RE-ENGINEERING ATLAS SYSTEMS WITH WATLAS

by

Stuart W. Rivenbark

December, 2021

Director of Thesis: Dr. Mark Hills

Major Department: Computer Science

The ATLAS language is a legacy language that currently runs on OpenVMS systems. Here, we describe a Windows-based system for re-engineering existing ATLAS applications, transforming them into equivalent C# source code that can be compiled and executed on Windows. ATLAS is used for developing testing programs that interact with avionics systems connected to a test station. ATLAS utilizes automated test equipment to issue commands and interrogate results in response to direct stimulus and signals from the unit under test. Windows-based ATLAS, or WATLAS, is composed of a Rascal-based transpiler, a "pre" and "post" processor, the target environment framework, and a Windows-based CASS Station simulator to execute the transpiled target source code. The thesis also provides an overview of the legacy CASS station and the ATLAS language, motivation for development of WATLAS, and a review of some of the competing technologies in this information technology space. Finally, a working prototype with minimal functionality will demonstrate the viability of this approach.

RE-ENGINEERING ATLAS SYSTEMS WITH WATLAS

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

Stuart W. Rivenbark

December, 2021

RE-ENGINEERING ATLAS SYSTEMS WITH WATLAS

by

Stuart W. Rivenbark

APPROVED BY:

DIRECTOR OF THESIS:	Dr. Mark Hills
COMMITTEE MEMBER:	Dr. Nasseh Tabrizi
COMMITTEE MEMBER:	Dr. Rui Wu
CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:	Venkat Gudivada, PhD
DEAN OF THE GRADUATE SCHOOL:	Paul J. Gemperline, PhD

Table of Contents

LIST OF FIGURES viii

1 INTRODUCTION 1

Research Contribution 2

2 CASS/ATLAS OVERVIEW 4

2.1 The Operating System 5

2.2 History of OpenVMS 5

2.3 CASS Station Components 8

2.3.1 Control Subsystem 9

2.3.2 General Purpose Interface 9

2.3.3 Digital Test Unit 10

2.3.4 UUT Power Supplies 10

2.3.5 Digital Multimeter 11

2.3.6 Frequency Time/Interval Counter 11

2.3.7 Waveform Digitizer 12

2.3.8 Pulse Generator 12

2.3.9 Arbitrary Waveform Generator 13

2.3.10 Ancillary Equipment 13

3 OVERVIEW OF ATLAS 15

3.1 Legacy Similarities 15

3.2 Types of Statements and Capabilities 16

3.3 Comment Blocks 17

3.4 Branch Blocks 17

3.5 Entry Point Lines 18

3.6 Logical Line Numbers (LLNs) 18

3.7 Abbreviated LLN 20

3.8 PROCEDURE Statements 21

3.9 The LU File 21

4 STRATEGY AND TECHNOLOGIES USED TO DEVELOP WATLAS . . 24

4.1 Step 1: Developing a RASCAL Grammar 25

4.1.1 Lexical Rules 25

4.1.2 Parser Rules 27

4.2 Step 2: Using Grammar and ATLAS to Generate C# Equivalence . . 29

4.2.1 Defining Expected Output 29

4.2.2 Abstract Syntax Tree 30

4.2.3 Generating Output 33

4.3 Step 3: Developing the WATLAS Integrated Framework 35

4.3.1 Framework Components 35

4.3.2 Defining a Project Template 36

4.3.3 The CASS Station Human Interface 37

4.3.4 Pre and Post Processor 38

4.4 Next Steps - Enhancing WATLAS 40

4.4.1 Rascal Project Enhancements 40

4.4.2 C# Project Enhancements 43

4.4.3 Tool Enhancements 43

5 COMPETING TECHNOLOGIES AND SIMILAR EFFORTS 45

5.1 PAWS 46

5.1.1 Transfer-ability of OpenVMS-Compliant ATLAS Source Code 46

5.1.2 Target Executable Implementation 46

5.1.3 Asset Interfaces 47

5.1.4 UUT Latching 47

5.1.5 Industry Standard IDE Integration 47

5.2 ANTLR 48

5.3 Haskell 51

5.4 RTCASS 52

5.5 ECASS 55

5.6 LLVM 56

5.7 Translating JOVIAL 57

6 EVALUATION 63

6.1 Semantics Analysis 63

6.2 Correctness 64

6.3 Viability Concerns 65

6.3.1 DataSegment Design 65

6.3.2 DataSegment Concerns 66

7 CONCLUSIONS AND FUTURE WORK 70

7.1 Improving the DataSegment Design 70

7.1.1 Overloading Return Types 71

7.1.2 Constraint Solver Solution 73

7.2 Benchmark Analysis of WATLAS 73

7.3 Conclusions 76

A APPENDIX A - ONLINE LINKS AND PARSING RULES BY ATLAS STATEMENT TYPE 78

A.1 Links to the Online Repositories 78

A.2 Parsing Rules Examples by ATLAS Statement Type 79

BIBLIOGRAPHY 81

LIST OF FIGURES

2.1 CASS ATE Station Example 5

3.1 ATLAS Syntax Example 16

3.2 LLN with Fully Qualified Numbers 21

3.3 LLN with Abbreviated Numbers 21

4.1 ATLAS Variable Name Lexical Rules. 27

4.2 ATLAS Parser rules for Procedural Statements. 28

4.3 ATLAS to C# Translation. 30

4.4 CALCULATE Statement Tokens. 31

4.5 AST Data Structure for CALCULATE in Rascal 32

4.6 AST Production for CALCULATE in Rascal 33

4.7 Rascal Code to Generate Target Statement from AST 34

4.8 IMOM Interface 37

4.9 Example PostProcessor Structured Intermediate Token Substitution . 40

4.10 Example Flow of Translation 42

5.1 ANTLR Grammar Example. 49

5.2 SPEAK Example with no errors. 50

5.3 SPEAK Example with errors detected. 50

5.4 Example Haskell Parser using Parsec. 52

5.5 RTCASS Station 54

5.6 eCASS Station 55

5.7 LLVM Bird's Eye View. 57

5.8 Example JOVIAL to Ada and C Translation 59

6.1 Using CALCULATE with variables in expression. 67

6.2 Another Naming Convention for ATLAS to C# Variable Naming. . . 68

6.3 Simplified Dictionary-Based Implementation for ATLAS to C# Variable Naming. 69

7.1 Fully Abbreviated Variable Name Translation Approach 71

7.2 Run-time Diagnostics Example. 75

Chapter 1

Introduction

The goal of this thesis is to develop a Windows-based (x86) transpiler for ATLAS source code that will correctly interpret the ATLAS syntax and generate a syntactically correct C# target source code representation. After compilation on Windows, it will accomplish the equivalent functionality that the OpenVMS-based ATLAS compiler generates. Considerable explanation is required in order to understand the motivation for this thesis. For example, the reader most likely has never heard of ATLAS and would have no idea what or who would benefit if this goal is achieved. Further, one must consider the many paths to achieve the goal and analyze the competing technologies and approaches to accomplish it. Finally, in order to appreciate the fruit of this labor, an explanation of how and what was developed must be revealed.

Therefore, this thesis focuses on answering the following questions:

- **RQ1:**What existing work has been done on providing modern support for legacy languages, including ATLAS?
- **RQ2:**What are the challenges to providing support for ATLAS on Windows? How well do existing solutions handle these challenges?
- **RQ3:** How well does WATLAS address these challenges?

The rest of this thesis is organized as follows. In Chapter 2, an overview of CASS/ATLAS is detailed along with its purpose and how it is used in the real world today. In Chapter 3, we provide a deeper dive into the existing ATLAS Test Language and provide analysis of language features that WATLAS must support. In Chapter 4, a detailed analysis of the tools and technologies used in this thesis is provided, as well as a step-by-step review of the steps taken to develop WATLAS. In Chapter 5, a consideration of other research projects related to analysis of legacy compilers is given along with a consideration of how these technologies compete with WATLAS. Finally, in Chapter 6, we share an analysis of the results from our work developing WATLAS thus far, and concerns about the viability of this effort. Finally, in Chapter 7 we propose further work on WATLAS and ways to improve it with respect to the viability concerns raised in Chapter 6. We also provide some suggestions for providing benchmark analysis for WATLAS.

Research Contribution This thesis conducted a review of existing work on modernizing tool support for ATLAS, as well as related work on similar support for other legacy languages. Further, an investigation of existing challenges currently occurring in the "ATLAS on Windows" domain was conducted and representations as to their effect in this technology space is provided. Finally, this thesis presents the progress that WATLAS has made to circumvent these challenges and how it compares to similar technologies.

The topics discussed in this work present insights into how developers construct compilers and transpilers from existing languages. This analysis is designed to enable future work on building compilers and transpilers, and understanding of the transformation tools used to migrate from legacy operating systems to newer technologies. To enable future work in this area, all results, source code, a video presentation, a PowerPoint presentation, and LaTeX materials shown in this work can be generated using the functions found in the repository, which is publicly available and can be downloaded using the links provided in the Appendix.

Chapter 2

CASS/ATLAS Overview

ATLAS, which stands for Abbreviated Test Language for All Systems, [1] is an IEEE Standard MIL-SPEC language. United States defense standards, often called a military standard, "MIL-STD" or "MIL-SPEC", are used to help achieve standardization objectives by the U.S. Department of Defense.

ATLAS is used to develop avionics software for testing on Automated Test Equipment (ATE) and specifically for Consolidated Automated Support Systems (CASS) for the U.S. Department of Navy. The language has a large collection of dialects that suit themselves to particular applications. While ATLAS is used for a variety of aircraft, aerospace and other commercial ATE applications, we will limit the scope of this project to features of ATLAS that provide compatibility to CASS applications that are used primarily by the U.S. Navy to support naval aircraft systems. CASS stations have embedded OpenVMS computers that ATLAS programs execute on and they provide interfaces to the hardware assets and Units Under Test (UUT). A visual of a typical CASS station is provided in Figure 2.1.

[1] https://web.archive.org/web/20051110020029/http://grouper.ieee.org/groups/scc20/atlas/

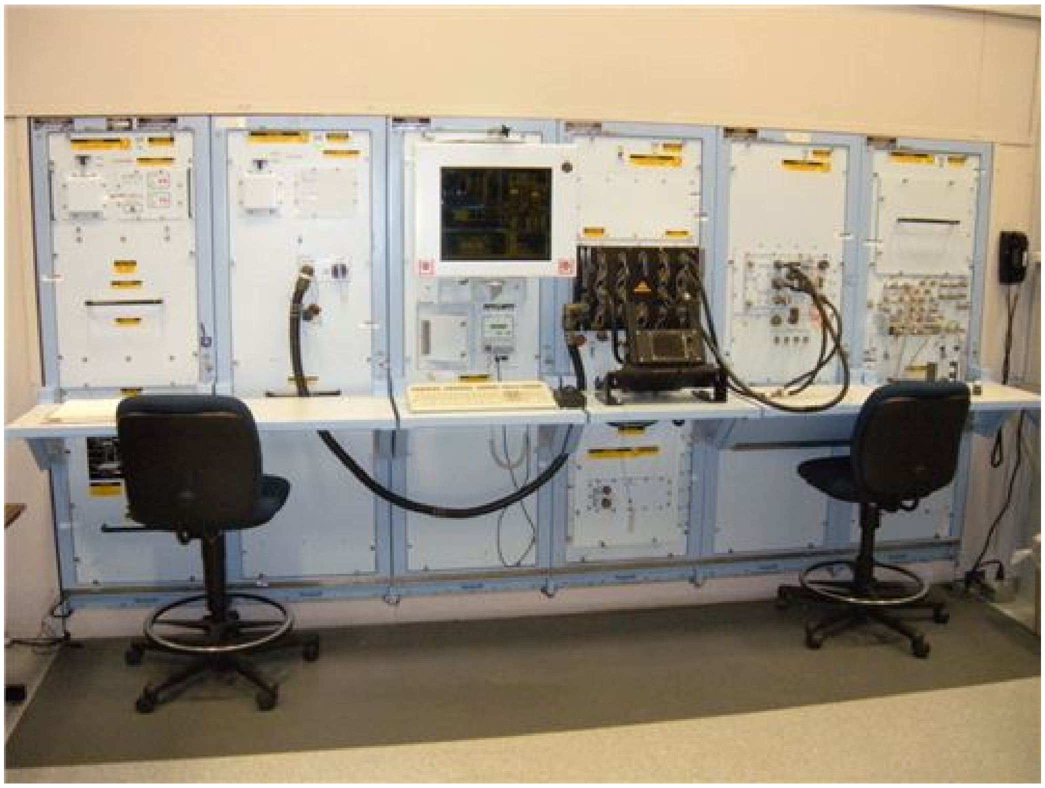


Figure 2.1: CASS ATE Station Example

2.1 The Operating System

CASS/ATLAS compilers have traditionally been deployed on computers running the OpenVMS operating system. OpenVMS was first released in 1977 and was prevalent in industries that supported engineering applications. In particular, electrical and mechanical engineering applications of the 1980s and 1990s were well-suited to utilization of OpenVMS due to its high-reliability and due to the fact that the operating system was built by engineers specifically for engineering applications.

2.2 History of OpenVMS

Digital Equipment Corporation (DEC) originally developed OpenVMS as VAX (or VMS) in the 1970's. Subsequently, DEC was acquired by Compaq in 1998 and Compaq merged with Hewlett Packard (HP) in 2002. The licensing rights were passed to HP which was building a portfolio of many legacy operating systems such as OpenVMS and Guardian (a.k.a.; NonStop) among others. Development on OpenVMS virtually ceased around 2005 and HP set it on a path to extinction around 2012. Despite this, there still exist many industries and hobbyists[2] that remain devoted to OpenVMS. There are several virtualized OpenVMS offerings on the market (vtAlpha and vtVAX)[3] and Charon[4] solutions. There was even a project named FreeVMS that undertook an effort to move OpenVMS to open source, but this project was abandoned shortly after it started. Since 2005, support for OpenVMS has primarily been provided by third-party vendors on a contractual basis, and these contracts can be quite costly.

It is interesting to note why the OpenVMS service life took the trajectory it has experienced. While it is remarkable to see how design decisions and the flux of the computer industry affect various technologies, we can also see why it is necessary to continually evolve technologies. This also contributes to understanding the motivation for developing WATLAS.

1) The VAX instruction set was a member of the Complex Instruction Set Computer (CISC) families which were prevalent in the late 1970s. While aiding compiler development, this made hardware development more time-consuming and costly which is why most hardware developers marketed not only their hardware but also the software that ran on it. With the advent of Reduced Instruction Set Computers (RISC), CISC technologies began to suffer due to their inflexibility.

2) VMS was written in VAX assembly language and there was neither broad knowledge of its intricacies, nor a desire to acquire this knowledge.

[2] http://www.eight-cubed.com/
[3] https://www.salemautomation.com/vtalpha-bare-metal
[4] http:/https://www.stromasys.com/solutions/charon-vax/

3) The architectural design of VMS had a major flaw involving early decisions made by the operating systems developers concerning critical locking mechanisms. Per Stanford university: "VMS used a highly non-standard feature of interrupt priority levels to get the effect of locking. There were only 32 such levels, leading to only 32 major locks. The physical lock table was only 32 by 32 while the effective lock table was more like thousands by thousands. Every lock was highly overloaded in meaning. This did not scale and made VMS more complex. After a decade of development, VMS was moved to the alpha chip set, but only after the alpha chip set was modified to support interrupt priority levels. Had the original team separated out the locking mechanism to a more general architecture, this dependency could have been avoided". [2]

4) The locking scheme described above in the 3rd point was neither well-documented, nor well understood by anyone outside of the original development team. This made further development very costly and nearly impossible.

In 2014, HP reversed course on OpenVMS and moved the licensing rights to a new entity named VMS Software Inc. (VSI). Since then, VSI has been actively engaged in bringing OpenVMS back from its near death experience. This is a new development for this author since the start of this thesis project back in 2018. It was not widely known that this effort was underway, but VSI released an x86-64 Limited Early Adopters Kit in 2021[5]. This is an encouraging development for OpenVMS, but it may be a little too little, and a little too late.

Electrical engineers are highly skilled workers and are often very motivated to learn new technologies. This has presented a challenge for legacy CASS and ATLAS since the newer entrants into the workforce do not want to expend the effort to learn something considered to be very old with a fast approaching end-of-service-life

[5]https://vmssoftware.com/about/roadmap/

(EOSL). This makes it hard to attract the best candidates to work on these types of systems. At Marine Corps Air Station Cherry Point, experienced CASS/ATLAS developers are retiring, yet the aircraft that depend on this technology are still on the flight line daily and have been deployed to various parts of the world in recent conflicts. It is neither feasible to simply decommission these costly weapons systems, nor can replacement technologies be fielded as quickly as needed. Tools like WATLAS will hopefully provide an interim solution that attracts workforce participants since it takes from the old to build an application with a development experience that feels more current and relevant.

2.3 CASS Station Components

The CASS station depicted in Figure 2.1 shows most of the components (a.k.a.; assets) of the CASS station are embedded in the station itself. All of these components are present when developing ATLAS programs, debugging them, and testing them. However, the nature of the avionics equipment being tested determines which asset classes will be utilized during test execution. Test Program Set (TPS) developers are electrical engineers with experience regarding these components, their proper usage, and how to set and read the input and output from them.

An interview with a senior CASS/ATLAS TPS developer who is a subject matter expert was conducted to provide some descriptive information regarding these assets. For each asset class, a brief description of what purpose they serve is provided. Where an example that can be easily understood could be explained, it has been provided. Finally, for each asset class, an estimate of the frequency of their usage in a typical TPS is given. Note that percentages do not apply to the entire universe of TPSs, but are limited to the knowledge of this particular subject matter expert (SME). This TPS developer works mainly with V22 Osprey aircraft, but has had significant exposure to several other vertical and/or short take-off and landing (V/STOL) aircraft.

A typical early release CASS station configuration consists of the following asset classes:

2.3.1 Control Subsystem

The Control Subsystem is comprised of one 133 MHz OpenVMS Computer with 64 Mbytes memory, a 2.1 GB hard drive, a 1.3 GB optical disk, keyboard, trackball, bar code reader, 16 inch display, and self-test diagnostics.

This asset class is the subject of this thesis and has been described above and elsewhere in this document. Therefore, neither a purpose will be provided here nor will an example of how it is used will be provided. As far as usage in typical TPSs, 100% of TPSs rely on this asset class.

2.3.2 General Purpose Interface

The General Purpose Interface (GPI) includes; 1,486 usable pins, a latching mechanism for holding the UUT ID, instrumentation I/O brought directly to the pins, and a user configuration switch.

The GPI is the main rectangular panel on the front of the CASS station to which all UUT connect via some sort of Ancillary Equipment, or vendor-provided equipment. Many of these are custom interfaces that the GPI mates to. All of the signals in the station, be it measurement, response, switching, etc. are provided or interrogated through this panel. The only thing these interfaces have in common is that in one way or another, they will satisfy all of their connectivity needs via the GPI. The cabling is generally the same as the cabling requirements inside the aircraft.

2.3.3 Digital Test Unit

The Digital Test Unit (DTU) includes programmable logic levels negative 5V to positive 15V, 384 bi-directional stimulus/response channels, expandable to 512, 50 Mbits/sec stimulus/response data rate, 25 MHz clock rate, 64K memory depth per channel with 20 ns pulse detection, and a dynamic fault dictionary with data acquisition RAM and a remote probe ranging from 0 to 50 MHz.

The DTU falls outside the scope of this thesis since we have limited the scope of this paper to ATLAS. The DTU is used for very specialized testing of Printed Circuit Boards (PCBs) and is not accessed via ATLAS-based syntax. Rather, a language known as L200 performs DTU testing and executable L200 code is bound into ATLAS programs to issue calls when necessary. A very small percentage of TPS testing (less than 10%) involve the DTU on the CASS Station.

2.3.4 UUT Power Supplies

The following UUT Power Supplies must be present: (1) Direct Current (D/C) Programmable (800 W), (8) 0 to 32V at 25A, (1) 0 to 100V at 8 A, (2) 50V to 400V at 2A, (4) Alternating Current (A/C) Programmable (1 to 135 Vrms at 4.5 A max, and 55 to 1200 Hz, in 1, 2, or 3 phase)

An A/C Power Supply is generally used to provide 120 volt A/C power in the 135Hz to 400Hz range to a UUT. The usage of 400Hz is to mimic naval ship power (a.k.a.; afloat environment). An example usage might be to provide lighting to a display unit. The percentage of TPS that utilize the A/C asset is estimated at 50%.

The D/C Power Supply is generally used to provide primary or secondary power to a UUT. It can also be used to provide discrete stimulus for something like 20 volt or lower voltage logic inside the aircraft. It can be used for any device (depending on the electrical current requirement) to drive a particular signal to the device. An example usage might be to drive a mission computer being powered up by the primary 28 volt bus. The percentage of TPS that utilize the D/C asset is estimated at 90 to 95%.

2.3.5 Digital Multimeter

The Digital Multimeter (DMM) provides 6-1/2 digit resolution, voltage of 200 V at GPI, a 0 to 1000 VDC probe, and a 0 to 700 Vrms probe with current of 0 to 2A A/C-D/C and resistance of 0 to 30 mega-ohms.

DMMs are used to measure A/C or D/C voltage and resistance as well. Primary usage is for ”safe to turn on tests” to detect short circuit situations before applying power to a UUT. They are also used for test point measurements periodically throughout TPS execution. The percentage of TPS that utilize the DMM asset is estimated at close to 100%.

2.3.6 Frequency Time/Interval Counter

The Frequency Time/Interval Counter (FTIC) provides 2 Channels with D/C coupling (0.001 Hz to 200 MHz) and A/C coupling (100 Hz to 200 MHz) It supports time intervals of 4 ns to 15,000 sec. Channel A ranges from negative 5 ns to 1000 sec, Channel B ranges from negative 10 ns to 1000 sec with a maximum count event rate of 20 MHz, with input voltage specified at +10 Vp D/C and Sensitivity of 0.1 Vpp.

