Introducing Legacy Program Scripting to Molecular Biology Toolkit (MBT)

Graduate Project Report for MS Computer Science Degree

Committee:

_____

Professor Joe Geigel, Chair Date

_____

Professor Paul Craig, Reader Date

_____

Professor Hans-Peter Bischof, Observer Date

Todd A. Newell

October 28, 2005

Rochester Institute of Technology

Abstract:

Successful navigating of the ever-changing landscape of molecular visualization programs requires a common thread that can offer users an oasis from the maelstrom of new languages, proprietary applications, and miniscule software life-cycles. Cross-application scripting remains the benchmark, allowing scientists and researchers to speak a common tongue.

The introduction of a new, powerful visualization language, Molecular Biology Toolkit (MBT), has tempted many users to abandon previous methodologies and adopt a new mode of research. MBT, however, is not without drawbacks. Its lack of scripting capabilities creates unmanageable complexity for unsophisticated end-users, namely those without the ability to program. MBT, thus, lacks the basic handholds for its widespread acceptance in the molecular visualization community. As a toolkit package without its own mode of execution, its design challenges users to develop their own customized features and applications. However, able to contribute as a text based virtual molecular collection or a fully rendered 3D molecular representation, MBT has the tools researchers want in a new visualization program.

Using JavaCC to parse legacy commands and in turn executing MBT methods all from a single, simple command, I have reintroduced scripting to the modern molecular visualization landscape. Combining these two programs, this project takes steps to encourage the exciting molecular manipulations capable in MBT while bridging to a friendly, user-centric scripting patterns required by end-users not entrenched in software development.

Table of Contents

1. Introduction

1.1 The Task at Hand:

The proliferation of custom designed applications, targeting specific parts of the educational and commercial sectors, has led to a constant array of new molecular visualization programs dotting the collective landscape. Some of these new programs are unsupported, *i.e.*, it is outside of the typical software lifecycle; their authors have left little time for software maintenance and upgrades, or they remain so proprietary that they classify themselves as unique entities in use and appearance, devoid of mainstream support. Although less useful than they may have been, many of these applications contribute a partial solution to an existing challenge. It is important, therefore, for an educator (or commercial scientist) to be able to smoothly transition from one application to another, and adopt programs that better address his or her needs. However, with that move, the end-user shouldn't lose his or her feeling of support they had from their legacy applications as they move toward something completely new. It is this dichotomy that we sought to at least offer a partial (*i.e.*, specific to the world of molecular visualization programs) solution.

In this paper, and the project it describes, I looked at the task of creating a scripting language that would allow a bridge between two commonly used molecular biological visualization applications, Chime and RasMol, as well as a newly developed, free, open source, visualization toolkit, MBT. This application was built as an experiment in scripting languages, in conjunction with the work being done at Rochester Institute of Technology's Chemistry Department, and with the non-computer scientist educator in mind as the eventual end-user. The scripted commands found in these older applications will be parsed, run, and used to control a new application as an alternative to classic methods of execution.

1.2 Chime and RasMol:

Molecular visualization is the process of rendering a representation of a real-life chemical molecule from a collection of inputted data. Depending on the tools used to accomplish this, the result can be a very basic wireframe outline of a simple molecule or a detailed, fully labeled rendered image of a complex multi-level structure. While many languages exist to try to fill a similar need, we chose to focus our research solely on two. Each offered distinct tools and challenges that my committee and presented the experiment with a wide range of operating parameters. Both languages are discussed in greater detail and the reasons behind their choice are made clearer.

The first of the two applications is a product from MDL (http://www.mdl.com/) known as Chime. To begin characterizing Chime's strengths and weaknesses we first asked ourselves what we'd need in an application of this type. We knew that a program being used for research in a scholastic environment had to be available to the educators at

little or no cost. It should also be open source so it can be easily modified to encompass nonstandard or unique applications. Chime meets neither of these requirements.

MDL does distribute a free version of Chime on its website, but it is not a full version of the product and has many of its key features hamstrung. While molecules can still be rendered in the free version, functions such as labeling and rotation are not offered. This leaves the user with less than complete information and an inaccurate picture of the subject of study. Secondly, Chime is a browser plug-in that isn't distributed as open source. Modifications to the application's functions are nearly impossible, leaving the end-user at the mercy of the behaviors predetermined important by MDL's original development team. Because every research lab focuses on a unique point of study, this inflexibility is a major omission. (1)

This is not to say, however, that Chime does not have any features we sought in our sample language. It is fully supported by MDL. As a commercial product, it must to improve over time through updates and revisions, or it will be left in the proverbial dust by those applications that do evolve. This is a vital function that we chose to include in the study, recognizing the need for our experiment to be accessible to future users that might choose to continue our work. Lastly, Chime is powerful and well respected in the scientific community as a leader in visualization. This well deserved reputation has led to countless complex, large, and very detailed scripts that have been written to harness its power. While the fields of chemistry and biology push their boundaries, Chime has been able to keep up.

The second program considered contained features lacking in Chime. RasMol (http://www.umass.edu/microbio/rasmol/index2.htm) is a language developed to give the end-user greater control and more options. It was created with the intent of incorporating many of Chime's commands into a freely distributed rendering engine. Users of RasMol can use the distributed source as a base on which they can add their own functionality. RasMol is also old (created in 1995) and is no longer adequately maintained or supported. This was an important fact when considering the selection of RasMol as a supporting language. However, RasMol's former popularity, and the fact that its command set is encased as a subset of Chime's commands, has led to the creation of many of its own scripts.

1.3 What is MBT (Molecular Biology Toolkit):

This leads us into our target language. Recently developed by a team from the San Diego Supercomputer Center (SDSC), MBT (http://mbt.sdsc.edu/) is a toolkit package that exhibits many of the attributes we required in a visualization program. Even more appealing, it also offered us a tempting opportunity to fill in a gap in its otherwise strong foundation. That gap derives from one function that both Chime and RasMol offer users but that MBT does not – the ability to script. Scripts are commands, either inputted from standard-in or from a separate file fed to the application, which can be easily used to control a program without directly interacting with any of its methods or variables.

Before beginning to examine how scripting could improve MBT, it is important to understand the package itself. MBT is a library of files which can be interfaced in a simple manner. Interaction leads to the eventual creation of customized, application specific programs which will be explored further in the coming sections. However, regardless of how MBT will be used, all work begins with the creation of a target input molecule.

MBT is similar to Chime and RasMol in the types of files it will accept (we used .pdb files exclusively). The input files contain all the chemical data about a molecule, including the name of the atom, the name of the residue in which it resides, and the coordinates the atom occupies. As illustrated by Table 1, .pdb files are filled with densely packed columns of data.

```
ATOM     17  P     C D   2      21.985  70.158  32.911  1.00 35.04      1D66 104
ATOM     18  O1P   C D   2      21.747  69.576  34.250  1.00 39.32      1D66 105
ATOM     19  O2P   C D   2      20.900  70.926  32.235  1.00 33.05      1D66 106
ATOM     20  O5*   C D   2      22.543  68.998  32.013  1.00 36.35      1D66 107
ATOM     21  C5*   C D   2      22.845  69.146  30.643  1.00 35.49      1D66 108
ATOM     22  C4*   C D   2      23.909  68.156  30.307  1.00 34.94      1D66 109
ATOM     23  O4*   C D   2      25.122  68.504  30.990  1.00 33.48      1D66 110
ATOM     24  C3*   C D   2      23.549  66.709  30.739  1.00 33.71      1D66 111
ATOM     25  O3*   C D   2      23.626  65.926  29.562  1.00 41.13      1D66 112
ATOM     26  C2*   C D   2      24.685  66.394  31.674  1.00 33.19      1D66 113
ATOM     27  C1*   C D   2      25.831  67.264  31.170  1.00 27.09      1D66 114
ATOM     28  N1    C D   2      26.992  67.379  32.129  1.00 22.75      1D66 115
ATOM     29  C2    C D   2      28.267  67.635  31.609  1.00 26.68      1D66 116
ATOM     30  O2    C D   2      28.469  67.860  30.410  1.00 31.28      1D66 117
```

Table 1: Sample .pdb file

Once created, these .pdb files conform to strict standards and exist with little or no modification, guaranteeing to users their accuracy and uniformity. For example, if a user decided to download a representation of the protein insulin, the user could find the 9INS.pdb on one of countless biological databases (*e.g.*, http://www.rcsb.org/pdb) and be assured he is operating on the same insulin protein as every other researcher in the world. This file is what the visualization software reads and interprets to form the initial view of the molecule.

