applications overall implementation. Popular targets for specialized code generation are lexical analyzers and parsers. One of the earliest code generation tools is the compiler-compiler (parser generator), a term first presented in the early 1960s [9].

The Yacc (Yet Another Compiler Compiler) [10] compiler-compiler is a popular tool that generates parsers given (1) a context-free grammar to describe input, (2) an action (code snippets) to run for each token grouping that matches a grammar rule, and (3) code to provide input tokens to the parser. This input specification is a hybrid between a declarative domain-specific language (i.e. a grammar) and an imperative programming language like C [11–13]. Nevertheless, Yacc lacks the ability to read an input stream and convert it into tokens for parsing.

Lexical analyzer generators such as Lex and Flex must be used in conjunction with Yacc to accomplish this separate task of lexical analyzer generation [10, 14]. The practice of using separate tools to generate a lexical analyzer and parser mean that input formats for each tool must be learned and used correctly to achieve the desired result.
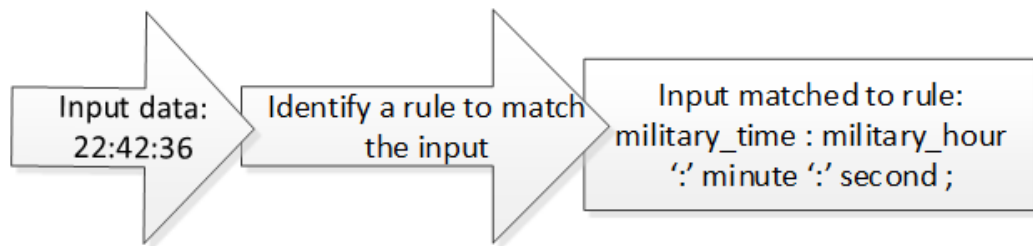
Figure 1.1: Example Yacc grammar rule matched to input.

Figure 1.1 shows an example Yacc grammar rule matched to input formatted as military time. In this example, military_hour, minute, and second (all defined somewhere else in the grammar) describe specific parts of input data as military_time. When tokens match this rule, a code action is executed [10, 15].

This report will present a new input specification based on Extended Backus-Naur Form (EBNF) called Typed-EBNF (TEBNF) (see appendix A), and a new prototype code generation tool that uses TEBNF as its input specification. The tool will validate key features of TEBNF:

1. TEBNF grammars can describe input patterns that include a mixture of strings, numbers, and/or raw groupings of bytes.

2. TEBNF integrates grammar rules with states and actions.

3. TEBNF can specify different methods of receiving input and sending output.

4. TEBNF can declare how raw input data should be unmarshaled into well-known types of specific sizes (in bytes).

5. TEBNF can declare how well-known types should be marshaled back into their original format.

6. TEBNF provides typed and non-typed EBNF terminals.

7. TEBNF supports arithmetic and non-arithmetic operations inside grammar rules.

8. TEBNF grammars match specific pieces of a given input to well-known types of varied sizes (in bits or bytes).

9. TEBNF supports the use of variables.

10. TEBNF is Turing complete (see appendix B).

The prototype code generation tool will demonstrate that it can generate a console application that can:

1. Parse data from different kinds of inputs.

2. Process data to produce specific output(s).

3. Direct output(s) to a network destination (UDP/IP), a file, or a console-based user-interface.

4. Support custom network protocol handling and interaction (UDP/IP).

5. Provide a console-based user interface for the application.

CHAPTER 2

Background and Related Works

The purpose of code generation tools is to help developers improve their productivity, ensure correctness of code syntax, and lessen the number of errors in software [8]. The input specifications of these tools introduce additional levels of indirection to solve these problems [16]. A range of tools have been created to generate code that can be complex and tedious to write by hand:

- Scanners

- Parsers

- Interpreters

- Compilers

- Graphical user interfaces

2.1 Traditional Scanner and Parser Code Generation

The Yacc compiler-compiler [10] first introduced in 1975 generates LALR parsers that must be run with a lexical analyzer generator such as Lex. Similar to Yacc is Bison [17,18]a Yacc-compatible parser generator that accepts any properly written Yacc grammar. Like Yacc, Bison-generated parsers read a sequence of tokens from a scanner generated by a lexical analyzer generator like Lex or flex.

To illustrate the steps of traditional parser generation using Lex and Yacc (figure 1.1) [10, 14, 15], a file is provided by the developer containing a set of patterns that define how to separate strings found in source data. This file is read by Lex, which uses these patterns to generate the C source code of a lexical analyzer. This newly generated lexical

analyzer uses the patterns to identify specific strings in the input and split them into tokens to simplify processing.
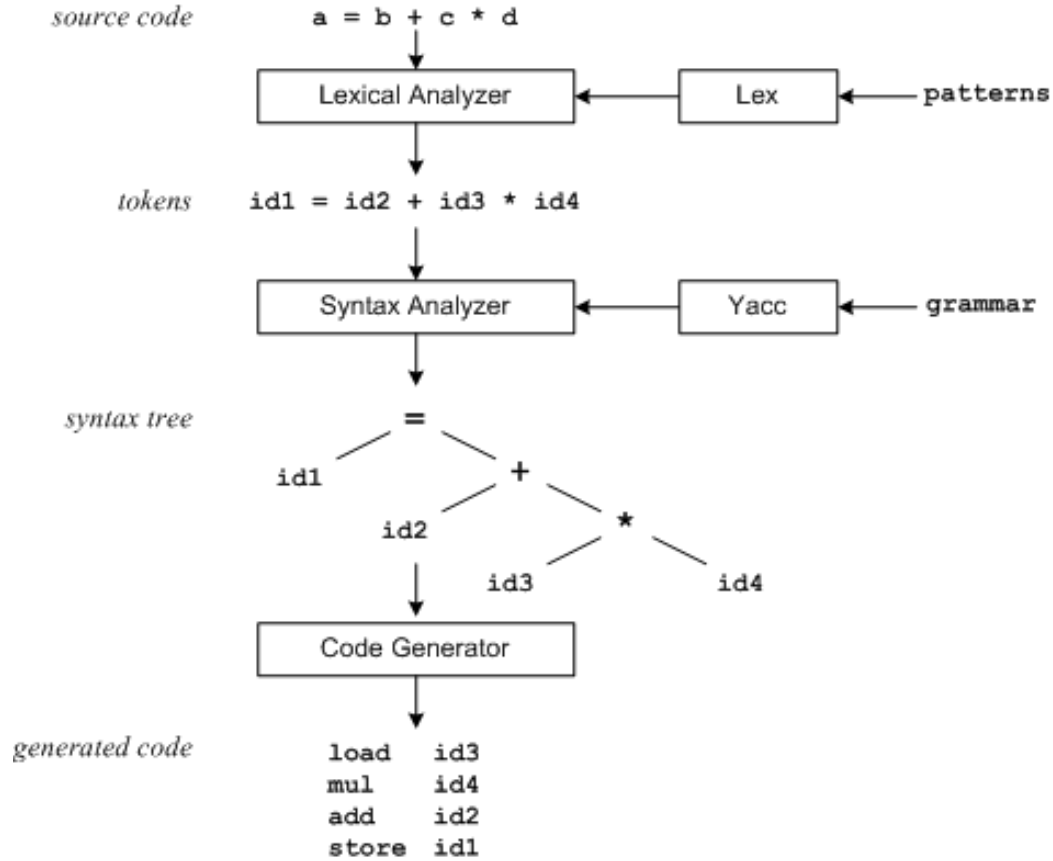
Figure 2.1: Lex and Yacc Compilation Sequence [15]

A file containing grammar rules is provided by the developer to Yacc, which uses those rules to generate C source code for a syntax analyzer (i.e. parser). The syntax analyzer uses this grammar to transform the tokens output by the lexical analyzer into a syntax tree. The structure of this syntax tree implies the precedence and associativity of operators found within the tokens. The syntax tree is then traversed in depth-first order to generate the desired source code (figure 2.1) [15].

A predicated-LL(k) parser called ANTLR [19] was introduced by Parr and Quong in 1995. The ANTLR parser generator was advertised to be easier to use than generators like YACC or BISON. An LL(k) parser is a top-down parser that parses from left to right,

6

utilizing a look-ahead of k tokens. All parsing decisions are made solely from the next k tokens, which means that it does no backtracking.

The HYACC (Hawaii Yacc) parser generator first released in 2008 supports complete LR(0), LALR(1), LR(1), and partial LR(k) [20, 21]. HYACC is compatible with Yacc and Bison input grammars and works with Lex. The HYACC parser generator is notable because it can resolve reduce/reduce conflicts through its implementation of the LR(1) parser generation algorithm [20]. Reduce/reduce conflicts occur when two or more rules in an input grammar apply to the same input sequence [22]. These conflicts are typically the result of a serious problem with an input grammar [22].

Figure 2.2: Traditional code generation process using a lexical analyzer generator in conjunction with a parser generator and actions code.

