Biological Scripting Language (BSL)

Jared Eng

Jay Kota

Igor Marfin

Amna Qaiser

Contents

Chapter 1

BSL Whitepaper

Introduction

Over the last decade there has been an accelerating interest in acquiring programming skills on the part of biologists to solve problems using DNA, amino acid sequences, and related information. Biological Sequencing Language (BSL) is designed to help biologists continue the trend of using computers to achieve their goals by making biological data processing easy to use and simple to understand. BSL specializes in sequence manipulation, translation, and analysis. Moreover, the language is compiled into Java bytecode, which together with the Java Virtual Machine makes BSL platform independent. These qualities make BSL a powerful, flexible, and easy to use tool for programmers and biologists alike.

Background

There is a long history of computer use in biological research, dating back to the early days of digital computers. About ten years ago, the large-scale international human genome project began to generate data of volume and of a kind that required a carefully planned and significant computer programming effort. Since then, the field of bioinformatics has been growing at an increasing rate.

There are as many kinds of biological data as there are experiments. In biology, the most basic data type is a base such as A, C, T, U, and G. Multiple bases in specific combination link together to form sequences called Deoxyribonucleic (DNA) and Ribonucleic (RNA) acids. Living bodies use these sequences to create amino acids, which in turn join to form various protein sequences. These proteins are used to form complex organic structures. By studying

these sequences, biologists hope to understand the inner workings of living things and, most importantly, the human body.

Currently the field of bioinformatics deals with the manipulation and analysis of DNA, RNA, and protein sequences. Biologists use computers to store large strings of these sequences. They also write programs to access, search, align, map, and translate sequences. They use this data to identify unique organisms, locate different characteristics of those organisms, and make predictions about their behavior.

Related Work

Current technologies that are available to a biologist include library extensions for existing programming languages such as BioPerl, BioJava, and BioPython. Fast string manipulation and built-in use of regular expressions make these libraries very useful in biological research. However, using these tools requires the user to learn computer programming including the involved data structures, complex language syntax, and many different programming paradigms. Software packages are also available for biologists to perform these analysis techniques. They are well designed and aesthetically pleasing. While these programs provide a wide range of functionality, they can get inflated with unnecessary features. In addition, these programs can be very expensive and restrictive. BSL provides the flexibility, the sophistication, and the computational power comparable to its peers, but without the complexity of a full-blown programming language or software package.

Ease of Use

BSL makes programming easy even for biologists with limited or no prior experience because of its limited terminology and high-level syntax. Also, use of keywords related to the biological sciences allows a user to identify and quickly understand the specification of BSL. A person who wishes to compare two DNA sequences does not need to have a detailed knowledge of BSL to write a script that returns results in a quick and efficient manner. A sample script is as follows:

```
/* Here is a test script */
start method int GetOne (Sequence a, Sequence b)
      return 1;
end method;

new Sequence dnaSeq1 type DNA = "AGGGAACCTT";
new Sequence dnaSeq2 type DNA = "AGGAACTC";

new int number;
number = GetOne: dnaSeq1, dnaSeq2;

Print: number;
/* End script */
```

At the other end of the spectrum, a programmer does not need to have a deep understanding of the biology to write a script that gives him results. With basic understanding of DNA and protein elements, a computer programmer can build a script that provides advanced programming techniques for biology. While the types identified in BSL are from a biological standpoint, they are simple enough that they can be applied to a variety of other topics.

Object-Oriented

BSL implements basic concepts of the object-oriented (OO) paradigm. Simple objects are used to facilitate efficient use of data behind the scenes of a script. Objects allow for easy cataloging of data types and provide limited scope for errors. However, the user script does not need to contain any OO rules. This background implementation will not force a user to spend time learning OO theory. Instead, the focus is shifted to writing a powerful script. At the same time biologists can take advantage of intuitive and basic objects upon which they can build on. For example, a biological macromolecule is made up of monomers, which are usually represented by a single character. Usual handling of these characters is performed using String API of Java. This however leads to several inefficiencies. Monomers that make up a macromolecule or sequence are represented using a single character. And since this character can mean different things in a DNA sequence and an RNA sequence, handling of sequences becomes very error prone. Instead, BSL handles these problems in the background by referencing objects for the character representations. For example, while the character "T" means Thymidine in a DNA sequence, the same "T' represents Threonine in a protein sequence. In BSL, a T building block for DNA sequencing is different from a protein sequence's T building block.

Data Types

BSL data types are very specific in meaning but yet also can be applied to a variety of things. There are two types in BSL, primitives and BSL types. Primitives consist of types such as ints and chars. The BSL types are the more advanced types that allow for functionality of BSL to be used. These types include Sequence and Building Block. Only one variable of the current name can be declared to be used and multiple values cannot exist, meaning BSL is statically scoped. Furthermore, since these objects will usually have high memory consumption, static data typing will allow the script to run more efficiently.

Error Handling

BSL provides error handling which allows the user to write robust code, controlling the execution flow of the script. Errors that occur will be described in understandable terms and allow for quick fixes and shorter debugging time. Mainly if the user is performing an illegal action upon a sequence, BSL will report an error to the user and terminate execution.

Bytecode

BSL is compiled into Java bytecode. This allows a script file to run in any environment that is currently running the Java Virtual Machine (JVM). A user can write a script on one machine, transfer the script to another machine that is running the JVM, and then execute it without compatibility issues. This form of compilation allows BSL to have all the benefits that JVM provides. The JVM implementation of the automatic garbage collector means easier memory management since allocation and de-allocation are taken care of. As mentioned above, portability is also added to BSL.

Conclusion

The Biological Scripting Language offers a wide range of new capabilities to both biologists and computer programmers. Its powerful and flexible nature allows a user to write complex code simply and obtaining results efficiently. All in all, BSL scripts will save its users numerous resources.

Chapter 2

BSL Reference Manual

Biological Scripting Language (BSL) is especially suited for fast, easy script creation. Its main focus is to handle the analysis of biological data such as DNA sequences and amino acid chains. The language has simple yet powerful structures that provides for efficient functionality to be coded. The language syntax and semantics are designed to closely resemble other programming languages in order for scripting to be fast and easy to learn. Porting code from other languages has also been made straightforward so that BSL scripts will comprise of much simpler code.

BSL operates by translating a script into Java code and then invoking the Java compiler to create bytecode that can be executed. This reference manual describes the rules and built-in functionality that will help a user to write a BSL script.

1.0 Lexical Transformations

BSL syntax rules describe how words and symbols appear and are used within a script. Comments and white space, when used correctly, can be a great asset to code maintainability and more importantly, readability.

1.1 Comments

Comments in BSL scripts are delimited by starting symbols `/*` and terminating symbols `*/`. They provide explanation for the programming code. The compiler disregards all words within the starting and terminating comment symbols. All BSL keywords and any character can be used but when the comment symbols are placed around them, they are rendered meaningless. Finally, unlike regular statements, comments do not need to end in a semicolon (`;`) and comments are not nested. BSL comments provide the same functionality as C++ and Java comments.

1.2 White space

The compiler ignores white space in BSL. Its purpose is to make the code for the script easier to understand. It is defined as ASCII space with horizontal tabs, line feeds, terminal characters and comments.

1.3 Terminal Characters

The end of any statement of code in BSL is signified with a semicolon (`;`). ASCII characters for carriage return (`CR`) and/or line feed (`LF`) also signify new lines in the code. These characters are without semantic meaning and are ignored by the complier.

1.4 Tokens

Tokens in BSL include identifiers, keywords, and separators. White space is important to separate numerous words to allow for correct identification of the tokens.

1.4.1 Identifiers

An identifier is any combination of letters and digits given certain rules. It is used to identify variables used throughout the script. There is no limit to the size of the identifier but every identifier must begin with a letter and cannot contain any white spaces. Underscores and dashes can also be used in addition to digits (0-9) and all letters of the English alphabet.

Uppercase and lowercase letters are different. For example, 'A' is not equal to a variable defined as 'a'. However two identifiers with the same Unicode character definition for every letter and alphabet are equal.

1.4.2 Keywords

There are certain reserved words in the BSL language that have specific meanings. These words cannot be used as variable names and they are as follows:

```
AMINOACID       boolean   break
BuildingBlock   byte      char
DNA             else      elseif
end             exit      false
for             if        int
method          mod       next
new             null      object
overload        PROTEIN   return
RNA             short     start
Sequence        STRING    then
thread          true      type
UNKNOWN         void      while
```

Note: Keywords `object` and `thread` are reserved for future use

1.4.3 Separators

These ASCII characters act as separators and have programmatic functionality:

```
{} () [] : ;  , .
```

2.0 Types

There are two categories of data types allowed in BSL. They are primitive types and BSL types. BSL uses the same type referencing rules as Java, and therefore, primitive types are passed by value and all remaining objects are passed by reference.