It is important to note that MBT, unlike Chime and RasMol, cannot render an input object for viewing without an external application. This is true of all the functions of MBT – a user must supply an application which will run MBT. This is because MBT is a collection of Java classes arranged in a logical hierarchy that allows the user to develop customized applications as the need arises. In this way it is a two-step process. A user identifies how MBT can be used to alter a target molecule, and then develops an application tailored to make those changes. These applications allow the user to control a myriad of different functions within MBT: from rendering the object to traversing through the molecule, making specific changes to target atoms or bonds. MBT does not have a common interface, changing its appearance and function from one user to another.

Although MBT is built around such a deep collection of classes, only a portion of them are used to cause direct changes to the molecule. Figure 1 outlines one Java class in particular, StructureStyles.java, which organizes the discrete components within a

molecule and provides the methods to alter them. StructureStyles.java was used to make most of the changes coming from command scripts.

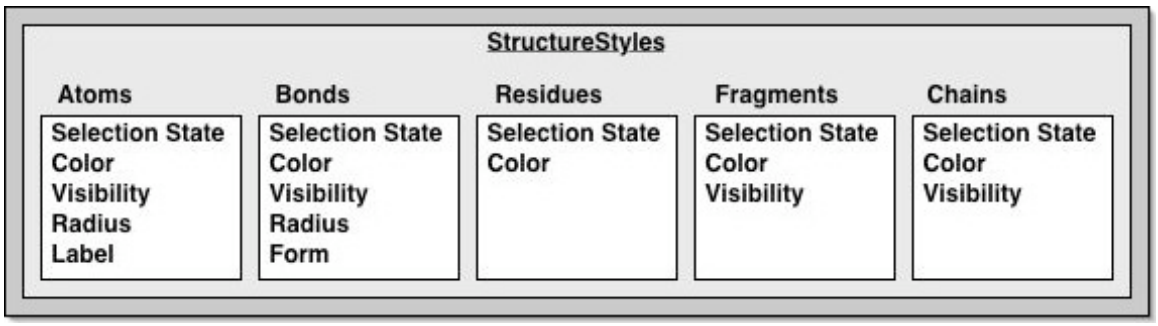

Figure 1: Component Variables

This class is discussed in greater length in the coming sections however, it was important to note that I was moving from an application that was completely self-contained to one that was much more open with its capabilities. This acted as a bookend for the project, which gave me the chance to reproduce some of the features considered vital in previous programs in an environment that afforded greater freedom at the molecular level.

1.4 Goals:

Professor Paul Craig (RIT, Professor of Chemistry) had done previous work with Chime and RasMol, and at the beginning of this project he wanted to use these programs as our base. In addition, his relationship with the SDSC led me to MBT. His work at RIT included significant time using scripts to change input molecules, and he challenged us to move that capability to MBT. While MBT proved to be an exciting framework from which to start, a goal of the project was to minimize the learning curve for its users. I decided to merge the scripting functions of two popular applications to a newer visualization program while keeping improvements simple for the end-user. This idea became the eventual focal point of this project.

After examining how best to combine the important characteristics into one discrete exercise, I decided to merge the existing Chime and RasMol scripting ability into the power of MBT. As a means to limit the project's scope, I decided to include only a handful of the most common commands. The next step was to identify and introduce a parsing language that would evaluate application specific input streams and break them into command-sized bits. The language chosen as my parser was JavaCC. As a free and fully supported language, JavaCC has the required characteristics. It also has the responsibility to check the commands for accuracy, by evaluating them against a supplied grammar. The architecture of JavaCC, and how we used it in this project is discussed further in the coming sections.

At this point all the elements the project required were assembled: a parser, which was easy to use, and an eventual target program which could be improved through modification. The final goal was then solidified. This project was designed to create a parsing language that would accept scripting commands from two ancestor programs, checking them for accuracy, and use the commands to modify the chemical components in our alters molecular visualization toolkit.

2. Overview

2.1 Scripting:

We modeled our work on end-user interactions similar to the work being done by Professor Craig. As a chemist, he was interested in how MBT could be made to work for him as a supplemental tool but not at the cost of learning a new programming language. MBT would remain a development tool and not a research tool if it weren't for our addition of scripting.

When a program is executed, the code that it contains is considered by a compiler, checked for correctness, and then converted into an executable. However, once in executable form there is no way to change its variables or options. A user would have to change the source code and recompile. Scripting is this portal. Similarly, graphical programs such as web viewers and games also grant the user the ability to make changes after the program is compiled. However these actions are done through menu bars or by constantly checking to see if a condition has been changed. They are all done at the discretion of the program the user is running. Instead, in our case, scripts allow the user to make changes without the knowledge of the target application. A script program, such as the one we created, operates outside the scope the primary application.

As mentioned, this is not a concept new to users of previous molecular visualization programs. Chime and RasMol users have always included scripting in there toolset. Either from standard-in, or inputted from a file, scripting commands which were designed to be logical and easy to remember and control the functions of the language under them. The designers of these languages understood who their target users were and they made an effort to make using their syntax as straightforward as possible.

As the following example will illustrate how scripts can change awkward method calls into simple, everyday language.

Script command to select all components in a molecule: "select all"

MBT method to select all components:

```
public void selectAll() {
        selection.clearAll( );
        setAtomSelection( selection.getMin(), selection.getMax(), true );
```

```
bondSelection.clearAll( );
setBondSelection( bondSelection.getMin(), bondSelection.getMax(), true ); }
```

In this way, the designers of Chime and RasMol have hidden the commands that their programs uses behind easier to use, simple scripts. It was also something MBT was lacking (http://mbt.sdsc.edu/apps/index.html under the Scripting header). Whether a conscious omission or not, we decided to fill a void in the MBT libraries.

Adding a scripting capability to MBT that recognizes Chime and RasMol commands has one other advantage. That is, previous users of Chime and RasMol have had scripts for those languages sitting on their workstations for years. They are confident in the integrity of these scripts. Adding a portal to JavaCC through which a user can execute their scripts and affect the molecule, controlled not by Chime or RasMol but instead by MBT, would make the visualization program transparent to the user. They wouldn't miss a step when transitioning from old to new.

By preserving the ability for the user to reuse these scripts we created a twofold solution: firstly, we decided a user may not have the background to use a comprehensive programming library like MBT to directly develop applications. This was the first stage of the project. Commands that were used to run Chime and RasMol would be made to execute MBT methods and affect the molecule in similar ways. Users could reuse bits of command sequences that they were familiar and comfortable with. I tried to hide any interaction with MBT, making the transition from script command to JavaCC and onto MBT as transparent as possible. Ideally, using the same .pdb file and script commands would set-up up a molecule identically, independent of the application used, normalizing it for the user. Secondly, and as importantly, adding the ability to script to MBT gives the user a simpler method of interaction with the program once the initial application had been compiled and executed.