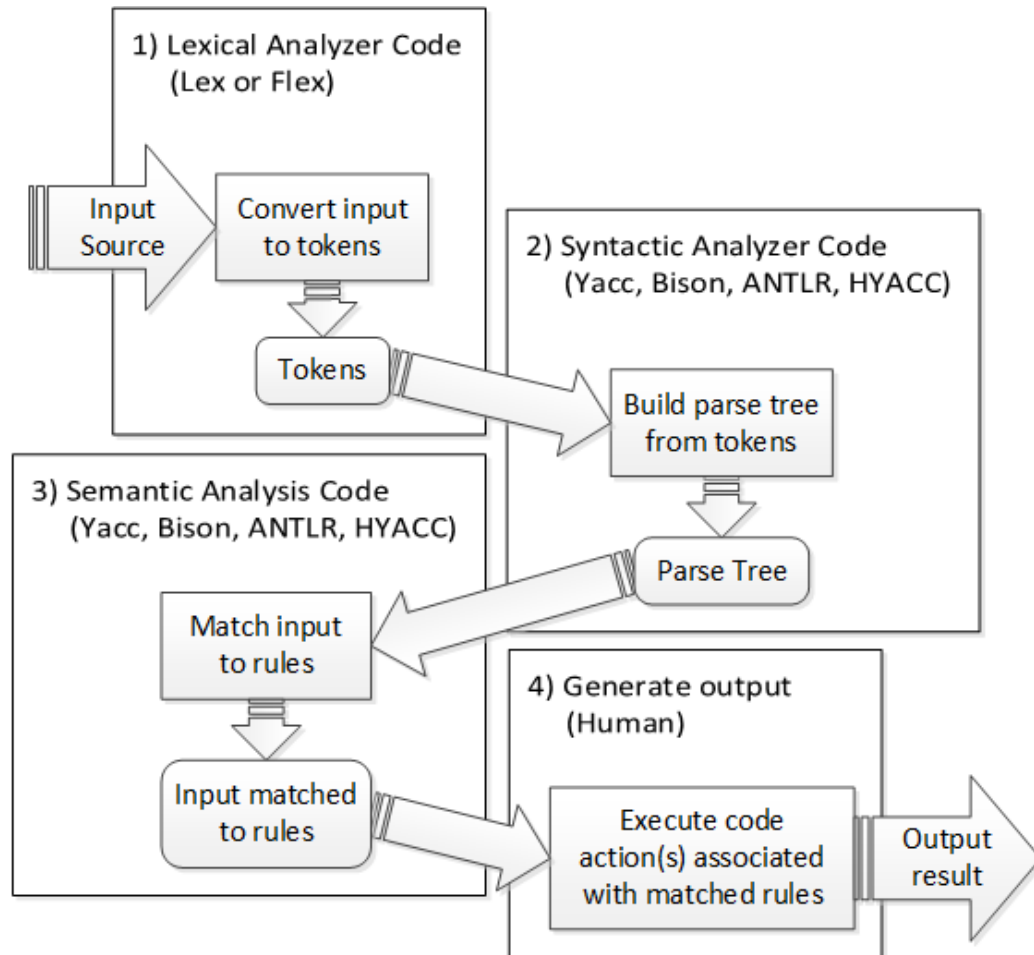All of these parser generators provide an effective means to reduce human interaction with code (figure 2.2). They have the added benefit of generating logical and syntactically correct code as long as the grammar is correct. On the other hand, the input grammars used by these parser generators cannot be used for lexical analysis of the parser input. Moreover, code actions that generate output are not provided as part of the input grammar. These actions must be manually written and inserted into the code generated by the parser generator.

2.2 Model-Based Parser Code Generation

Model-based parser generators provide an alternative to traditional parser generators using a model-based language specification. This kind of specification is explained by [23] and [24] as starting with an abstract syntax model (ASM) embodying the main concepts of a given language. One or more concrete syntax models (CSMs) are created from this ASM. Each CSM defines specific details about the language being modelled. Elements within the ASM can then be converted into their concrete representation using a mapping of the ASM to its CSM(s). This mapping is created by annotating the ASM with pertinent constraint metadata. With this mapping in place, any changes to the ASM by the user will cause the language processor to automatically update to reflect those changes.

Since grammar specifications are not needed by model-based parser generators they can offer several advantages over traditional parser generators [24]:

- An easier language design process. Language design is decoupled from language processing because the language grammar is automatically generated.

- Non-tree structures can be modeled. This is different from traditional parser generators that force users to model a tree structure.

- Some semantic checks like reference resolution can be automated.

- Handles references between language elements, as opposed to the traditional way of resolving references manually using a symbol table.

The unique capability of model-based parser generators to use an ASM as input rather than a grammar make it possible for them to behave in two different ways [24]:

1. An ASM representing a tree structure (Abstract Syntax Tree) causes a model-based parser generator to behave like a traditional parser generator.

2. An ASM representing a non-tree structure (Abstract Syntax Graph) causes a model-based parser generator to use reference resolution.

ModelCC is a model-based parser generator that accepts an annotated conceptual model as its input rather than a context-free grammar [23]. ModelCC uses this annotated model to generate a parser written in Java that automatically instantiates the language conceptual model [23, 25].

The process of generating a parser using ModelCC as explained by [24] begins with an ASM that is defined by classes representing language elements and the relationships between them. Metadata annotations are added to these language elements and their corresponding relationships to produce an ASM to CSM mapping. Reference resolution is performed to match referenced objects to their equivalent object instantiations. A parser is then generated that automatically instantiates the conceptual model.

ModelCC delivers an effective way to decrease human interaction with code by generating grammar code and lexical analysis code from an input model specification. On the other hand, it is similar to other parser generators because it does not implicitly support specific input/output methods such as UDP/IP, etc.

CHAPTER 3

Design

The TEBNF language makes it possible to convert a software design expressed as a state machine and convert it into a specification that can be provided to the TEBNF code generation tool. The TEBNF code generation tool generates C++ 11 code from TEBNF grammars that can be built by a C++ compiler into a functioning console application. This chapter discusses the TEBNF language, design decisions, the TEBNF code generation tool, the architecture of the tool, and the code generation process.

3.1 The TEBNF Language

The TEBNF specification is composed of language constructs called elements. A TEBNF grammar is the combination of these elements and their contents to define an application. The primary function of each element type is to convey one specific functionality. One advantage of this approach is scalability. The complexity of each element is reduced because each element does one thing only. This follows the established design paradigm of making something do only one thing but do it well. There are five kinds of elements in the TEBNF specification:

- Input methods: a method of receiving data as input.

- Output methods: a method of outputting data.

- Grammar sections: parses input data.

- Actions: action code to execute using data from other elements.

- State transition tables: defines execution path.

Within each element are separate constructs known as subelements. Different kinds of subelements exist for each element type, and are enumerated in detail in appendix A.

The glue that brings these elements together to describe an application is two-fold: 1) static variables that can be defined within grammar elements and used by other elements to get and store information, and 2) state transition table elements that use all of the other elements in a grammar to express the order of task execution.

3.1.1 Design Decision: Singleton Paradigm

One of the primary goals of TEBNF is to reduce the difficulty of translating high-level designs into grammars. Keeping TEBNF simple means that developers are spending more time designing how an application will be used rather than worrying about many of the details associated with translating their design into code. Elements have been designed to behave as singletons in TEBNF, as well as the code generated from them.

Multiple reasons and advantages exist for treating TEBNF elements and the C++ classes generated from them as singletons:

- Lazy instantiation. Singleton C++ classes make it possible to avoid allocating memory for element class objects until they are actually used in the generated code. This is different from static initialization that allocates memory to a variable at the time of declaration.

- Thread synchronization. Singletons can yield the best results in situations where multiple threads try to access the same resource. This applies to code generated from a TEBNF grammar because it is possible for multiple state table threads to need access to the same subelement or static variable class members at the same time.

- Instance control. A singleton prevents other objects from instantiating their own copies of the singleton object, ensuring that all objects access the same instance. This simplifies the design process in TEBNF because grammar writers know exactly what they are accessing in their grammar.

- Flexibility. Generated singleton element classes control the instantiation process. This control over the instantiation process gives them the flexibility to change the

process. Since TEBNF purposely abstracts these details from grammar writers, it is advantageous for each element class to control the way it is instantiated based on how it is used/interacts in the generated application code.

3.1.2 Design Decision: Adapting TEBNF

Some complexities in the TEBNF language were exposed as the code generation tool was written. When encountered, adaptations to TEBNF were made to better represent their nature and interaction of these complexities within the language. Design decisions made with respect to TEBNF elements are discussed in 3.1.3 and 3.1.4.

3.1.3 Design Decision: Input and Output Elements

Console applications typically get string or numerical input from users by displaying questions and waiting for the user to type an answer. Attempting to determine the type of input information in generated code could lead to incorrect interpretations of the input type at run time. One technique that was considered to avoid this problem is to directly tie grammar subelements or static variables to input subelements. This method proved to be unwieldy because grammar elements change during the natural process of writing a TEBNF grammar. Simply changing the name of a grammar subelement would require that name to be changed in every input subelement tied to it. This problem could be compounded as other input elements are created that contain subelements tied to the same grammar subelements or static variables. To resolve this problem, console input subelements tie each question string to a type (appendix A). The responsibility of knowing the expected input type falls to the grammar writer rather than a risky prediction.

Usage of console output elements in TEBNF was found to be simpler than console input. This is due to the fact that generated console output code performs a simple passing of output data to a C++ std::cout statement. Because std::cout easily handles outputting of numerical and string data, there is no need for special handling in TEBNF.

Support for sending and receiving data over UDP is achieved in TEBNF using UDP elements. The TEBNF language uses UDP I/O elements to abstract most of the details

involved with setting up and using UDP sockets. Generated UDP I/O element classes prompt users for an IP address and port number before initiating UDP communication. It became apparent that a way was needed to indicate that a UDP output element should send data on the same socket instance used by an existing UDP input element to receive data. The "AS" keyword expresses this relationship between two I/O elements of the same type. The "AS" relationship defines an element that shares all of the same characteristics as another I/O element of the same type. Because TEBNF elements and their respective C++ element classes are singletons, all that is needed to represent this case in generated C++ code is a type definition of the I/O element in question (typedef). This reduces the number of C++ classes generated.

3.1.4   Design Decision: Actions Elements

Actions elements in TEBNF share two similarities to C++ functions. The first similarity is the ability to have one or more parameters, allowing arguments to be passed inside state transition table elements. Second, actions elements can contain multiple instructions. Actions elements also allow grammar writers to access subelements and/or static variables found in any grammar element. The syntax for accessing subelements defined inside other grammar elements can be found in appendix A.

Unlike C++ functions, actions elements cannot return values. TEBNF purposely abstracts this kind of complexity from grammar writers. This kind of functionality was intended to be expressed in state transition table elements. In TEBNF, a return value is expressed in a state transition table when a state's input or condition criteria is met, resulting in data being routed (returned) to an output and/or optionally transitioning to a different state. This supports their intended usage as the action(s) component of one or more states in a state machine. A side effect of actions elements not returning values is that they cannot be used as the input or condition of a state within a state transition table element. This is due to the fact that the resulting type of a states input or condition must be a Boolean. This also illustrates one of the ways that TEBNF abstracts complexity through state machine design.

Actions elements are essentially C++ code blocks that allow direct reference to TEBNF subelements and static variables. This means they have loose parsing requirements compared to other TEBNF elements so that the chosen C++ compiler can catch complex errors in actions element code.

3.2 The TEBNF Code Generation Tool

This report presents a prototype code generation tool that generates the lexical analysis, parsing, and actions code of a basic console application using a single TEBNF grammar as input. Generated applications accept user input where necessary and can provide meaningful status. The tool outputs a set of classes that:

1. Accept input data through console, file, or UDP/IP.

2. Provide a set of functions that unmarshal raw data into human-readable types such as numbers and strings, and can marshal it back into its original form.

3. Use these unmarshal functions to match input data to pattern(s) specified in the grammar and convert them to human-readable values.

4. Run one or more state machines with each on its own thread to receive data through input methods described in the grammar. As input data arrives, the state machine finds matches to grammar patterns and executes actions that produce the desired output.