2.1 Declaration

Before any variable can be used in BSL, that variable must be declared in the script. An example of a basic declaration statement is as follows:

```
new variable_type variable_name;
```

When declaring a BSL data type, an extra attribute must also be defined; this is an identification attribute and takes the form:

```
new BSL_type variable_name type BSL_Type;
```

BSL_Type can take the value of DNA, RNA, AMINOACID, PROTEIN, STRING, or UNKNOWN. Also when declaring, a user can assign a value to the variable by placing an = and then the necessary value after. For example, here is the creation of an integer with the value of 10:

```
new int myInt = 10;
```

2.2 Primitive Types

Primitive types include:

```
short  int  char  boolean  byte.
```

To define any of these variables, the `new` *primitive_type* statement should be used. The `short` variable is defined as a 32 bit integer and can range from -2,147,483,648 to 2,147,483,647. `int` is a 64-bit word that can also be used to define decimal numbers along with integers of larger values. An `int` can range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 5.0×10^{-324} to 1.7×10^{308} with a precision of 15-16 digits. `Char` can be any character value allowed in ASCII character alphabet. `Boolean` values are `true` and `false`, which can also be equal to 1 or 0, respectively. `Byte` is a variable representing a byte of data. Note that if a primitive type is not assigned a value, then it has a default value of zero.

2.3 Basic Types

Basic types are more advanced data structures that are built upon the primitives. These types are `array` and `matrix`. Similar to most programming languages, these types allow for greater functionality to be scripted.

2.3.1 Array

An array is a one-dimensional list of variables of the same data type. To declare an array, make this call:

```
new Sequence temp[x];
```

This initializes an array of sequences of size x. You can access any variable of the array by the index number, which ranges from zero to the size of the array minus one. BSL arrays are very similar to Java arrays but multidimensional arrays are not allowed. Arrays have an internal attribute named length that can be accessed by using this call:

```
temp.length;
```

2.3.2 Matrix

A matrix is specifically a two-dimensional array. Any user can create a matrix using the call:

```
new Sequence temp[x,y];
```

This creates a matrix of height x and width y. To access any value inside the matrix, make the call:

```
temp[1,1];
```

This code returns the object located at height 1, width 1. Exactly like an array, this object has a length attribute.

2.4 BSL types

BSL types include `BuildingBlock` and `Sequence`. These objects provide the power and functionality of BSL. A `Sequence` is any sequence of characters; most familiar uses will be for a DNA sequence, which comprise of bases {ACGT} or a protein sequence which comprises of a sequence of amino acids.

2.4.1 Identification

All BSL types have an identification attribute that will identify the data type as a DNA, RNA, AMINOACID, PROTIEN, STRING or UNKNOWN. All initial objects will be of the UNKNOWN identification. When biological data is entered into the object, BSL dynamically attempts to determine the type of sequence or building block that is being used. If no correct identification can be completed, the UNKOWN id remains. To access this attribute, use the `.seqType` syntax.

2.4.2 `BuildingBlock`

The `BuildingBlock` object represents a single character that is part of a sequence. It can be assigned any single character value that is identified as an ASCII character. To access the value, make the call `BuildingBlock.value`.

2.4.3 `Sequence`

A `Sequence` is made of building blocks. It can be created using an array of `BuildingBlocks` or a list of characters delimited within " and ". Each character of the list will be assumed to represent a `BuildingBlock` of the sequence. Besides the identification attribute, the `Sequence` object also has a length attribute, signifying the size of the sequence. Every `Sequence` also has a confidence attribute. The confidence level is based on the number of known bases, and this value is directly useful for DNA and RNA sequences. Each `Sequence` begins with a confidence value of 100 and each unknown base will lower the confidence level by a penalty value. This can be thought of as the percentage of `BuildingBlocks` known within a `Sequence` object.

Use the `Sequence(x)` notation to obtain the x^{th} block value from within the sequence. To obtain a subsequence from the total sequence, use `Sequence(x,y)` where x is the starting position of the subsequence and y is length of the subsequence. Note that y has to be smaller than the length of the total sequence. To access the total sequence of the object, use `Sequence.value`, which returns an array of characters.

3.0 Operators

The following is a list of the regular mathematical operators: +, -, *, /. The + operator adds two numbers or concatenates two `Sequences`. The – operator subtracts two numbers. The * operator multiplies two numbers and / returns the quotient after the division of two numbers. Note that -, *, / bear no significance when used with `Sequence` objects. Note that when adding a `short` to an `int`, any existing decimal characters will be lost because the `int` will be demoted to a `short` value.

The `mod` keyword also acts as an operator and returns the remainder after a division of two numbers. For example:

`7 mod 3` would return the value of 1.

In addition, all mathematical operations will be calculated in the standardized approach (left to right), but use of parentheses ( ) will allow the precedence rules of mathematics to be followed.

3.1 = operator

The = operator assigns the value on the right side of the statement to the variable on the left side of the statement. Importantly when using this operator with objects, the objects must be delimited by parenthesis. Constants do not have this requirement. For instance:

```
i = 1;
```

is allowed, however the following syntax should be used for objects:

```
i = (i);              i = (i + 1);
i = (seq1.length);    i = (seq1.length + (i));
```

3.2 dot (.) operator

This operator is used to obtain internal attributes of an object. An example of this operator in usage is:

```
tempSequence.length;
```

This line returns the size of the sequence.

4.0 Control Statements

A user can define how the script performs certain actions or enters into certain states during execution. The following section describes condition statements and loop statements that allow for the control of execution.

4.1 `break`

If a user wants to leave a loop before the full execution of the loop, a `break` statement can be used to exit the loop.

4.2 Conditionals

In conditionals, a certain action is taken given some prerequisites that are met.

4.2.1 `if` statement

`if` statements have the following syntax:

```
if (condition1) then
    /* code1 */
elseif (condition2)
   /* code2 */
```

```
else
   /* code3 */
end if;
```

All `if` statements begin with an initial condition that must be true to perform the actions specified within code block 1. If the first condition is not met, then code block 1 is not executed. All `if` statements must end with

```
end if;
```

To have the run-time execute other actions if condition1 is not true, then `elseif` and `else` statements can be included. `elseif` statements require conditions that must be meet to code block 2 to be executed. It is recommended that condition from `elseif` statements be different from `if` statements. Finally, `else` statements do not require a condition and code block 3 is executed if condition 1 and condition 2 are not true. Nested `if` statements are permissible in BSL.

4.2.1a Conditions

Conditions evaluate to true or false and usually involve comparisons. The operators that are used in comparison operators are

`>` `<` `==` `!=` `>=` `<=`

The > will return true if the left side is greater than the right side.

The < will return true if the left side is less than the right side.

The == will return true if the left side is equal to the right side.

The != will return true if the left side is not equal to the right side.

The >= will return true if the left side is greater than or equal to the right side.

The <= will return true if the left side is less than or equal to the right side.

A condition statement can be composed of more than one child condition statement and use of logical operators && and || allow for this type of execution. Two things can be tested and a && condition will be true if and only if both of the children conditions evaluate to true. For the || operator, the condition will evaluate to true if one of the children is true.

4.3 `for`

The `for` statement creates a loop that executes for the specified number of times. The syntax is as follows:

```
for (declaration; condition; increment)
     /* code */
next;
```

This statement executes the code as long as the *condition* returns true. This *condition* statement has the same form as described in 4.2.1a. The *declaration* statement instantiates a variable that will be used in the *condition* and that is incremented in the *increment* statement.

Note that if a variable is created in the `for` loop, then it will not be available for use after the loop is completed. Also, all `for` loops should end with a `next` statement to keep the iterations going.

4.4 `while`

The while statement block signifies a loop but without the added declaration and increment statements of the `for` loop. The syntax is as follows:

```
while (condition)
     /* code */
next;
```

This will execute the code block until the *condition* evaluates to false.

5.0 Methods

BSL contains built-in methods that allow quick and efficient analysis of data.

5.1 Usage

To use any method, the following syntax must be used:

method_name: parameters;

User can also create new methods within a script by the following syntax:

```
start method return_type method_name (parameters)
     /* code follows */
end method;
```

Once a method is declared, it can be accessed throughout the script. If a certain method does not return any value, then it will have a *return_type* of `void`. Users can also overload methods by just setting this user method to have the same name as the original method name as in:

```
start method return_type existing_method_name (parameters)
```

Note however, this will overwrite all previous declarations of this method for the duration of the current script. All references to the method will reference the overloaded method.

The position of the parameters is important and the calling statement should have the correct parameter types when passing them into a method.