2.2 The End-user and Program Use:

As mentioned above, MBT is a library and not a free-standing application. It can be used to develop any solution that a user might need, but it is not ready to go until additional work is added (such as the work we are contributing). Intuitive, the hierarchy the libraries make up are easy to start working in. This ides will be examined in greater detail in the Architecture section. For now I wanted to explore why MBT is a logical choice to develop in and its advantage to the user.

Because of the flexibility to be tailored to fit any specific need, MBT affords users various methods of use. Molecular visualization applications, by definition, are designed to give the user the best visual representation possible of an input compound. Zoom functions, rotation, and atom-level color changes are some of the features that are standard across the many applications available. MBT has all these abilities as well, and many more. All these are designed to let the user 'touch' the molecule as needed for proper study.

Because of MBT's composition of classes, it can allow the user the ability to work with a molecule that hasn't been rendered by any third-party application. MBT has a very capable rendering and visualization engine; however the way the package is designed, the visualization elements are discrete from the classes that operate the chemistry. This way, a molecule can be loaded into a text based study aid or tested for correctness before rendered into an entire compound. Each part of the overall molecule is kept in a collection, controlled by another class in the hierarchy. At any time an application may reference the components without needing to render the molecule itself. This means that test files can be created to study the components of the molecule or a user can work without the overhead of a rendering engine and still get meaningful data. MBT creates a virtual molecule solely from the input .pdb file. What it does from there is up to the user.

This capability extends to scripting as well. The rendered molecule is a helpful tool, and is the eventual product of the user's studies but does not need to be drawn in order for the user to control it through scripts. Once it is loaded, the molecule exists and can be affected. The idea about the importance of scripts bridging JavaCC and MBT still apply. A user can make a change to the molecule with a single command string from standard-in or a file and be confident that when that molecule is then drawn the changes with take effect.

To match MBT's flexibility in how it can accept input and product output, we wanted to make sure that a user would not be locked in to using our scripting language if it no longer met their needs. The scripting language that we created can be improved or augmented as needed. Instead of requiring a background in programming languages, creating new scripts for MBT needs only an understanding of how the libraries combine to control components. In order to guarantee relevance to future users, the MBT team has named its methods logically and the package API lets users quickly step through the commands needed to make a change. Creating the script is then an exercise of editing the ChimeParser.jj file to make these calls.

A user deciding to fit MBT into his or her studies can have the flexibility to use that tool exactly as needed. Having the choice to render molecules at the time a change to a molecule is made, or diverge from the supplied scripting functions are powerful abilities. Our project lets these abilities work for the end-user.

3. Architecture

In the following section, I will go deeper into the reasons and the criteria followed throughout the project in an effort to highlight many of the design choices.

3.1 JavaCC:

I will start by discussing parsing and the tool picked to do that job. JavaCC, or Java Compiler Compiler, is a freely distributed development kit primarily designed to aid in the creation of programming languages by validating their complex grammars. While

it excels in this role, it may also be used with any rule-based grammar. Our project parser was named ChimeParser.jj and will be referenced directly to illustrate specific choices.

3.1.1 Lexical Analyzer and Parser

There are two distinct parts of JavaCC, each used in order when breaking down an input stream. The first is the lexical analyzer, or token manager, which is used to separate any input command into its individual components. Characters, white-space, even user defined keys words and regular expressions are all broken down according to the grammar rules the programmer gave to JavaCC into unique objects called tokens. Figure 2 below demonstrates a partial list of the tokens acceptable in the Java 1.1 language. The keywords are defined as tokens and are therefore able to travel through the token manager intact.

```
        TOKEN : {
   < ABSTRACT: "abstract" >
   | < BOOLEAN: "boolean" >
     | < BREAK: "break" >
      | < BYTE: "byte" >
      | < CASE: "case" >
     | < CATCH: "catch" >
      | < CHAR: "char" >
     | < CLASS: "class" >
     | < CONST: "const" >
  | < CONTINUE: "continue" >
   | < _DEFAULT: "default" >
        | < DO: "do" >
    | < DOUBLE: "double" >
      | < ELSE: "else" >
   | < EXTENDS: "extends" >
     | < FALSE: "false" >
     | < FINAL: "final" >
   | < FINALLY: "finally" >
     | < FLOAT: "float" >
       | < FOR: "for" >
      | < GOTO: "goto" >
        | < IF: "if" >
| < IMPLEMENTS: "implements" >
    | < IMPORT: "import" >
| < INSTANCEOF: "instanceof" >
       | < INT: "int" >
 | < INTERFACE: "interface" >
      | < LONG: "long" >
    | < NATIVE: "native" >
       | < NEW: "new" >
      | < NULL: "null" >
   | < PACKAGE: "package">
  | < PRIVATE: "private" >
 | < PROTECTED: "protected" >
    | < PUBLIC: "public" >
    | < RETURN: "return" >
     | < SHORT: "short" >
```

```
      | < STATIC: "static" >
       | < SUPER: "super" >
      | < SWITCH: "switch" >
| < SYNCHRONIZED: "synchronized" >
       | < THIS: "this" >
       | < THROW: "throw" >
      | < THROWS: "throws" >
   | < TRANSIENT: "transient" >
       | < TRUE: "true" >
        | < TRY: "try" >
       | < VOID: "void" >
    | < VOLATILE: "volatile" >
      | < WHILE: "while" > }
```

Figure 2: Partial Java 1.1 Token List

Secondly, the tokens are then fed back into JavaCC to be analyzed by the parser function. It is here that they are checked against user defined functions and validated for correctness. Once a token has successfully made its way through the lexical analyzer and eventually a function, the program can be tasked to anything the programmer would like with that token.

Because tokens act as reserved words, they are subsequently disallowed to be used anywhere in the program without explicitly referencing them. Input strings that are not predefined are also tokens, as long as they fit some defined rule. This is an important concept and the way JavaCC handles variables. In me program regular expressions were defined to accept strings of characters and digits in patterns which were decided to fit potential input. In this way all the possible ways a scripting command from Chime or RasMol might be introduced into the parser were accounted for.

3.1.2 JavaCC Execution and Errors

The defined grammar and the creation of our JavaCC program were the primary focus of the project and the portion to which I devoted most of the time. As stated, the grammar was built to mirror the possible forms of the input commands from Chime and RasMol. If a command was correct in a Chime script it had better be allowed through our parser.

How the program responds to each discrete token is the only indication whether that string is part of the grammar. In the event of a token correctly making its way through a function within the parser, it is possible that the user won't have any feedback. Without additional Java code to take action once it is reached, a correct token will do nothing. It is only when a token is incorrect is the user given any feedback. A token is only considered incorrect when every possible path has been exhausted and the token is still unable to be matched to a rule. As Figure 3 shows, the errors generated from incorrect tokens are similar to errors from a classic Java stack trace. However JavaCC error messages are primarily caused when a rule was expecting a defined token of one type and was fed another. They can be modified to be more user friendly just like Java as well.

```
Lexical error at line 1, column 2.  Encountered: "x"
TokenMgrError: Lexical error at line 1, column 2.  Encountered: "x"
(120), after : ""
                at Simple1TokenManager.getNextToken
                       (Simple1TokenManager.java:146)
                at Simple1.getToken(Simple1.java:140)
                at Simple1.MatchedBraces(Simple1.java:51)
                at Simple1.Input(Simple1.java:10)
                at Simple1.main(Simple1.java:6)
```

Figure 3: JavaCC Parsing Error

3.1.3 Order of Operation

The order of the commands might matter so I had to make sure to maintain that order while parsing and executing. For example:

Input script: select protein
color red
select protein and helix

This basic script can be translated as the end user wanting to select all the atoms belonging to a protein ("select protein"), change their color to the RGB macro color red ("color red"), and finally continue by then selecting all the atoms occur both in proteins and the helix portion of the molecule ("select protein and helix"). If these commands are out of order then the incorrect portions of the molecule could easily be acted upon.