5. Provide a console-based user interface that prompts for input as needed and provides status.

The architecture of the TEBNF code generation tool consists of four stages, shown in figure 3.1. The TEBNF code generation tool is a console application that accepts three arguments in order:

1. Path of the file containing a TEBNF grammar.

2. Path of directory where the tool will write generated code.

3. The name to give to the generated application.
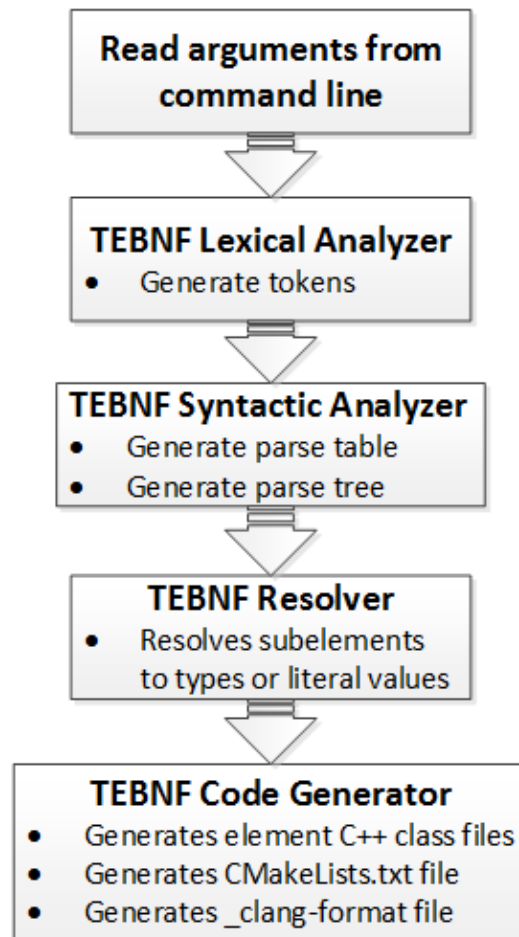
Figure 3.1: Architecture of the TEBNF prototype code generation tool.

Upon receiving these arguments, the tool reads the grammar file at the location provided to the tool. The grammar is lexically analyzed by the TEBNF scanner to produce a list of tokens, with some metadata attached to some tokens as needed.

The TEBNF parser syntactically analyzes these tokens using a recursive descent algorithm to verify correctness of format. A pointer to each element discovered by the parser is then added to a table for quick look up and inserted into a parse tree. Subelements discovered during this parsing stage are added to their parent elements and subelements as they are found in the parse table and parse tree.

After all of the elements and subelements have been added to the parse table and parse tree, the TEBNF resolver traverses the subelements in the parse tree and their descendants to their furthest extent (leaves). This ensures that all subelements resolve to a terminal type or literal value.

After all elements and their subelements have been resolved, the TEBNF code generator iterates through each element in order of declaration in the input grammar, and generates a C++ class or other appropriate C++ code. The name of each C++ source file corresponds to the element it was generated from. A CMakeLists.txt file is generated with these C++ source files so that CMake can be used to generate Microsoft Visual Studio 2013 solution and project files. For convenience, a _clang-format file is generated so that any clang-formatting (optional) follows the intended format.

3.3   Lexical Analysis and Parsing Code Generation

The TEBNF code generation tool generates a class for each input/output (I/O) element (method) defined in a TEBNF grammar (see appendix A). These I/O classes perform no lexical analysis. They only receive input and/or send output. Supported I/O methods are as follows:

- Console

- File

- UDP/IP

These I/O methods are a powerful feature of TEBNF and the TEBNF code generation tool because the tool automatically integrates the code to do I/O using these methods. The complexities of their usage are abstracted by the tool, which is one of the key advantages of TEBNF and the TEBNF code generation tool. Contrast this to the most common code generation tools, which do not provide this built-in I/O capability.

Lexical analysis and parsing is performed by the generated code in one step rather than separate steps. This is possible because of the way patterns are described in TEBNF

grammar elements (see appendix A). A typical grammar element pattern is composed of groupings of bytes broken into sub-groupings of bytes. These sub-groupings can be translated into specific types (e.g. numbers, and strings) and literal values. The size in bits or bytes is defined in the grammar element based on its type.

Matching user-defined TEBNF grammar patterns to incoming data requires that one or more literal values be defined somewhere in the grammar. The value of a literal value makes it possible to find it in the input data. The size and type of that initial literal value is defined implicitly or explicitly in the grammar (e.g. 4-byte integer, etc.). Given the initial reference offset α_f of the literal f and its size z_f, the offset of the literal or type immediately following it is defined as α_{f+1}, where $\alpha_{f+1} = \alpha_f + z_f$. The offset of the literal or type defined immediately before f can be defined as α_{f-1}, where $\alpha_{f-1} = \alpha_f - z_{f-1}$ and $z_{f-1} \geq \alpha_f$. The data offsets of subsequent literals and/or types found before and after the initial reference offset are calculated based on the one after and before it, respectively.

When multiple patterns must be matched to incoming data, a separate grammar element must be written to describe each pattern. This design makes it possible to refer to any given pattern using its grammar element name, making it easy to distinguish from other grammar patterns in the same TEBNF grammar.

Grammar elements are translated by the prototype code generation tool into classes that serve a three-fold purpose. Each grammar class can (1) unmarshal raw input data into specific user-defined types, (2) verify matches to byte patterns in the raw input data, and (3) marshal the values stored in the grammar class back into their original form and byte ordering. Patterns in data arriving through TEBNF input methods are located by the state machine using unmarshal functions of grammar element classes. As data arrives through an input method, patterns are recognized in the data by the grammar classes and simultaneously unmarshaled into the data types of those classes. This means lexical analysis and parsing are performed in the same step, using a single TEBNF input (figure 3.2). This approach is different from traditional parser code generation methods. Traditional methods require the use of a lexical analyzer generator tool and parser generator tool; each with
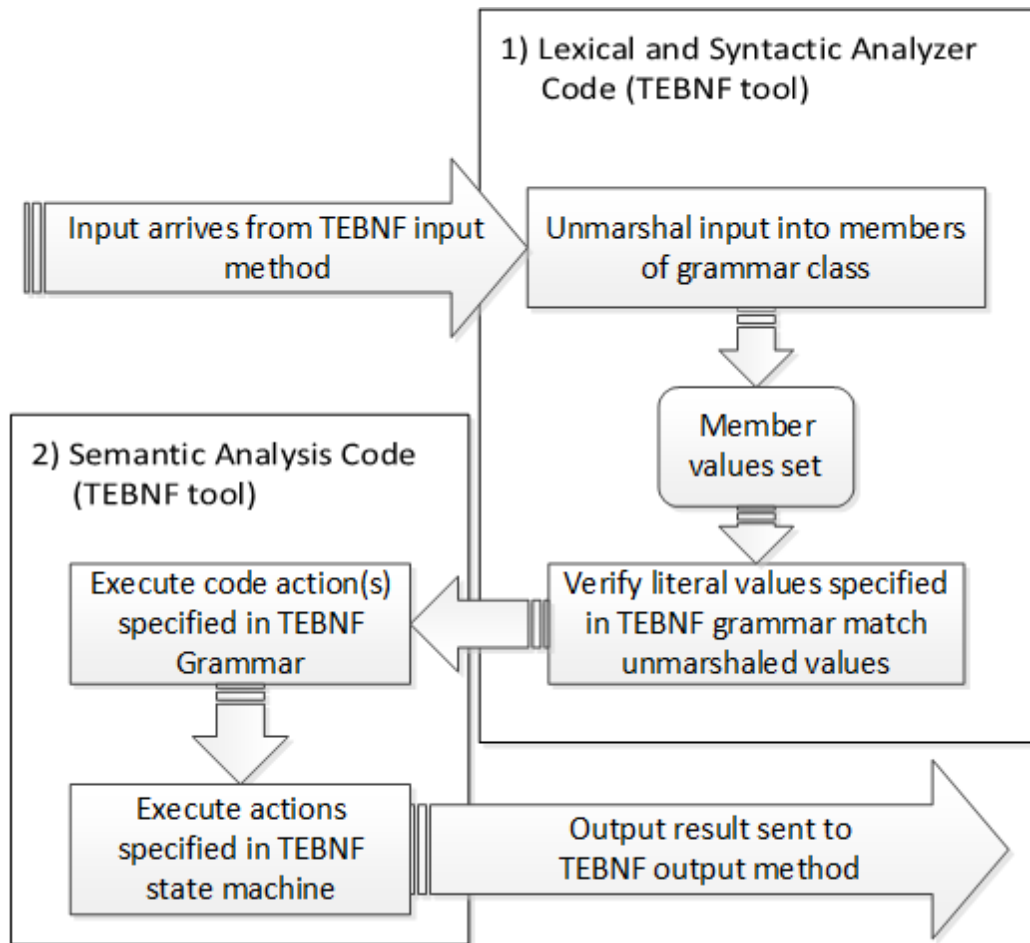
Figure 3.2: Execution path for code generated by the TEBNF code generation tool.

their own input specification formats (see figure 2.2).

3.4   State Machine and Actions Code Generation

Path(s) of execution are defined in TEBNF using state machines, as shown in figure 3.2. State machines are represented in TEBNF using state transition table elements (see appendix A). A state machine in TEBNF describes the order of tasks executed by a single application thread. States are divided into a series of six steps. These six steps are given below, in the order they are expressed in TEBNF, which is the same order they are evaluated by the generated code:

1. A unique name that identifies the state and allows other states to reference it

2. A Boolean condition that can be either a grammar to match with input data or an explicit Boolean condition.

3. The method of input for receiving the input data (console, file, or UDP/IP).

4. The next state to jump to if the Boolean condition provided in the second step evaluates to true.

5. An output to send through the output method provided in the sixth step, an explicit code action to execute, or nothing.

6. The output method to send output specified in the fifth step. If an action or nothing was provided for the fifth step, then this step is empty as well.

These state steps shape the way TEBNF elements interact with each other and input data by determining (1) what input methods data is received on, (2) what parts of the received data match defined grammar patterns, (3) what method is used to output that data, and (4) what code actions are executed.

CHAPTER 4

Implementation

This chapter focuses on the implementation of applications using TEBNF and the TEBNF code generation tool. The chapter is divided into four sections. The first section will showcase a real world example using a TEBNF grammar to parse data from a NITF file. The second section will walk through the implementation of two test cases. The third section will cover the testing and verification of these test cases. The fourth section of the chapter will cover the results of testing.