5.2 Built-in Methods

5.2.1 `Align`

Align takes two Sequences as parameters and returns a new optimally aligned Sequence representing the first sequence with the second sequence. A local bitwise pair alignment algorithm based on the Smith-Waterman algorithm is used to perform the actual alignment (see T. F. Smith and M. S. Waterman, (1981) J. Mol. Biol. 147:195-197 for a detailed description of the algorithm and the reasoning behind it.). Pseudo code for the algorithm follows:

```
Align Part 1, The Matrix
input: Sequences S and T, gap penalty g,
output: matrix A
m = length of S
n = length of T

//initialize a matrix and set the extreme values
for i = 0 to m do
    A[i,0] = 0
for j = 0 to n do
    A[0,j] = 0

//fill in the matrix using the following rule
for i = 1 to n do
    for j = 1 to n do
        A[i,j] = max(A[i-1,j]+g, A[i-1,j-
1] + score(S[i],T[j]), A[i,j-1]+g)
return A
```

(This pseudo code was modified for BSL; the original code was obtained from
http://www.che.iitb.ac.in/faculty/sbn/423-557/lects/module01/010204-dynamic_programming.html)

Score basically returns a +2 if there is a match of the two blocks at that location or a -1 for a gap penalty or a -3 if there is a mismatch (meaning a gap is necessary). Once the matrix is finalized, we backtrack through the matrix starting from the largest value to zero, building the aligned sequences. The backtracking algorithm is as follows:

```
Align Part 2, The Backtrack
input: Sequences S and T, matrix A.
output: optimally aligned Sequence S
i = length of Sequence S
j = length of Sequence T
S_opt = ""
T_opt = ""
while (A[i,j]>0) do
```

```
        //there is gap in Sequence T
        if A[i,j] = A[i-1,j] + g then
            prefix S[j] to S_opt
            prefix '-' to T_opt
            i = i-1

        //Matching block symbols at this position
        elseif A[i,j] = A[i-1,j-1] + score(S[i],T[j]) then
            prefix S[i] to S_opt
            prefix T[j] to T_opt
            i = i-1
            j = j-1

        //There is a gap in Sequence S
        else   // A[i,j] = A[i,j-1] + g
            prefix T[j] to T_opt
            prefix '-' to S_opt
            j = j-1
    return S_opt
```

(Original code was obtained from http://www.che.iitb.ac.in/faculty/sbn/423-557/lects/module01/010204-dynamic_programming.html)

S_opt is final optimally aligned version of Sequence S that is returned to the user. This algorithm was chosen because of fast performance times on most inputs.

5.2.2 `Compare`

This method will compare two `Sequences` and returns a boolean value of `true` if they are equal i.e., if the two `Sequences` contain the same sequence of blocks and they are of the same *BSL_type*. If they are not equal in *BSL_type*, an error will be thrown statement an illegal comparison. If they are of the same type and have different values of sequences, then `false` is returned.

5.2.3 `Complement`

This method is specific to RNA and DNA sequences and will return errors for all other types of sequences inputted. Otherwise, Complement will return the complement sequence of the inputted RNA sequence or DNA sequence.

5.2.4 `Consensus`

The `Consensus` method takes two `Sequences` and after alignment of the sequences. This will make the best possible sequence which is the combination of the two inputted sequences. A new `Sequence` object is returned with the combination as it value.

The confidence attribute of the sequence object is important when dealing with consensus sequences. As the child sequence is created out of the parent sequence, the confidence value is

modified upon the data. It is up to the user to decide what the threshold of the confidence level will be before further analysis can be completed.

5.2.5 `Find`

Given two `Sequences`, `Find` will return the starting position of the first sequence within the second sequence if first exists in the second, or else a value of –1 will be returned.

5.2.6 `FindAll`

This method works just as `Find`, except `FindAll` returns an array of all starting positions of the first sequence within the second sequence. The search is conducted using regular expressions and all possible matches are included. If no matches are found then a –1 is returned.

5.2.7 `FindPrimer`

FindPrimer is a special implementation of the Find methods that tries to identify the first sequence within the second sequence using the following algorithm:

```
     Start search
          Find firstSequence in SecondSequence
          If Found return startingposition
          Else remove a buildingblock from the firstSequence and
try again
          If the first sequence get smaller than 4 blocks and not
found,
                    Return -1
```

5.2.8 `Print`

`Print` can print to the console or to a specified file. The values are passed in as parameters. There are two parameters; the first is value to be printed. The second parameter signifies the location to be printed to. If this value is not specified then, console is assumed to be the default location. To print to a file, the filename should be specified. Currently BSL only supports text file format (.txt). If the specified file does not exist, BSL will create the file and if the file does exist, the file is overwritten. Note that `Print` will move to the next line after printing the inputted value.

5.2.9 `Reverse`

`Reverse` will return a `Sequence` object with the value of the reversed sequence of the inputted `Sequence` object. For example, if the inputted sequence is “today”, then “yadot” is returned.

5.2.10 `Translate`

`Translate` takes two parameters. The first is Sequence object that needs to be translated. The second parameter describes the form that the Sequence should be translated into. Those forms are RNA, DNA, AMINOACID, and PROTEIN. There are rules as to how the translations are completed and only certain translations are allowed. The allowed translations are

RNA to DNA
DNA to RNA
DNA to PROTEIN
RNA to PROTEIN

Any other translations will cause an illegal translation error. `Translate` returns a new Sequence that represents translated sequence.

The rules that define translations are biological in nature. For instance, an amino acid is made of three DNA bases.

5.3 File Methods

These methods provide for manipulation of files from within BSL. Currently, this iteration of BSL only supports text files.

5.3.1 `AccessFile`

`AccessFile` takes the filename as parameter and returns an array of `Sequences` created from the data in the file. The required format for the text file is to have a sequence followed by a blank line and then another sequence and so on. Otherwise, each line in the file is used to create a new `Sequence` object and added to the array. If no sequences can be created, then an array of length zero is returned. If the file format is not .txt, then errors will be outputted.

BSL GRAMMAR

The following is the BSL grammar, shown in ANTLR syntax.

```
options { language="Java"; }


class BSL_Parser extends Parser;

options {
  k = 2;
  buildAST = true;
  exportVocab = BSL;
  defaultErrorHandler = false;
}

tokens {
  DECLS;
  MCALL;
  PROP;
  BSLSCRIPT;
  IFER;
  EXPER;
  ARGS;
}

bslscript
  :  (method)*  script  EOF!  {  #bslscript  =  #([BSLSCRIPT,  "bslscript"],
#bslscript); }
  ;

method
  : "start"! "method"^ type methodName LPAREN! (args)? RPAREN!
      script (returnStatement)?
    "end"! "method"! SEMICOLON!
  ;

methodName
  : ID
  ;

args
  : type ID (COMMA! type ID)*
    { #args = #([ARGS, "args"], #args); }
  ;

returnStatement
  : "return"^ (ID | constant) SEMICOLON!
  ;

script
  : (expression SEMICOLON!)+
  ;

expression
```

```
  : (declaration | statement | forBlock | ifBlock | whileBlock)
    { #expression = #([EXPER, "exper"], #expression); }
  ;

declaration
  : "new"! type ID (dataStructureAccess)? (typedecl)? (assgnValue)?
    { #declaration = #([DECLS, "decls"], #declaration); }
  ;

typedecl
  : "type"^ bioType
  ;

assgnValue
  :  EQUALS^  ((ID  (dataStructureAccess)?  (methodCall  |  sequenceCalls  |
property)?) | operation)
  ;

statement
  :   ((ID (dataStructureAccess)? (assgnValue | methodCall | sequenceCalls |
property)?) | operation | atomicStatement)
  ;

methodCall
  : COLON! (parameters)?
    { #methodCall = #([MCALL, "mcall"], #methodCall); }
  ;

parameters
  :   (ID  (dataStructureAccess  |  sequenceCalls)?  |  constant)  (COMMA!  (ID
(dataStructureAccess | sequenceCalls)? | constant))*
  ;

property
  : PERIOD! ID
    { #property = #([PROP, "prop"], #property); }
  ;

ifBlock
  : "if"^ operation "then"! ifinside (elseif)* (elsePart)? "end"! "if"!
  ;

ifinside
  : (expression SEMICOLON!)*
    { #ifinside = #([IFER, "ifer"], #ifinside); }
  ;


elseif
  : "elseif"^ operation "then"! ifinside
  ;

elsePart
  : "else" ifinside
  ;

whileBlock
```

```
  : "while"^ operation ifinside  "next"!
  ;

forBlock
  : "for"^ forConditional ifinside  "next"!
  ;

forConditional
  :   LPAREN!   (declaration)?   SEMICOLON!   (operation)?   SEMICOLON!   (ID
assgnValue)? RPAREN!
  ;

bioType
  : "DNA" | "RNA" | "AMINOACID" | "PROTEIN" | "STRING" | "UNKNOWN"
  ;

operation
  : orexpr
  ;

orexpr
  : andexpr (PARALLEL^ andexpr)*
  ;

andexpr
  : compOp (DOBAND^ compOp)*
  ;

compOp
  : addexpr ( (DOBEQUAL^ | LESSTHAN^ | GREATERTHAN^ | LEQUAL^ | GEQUAL^ |
NEQUAL^) addexpr )*
  ;

addexpr
  : mulexpr ( (PLUS^ | DASH^) mulexpr)*
  ;

mulexpr
  : unaryexpr ( (STAR^ | SLASH^ | "mod"^) unaryexpr )*
  ;

unaryexpr
  : LPAREN! (DASH^ unaryexpr | statement) RPAREN! | constant
  ;

type
  : primitive | bslType | "void"
  ;

primitive
  : "boolean" | "byte" | "char" | "int" | "short"
  ;

bslType
  : "BuildingBlock" | "Sequence"
  ;
```

```
constant
  : constantLiteral
  | Number
  ;

constantLiteral
  : "true"
  | "false"
  | stringConstant
  ;

stringConstant : StringConstant ;

atomicStatement
  : "exit"
  | "break"
  ;

dataStructureAccess
  : LBRACKET^ (Number (COMMA! Number)?)? RBRACKET!
  ;

sequenceCalls
  : LPAREN^ Number (COMMA! Number)? RPAREN!
  ;



class BSL_Lexer extends Lexer;

options {
  k = 2;   // set lookahead to 2
  charVocabulary = '\3'..'\377';   // Handle all 8-bit characters
  exportVocab = BSL;   // Export these token types for tree walkers
  testLiterals = false;   // Disable checking every rule against keywords
}

PERIOD : '.';
PLUS : '+';
DASH : '-';
SLASH : '/';
STAR : '*';
PARALLEL : "||";
DOBAND : "&&";
LESSTHAN : '<';
GREATERTHAN : '>';
COMMA : ',';
DOBEQUAL : "==";
EQUALS : '=';
SEMICOLON : ';';
COLON : ':';
LPAREN : '(';
RPAREN : ')';
LBRACKET : '[';
RBRACKET : ']';
LEQUAL : "<=";
GEQUAL : ">=";
```

```
NEQUAL : "!=";

ID options { testLiterals = true; }
  : ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
  ;

//  Numbers (Integers)
Number
  : ('0'..'9')+
      ( '.' ('0'..'9')*
      | /* empty */
      )
  ;


// Strings
StringConstant
  : '"'! ( ~('"' | '\n') | ('"'! '"'))* '"'!
  ;

Whitespace
  : (' ' | '\t' | '\f')+
    { $setType(Token.SKIP); }
  ;

Newline
  : ('\n' | "\r\n" | '\r')
    { $setType(Token.SKIP); }
  ;


/**
  All comments
*/

Comment
  : "/*" ( options {greedy=false;}: // Prevents .* from eating the whole file
            (
              ('\r' '\n') => '\r' '\n'    { newline(); }
              | '\r'                { newline(); }
              | '\n'                { newline(); }
              | ~( '\n' | '\r' )
            )
          )*
      "*/"
    { $setType(Token.SKIP); }
  ;
```