FTICs are used to measure the frequency of a periodic signal. For example, suppose a square wave is encountered during testing after a certain amount of time, and that event might be a 1 KHz square wave. The FTIC could measure that signal to a very tight tolerance. It could also measure the exact time between two different events that occur during the test execution. Many times, this is used for checking or verifying clock signals inside of units. The percentage of TPS that utilize the FTIC asset is estimated at no more than 40%.

2.3.7 Waveform Digitizer

The Waveform Digitizer (WVFRMDIG) supports 0 to 500 MHz, 4 Channels (2 at GPI, 2 external), Vertical Voltage at GPI: 8 mV to 40 V full scale, Maximum input voltage: 5 Vrms (50 ohm input), Maximum sample rate: 20 mega samples/sec., and Memory depth: 1024 points. It supports waveform types as follows: D/C, Sine, Square, Step, Triangle and Pulsed D/C, Low Power Wattage Load Range: 1.5 to 99,999 ohms, Increments: 0.1 ohm, Power dissipation: 5 watts, High Power Wattage Load, with programmable ranges of 0 to 20 Amps, 1 ohm to 5 Kohm, and Power dissipation of 500 watts with Unipolar D/C only.

WVFRMDIGs are used to take samplings of analog signals and turn them into a discrete representation of that signal. It can be thought of as a programmable oscilloscope for any type of A/C wave form. For example; for a signal that should be going to an A/C motor, this asset can ensure that the signal is at a certain frequency and ripple. It is frequently used to detect noise and ripple inside a signal. The percentage of TPS that utilize the WVFRMDIG asset is estimated at around 30 to 40%.

2.3.8 Pulse Generator

The Pulse Generator (PGEN) supports two channels with these operating modes: Continuous, Gated, Burst, and Trigger, Output Voltage: +100 mV to +5 V into 50 ohms, Pulse period: 4 ns to 99.9 ms, Pulse width: 2.0 ns to 89.9 ms and Pulse delay: 0 ns to 89.9 ms.

PGENs are primarily used by TPS developers for creating timed events possibly of a certain wave form; a square wave every ”n” microseconds, or a continuous pulse train. They allow for the creation of a synthetic testing environment to see how the UUT reacts. They also allow for creation of circuitry signals that act as a clock and allow the TPS developer to control the clock. The percentage of TPS that utilize the PGEN asset is very low at probably no more than 15%.

2.3.9 Arbitrary Waveform Generator

The Arbitrary Waveform Generator (ARBWVFRMGEN) supports two channels with Amplitude: +5V, Maximum amplitude: 10 Vpp, 0 to 25 MHz sine, pulse, ramp, 48 Hz to 200 MHz arbitrary point generation, 48 Hz to 100 MHz digital patterns (11 bit). Rise/fall times are, Channel A - 10 ns to 100 sec, Channel B - 30 ns to 100 sec, Minimum pulse width: 10 ns, Sweep Time: 1.4 us to 40 sec with Communication Buses as follows: MIL-STD 1553 A/B, MIL-STD-1773, IEEE-488, RS-232, RS-422, IEEE-802.3, ARINC-429, and MIL-STD-1397.

ARBWVFRMGENs are used by TPS developers to create their own kind of wave forms; square waves, triangular waves, saw tooth waves, and different types of A/C signals, of different frequencies and different amplitudes. The ARBWVFRMGEN is very good for mimicking stimulus from a communication system. It can mimic the tone of a voice down to a single Hz tone that could be routed through an audio switcher. The percentage of TPS that utilize the ARBWVFRMGEN asset is estimated at around 25 to 30%.

2.3.10 Ancillary Equipment

Finally, the ancillary equipment includes RS-485/MH switch Assemblies with Power switch (D/C to 1000 Hz), (5) 1 X 4 ganged high current (18.75 A), (2) 1 X 2 ganged high current (18.75 A), (6) 1X 2 ganged low current (9 A), (1) 1X 2 high current

(18.75 A), low frequency switch (D/C to 1 MHz), (21) 1 X 4 low frequency, (35) 1 X 2 low frequency, coaxial switch (D/C to 1 GHz), (11) 1 X 4 coax and (3) 1X2 coax.

Ancillary Equipment (AE) is coupled to the General Purpose interface and is used in one form or another by nearly every TPS. This equipment has changed through the years so much that there is no one set of equipment generic enough for all UUTs. As new UUTs are developed, they have their own set of AE unique to their connectivity requirements (i.e.; cabling, interface cards, etc.) All UUTs require some form of AE to allow connectivity to the General Purpose Interface.

Chapter 3

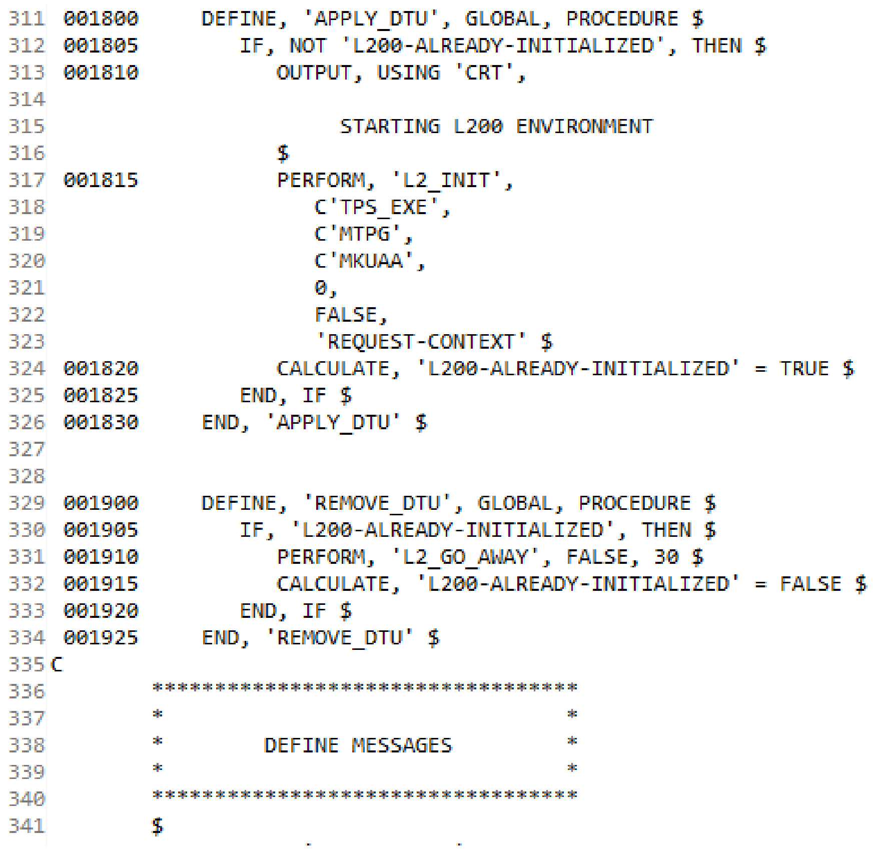
Overview of ATLAS

The ATLAS for Windows transpiler (WATLAS) developed for this thesis is designed to support legacy ATLAS syntax available in the OpenVMS-compliant ATLAS compiler. The first ATLAS specification was published in 1968. ATLAS resembles other procedural legacy languages of the 1970s, in particular COBOL. This is a brief overview of ATLAS syntax.

3.1 Legacy Similarities

Referring to the sample code in Figure 3.1, one can note several similarities to COBOL such as:

1. There is a 80 character line limit.
2. There is a 6-digit leading line number in columns 2 through 7.
3. Column 1 may be used for commenting code and other special purpose features.
4. The source code must be provided in upper-case letters.
5. Similar reserved words (i.e.; IF-THEN-ELSE, OPEN, CLOSE, PERFORM, etc.) are used.



```
 001800     DEFINE, 'APPLY_DTU', GLOBAL, PROCEDURE $
 001805        IF, NOT 'L200-ALREADY-INITIALIZED', THEN $
 001810           OUTPUT, USING 'CRT',

                       STARTING L200 ENVIRONMENT
                  $
 001815           PERFORM, 'L2_INIT',
                     C'TPS_EXE',
                     C'MTPG',
                     C'MKUAA',
                     0,
                     FALSE,
                     'REQUEST-CONTEXT' $
 001820           CALCULATE, 'L200-ALREADY-INITIALIZED' = TRUE $
 001825        END, IF $
 001830     END, 'APPLY_DTU' $


 001900     DEFINE, 'REMOVE_DTU', GLOBAL, PROCEDURE $
 001905        IF, 'L200-ALREADY-INITIALIZED', THEN $
 001910           PERFORM, 'L2_GO_AWAY', FALSE, 30 $
 001915           CALCULATE, 'L200-ALREADY-INITIALIZED' = FALSE $
 001920        END, IF $
 001925     END, 'REMOVE_DTU' $
C
        **********************************
        *                                *
        *        DEFINE MESSAGES         *
        *                                *
        **********************************
        $
```

Figure 3.1: ATLAS Syntax Example

3.2 Types of Statements and Capabilities

The source code line for an ATLAS program is no more than 80 characters in length and consists of four sections:

1. Offset 0, Length 1: This single character that starts the line can be either:

   (a) The character C (upper case), indicating the start of a comment block.

   (b) The character B, indicating a branch block.

   (c) The character E, indicating an entry point (similar to "main" in the "C" language).

   (d) The "space"character, which has no meaning.

2. Offset 1, Length 6: These six numeric characters (if provided) indicate a logical line number.

3. Offset 7, Length 1: Dead space. There is usually nothing in this position. None of the source code reviewed for this project ever made use of it.

4. Offset 8, Length 72: Actual source code or comments.

3.3 Comment Blocks

ATLAS comments are free form text that can span multiple lines. The comment area will commence with a "C" in column 1 of the source code line and continue until a dollar sign ($) character occurs which is considered the comment termination character. Also, all ATLAS executable and declarative statements use the dollar sign ($) character to indicate termination of the statement.

3.4 Branch Blocks

ATLAS programs can be developed in two manners: modular or segmented. In the modular implementation, the functionality that may be exposed is defined as "GLOBAL" and any potential callers of the functionality in that module must reference the module in an INCLUDE statement and define the function reference as an EXTERNAL implementation. Segmented implementations are similar to COPYLIBs of the 1970's and 1980's. The source code of the caller performs a GOTO to the line number just after the branch block in the called source code. The start of a branch block must have a "B" in column 1 of the source code line and it continues until a dollar sign ($) character occurs which is considered the block termination character. Branch blocks rarely span more than two lines of actual source since they are only pointer locations into the source code. There are typically some sort of free form comments after the ”B” indicator, but before the branch terminating ($).

3.5 Entry Point Lines

There is only 1 entry point line, and that line starts with ”E” and indicates where the actual program starts. Unlike Comment and Block lines, a six-character numeric line number will follow the ”E” in positions 2 through 7 of the entry point line. Also, procedural source code can follow the line number after character position 8. In the opinion of this author the best structured ATLAS programs have the ”E” line near the end of the source code file just before the END statement. Everything that follows the entry point line is just calls to functions that have been previously defined above the entry line.

3.6 Logical Line Numbers (LLNs)

LLNs have assorted rules that make these not quite as straightforward as they may seem. These rules vary somewhat depending on the type of ATLAS statement. For sake of clarity, ATLAS has only 9 statement types, three of which have been discussed above (comments, branches and entry points), and a plurality of PROCEDURAL statements which are typically embedded within a DEFINE block. The rules of LLNs are described for these six statement types:

1. BEGIN: There is only one BEGIN statement in an ATLAS program. It is the first line in the program and it starts with a full six-character LLN. It terminates with the $ character.

2. INCLUDE: There may be several INCLUDE statements in an ATLAS program, but each starts with a full six-character LLN. It terminates with the $ character.

They rarely span more than one line.

3. REQUIRE: There may be several REQUIRE statements in an ATLAS program, but each starts with a full six-character LLN. It terminates with the $ character. They usually span more than one line, but there is only one LLN for the REQUIRE and it is on the first line of the statement.

4. DECLARE: There may be several DECLARE statements in an ATLAS program, but each starts with an LLN. However, the LLN may be abbreviated (see section 3.7). It terminates with the $ character. Also, several ATLAS variables of the same data type can be concatenated into a single DECLARE statement so they can and do span multiple lines. But, there is only one LLN for a single DECLARE.

5. DEFINE: There may be several DEFINE statements in an ATLAS program, but each starts with an LLN. However, the LLN may be abbreviated (see section 3.7). It terminates with the $ character. Also, and this is unique among all the statement types, each line of a DEFINE must start with an LLN. By contrast, a REQUIRE may span several lines, but only starts with one LLN on the first line of the REQUIRE. Since a DEFINE spans several lines, then each line starts with a LLN. The reason DEFINES must span at least two lines is because a DEFINE starts on one line indicating the name of the procedure to be defined. The definition terminates with an END statement indicating the end of the DEFINE block. Every complete statement in a DEFINE will end with the $ character. DEFINES often have embedded local DECLAREs. DEFINES are of three types:

   (a) GLOBAL: Can be called by any other module "bound" to this program.

   (b) EXTERNAL: References a GLOBAL that is defined in another module.

   (c) Local: These will contain PROCEDURE statements and are only available to be called by this program (i.e.; must have scope local to this source file).

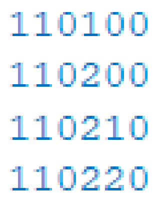
6. END: Other then the DEFINE-END statement, there must be one END statement in an ATLAS program. It is the last line in the program and it starts with a full six-character LLN. It terminates with the $ character.

The other statement types that make use of LLNs are PROCEDURE statements. These statements usually occur inside of a DEFINE section, although they can be used inline in a free form manner for particularly small programs where no subroutine-like grouping is implemented. DEFINES can be thought of as blocks of logic similar to functions in block-structured languages or methods in object-oriented languages. More information on sub-types of PROCEDURE statements is found in subsection 3.8.

3.7 Abbreviated LLN

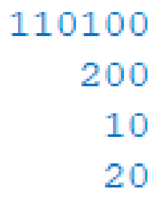
Abbreviated LLNs are unique to ATLAS and this author knows of no other language where this construct has been implemented. The only other language with a vaguely similar construct is JOSS [1] which made use of meaning in line numbers. But, in ATLAS abbreviated LLNs are merely for convenience and for visual affect. They essentially let the programmer only provide the least significant character digits of an LLN if the missing digits are the same as the line above it. For example, LLNs could be written like the following in Figure 3.2. However, the abbreviated line numbers of Figure 3.3 express the same LLN scheme.

[1] https://en.wikipedia.org/wiki/JOSS



```
110100
110200
110210
110220
```

Figure 3.2: LLN with Fully Qualified Numbers



```
110100
   200
    10
    20
```

Figure 3.3: LLN with Abbreviated Numbers

3.8 PROCEDURE Statements

These statements are the actual executable statements that are performed in a DEFINE. This list is not exhaustive, but they specify these types of procedures: APPLY, IF, CALCULATE, FILL, OUTPUT, PERFORM, READ, REMOVE, VERIFY, WAIT, and WHILE.

While this project has limited the scope of PROCEDURE statements it will support due to time constraints, WATLAS is being developed as an extensible product. Support for additional PROCEDURE types can be added as needed.

3.9 The LU File

ATLAS programs support portability between CASS stations by means of an LU file. Test Program Set (TPS) developers coined the term ”LookUp” file and it has LU as its file name extension. Given an ATLAS program named EXAMPLE.AS, there may be an EXAMPLE.LU file in the same directory as the EXAMPLE.AS file. An LU file is required if there exists one or more REQUIRE statements in the AS file, since LU files provide configuration mapping information about assets referred to in REQUIRE statements in the AS file. The LU file must have the exact same file name with the exception of the extension (i.e.; EXAMPLE.AS specifies its mappings in EXAMPLE.LU).

The purpose of the LU file is to map logical hardware requirements to their physical hardware device name on a given CASS station. For example, suppose a program makes use of a printer, and the device name of the printer hardware on a CASS station is "PRT". The TPS developer may reference the logical device name as "PRINTER" without regard to the actual device name in their REQUIRE or OUTPUT statements and the source code file will compile without errors provided there is a related entry in the LU file. The compiler only checks the LU file to determine if a device mapping exists. But it is the responsibility of the CASS station at run-time in response to execution of the REQUIRE statement to create the connection. Thus, to complete the mapping at compile time, the LU file must contain a one line entry like the following:

PRINTER PRT

In this way an ATLAS program may be "ported" to a CASS station with a different hardware configuration by simply updating the LU file entries. For standard hardware assets that the majority CASS stations do support, this rarely occurs since military software applications and hardware configurations are thoroughly reviewed for compliance to standard naming conventions. But, the LU file provides additional capabilities that are not well suited to stringent standards requirements. It also provides for configuration of assets that are not widely used in TPS's that are peculiar to certain automated test equipment (ATE).

The LU file also may contain other optional configuration characteristics that are pertinent to avionics software testing on CASS stations. Among these one might find PIN configuration statements if the program utilizes wire connections to printed circuit boards (PCB) or other hard-wired components of a specific hardware asset. For example, one pin configuration statement for a digital multi-meter in the LU file might look like the following (and will span three lines):

```
DMMHIR1HI    N1J9-14A
             N1J9-32A
             N1J9-32C
```

This brief explanation of an LU file has been included in this section because it influences the translation of our ATLAS source program into its C# equivalent. It is also the reason that the WATLAS Post Processor is necessary since it provides the class name for the asset as we will see in the next chapter. Finally, this construction necessitated the post processor steps since this feature of ATLAS could not easily be accommodated by Rascal.

Chapter 4

Strategy and Technologies Used to Develop WATLAS

The author of this thesis chose to utilize Rascal MPL as much as possible in the development of the ATLAS transpiler for Windows (WATLAS). That decision could not be characterized as an "informed decision". Rather, the suggestion was made by Dr. Mark Hills to look into using Rascal as he felt that the tool was up to the task. He is also a contributor to the Rascal open source project and this provided a level of confidence in acceptance of his recommendation since he is a subject matter expert. It also made sense to pursue it in order to learn a new technology that will no doubt prove useful in future projects. After a few weeks of working with Rascal to assemble a preliminary grammar file, it appeared that Rascal did have most of the necessary capabilities. For those that were lacking, the tool appeared to be flexible enough to build a customized implementation; for example, the ability of Rascal to allow the compiler developer to handle Atlas's logical line numbers and comment structure as specified by a positional comment character. Both of these anomalies are relics of legacy languages that present parsing challenges.

This chapter will explain how Rascal was used to develop WATLAS, the development of the C# output source, the WATLAS transpiler and its integrated framework, the Pre and Post processing functionality, and the CASS Station emulator that is built into the framework to allow the TPS developer to test the generated output. It should be noted here that the WATLAS transpiler is not exhaustive of all the capabilities of the ATLAS language. WATLAS is an extensible framework that electrical engineers can build upon to continuously add new language features. Finally, this section will provide ”next steps” for those interested in building upon this framework and enhancing the tool for further use.

4.1 Step 1: Developing a RASCAL Grammar

Rascal is similar to many lexers and parsing tools with respect to the fact that the basic building block of the tool is a grammar file. Rascal makes the task of developing the grammar easier by providing a framework that integrates with Eclipse. Once the Rascal plugin is downloaded and installed [1] into Eclipse, the developer can create an RSC file in Eclipse's text editor and use the integrated Rascal Console to test the grammar in an iterative manner.

4.1.1 Lexical Rules

Both the lexical rules and parsing structures can be provided in a single RSC file. In fact, the combined lexical rules and parsing syntax for WATLAS are in a single file in this Eclipse project (atlassyntax.rsc). But it is helpful to think of these components as separate entities. Lexical rules deal with identifying what the tokens are in a grammar. The technical term for these components is either a lexeme, a token, or a terminal. A token is a lexeme along with the type of the lexeme. So, the lexeme is the recognized character sequence. The lexeme plus the type are the token which become a ”terminal” in the parser.