4.1 NITF File Parsing: A Real World Example

Software developers are oftentimes tasked to write software that parses and processes data in different formats. A well-known example is word processing software. Word processing software must be able to open and read documents structured in its own proprietary format and others (e.g. pdf, txt, etc.). Creating software that reads specific data formats requires access to documentation detailing the exact structure of each format.

An example data format read by different applications is the National Imagery Transmission Format (NITF) version 2.1 file format. The NITF 2.1 file format is part of a suite of standards established by the United States (US) Government for formatting digital imagery [26]. Documentation detailing the NITF 2.1 standard is freely available for download from the Geospatial Intelligence Standards Working Groups website [27]. This file format is compatible with software used by members of the Intelligence Community (IC), which includes the US Department of Defense (DOD) [26]. The VANTAGE$^{\text{TM}}$ software suite produced by the Space Dynamics Laboratory and the SOCET Services suite produced by BAE Systems are all capable of reading and exploiting NITF 2.1 formatted files [28, 29].

As explained by [26], the stated purpose of the NITF 2.1 file format is to provide the
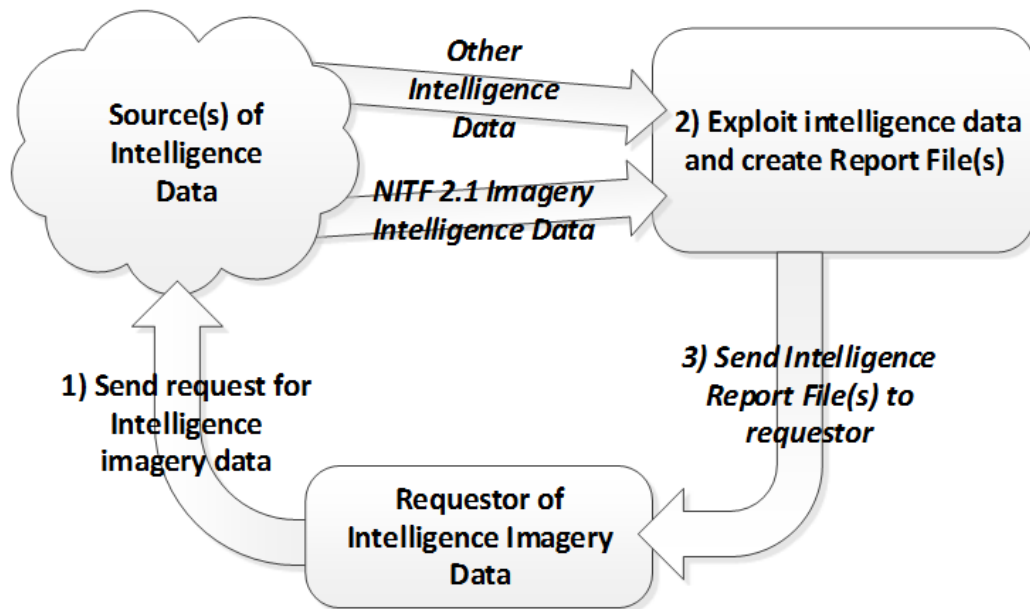
Figure 4.1: CONOPs for NITF 2.1 format files.

IC an interoperable means of transmission and/or storage of electronic imagery data. The NITF 2.1 format is intended to be used in the dissemination of imagery derived intelligence data. A general concept of operations (CONOPs) involving NITF data is shown in figure 4.1. First, imagery data in NITF 2.1 format is requested by an IC member. Second, the NITF data is received and combined with other collateral information to create intelligence report file(s) and/or products containing the requestors Essential Elements of Information (EEI). Third, these report files and/or products are given to the requestor of that intelligence information.

Exploiting NITF 2.1 files requires a detailed understanding of the format. With this understanding, it is possible to create software than can parse NITF 2.1 files to find information of interest (i.e. EEI), generate report files, and/or products.

The NITF 2.1 file format is composed of a file header and one or more segments. Each segment contains a subheader with data fields. Each data field has a specific size and format (depending on the type of field) and is located at a specific byte offset within the file.

Conditional data and/or data characteristics can be added to NITF 2.1 files. This

flexibility to extend the format is done using conditional fields in the file header and sub-headers indicating the existence of Tagged Record Extensions (TREs) and Data Extension Segments (DES). Tagged Record Extensions contain data fields, while extension segments can contain data in new formats.


```
GRAMMAR @use00a
    cetag = 'U','S','E','0','0','A';
    cel = '0','0','1','0','7';
    angle_to_north = UNSIGNED FLOAT_STR_24;
    mean_gsd = UNSIGNED FLOAT_STR_40;
    blank = BYTE;
    dynamic_range = UNSIGNED INT_STR_40;
    reserved1 = BYTE{3};
    reserved2 = BYTE;
    reserved3 = BYTE{3};
    obl_ang = UNSIGNED FLOAT_STR_40;
    roll_ang = BYTE{6};
    reserved4 = BYTE{12};
    reserved5 = BYTE{15};
    reserved6 = BYTE{4};
    reserved7 = BYTE;
    reserved8 = BYTE{3};
    reserved9 = BYTE;
    n_ref = UNSIGNED INT_16;
    rev_num = UNSIGNED INT_STR_40;
    n_seg = UNSIGNED INT_STR_24
    max_lp_seg = UNSIGNED INT_STR_48;
    reserved10 = BYTE{6};
    reserved11 = BYTE{6};
    sun_el = UNSIGNED FLOAT_STR_40;
    sun_az = UNSIGNED FLOAT_STR_40;
END
```

USE00A - Exploitation Usability Extension Format TRE			
Field	Name/ Description	Size	Value Range
CETAG	Unique Extension Identifier	6	USE00A
CEL	Length of CEDATA field	5	00107
ANGLE_TO_NORTH	Angle to North	3	000 to 359
MEAN_GSD	Mean Ground Sample Distance	5	000.0 to 999.9
	reserved	1	space
DYNAMIC_RANGE	Dynamic Range	5	00000 to 99999
	reserved	3	spaces
	reserved	1	space
	reserved	3	spaces
OBL_ANG	Obliquity Angle	5	00.00 to 90.00
ROLL_ANG	Roll Angle	6	+90.00
	reserved	12	spaces
	reserved	15	spaces
	reserved	4	spaces
	reserved	1	space
	reserved	3	spaces
	reserved	1	spaces
N_REF	Number of Reference Lines	2	00 to 99
REV_NUM	Revolution Number	5	00001 to 99999
N_SEG	Number of Segments	3	001 to 999
MAX_LP_SEG	Maximum Lines Per Segment	6	000001 to 999999
	reserved	6	spaces
	reserved	6	spaces
SUN_EL	Sun Elevation	5	-90.0 to +90.0, or 999.9
SUN_AZ	Sun Azimuth	5	000.0 to 359.0, or 999.9

Figure 4.2: TEBNF Implementation of the NITF 2.1 USE00A TRE.

Gleaning data from NITF 2.1 formatted files for exploitation (see figure 4.1) is best achieved using applications that can read it. Figure ?? demonstrates how this can be achieved using a TEBNF grammar. The table shows a side-by-side comparison of the NITF 2.1 USE00A (Exploitation Usability Extension Format) TRE compared to its equivalent implementation in TEBNF. Each data field for this TRE is compared to its equivalent TEBNF grammar expression. This comparison highlights several important advantages of

using TEBNF to implement parsers of specifications like NITF:

- Convenience. There is no need to keep track of the offset of each data field because it is determined based on the size of each type.

- Readability. TEBNF grammar syntax looks very similar to actual specifications as shown in figure **??**, making it is easier to understand.

- Self-documenting. Since TEBNF grammar syntax ties the size of each data field to the size of the subelement type, the grammar helps to document itself.

4.2 Test Cases

In order to verify the functionality of the TEBNF language, two case studies were implemented. Both case studies showcase the strengths of TEBNF and the TEBNF code generation tool. Further case studies are possible because TEBNF is Turing complete (see appendix B).

4.2.1 Basic Calculator

The first test case was to create a basic calculator console application that supports addition, subtraction, multiplication, and division of integers. Producing this calculator required the tool to generate code that:

1. Parses numeric data received over a console input.

2. Makes calculations based on the data received from that console input.

3. Sends the results of those calculations to console output.

4. Allows the user to exit when finished.

The calculator begins with a step executed once at the beginning of the program. This initial step prompts users to enter a number, which is then saved as an initial result value because all subsequent math operations are executed against it. From this point onward, the calculator repeats a cycle that 1) asks for a number, 2) asks for a math operator, 3)

applies that math operator against the saved result and the last number entered, 4) saves the result, and 5) displays that result. This cycle then repeats until the user enters a $=$ operator, which then displays the result and exits the program.

Input for the calculator is achieved with a single TEBNF console input element containing two prompt values. The first is used for prompting the user to enter a number, accepting a signed 64-bit integer. The second one is used for prompting the user to enter a single character (math operator). Outputting the result of math operations is accomplished with a single TEBNF console output element.

Acceptable input values for the calculator are limited to integers or one of five math operator characters (+, -, *, /, =). TEBNF grammar elements are defined for each math operator. Another grammar element was created to represent a signed 64-bit integer, providing a place to store integers as they are unmarshaled from input. The saved result value is represented as a static variable within the number grammar element, serving as a place to store the result of each math operation. An actions element was created for each of the supported math operators. Each actions element adds, subtracts, multiplies, divides, or sets the saved result using the last unmarshaled integer.

The last TEBNF element included in the grammar describes the calculator as a state machine utilizing the elements described above to define the execution path of the calculator. The calculator state machine is represented with five states, as shown in figure 4.3.

The first state of the machine is a special case entered only once at the beginning of execution. This initial state is a necessary special case that sets the ongoing result value for subsequent math operations. In this state the user is prompted to enter the initial number via the console input element. The number grammar element unmarshals this input value as a signed 64-bit integer and retains a copy for later use. The state table then executes an actions element. This actions element sets the result static variable equal to the unmarshaled value and transitions to the next state.