Chapter 3

BSL Tutorial

Biological Scripting Language provides a framework that can be used to analyze and manipulate biological data. This tutorial illustrates some of the core functionalities of the language and will help the programmer jumpstart the process of application development in BSL.

In order to write your first script in BSL, you require the Java 2 Platform, Standard Edition and consult the installation instructions. **You will also need the BSL Platform, which may be downloaded and installed from the BSL website**. Any simple text editor may be used to write a BSL script.

The examples below have been given line numbers to allow easy reference. These line numbers should not be included when writing an actual script.

1.1 A First Example

Example 1.1 illustrates the famous "Hello World" example in a slightly more complicated form written in BSL. The code given below prints "Hello World" numerous times as specified in the "for" loop. The notation `int j = 10;` declares an integer "`j`" which is initialized to the value 10. "`for(int i = 0; (i) < (j); i=(i)+1)`" then declares another integer "`i`" within the "`for`" loop and initializes it to 0. Within this piece of code, the condition whether `i` is less than `j` is tested after the first semicolon. If the condition holds, it prints the phrase "Hello World". It then increments `i` according to the syntax `i=(i)+1` and repeats the `for` loop. The "`next`" indicates repetition of the `for` loop iteration. Once the condition `i < j` does not hold, the `for` loop is broken and the script ends. `print: "Hello World"` prints the phrase Hello World to the command line by default. To be able to print it to file, one would use the same syntax but add ", `<filename>`" after the call "`print:` ".

```
[1] new int j = 10;
[2]
[3] for(new int i = 0; (i) < (j); i = (i) + 1)
[4]     new Sequence printout = "Hello World";
[5]     Print: printout;
[6] next;
```

1.2 Compiling and Running BSL scripts

Once a script in BSL has been written, it should be saved as "<filename>.bsl", i.e. with ".bsl" as the file type. The program can then be compiled using BSL. Simply type:

$ java BSL_TreeParser <filename>.bsl

If everything goes well, and there are no errors in the program, this will produce a java bytecode file called <filename>.class. In order to run the compiled file, type:

$ java <filename>

1.3 Second Example

The example below illustrates the use of `for` loops, arrays, and the **Find** method in BSL. First, a new sequence object "sequence1" is created. Then, an array of sequences "file1_sequence" is created from a text file "RNAsequence.txt". The **AccessFile** method returns an array of sequences that are contained in the file. The length of the sequence array "file1_sequence" is stored as an `int` in "file1_length". In the following `for` loop, the sequence array is traversed (from 0 until its length) printing the type and the actual sequence for each iteration.

The next part of the example illustrates the use of the **Find** method. First, we create a sequence "testSequence" of **type DNA**. Then we assign a string value to our test sequence. Then, we have a `for` loop where we first check if our test sequence matches any part of a sequence in our file1_sequence array. If a match is found, we print out the test sequence followed by the index of the array that corresponds to the proper sequence in the array that contains our test sequence, and the starting position (index) within the array sequence which identifies the match.

```
[1] new Sequence sequence1;
[2]
[3] Sequence[] file1_sequence = AccessFile: "RNAsequence.txt";
[4]
[5] new int file1_length = file1_sequence.length;
[6]
[7] for(new int i = 0; (i) < (file1_length); i = (i) + 1;)
[8]     Print: file1_sequence[i].type;
[9]     Print: file1_sequence[i];
[10] next;
[11]
[12] new Sequence testSequence type DNA;
[13] testSequence = "ACCTGA";
[14]
[15] for(new int i = 0; (i) < (file1_length); i= (i) + 1)
[16]    if((Find: testSequence, file1_sequence[i]) != -1) then
[17]            new Sequence printout = "Test sequence" + (testSequence) + " is present
[18]                                    within file sequence number " + (i) + "at position" +
[19]                                    (Find:testSequence1);
[20]            Print: printout , file1_sequence[i];
[21]    end if;
[22] next;
```

1.4 Third Example

```
[1] start method boolean typeComparison(Sequence a, Sequence b)
[2]
[3]    if((a.type) == (b.type)) then
[4]            Print: "type match for sequences";
[5]            return true;
[6]
[7]    else if((a.type) != (b.type))
[8]            Print: "type does not match for sequences";
[9]            return false;
[10]
[11]   end if;
[12]
[13] end method;
[14]
[15] new Sequence[] sequenceArray1 = AccessFile:"DNAsequence.txt";
[16] new int arrayLength1 = sequenceArray1.length;
[17]
[18] new Sequence[] sequenceArray2 = AccessFile: "sequenceFile.txt";
[19] new int arrayLength2 = sequenceArray2.length;
[20]
[21] new int count = 0;
[22]
[23] for(new int i = 0; (i) < (arrayLength1); i= (i) + 1)
[24]   if((i) < (arrayLength2)) then
[25]           if((typeComparison: sequenceArray1[i], sequenceArray2[i]) == true) then
[26]                   if((Compare: sequenceArray1[i], sequenceArray2[i]) == true) then
[27]                           count = (count) + 1;
[28]                   end if;
[29]           end if;
[30]   else
[31]           break;
[32]   end if;
[33] next;
[34]
[35] Print: "number of matching sequences: " + count;
```

Example 1.4 illustrates how a programmer may create his or her own methods that can be called by the script later on. On running program 1.4, the program actually begins at line 15, after the end of the methods declared by the programmer. At line 15, an array of `Sequence` objects, called `sequenceArray1`, is returned on instantiating the method "`AccessFile:` " and by giving the name of the file to access. The array can be handled by its name `sequenceArray1` from here onwards. Line 16 declares the variable `arrayLength1` of type integer and it has the length of the array `sequenceArray1` as its value. Therefore, `<array name>.length` can be used to obtain the length of an array. At line 21 another variable `count` is declared with an initial value of 0. If no initial value is declared, it is automatically assumed to be 0 in case of integers and doubles. However, in the case of Strings and chars, the value is assumed to be null. Line 23 starts the `for` loop with initial value of the counter `i` set to 0. The `for` loop checks the condition whether `i` is less than the `arrayLength1` and keeps incrementing variable `i` until the condition does not hold. Line 24 illustrates a simple `if` statement. On line 25, the `if` statement includes a call to the method that was declared earlier in the script. It calls the method `typeComparison` with two `Sequence` objects as parameters. The method returns a `boolean` value, which is checked. If `true`, the statements within the `if` block are executed. Otherwise, it skips to the `else` statement on line 30 which simply breaks out of the `for` loop and skips to line 34. The program ends with a call to `print` and prints the value of `count`. Line 26 within the `if` statement compares the two sequences which at this point are known to be of the same type and check if the two sequences are equivalent. It returns a

`boolean` value of `true` to signify equivalent sequences and `false` otherwise. The method declaration in the beginning of the script (at line 1) explains that a method that returns `boolean` is being declared with the name `typeComparison`. If the method does not return anything, the `boolean` must be replaced by the keyword `void` to signify this. The method declaration also delineates the parameters that must be supplied on method invocation. While on invocation the parameters being sent in may be named anything, they will be referred to as `a` and `b` within the method. Please note that the types of the parameters and their order matters when invoking the method. Within the method, the type of the two `Sequence` objects is checked and if same, "`type match for sequences`" is printed and a `boolean` value of `true` is returned. Otherwise `false` is returned. Once something is returned, the method ends immediately. The `end method` on line 13 signifies the end of the method.