For example; ATLAS variables have an unusual construction. They have the following composition; a single quote (‘), followed by one or more upper-case alphanu-

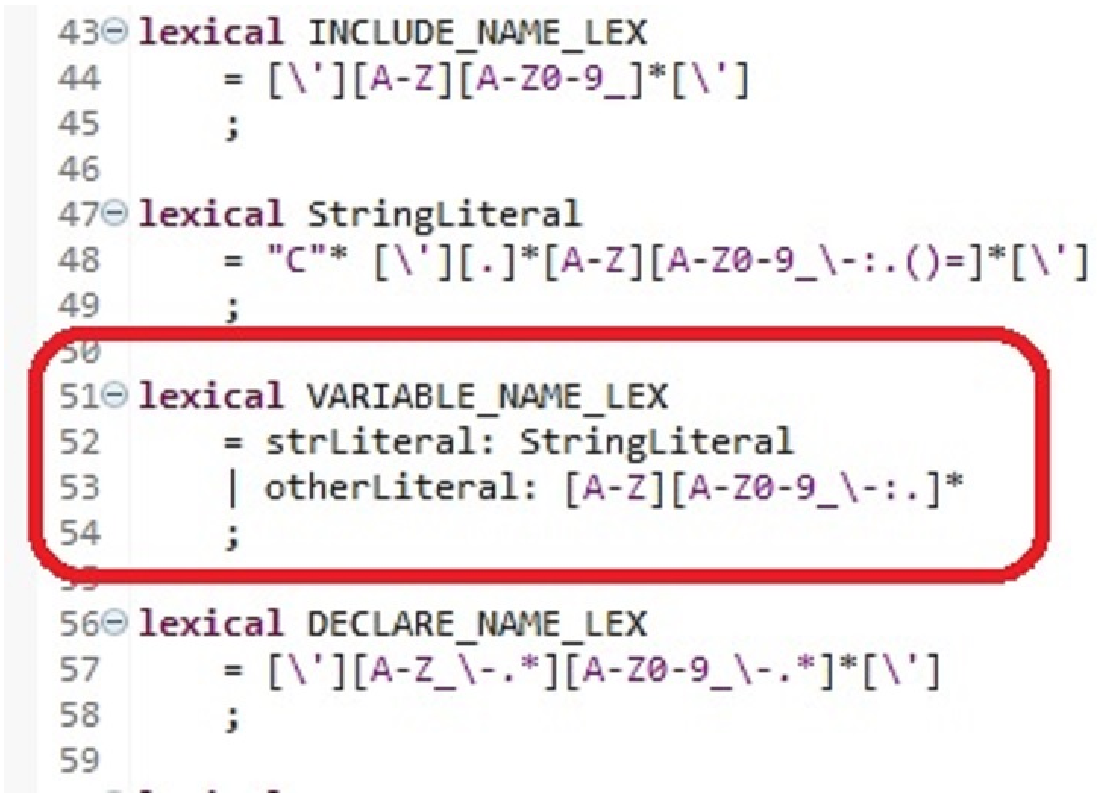
[1]https://www.rascal-mpl.org/start/

meric characters including the ”period” character (.), the ”hyphen” character (-), and the colon character (:), terminating with a single quote. The ”hyphen” character must have a backslash (\) before it because it is a special character to Rascal. This is very similar to the way I had to provide a special character sequence in Overleaf to render the backslash. A valid ATLAS variable name might be:

‘.SNDSEQ:PHASE-1‘

This composition introduces many output problems in most modern languages; hence the need for a data segment for variable storage. There is more discussion about this in the section about the WATLAS integrated framework. The lexical rule defined in the Rascal grammar to define a valid ATLAS variable is encircled in red in Figure 4.1. The rule says an ATLAS variable can be either a string literal which is defined elsewhere in the grammar, or another literal of the form displayed in the regular expression. The regular expression syntax is standard ”regex” syntax. Someone familiar with writing lexers would look at this and find it very familiar. Notice also the pipe character (|) in the lexical rule. This character means ”OR”.

It must also be mentioned here that this construction is not ideal. It works for the sake of this thesis, but the reader must remember that this author was learning Rascal during this thesis and looking back, it would have been better to simply say that an ATLAS variable is only of the form of the regular expression. To minimize the development time of the grammar, shortcuts were taken that would be too hard to undo given the time limitations. But, the thrust of this thesis is not to train the reader in the best grammar construction rather than to explain what it is. Readers with prior grammar construction experience may have noticed the inelegance of the construction.

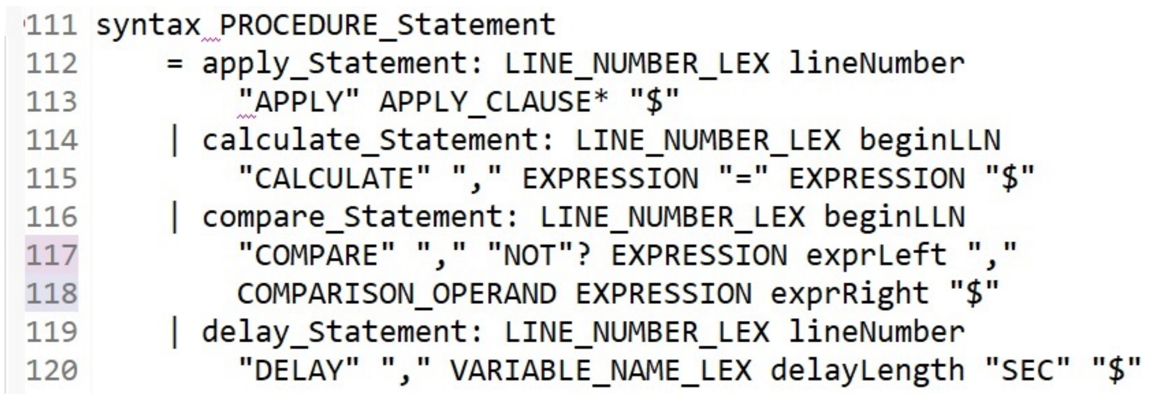


```
lexical INCLUDE_NAME_LEX
    = [\'][A-Z][A-Z0-9_]*[\']
    ;

lexical StringLiteral
    = "C"* [\'][.]*[A-Z][A-Z0-9_\-:.()=]*[\']
    ;

lexical VARIABLE_NAME_LEX
    = strLiteral: StringLiteral
    | otherLiteral: [A-Z][A-Z0-9_\-:.]*
    ;

lexical DECLARE_NAME_LEX
    = [\'][A-Z_\-.*][A-Z0-9_\-.*]*[\']
    ;
```

Figure 4.1: ATLAS Variable Name Lexical Rules.

4.1.2 Parser Rules

Parsing rules can be thought of as the structure of a language's statements. These are often described in a language syntax manual to define the language and are traditionally represented in Backus-Naur Format (BNF). They are usually represented in a visual variation known as railroad diagrams. Therefore, ATLAS has it own language specification that provides the syntax structure of ATLAS statements. ATLAS syntax is represented in its language documentation specification in Extended Backus-Naur Format (EBNF). [8] Some example parsing rules for procedural statements in ATLAS are provided in Figure 4.2.



```
syntax PROCEDURE_Statement
    = apply_Statement: LINE_NUMBER_LEX lineNumber
        "APPLY" APPLY_CLAUSE* "$"
    | calculate_Statement: LINE_NUMBER_LEX beginLLN
        "CALCULATE" "," EXPRESSION "=" EXPRESSION "$"
    | compare_Statement: LINE_NUMBER_LEX beginLLN
        "COMPARE" "," "NOT"? EXPRESSION exprLeft ","
        COMPARISON_OPERAND EXPRESSION exprRight "$"
    | delay_Statement: LINE_NUMBER_LEX lineNumber
        "DELAY" "," VARIABLE_NAME_LEX delayLength "SEC" "$"
```

Figure 4.2: ATLAS Parser rules for Procedural Statements.

This is only a small snippet of the PROCEDURE_Statement syntax rule. But one can see from the example that a PROCEDURE_Statement may be either an applyStatement, or a calculateStatement, or a compareStatement, or a delayStatement, and so on. The complete syntax for these rules is provided in atlassyntax.rsc in the Github links provide in the Appendix. A closer examination of the calculateStatement specifies the valid syntax for an ATLAS CALCULATE statement as follows:

1) A Logical Line Number (LLN) which is defined as a LINE_NUMBER_LEX elsewhere in the grammar file.

2) The keyword CALCULATE followed by a comma (,).

3) An EXPRESSION which is defined elsewhere in the grammar file.

4) An equal sign (=).

5) An EXPRESSION which is defined elsewhere in the grammar file followed by a dollar sign ($). It is an ATLAS convention to start procedural statements with an LLN and terminate with a dollar sign.

As an example, here is a valid sample CALCULATE statement in ATLAS:

123456 CALCULATE, ‘SOME-VARIABLE-NAME‘ = 31 $

One can see that CALCULATE statements are essentially assignment statements.

4.2 Step 2: Using Grammar and ATLAS to Generate C# Equivalence

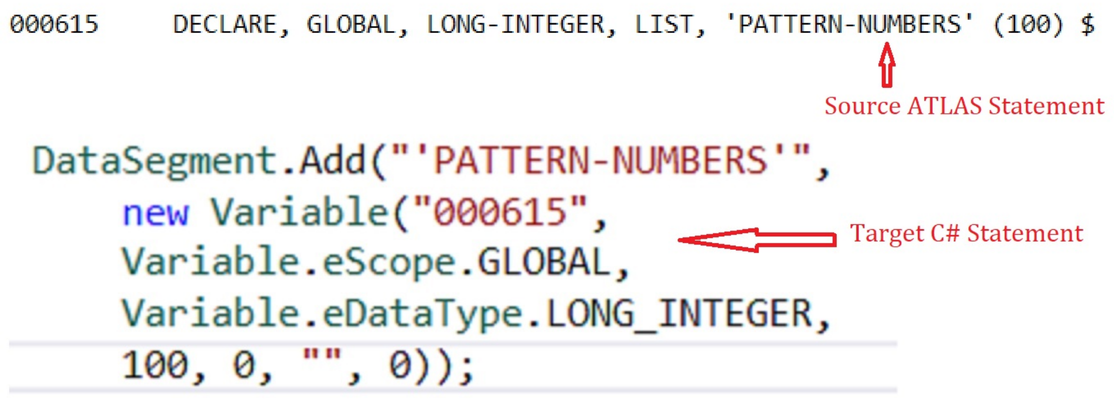
Once the lexical rules have been specified, and the syntax is provided to define a valid statement that can be parsed in the source language, the next step in the creation of the transpiler is create the methods to generate target statements. Before we can do that, we need to develop the abstractions of the statements from the input source language so that we can specify them in the target language.

4.2.1 Defining Expected Output

To help us create an output syntax generation, it is helpful to define expected target output from the source language input statements. For this thesis, the source language is ATLAS and the target language is C#. However, since this thesis generates C# statements that must be compliant within the WATLAS framework, the exact representation of the formatting of WATLAS-framework compliant statements should be defined. There are several considerations and we will discuss one of these.

As stated in subsection 4.1.1 above, ATLAS has an unusual allowable variable name syntax. These lexical rules are so foreign to the target language that to preserve the exact variable name, it is necessary to construct our own data segment to manage variables including not only the variable values, but also the variable's attributes. In the WATLAS framework, a "Variable" class that describes an ATLAS variable object is provided as well as a static class called the "DataSegment" that stores all variables required during WATLAS run-time execution. Thus, the construction of a variable is exemplified below in Figure 4.3.

This describes the process for only one type of translation construction for only one statement type; namely, the DECLARE statement. Each statement type's translation should be defined so that an abstract representation of the statement and the rules



```
000615     DECLARE, GLOBAL, LONG-INTEGER, LIST, 'PATTERN-NUMBERS' (100) $
```

```
DataSegment.Add("'PATTERN-NUMBERS'",
    new Variable("000615",
    Variable.eScope.GLOBAL,
    Variable.eDataType.LONG_INTEGER,
    100, 0, "", 0));
```

Figure 4.3: ATLAS to C# Translation.

to build it from the source input can be specified. In order to facilitate this, there is a small application called ”Translator” that was developed that has a syntax guide for each ATLAS statement type. The user selects a statement type like CALCULATE, and one or more example CALCULATE statements are displayed in ATLAS source syntax. The user can then select one to see the target statement syntax that is compliant with the WATLAS framework.

4.2.2 Abstract Syntax Tree

The Abstract Syntax Tree (AST) is represented as a data structure in Rascal and in most compilers. It is used to represent the structure of source code. We make use of it in Rascal to describe not only the source syntax structure, but also ”what changes” given a particular statement type. This is best described by example and we will refer to the CALCULATE statement again since it is easy to grasp how the AST not only describes the source code input, but assists in the generation of target language output.

Recall from section 4.1.2 that the syntactical structure of the ATLAS source language's CALCULATE statement is:

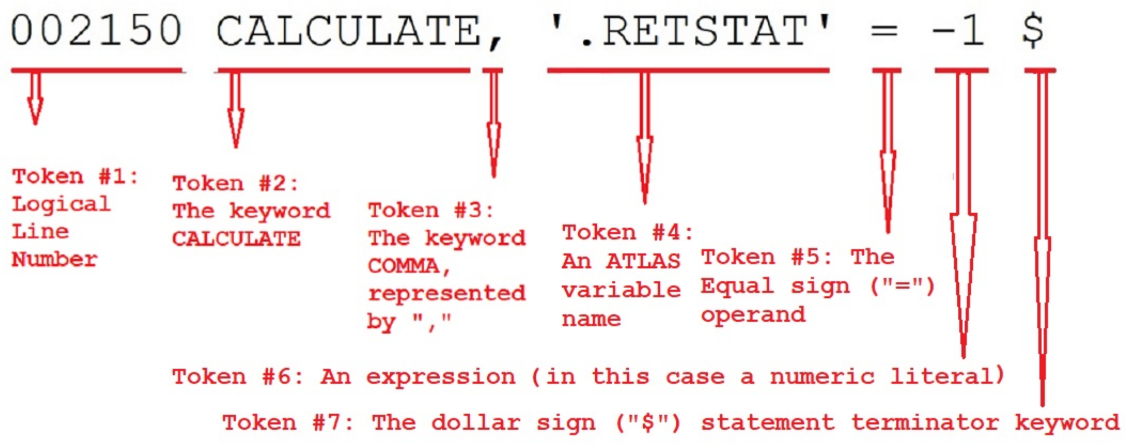

002150 CALCULATE, '.RETSTAT' = -1 $
Token #1: Logical Line Number
Token #2: The keyword CALCULATE
Token #3: The keyword COMMA, represented by ","
Token #4: An ATLAS variable name
Token #5: The Equal sign ("=") operand
Token #6: An expression (in this case a numeric literal)
Token #7: The dollar sign ("$") statement terminator keyword

Figure 4.4: CALCULATE Statement Tokens.

LLN CALCULATE, EXPRESSION = EXPRESSION $

Referring to Figure 4.4, the tokens of this statement are displayed. There are seven tokens in the statement above and of those, only three of them will change depending on the needs of the program.

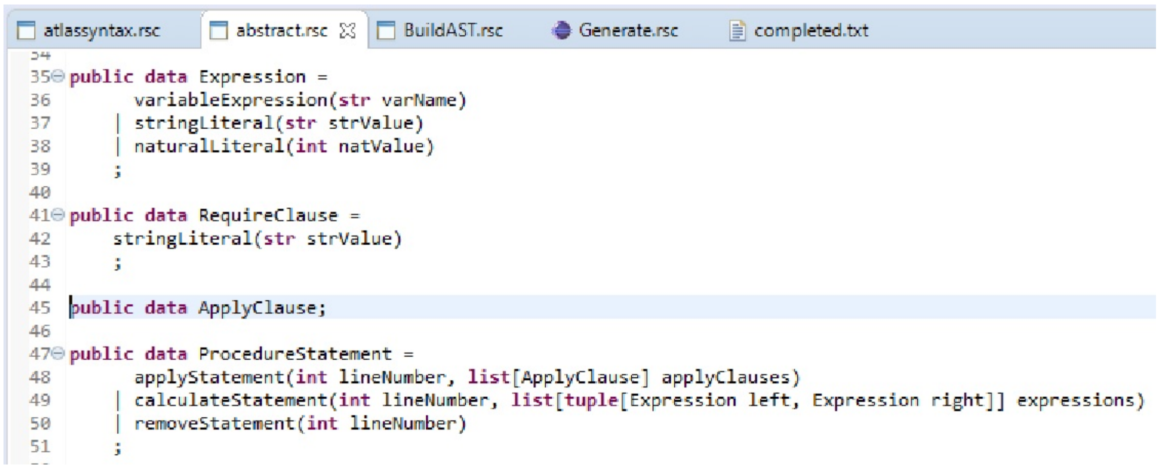
1) The Logical Line Number (LLN) will change,

2) The first EXPRESSION will change, and

3) The Second EXPRESSION will change.

(For the time being, ignore that fact that a CALCULATE statement allows several assignments within one statement, This is accomplished by means of the "one-or-more" operator "()+" in Rascal which we have deliberately left out to minimize confusion). In compiler design we call the tokens that will not change "terminals" and these are often thought of as things like "reserved" words. The tokens that will change are referred to as "nonterminals". The entire statement's syntax and how it is represented is referred to as a "production".

The "CALCULATE" token, the comma (",") token, the equals ("=") token, and the dollar ("$") token must be consistently present in any ATLAS CALCULATE

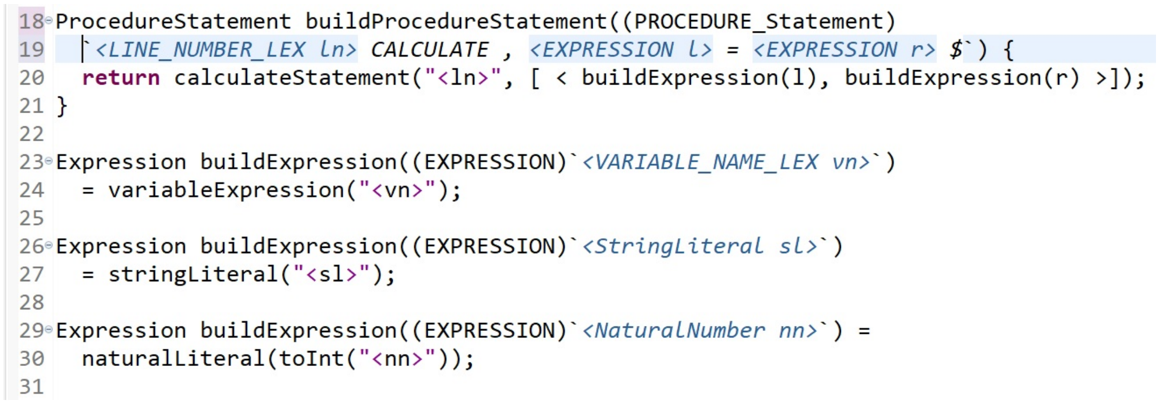


```
public data Expression =
      variableExpression(str varName)
    | stringLiteral(str strValue)
    | naturalLiteral(int natValue)
    ;

public data RequireClause =
    stringLiteral(str strValue)
    ;

public data ApplyClause;

public data ProcedureStatement =
      applyStatement(int lineNumber, list[ApplyClause] applyClauses)
    | calculateStatement(int lineNumber, list[tuple[Expression left, Expression right]] expressions)
    | removeStatement(int lineNumber)
    ;
```

Figure 4.5: AST Data Structure for CALCULATE in Rascal

source code statement in order for the statement to transpile without error. These are the terminals. The programmer decides what the LLN, EXPRESSION 1 and EXPRESSION 2 will be. These are the nonterminals. The ATLAS syntax specifies what tokens must be present and the order in which they are provided in relation to the other tokens in order for the statement to make sense to the ATLAS transpiler so that it can parse the programmer's intended meaning.

Thus, Rascal makes use of the AST construct to describe ”what changes” (i.e.; the programmers choice, or nonterminals) in the ATLAS CALCULATE statement as shown in Figure 4.5. One can see that a CALCULATE statement is considered to be one of many PROCEDURE statements that has three tokens that may change and they are called in that Rascal statement ”lineNumber” for the LLN, ”left” for the left-hand side EXPRESSION, and ”right” for the right-hand side EXPRESSION. Again, referring to the fact that CALCULATE statements allows several assignments within one statement, the Rascal code makes use of the ”list of tuples” to provide for that syntax construction. Once the AST data structure has been defined it may be used to generate the target language source code output.



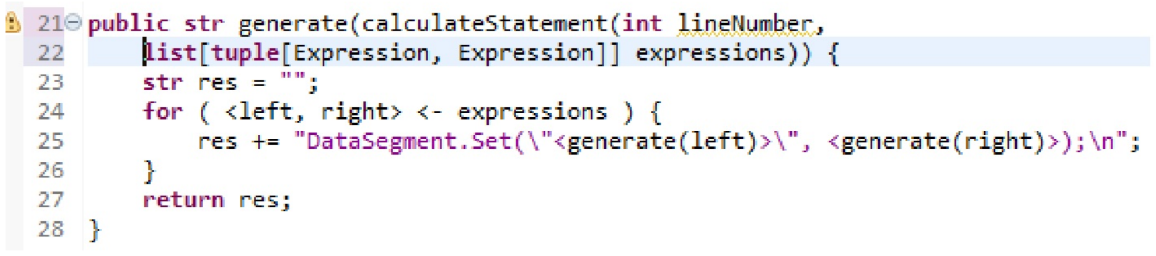
```
ProcedureStatement buildProcedureStatement((PROCEDURE_Statement)
  `<LINE_NUMBER_LEX ln> CALCULATE , <EXPRESSION l> = <EXPRESSION r> $`) {
  return calculateStatement("<ln>", [ < buildExpression(l), buildExpression(r) >]);
}

Expression buildExpression((EXPRESSION)`<VARIABLE_NAME_LEX vn>`)
  = variableExpression("<vn>");

Expression buildExpression((EXPRESSION)`<StringLiteral sl>`)
  = stringLiteral("<sl>");

Expression buildExpression((EXPRESSION)`<NaturalNumber nn>`) =
  naturalLiteral(toInt("<nn>"));
```

Figure 4.6: AST Production for CALCULATE in Rascal

4.2.3 Generating Output

Once the lexical rules, parsing syntax, translation representations and AST are defined, it is possible to begin building code generation procedures. This is not to say that the preceding steps must be completed in their entirety. Rascal allows iterative development of these components, but it is a good idea to have the bulk of the lexical rules specified and the large majority of the parsing syntax created. Changing an elemental rule late in the process can break a lot of the building blocks upon which a grammar is built.

To construct a code generation procedure, we start with an abstraction of the statement we will be receiving as input. We have shown this abstraction as an AST data structure in Figure 4.5. Then we create a procedure (method) to construct the abstract syntax tree assuming that abstraction will be our input. This procedure is displayed in Figure 4.6. Notice that this method calls other overloaded methods to process input data according to its allowable types. Remember that allowable types for Expressions to CALCULATE statement operands may be ATLAS variable names (VARIABLE_NAME_LEX) or string literals (StringLiteral) or natural numbers (NaturalNumber).



```
public str generate(calculateStatement(int lineNumber,
    list[tuple[Expression, Expression]] expressions)) {
    str res = "";
    for ( <left, right> <- expressions ) {
        res += "DataSegment.Set(\"<generate(left)>\", <generate(right)>);\n";
    }
    return res;
}
```

Figure 4.7: Rascal Code to Generate Target Statement from AST

Finally, we will code up a generation method to receive the abstract syntax tree and pull out the pieces meaningful to us in our target language output and format the output statement. For example, we will once again consider the construction of a CALCULATE statement and consider how its target syntax is generated. Recall from 4.1.2 that CALCULATE statements are data assignment statements that allow expressions for the right-hand side of the equivalence operator.