After setting the result value, the machine transitions to the first state of the main cycle and uses the console input element to prompt the user to enter a number again.
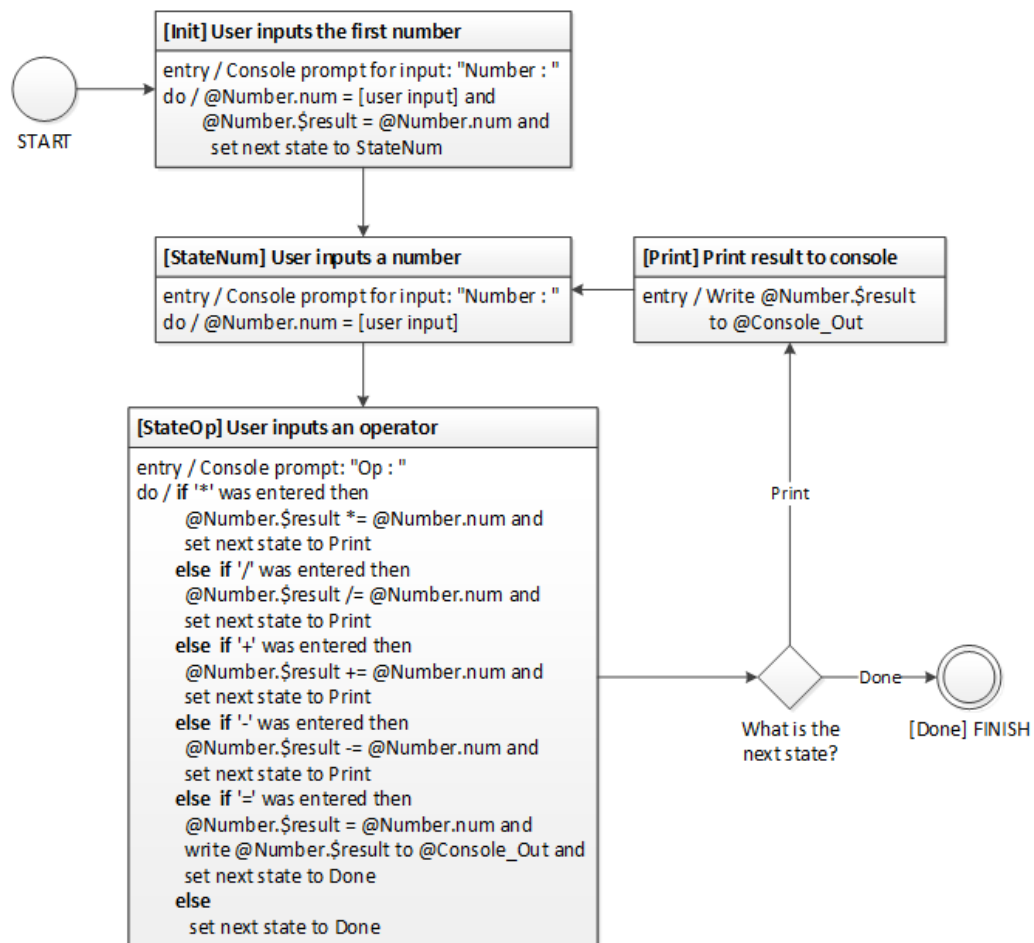
Figure 4.3: State machine representing a calculator.

This number is also unmarshaled from input and stored as before. The state machine then transitions to the next state.

This state uses the console input element to prompt the user to enter one of the supported math operators. The state machine does this by moving through a series of else-if states. Each state uses one of the math operator grammar elements to check the same console input for a match. A match occurs when a grammar successfully unmarshals the input value. Upon success, the actions element tied to that state is executed by the machine, performing one of the supported math operations. At this point, the machine transitions to the next state, which is determined from the operator entered by the user. If

a '=' operator was entered, the result is written to the console and the state machine exits. Otherwise the machine transitions to the print state and writes the result to the console. The print state then transitions to the first state of the main cycle, which continues to repeat until the user enters a '=' to exit.

Table 4.1: Sample input and output data for the calculator.

Operand	Operator	Result
5		
10	+	15
95	-	-80
110	+	30
10	/	3
111	*	333
33	-	300
20	/	15
5	*	75
0	=	75

Suppose this calculator is run using the sample data in table 4.1. The first line in the table is the initial value 5. The second operand entered by the user is the number 10. The '+' operator is entered by the user, and the result displayed is 15. Entering the next value of 95 followed by the operator '-' displays the number -80. This cycle continues until the user enters the '=' operator.

4.2.2 NITF 2.1 UDP/IP File Transfer Client

The second test case is a file transfer client that receives NITF 2.1 files over UDP/IP upon requesting them from a file server. The server tells the client when it is done sending files so the client knows when to exit. This requires the tool to generate code for a file client that:

1. Asks the server to send it a file by sending it a message over a UDP/IP output.

2. Receives the NITF 2.1 file from the server over a UDP/IP input.

3. Uses the file length data field in the NITF 2.1 file to determine that it has received the entire file from the server as it was sent over UDP/IP.

4. Writes that file to disk using a file output.

5. Quits upon receiving a message from the server that says it is done sending files.

The NITF 2.1 file transfer client starts by prompting the user to enter the IP address and port that will be used for sending and receiving data over UDP/IP. After entering this information, the client immediately sends the message "send" to the server to request that it send a file. Once the server receives the "send" message from the client, it reads a NITF 2.1 file from disk and begins sending it to the client over UDP/IP. As the client receives data from the server, it continually checks to see if it has received the entire NITF 2.1 file from the server. The client knows a file transfer is complete when it has received the amount of data specified in the NITF files file length data field. The client then prompts the user to enter a path including the file name specifying where the file will be written to disk. After writing the file to disk, the client requests the next file from the server.

Allowed input values for the client are limited to a string for the IP address, an unsigned integer for the port, and strings for file paths. All other input to the client comes from the server through a TEBNF UDP/IP input element. A UDP/IP output element is used by the client to send data to the server. One file output element is used by the client for writing files to disk received from the server.

A grammar element was created to find NITF 2.1 files in data received over UDP/IP. The beginning of each NITF 2.1 file can be found by searching for the byte sequence "NITF02.10", which is always found at the beginning of each file. The NITF 2.1 file header has a data field containing the length of the file. This is leveraged by the TEBNF grammar element to calculate the end offset of each file. A second grammar element was created for finding the "done" message in incoming data. A third grammar was created to define the send message.

The file transfer client can be represented as a state machine with three states, as shown in figure 4.4. The first state sends the send message via the UDP/IP output element
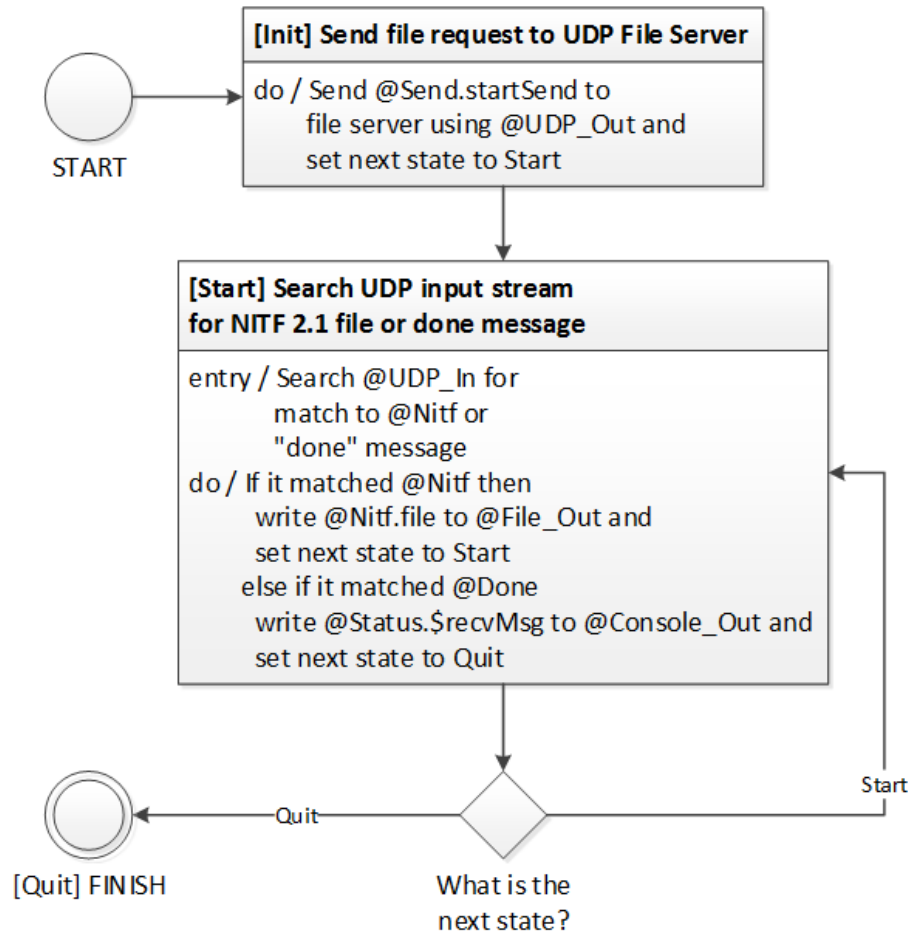
Figure 4.4: State machine representing a NITF 2.1 UDP/IP file transfer client.

to the file server. After sending this request message, the machine transitions to the second state.

The second state looks at data received from the file server over the UDP/IP input element to determine if it contains a NITF 2.1 file or the "done" message. If the incoming data contains a file, the state machine writes it to the file output element, which prompts the user for a path to write the file to. The state machine then transitions back to the first state which tells the server to send the next file. If the incoming data does not contain a file, an "else-if" case in this same state checks for the done message. If the done message was received, the machine writes a message to the console output telling the user that all

files have been received, after which the machine transitions to the quit state and exits.

In a hypothetical case, assume there are five NITF 2.1 files to be sent to a test case file client. Each file has a specific file size, as shown in table 4.2.

Table 4.2: Sizes of sample NITF 2.1 files sent to the test client.

NITF 2.1 File Size (bytes)
828710
4021118
912294
3516054
998822

The NITF 2.1 file server would be started and begin listening on its UDP/IP socket for the "send" request from a client. The test case file client is then started and immediately sends a "send" request to the server, and a file transfer begins. The UDP server reports the size in bytes of each file sent. The file client writes each file to local disk. The size in bytes of each file received corresponds to the size the files sent by the server.

4.3   Results

C++ 11 code was successfully generated by the TEBNF code generation tool for each of the test cases. The TEBNF code generation tool provides useful output when it generates code from a supplied input grammar. The tool displays information indicating when each stage of the code generation process finishes. Status for the generation stage displays the classes generated for each element. After the generation stage finishes and all of the code has been generated, the tool reports success and the number of element files generated. Console output from generating code for the calculator test case and the NITF 2.1 file client test case is shown in figures 4.5 and 4.6.

A CMakeLists.txt file was correctly generated by the TEBNF code generation tool for each test case, and CMake version 3.0.2 was run using those CMakeLists files to generate Microsoft Visual Studio 2013 solution and project files. The project files generated by

Figure 4.5: TEBNF code generation tool output for the calculator test case.