1.5 Fourth Example

This code takes in the first two sequences from a specified file, aligns them, and translates the resulting sequence into one of type RNA. At first, multiple sequences of *Sequence* object are created along with a sequence array "seqList". Then, the array is populated with sequences from the "SeqBldr.txt" file. Before we instantiate our two sequences, we have to make sure that the array is not empty and contains more than one sequence. If these conditions do not hold, then the program is terminated. If two or more sequences exist, then the first two are chosen, and are aligned with the **align** method, which returns the aligned sequence (for a detailed explanation of how **align** is structured please see the Language Reference Manual). The new sequence is then translated to RNA with the **translate** method, which returns the specified type of the sequence (for more details please refer to the LRM). At the end, the aligned sequence is printed out to a file "Align_out.txt" along with its original and translated types.

```
[1] new Sequence a;
[2] new Sequence b;
[3] new Sequence c;
[4] new Sequence trans;
[5] new Sequence[] seqList;
[6]
[7] seqList = AccessFile: "SeqBldr.txt";
[8]
[9] if ((seqList.length) == 0) then
[10]    Print: "File doesn't contain any sequences! Exiting now…";
[11]    exit;
[12] elseif ((seqList.length) < 2)
[13]    Print: "Only one sequence is found! Exiting now…";
[14]    exit;
[15] else
[16]    a = (seqList[0]);
[17]    b = (seqList[1]);
[18] endif;
[19]
[20] c = Align: a, b;
[21] trans = Translate: c, "RNA";
[22] new Sequence printout =  "Sequence " + (c) + " of type " + (c.type) +
[23]                          " has been translated to type: " + (trans.type);
[24] Print: printout, "Align_out.txt";
```

1.6 Final Reminders

All objects that used in statement such as conditionals and operations should be enclosed in parenthesis as in

i = (i); i = (i) + 1; i = (i) + (a.length);

Method calls and properties are also the under same consideration.

1.7 Some Do's and Don'ts of BSL

(i) BSL is a scripting language and as such, does not contain the main() statement.

(ii) When declaring a variable, its type must be specified (e.g. int, sequence, etc.)

(iii) Declaration is done with the keyword *new*.

(iv) Usage for keyword *print: "filename", "<input string here>";*

(v) All user defined methods end with: *end method <method name>*

(vi) If the file does not contain any sequences, **AccessFile** will return an array of length 0.

(vii) Sequence should have a type that could be declared or, if not specified, left as NULL. The type of a sequence can be accessed by *"<sequence name>".type*.

(viii) User can force an exit of the program by using the exit statement. Break is for loops only.

Chapter 4

BSL Project Plan

1.1 Team Responsibilities

Specific tasks will be assigned to each team member to complete whilst abiding by the tentative project timeline. These tasks will largely be the responsibility of its corresponding team member, but ultimately everyone will be held responsible for the language's working implementation.

Team Leader: Amna Qaiser
Team Nitpicker: Jared Eng
Team Jack-of-Useless-Trades: Jay Kota
Team Jester: Igor Marfin

1.2 Project Timetable

This tentative schedule of events was set-up as a guideline for our team to follow as closely as possible. We abided by this agenda for the most efficient and effective implementation of our language, BSL.

02/18/03 Language whitepaper, core language features defined
03/04/03 Development environment and code conventions defined
03/27/03 Language Reference Manual, grammar complete
04/23/03 Parser complete
05/03/03 Code generation complete
05/07/03 Error recovery complete
05/11/03 Presentation complete
05/13/03 Final Project Presentation

1.3 Software Development Environment

BSL will be a public domain scripting language with full source distribution. It will be developed on Windows XP using Java SDK 1.4.1. The parser and lexer will be developed using ANother Tool for Language Recognition (ANTLR) 2.7.1, a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions. Source code will be controlled using Concurrent Versions System (CVS) - source files being modified will be set aside as back-up.

1.4 Project Log

The group had biweekly meetings with the TA, Michael E Locasto on Mondays at 8.00 pm. In addition to the meetings with the TA, we met regularly three times a week on Mondays, Wednesdays and Sundays.

01/26/03 Project bainstorming session: Decided on the problem space we wanted to work on.
01/29/03 Project launch: Chose genetic sequencing and manipulation as the project goal. Began research
02/02/03 Focused on existing languages dealing with genetic sequencing like BioPerl
02/03/03 Met with Michael, language finalized
02/05/03 Began work on white paper specifications, christianed AJJI
02/09/03 First draft for white paper completed
02/10/03 Met with Michael to give updates on white paper
02/12/03 Birth of BS….L: Biological Scripting Language
02/16/03 White paper completion
02/19/03 Code conventions, LRM tasks broken up
02/26/03 Revision of LRM first draft
03/05/03 Development environment and code conventions defined
03/09/03 Outline of the Language Reference Manual finalized
03/23/03 Finished Language Reference Manual
03/24/03 Met with Michael to go over LRM and found out that tutorial was NOT due!
03/26/03 Finishing touches on the LRM; started first draft of grammar
04/02/03 Revision of ANTLR grammar
04/06/03 Libraries for BSL decided on and tasks split up
04/09/03 Discovered Esterel grammar and started making changes to ours…
04/12/03 Parser and lexer, first working version
04/14/03 Met with Michael and discussed grammar and problems encountered
04/23/03 Testing of lexer and parser, Code generation phase start
05/03/03 Further debugging of lexer, parser, and grammar
05/05/03 – 05/10/03 Worked daily on code generation (approx. 3 hrs. per day)
05/10/03 Code generation completion
05/11/03 Final Project Presentation finalized
05/12/03 Last minute testing and debugging, Final Report and Presentation touch-up
05/13/03 PLT in-class Presentations!
05/14/03 Meeting with Prof. Edwards

Chapter 5

BSL Architectural Design

1.1 Architecture

BSL uses a modular approach to create and run a program for the user. Each component does some amount of work before passing the BSL script to the next component. The final result will be a program that performs the requested actions in the JAVA runtime environment. The components involved in the compilation of a BSL script are: lexer, parser, code generator and java compiler. The relationship is diagrammed in figure below. As input, the BSL compiler takes one BSL script file that is a file with the extension '.bsl' and contains BSL specific code. This file is translated into a .java file that is compiled using the java compiler. The output of the java compiler is the run using the JVM. The lexer and parser are generated using ANTLR. The code generator is written using java and contains the logic to create the java program based on the data sent by the parser. The BSL error handler is also built into the code generator, which will check to make sure that the correction type of actions are performed in the program. Whenever a rule is broken, an error message will be printed to the screen and the compilation is exited. These the modules are shielded from the details of the each other's construction.

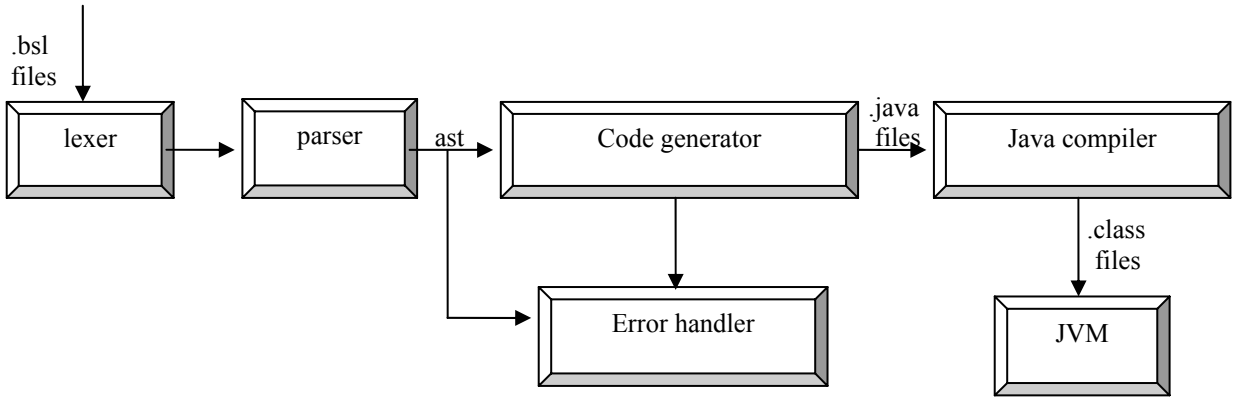

figure 1. Block diagram for the BSL compiler.

The entry point is BSL_TreeParser.Main. The main() method processes the BSL file pointed to by the name inputted by the user. The method creates a parser, which creates a lexer. This lexer reads in the file and removes unnecessary data from the file, such as comments and white space. Along with this process, the lexer also creates tokens for each item in the script. Each token then is passed along to the parser, where semantics of the script is checked. The interface between the lexer and the parser is get_next_token(). The parser also acts in two ways; while acting as a semantic checker, it also creates an abstract syntax tree. The main() method

will retrieve the tree once the entire file is completely parsed into this tree by using the method getAST().

The BSL_TreeParser traverses through the abstract syntax tree and generates the necessary Java code. While this code is generated, the semantics of the script are checked for accuracy. Any user written methods are added to the main method symbol table allowing for usage in the script. Methods that the user wants to rewrite are also taken care of by BSL_TreeParser. The original method calls are replaced with the new definitions in the symbol table. Variables are also allotted a symbol table of their own. Each method has static scope and only definitions declared in that method are allowed to be used. To use variables in a method, the were created in main should be passed in as parameters.

Our language enforces the Chain of Responsibility pattern which allows a number of classes to attempt to handle a request, without any of them knowing about the capabilities of the other classes. It provides a loose coupling between these classes; the only common link is the information that is passed between them. The information is passed along until one of the classes can handle it.