Here is an ATLAS source example:

123456 CALCULATE, ‘SOME-VARIABLE-NAME‘ = 31 + 4 $

In Figure 4.7, the Rascal code generation procedure that was used to construct the target statement is displayed. Note that from the ”Build” method, the generate method receives the ”lineNumber” arguments and the list of tuples containing the operand on the left-hand side of the CALCULATE operator, and the right-hand side of the CALCULATE operator. In our ATLAS source example, we are only setting ‘SOME-VARIABLE-NAME‘ to the sum of 31 + 4, but we could have included more assignment statements since this is allowed in ATLAS CALCULATE statements. But, here we just process an array with one tuple of values. From that we build that output string which looks like the following C#:

DataSegment.Set(”’SOME-VARIABLE-NAME’”, 31 + 4);

4.3 Step 3: Developing the WATLAS Integrated Framework

4.3.1 Framework Components

WATLAS is only designed to work within a discrete framework. While the code generation routines output standard C#, much of that C# code only makes sense within the WATLAS framework. From our example in 4.2.3, one can see that the generated code references a structure which is called the "DataSegment". The "DataSegment" is used to simulate a run time data structure to manage variables including their creation and definition, setting values for those variables, and subsequently retrieving values stored in those variables in the "DataSegment". This "DataSegment" is often, in definitions of languages, also called an "environment". This construction is common in programming languages, and "every programming language specifies an execution model, and many implement at least part of that model in a run-time system". [2] For example, in the LISP language "the compiler may assume that functions that are defined and declared inline in the compile-time environment will retain the same definitions at run time". [3]

Similarly, ATLAS REQUIRE statements make reference to an "AssetManager" which simulates the configuration and I-O interface to attached hardware which are referred to as "assets" in the CASS world. Also ATLAS INCLUDE statements refer to framework objects like "L2_CHAIN" and so on. These structures are not provided "out-of-the-box" in a C# environment. They were built beforehand to allow our transpiled code to interface correctly with the CASS Station Emulator. In this way, we accomplish one of the goals of this thesis to assess the viability of this approach to migrating ATLAS from OpenVMS to Windows. So, why did we follow this path?

[2] https://en.wikipedia.org/wiki/Runtime_system

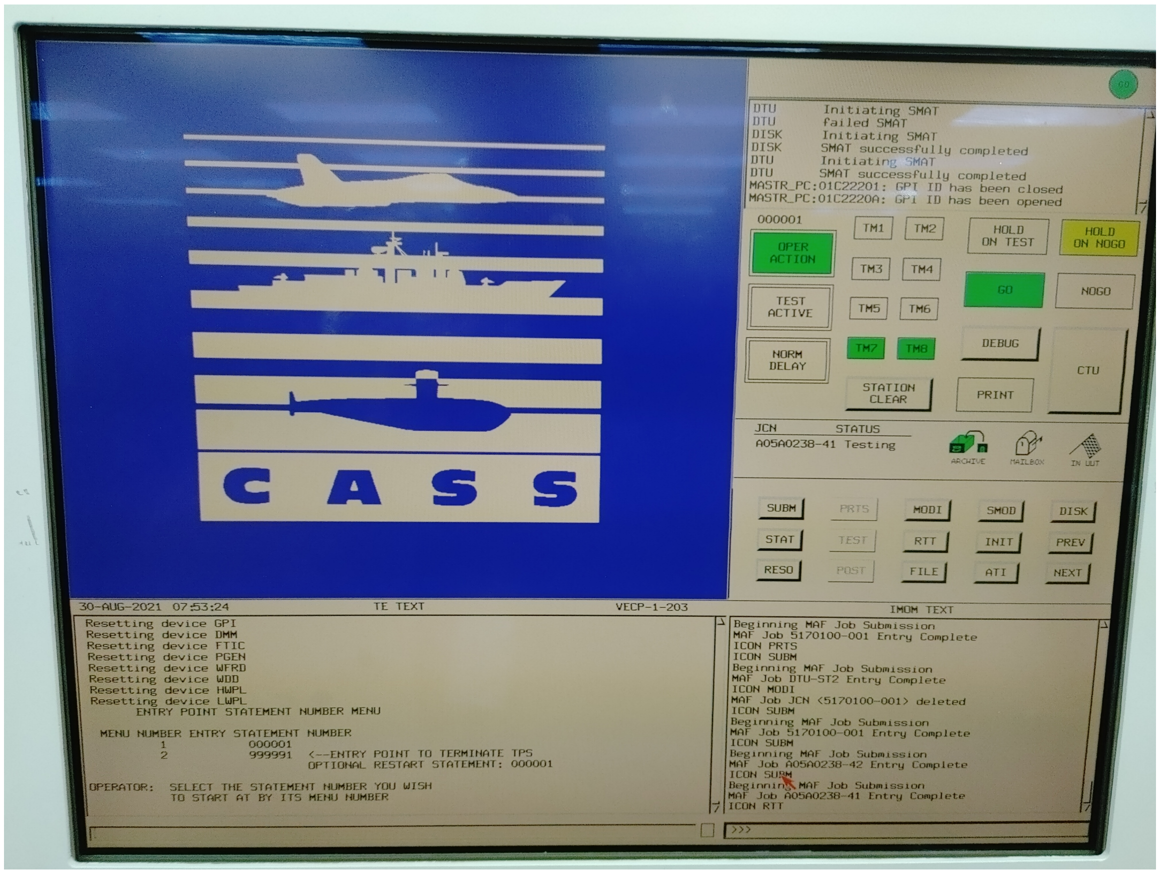
[3] https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node227.html

The main reason for creating a discrete framework was the time limitation to building this tool. A project of this scope could easily take a team of experienced programmers several years to accomplish. As we will see in our next chapter on competing technologies, we find that RTCASS which provides a similar capability, although with an integrated hardware appliance, took the Boeing Corporation more than a decade to field. A large team of developers and hardware engineers and several million dollars were invested to develop RTCASS. And even their tool with all of its capabilities only operates within the framework that they had previously built. The RTCASS translater generates native C source code, but relies on custom components that Boeing built to support RTCASS hardware functionality. Building a tool using the technologies we employed would also incur a huge effort that would be well beyond the scope of this thesis.

Our framework provides the structures that will be needed to support our simple "Hello World" prototype and nothing else. This includes the libraries and classes discussed above and an integration to a CASS Station emulator that will provide a visual display as to how our tool will operate on the ATLAS source code that has been transpiled into C#. The WATLAS emulator provides a look-and-feel very similar to the CASS Station's Human Interface.

4.3.2 Defining a Project Template

Given that WATLAS operates within a discrete framework, a review of how we expose that framework is in order here. Since C# is typically developed in the Microsoft Integrated Development Environment (IDE) known as Visual Studio, we constructed a project template that incorporates our framework and developed routines within our Pre/Post Processor to establish the project framework by instantiating the project based on the WATLAS project template. After the processor prepares that ATLAS



```
DTU        Initiating SMAT
DTU        failed SMAT
DISK       Initiating SMAT
DISK       SMAT successfully completed
DTU        Initiating SMAT
DTU        SMAT successfully completed
MASTR_PC:01C22201: GPI ID has been closed
MASTR_PC:01C2220A: GPI ID has been opened
```

```
000001
```

```
JCN            STATUS
A05A0238-41  Testing
```

```
30-AUG-2021  07:53:24                 TE TEXT                          VECP-1-203
 Resetting device GPI
 Resetting device DMM
 Resetting device FTIC
 Resetting device PGEN
 Resetting device WFRD
 Resetting device WDD
 Resetting device HWPL
 Resetting device LWPL
        ENTRY POINT STATEMENT NUMBER MENU

  MENU NUMBER ENTRY STATEMENT NUMBER
          1            000001
          2            999991  <--ENTRY POINT TO TERMINATE TPS
                               OPTIONAL RESTART STATEMENT: 000001

OPERATOR:  SELECT THE STATEMENT NUMBER YOU WISH
           TO START AT BY ITS MENU NUMBER
```

```
IMOM TEXT
Beginning MAF Job Submission
MAF Job 5170100-001 Entry Complete
ICON PRTS
ICON SUBM
Beginning MAF Job Submission
MAF Job DTU-ST2 Entry Complete
ICON MODI
MAF Job JCN <5170100-001> deleted
ICON SUBM
Beginning MAF Job Submission
MAF Job 5170100-001 Entry Complete
ICON SUBM
Beginning MAF Job Submission
MAF Job A05A0238-42 Entry Complete
ICON SUBM
Beginning MAF Job Submission
MAF Job A05A0238-41 Entry Complete
ICON RTT
>>>
```

Figure 4.8: IMOM Interface

source code for translation and performs the translation, the processor detects the name of the project (in our case: HELOWRLD) and builds a project from the WATLAS project template and names the source code file within it as HELOWRLD.CS. Any experienced C# developer can customize the project after its initial creation to suit their needs. But all of the necessary components of the framework are included in the initial project deployment. One of these is the CASS Station emulator.

4.3.3 The CASS Station Human Interface

In Figure 4.8 we see a depiction of the actual human interface on a CASS station for a TPS developer. It is referred to as the ATE display, but is known as the Intermediate

Maintenance Operations Management System (IMOM) in program code. One can see from the graphic that it certainly has a very "dated" look. Since it was initially designed in the 1970's, and later updated in the 1990's, one can understand why it does have this antiquated look. ATLAS programs can be written with a minimal interface to the IMOM, but more complex interaction allows the TPS developer to have greater control of the execution of the ATLAS instructions and interrogate values and reset them. For this thesis, only minimal interface with components of the IMOM is depicted.

For WATLAS, the Project Template includes a Windows Form that is referenced as "fMain" in the transpiled WATLAS statements for those statements that may need to reference the ATE Display. These include statements directed to the CRT output, and statements that reference "assets" since the CASS station reports on the status of any station asset. Further, there is an output window that reports on TPS program status (start, stop, run-time errors, etc.). For this simulator, the effort was made to give a visual representation for those statements with a similar affect as it occurs on the actual CASS Station's ATE display.

4.3.4 Pre and Post Processor

One more tool that was developed to support this thesis should be discussed here. The PrePostProcessor is used to format ATLAS source code in a format that is easily processed by the Rascal console. For example, consider the source code displayed in Figure 3.1. One can see that ATLAS developers apply a standardized indentation scheme that lends itself to easy interpretation by a human reader. There also exists a coding standard that enforces this practice in all ATLAS source. The standard makes no difference to the ATLAS transpiler since it is just processing tokens and knows what characters to pay attention to, and which should be ignored. But, the line end-of-line characters, tabs and carriage returns make the task of submitting formatted ATLAS source statements to our Rascal console a bit challenging. Also, the need to ignore ATLAS comments needs to be performed by pre-processing since no effort to handle the parsing necessary to ignore comments is included in this project's Rascal parser.

So, pre-processing would take a line of formatted ATLAS code structured like this:

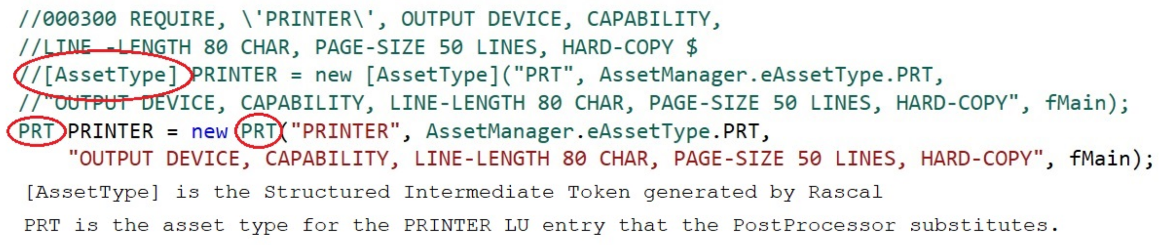
```
011015 CALCULATE,
 'REPEAT-ENABLED' = TRUE,
 'REPEAT-TEST' = TRUE $
```

and transforms it into this as a single line:

```
parse(#PROCEDURE_Statement, "011015 CALCULATE, 'REPEAT-ENABLED' = TRUE,
'REPEAT-TEST' = TRUE $");
```

This is the statement format that our Rascal console needs to work with.

The purpose of the post-processing is to make substitutions for intermediate tokens that our Rascal code generator has created. Rascal has no awareness of the hardware "asset" mapping requirements, so it simply writes out a structured intermediate token. This is due to the substitutions that take place for CASS station "assets" that have been defined in the "LU" file. Recall from subsection 3.9 that the LU file contains the mapping between a logical asset name and its physical device. Post-processing searches the Rascal-generated code and finds the structured intermediate tokens that it must replace before finalizing the C# target source code. We see an example of this in Figure 4.9. Note that the lines were originally output as three lines, but carriage returns have been included here to enhance legibility. It is six lines in the graphic.



```
//000300 REQUIRE, \'PRINTER\', OUTPUT DEVICE, CAPABILITY,
//LINE -LENGTH 80 CHAR, PAGE-SIZE 50 LINES, HARD-COPY $
//[AssetType] PRINTER = new [AssetType]("PRT", AssetManager.eAssetType.PRT,
//"OUTPUT DEVICE, CAPABILITY, LINE-LENGTH 80 CHAR, PAGE-SIZE 50 LINES, HARD-COPY", fMain);
PRT PRINTER = new PRT("PRINTER", AssetManager.eAssetType.PRT,
    "OUTPUT DEVICE, CAPABILITY, LINE-LENGTH 80 CHAR, PAGE-SIZE 50 LINES, HARD-COPY", fMain);
[AssetType] is the Structured Intermediate Token generated by Rascal
PRT is the asset type for the PRINTER LU entry that the PostProcessor substitutes.
```

Figure 4.9: Example PostProcessor Structured Intermediate Token Substitution

4.4 Next Steps - Enhancing WATLAS

In Chapter 6, there is a section on Viability Concerns and a final solution is offered in Chapter 7 at Figure 7.1 that looks like the best option going forward for variable name translation. Implementing that is the immediate next step to enhance WATLAS. This will allow for the complete removal of the ”DataSegment” from the WATLAS framework. This is good because it minimizes the need for C# structures that do not come immediately ”out-of-the-box” with C# thereby minimizing the learning curve of the WATLAS framework. It also resolves the entire ”casting” issue discussed in Chapter 7 as well.

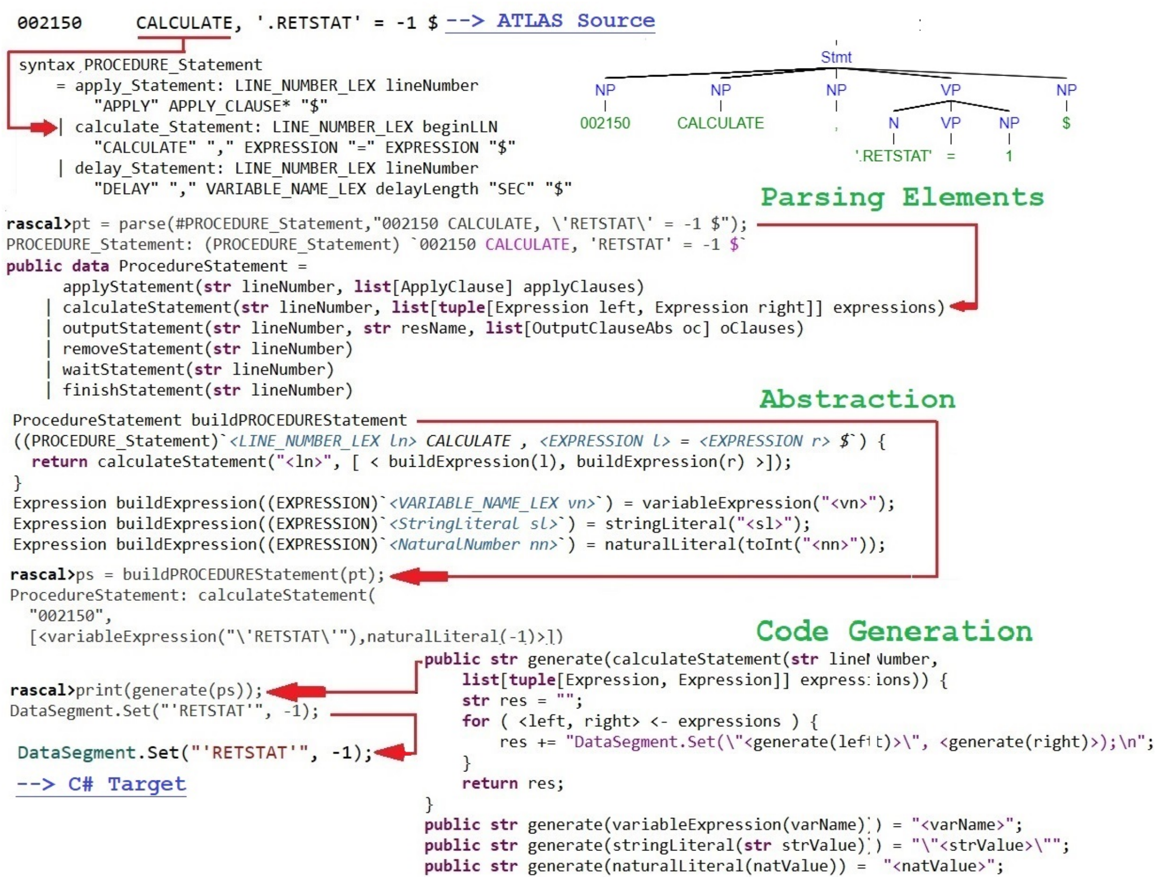
4.4.1 Rascal Project Enhancements

Assuming this enhancement is complete, here we discuss how individuals may contribute to this effort and bring together a more comprehensive WATLAS. The Appendix provides the links to all of the source code that contributors should download as a starting point. The next effort should involve completion of the parsing syntax in atlassyntas.rsc in the GitHub project. Extensive effort was made during the first semester of this thesis to provide most of the required syntax parsing, but it is not comprehensive. The best path forward would involve lifting the transpile logic from the PrePostProcessor and incorporating it into a global parser that would iterate through a directory of ATLAS source code and report statements that fail to parse correctly. The transpile logic is in a standalone class named ATLAS.cs in the ATLAS_Rascal project in the Azure DevOps solution. It would be fairly easy to lift that class out of the transpiler for implementation in a different project. The class constructor accepts as string parameters all the inputs it needs to run in any C# process. This new process could scan directories looking for *.AS files, transpile them, and provide report output for statements that fail to parse. This would provide analysis of what syntax is not supported by the translator and one could add syntax corrections and enhancements for what is missing.

Refer to Figure 4.10 which shows the flow of translation and specifically to the green "Abstraction" portion of the graphic. The abstraction definitions found in abstract.rsc need to be completed. Only about seven statement types are supported as of this writing. They are the statement types that were needed to perform the initial evaluation of WATLAS functionality. But, the remaining statements that need abstraction are all PROCEDURAL statements as one can see in the data structure named:

```
public data ProcedureStatement
```

Once all of the remaining abstractions are added, the "build" procedures for the abstract syntax tree in BuildAST.rsc need to be included. One can copy one of the existing `buildPROCEDUREStatement` functions and modify it to suit the needs of the abstraction. Finally, for each build procedure, a "generate" function in the Generate.rsc module would need to be included. Referring again to Figure 4.10, the green "Code Generation" portion of the graphic shows the structural elements that comprise generation of target C# code. As with the AST, one can copy one of the existing "generate" functions and modify it to suit the needs of the abstraction. There is a TXT file in the Documentation folder in the ATLAS_Rascal project in



```
002150     CALCULATE, '.RETSTAT' = -1 $
```

```
syntax PROCEDURE_Statement
    = apply_Statement: LINE_NUMBER_LEX lineNumber
        "APPLY" APPLY_CLAUSE* "$"
    | calculate_Statement: LINE_NUMBER_LEX beginLLN
        "CALCULATE" "," EXPRESSION "=" EXPRESSION "$"
    | delay_Statement: LINE_NUMBER_LEX lineNumber
        "DELAY" "," VARIABLE_NAME_LEX delayLength "SEC" "$"
```

```
rascal>pt = parse(#PROCEDURE_Statement,"002150 CALCULATE, \'RETSTAT\' = -1 $");
PROCEDURE_Statement: (PROCEDURE_Statement) `002150 CALCULATE, 'RETSTAT' = -1 $`
public data ProcedureStatement =
        applyStatement(str lineNumber, list[ApplyClause] applyClauses)
      | calculateStatement(str lineNumber, list[tuple[Expression left, Expression right]] expressions)
      | outputStatement(str lineNumber, str resName, list[OutputClauseAbs oc] oClauses)
      | removeStatement(str lineNumber)
      | waitStatement(str lineNumber)
      | finishStatement(str lineNumber)
```

```
ProcedureStatement buildPROCEDUREStatement
((PROCEDURE_Statement)`<LINE_NUMBER_LEX ln> CALCULATE , <EXPRESSION l> = <EXPRESSION r> $`) {
  return calculateStatement("<ln>", [ < buildExpression(l), buildExpression(r) >]);
}
Expression buildExpression((EXPRESSION)`<VARIABLE_NAME_LEX vn>`) = variableExpression("<vn>");
Expression buildExpression((EXPRESSION)`<StringLiteral sl>`) = stringLiteral("<sl>");
Expression buildExpression((EXPRESSION)`<NaturalNumber nn>`) = naturalLiteral(toInt("<nn>"));
```

```
rascal>ps = buildPROCEDUREStatement(pt);
ProcedureStatement: calculateStatement(
  "002150",
  [<variableExpression("\'RETSTAT\'"),naturalLiteral(-1)>])
```

```
public str generate(calculateStatement(str lineNumber,
    list[tuple[Expression, Expression]] expressions)) {
    str res = "";
    for ( <left, right> <- expressions ) {
        res += "DataSegment.Set(\"<generate(left)>\", <generate(right)>);\n";
    }
    return res;
}
public str generate(variableExpression(varName)) = "<varName>";
public str generate(stringLiteral(str strValue)) = "\"<strValue>\"";
public str generate(naturalLiteral(natValue)) =  "<natValue>";
```

```
rascal>print(generate(ps));
DataSegment.Set("'RETSTAT'", -1);
```

```
DataSegment.Set("'RETSTAT'", -1);
```

Figure 4.10: Example Flow of Translation

Azure DevOps that has a lot of test statements that work. The file is appropriately named "THIS_WORKS.TXT" and they can be copied and pasted into the Rascal console to see how they work. The ATLAS_Rascal project can be used to test any ATLAS source code file and should not need any enhancements to make it work with any new PROCEDURAL statements that you might create, since they were generalized in the PrePostProcessor. The contributor only needs to take care to name their build procedure like the other examples. Any new build methods for PROCEDURAL statements should be named "buildPROCEDUREStatement" since that method is overloaded and Rascal knows which method to use in response to a call.