Figure 4.6: TEBNF code generation tool output for the UDP NITF 2.1 file client test case.

CMake for both test cases were successfully opened and built in Microsoft Visual Studio 2013.

4.3.1 Calculator

The calculator test case executable was run using the test data from table 4.1. As expected, the input and output of the calculator (figure 4.7) matched what is defined in table 4.1. This means the behavior of the calculator test case matches the behavior defined in the TEBNF grammar it was generated from.


```
C:\windows\system32\cmd.exe
C:\src\OUTPUT_FILES\Calculator\_bld\Release>Calculator
Number: 5
Number: 10
Op: +
15
Number: 95
Op: -
-80
Number: 110
Op: +
30
Number: 10
Op: /
3
Number: 111
Op: *
333
Number: 33
Op: -
300
Number: 20
Op: /
15
Number: 5
Op: *
75
Number: 0
Op: =
75

C:\src\OUTPUT_FILES\Calculator\_bld\Release>
```
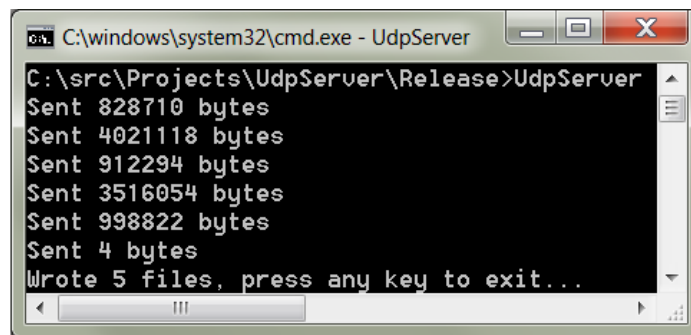
Figure 4.7: Running the calculator test case.

4.3.2 NITF 2.1 File Client

The NITF 2.1 file client test case executable was run using the five sample NITF 2.1 files whose sizes are found in table 4.2. A NITF 2.1 file server was created that listened for client "send" requests over a UDP/IP socket on localhost port 10042. This server was started, then the test case client was started and configured to send requests and receive

file transfers on localhost port 10042. The expected outcome occurred, with the server successfully sending five files as indicated by the first five send messages output by the server in figure 4.8. This was followed by the last four byte "done" message is sent to notify the client it was done sending. The same five files were received by the client which prompted for a file name to save each file as shown in figure 4.9. After saving the files, the client exited because the server had sent the "done" message.
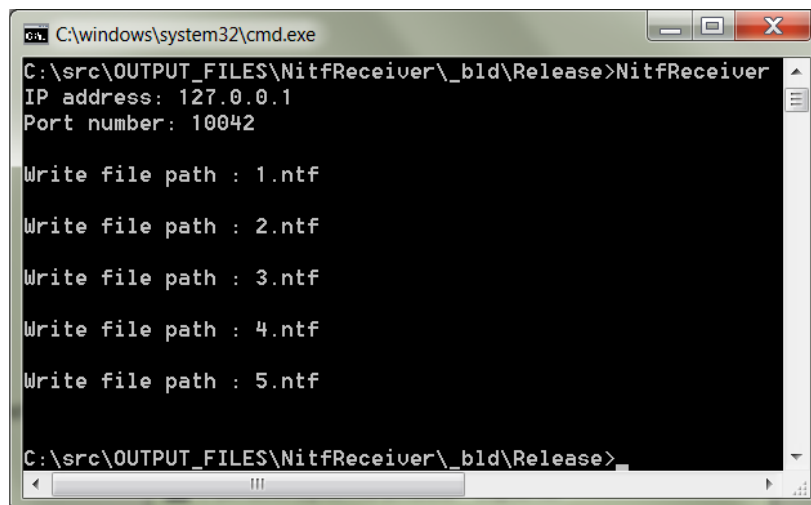
Figure 4.8: Running the NITF 2.1 file server test case.

Figure 4.9: Running the NITF 2.1 file client test case.

The sizes of the files received were an exact match to the file sizes listed in table 4.2, as the output of the server and client test case executables show in figures 4.8 and 4.9, respectively. This verifies that the behavior of the NITF 2.1 client test case matches the behavior defined in the TEBNF grammar it was generated from.

CHAPTER 5

Future Work

Code generation using TEBNF creates several areas for future work.

- Additional I/O methods. Some of these could include support for TCP/IP, MySQL databases, and others.

- A TEBNF Integrated Development Environment (IDE). The IDE could provide intelligent code completion or incorporate a drag-and-drop interface for adding TEBNF elements to a grammar.

- Aspect oriented code generation. This could involve generating code that leverages aspects. It could also involve the integration of aspects into the specification of TEBNF itself.

- Exploring the use of template meta-programming in generated C++ code. This programming paradigm emphasizes the use of types, similar to TEBNF.

- Software Mining for Graphical User Interface (GUI) code generation [30,31]. TEBNF lends itself to automatic generation of GUI code because of its merging of different input specifications into one. Kennard proposed the usage of a technique called runtime data mining to generate GUIs [30]. Software mining [31] is a form of data mining that focuses on the inspection of software information characteristics:

- Static characteristics: e.g. source code files and database schemas.

- Runtime characteristics: e.g. polymorphic data-types, other data values, reading and modification of an instantiated objects current state.

CHAPTER 6

Conclusion

TEBNF merges the strengths of declarative and imperative programming paradigms into one format. A prototype code generation tool was presented that accepts a TEBNF grammar as its sole input specification. The prototype code generation tool demonstrated how TEBNF makes it possible to:

- Integrate lexical analysis and parsing input specifications into one format.

- Generate C++ classes that handle I/O over console, file, or UDP/IP, and abstract the complexities of their usage through TEBNF.

- Generate C++ classes that simultaneously unmarshal and match incoming input data to TEBNF grammar patterns.

- Describe the execution path of an application that accepts input over various I/O methods, is able to parse that input, use it, and send output over various I/O methods.

Software developers using TEBNF are able to focus on what they want to implement rather than how to represent their designs in a language like C++. TEBNF embodies the idea of using additional levels of indirection to solve problems in computer science.

REFERENCES

[1] B. Livshits and T. Zimmermann, "Dynamine: Finding common error patterns by mining software revision histories," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13, pp. 296–305. New York, NY, USA: ACM, 2005 [Online]. Available: `http://doi.acm.org/10.1145/1081706.1081754`.

[2] N. G. Leveson, "Software engineering: Stretching the limits of complexity," *Commun. ACM*, vol. 40, no. 2, pp. 129–131, Feb. 1997 [Online]. Available: `http://doi.acm.org/10.1145/253671.253754`.

[3] C. Smidts, "A stochastic model of human errors in software development: impact of repair times," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pp. 94–103. IEEE, 1999.

[4] T. Nakajo and H. Kume, "A case history analysis of software error cause-effect relationships," *Software Engineering, IEEE Transactions on*, vol. 17, no. 8, pp. 830–838, 1991.

[5] V. Sinha, *Using diagrammatic explorations to understand code*. Ph.D. dissertation, Massachusetts Institute of Technology, 2008.

[6] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, p. 426, 2005.

[7] K. P. Boysen and G. T. Leavens, "Automatically generating consistent graphical user interfaces using a parser generator," 2005.

[8] I. Groher and S. Schulze, "Generating aspect code from uml models," in *The 4th AOSD Modeling With UML Workshop*, 2003.

[9] R. Brooker, I. MacCallum, D. Morris, and J. Rohl, "The compiler compiler," *Annual review in automatic programming*, vol. 3, pp. 229–275, 1963.

[10] S. C. Johnson, *Yacc: Yet another compiler-compiler.* Bell Laboratories Murray Hill, NJ, 1975, vol. 32.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini, "Reactive imperative programming with dataflow constraints," *ACM SIGPLAN Notices*, vol. 46, no. 10, pp. 407–426, 2011.

[12] J. W. Lloyd, "Practical advantages of declarative programming," in *Joint Conference on Declarative Programming, GULP-PRODE*, vol. 94, p. 94, 1994.

[13] D. K. Gifford and J. M. Lucassen, "Integrating functional and imperative programming," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86, pp. 28–38. New York, NY, USA: ACM, 1986 [Online]. Available: `http://doi.acm.org/10.1145/319838.319848`.

[14] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

[15] T. Niemann, "Lex & yacc tutorial" [Online]. Available: `http://epaperpress.com/lexandyacc`.

[16] D. Spinellis, "Another level of indirection," *Beautiful code: leading programmers explain how they think*, pp. 279–291, 2007.

[17] J. Aycock, "Why bison is becoming extinct," *Crossroads*, vol. 7, no. 5, pp. 3–3, 2001.

[18] A. Demaille, J. E. Denny, and P. Eggert, "Bison - gnu parser generator," 2012 [Online]. Available: `http://www.gnu.org/software/bison`.

[19] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[20] X. Chen and D. Pager, "Full lr (1) parser generator hyacc and study on the performance of lr (1) algorithms," in *Proceedings of The Fourth International C* Conference on Computer Science and Software Engineering*, pp. 83–92. ACM, 2011.

[21] X. Chen, "Tutorial: Lr(1) parser generator hyacc," 2014 [Online]. Available: `http://www2.hawaii.edu/~chenx/hyacc_tut`.

[22] Free Software Foundation, Inc., "5.6 reduce/reduce conflicts," 2014 [Online]. Available: `http://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html`.

[23] L. Quesada, F. Berzal, and J.-C. Cubero, "A language specification tool for model-based parsing," in *Intelligent Data Engineering and Automated Learning-IDEAL 2011*. Springer, 2011, pp. 50–57.

[24] L. Quesada, F. Berzal, and J.-C. Cubero, "A model-driven parser generator, from abstract syntax trees to abstract syntax graphs," *arXiv preprint arXiv:1202.6593*, 2012.

[25] The ModelCC Development Team, "Modelcc," 2015 [Online]. Available: `http://www.modelcc.org`.

[26] The NITFS Technical Board (NTB), "Department of defense interface standard national imagery transmission format version 2.1 for the national imagery transmission format standard," October 2006 [Online]. Available: `http://www.gwg.nga.mil/ntb/baseline/docs/2500c`.

[27] Geopspatial Intelligence Statndards Working Group, "Geopspatial intelligence statndards working group," 2015 [Online]. Available: `http://www.gwg.nga.mil/ntb/baseline/docs/2500c`.

[28] Space Dynamics Laboratory, "C4isr deployed programs," 2015 [Online]. Available: `www.sdl.usu.edu/capabilities/c4isr`.

