THE USE OF COMPUTER LANGUAGE COMPILERS IN LEGACY CODE MIGRATION

Kenneth C. Chisolm
Software Engineering Division (TISAD)
Ogden Air Logistics Center
Hill Air Force Base, Utah 84056
801-775-4445
chisolmk@software.hill.af.mil

James C. Lisonbee
Software Engineering Division (TISAD)
Ogden Air Logistics Center
Hill Air Force Base, Utah 84056
801-775-4445
lisonbei@software.hill.af.mil

Abstract – This paper begins with the premise that computer hardware will be replaced, and this will cause a significant impact on the associated legacy software. The paper will present the advantages of utilizing compiler design techniques in solution of legacy code migration. The primary advantages include lower costs to support a large body of test programs, decreased costs of future system upgrades, and continued support of Test Program Sets (TPS) code in the legacy language. A brief review of compiler design theory will also be presented. This review will concentrate on unique aspects involved in use of ATLAS combined with manufacturers native languages.

INTRODUCTION

Many Automatic Test Equipment (ATE) organizations are beginning to look at replacing aging hardware. The F-16 Analog Test Station Sustainment (FATSS) group found the cost to replace actual hardware was quantifiable and easily justified. Replacement of computer hardware almost always requires new software. This is especially true when computer systems to be replaced are extremely old. All new test equipment reviewed provided a wide range of additional capabilities. The FATSS group reviewed several software tools designed to provide test station interface for the operators, and computer control of instrumentation (operating environment), Figure 1.

None of the operating environment software reviewed supported the particular brand of ATLAS used by older test equipment. Migration to new hardware would result in software impact to more than 220 test programs under configuration management.

Once the FATSS group decided on test station replacement hardware a comprehensive conversion plan was developed. The conversion plan included portions for legacy code migration and hardware driver development. The central requirement of legacy code migration was to identify legacy instrument calls and convert them to function calls in the new test station (public functions).

COMPILER USE

Legacy Code Support

The primary goal of an ATE organization is the accurate testing of customer products. The complete redesign and replacement of a test station is certain to have significant impact on all operational aspects of the station. When the replacement project is complete...
the new equipment must continue to test the customers equipment at least as accurately as before test station replacement. The replacement should introduce zero or a very limited number of new test station anomalies. It is undesirable to produce a new set of known problems in lieu of the old set of such problems.

Test station design teams must seriously consider the impact of test station migration on legacy software. Even when standardized languages such as ANSI C or Kernighan and Ritchie C [3] are used, the software migration problem can be difficult and expensive to solve. When any older non-standard languages are used the difficulties of code migration accelerate rapidly. Anyone who has ever undertaken a software migration project will realize that it is a daunting task to perform migration of more than one or two programs by hand. This leads quickly to the idea of employing some form of automated translation of legacy code.

**Hardware and Software issues**

The FATSS project was able to use Commercial Off The Shelf (COTS) hardware in its new test station. A COTS operating environment was also purchased.

Some operating environments required that TPS be developed locally. They provided no means of TPS development external to the operating environment. This meant that a complete new development project would be necessary for any TPS to be run on the new system. No matter how many advantages these operating environments provided, the number of TPS files that FATSS customers support necessitated that the operating environment software be rejected as a possible solutions.

The operating environment software selected by the FATSS group provided a means for implementation of TPS code in a sub-ANSI C language. The C language compiler along with the operating environment did not fully implement ANSI capabilities. It did provide capabilities to convert from ATLAS to C language code in the new test station. Developers should not rule out use of sub-ANSI compilers, because it is possible to write code to work in a sub-ANSI environment and still conform to ANSI rules. This is accomplished by writing code to ANSI rules and not using features the sub-ANSI compiler will not allow.

**Translate legacy code to a new language.**

At this point our professional software experience, test support requirements, and COTS equipment availability have all combined to indicate the need for an automated means of legacy code migration. Conversion of large amounts of code requires a compiler.

In many cases legacy compilers have long since been lost or the languages they were written in are no longer supported on modern computer systems. The later was the case for the FATSS project. Organizations that have a useful and well-written legacy compiler may be able to modify the back end to produce code for the new test station rather than legacy assembly language. Organizations that no longer have useful compiler code available may be faced with the more difficult task of compiler development.