4.4.2 C# Project Enhancements

The major enhancements to the C# code involves implementing the various assets in the WATLAS template project in the folder named "Assets". This effort would require the assistance of electrical engineers with subject matter expertise in how interfaces are written to communicate with the various devices. This is not a trivial effort. The specifics of the communication protocols differ depending on type of asset, the asset's vendor, and the differentiation between the vendor's model offerings. The good news is that most of the product offerings on the market are supported on Windows and some of the products reviewed come with an API to assist in the development of an interface with your application. As an example, CONTEC has a vast array of hardware to support engineering applications and a link to their Digital Multimeter's (DMM) API is provided in the footnotes. [4]

4.4.3 Tool Enhancements

Rascal is a powerful meta-programming tool. But, at this time the Rascal console is the main interface to issuing Rascal commands and it has a limited set of visual tools to assist the developer. Meta-programming is a hard concept to grasp and the cumbersome nature of its interface adds effort to learning and coding, unless you are a very good typist (which this author is not).

One tool that would be very useful in Rascal would be a Backus-Naur Format (BNF) visualizer. In other words it would be nice if one could select a grammar file and open it as a series of railroad diagrams. All the lexical structures could be exposed in a property window, and all the syntax statements exposed in a property window, and when the user clicks on a statement, it could show how it complies to a

[4] https://www.contec.com/products-services/daq-control/pc-helper/daq-software/api-dmm(wdm)/specification/

BNF visual. If that visual was editable, then the graphical interface could take care of the Rascal syntax that the visual implies.

Chapter 5

Competing Technologies and Similar Efforts

As stated in the last chapter, this thesis' project employs a number of tools such as Rascal, Java and Visual Studio to create a parsing engine for ATLAS source code on Windows, create an abstract syntax tree and eventually source code output that will compile on Windows. This chapter will examine a number of other solutions that can create the same or similar deliverable. These approaches range from a market-ready Windows-based ATLAS compiler, meta-programming languages similar to Rascal, tools developed by Boeing specifically for the Department of Navy, and a generic compilation strategy for any language but targeted mainly for legacy mainframe applications. We will also examine one solution that does not satisfy the requirements completely, but is worth mentioning since it provides the ability to generate executable code from an interim language. For this solution, Rascal or a tool that provides similar capabilities of defining lexical rules, parsing, defining abstractions, and generating target code, could be tailored to create the interim language as output. Finally, we will take a look at a translation of another legacy language that enables modern support in military applications.

5.1 PAWS

PAWS Developer's Studio gives you the power to compile, modify, debug, document, and simulate the operation of ATLAS test programs in a Windows environment. [1] The PAWS ATLAS compiler "processes any of the available ATLAS subsets supported by PAWS. This fast, comprehensive compiler performs source code parsing, syntax verification, full signal flow analysis, and automatic resource allocation and code generation". [1] The company's website provides a high-level glossy of information on product features. An informal phone interview with Mike Rutledge of Astrionics provided much of the material for this review of PAWS' capabilities. These are the answers to five specific questions:

5.1.1 Transfer-ability of OpenVMS-Compliant ATLAS Source Code

With the exception of legacy source code that does not adhere to strict syntax guidance, PAWS will process legacy source in the same manner as an OpenVMS ATLAS compiler. The OpenVMS compiler can be a bit more forgiving of syntax nuances, particularly where the program's development predates the ATLAS standard library. PAWS is entirely compliant with the IEEE ATLAS standard and will provide errors for erroneously constructed ATLAS syntax. PAWS provides a PreProcessor to assist with migration of code that is not fully compliant to strict syntax adherence since Astrionics has years of experience encountering these types of syntax errors.

5.1.2 Target Executable Implementation

PAWS compiles ATLAS to "ATLAS Intermediate Language" (AIL) and does not generate an x86 executable. AIL will only execute inside of the PAWS executive

[1] https://www.astronics.com/productinfo?productgroup=test

which recognizes the intermediate representation of the ATLAS source. Mr. Rutledge conveyed that this provides an extra layer of obfuscation that is valuable with regard to Cyber Security concerns since the AIL is immediately disposed.

5.1.3 Asset Interfaces

Surprisingly, Astrionics does not market or recommend any hardware assets, but claims to support all Windows-connectable devices. For calls to those assets that may not exist at run-time, the PAWS executive provides a virtual simulation which is very similar to the deployment of hardware assets scenarios envisioned for WATLAS (see subsection 4.4.2). Astrionics does provide an online library of downloadable virtual assets with simulation capabilities for free to registered PAWS users.

5.1.4 UUT Latching

PAWS makes no distinction between Automated Test Equipment (ATE) assets and Unit Under Test (UUT) assets. While a CASS Station has embedded ATE and the engineer latches the UUT to the UUT latching mechanism which is a general-purpose interface, these types of hardware interfaces are not present for the Windows-based developer. The developer needs to determine cabling and interface needs and PAWS does not distinguish either as unique hardware (i.e. the avionics hardware being used to test, and the avionics hardware being tested). It is the responsibility of the electrical engineer to connect the correct device with the correct interface connector(s). Thus, there is a no latching mechanism.

5.1.5 Industry Standard IDE Integration

PAWS is deployed with its own IDE that does not provide additional interface with other IDEs. There are no future plan to provide any further integration.

5.2 ANTLR

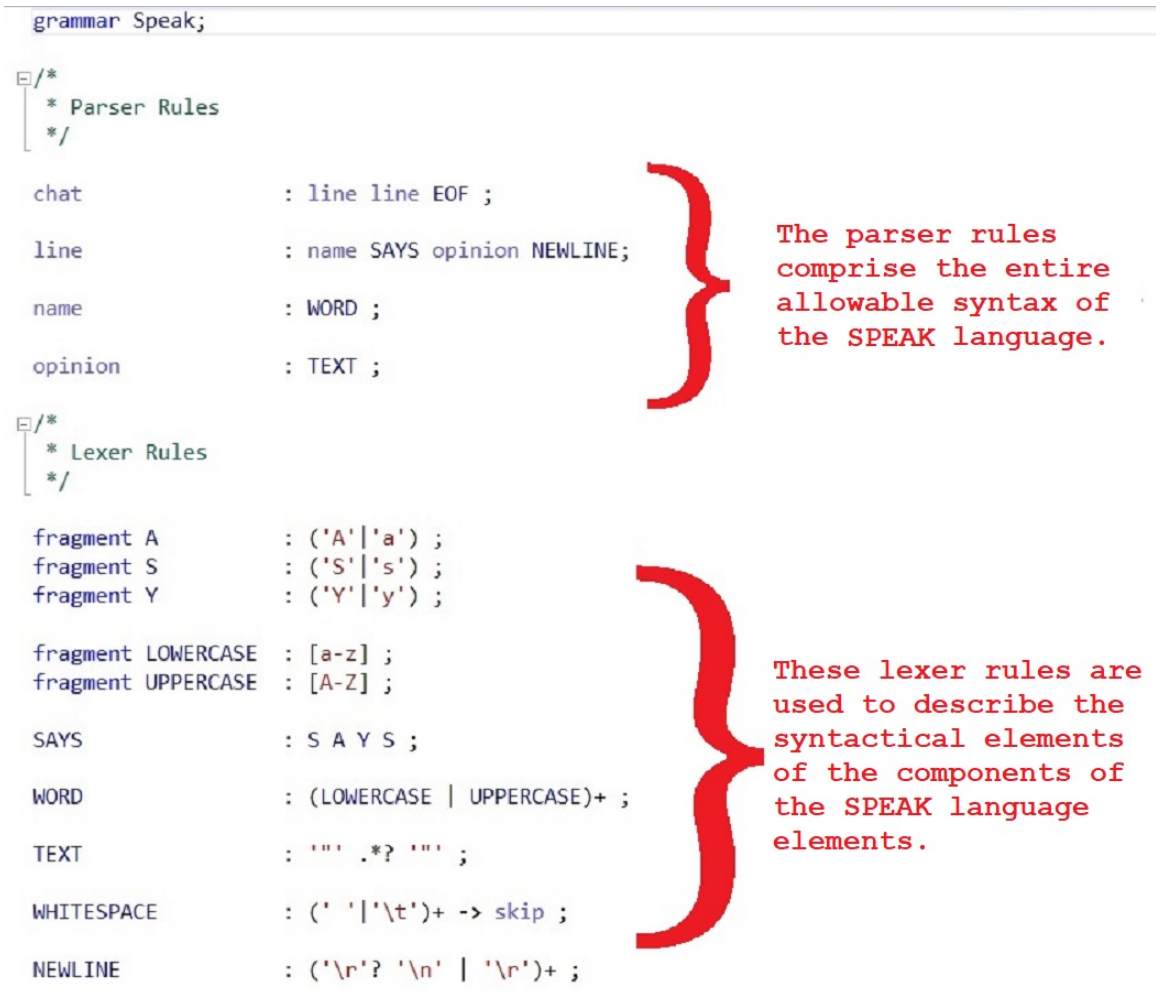
ANTLR (Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. [12]

Setting up ANTLR for this thesis was very intuitive. "While the tool itself is written in Java, it can also be used to generate a parser in several other languages, for instance Python, C# or JavaScript (with more languages supported by the recently released 4.6 version). " [14] A comparative project for using ANTLR with a parser/lexer that is written in C# is demonstrated below.

An example grammar file is provided in the "Getting started with ANTLR in CSharp" GitHub project. It is depicted in Figure 5.1. Notice that the grammar states that valid SPEAK source code programs are comprised of two lines that terminate with end-of-file. Further, a valid line is a "name", followed by the "SAYS" keyword, followed by an "opinion" and terminated by a "NEWLINE" character. The rules for what a "name" is are given in the lexer rules and those rules indicate it is comprised of upper or lower case alpha characters. The rules about what an "opinion" is indicate that it is a quoted string expression, and so on.

An example program with a syntactically correct stream of text is depicted in Figure 5.2. Also, an example of a syntactically incorrect stream of text follows in Figure 5.3.

One can see that ANTLR relies on the development of a grammar file that is similar in nature to Rascal's RSC grammar file. Both technologies approach parsing and lexing rules with Backus Naur Form (BNF) meta-syntax notation. Both also utilize regular expression-like syntax for character encoding rules. With this in mind,

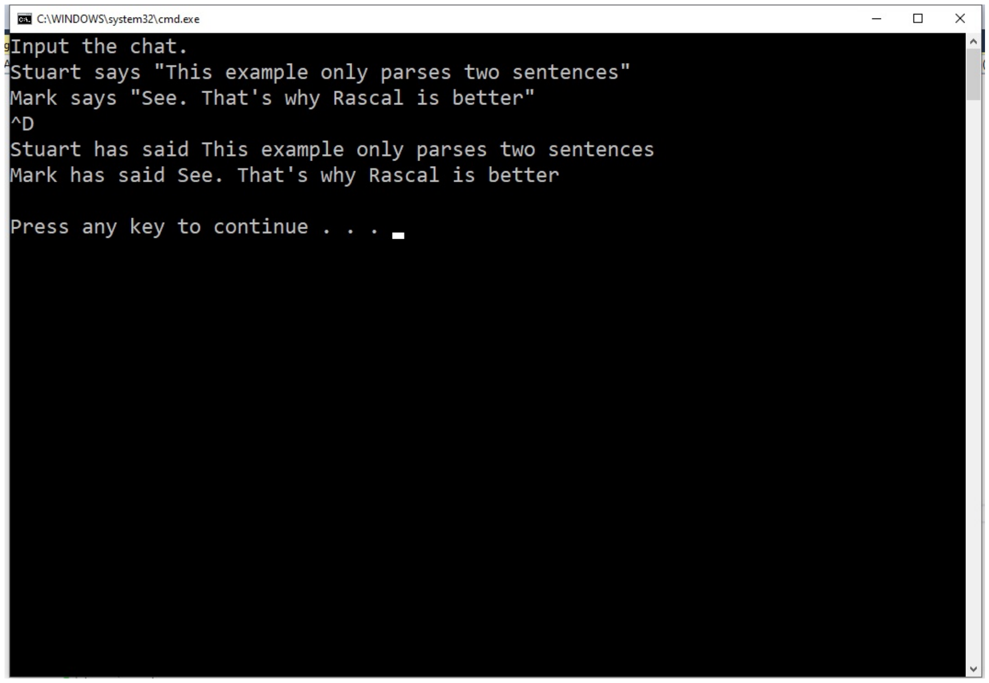


```
grammar Speak;

/*
 * Parser Rules
 */

chat                : line line EOF ;

line                : name SAYS opinion NEWLINE;

name                : WORD ;

opinion             : TEXT ;

/*
 * Lexer Rules
 */

fragment A          : ('A'|'a') ;
fragment S          : ('S'|'s') ;
fragment Y          : ('Y'|'y') ;

fragment LOWERCASE  : [a-z] ;
fragment UPPERCASE  : [A-Z] ;

SAYS                : S A Y S ;

WORD                : (LOWERCASE | UPPERCASE)+ ;

TEXT                : '"' .*? '"' ;

WHITESPACE          : (' '|'\t')+ -> skip ;

NEWLINE             : ('\r'? '\n' | '\r')+ ;
```

The parser rules comprise the entire allowable syntax of the SPEAK language.
These lexer rules are used to describe the syntactical elements of the components of the SPEAK language elements.

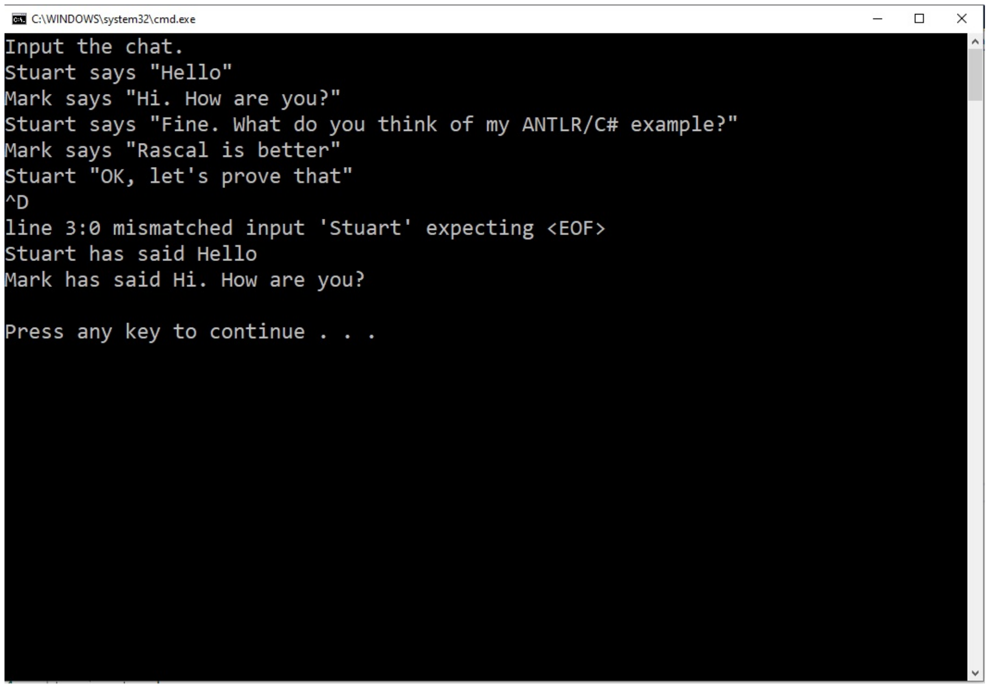
Figure 5.1: ANTLR Grammar Example.

it is highly likely that the amount of effort expended to develop WATLAS using Rascal would be very similar had ANTLR been employed.



```
Input the chat.
Stuart says "This example only parses two sentences"
Mark says "See. That's why Rascal is better"
^D
Stuart has said This example only parses two sentences
Mark has said See. That's why Rascal is better

Press any key to continue . . .
```

Figure 5.2: SPEAK Example with no errors.



```
Input the chat.
Stuart says "Hello"
Mark says "Hi. How are you?"
Stuart says "Fine. What do you think of my ANTLR/C# example?"
Mark says "Rascal is better"
Stuart "OK, let's prove that"
^D
line 3:0 mismatched input 'Stuart' expecting <EOF>
Stuart has said Hello
Mark has said Hi. How are you?

Press any key to continue . . .
```

Figure 5.3: SPEAK Example with errors detected.

5.3 Haskell

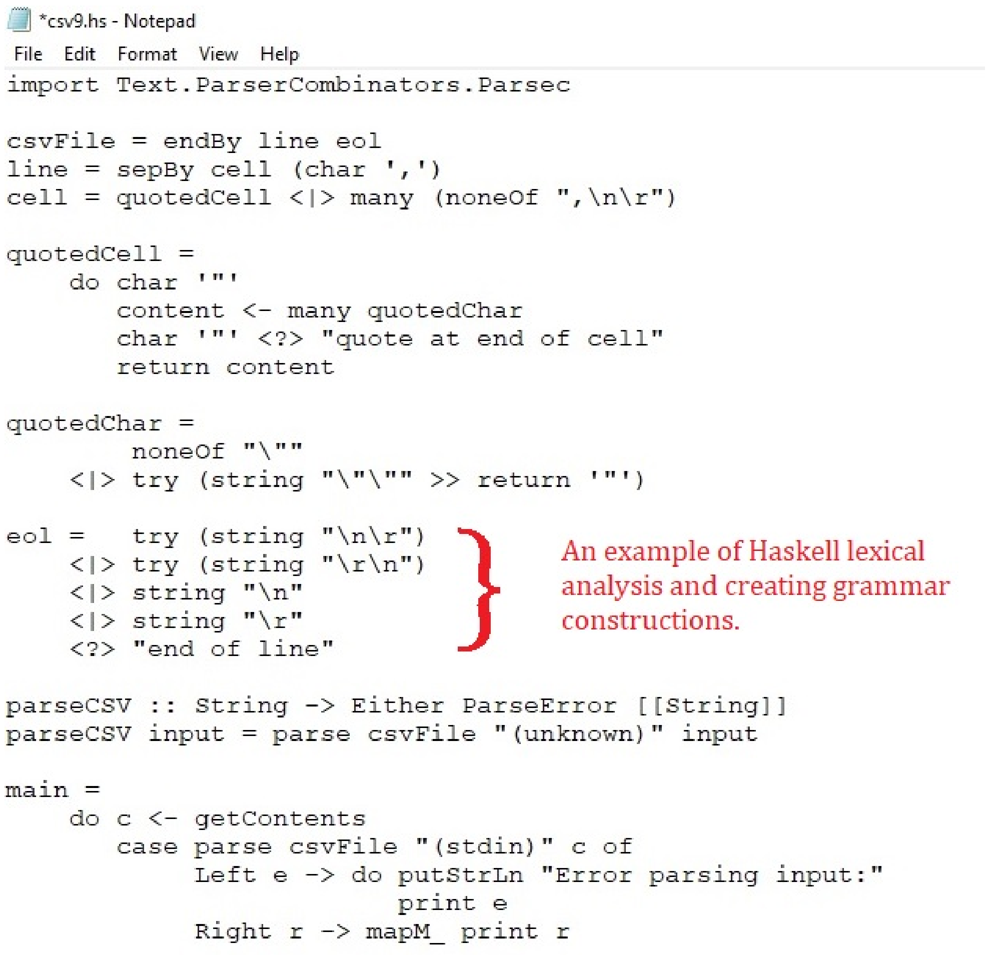
Haskell is a polymorphic statically typed, lazy, purely functional language, quite different from most other programming languages. The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on the lambda calculus. [7]

Haskell provides a parsing library that is capable of lexing and parsing in the Text.ParserCombinators.Parsec import library. The Haskell Language Server also provides a extension plug-in for Visual Studio code. [2] But there is no tool currently available that tightly integrates with this thesis' target language of C#. In spite of these obstacles, there is ample literature on using Haskell with multiple parsers and implementation of the Haskell "map" function to create an Abstract Syntax Tree (AST). [3]

An example parsing Haskell program is provided in Figure 5.4. This algorithm parses a comma-separated values (CSV) file and shows how a functional programs does lexical analysis parsing. This example reveals the complexities of a rather simple parser/lexer. [11] Therefore a solution that would do this would be entirely homegrown which would make this approach very difficult given the limited time constraints.

[2]https://github.com/haskell/haskell-language-server

[3]https://stackoverflow.com/questions/12712149/haskell-parser-to-ast-data-type-assignment



```
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
cell = quotedCell <|> many (noneOf ",\n\r")

quotedCell =
    do char '"'
       content <- many quotedChar
       char '"' <?> "quote at end of cell"
       return content

quotedChar =
        noneOf "\""
    <|> try (string "\"\"" >> return '"')

eol =   try (string "\n\r")
    <|> try (string "\r\n")
    <|> string "\n"
    <|> string "\r"
    <?> "end of line"

parseCSV :: String -> Either ParseError [[String]]
parseCSV input = parse csvFile "(unknown)" input

main =
    do c <- getContents
       case parse csvFile "(stdin)" c of
            Left e -> do putStrLn "Error parsing input:"
                         print e
            Right r -> mapM_ print r
```

An example of Haskell lexical analysis and creating grammar constructions.