With the Chain of Responsibility pattern in mind, the BSL_TreeParser, BSL_Parser and BSL_Lexer all participate in the class inheritance structure. Information is passed up the inheritance chain. The lexer receives the BSL code file and erases the whitespace, generating the necessary tokens to be parsed through. The parser will begin to obtain each token from the lexer. If a syntax error exists, the parser invokes the error handler. The error handler passes the appropriate error message and prints it to the screen. As each correct token is passed in, it starts building the abstract syntax tree based on the tokens' semantics. The parser passes the tokens to BSL_TreeParser which semantically verifies the tokens in the abstract syntax tree. The BSL class file is translated into a stand-alone Java file when BSL_TreeParser is invoked.

Below is Figure 2, validating the relationship within the Chain of Responsibility.

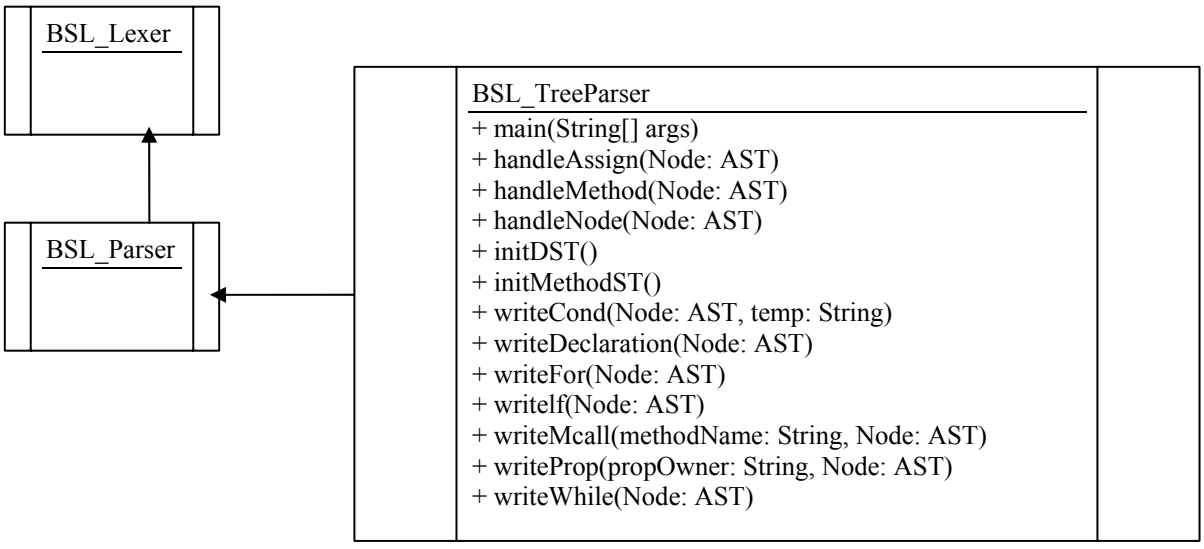

Figure 2: Chain of Responsibility within BSL's infrastructure

The following are samples of the code from the compiler, mainly the BSL_TreeParser. This is the while loop in the `main()` that traverses the tree. At each node, the internal method `handleNode(currentNode)` is called to handle this branch.

```
while (currentNode != null) {
        if (!(currentNode.getText()).equals("method") && !wroteMainMethod) {
          // no method calls
          bslScriptFile +=  "\tpublic static void main(String[] args) {\n";
          wroteMainMethod = true;
        }
        else {
          //handle this method
        }

         //handle the node
         handleNode(currentNode);
         lineCount ++;
         currentNode = currentNode.getNextSibling();
      }
```

This is a sample of how each branch is checked to determine the type of statement that is written. Depending on the identifying value, the specific function is called to write the equivalent java code.

```
if (statementType.equals("decls")) {
      writeDeclaration(node);
    }
    else if (statementType.equals("if")) {
      writeIf(node);
    }
    else if (statementType.equals("for")) {
      writeFor(node);
    }
    else if (statementType.equals("while")) {
      writeWhile(node);
    }
```

This is a sample of the code that interprets the if statement block written by the user in the bsl script. Private methods are called to write the conditions and also the internal statements of the if block.

```
    bslScriptFile += writeCond(ifNode, "") + ") {\n";
    AST elsePart = null;
    //write body
    if (ifNode.getNextSibling() != null) {
      //check to see if else or else if exists
      if ((ifNode.getNextSibling()).getNextSibling() != null) {
        elsePart = (ifNode.getNextSibling()).getNextSibling();
      }
      ifNode = (ifNode.getNextSibling()).getFirstChild();
      while (ifNode != null) {
        handleNode(ifNode);
```

```
        lineCount ++;
        ifNode = ifNode.getNextSibling();
    }
```

Below is a sequence diagram demonstrating the flow of calls made within BSL. Note that after the BSL_TreeParser receives the abstract syntax tree, it builds the stand-alone Java file.

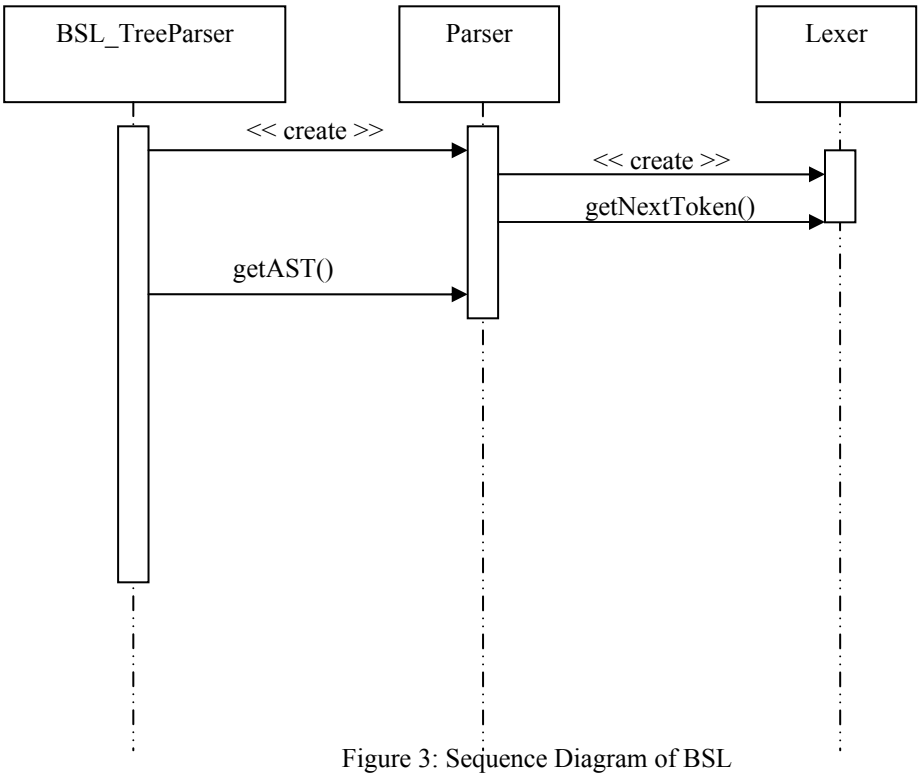

Figure 3: Sequence Diagram of BSL

1.2 Runtime Environment

After the standalone Java file is created, the Java runtime environment handles the generated Java file. To run the Java class file, BSL classes are needed (our pseudo-runtime environment). The BSL method calls in the user script are passed to these classes. These classes perform the necessary actions. For instance, if the user calls Align, the Aligner handles the call and returns the pertinent results.

Chapter 6

BSL Testing Plan

1.0 Goals

Our testing plan is designed to help us determine if the BSL compiler is doing any of the things that we expect it to do. We decided to test thoroughly at each milestone to ensure that individual pieces were working correctly before combining everything together. In designing our test methods we tried to account for the modularized environment of BSL to fully exploit the functionality of our design. After the testing process, we want the user to feel comfortable using our system and be confident that the results from the program are suitable for future analysis.

1.1 Hypothesis

Using a variety of different types of coding styles and schemes along with detailed testing phases will allow for an easy and efficient scripting language to work with.

1.2 Methods

The BSL programming group will split up the testing cycle into two phases: code parsing and tree building, and code generation and output testing. The two phases will be sequential as we can't start testing the code until our tree is fully built. At each phase we hope to eliminate all the bugs particular to that phase.

1.2.1 Phase I

Phase I deals with the initial development of the compiler and the pieces that make up the BSL runtime environment, including the objects that should be available during runtime of a script. Communication was an important part of this phase since the grammar has to co-exist with the objects that handle the requests made using the grammar in future BSL scripts.

The implementation of our compiler began with creation of the lexer and the parser using the ANTLR software. The grammar represented the first major milestone of our testing phase since it was supposed to directly correspond to the functionality described in our white paper. When testing the grammar we were careful in making sure to eliminate all warnings of non-determinism to simplify the task of the code generation. Each revision of the grammar brought us closer to this goal.

After we had the grammar, lexer, and the parser, we moved on to code generation. We divided our code generation so that each team member was responsible for writing classes that would implement the basic method functionality of BSL. The internal design of these classes was left to the discretion of the author, but the exact function of the classes was thoroughly

discussed and decided upon by the entire group. This set up conventions that allowed for future combination of all the objects without violating functionality of any particular object.