**Support of TPS programs may continue in legacy language.**

Successful translation of legacy code to a new language results in TPS code existing simultaneously in two different languages. This presents the question of what language will the TPS be maintained in on the new station. Traditional compilers (C to assembly language) maintain source code in the higher level language. The point is to maintain software in a human readable language rather than in a machine oriented assembly language. When compiling from one high level language to another the choice is less obvious. Some organizations may want to set aside older source code and maintain TPS software in the new test station language. Other organizations may desire to implement a gentler transition by continuing to maintain software in the legacy language.

The FATSS group chose to maintain software in the legacy language. Then we were able to view the resultant C language code as an intermediate step to production of executable code. This viewpoint is consistent with traditional compiler concepts where many users never truly realize their C code ever becomes an intermediate assembly language. They usually deal with the source language and the resultant executable file.

Later we may modify the software maintenance process to state the intermediate C language code will become the supported TPS source code. The advantage to this plan is that after the new test station has been in operation for some time it will be clear that changing source language involves a paradigm shift, without any functional impact.
V & V performed only on translation process.

Many organizations are burdened with expensive software acceptance procedures. In a TPS migration project keeping the initial source code provides a means to avoid the expense of individual acceptance tests on each TPS. This was done in the FATSS project by claiming that TPS code would remain unchanged and only the compiler itself required verification and validation (V & V) procedures.

Yes this is a large leap, and requires a lot of explanation. The F-16 analog organization had previously designed and implemented a cross compiler to allow compilation of TPS code on standard personal computers (PCs). The cross compiler acceptance process began with selection of TPS test set files. The test set files were selected to represent the full range of station complexity and hardware usage required by all TPS code under management. The compiled code had to successfully execute the test set programs it compiled, both recognizing good hardware, and isolating the same failures the old compiler could locate.

The cross-compiler project was extremely successful. More than 9 months passed before any anomalies were discovered in the cross-compiler. To date only three or four bugs have been encountered in the cross-compiler. All of the bugs encountered have been easily repaired.

The success of that project encouraged the FATSS group and its customers to employ the same approach to develop a translator for the new test station. It had been proven that the FATSS organization could successfully produce a functional ATLAS compiler that will compile and execute all TPS code the organization is required to support. The previously employed compiler acceptance procedure proved rigorous enough to be reused.

A conservative approach in this process is to retain one or more old test stations until a high degree of confidence is attained with the new hardware and software.

It is important to note that about half the development time for the cross-compiler was spent in reverse engineering the legacy system operation. Production of an effective ATLAS compiler requires an in depth knowledge of legacy software and hardware. It is difficult to believe that anyone can produce a fully functional ATLAS translation without extensive experience on the legacy system.

Cost of 2 man-years for compiler and 1 – 2 years for reverse engineering of old code.

Two legacy code translation options must be considered. The first is to translate all legacy programs by hand. If a limited number of these programs exist, this is not a bad solution. Individuals that have done this report an order of magnitude improvement in the time it takes to translate a TPS after the first few are done. Methodical use of the find and replace features found in text editors is one means of extending the cost effectiveness of hand translation of legacy TPS code.

Organizations with cumbersome software management requirements or with a huge number of TPS files may be required to employ a more automated means of code translation. The FATSS customer had a significant cost to open a TPS plus additional costs for any changes made. Translator engineering costs were repaid by avoiding modification of legacy code. A simple equation for comparing costs to modify TPS software versus paying for translator development is:

\[ n \cdot oh + (t^*max + (n - t^*min) <> oh + comp + rev \]

where:
- rev – reverse engineer current compiler
- comp – compiler development time
- n – number of TPS under management
- oh – TPS modification bookkeeping
- max – max estimated time to convert TPS
- min – minimum time to convert TPS
- t – number of TPS conversions at max time.

If we assume we can convert all TPS files in zero time we see the following:

\[ oh(n - 1) <> comp + rev \]

This tells us compiler design will pay for itself if our compiler design time is less than the overhead to open each TPS.

Increased corporate knowledge of systems under management

Our in house cross-compiler development uncovered numerous problems currently existing in F-16 Analog TPS code. The new compiler produces warnings for each of the conditions described below. Some of these should be error conditions where no executable code
is produced. Since the goal was to reproduce legacy TPS performance on the new station, the cross-compiler reproduced the same bad code. FATSS test station re-host was a more permissive environment, where some TPS impact was allowed. In a few cases the FATSS group did not translate code to match legacy system operation, but forced TPS repair. This is done so that we don't continue to allow poor coding practices that the legacy compiler allowed.