The order of our token declarations also mattered to the parser. Since it is traversing the rules of the grammar supplied, JavaCC considers each token against a rule until a match is found. Since several tokens are declared in the program, as well as regular expressions that are designed to match variable input streams, the placement of the rules in relation to one another is important. An input token could match a rule declared towards the beginning of the language but designed to fulfill a rule further which comes after. A false positive would result, potentially drawing a token into the incorrect function and rejecting it as not matching the entire rule when it was indeed correct but misplaced.

ChimeParser.jj was designed around a grammar that had many commands that appeared to be very similar to one another. The language had to be dexterous enough to pick the correct path through the parser with a token that was nearly identical to others in location or form. Commands also had multiple levels of their own trees. Once a token's path was decided, often the next token consumed would require a choice as well resulting in complicated solutions. By using regular expressions we were able to fix these issues but had to design our rules to match only one input token and be unambiguous. Arranging our grammar from specific to general we avoided false positives. A token that only appeared once or could be declared as a fixed string of characters would be towards the top of the rule list where more general tokens or regular expressions that could match many strings came after.

3.1.4 LL(1) Parsing and LOOKAHEAD

JavaCC is a full developed language, with many functions and capabilities that we did not utilize throughout our work. However we will outline some of the important functions we did use and supply the reason behind their use as well. Most notably is that JavaCC is based on LL(1) parsing (3). This means that the languages that JavaCC can parse are non-ambiguous. This feature allows the parser to essentially consume a token and then consider the next token left unconsumed while trying to determine the parsing path of a command. Languages defined by the user are often not LL(1), meaning there is a degree of ambiguity in them at the single token look ahead level. JavaCC includes a powerful command LOOKAHEAD that will help guide the parser around any parts of the input language that don't conform to the proper form. As a result, JavaCC can handle any grammar that is not left-recursive (3). There are several ways to properly use the LOOKAHEAD command, and if the developer ever fails to recognize a fork in the pathway at compile-time, JavaCC will remind them with a warning. By using the LOOKAHEAD function, potentially ambiguous portions of the grammar, or those sections that don't conform to the LL(1) rule can still be evaluated. The most basic method, and the method preferred in the creation of our parser, was the local LOOKAHEAD. The programmer can specifically tell the parser how many tokens it should consider ahead of the consumed token before it makes a decision on a path. For example, a command that might be similar in the first two tokens would be uniquely parsed around this ambiguity if a LOOKAHEAD of 2 or more was used (LOOKAHEAD(2)). Since we used an input language that was fairly linear this method was enough to allow consideration of the grammar for possible choke points. However, more powerful LOOKAHEAD methods exist that would point the parser to consider entire functions before making a choice. These are called 'syntactic lookahead' and while not used in the creation of our parser, offer an idea into just how powerful the JavaCC language can be.

While ChimeParser.jj was not a processor intensive application it is also important to note that the closer a developer can get to a true LL(1) language, the more streamlined and processor efficient the grammar becomes. We did not have the need to explore just how much savings exists so we offer it as an exercise for future researchers.

3.1.5 Layout

JavaCC programs have a unique layout that allows them to function as both a parser and a Java program. They start execution classically with a static main method within which the parser is started via some form of a run function. However the main method is nestled inside a "PARSER_BEGIN" command that will bookend the start and stop points when paired with its inverse "PARSER_END" command. Tokens are then declared, behaving as keywords which trigger the parser to recognize a specific order of characters. The tokens can either be strings, numbers, or regular expressions. We used many of the Chime and RasMol commands as keywords, as the following examples illustrate, reserving them for the parser and not allowing them to be used in other forms such as filenames.

```
< DIRECTION : “direction” >
< VALID_COLOR : “white” | “black” | “red” | …. >
```

We also nested tokens to create larger, more complex regular expressions that would logically satisfy a specific input requirement.

```
< FILE_STRING : < CHAR_LIST > “.” < SUFFIX > |
“.” “.” “/” (< CHAR_LIST > “/” | “.” “.” “/” )* < CHAR_LIST > “.” < SUFFIX >

< #CHAR_LIST : < CHAR > (< CHAR > )* >
< #CHAR : [ “a”-“z”, “A”-“Z”, “-“, “~”, “0”-“9” ] >
```

These lists of token definitions were then references from the functions to determine if a token was correct of not.

The input strings were broken down into their subcommands and specific tokens and functions were created. Some commands would only consist of one word or an input string, and these were easily fed into the parser by short, simple functions. Others would be more complex, involving ambiguity as some level or choices deep in the parsing tree. As the supplied source illustrates, when working with JavaCC, tokens must be immediately handled or their scope will be lost by the parser. It is not enough to carelessly consume token after token and expect to then reference them in turn. Without proper nesting and assignment, tokens might be lost for good once consumed. While we have celebrated the capacity of JavaCC to execute classic java methods within a parsing function, due to the sheer volume of the commands we accepted as our subset, we decided many would echo back to standard-in after being checked. This was a design choice, remembering that this project was an exercise in the feasibility of such a parser and a basic control of the visualization software on the other side. We did not long entertain the idea of implementing every Chime and RasMol command, but offer that as future work. Additionally, this software is completely flexible and whatever execution we chose for a particular command could very well be overwritten by the next researcher to use this source.

Every function that we wrote was divided into two distinct portions. The first is where we declare the variables which would be used throughout the scope of the function. Unlike classic Java, this part was segregated from the main code of the function by open/closed braces, { }, and would appear directly after the function header.

```
void HelloWorld() :
{
        Token worldTok;
}
```

This would then be followed by another set of open/closed braces which would contain the JavaCC token manager code. Finally, any Java code that the developer wishes to include is further offset by yet another set of braces. By nesting distinct blocks of code like this, the parser knows what to consider during its parsing stage and what to

ignore until it is time to generate the .java files. Continuing the above examples, the entire function would appear as:

```
TOKEN :
{
        < WORLD = "world">
}
void HelloWorld() :
{
        Token worldTok;
}
{
        worldTok = < WORLD >
                {
                        System.out.println( "Hello " +
                        worldTok.image );
                }
}
```

We will be exploring specific JavaCC commands and stepping through the decision process it follows in the coming sections.

3.1.6 Special Characters

In addition to composing regular expressions that would allow strings to be used as tokens in the program, we also reserved some tokens to be ignored by the parser. JavaCC allows the user to specify is special characters are ignored such as white-space, carriage returns, or new lines. However we added an additional element to this with our design that was more complicated then ignoring a specific token. Chime and RasMol allow commands to be separated by carriage returns or together on a single line, separated by a semicolon. This meant that we needed to consume the semicolon, recognize it as a macro for a carriage return, and then throw it away without letting it disrupt our other rules. We decided to define the token < SEMICOLON : ";" > and wrote a helper function in JavaCC that would execute a Java carriage return that could be added to each command in the program.

If during parsing a semicolon was consumed JavaCC would take the proper action but would not mix it in with any of the other tokens in the rule list.

3.1.7 Creating JavaCC Functions from Input Commands

The process of creating a JavaCC function is straightforward. Since JavaCC evaluates each token in turn, functions are designed to step through incoming tokens as they appear. Each token is considered against the rule list built within the JavaCC grammar.

Input command : "anim { on | off }"

The “anim” command is declared in the token list within the JavaCC program. The declaration appears below the “public static void main(String args[])” method that starts the execution of the parser, but before any of the parser-specific functions are defined. In order to declare the “anim” command, the following was added to ChimeParser.jj:

TOKEN :
< ANIM : “anim” >

This reserves any input string that matches “anim” as being only considered as the “ANIM” token. If another command uses the same string an error will be generated.