Figure 5.4: Example Haskell Parser using Parsec.

5.4 RTCASS

The Re-configurable Transportable Consolidated Automated Support System (RT-CASS) is the Marine Corps specific test set within the CASS standard automatic test equipment family. RTCASS provides intermediate and depot level avionics support for the Marine Corps, Air Force Special Operations Command (AFSOC), Navy depots and foreign allies. The RTCASS provides a portable CASS station configuration using Commercial Off-The-Shelf (COTS) hardware and software to meet USMC V-22 and H-1 support requirements as well as to replace legacy CASS stations at USMC fixed wing aircraft (EA-6B, F/A-18 and AV-8B) support sites. [5] The first release of RTCASS stations became widely available around 2012. While the RTCASS station still has a large footprint (see Figure 5.5), it is much more manageable than traditional CASS stations.

The first deployment of RTCASS stipulated that it had to run on the Windows NT Server 4.0 operating system. There was a requirement in the contract with the Department of Navy that code generated from the RTCASS station had to be backwards compatible. The development team accomplished this through the use of Test Program Markup Language (TPML) that was used to generate C-code on the RTCASS station. The RTCASS compiler reads the ATLAS source input and generates the TPML which is essentially an XML document. This is considered the intermediate language from which executable code is ultimately generated. As a result of this implementation it is possible to generate not only C code, but revert back to ATLAS code. Hence, the TPML could be reverse-engineered back to the original ATLAS source used to generate it.

A study by Cheng-Wei Chen and Jenq Kuen Lee at the Programming Language Lab [4] details a PC-based ATLAS compiler development technology that very closely resembles the approach that the Boeing Corporation followed in developing the RT-CASS compiler. In this study, the authors created a compiler aimed at providing control of PC-based automated test equipment in full compliance with the ATLAS IEEE standard. The compiler was developed in an object-oriented language and incorporates XML representations. The abstract syntax trees are treated as linked objects and linguistic constructs are defined in a class hierarchy. The compiler uses object serialization for storing and retrieving syntax trees and program graphs which the authors state greatly reduced compiler development effort. They transformed EBNF grammars into Yacc-format grammars, class hierarchies of program graphs,

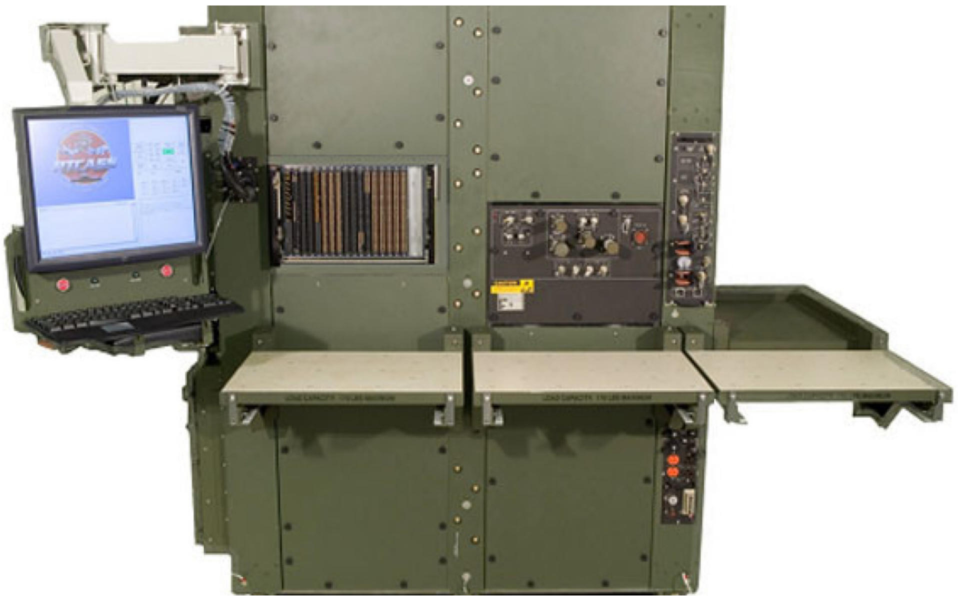


Figure 5.5: RTCASS Station

and object serializations of program graphs. The class definitions in this compiler are for Module, Statement, Node, Symbol, and Type Classes from which the XML representation is generated utilizing object serialization. This process yields the same benefit as the TPML described above. While these classes were designed in C++, the intermediate nature of the object store provides the same flexibility as the TPML intermediate representation.

RTCASS competes with WATLAS since it provides a Windows-based compiler that generates x86 instructions after compilation of the C code generated from the TPML. But, the solution is neither lightweight due to its backwards-compatibility requirement, nor due to its requirements that the generated output executable must run under the slightly modified Windows NT Server and its associated ATLAS executive that runs on top of Windows NT Server. Finally, RTCASS‘s cost to develop was very expensive.

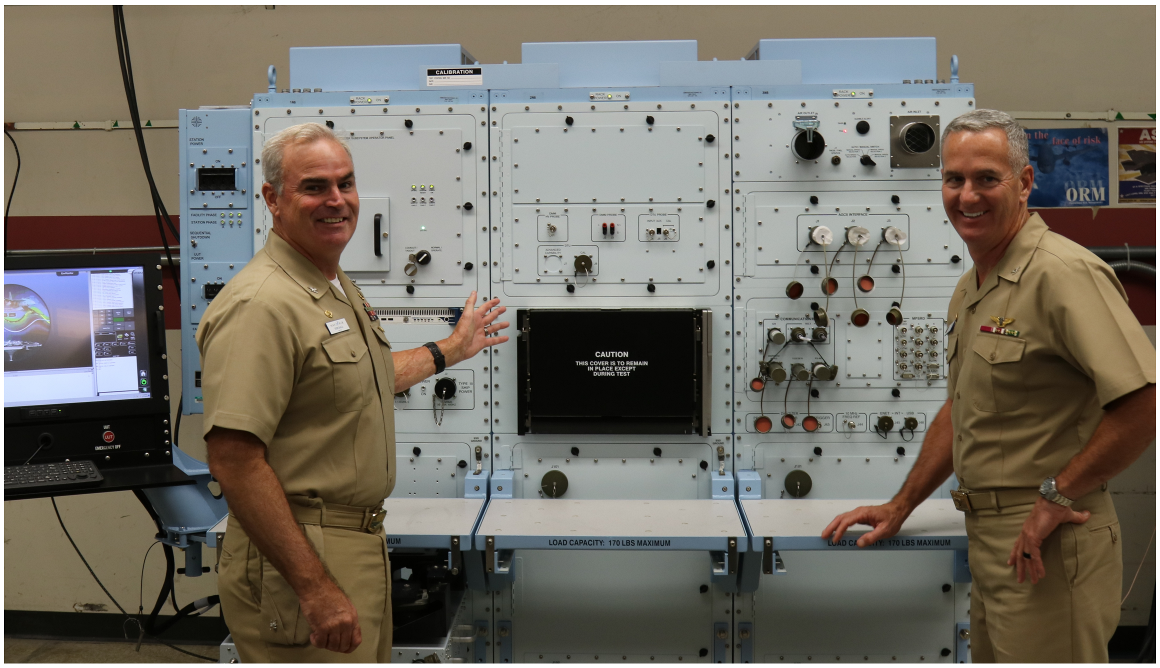

CALIBRATION
CAUTION
THIS COVER IS TO REMAIN IN PLACE EXCEPT DURING TEST
LOAD CAPACITY: 170 LBS MAXIMUM

Figure 5.6: eCASS Station

5.5 ECASS

The electronic Consolidated Automated Support System (eCASS), is the Navy's most recent addition to the CASS automatic test equipment family (see Figure 5.6). eCASS provides shore-based and afloat intermediate and depot level maintenance and repair capabilities for all naval aircraft, ship and submarine electronics systems. eCASS was fielded and operational in 2018 and began replacing the existing aging legacy CASS stations at Naval Air Systems Command and Naval Sea Systems Command activities. [10]

eCASS stations come equipped with a translator that takes ATLAS source code and translates it into an executable form that is compatible with Common Development Environment for TPS's (CDET). Under the covers, a CDET module is a Windows OBJ file much like C or C++ object files. eCASS specifies an entry-point standard to make the module callable from the Standard Test Operations and Run-time Manager (STORM). In this development environment, TPS developers continue to provide ATLAS source code which the CDET translator processes into the CDET executable object library. When the TPS developer executes their run-time module, STORM evaluates the CDET object library, loads the correct module and executes it on the eCASS station. In a brief discussion with an electrical engineer at Cherry Point, he indicated that the eCASS station also provided processing enhancements for hardware assets that eCASS has also incorporated beyond what the legacy CASS stations provided. Hence, eCASS ATLAS provides some additional syntax functionality to accommodate the new capabilities.

5.6 LLVM

LLVM began as a research project at the University of Illinois, with the goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages. Since then, LLVM has grown to be an umbrella project consisting of a number of sub-projects, many of which are being used in production by a wide variety of commercial and open source projects as well as being widely used in academic research. Code in the LLVM project is licensed under the "UIUC" BSD-Style license. [9]

While LLVM is a compiler, it also a toolkit used to develop the other compilers. The authors of LLVM have divided the main components of compiler generation into phases and LLVM is injected into the process after the interim code language is created from lexing, parsing and analysis of the original source code. LLVM is not concerned with the original programming language since it only operates on the interim code referred to as "LLVM bytecode". LLVM has multiple language-specific front-ends that target LLVM bytecode, and then a shared LLVM backend for code analysis and

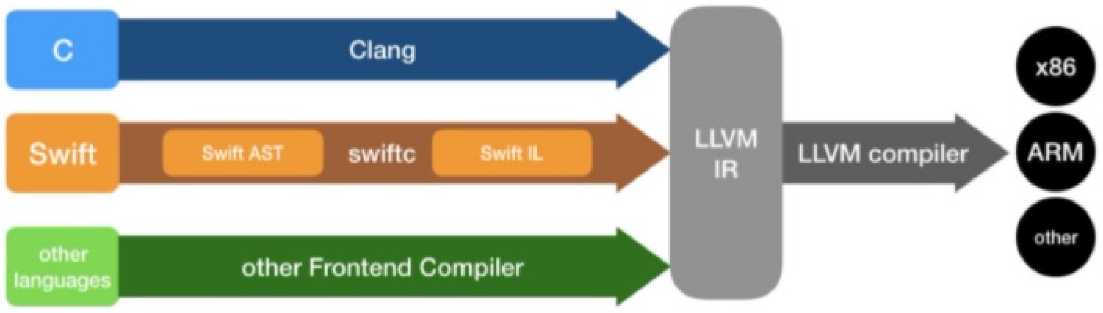


Figure 5.7: LLVM Bird's Eye View.

executable code generation. From that, LLVM optimizes the code and generates the executable file. LLVM is also platform independent. It can take the interim code and make it compliant in an executable format for many target processors as depicted in Figure 5.7. The beauty of this approach is that the compiler developer needs only to perform lexical analysis of the source code, parse it and output the semantically correct interim code and LLVM handles the rest of the hardware details.

LLVM also provides tight integration in a number of different integrated development environments (IDEs). One of the supported IDEs is Visual Studio through installation of the LLVM Toolchain Extension. [4]. With the LLVM Extension, compiler developers can target either 32-bit or 64-bit Windows compilers. In light of all of these capabilities, it is possible to use a tool like Rascal or ANTLR to develop the lexical analyser and parser, and then output the semantically correct interim code from the AST and utilize LLVM to generate the executable.

5.7 Translating JOVIAL

JOVIAL is a MIL-STD legacy language that has been prevalent in military applications and was specifically developed for embedded military avionics systems. The

[4]https://marketplace.visualstudio.com/items?itemName=LLVMExtensions.llvm-toolchain/

JOVIAL Integrated Tool Set (ITS) is a set of software support tools used for development and maintenance of MIL-STD-1750A targeted applications.[5] JOVIAL is also used to support older military vehicles. It was originally developed in 1959 based on ALGOL58 and shares many of the features of legacy languages including uppercase-only tokens and a syntax very similar to block-structured languages like Pascal. It was also among the group of languages that ran on OpenVMS and was licensed by DEC.

The list of military applications that still use JOVIAL-based embedded software is impressive and includes aircraft such as:

1) The B52 Stratofortress which is not slated to go End-of-Service-Life (EOSL) until the 2050's.

2) The C-130 Hercules which is still in production today with no slated EOSL.

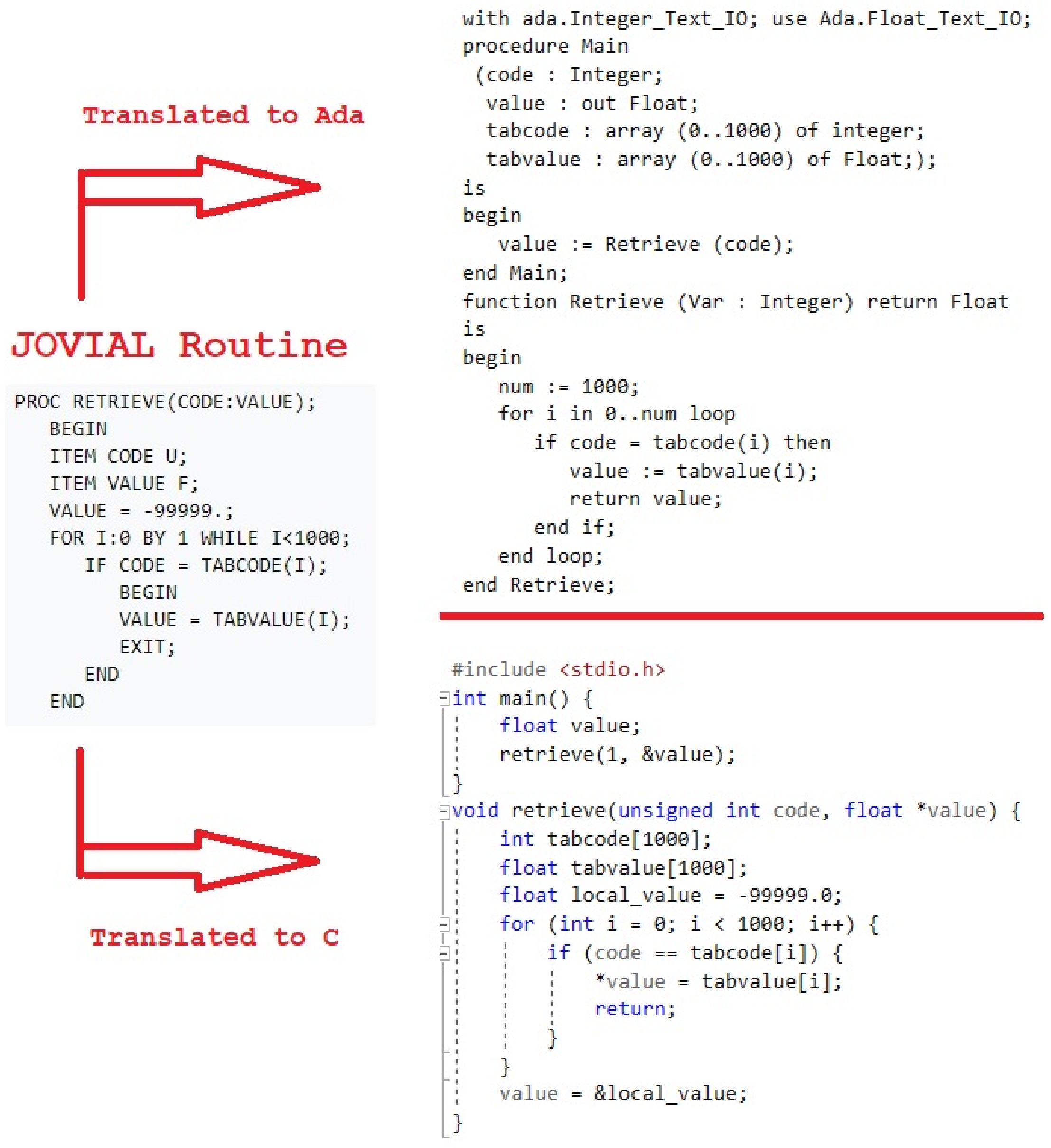
3) The UH-60 Black Hawk helicopter which is also still in production today with no slated EOSL.

In 1983, the U.S. Air Force (USAF) Command at Wright Patterson Air Force Base in Dayton, Ohio began a feasibility study [15] to create a translator to migrate JOVIAL programs to a different language. The goal of the study was to translate JOVIAL to Ada and eventually this led to an effort to translate JOVIAL to the C language.[6] In Figure 5.8, an example translation from JOVIAL to the equivalent functionality in Ada and C shows that JOVIAL is a very concise language and can express functionality in approximately the same number of lines of code as C, but Ada is a bit more verbose. This JOVIAL snippet does not show library includes because none are needed to perform this small unit of functionality. Some other JOVIAL programs found online [7] reveal that the sparse source code example here

[5]https://web.archive.org/web/20090423065636/http://www.jovial.hill.af.mil/

[6]http://www.semdesigns.com/Products/MigrationTools/JOVIAL2C.html

[7]http://bitsavers.trailing-edge.com/pdf/cdc/cyber/lang/jovial/60252100A_JOVIALgenIf_Mar69.pdf



```
PROC RETRIEVE(CODE:VALUE);
   BEGIN
   ITEM CODE U;
   ITEM VALUE F;
   VALUE = -99999.;
   FOR I:0 BY 1 WHILE I<1000;
      IF CODE = TABCODE(I);
         BEGIN
         VALUE = TABVALUE(I);
         EXIT;
      END
   END
```

```
with ada.Integer_Text_IO; use Ada.Float_Text_IO;
procedure Main
 (code : Integer;
  value : out Float;
  tabcode : array (0..1000) of integer;
  tabvalue : array (0..1000) of Float;);
is
begin
   value := Retrieve (code);
end Main;
function Retrieve (Var : Integer) return Float
is
begin
   num := 1000;
   for i in 0..num loop
      if code = tabcode(i) then
         value := tabvalue(i);
         return value;
      end if;
   end loop;
end Retrieve;
```

```
#include <stdio.h>
int main() {
    float value;
    retrieve(1, &value);
}
void retrieve(unsigned int code, float *value) {
    int tabcode[1000];
    float tabvalue[1000];
    float local_value = -99999.0;
    for (int i = 0; i < 1000; i++) {
        if (code == tabcode[i]) {
            *value = tabvalue[i];
            return;
        }
    }
    value = &local_value;
}
```

Figure 5.8: Example JOVIAL to Ada and C Translation

is anomalous with other JOVIAL examples in general. It appears this language is rather elegant, simple, and easy to learn. So, why did the USAF want to replace this language?

Before an attempt to answer that question is provided, a review of the feasibility study is in order. In the feasibility study, the author begins with a section on the critical need for "execution equivalence". JOVIAL is used in mission critical applications, so this concern is paramount. The next concern is for efficiency as it pertains to CPU execution cycles largely due to timing requirements. The resulting machine instructions generated from the translated Ada source code should be the same, smaller, or even more efficient than what was produced from the original JOVIAL source code. Finally, the resultant source code quality should promote maintainability and other non-functional requirements like readability, robust implementation of language features, supportive of debugging automation, reliability, and free of underlying dependencies and ambiguities. The study goes on to state how Ada best meets the requirements as the target language for the translator.

In the next section of the study, some of the major differences between JOVIAL and Ada are introduced. It discusses Compool modules, scope of names, implementation parameters, data declarations by data type including integer, float, fixed, bit, character, enumerations, pointers, tables–that is arrays, constants, blocks–not supported in Ada, and data objects. Further specifications concerning declarations of types, statement names, defines, external references, overlays, procedures and functions were defined along with ramifications and anomalies for each. The author goes on to state how certain procedural statement translations would be trivial (WHILE, FOR, IF, and CASE) while others would require more evaluation (ELSEIF, FALLTHRU, RETURN, GOTO, EXIT, STOP, ABORT and Procedure Call syntax). Finally, the study indicates numeric formulas (a.k.a.; expressions) carry certain nuances depending on the data type, exposes the full list of Ada reserved words that should be excluded, and a brief discussion of variable name conflicts that might occur during translation due to different lexical rules for variable names is provided. My review reveals that the document is quite comprehensive and offers some insights on the path forward. The author was a subject matter expert and wanted to bring this project to completion using Ada.

An effort was made to reach out to the author of this study. Daniel H. Ehrenfried served as an officer [8] in the USAF. And starting in 1982, he was the project engineer, in-house Ada expert, compiler evaluator, and Ada tutorial instructor. This translator was his "project conception" and it appears he shepherded this project along until 1985. In the end, it turned out to be "a $2.5M research effort to investigate the application of Ada to real-time embedded systems and related support tools". Since he has not returned emails, the following is only conjecture. Even in 1983 dollars, $2.5M is a meager price tag and it is doubtful that this project was completed as he envisioned it - an Ada-only shop. According to the study, "with the translation of all J73 software into Ada, J73 software development systems could be phased out of use, the cost of maintaining the J73 system could be recovered, and programmers would be freed earlier for their eventual transition to Ada". This was an ambitious effort and cost concerns may have legitimately been a driving factor, but one does not need to work around the DoD world long to know that cost concerns are usually cover for ulterior motives.

At this point in this research, it is conceivable that there were many pulling on the rope in this technological tug-of-war. The DEC Ada toolbox [9] was developed and in all likelihood, some of the existing JOVIAL-based avionics software may have been re-written in Ada, and some may have been re-written in C. It is also likely that many DoD contractors were involved in the decision-making process. What is known for sure is that "as of 2010, JOVIAL is no longer maintained and distributed by the USAF JOVIAL Program Office (JPO). Software formerly distributed by the

[8]http://www.littletree.com/resume.html
[9]https://sw-eng.falls-church.va.us/ajpo_databases/products_tools2.html

JPO is still available through commercial resources". [10] This does not confirm that JOVIAL is no longer in use. It is just not supported by the JPO. ATLAS still remains in active use today despite limited vendor and Joint Program Office support. This shows that there is no winner-take-all in the software and technology realm of defense acquisitions.