After our classes we written, we tested each one of them separately to ensure proper operation before combining them with the rest of our design. This process allowed us to discover many bugs that would otherwise be very hard to track down later on during the implementation.

1.2.2 Phase II

The next phase of testing involved bringing together all of the separate objects to work in unison. This part didn't target standalone classes and required different test cases from the initial code generation in the previous phase. Testing was run in parallel with the production of the code generator. Once the lexer and parser were finished, our next milestone was to output a java file based on the BSL script. To accomplish this task, our compiler had to traverse the abstract syntax tree produced by the parser and output the appropriate java file.

To simplify the testing design we wrote our BSL script in a text file and used our Tree Parser to read the file, and build the corresponding java code. At the end, we output the completed java file and run the java compiler on it. As a result of running the tree parser on our test program we obtain a functional (and runnable) java file that implements the BSL script written.

1.3. Conclusion

Phase I tested the validity of the syntax of a BSL script, and in the next phase our task was to check its semantics. We also examined the resulting output java file and determined whether that file performs the task expected by the user.

We found that splitting up testing into two phases was beneficial because it reduced the amount of testing in any particular phase and gave us some time to reflect on each phase. We were also able to start troubleshooting early and were able to find many bugs that would prove to be more difficult to find once the entire project was compiled together.

We were able to complete the two testing phases described above and obtain a functional java program from various BSL scripts. The next phase of our project would be to further test our compiler for all ranges of functionality and deal with various kinds of erroneous user input.

Chapter 7

BSL Lessons Learned

Our group learned an immense amount about collaborating and working together as a team. During the conception of our language, we exchanged all of our ideas about BSL including concerns that could potentially be problematic. This helped us lay everything out for our future plans. During our meetings, we tried to be as efficient and effective as possible by dividing all of the work up pretty evenly amongst all of the group members to be certain that everyone had a role and played a part in the creation of BSL. Our advice to future teams would simply be to start early. A standard should also be set early on in the semester about how often to meet. Our team decided to meet bi-weekly and that was a great idea because we met regularly and stuck to it. As our work together got more intense, we increased the frequency of our meetings together to ensure the successful completion of our project.

It is also important to keep track of the scope of the project. We started out with a slightly higher goal, but had to scale it down due to time constraints. Therefore, we believe that it is always a good idea to keep the time limitations in mind and to set concrete milestones. Food was also a major concern that we did not address beforehand, which lead to many minutes of digretion and despair among the teammates. An abundant supply of caffine and other stimulants kept us focused and on track for the next concrete milestone.

However, on a more serious note, our group benefited from the fact that we discussed and premeditated many details that helped the implementation phase go smoother, so this is something we definitely recommend. And, we also found it helpful to have (and actually attend) scheduled sessions with our TA. We were also aided by the fact that most of our team mates had previous experience with the topic we were working on.

We wish future teams the best of luck, long life, and prosperity!