Nuisance warnings

Our new compiler is somewhat more rigorous than the compiler it replaced. It is able to provide the user with warnings for syntax problems not previously noted. Missing commas is one of the more frequently encountered new warnings.

Obvious functional errors

The old compiler often accepted obvious problems without producing errors. If a print statement were missing a terminator the old compiler would simply suck the next line of code into the print statement. The print statement problem was considered prevalent enough to require the new test station to emulate legacy station behavior.

Hidden functional errors

Investigation of bugs encountered during compiler development revealed problems hidden deeply in the legacy compiler. These problems produced bad code without issuing any warnings. One of these cases involved an ATLAS jump statement that explicitly required the jump occurs within the same segment. If the programmer asked for a jump to a new segment no error was issued from the old compiler. The executable code produced by the old compiler forced a similar jump within the same segment, rather than to the segment programmed. This problem did not occur often in the TPS files supported. The new translator executes the jump as requested and warns the programmer that the legacy system did not actually perform the programmed jump.

Suspected functional problems

Often the Honeywell compiler produced executable code that was never entirely trusted. Multiple definitions for variables were allowed. Usually there was default behavior for selection of the variable address used. Frequently 'X' was defined more than once as a variable. These definitions were sometimes obviously for use in separate sections of code where the TPS programmer had cut and pasted code. We simply suspect that frequently the compiler forcing multiple sections of code to use the same address for a variable will cause interference problems. Example: Code fragment 1 writes to 'X' code fragment 2 writes to 'X' then code fragment 1 comes back to use its value of 'X' and gets the value from code fragment 2. This problem was so prevalent the FATSS translator team was required to emulate legacy system behavior. Often 'X' was also declared as both an integer and a real value. The legacy compiler would choose the first occurrence of the declaration to be used throughout the code. This makes the code so that it is questionable whether the correct value is being used due to improper type casting.

COMPILER THEORY REVIEW

Traditional compilers consist of three major components, see Figure 2. The first is a lexical analyzer. The lexical analyzer converts an input file into a string of tokens. The tokens are passed to a parser that recognizes meaningful concepts in a stream of tokens. The final stage of a compiler is the production of code.

Back in the 1970's a pair of tools were developed to facilitate the design of compilers. These tools are known by the names LEX (LEXical analyzer) and YACC (Yet Another Compiler Compiler). LEX and YACC work as a pair to provide the ability to recognize distinct tokens (LEX), then parse them into functional concepts (YACC).

These tools are widely available with Unix and Linux operating systems. They can produce C language compiler code that is portable to standard PC based C language compilers. There are versions of LEX and
YACC available on the web [2] for use on a windows based PC.

The remainder of this paper presents the advantages of using these tools to solve the legacy code migration problem, see figure 3. A related, yet separate, paper describes how to use LEX and YACC to develop an ATLAS compiler [9].

LEX

The first task required in compiler design is breaking an input file into a sequence of tokens. Each token should be classified by type, see Table 1.

<table>
<thead>
<tr>
<th>Token type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1234</td>
</tr>
<tr>
<td>Real</td>
<td>.12E+3.4</td>
</tr>
<tr>
<td>Terminator</td>
<td>$</td>
</tr>
<tr>
<td>Variable</td>
<td>'ABCD'</td>
</tr>
</tbody>
</table>

A compiler design team can build software to recognize tokens. It is a relatively easy task to get a basic token recognition routine running. The designers of LEX created very good public domain tools for token recognition years ago. LEX comes with many features that are extremely useful in the larger realm of compiler design. Some of its major features include context switching, trailing token recognition, echoing of tokens, and file wrapping. I cannot emphasize enough the utility of this tool. A compiler design team that builds its own token recognition software is almost certain to encounter problems later in their project development that have already been solved by LEX.

Identification of many tokens used as language elements in ATLAS.

One of the first types of tokens that occur to be identified is reserved words. Many computer languages have a limited number of reserved word tokens (if, else, do, while,...). Implementations of ATLAS have a very large number of tokens that are elements of the language, see Table 2.