In 2009, USAF at Wright Patterson AFB approved the following for release into the public domain. "As the Air Force's B-2 prime contractor, Northrop Grumman contracted Semantic Designs to construct cost-effective, automated tools and translation methods to modernize the flight management computer system software applications, beginning with the (Operational Flight Program) OFP, by converting the legacy JOVIAL code to C. Northrop Grumman applied its system engineering methodologies to integrate new hardware and used the JOVIAL2C translator to convert the flight software to operate in the new processing environment". [11] So, according to this release, it appears the JOVIAL to Ada program was not successful. Instead, Northrop Grumman was able to persuade the USAF that the Semantic Designs translator was the best solution.

As a final note on language usage, it is worth mentioning the relationship between the languages referenced in this section and the prevalence of their current usage. According to the Statistics and Data Organization's report on the "The Importance of Being Earnest" (TIOBE) Index [12], the C language is the most popular language of all programming language coming in with a 13.38% rating, which is higher than Java, C++ and C#. Ada has a rating of 0.57% while JOVIAL and ATLAS did not even make it onto the list. Interestingly though, Ada is below Assembly Language at 2.43%. This report is as of May of 2021.

[10] https://en.wikipedia.org/wiki/JOVIAL
[11] http://www.semdesigns.com/Products/Services/NorthropGrummanB2.html
[12] https://statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021/

Chapter 6

Evaluation

Here we take a look at some of the results of our effort in developing WATLAS. Specifically we want to address issues that could be improved in our implementation. Further, we will examine some areas of the WATLAS prototype that affect its viability and deserve further investigation.

6.1 Semantics Analysis

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e, that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. [1] Examples relevant to our transpiler would include:

1) Utilizing variables prior to their creation is semantically incorrect. This would mean that the placement of DataSegment.Set or DataSegment.Get statements for any variable can only occur after one, and only one DataSegment.Add statement for that variable name.

2) Implementing procedural statements that operate on assets (like OUTPUT, APPLY, etc.) can only occur after one, and only one REQUIRE statement for that asset.

[1]https://home.adelphi.edu/ siegfried/cs372/372l8.pdf

3) More subtle semantic violations might involve assigning a value of an improper data type in a DataSegment.Set statement, or assuming an improper data type return results from a DataSegment.Get statement.

For this iteration of WATLAS, there was no implementation of any semantics processing to ensure that any of these example semantic errors will not occur. WATLAS is offered as a transpiler for valid ATLAS programs. That is to say that the only ATLAS input source that WATLAS should be used to transpile is source that is known to be semantically correct. The main reason for this is the limited time constraints to develop this prototype. There exist scoping capabilities in Rascal to assist with #3 above, but other semantic violations would need to occur in either the Pre or Post Processor. It is interesting to note that the first several releases of RTCASS (see subsection 5.4) had a similar limitation on its use for translation of existing ATLAS source. Had the inventors of RTCASS included all semantic processing in the first release of their translater, the deliverable would have been pushed considerably behind schedule. So, they suggested that RTCASS only be used to translate "known and working" ATLAS source code.

6.2 Correctness

Correctness from a software engineering perspective can be defined as the adherence to the specifications that determine how users can interact with the software and how the software should behave when it is used correctly. [2] At this time, the only proof of the correctness of the WATLAS transpiler is to observe the ATE Display to see if it behaves the same way as it does when the original program executes on the CASS Station. As with Semantic Analysis, WATLAS does implement any "Correctness" processing features to ensure that the program behaves correctly and

[2]https://www.tutorialspoint.com/software_testing_dictionary/correctness.htm

as stated above, WATLAS is only offered as a transpiler for valid ATLAS programs.

6.3 Viability Concerns

An honest review of the quality of the generated code and the translation design reveals some shortcomings. There is ample room for improvement and it is not realistic to believe this design is optimal. The majority of the viability concerns involve the "DataSegment" framework. First we will look at why the existing design was implemented in this iteration of WATLAS. Then we will explain why this design is deficient.

6.3.1 DataSegment Design

Recall from subsection 4.1.1 that a valid ATLAS variable name might be:

'.SNDSEQ:PHASE-1'

For C#, the lexical rules for a valid variable name are that variable names can contain letters, digits, and the underscore (_) only, must start with a letter, and are case-sensitive. Right away we can see that there are four unique characters in the ATLAS variable name that violate the lexical rules for a variable name in C# and these are highlighted in **red** below.

'.SNDSEQ:PHASE-1'

Since the variable name cannot directly transpiled using characters that violate the lexical rules for a variable name in C#, a scheme that is flexible enough to uniquely represent any ATLAS variable name in C# needs to be devised. In the variable name above, there are multiple characters that are valid for ATLAS variables that are not valid for C#, so a simple replacement could be considered. But, if one thinks that we could just substitute an underscore for an invalid character (i.e.; quote, period, colon, hyphen, etc.), what would happen if we had these three unique variables in ATLAS?

‘.SNDSEQ:PHASE-1‘

‘.SNDSEQ-PHASE-1‘

‘.SNDSEQ.PHASE-1‘

They would all be transpiled as:

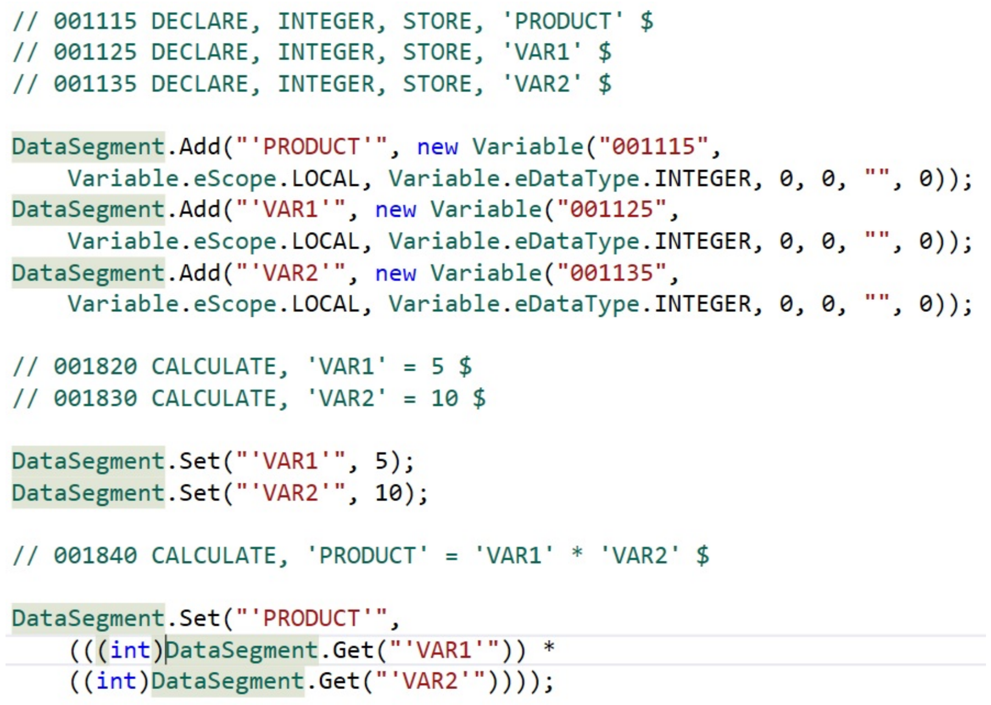
__SNDSEQ_PHASE_1_

Three unique variables would transpile into one variable name, which would not be functionally correct. Therefore, a more robust solution needs to be devised.

For this iteration of WATLAS, the ”DataSegment” class was provided in the WATLAS framework which allows us to preserve the original ATLAS variable name. For the three similarly named variables above, three unique entries will be added to the ”DataSegment”. As a consequence of this solution, the target C# code never references a variable name inline. Rather, it transpiles a variable name into a C# function to represent variables in operations. But this introduces a different problem that threatens the desirability of this approach.

6.3.2 DataSegment Concerns

Consider a simple assignment statement that utilizes an expression involving the three variables PRODUCT, VAR1 and VAR2. An example C# implementation of this is depicted in Figure 6.1. In ATLAS, this would be accomplished through a CALCULATE statement. Suppose the ATLAS source code specifies computing the product of VAR1 and VAR2 and storing the result in the PRODUCT variable. Here is an example that shows the ATLAS code and how it will be represented in WATLAS C#. After the three variables have been added to the ”DataSegment” as a result of encountering the three ATLAS DECLARE statements. the initial values of VAR1 and VAR2 are set to 5 and 10 respectively.

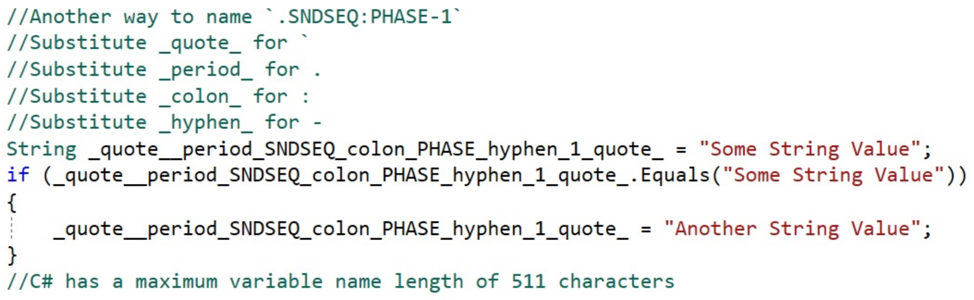


```
// 001115 DECLARE, INTEGER, STORE, 'PRODUCT' $
// 001125 DECLARE, INTEGER, STORE, 'VAR1' $
// 001135 DECLARE, INTEGER, STORE, 'VAR2' $

DataSegment.Add("'PRODUCT'", new Variable("001115",
    Variable.eScope.LOCAL, Variable.eDataType.INTEGER, 0, 0, "", 0));
DataSegment.Add("'VAR1'", new Variable("001125",
    Variable.eScope.LOCAL, Variable.eDataType.INTEGER, 0, 0, "", 0));
DataSegment.Add("'VAR2'", new Variable("001135",
    Variable.eScope.LOCAL, Variable.eDataType.INTEGER, 0, 0, "", 0));

// 001820 CALCULATE, 'VAR1' = 5 $
// 001830 CALCULATE, 'VAR2' = 10 $

DataSegment.Set("'VAR1'", 5);
DataSegment.Set("'VAR2'", 10);

// 001840 CALCULATE, 'PRODUCT' = 'VAR1' * 'VAR2' $

DataSegment.Set("'PRODUCT'",
    (((int)DataSegment.Get("'VAR1'")) *
    ((int)DataSegment.Get("'VAR2'"))));
```

Figure 6.1: Using CALCULATE with variables in expression.

One can easily see that the generated C# code is far more complicated than the ATLAS source code. Even though the code was cut off halfway with a carriage return for legibility in this paper, it is still overly complex given what it is actually doing. So while the ATLAS source code is correctly interpreted into the WATLAS target source code, and is functionally correct and produces the correct results at run time, it is cumbersome in its representation. This obfuscation of the purpose of the instructions lends credence to the feeling that this is a viability concern. Additionally, this implementation requires type casting for the values of variables when they are returned from the ”DataSegment”. There is more discussion on the ramifications of casting and return types in section 6.3.3.

A better solution to this problem might probably involve something akin to name mangling, name decorating, or display name attributes. But the research did not reveal any other more accommodating solution in this regard for the target language



```
//Another way to name `.SNDSEQ:PHASE-1`
//Substitute _quote_ for `
//Substitute _period_ for .
//Substitute _colon_ for :
//Substitute _hyphen_ for -
String _quote__period_SNDSEQ_colon_PHASE_hyphen_1_quote_ = "Some String Value";
if (_quote__period_SNDSEQ_colon_PHASE_hyphen_1_quote_.Equals("Some String Value"))
{
    _quote__period_SNDSEQ_colon_PHASE_hyphen_1_quote_ = "Another String Value";
}
//C# has a maximum variable name length of 511 characters
```

Figure 6.2: Another Naming Convention for ATLAS to C# Variable Naming.

of C#. After some time spent trying to come up with a more viable alternative, two additional approaches came to light. Option one would be variable name encoding seen in Figure 6.2 or a simple dictionary entry as seen in Figure 6.3 for option two. After consideration of option one, a refinement of that option yields what is probably the best implementation.

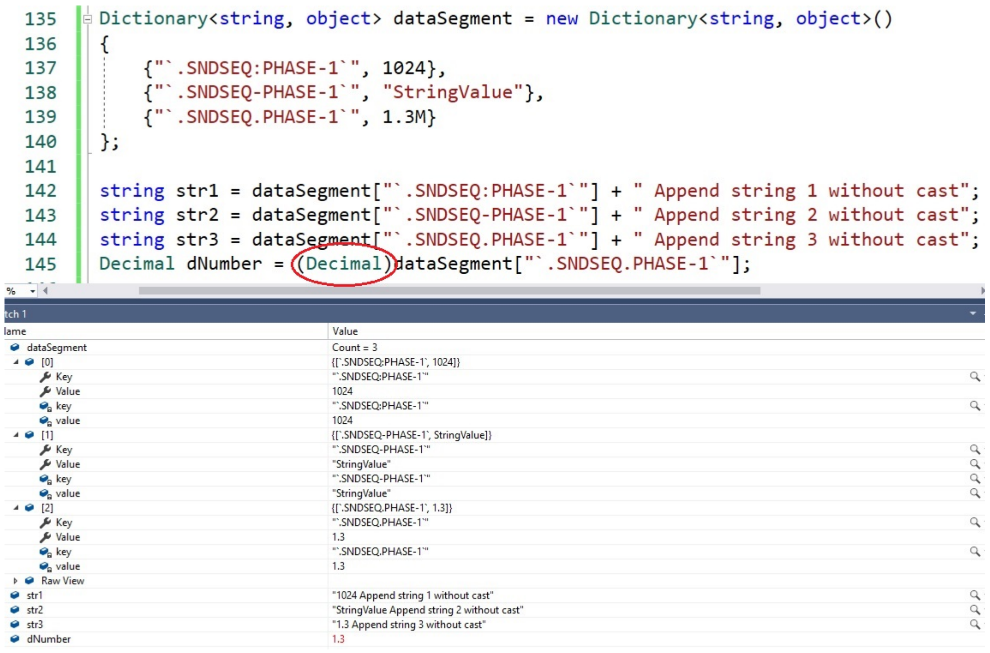
For the variable name encoding (Option 1), a naming rule convention that implements these two rules would work:

1) Use the variable's characters in the ATLAS variable name where they do not violate C# variable lexical rules, and

2) Replace invalid ATLAS variable characters with their type prefixed by an underscore, suffixed by an underscore and displayed as lower case. Since ATLAS variables must use upper case alphanumeric characters, creating a substitute name would not violate C# lexical rules, and it would be feasible to map the C# variable name back to the ATLAS variable name.

The main concern with this option was that the resultant C# variable name might be too long; however, C# allows variable names up to 511 characters in length. As a side note, a test was made with a variable name up to 1024 characters in length, and it compiled and worked. But, this still seems like an ugly implementation.

The simplified Dot NET Framework's Dictionary class (Option 2) is similar to



```
Dictionary<string, object> dataSegment = new Dictionary<string, object>()
{
    {"`.SNDSEQ:PHASE-1`", 1024},
    {"`.SNDSEQ-PHASE-1`", "StringValue"},
    {"`.SNDSEQ.PHASE-1`", 1.3M}
};

string str1 = dataSegment["`.SNDSEQ:PHASE-1`"] + " Append string 1 without cast";
string str2 = dataSegment["`.SNDSEQ-PHASE-1`"] + " Append string 2 without cast";
string str3 = dataSegment["`.SNDSEQ.PHASE-1`"] + " Append string 3 without cast";
Decimal dNumber = (Decimal)dataSegment["`.SNDSEQ.PHASE-1`"];
```

```
tch 1
lame                 Value
dataSegment          Count = 3
  [0]                {[`.SNDSEQ:PHASE-1`, 1024]}
    Key              "`.SNDSEQ:PHASE-1`"
    Value            1024
    key              "`.SNDSEQ:PHASE-1`"
    value            1024
  [1]                {[`.SNDSEQ-PHASE-1`, StringValue]}
    Key              "`.SNDSEQ-PHASE-1`"
    Value            "StringValue"
    key              "`.SNDSEQ-PHASE-1`"
    value            "StringValue"
  [2]                {[`.SNDSEQ.PHASE-1`, 1.3]}
    Key              "`.SNDSEQ.PHASE-1`"
    Value            1.3
    key              "`.SNDSEQ.PHASE-1`"
    value            1.3
  Raw View
str1                 "1024 Append string 1 without cast"
str2                 "StringValue Append string 2 without cast"
str3                 "1.3 Append string 3 without cast"
dNumber              1.3
```

Figure 6.3: Simplified Dictionary-Based Implementation for ATLAS to C# Variable Naming.

the current "DataSegment" implementation in this iteration of WATLAS, but it is a bit more readable. However, like the current "DataSegment" implementation, it requires type casting when retrieving the value from the Dictionary. Unlike the current implementation it does not store additional data originally exposed in the ATLAS DECLARE statement. In the graphic, note that the "Decimal" cast is required. In Chapter 7, we discuss an implementation that has the potential to solve this problem.

Chapter 7

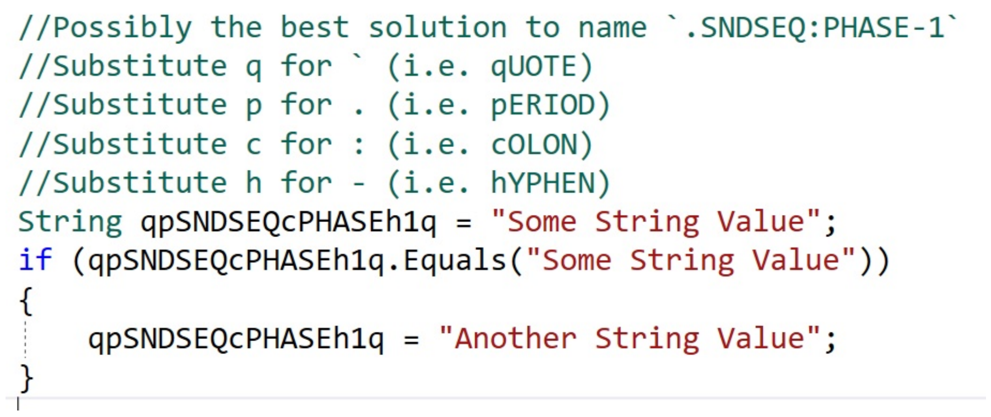
Conclusions and Future Work

There is much work yet to be done to WATLAS, but it has a solid foundation from which to build. This chapter will expose some strategies on how move forward. This is not a comprehensive list, but it is a starting point. If the TPS world began to embrace this product, this list would doubtless grow to a much larger size. In conclusion, thoughts on how well we addressed our research questions will be provided.

7.1 Improving the DataSegment Design

We have delivered a solution that we anticipate will need improving as all software projects do. After describing some possible resolutions to the DataSegment Design, we will now consider a solution than seems to resolve all the issues introduced in its implementation.

A character-for-character substitution for the offending characters with lower-case symbols solves the problem and is guaranteed to not cause name clashes because ATLAS does not support lower-case characters. This is the easiest substitution and results in the most ”readable” C# source code output. It is similar to the approach displayed in Figure 6.2, but it is far more abbreviated. In Figure 7.1, there is a simple character-for-character substitution. This modification was easy enough to implement, that it has been included in the source code generation routines. It solves



```
//Possibly the best solution to name `.SNDSEQ:PHASE-1`
//Substitute q for ` (i.e. qUOTE)
//Substitute p for . (i.e. pERIOD)
//Substitute c for : (i.e. cOLON)
//Substitute h for - (i.e. hYPHEN)
String qpSNDSEQcPHASEh1q = "Some String Value";
if (qpSNDSEQcPHASEh1q.Equals("Some String Value"))
{
    qpSNDSEQcPHASEh1q = "Another String Value";
}
```

Figure 7.1: Fully Abbreviated Variable Name Translation Approach

all three problems; the "DataSegment" complexity goes away, the lengthy C# variable name is shortened to the same length as the ATLAS variable name, and the need to "cast" return types is no longer necessary for simple data retrieval.

7.1.1 Overloading Return Types

In this section, the author is only going to summarize an observation that was made as a result of delving into compiler development. Prior to this thesis, I had wondered why overloaded methods never allow overloading of the return type of the method. A quick search on this revealed that there are some languages that support return type overloading (Swift and Ada). [1] But the majority of object-oriented languages do not provide this capability. In the cited article, a number of reasons are provided. But the development of the transpiler provided the most illumination for me as to why this is the case.

Referring again to Figures 6.1 and 6.3 one can see that casting is required to obtain the value of a variable from the "DataSegment" and the Dictionary. The Rascal code generation routines would not have generated this code in this manner.

[1]https://softwareengineering.stackexchange.com/questions/317082/why-isnt-the-overloading-with-return-types-allowed-at-least-in-usually-used-l

In fact, an issue with a JAR file upgrade in the middle of this project broke the Pre and Post Processor, and for this reason, CALCULATE statements were not included in the the ATLAS sample input. But, had they been, the Post-Processor would have had to insert the cast expression similar to the [AssetType] post-processing that occurs for the OUTPUT statements. And the post-processor would have had the additional responsibility to keep track of the data type so that it could insert the proper cast clause. The reason is because the Rascal code generator does not keep track of the data type, at least not in the current WATLAS implementation. Therefore, unless the ”DataSegment.Get()” supported overloaded return types, then the code generator would have to cast the returned value. Since overloaded return types are not supported in C#, the responsibility would have been either in the code generation routines or the post processor. While that is not impossible to accomplish, it is a heavy lift for this project given the time constraints.