The first function definition in the stack of execution within ChimeParser.jj includes a list of possible command types that are acceptable. This function is the first grammar driven consideration the token is brought through. In ChimeParser.jj this function is called “cmdList()” and called from the main() method. Commands have the top of their parsing trees here, and if any functions match the first token seen by the parser, they start that specific command.

```
void cmdList() :
{}
{
        (  anim()
… // addition commands are added as created
        )
}
```

JavaCC knows which function the token belongs to by considering that token against all first declared expected tokens of all the functions in this list. Since we didn’t declare a LOOKAHEAD, only one token at a time is considered. If there is any ambiguity here (two or more functions are expecting the same starting token), JavaCC will warn the user and suggestion an action to correct it.

After the correct starting function is decided on JavaCC skips down to that function and then begins to evaluate the token in greater detail. It knows that the first input token in the function is expected to be the token it is considering, so it immediately drops into that command’s function.

```
void anim() :
{
        String animStr;
        Token lastTok;
}
{
        < ANIM >

        animStr = animParser()
                {
                        System.out.println( "anim " + animStr );
                }
        [ lastTok = < SEMICOLON >
                {
        insertSemiColon();
                }
        ]
}
```

Once within the anim() command function, JavaCC behaves a lot like a classic Java method. It first defines any variables used within the scope of the function. In ChimeParser.jj two variables were declared. The first was a String that would be sent to standard-out as the echo of the command. The top function in the command parsing tree had void return types while any supporting functions returned Strings. Since JavaCC would consume tokens and move onto the next while moving down the tree the string is generated at leaf functions first and then pushed back up the next, most peripheral function branch in a form of recursion. The second variable was a Token class object.

In ChimeParser.jj, the token manager had the "ANIM" token as its first object consumed. The first line of the anim() function was what it read when considering which function it would skip to as it continued to evaluate. The next line called a child function of anim() which would prompt JavaCC to consume another token, and move the evaluation one step down the command tree for "anim". It returns a String, as mentioned, so that String must be assigned to a variable or lost.

Once that String is returned to this function, execution of the entire "anim" command has completed. If a correct command was inputted to the parser, then this function would echo it back to the user via the System.out.println that is executed after the String returns. This Java code is only run it the parser reaches it through the proper parse tree.

The last lines enclosed by [] is an optional rule for this command. Since JavaCC is essentially comparing consumed tokens to user defined regular expressions it can be altered to optional portions (if the token is seen, in this case a "SEMICOLON"), told to run portions zero or more times with a Kleene Star, or fed regular exceptions expecting tokens of varying strings.

After completing the consideration of the input command, ChimeParser.jj returns to the original function, consumes another token, and starts the process again. In this way, as many commands as the user wants can be considered with one initialization of the class.

3.2 MBT (the libraries):

As the molecular visualization rendering engine, MBT was designed to work as a library of Java classes to create an image of a molecule specific to the needs of the user. From the MBT Introduction and Overview site (http://mbt.sdsc.edu/docs/ProgGuide/Introduction.html) we find the following applied definitions for each of the packages found in the library.

edu.sdsc.mbt	This package provides classes which define the core data storage containers for the MBT toolkit.
edu.sdsc.mbt.filters	This package provides classes which enable filtering or subsetting of a structure's constituent components.
edu.sdsc.mbt.gui	This package provides classes which implement graphical user interface (GUI) component classes for MBT.
edu.sdsc.mbt.io	This package provides classes which enable molecular biology data sets (such as protein structures, sequence data, etc) to be loaded into an MBT application as a Structure object.
edu.sdsc.mbt.util	This package provides classes which implement extra functionality on top of the core MBT classes (that is, capabilities that are not absolutely required by, or would otherwise overcomplicate, the core classes).
edu.sdsc.mbt.viewables	This package provides classes which enable molecular biology data to be represented as visible/renderable viewable objects, plus, this package defines a top-level StructureDocument object to encapsulate the complete state of these objects and properties.
edu.sdsc.mbt.viewers	This package provides classes which enable data to be graphically visualized or to be processed by other means based upon coordinated events (that is, any MBT component that wishes to observe data and then respond and interact to changes in that data).

Table 1: MBT Package Summary

By adding or subtracting the classes used from the MBT package (see above table), the developer can customize the molecule spanning a wide array of interesting characteristics. At its core, the package (http://mbt.sdsc.edu/docs/api/index.html) focuses on basic chemical forms: atoms, bonds, fragments, chains, and their interactions with one another. Since it was written in Java, it is completely object-oriented, maintaining specific classes for each abstraction. Slowly working up in complexity, it defines elements and where they sit in the periodic tables, form collections and assigns unique labels and colors to each. MBT can even distinguish between hydrogen, covalent, and ionic bonds.

It is not our desire to give a comprehensive chemistry lesson throughout this report, but it is important to absorb the hierarchy MBT's packages follow and how it closely resembles the logical increments of complexity exhibited in real-world molecular components. Thusly, MBT can be dexterously made to recreate the interactions that make life work the way it does.

Mixed in with the classes representing biological components, MBT offers the end-user a strong base of both 2D- and 3D-graphics which are used to create the molecule desired. Using many of the classes from the edu.sdsc.mbt.viewables and edu.sdsc.mbt.viewers packages, we were able to affect the appearances of out test molecules. A DNA strain, defined from the 1D66-PWZ.pdb file, was a convenient find to use as our test case in many of our examples since it exhibited the classic double helix form familiar to many of us, as well as having many of the bonds and atoms which we targeted in our experiments. We also used the insulin protein, 9INS.pdb, and deoxyhemoglobin, 4HHB.pdb. Large proteins gave the best views for many of the backbone altering commands as we shall see. In conjunction with the power of MBT

rendering classes we made an important decision. Since a large library of rendering frames is available, each with a unique GUI and each focusing on a different, specific view of a molecule, it was decided that we would not pursue the creation of our own viewers. Within the MBT example source (http://mbt.sdsc.edu/examples/index.html) a robust viewer entitled Explorer.java exists. While this viewer does not showcase the full breadth of MBT's capabilities, it has predefined macros to instantly change the molecule's form as well has access to atom and bond specific functions.

Instead we concentrated our efforts on bridging our JavaCC parser and the input commands we were feeding it to alter the form of a given molecule. For that we had to look at several of the classes within the edu.sdsc.mbt.viewables (as well as supporting classes from the edu.sdsc.mbt.viewer) package. StructureStyles and StructureMap provided us with most of the direct control at the molecular level we needed. StructureMap rounded out the methods, giving us access to many of the collection classes we needed to touch each atom or bond in a compound. Of course these classes are supported by the hundreds of other classes that compose MBT. John Moreland and his team did such a seamless job of layering the interactions that it is easy to stay at a high enough level when using the toolkit as to not get overwhelmed.

3.2.1 StructureStyles and StructureMap

Most of the work of altering an input molecule was performed by the edu.sdsc.mbt.viewables package within MBT. These classes contained the methods that were targeted at changing parts of a molecule after being loaded. The StructureStyles class operates at the top of the hierarchy and can therefore control the subclasses within MBT. By generating the change from this level of the library it's easy to have the dexterity to make even the most basic control change. Figure 4 is a graphical layout of the MBT hierarchy that is observed when creating a molecule.