The ATLAS language tokens behave as though they are reserved words. ATLAS reserved words can be grouped into functional categories based on how they are used by the ATLAS language. For example, any time in an ATLAS program the token MEASURE occurs, it can be interpreted as a command to perform a measurement. If we get into language syntax a little later this MEASURE will be required to be followed by a variety of other tokens of various types, nouns, measured characteristics, data types - real or integer, variables, etc. As a general rule any of these ATLAS language elements explicitly specify use of an associated hardware concept. Each type of token can be assigned a token designator.
Table 2: ATLAS reserved words

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verb</td>
<td>APPLY, MEASURE, CALCULATE, PRINT</td>
</tr>
<tr>
<td>Noun</td>
<td>AC SIGNAL, DC SIGNAL, IMPEDANCE,</td>
</tr>
<tr>
<td>Measured Characteristic</td>
<td>(VOLTAGE), (FREQ), (RISE-TIME), (RES)</td>
</tr>
<tr>
<td>Functions</td>
<td>SIN, COS, ABS</td>
</tr>
<tr>
<td>Other Mnemonics</td>
<td>VOLTAGE, FREQ, MAX, UL, LL,</td>
</tr>
<tr>
<td>Dimension</td>
<td>V, KV, OHM, A, MA, UA</td>
</tr>
</tbody>
</table>

FATSS experience indicates that default context should be disabled when other context modes are active.

**LEAVE/RESUME.**

Many implementations of ATLAS allow the programmer to switch out of ATLAS to another language. This clearly calls for use of context switching. The legacy code for the FATSS project has numerous instances of the test station’s native language code embedded within ATLAS. These native language commands were used when ATLAS did not support available hardware, or when the programmer simply did not know how to make ATLAS perform desired functions. The important thing here is that the native language consists of a different set of reserved words, and different variable formats. When LEX encounters a LEAVE ATLAS or RESUME ATLAS token it can switch the ATLAS context recognition on or off.

**Command CNX fields**

In order for a TPS to ever do anything, instruments must be connected to the rest of the test station. This is done with the CNX field in the ATLAS statement. After the CNX token there are a limited number of reserved words allowed. Tokens that are reserved words in the rest of ATLAS may be reused in the definition of a relay. Context switching can be used to prevent recognition of most ATLAS tokens. This is a very important decision to make.

**Other context switching**

No information following an ATLAS MESSAGE token conforms to ATLAS context requirements.

**Longest token recognition**

LEX will always return to YACC the token associated with the longest defined token that matches the input character stream. This prevents the VOLTAGE-PP from being returned as a VOLTAGE token. Another method to avoid this problem is to specify the trailing separator (whitespace or punctuation) that terminates a token.

**LEFT and RIGHT context**

Sometimes token definitions depend on characters to the right of a given token. LEX has built in methods to handle right context sensitivity. It can put trailing characters back into the input stream so they can be reused in recognition of the next token. The only built
in left context sensitivity is recognition of new line to the left of a token. This can be useful in recognition of assembly language labels. Available literature on LEX [1] suggests means to implement left context sensitivity.

**ECHO**

LEX contains features to allow any token identified to be echoed out. ECHO is a #define'd function call string that the compiler developer can overwrite to allow a new definition of the ECHO command. One of the most obvious uses of this is to develop an ECHO that allows programmed control of the output location.

**File Wrap**

LEX contains a function wrap() that is called when an END OF FILE is encountered. Many developers want to rewind the input file to allow multiple passes through the compiler. Usually the first pass is used to locate definitions of variables and functions. The second pass uses those definitions when generating output code.

**Regular Expressions**

Regular expression recognition is a set of rules for token recognition. Similar sets of rules are used by LEX, Microsoft text editor, and Codewrite to recognize generic token strings. One example is the use of "[0-9]+" to recognize an integer. Within brackets 0-9 recognizes any character between ASCII 0 and ASCII 9. The trailing plus indicates one or more of the proceeding characters are required. LEX includes a large body of rules to allow recognition of sophisticated tokens and to reuse these token definitions.

**YACC syntax analysis**

The second task required in compiler design is parsing of tokens. YACC will receive from LEX a string of tokens and group them into intelligible sequences. A math operation is described as a variable = number or variable followed by an infinite number of math fragments and a terminator. A math fragment is defined as a math operator followed by a variable or number. These same type of methods can be used to identify any other line of ATLAS code including instrument control, Input/Output (IO), and sequence control statements.