What this means is that overloaded method signatures are relatively easy for a code generator to support because they state the variable type in the declaration. The compiler just needs to ensure that the overloaded signature exists and the decision of which method to call can be made at compile time. But in contrast, for an overloaded return type to be supported, the decision is made at run time as to how and with which specific ”type” the variable's memory location will be populated. The only thing the compiler knows is that it is an object. Only the run-time knows the type of the variable it is returning. For the compiler to support this, the language must have strong static type implementations. At compile time, the return type is ambiguous without strongly typed languages. And while C# is a strongly typed language, it does support overriding through the implementation of casts. Nevertheless, the cast must be valid or a run-time exception will be thrown.

7.1.2 Constraint Solver Solution

After further investigation, it became clear that this is an instance of a known result for prior work on type systems for programming languages. [3] ”In mathematics as in programming, types impose constraints that help to enforce correctness. Typed versions of set theory, just like typed programming languages, impose constraints on object interaction that prevent objects from inconsistent interaction with other objects”. [3] Overloaded return types complicate type checking. Overloading argument types which provide deduction of variable types from method signatures (i.e. their initialization declaration) causes all of the ”type” information to flow up the syntax tree. But allowing ”type” information to travel up and down the syntax tree by allowing a variable’s type to be determined from its usage requires a constraint solver in order to determine the type. Grabmuller has created just such a constraint solver using the Hindley-Milner type system, but his implementation was created for Haskill. [6] The practical implications of this for WATLAS would mean that our transpiler would need to implement type-checking at translation time and cast ”DataSegment.Gets” return values. So, for this reason and other non-functional requirements like readability and maintainability, we believe the new implementation explained in subsection 7.1 is the best path forward to support these requirements.

7.2 Benchmark Analysis of WATLAS

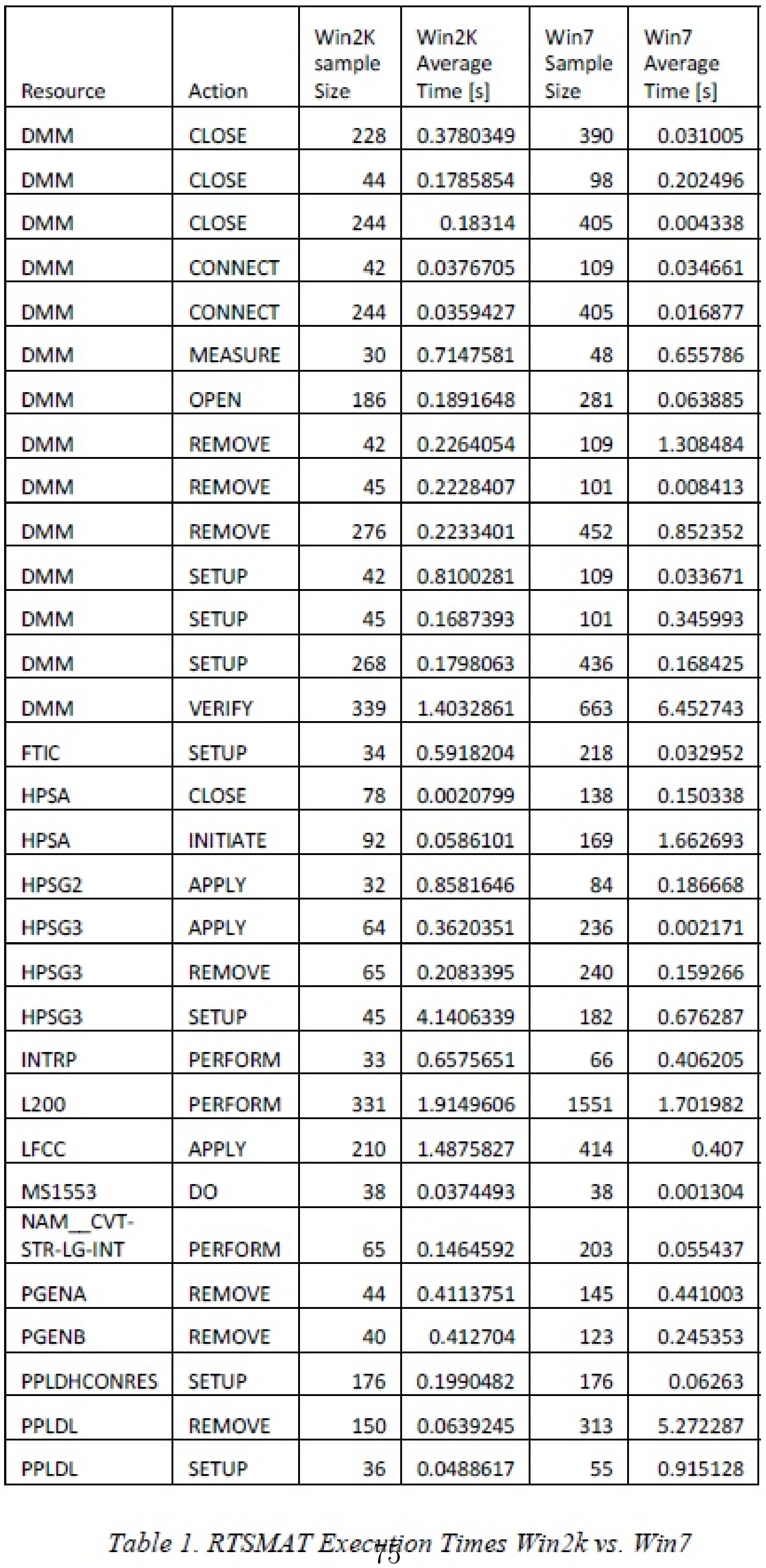
One of the most important things that a business who relies on software development can do is to establish a software performance benchmarking system. This is used to determine how a system performs when tested under a particular workload. [2] Sparr, Fox and Song have noted that ”the use of Commercial-Off-The-Shelf (COTS) operat-

[2]https://www.castsoftware.com/glossary/software-performance-benchmarking-modeling

ing systems in newer generations of Automatic Test Equipment (ATE) has introduced challenges that did not exist with legacy ATE. Unfortunately, COTS instruments and ATE operating systems do not have well documented test sequence execution time." [13] For mission-critical systems like avionics testing systems, the performance of the software is very important. As Colonel Jessup said in "A Few Good Men", "we follow orders, or people die". If avionics systems do not perform as they should, human life can be put in jeopardy.

In the cited work, the authors state that their integration and regression team targeted critical avionics systems, RTCASS Self Maintenance and Test (SMAT), RTCASS Calibration (CAL) software, and a selection of other TPSs representative of test programs that had a high likelihood of being effected by an Engineering Change Proposal (ECP). This kind of analysis is not new in the TPS world, and TPS developers have at their disposal some tools developed in-house to aid their analysis. "Prior regression testing had focused on the measured values and whether the test passed or failed. While this method suffices for end-to-end go chain runs in the case where one hundred percent of the population is tested, it may not expose problems when limited samples are tested and ignores diagnostic testing completely. During the initial integration of RTCASS on Windows 2000, the software developers embedded code in the run-time that logs instrument execution times to a file that is stored on disk and can be accessed at a later time". An example of that run time analysis is provided in Figure 7.2.

WATLAS is designed in such a way that providing this level of support is a trivial matter. Run time logs that write timing events to a disk file are very easy to implement. Due to the fact that WATLAS runs under its own discrete framework, an enhancement to that framework could be quickly implemented that would "log instrument execution times to a file that is stored on disk and can be accessed at a



Resource	Action	Win2K sample Size	Win2K Average Time [s]	Win7 Sample Size	Win7 Average Time [s]
DMM	CLOSE	228	0.3780349	390	0.031005
DMM	CLOSE	44	0.1785854	98	0.202496
DMM	CLOSE	244	0.18314	405	0.004338
DMM	CONNECT	42	0.0376705	109	0.034661
DMM	CONNECT	244	0.0359427	405	0.016877
DMM	MEASURE	30	0.7147581	48	0.655786
DMM	OPEN	186	0.1891648	281	0.063885
DMM	REMOVE	42	0.2264054	109	1.308484
DMM	REMOVE	45	0.2228407	101	0.008413
DMM	REMOVE	276	0.2233401	452	0.852352
DMM	SETUP	42	0.8100281	109	0.033671
DMM	SETUP	45	0.1687393	101	0.345993
DMM	SETUP	268	0.1798063	436	0.168425
DMM	VERIFY	339	1.4032861	663	6.452743
FTIC	SETUP	34	0.5918204	218	0.032952
HPSA	CLOSE	78	0.0020799	138	0.150338
HPSA	INITIATE	92	0.0586101	169	1.662693
HPSG2	APPLY	32	0.8581646	84	0.186668
HPSG3	APPLY	64	0.3620351	236	0.002171
HPSG3	REMOVE	65	0.2083395	240	0.159266
HPSG3	SETUP	45	4.1406339	182	0.676287
INTRP	PERFORM	33	0.6575651	66	0.406205
L200	PERFORM	331	1.9149606	1551	1.701982
LFCC	APPLY	210	1.4875827	414	0.407
MS1553	DO	38	0.0374493	38	0.001304
NAM__CVT-STR-LG-INT	PERFORM	65	0.1464592	203	0.055437
PGENA	REMOVE	44	0.4113751	145	0.441003
PGENB	REMOVE	40	0.412704	123	0.245353
PPLDHCONRES	SETUP	176	0.1990482	176	0.06263
PPLDL	REMOVE	150	0.0639245	313	5.272287
PPLDL	SETUP	36	0.0488617	55	0.915128

Table 1. RTSMAT Execution Times Win2k vs. Win7

Figure 7.2: Run-time Diagnostics Example.

later time". Assembling that logging data in the exact same format would not be difficult. Note how the analysis provides results by hardware asset, by PROCEDURAL statement, and by elapsed time with average results. WATLAS's AssetManager class enforces every asset's adherence to a standard implementation by ensuring all assets inherit from the "Asset" interface. That interface could be enhanced to include a "Log();" method that could "log instrument execution times to a file that is stored on disk and can be accessed at a later time". That logging mechanism could either be called by the TPS developer at their discretion by adding a single line of C# code, or the Rascal routines could guarantee that it is called by implementing that single line of C# code in the Rascal code generation routine. Either way, the time to implement this modification in WATLAS is not significant at all.

7.3 Conclusions

In our introduction in Chapter 1, we asked in RQ1, "what existing work has been done on providing modern support for legacy languages, including ATLAS?" For RQ2, we asked "what are the challenges to providing support for ATLAS on Windows and how well do existing solutions handle these challenges?" Finally, in RQ3, we wanted to know "how well does WATLAS address these challenges?"

To address existing work to provide modern support for legacy languages including ATLAS, we identified significant product offerings from the industrial military complex like Boeing, Lockheed Martin, and Northrop Grumman that provide a robust level of support for ATLAS on Windows, but at a very high cost. There is a private sector offering from Astronics that provides support very similar to WATLAS's CASS Station Emulator. We also showed that there have been previous efforts to migrate legacy languages for military applications that were successful (JOVIAL2C) and some that were not (JOVIAL to Ada).

To uncover the challenges to providing support for ATLAS on Windows, we found that one of the most significant challenges is not technical, but it is a challenge of finding talent willing to work on legacy systems in general. The nature of avionics testing and the requirement to connect multiple automated test equipment assets presents a hardware challenge that make comprehensive testing difficult. ATLAS further complicates translation of the language due to its lexical and syntactical constructions.

We answered the question of how well WATLAS addresses these challenges by demonstrating that WATLAS is a viable alternative worthy of further investigation and development. The choice to utilize Rascal for its parsing and trans-formative capabilities was the correct choice. It provides the level of flexibility in the evaluation of legacy languages that they require while providing the ability to transpile them into constructions that modern day languages share. Finally, the tight integration of the entire WATLAS framework makes these tools easy to use, easy to modify, and ultimately very reasonable to support.

Appendix A

Appendix A - Online Links and Parsing Rules by ATLAS Statement Type

A.1 Links to the Online Repositories

The following tools and repositories are enumerated below for those who might be interested in further research into this subject matter.

1) All of the Rascal files can be found on Github [1] at the link in the footnotes.

2) This thesis used Eclipse version 2020_06 RCP which can be downloaded at the link [2] in the footnotes.

3) This thesis used the latest stable release of Rascal which can be downloaded at the link [3] in the footnotes.

4) This thesis used Microsoft Visual Studio 2017 for all C#-related programming tasks which can be downloaded at the link [4] in the footnotes.

5) The C# framework, Pre and Post Processor, WATLAS project template, and many useful projects that were used to assess the competing technologies can be found in Azure [5] at the link in the footnotes. In addition to the C#-related projects there is a folder at this link where the video presentation has been stored as well as the

[1]https://github.com/ecu-pase-lab/ATLAS
[2]https://www.eclipse.org/downloads/packages/release/2020-06/r/
[3]https://www.rascal-mpl.org/start/
[4]https://my.visualstudio.com/Downloads
[5]https://dev.azure.com/stuartrivenbark/ATLAS_Rascal/

PowerPoint presentation.

A.2 Parsing Rules Examples by ATLAS Statement Type

During the first phase of this thesis in the Fall of 2018, the author compiled a list of parse commands that exemplified all of the statements from an existing ATLAS program. This was performed to test the robustness and utility of the lexical rules and parsing grammar structures that might be encountered in a typical ATLAS program. From this sample, the PrePostProcessor discussed in Chapter 4 could be used to automate translation of an existing ATLAS program without the need for parsing statements by hand. A small sample of each "Statement Type" is included here so that the reader can appreciate some of the effort involved in developing and testing a grammar in Rascal as it pertains to native ATLAS syntax.

```
    import lang::atlas::modules::atlassyntax;
    import ParseTree;
    parse(#BEGIN_Statement, "000000 BEGIN, ATLAS PROGRAM ’18D4305-2’ $");
    parse(#INCLUDE_Statement, "000100 INCLUDE, NON-ATLAS MODULE ’L2_CHAIN’ $");
    parse(#REQUIRE_Statement, "000315 REQUIRE, ’DMM-RES’, SENSOR (RES), IMPEDANCE, CAPABILITY, RES RANGE
0 OHM TO 300 KOHM, CNX HI LO VIA $");
    parse(#DECLARE_Statement, "000600 DECLARE, GLOBAL, BOOLEAN, STORE, ’TEST-FAILED’, ’ASYNCHRONOUS-FLAG’,
’DISCARD-DATA’, ’COMPLETION-FLAG’, ’L200-ALREADY-INITIALIZED’ $");
    parse(#DECLARE_Statement, "000600 DECLARE, GLOBAL, BOOLEAN, STORE, ’TEST-FAILED’, ’ASYNCHRONOUS-FLAG’,
’DISCARD-DATA’, ’COMPLETION-FLAG’, ’L200-ALREADY-INITIALIZED’ $");
    parse(#DEFINE_Statement , "001200 DEFINE, ’CLEAR_SCREEN’, EXTERNAL, PROCEDURE $");
    parse(#END_DEFINE_Statement, "011205 END, ’CLEAR_SCREEN’ $");
    parse(#DEFINE_Section , "001260 DEFINE, ’DISPLAY_GRAPHICS’, EXTERNAL, PROCEDURE (’SID-GRAPH’) $ 001265
DECLARE, MSGCHAR, STORE, ’SID-GRAPH’, 12 CHAR $ 001270 END, ’DISPLAY_GRAPHICS’ $");
    parse(#DEFINE_Section, "001500 DEFINE, ’RUN_COM_FILE_G’, GLOBAL, PROCEDURE (’COM_FILE_NAME’,’ASYNCHRONOUS-FLAG’)
$ 001505 DECLARE, MSGCHAR, STORE, ’COM_FILE_NAME’, 80 CHAR $ 001510 DECLARE, BOOLEAN, STORE, ’ASYNCHRONOUS-FLAG’
$ 001515 PERFORM, ’RUN_COM_FILE’, ’COM_FILE_NAME’, ’ASYNCHRONOUS-FLAG’ $ 001520 END, ’RUN_COM_FILE_G’ $");
    parse(#DEFINE_Section, "002100 DEFINE, ’END_ENTRY_MSG’, GLOBAL, PROCEDURE $ 002105 PERFORM, ’CLEAR_SCREEN’
$ 002110 IF, ’.OVERRIDES’ NE 0, THEN $ 002115 CALCULATE, ’.RETSTAT’ = -1 $ 002215 END, IF $ 002220 PERFORM,
’CLEAR_SCREEN’ $ 002225 END, ’END_ENTRY_MSG’ $");
```

```
    parse(#DEFINE_Section, "011200 DEFINE, 'CLEAR_SCREEN', EXTERNAL, PROCEDURE $ 011202 REMOVE, ALL $ 011204
REMOVE, ALL $ 011205 END, 'CLEAR_SCREEN' $"); parse(#PROCEDURE_Statement, "011005 PERFORM, 'STATNO_TESTNO',
'.STEP', 'EP' $");
    parse(#PROCEDURE_Statement, "011015 CALCULATE, 'REPEAT-ENABLED' = TRUE, 'REPEAT-TEST' = TRUE $");
    parse(#PROCEDURE_Statement, "011025 VERIFY, (RES INTO '.MEASUREMENT'), IMPEDANCE USING 'DMM-RES', NOM
8.56 KOHM UL 8.646 KOHM LL 8.474 KOHM, RES MAX 10 KOHM, CNX HI R1HI LO R1LO $");
    parse(#PROCEDURE_Statement, "023010 REMOVE, DC SIGNAL USING 'DCPSLVA2', VOLTAGE 12.0 V, CURRENT MAX
12.5 MA, CNX HI J1-A59 LO J1-A57 $");
    parse(#PROCEDURE_Statement, "024010 APPLY, DC SIGNAL USING 'DCPSLVA2', VOLTAGE 12.0 V, CURRENT MAX 12.5
MA, CNX HI J1-A1 LO J1-C1 $");
    parse(#PROCEDURE_Statement, "041012 FOR, 'J' = 1 THRU 2, THEN $ 041014 PERFORM, 'CLEAR_SCREEN' $ 041078
END, FOR $");
    parse(#PROCEDURE_Statement, "041018 MONITOR, (VOLTAGE INTO '.MEASUREMENT'), DC SIGNAL USING 'DMM-VDC',
VOLTAGE MAX 34 V, CNX HI PROBE-HI LO PROBE-LO $");
    parse(#PROCEDURE_Statement, "041058 LEAVE, FOR $");
    parse(#PROCEDURE_Statement, "996040 COMPARE, '.MEASUREMENT', GT 0.0 $");
    parse(#ENTRY_Statement, "E049990 'BEGIN TESTING' $");
    parse(#EXIT_Statement, "E 'TERMINATE TPS' $");
    parse(#PROCEDURE_Statement, "050000 OUTPUT, USING 'CRT', DEPOSIT/WINDOW=TE_TXT/CLEAR $");
    parse(#PROCEDURE_Statement, "050005 REMOVE, ALL $");
    parse(#PROCEDURE_Statement, "050010 OUTPUT, USING 'CRT', ((T9,'HELLO WORLD...'))  ((T9,' ')) $");
    parse(#PROCEDURE_Statement, "050015 WAIT FOR, MANUAL INTERVENTION $");
    parse(#PROCEDURE_Statement, "050020 FINISH $");
    parse(#TERMINATE_Statement, "999999 TERMINATE, ATLAS PROGRAM '18D4305-2' $");
```

BIBLIOGRAPHY

[1] Astronics. PAWS Developer's Studio for ATLAS. `https://www.astronics.com/productinfo?productgroup=test%20%26%20measurement&subproduct=Test%20Management%20Software&subitem=PAWS%20Developer%27s%20Studio%20for%20ATLAS`.

[2] Burback, R. L. Digital's Virtual Memory System. `http://infolab.stanford.edu/~burback/watersluice/node97.html`.

[3] Cardelli, L., and Wegner, P. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv. 17*, 4 (Dec. 1985), 471–523.

[4] Chen, C., and Lee, J. Case study: An infrastructure for C/ATLAS environments with object-oriented design and XML representation. `http://pllab.cs.nthu.edu.tw/~cwchen/pub/atlas-jss.pdf`.

[5] Garcia, G., Bravo, B., and Cifredo, J. Automation tools for CASS and RTCASS. In *2008 IEEE AUTOTESTCON* (2008), pp. 7–12.

[6] Grabmuller, M. Algorithm W Step by Step, 2006. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.65.7733`.

[7] HaskellWiki. Haskell, 2013. `https://wiki.haskell.org/index.php?title=Haskell&oldid=56799` [Online; accessed 11-November-2021].

[8] IEEE. IEEE Standard C/ATLAS Test Language. *ANSI/IEEE Std 716-1985* (1985).

[9] LLVM. The LLVM Compiler Infrastructure Project. `http://llvm.org//`.

[10] Office of the Assistant Secretary of Defense for Sustainment. CONSOLIDATED AUTOMATED SUPPORT SYSTEM. `https://www.acq.osd.mil/log/MPP/ats_CASS.html`.

[11] OSullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell.* O'Reilly Media, Inc., 2008.

[12] Parr, T. ANTLR Another Tool for Language Recognition. `https://www.antlr.org/`.

[13] Sparr, C., Fox, R., and Song, Y. Optimizing Regression Testing of Software for the Consolidated Automated Support System. pp. 1–5. `https://www.researchgate.net/publication/328992009_Optimizing_Regression_Testing_of_Software_for_the_Consolidated_Automated_Support_System`.

[14] Tomassetti, G. Getting Started with ANTLR in C Sharp. `https://tomassetti.me/getting-started-with-antlr-in-csharp//`.

[15] USAF. Feasability Assessment of JOVIAL to Ada Translation. Tech. rep., Air Force Wright Aeronautical Labs Wright-Patterson Air Force Base, 1983. `https://apps.dtic.mil/sti/citations/ADA134857`.