[29] BAE Systems, "Socet services," March 2010 [Online]. Available: `http://www.geospatialexploitationproducts.com/content/products/socet-services`.

[30] K. Richard and S. Robert, "Application of software mining to automatic user interface generation," 2008.

[31] R. Kennard and J. Leaney, "Towards a general purpose architecture for ui generation," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1896–1906, 2010.

[32] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[33] V. Kahlon, F. Ivani, and A. Gupta, "Reasoning about threads communicating via locks," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds. Springer Berlin Heidelberg, 2005, vol. 3576, pp. 505–518 [Online]. Available: `http://dx.doi.org/10.1007/11513988_49`.

[34] S. Kepser, "A simple proof for the turing-completeness of xslt and xquery." in *Extreme Markup Languages*®. Citeseer, 2004.

[35] E. F. Moore, "A simplified universal turing machine," in *Proceedings of the 1952 ACM National Meeting (Toronto)*, ser. ACM '52, pp. 50–54. New York, NY, USA: ACM, 1952 [Online]. Available: `http://doi.acm.org/10.1145/800259.808993`.

[36] Y. Rogozhin, "Small universal turing machines," *Theoretical Computer Science*, vol. 168, no. 2, pp. 215–240, 1996.

[37] C. E. Shannon, "A universal turing machine with two internal states," *Automata studies*, vol. 34, pp. 157–165, 1957.

[38] T. Neary, *Small universal Turing machines*. Ph.D. dissertation, Citeseer, 2008.

[39] K. P. Kumar, B. Kiranagi, and C. Bagewadi, "Turing machine operation-a checks and balances model."

APPENDICES

Appendix A

TEBNF Syntax and Usage

A.1 TEBNF Grammar Syntax

Like standard EBNF, Typed EBNF (TEBNF) is used to express a context-free grammar that consists of non-terminal production rules and terminal symbols that may or may not have a type. The typing of terminal symbols makes it possible to describe exactly how the input is recognized, yet preserves the simple syntax of EBNF. TEBNF provides a set of input and output methods, grammar rules, and actions, tied together to a set of states using a state transition table. It allows users to:

- Describe how the input data will be sent to the generated application (UDP, file, or in-memory).

- Describe when and how responses will be sent to the sender as needed by the specified protocol.

- Describe what the input data sent to the generated application will look like. This combines the traditional specification of lexical and syntactic analysis.

- Describe how the data will be processed by the generated application after the lexical and syntactic analysis stage.

- Describe the expected output of the application after it has been processed.

- Leverage existing knowledge of EBNF grammars rather than require the learning of a completely new format.

A.2 TEBNF Structure

The structure of TEBNF is composed of a set of elements. There are five kinds of elements in TEBNF: input methods, output methods, grammar sections, actions, and state transition tables. Collectively, these elements and their contents are known as a TEBNF grammar. Each TEBNF grammar requires at least one or more of each kind of element.

A.3 TEBNF Elements and Subelements

TEBNF elements (table A.1) contain one or more subelements. Subelements within TEBNF consist of production rules, typed terminals, non-typed terminals, literal values, operators, states, and static variables.

Table A.1: TEBNF Elements.

Element	Description
INPUT @name	specifier Input element of a specific input specifier.
OUTPUT @name	specifier Output of a specific output specifier.
GRAMMAR @name	Start of grammar section.
ACTIONS @name	Contains one or more actions.
STATES @name	Start of state transition table section.
END	End of element section.

Subelements are declared where they are first used. TEBNF infers what a subelement is by the way it is used. Each element requires a name be given to it. Element names are case-sensitive, can only contain visible characters, and are always prefixed by the '@' character as shown in example A.1:

$$GRAMMAR \ @packet \tag{A.1}$$

A.4 TEBNF Scoping Rules

Elements and subelements are directly accessible at the scope they are created, similar to the C++ language. Elements can be declared within the scope of other elements. Subelements exist within the scope of their respective elements. Each type of element has specific types of subelements that can be declared only within the scope of that type of element.

A.5 TEBNF Types

Association of types (table A.2) with terminal symbols offers an extra level of precision when matching specific patterns found in input data. These symbols are called typed terminals. TEBNF can infer the type of a terminal symbol because each type has two important components.

Table A.2: TEBNF symbols, production rules, non-typed terminals, and typed terminals.

Type	Description	Example
symbol	Production rule (non-terminal)	alphabet
symbol	Non-typed terminal (1 or more literals)	'a', 'b', 'c', 0xAB, "String literal"
#comment	Single-line comment	#This is a comment.
## comment ##	Multi-line comment	## This is a really, really, really long multi-line comment. ##
$var	Static variable	$myValue = 42 ;
type	Typed terminal	CHAR{0,} ;
BYTE	Represents a 8-bit byte	My_Kb = BYTE{1024} ;
INT_X	Represents an integer of size X bits	My_Int = INT_64;
INT_STR_X	Represents an integer as a string of size X bits	fileLength = UNSIGNED INT_STR_96;
CHAR	Represents a 8-bit character	MyChar = CHAR ; MyString = CHAR{128} ;
FLOAT_X	Floating-point number of size X bits	MyFloat = FLOAT_64 ;
UNSIGNED	Unsigned number terminal	UNSIGNED INT_16{2} ;

First, each type has an inherent size in bits or bytes. Whenever data is matched to a specific type the size is immediately known. Because the size of each type is known beforehand, the offset of the next symbol is immediately known. Second, each type inherently identifies how it should be used in a given context. TEBNF types are expressed by assigning a type to a terminal production symbol, as shown in example A.2:

$$payload \ = \ INT\_64 \tag{A.2}$$

TEBNF also has static variables, which are a kind of subelement that can assume the type of whatever is assigned to them. They are static because they have global visibility−i.e. they can be declared anywhere and accessed from any other element, regardless of where they were declared. Static variables are always prefixed by the '\$' character (example A.3):

$$\$payloads \ = \ payload \tag{A.3}$$

A static variable can become typed when a typed terminal is assigned to it. Production rules, terminals, and literals can also be assigned to static variables. This capability makes static variables the most flexible subelement available in TEBNF.

A.6 TEBNF Operators

There are three categories of operators in TEBNF. First are production rule operators (table A.3), which are used to build production rules. Second are arithmetic and comparison operators (table A.4). Arithmetic and comparison operators are a key difference between TEBNF and standard EBNF, which does not have them. Third is inter-element operators that perform operations on and/or between elements (table A.5). The only operator that falls in this category is the "AS" operator.

A.7 TEBNF Grammar Elements

Grammar elements contain production rules. Production rules are subelements of their containing grammar element. Terminal and non-terminal symbols have no prefix character, are made up only of alpha-numeric characters, and are case-sensitive.

Listing A.1: A TEBNF Grammar element.

```
GRAMMAR @Nitf
   # Describe the file to receive.
   sync = 'N','I','T','F','0','2','.','1','0';
```

Table A.3: TEBNF production rule operators.

Operator	Description	Example
=	Definitioni.e. is defined as	alphaNumericCharacter = letter | number ;
=	Grammar total length	= fileLength ;
,	Concatenation	twoLetters = letter , letter ;
;	Termination	twoVals = val1, val2 ;
|	Or	letter | number
|	State Transition table delimiter	Start | packet |
.	Element member access	@packet.payloadSize
{min,max}	Occurrence range	letter{0,}
len	Exact range, where len is the min and max.	letter{26}
[ ]	Array subscript	myChar[5]

```
skip1 = BYTE{333}; # FL offset is 342.
fileLength = UNSIGNED INT_STR_96; # 12 bytes
skip2 = BYTE\{,};
header = sync, skip1, fileLength;
file = header, skip2;
= fileLength; # Overall size of file.
END
```

Production rule subelements can only be declared within TEBNF grammar elements. Production rule subelements can be referred to directly using the containing elements name along with the dot operator followed by the subelement (symbol) name.

$$@packet.payloadSize \tag{A.4}$$

This format is similar to the way object-oriented languages like C++ and Java provide

Table A.4: TEBNF arithmetic and comparison operators.

Operator	Description	Example
==	Equality	$X == $Y
=	Assign	$x = 45 ;
+	Add	$x = $y + $z
-	Subtract	$x = $g - $h
+=	Addition assignment	
++	Post-increment	
-=	Subtraction assignment	
--	Post-decrement	
*	Multiply	$x = $y * 92
/	Divide	X = 1 + (y 2)

Table A.5: TEBNF inter-element operator.

Operator	Description	Example
AS	Defines a new element to be the same as an existing element	OUTPUT @UDP_Out AS @UDP_In END

access to object members, though it is important to note that TEBNF itself is not object-oriented. The dot operator permits other grammar elements and state transition table elements to have access to a given production rule subelement.

A.8   TEBNF State Transition Table Elements

The function of any given application can be described as a finite state machine composed of a finite set of states [32]. The machine is in one state at a time, and movement from one state to another is triggered by specific inputs [32]. Because applications can be asynchronousi.e. performing work on multiple threads, it is possible to have a finite state machine on each thread of execution. Concurrent behavior is becoming common in todays software [33].

Representation of a finite state machine in TEBNF is done using a state transition

table that can access any TEBNF input, output, grammar symbol, or static variable. This state transition table ties the various elements and subelements of a TEBNF grammar into a coherent description of a single thread of execution. This means that TEBNF makes it possible to represent multiple threads of execution using multiple state transition tables. Each state transition table consists of six columns delimited by the pipe operator. Columns flow from left to right in the following order:

1. State

2. Input and/or condition

3. Input Method

4. Next State

5. Output and/or action

6. Output Method

The first column is the current state. Each state identifies where to go in the finite state machine when the right conditions are met. The name of each state must be unique within the scope of the state transition table it belongs to. The second column specifies a condition that must be met to move to the next state. The condition can be but is not limited to a logical statement such as checking the value of a static variable or checking if the input received via a specific input method (specified in the third column) matches a given grammar production. The fourth column is the state to transition to upon satisfaction of the input or condition. The fifth column identifies the action to execute when transitioning to this next state. The action can range from setting a static variable to sending an output described by a grammar production via an output method (sixth column).

A.9   TEBNF Actions Elements

Actions elements contain a list of actions to be executed in the order they are listed. Actions elements can only be used within a state transition table element row. Since they

are not required in a TEBNF grammar, actions can be listed directly inside a state transition table. Actions elements are similar to macros in C++, and can accept zero or more comma-delimited parameters, as shown in the example taken from appendix listing B.1. Actions elements can also contain calls to C or C++ functions, making them one of the most powerful elements available in TEBNF.