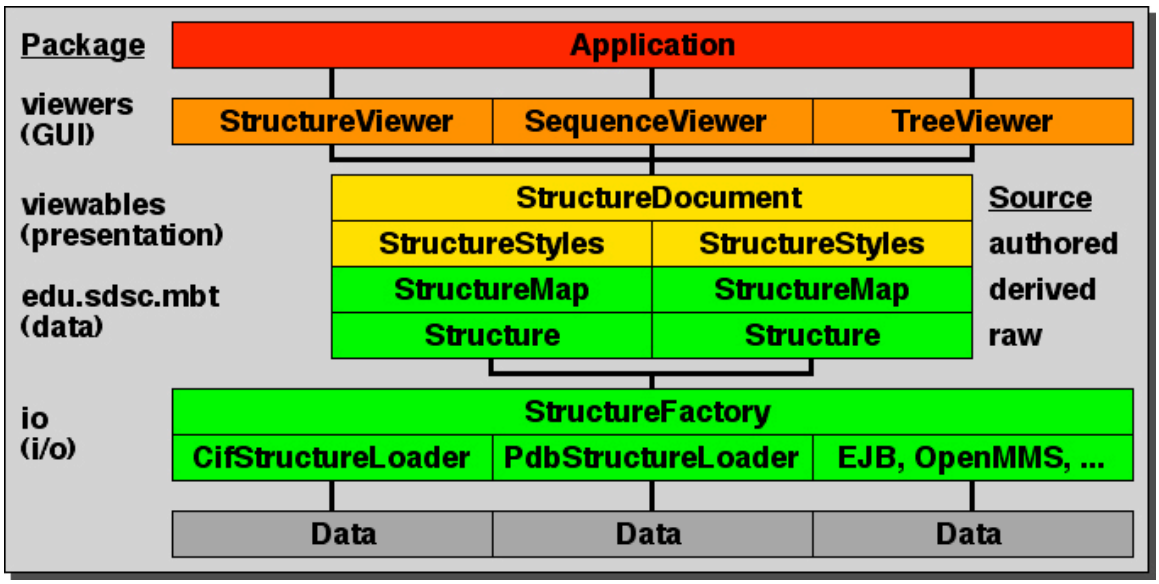

Figure 4: MBT Hierarchy

The StructureMap class, in the edu.sdsc.mbt package, is at the base of the library and holds the representations of the bonds, atoms, chains, and residues in the molecule. By linking between these classes across the breadth of the MBT package, the input scripts only need to access high level classes to make base changes as the following illustrates:

edu.sdsc.mbt

structure = StructureFactory.load(new URL(<url string>)); //builds a structure
StructureMap structureMap = structure.getStructureMap( ); //fills a map with objects

edu.sdsc.mbt.viewables

StructureMap structureMap = structure.getStructureMap( ); //links highlevel methods

3.3 Explorer.java

It is worth noting that even though we decided to use a molecular view that was pre-built we found we needed to add some methods to the class.

When we began to render with Explorer.java it contained various functions that allowed it to select, unselect, and save a molecule that was loaded into it. Many of the methods within it were useful such as selectAll() and unselect() and served as a good starting point for our additions. There were also methods that we didn't use but we left in the class so its drop down menus remained functional.

Due to Explorer.java instantiating many of the objects we needed from the edu.sdsc.mbt.viewables package on its initialization, we were forced to work in the scope of the class. Throughout many of the operations we would need to make calls to the methods within the StructureStyles class. While this class could be correctly accessed from ChimeParser.jj, we also needed to alert the rendering software in Explorer.java any time a change was made. However this was not disadvantageous for us since it also gave us a chance to move some of the functionality that wasn't appropriate in a parser (such as firing color update events) to the viewer where it belonged. The relevancy of these operations staying in the viewer and not the parser was a cleaner solution.

We did not have to alter Explorer.java drastically in order to make it work for us. We did add methods that allowed the parser and viewer to choose specific bonds selected instead of highlighting the entire structure. These methods, setBonds(boolean selection) and setAllBonds(boolean selection) served as our test targets for many of the script commands. By proving that MBT could be made to select user specified objects within the compound a user could easily add methods to manipulate any combinations of objects they desired.

Due to our changes in the supplied source of the viewer, we have included the modified code with our project package. All methods that have been added come at the end of the class and are documented accordingly. We did not affect any of the initial abilities that were built into the viewer when we first used the source.

4. Execution and Testing

In this section we will explore what steps we took to run the programs, what commands we picked to implement, and the testing we applied when developing the software.

The complete project package includes the JavaCC file ChimeParser.jj, the MBT supplied libraries, a viewer application, the collection of any .pdb files, and any script files we wanted to run. ChimeParser.jj had several key responsibilities and was the primary piece of software throughout the project. Acting as a token manager was the first key portion of execution. All input commands, since we assumed them to be from the Chime and RasMol command libraries, were subjected to the lexical analyzer. These input strings were broken down into discrete tokens, as previously mentioned and illustrated by Figure 5.

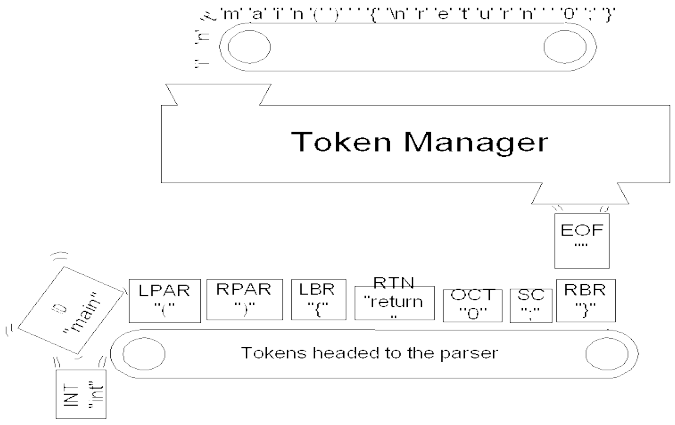

Figure 5: JavaCC Token Manager

These tokens were then fed into the parsing portion of the JavaCC file.

Tokens were the end result of the token manager breaking up a complete input stream into its basic components. Just like an acid might dissolve a penny into its atoms, base elements, larger composition compounds, and any impurities, the analyzer brings the command down to its lowest common components. At this point it does not make decisions on the correctness of the token, instead just classifying them and readying them for the parser to dissect.

Only once the token manager was done with the input stream would the second responsibility of ChimeParser.jj take over. The parser function would then check against the declared tokens existing within it as well as predetermined function, as in Figure 6.