**Error detection and Recovery**

YACC has built in tools to assist with error recovery. This problem remains one of the toughest in compiler design. In normal operation YACC receives a sequence of tokens and attempts to correlate them with legal command syntax. When a complete legal command is encountered, YACC processes it. Sometimes an impossible command will arrive. If an ATLAS compiler receives a CALCULATE statement the next three tokens must be a label, an equal sign, and a number, in that exact order. Any other token sequence will produce an error. Individuals that have used many compilers know that this is where trouble starts. Telling the programmer what went wrong and then getting back to normal compiling is a very difficult and time consuming task. YACC provides one simple means to apply brute force and ignorance to solve the error recovery problem. It can recognize an error line. An error line can be defined as a sequence of error tokens followed by a terminator token ($ in ATLAS). This will notify the programmer only of the first error in a line of code. The error may propagate as much as another line or two when quote marks, parentheses, or missing terminators are involved. This method will recover from an error and resume normal compiling within one or two lines in the vast majority of cases. Many of us have used compilers that with missing terminators or quotation marks produce errors until the compilation program crashes. This is not a clean method of compiling.

**Code Production**

The final step in compiler design is the actual output of code. Traditional compilers produce atoms. Atoms are thought of as simple processing concepts. A typical atom might add A to B and put the result into C. Traditional compilers will call output routines to turn atoms into assembly language. The use of different sets of atom processors allows production of code for different CPUs. Fortunately we do not need to break our code down so far. We are translating from one high level language to another.

**Instrument Recognition**

An important feature of ATLAS is the ability to define instrument configurations and reuse them. This is done using the DEFINE command. An instrument configuration can be assigned to specific hardware with a RENAME command. The use of multiple DEFINEs with one RENAME statement allows programmers to define separate ammeters and voltmeters for one DMM. The reuse of a DEFINEd label within another
DEFINE statement works as a form of hardware inheritance. It is important to recognize that the bulk of ATLAS software is organized around the use of hardware.

When a design team replaces a test station they may review each instrument, and select replacements as good as or preferably better than the old instrument. The compiler design can also employ the same strategy. A compiler can isolate the required instrument in an ATLAS command. After this is done it can produce a function call to a new instrument driver function.

Often a hardware design team will not have one-to-one instrument replacement. It may combine multiple instruments into a new one or the other way around. In the case of combining multiple instruments into one the compiler may recognize all instruments from the old station and produce a single call to the new instrument. Example calls to power supply A, B, and C may be replaced with a call to power supply 1 at ranges A, B, or C. The problem of splitting an instrument from the old station into two new instrument calls is slightly more difficult, but still very easy. The compiler designers must key on the command differences in the 2 instruments. One example is the replacement of a DMM with a voltmeter and a counter. The compiler can utilize the measured characteristic in addition to the instrument name to determine which hardware to use.

Instrument Processing Details

Once an instrument command is recognized several additional processing elements may be required. Most instruments require relays to be thrown. These relay commands are often embedded within the instrument statement. The compiler can recognize a CNX token and subsequent relay fields, then generate calls to relay drivers. Another important concept is the post processing required by use of VERIFY commands. Appropriate design of the compiler can allow for the structured implementation of these requirements. Each and every instrument can throw relays, setup the instrument, activate the instrument, and perform pre or post processing as required. A single subroutine can be employed to test the verb and determine which instrument activities are required.

Conversion of other lines of code

Most other ATLAS commands have direct conversions to other computer languages. Translation of an ATLAS CALCULATE statement to C requires the elimination of the verb CALCULATE, evaluation field, conversion of ‘$’ to ‘;’ and conversion of function calls to C format. ATLAS and C label definitions are incompatible and must be changed throughout the program.

ATLAS commands like PRINT, DECLARE, and others have similar direct conversion methods. These methods were all easier than those required by CALCULATE.

Organization of compilers with front end and back end processing

Even though our translator will not produce assembly language code we still want to separate the output requirements from language parsing. This is so we can easily port code to new environments. The recognition and parsing of our source code will remain the same forever. It is impossible to believe that anyone will go back and change the ATLAS language specification and retroactively modify the legacy code we are supporting. One of the ideas behind re-hosting the test station to a VXI based station is to allow gradual upgrade of hardware. At some time in the future we may want to modify our instrument function calls to conform to National Instrument’s IVI or Hewlett Packard (HP) instrument formats. A compiler might use separate back ends for HP vs. IVI instrument calling formats. Another option is that mismatched replacement of hardware can be handled purely by modification of the compiler back end. Code can be produced in a completely new language by modification of the back end of the compiler, if the front end is well written. At a later date a back end can be designed for production Object Oriented (OO) languages, assembly languages, other flavors of C etc.

REFERENCES