Listing A.2: A TEBNF Actions element.

```
ACTIONS @right ( val )
    @tape . elem [ @tape . $i ] = val ;
    @tape . $i++;
END
```

A.10   TEBNF Input and Output Method Elements

Table A.6 shows all of the possible inputs and outputs available in TEBNF. Each input method describes a way for the generated application to receive input. No other settings information is required by an input method because the generated application will provide a way for the user of to give it the needed information through the user interface. Console elements allow prompt s to be defined within their scope, which tie a prompt string to a typed value to be input by a user. Listing A.3 shows a console input element with two prompt values. The first prompt value will display the prompt "Number: " and treat the input provided through the console as a signed 64-bit integer. The second prompt value will display the prompt "Op: " and treat input as a signed 8-bit integer.

Listing A.3: A TEBNF Console Input element.

```
INPUT  @Console_In = CONSOLE
    promptNum = INT_64 = "Number: ";
    promptOp = INT_8 = "Op: ";
END
```

Table A.6 shows all of the possible outputs available in TEBNF. Output methods describe a way for the generation application to send or display output.

Table A.6: TEBNF input/output specifiers.

Input/Output Types	Description
UDP_IP	Read/write input from/to from UDP.
FILE	Read/write input from/to a file.
CONSOLE	Read/write input from/to command line.

Networking inputs and outputs in TEBNF are based on UDP/IP. There are many other networking protocols, but many of them are transport and session layer protocols. Thus, describing other protocols can be done by using a state transition table to describe the protocol that will work over UDP/IP.

Custom input/output (I/O) is accomplished by using the state transition table to specify which grammar is used when receiving data as input or for sending data as output. In the case of a custom input, a GUI is generated that will ask for input to match the described grammar. In the case of output, data will be sent to the desired output following the format described in the provided grammar.

A.11   A TEBNF Example

A TEBNF example is shown in listing A.4 that describes a client application that transfers NITF 2.1 files over UDP/IP. This client application guarantees one-time delivery of each packet or none at all. The generated application receives packets matching the description shown in the examples grammar element. Upon reception of a packet, it will increment the payloads static variable, check if all of the data (entire file) has been received, and send an ACK message to tell the sender that a packet with a specific packet number has been received successfully. Once all of the packets have been received to reconstruct the file being sent (tracked by $payloads), the output is written to file and $payloads is cleared and ready to receive more file packets.

Listing A.4: TEBNF grammar describing a UDP NITF 2.1 file transfer client.

```
INPUT @UDP_In = UDP_IP END # UDP/IP input socket.
OUTPUT @UDP_Out AS @UDP_In END
```

```
OUTPUT @File_Out = FILE END
OUTPUT @Console_Out = CONSOLE END


GRAMMAR @Nitf
  # Describe the file to receive.
  syncWord = 'N', 'I', 'T', 'F', '0', '2', '.', '1', '0';
  skip1 = BYTE{333}; # FL offset is 342.
  fileLength = UNSIGNED INT_STR_96; # NITF file length field = 12 bytes.
  skip2 = BYTE{,};
  header = syncWord, skip1, fileLength;
  file = header, skip2;
  = fileLength; # Overall size of this file.
END
GRAMMAR @Send startSend = "send"; END
GRAMMAR @Status recvMsg = "All files received from server, exiting."; END
GRAMMAR @Done done = "done"; END


STATES @File_Transfer
```

# State	Input or Condition	Input Method	Next State	Output or Action	Output Method	
Init			Start	@Send.startSend	@UDP_Out	;
Start	@Nitf	@UDP_In	Init	@Nitf.file	@File_Out	;
	@Done	@UDP_In	Quit	@Status.recvMsg	@Console_Out	;
Quit						;

```
END
```

Appendix B

TEBNF Turing Completeness Proof

B.1 Turing Completeness

Turing machines provide the most powerful computational model known to exist [34]. The Turing completeness of a programming language is important because anything computable can be computed using that language [34].

A Turing machine that can perform any operation of any other ordinary Turing machine is known as a universal Turing machine [35]. Therefore, a programming language that can simulate a universal Turing machine is Turing complete.

B.2 Proof

Multiple examples of universal Turing machines have been presented [36–38]. A universal Turing machine that simulates a 2-tag system can be implemented with relatively few states and symbols. Tag systems simulate the game of tag, where the goal is to see if it will ever terminate by reaching the end of the sequence of symbols.

Rogozhin proved the universality of several classes of tag systems including a 4-state 6-symbol universal Turing machine called UTM(4,6) [36]. The tag system simulated by UTM(4,6) consists of 22 commands and is the lowest known number of commands for a universal Turing machine [36]. The machine is comprised of:

- A set of states: $q1, q2, q3, q4$

- Input symbols: 0 (blank), b, x, y, c (mark)

- Tape symbols

- An initial state: $q1$

- A transition function (executed in order):

    1. Print (write) a symbol.

    2. Move head left L, right R, none $-$.

    3. Go to the next state.

UTM(4,6) [36] is described as a list of 5-tuples (table B.1). These 5-tuples are formatted in order of evaluation using the Turing/Davis convention (q_i, S_j, S_k, left L, right R, none $-$, q_m) [39]:

1. Current state q_i.

2. Scanned symbol S_j.

3. Print symbol S_k.

4. Move tape head left L, right R, none $-$.

5. Next state q_m.

Since Rogozhin proved the universality of UTM(4,6) [36], an implementation of this machine in TEBNF is provided in listing B.1 to demonstrate the capabilities and Turing Completeness of TEBNF.

Table B.1: List of 5-tuples describing UTM(4,6).

$q11xLq1$	$q210Rq3$	$q311Rq3$	$q410Rq4$
$q1byRq1$	$q2byLq3$	$q3bxRq4$	$q4bcLq2$
$q1ybLq1$	$q2yxRq2$	$q3ybRq3$	$q4yxRq4$
$q1xbRq1$	$q2xyLq2$	$q3x-$	$q4x-$
$q10xLq1$	$q201Lq2$	$q30cRq1$	$q40cLq2$
$q1c0Rq4$	$q2cbRq2$	$q3c1Rq1$	$q4cbRq4$

B.3 Stepping Through the Machine

The tag system simulated by UTM(4,6) [36] has three stages. The 5-tuples each stage refers to are shown in table B.1. The corresponding TEBNF implementation is given in

listing B.1. The TEBNF implementation starts on the first line of the transition table at the begin state. The begin state reads the contents of a file into the array @tape.elements that functions as the tape.

Stage 1. The first stage is complete when the head of the machine moves right and meets the mark. The mark is deleted and the first stage ends at $q1c0Rq4$. The end of this stage corresponds to the seventh line in the TEBNF state table.

Stage 2. The machine executes a series of jumps to arrive at $q40cLq2$. If the head reaches pair xb, the machine jumps to $q2byLq3$ and halts at $q3x-$. Otherwise, the second stage ends upon reaching pair 1b.

Stage 3. The machine jumps to $q3ybRq3$, then $q311Rq3$. Upon moving to the right, the machine head reaches c (the mark), deletes it, and jumps to $q3c1Rq1$ to begin a new cycle.

Upon reaching one of the halt states, the TEBNF implementation of the machine writes the contents of the tape to a text file called Tape_Out.

Listing B.1: Rogozhin's UTM(4,6) implemented in TEBNF.

```
# TEBNF implementation of UTM(4,6) (a 4-state 6-symbol
# universal Turing machine) presented by Y. Rogozhin in
# "Small universal Turing machines", 1996.


INPUT @TpIn = FILE END;            # For reading tape from file.
OUTPUT @TpOut AS @Tape_In END; # For writing to tape file.


GRAMMAR @tape
    elem = BYTE{,} ; # Array with no min or max number of elements.
    $i = 0 ;         # Index for moving left or right on tape.
END


ACTIONS @right (val)
    @tape.elem[@tape.$i] = val;
    @tape.$i++;
```

```
END


ACTIONS @left (val)
   @tape.elem[@tape.$i] = val;
   @tape.$i— ;
END


# Read the tape and run the universal Turing machine. Moving right
# and left on the tape (represented by the @tape.elem array) is
# represented by incrementing and decrementing $index, respectively.
#
# Symbols: 0 (blank), 1, b, x, y, c
# States:  q1, q2, q3, q4
#
STATES @UTM_4_6
```

#State	Input or Condition	Input Method	Next State	Output or Action	Output Method
begin	@tape	@TpIn	q1		;
q1	@tape.elem[@tape.$i]== '1'		q1	@left('1')	;
	@tape.elem[@tape.$i]== 'b'		q1	@right('0')	;
	@tape.elem[@tape.$i]== 'y'		q1	@left('b')	;
	@tape.elem[@tape.$i]== 'x'		q1	@right('0')	;
	@tape.elem[@tape.$i]== '0'		q1	@left('x')	;
	@tape.elem[@tape.$i]== 'c'		q4	@left('0'	;
q2	@tape.elem[@tape.$i]== '1'		q2	@right('0')	;
	@tape.elem[@tape.$i]== 'b'		q3	@left('y')	;
	@tape.elem[@tape.$i]== 'y'		q2	@left('x')	;
	@tape.elem[@tape.$i]== 'x'		q2	@left('y')	;
	@tape.elem[@tape.$i]== '0'		q2	@left('1')	;
	@tape.elem[@tape.$i]== 'c'		q2	@right('b')	;

```
q3  |@tape.elem[@tape.$i]==  '1'|        |  q3     |@right('1')|         ;
    |@tape.elem[@tape.$i]==  'b'|        |  q4     |@right('x')|         ;
    |@tape.elem[@tape.$i]==  'y'|        |  q3     |@right('b')|         ;
    |@tape.elem[@tape.$i]==  'x'|        |         |@tape        |@TpOut;
    |@tape.elem[@tape.$i]==  '0'|        |  q1     |@right('c')|         ;
    |@tape.elem[@tape.$i]==  'c'|        |  q1     |@right('1')|         ;
q4  |@tape.elem[@tape.$i]==  '1'|        |  q4     |@right('0')|         ;
    |@tape.elem[@tape.$i]==  'b'|        |  q2     |@left('c')  |        ;
    |@tape.elem[@tape.$i]==  'y'|        |  q4     |@right('x')|         ;
    |@tape.elem[@tape.$i]==  'x'|        |         |@tape        |@TpOut;
    |@tape.elem[@tape.$i]==  '0'|        |  q2     |@left('c')  |        ;
    |@tape.elem[@tape.$i]==  'c'|        |  q4     |@right('b')|         ;

END
```