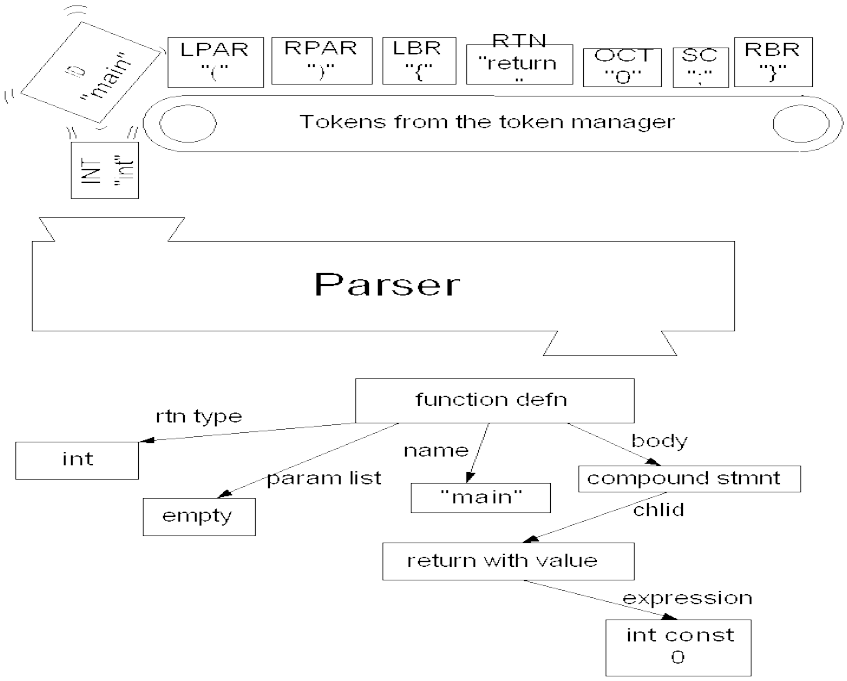

Figure 6: JavaCC Parser

Running ChimeParser.jj, regardless of the presence of input commands, would initialize a viewer. This viewer, Explorer.java, was taken from the MBT example source and served us as an adequate tool to render molecules. Since it resided in the same directory as ChimeParser.jj and was included in the script file created to easily compile and then cleanup up and source in the directory, it was compiled along with the other .java files. Explorer itself contains a clean, simple GUI with drop-down menus and other Java Swing components. However, as an example, it does contain some unimplemented portions as well. This didn't bother us since we were concerned with our own commands rather than the viewer's predefined set. A load command was required to execute the rendering of a molecule into the Explorer. This load could stimulate the viewer to search for a .pdb file residing locally within directories, or on the web via a correct URL. Once loaded, we could act on that molecule from the command line or run a script file.

Due to our original choice to remain a feasibility experiment, we did not implement all the Chime and RasMol commands available. The command set we did choose included those that dramatically changed the molecule's appearance or selected a target portion of the molecule. Dr. Paul Craig from RIT's chemistry department supplied an example RasMol script and we implemented a portion of those commands. We decided that would complete our goals if we could prove we could touch the molecule directly from within JavaCC as well as controlling the actions of the molecules in a manner similar to other MBT viewers. We have included the script used as well as a complete list of the commands implements in the Appendix.

Testing consisted of identifying border conditions for each command we implemented, and ensuring the correct action when those conditions are run. We tried to anticipate areas of vulnerability at the time of development, minimizing the testing phase. Correct and complete results were required at each step of the project before additional functionality could be introduced. Particular attention was given to the character and numeric literal token definitions in ChimeParser.jj. Since the parser checks for the first correct definition, order was an important consideration. Additionally, regular expressions designed to simulate some of the more complex input streams could inadvertently mimic more basic commands, and each had to be tested in turn.

5. Process

In this section we will explore what it is required to download, link, compile, and run the applications involved in the project. It is our goal to be able to use the following as a complete guide to running the program.

All work was done within the Computer Science labs in the Thomas Gollisano building on the campus of Rochester Institute of Technology. We used Sun Microsystems workstations running Solaris. The Java SDK as well as the Java3d applications were preinstalled and required no altering on our part. However it is important to note that the MBT team feels that since Java is platform-independent, that the MBT toolkit will run properly on other operating systems (http://mbt.sdsc.edu/docs/install.html). A minimum Java version of 1.2.1 is recommended. Previous versions remain untested and their performance not guaranteed. For the purpose of this experiment we used version 1.5.0 and found no Java related issues throughout the duration.

JavaCC version 3.2 was preinstalled within Professor Joe Geigel's faculty directory due to space limitations with student accounts. We added the package to our PATH so we could run the executable as needed.

MBT itself was retrieved from the MBT home page as a single .jar file. This file contained the complete package and was installed in Professor Geigel's directory as well. We modified our CLASSPATH to include the path to the .jar with the following command from standard-in:

```
% export CLASSPATH = $CLASSPATH:/home/fac/jmg/MBT/mbt_1.1.2_bin.jar
```

This allowed us to reference any of the MBT classes within the ChimeParser.jj program with simple imports, exactly like any other Java file:

```
import edu.sdsc.mbt.*;
import edu.sdsc.mbt.io.*;
import edu.sdsc.mbt.util.*;
import edu.sdsc.mbt.viewers.*;
import edu.sdsc.mbt.viewables.*;
```

It was only a manner of compiling both the .jj files and the resulting .java files before we could run the viewer and script. We created a basic shell script to aid in these tasks. The command 'runscript' would first compile all the .jj files within the directory. Since JavaCC is different then Java, it has a different compile executable command, JavaCC. Once invoked, it would try to compile the JavaCC program, creating in its wake functioning .java files. The next command in the 'runscript' file would then try to classically compile these .java files. The directory now contained any of the files generated by JavaCC's compiler as well as any stand-alone .java files such as Explorer. These were then compiled and resulted in the formation of .class files. We could then execute a functioning ChimeParser application. The third command in the 'runscript' file did just that. It ran ChimeParser with no arguments (meaning we were expecting commands from standard-in, a convenient default for the duration of the project for testing and debugging):

```
% java ChimeParser
```

To complete the execution, ChimeParser would initialize an Explorer object whenever run. An empty viewer window would load and await our command input. In the event of a load command, the viewer would search for the .pdb file and render the image on screen (Figure 7). From there, the user would input commands from the standard-in and, as long as they were supported by the application, would see immediate results.

Figure 7: MBT Explorer

6. Conclusion

So in conclusion we were successful after all of our efforts to modify a JavaCC program to act as a script controller for a visualization program. Our results were very positive and demonstrated easily.

Through the creation of the JavaCC program, ChimeParser.jj we built a parser that is capable of determining the correctness of an array of input commands. These commands can be supplied either from a file or from standard-in. Independent of the manner of input, our program would break down the incoming string to its base elements and then check those elements against a prescribed rule set. In the event of a correct command being supplied, the parser would complete its execution with any nested Java code. It was this point, where the command had moved from being a consideration to running Java methods that is defined as scripting.

Shuffling the command then to MBT's libraries, the program would have the ability to affect the visual characteristics of a supplied molecule, loaded into a viewer as a .pdb file. MBT has a much broader scope of functions than those we tested. However we remembered that our goal was to prove feasibility and therefore left the specific assignments of some of these abilities to users to come. Our initial goals were realized when we showed that we could make changes directly to the state of a molecule from outside the bounds of the MBT libraries. These changes were the direct result of script commands which originated with the two ancestor programs we chose to support our efforts.

Over the course of the project we did encounter our share of obstacles, but this was to be expected whenever undertaking the task of learning a new programming language. None are noteworthy as serious time sinks. We gained a deeper understanding of JavaCC and how parsers work. Creating an input grammar from a representative language and then accurately moving commands based on that grammar through our program proved the correctness of our work.

Collaborating with the chemistry department at RIT, we had the additional goal to build something noteworthy for their eventual use. After inspection of the project from both a computer science perspective it was examined for relevancy in both fields. As the Future Work section will outline, the potential is there for the groundwork we laid.

This project was an exciting, interesting scholastic effort that hopefully made headways into better understanding molecular visualization and the opportunities available.

7. Future Work

Within the various components of the complete package, a contributor has several areas on which to focus their own studies. During development, the team deemed some aspects better kept for projects yet to come. As a research university, RIT encourages its students to seek out how to best contribute to the school's knowledge base. Particularly motivating was the potential for further future collaboration with physical and life sciences, leading to possible publication in scientific journals. While our chemistry was rudimentary, we put in place a framework upon which can be built addition work of specific interest.

The most basic area of future work would be to implement the full library of Chime and RasMol commands into JavaCC and then port them to MBT via a scripting command. With the close help of the chemistry department, many of the Chime and RasMol commands could be modified to have more control in MBT. As noted, we wanted to prove feasibility so we stressed changes stimulating immediate visceral reactions. It was easy to see a backbone change form from wireframe to a ribbon, but much more subtle to understand how changing specific bonds might change the chemical framework of the molecule. Any future work to implement the complete array of Chime and RasMol commands would be an exercise in chemistry are much as computer science. While this would have tangible benefits to both departments in question, it remained out of the scope of our goals.

Perhaps the area of work most immediately relevant would be expanding the input languages the parser supports. While we chose Chime and RasMol for their popularity and the existence of legacy scripts, they are not the only visualization programs available. RasMol itself has been upgraded to another product, Protein Explorer, developed and maintained by the same team from the University of Massachusetts (http://www.umass.edu/microbio/chime/pe/protexpl/frntdoor.htm). Another product of particular interest to Dr. Craig is PyMOL (http://pymol.sourceforge.net/) into which he's devoted significant resources of time and energy. While did not consider Protein Explorer or PyMOL command lists when implementing our scripting routines, they tangible choices for molecular visualization as well. Additional programs all contribute to the field's overall erudition and each also doing something better than all the rest. PyMOL, for example, has a user interface that is lack but has stunning graphics. Our work can help bridge these Babel of applications by giving them a common language. Adding our backend to the capabilities and scripts of a wide array of programs will give the researcher the best possible choice. As long as these languages have command built on similar grammars and are open source, it would be possible to link them beneath a single, master scripting program.

Another area of future study would be adding a degree of machine learning to the process of visualization. Proteins, and their active sites, are complex in their composition but are vital to the understanding for countless fields in both the life and physical sciences. Being able to anticipate how molecules will interact with one another is of immense interest. These molecules have many similarities to one another, and by adding

machine learning to known structures, a future researcher might perhaps be able to predict how new structures will behave. At a fundamental level, gaining an understanding for the formation of a protein's complete structure would aid in predicting if new combinations of atoms could form a viable molecule. This area has such wide ranging implications it is impossible to gauge the complete span. Being able to visualize new drugs and viewing how they might interact with a popular target such as a G-protein coupled receptor across a cell membrane would speed up to-market research (6). This would be possible because the combination of MBT and our frame work would set the base to add artificial intelligence to the process. At the microbiology level any research and advances would be sure to be received readily by the scientific community. We have the facilities and the curious spirit at RIT to perhaps make a real attempt at the problem. While obviously deviating from the span of this project, starting to understand complex visualization programs such as MBT as well as some of the leading researchers in machine learning may yield a tempting nugget.

Additionally, future researchers could choose to add scripting directly into MBT instead of being controlled by a third-party program. Although MBT was designed not to incorporate this capability, future work could focus expand the MBT libraries to include a scripting class. Eliminating the JavaCC component would not even require a user learn a new language, interacting much as we did with ChimeParser.jj. However this class wouldn't have the flexibility to be as far reaching as an independent program such as ours. MBT would be the dominant language and the others would be modifying their commands to execute its classes.

Lastly, a future student would have boundless choices for implementation. We kept the exercise simple and used a supplied viewer, but this experiment could be expanded to include web based implementation, a XML study, or even as the basis for a complete end-user application. Commands, instead of being supplied solely by command line or via file, could be delivered by intelligent web systems. Being applied to a web environment brings with it its own challenges, but researchers could decide the degree to which they would like to go.

We made a leap across fields of study and were able to help the distance between those fields by designing and implementing this solution. By opening this portal for the physical and life sciences, we step towards merging the fields and offering our contemporaries a chance to build on our work. While we have maintained undertaking this challenge as an experiment in feasibility and possibility, the potential for the future is boundless.

Appendix A

- The Chime RasMol script commands: http://www.mdl.com/support/developer/chime/developer_tools/chimerasmol.jsp?start_download=true&resource=/support/developer/chime/downloads/public/rasmol_scripts.pdf

  Since Chime is a superset of RasMol commands, these commands are the ones that had been previous scripted in Chime taken from RasMol. They served as the bulk of the commands we ported to MBT.

  These are not implemented past echoing back to the user on standard-out if correct.

- RasMol v. 2.7.3 commands scripted to MBT:
  - Color Commands:
    - color { <target object> } { <color> | <RGB value> }
      selects the target object and changes the color of that object within the rendered molecule. Implemented for the 'bonds' target object. Other objects checks for correctness and echoes entire command to user.

      - Valid Target Object:
        - backbones
        - bonds
        - hbonds
        - ribbons
        - ssbonds

    - color { <color> | <RGB value > }
      changes the color for any selected object within the rendered molecule

      - Valid Colors:
        - 24 predefined RasMol colors
        - none
        - [<value>, <value>, <value>] RGB color specification
        - cpk – unimplemented, checks for correctness and echoes entire command to user

    - colour { <target object> } { <color> | <RGB value>}
    - colour {<color> | <RGB value >}

- Select Commands:
  - select { <expression> }
    examines the structure for atoms, bonds, chains, or residues that are defined within the scope of the target object – if found, will follow expression command (either select or not select). We implemented protein, helix, sheet, turn, dna, and water as target objects

    - Valid Expression:
      - all
      - none
      - not { <target object> }
      - not selected
      - { <target object> } (and { <target object> })*
      - { <target object> } (, { <target object> })*
- Set Commands:
  - set axes { on | off }
    checks for correctness and echoes entire command to user

  - set bonds { on | off }
    examines the bonds within the loaded structure for double and triple bonds – if found, selects those bonds and either turns them on or off

  - set boundingbox { on | off }
    checks for correctness and echoes entire command to user

  - set specular { on | off }
    checks for correctness and echoes entire command to user

  - set specpower { <value > }
    checks for correctness and echoes entire command to user

  - set unitcell { on | off }
    checks for correctness and echoes entire command to user

- Visual Commands:
  - backbone { on | off }
    alternates the appearance of the structure from backbone view (a multicolored line tracing path of molecular backbone) and ball-and-stick view (all bonds and atoms are rendered)

  - cartoon { on | off }

alternates the appearance of the structure ribbon view (portions of proteins are rendered as ribbons of various thicknesses) and backbone view (a multicolored line tracing path of molecular backbone)

- dots { on | off }
  alternates the appearance of the structure from ball-and-stick view (all bonds and atoms are rendered) and stick view (only the bonds are rendered)

- hbonds { on | off }
  examines the bonds within the loaded structure for hydrogen bonds – if found selects those bonds and either turns them on or off

- labels { on | off }
  either includes a label on each atom containing the name of that atom or no label on each atom

- monitors {on | off }
  checks for correctness and echoes entire command to user

- ribbons {on | off }
  alternates the appearance of the structure ribbon view (portions of proteins are rendered as ribbons of various thicknesses) and backbone view (a multicolored line tracing path of molecular backbone)

- spacefill { on | off }
  checks for correctness and echoes entire command to user

- ssbonds { on | off }
  checks for correctness and echoes entire command to user

- wireframe { on | off | <value> }
  checks for correctness and echoes entire command to user

Appendix B

Index of software included:

- README
- ChimeParser.jj – JavaCC program, compiled before .java files
- Explorer.java – molecular visualization software
- mbt_1.1.2_bin.jar – binaries of MBT packages. Used for classpath
- mbt_1.1.2_scr.jar – source of MBT. Used for additional development

References

1. MDL Chime
http://www.mdl.com

2. RasMol Home Page
http://www.umass.edu/microbio/rasmol/

3. Norvell, Theodore S. (2004). *The JavaCC Faq*. Retrieved from Memorial University of Newfoundland Web site: http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq.pdf

4. JavaCC Tutorial
http://www.cs.rit.edu/~jmg/javacc/

5. The Molecular Biology Toolkit. J.L. Moreland, A.Gramada, O.V. Buzko and P.E. Bourne 2005 The Molecular Biology Toolkit (MBT): A Modular Platform for Developing Molecular Visualization Applications. BMC Bioinformatics, 6:21. Funded by Grant NIGMS 1P01GM63208. Web site: http://mbt.sdsc.edu/

6. Kenekin, T. (2005). New Bull's-Eyes For Drugs, *Scientific American*. October 2005, pp. 50-57.