Chapter 8

BSL Appendix

```
// $ANTLR 2.7.2: "bslGrammar_2.g" -> "BSL_Parser.java"$
/*
 * BSLTokenTypes created by antlr.
 * @author ANTLR
 * cunix id: antlr
 */

public interface BSLTokenTypes {
      int EOF = 1;
      int NULL_TREE_LOOKAHEAD = 3;
      int DECLS = 4;
      int MCALL = 5;
      int PROP = 6;
      int BSLSCRIPT = 7;
      int IFER = 8;
      int EXPER = 9;
      int ARGS = 10;
      int LITERAL_start = 11;
      int LITERAL_method = 12;
      int LPAREN = 13;
      int RPAREN = 14;
      int LITERAL_end = 15;
      int SEMICOLON = 16;
      int ID = 17;
      int COMMA = 18;
      int LITERAL_return = 19;
      int LITERAL_new = 20;
      int LITERAL_type = 21;
      int EQUALS = 22;
      int COLON = 23;
      int PERIOD = 24;
      int LITERAL_if = 25;
      int LITERAL_then = 26;
      int LITERAL_elseif = 27;
      int LITERAL_else = 28;
      int LITERAL_while = 29;
      int LITERAL_next = 30;
      int LITERAL_for = 31;
      int LITERAL_DNA = 32;
      int LITERAL_RNA = 33;
      int LITERAL_AMINOACID = 34;
      int LITERAL_PROTEIN = 35;
      int LITERAL_STRING = 36;
      int LITERAL_UNKNOWN = 37;
      int PARALLEL = 38;
      int DOBAND = 39;
      int DOBEQUAL = 40;
```

```
	int LESSTHAN = 41;
	int GREATERTHAN = 42;
	int LEQUAL = 43;
	int GEQUAL = 44;
	int NEQUAL = 45;
	int PLUS = 46;
	int DASH = 47;
	int STAR = 48;
	int SLASH = 49;
	int LITERAL_mod = 50;
	int LITERAL_void = 51;
	int LITERAL_boolean = 52;
	int LITERAL_byte = 53;
	int LITERAL_char = 54;
	int LITERAL_int = 55;
	int LITERAL_short = 56;
	int LITERAL_BuildingBlock = 57;
	int LITERAL_Sequence = 58;
	int Number = 59;
	int LITERAL_true = 60;
	int LITERAL_false = 61;
	int StringConstant = 62;
	int LITERAL_exit = 63;
	int LITERAL_break = 64;
	int LBRACKET = 65;
	int RBRACKET = 66;
	int Whitespace = 67;
	int Newline = 68;
	int Comment = 69;
}
```

```
// $ANTLR 2.7.2: "bslGrammar_2.g" -> "BSL_Lexer.java"$
/*
 * Lexer parses the bsl script and removes extraneous characters while
 * creating tokens.
 * @author ANTLR
 * cunix id: antlr
 */

import java.io.InputStream;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;
import antlr.TokenStreamRecognitionException;
import antlr.CharStreamException;
import antlr.CharStreamIOException;
import antlr.ANTLRException;
import java.io.Reader;
import java.util.Hashtable;
import antlr.CharScanner;
import antlr.InputBuffer;
import antlr.ByteBuffer;
import antlr.CharBuffer;
import antlr.Token;
import antlr.CommonToken;
import antlr.RecognitionException;
import antlr.NoViableAltForCharException;
import antlr.MismatchedCharException;
import antlr.TokenStream;
```

```
import antlr.ANTLRHashString;
import antlr.LexerSharedInputState;
import antlr.collections.impl.BitSet;
import antlr.SemanticException;

public class BSL_Lexer extends antlr.CharScanner implements BSLTokenTypes,
TokenStream
 {
public BSL_Lexer(InputStream in) {
      this(new ByteBuffer(in));
}
public BSL_Lexer(Reader in) {
      this(new CharBuffer(in));
}
public BSL_Lexer(InputBuffer ib) {
      this(new LexerSharedInputState(ib));
}
public BSL_Lexer(LexerSharedInputState state) {
      super(state);
      caseSensitiveLiterals = true;
      setCaseSensitive(true);
      literals = new Hashtable();
      literals.put(new ANTLRHashString("DNA", this), new Integer(32));
      literals.put(new ANTLRHashString("start", this), new Integer(11));
      literals.put(new ANTLRHashString("short", this), new Integer(56));
      literals.put(new ANTLRHashString("boolean", this), new Integer(52));
      literals.put(new ANTLRHashString("for", this), new Integer(31));
      literals.put(new ANTLRHashString("if", this), new Integer(25));
      literals.put(new ANTLRHashString("method", this), new Integer(12));
      literals.put(new ANTLRHashString("type", this), new Integer(21));
      literals.put(new ANTLRHashString("STRING", this), new Integer(36));
      literals.put(new ANTLRHashString("UNKNOWN", this), new Integer(37));
      literals.put(new ANTLRHashString("while", this), new Integer(29));
      literals.put(new ANTLRHashString("char", this), new Integer(54));
      literals.put(new ANTLRHashString("BuildingBlock", this), new
Integer(57));
      literals.put(new ANTLRHashString("break", this), new Integer(64));
      literals.put(new ANTLRHashString("end", this), new Integer(15));
      literals.put(new ANTLRHashString("then", this), new Integer(26));
      literals.put(new ANTLRHashString("AMINOACID", this), new Integer(34));
      literals.put(new ANTLRHashString("mod", this), new Integer(50));
      literals.put(new ANTLRHashString("else", this), new Integer(28));
      literals.put(new ANTLRHashString("byte", this), new Integer(53));
      literals.put(new ANTLRHashString("void", this), new Integer(51));
      literals.put(new ANTLRHashString("true", this), new Integer(60));
      literals.put(new ANTLRHashString("Sequence", this), new Integer(58));
      literals.put(new ANTLRHashString("RNA", this), new Integer(33));
      literals.put(new ANTLRHashString("exit", this), new Integer(63));
      literals.put(new ANTLRHashString("next", this), new Integer(30));
      literals.put(new ANTLRHashString("PROTEIN", this), new Integer(35));
      literals.put(new ANTLRHashString("elseif", this), new Integer(27));
      literals.put(new ANTLRHashString("int", this), new Integer(55));
      literals.put(new ANTLRHashString("false", this), new Integer(61));
      literals.put(new ANTLRHashString("return", this), new Integer(19));
      literals.put(new ANTLRHashString("new", this), new Integer(20));
}
```

```
public Token nextToken() throws TokenStreamException {
      Token theRetToken=null;
tryAgain:
      for (;;) {
            Token _token = null;
            int _ttype = Token.INVALID_TYPE;
            resetText();
            try {   // for char stream error handling
                  try {   // for lexical error handling
                        switch ( LA(1)) {
                        case '.':
                        {
                              mPERIOD(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case '+':
                        {
                              mPLUS(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case '-':
                        {
                              mDASH(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case '*':
                        {
                              mSTAR(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case '|':
                        {
                              mPARALLEL(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case '&':
                        {
                              mDOBAND(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case ',':
                        {
                              mCOMMA(true);
                              theRetToken=_returnToken;
                              break;
                        }
                        case ';':
                        {
                              mSEMICOLON(true);
                              theRetToken=_returnToken;
                              break;
```

```
}
case ':':
{
      mCOLON(true);
      theRetToken=_returnToken;
      break;
}
case '(':
{
      mLPAREN(true);
      theRetToken=_returnToken;
      break;
}
case ')':
{
      mRPAREN(true);
      theRetToken=_returnToken;
      break;
}
case '[':
{
      mLBRACKET(true);
      theRetToken=_returnToken;
      break;
}
case ']':
{
      mRBRACKET(true);
      theRetToken=_returnToken;
      break;
}
case '!':
{
      mNEQUAL(true);
      theRetToken=_returnToken;
      break;
}
case 'A':  case 'B':  case 'C':  case 'D':
case 'E':  case 'F':  case 'G':  case 'H':
case 'I':  case 'J':  case 'K':  case 'L':
case 'M':  case 'N':  case 'O':  case 'P':
case 'Q':  case 'R':  case 'S':  case 'T':
case 'U':  case 'V':  case 'W':  case 'X':
case 'Y':  case 'Z':  case 'a':  case 'b':
case 'c':  case 'd':  case 'e':  case 'f':
case 'g':  case 'h':  case 'i':  case 'j':
case 'k':  case 'l':  case 'm':  case 'n':
case 'o':  case 'p':  case 'q':  case 'r':
case 's':  case 't':  case 'u':  case 'v':
case 'w':  case 'x':  case 'y':  case 'z':
{
      mID(true);
      theRetToken=_returnToken;
      break;
}
case '0':  case '1':  case '2':  case '3':
case '4':  case '5':  case '6':  case '7':
```

```
case '8':  case '9':
{
      mNumber(true);
      theRetToken=_returnToken;
      break;
}
case '"':
{
      mStringConstant(true);
      theRetToken=_returnToken;
      break;
}
case '\t':  case '\u000c':  case ' ':
{
      mWhitespace(true);
      theRetToken=_returnToken;
      break;
}
case '\n':  case '\r':
{
      mNewline(true);
      theRetToken=_returnToken;
      break;
}
default:
      if ((LA(1)=='=') && (LA(2)=='=')) {
            mDOBEQUAL(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='<') && (LA(2)=='=')) {
            mLEQUAL(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='>') && (LA(2)=='=')) {
            mGEQUAL(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='/') && (LA(2)=='*')) {
            mComment(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='/') && (true)) {
            mSLASH(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='<') && (true)) {
            mLESSTHAN(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='>') && (true)) {
            mGREATERTHAN(true);
            theRetToken=_returnToken;
      }
      else if ((LA(1)=='=') && (true)) {
            mEQUALS(true);
            theRetToken=_returnToken;
      }
```

```
                        else {
                              if (LA(1)==EOF_CHAR) {uponEOF(); _returnToken =
makeToken(Token.EOF_TYPE);}
                        else {throw new
NoViableAltForCharException((char)LA(1), getFilename(), getLine(),
getColumn());}
                        }
                        }
                        if ( _returnToken==null ) continue tryAgain; // found
SKIP token
                        _ttype = _returnToken.getType();
                        _returnToken.setType(_ttype);
                        return _returnToken;
                  }
                  catch (RecognitionException e) {
                        throw new TokenStreamRecognitionException(e);
                  }
            }
            catch (CharStreamException cse) {
                  if ( cse instanceof CharStreamIOException ) {
                        throw new
TokenStreamIOException(((CharStreamIOException)cse).io);
                  }
                  else {
                        throw new TokenStreamException(cse.getMessage());
                  }
            }
      }
}

      public final void mPERIOD(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = PERIOD;
            int _saveIndex;

            match('.');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mPLUS(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = PLUS;
            int _saveIndex;

            match('+');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
```

```
            _returnToken = _token;
      }

      public final void mDASH(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = DASH;
            int _saveIndex;

            match('-');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mSLASH(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = SLASH;
            int _saveIndex;

            match('/');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mSTAR(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = STAR;
            int _saveIndex;

            match('*');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mPARALLEL(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = PARALLEL;
            int _saveIndex;

            match("||");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
```

```
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mDOBAND(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = DOBAND;
            int _saveIndex;

            match("&&");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mLESSTHAN(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = LESSTHAN;
            int _saveIndex;

            match('<');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mGREATERTHAN(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = GREATERTHAN;
            int _saveIndex;

            match('>');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mCOMMA(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = COMMA;
            int _saveIndex;
```

```
            match(',');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mDOBEQUAL(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = DOBEQUAL;
            int _saveIndex;

            match("==");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mEQUALS(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = EQUALS;
            int _saveIndex;

            match('=');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mSEMICOLON(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = SEMICOLON;
            int _saveIndex;

            match(';');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mCOLON(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
```

```
            _ttype = COLON;
            int _saveIndex;

            match(':');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mLPAREN(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = LPAREN;
            int _saveIndex;

            match('(');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mRPAREN(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = RPAREN;
            int _saveIndex;

            match(')');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mLBRACKET(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = LBRACKET;
            int _saveIndex;

            match('[');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }
```

```
      public final void mRBRACKET(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = RBRACKET;
            int _saveIndex;

            match(']');
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mLEQUAL(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = LEQUAL;
            int _saveIndex;

            match("<=");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mGEQUAL(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = GEQUAL;
            int _saveIndex;

            match(">=");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mNEQUAL(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = NEQUAL;
            int _saveIndex;

            match("!=");
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
```

```
            _returnToken = _token;
      }

      public final void mID(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = ID;
            int _saveIndex;

            {
            switch ( LA(1)) {
            case 'a':  case 'b':  case 'c':  case 'd':
            case 'e':  case 'f':  case 'g':  case 'h':
            case 'i':  case 'j':  case 'k':  case 'l':
            case 'm':  case 'n':  case 'o':  case 'p':
            case 'q':  case 'r':  case 's':  case 't':
            case 'u':  case 'v':  case 'w':  case 'x':
            case 'y':  case 'z':
            {
                  matchRange('a','z');
                  break;
            }
            case 'A':  case 'B':  case 'C':  case 'D':
            case 'E':  case 'F':  case 'G':  case 'H':
            case 'I':  case 'J':  case 'K':  case 'L':
            case 'M':  case 'N':  case 'O':  case 'P':
            case 'Q':  case 'R':  case 'S':  case 'T':
            case 'U':  case 'V':  case 'W':  case 'X':
            case 'Y':  case 'Z':
            {
                  matchRange('A','Z');
                  break;
            }
            default:
            {
                  throw new NoViableAltForCharException((char)LA(1),
getFilename(), getLine(), getColumn());
            }
            }
            }
            {
            _loop116:
            do {
                  switch ( LA(1)) {
                  case 'a':  case 'b':  case 'c':  case 'd':
                  case 'e':  case 'f':  case 'g':  case 'h':
                  case 'i':  case 'j':  case 'k':  case 'l':
                  case 'm':  case 'n':  case 'o':  case 'p':
                  case 'q':  case 'r':  case 's':  case 't':
                  case 'u':  case 'v':  case 'w':  case 'x':
                  case 'y':  case 'z':
                  {
                        matchRange('a','z');
                        break;
                  }
                  case 'A':  case 'B':  case 'C':  case 'D':
                  case 'E':  case 'F':  case 'G':  case 'H':
```

```
                  case 'I':  case 'J':  case 'K':  case 'L':
                  case 'M':  case 'N':  case 'O':  case 'P':
                  case 'Q':  case 'R':  case 'S':  case 'T':
                  case 'U':  case 'V':  case 'W':  case 'X':
                  case 'Y':  case 'Z':
                  {
                        matchRange('A','Z');
                        break;
                  }
                  case '_':
                  {
                        match('_');
                        break;
                  }
                  case '0':  case '1':  case '2':  case '3':
                  case '4':  case '5':  case '6':  case '7':
                  case '8':  case '9':
                  {
                        matchRange('0','9');
                        break;
                  }
                  default:
                  {
                        break _loop116;
                  }
                  }
            } while (true);
            }
            _ttype = testLiteralsTable(_ttype);
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mNumber(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = Number;
            int _saveIndex;

            {
            int _cnt119=0;
            _loop119:
            do {
                  if (((LA(1) >= '0' && LA(1) <= '9'))) {
                        matchRange('0','9');
                  }
                  else {
                        if ( _cnt119>=1 ) { break _loop119; } else {throw new
NoViableAltForCharException((char)LA(1), getFilename(), getLine(),
getColumn());}
                  }

                  _cnt119++;
```

```
            } while (true);
            }
            {
            if ((LA(1)=='.')) {
                  match('.');
                  {
                  _loop122:
                  do {
                        if (((LA(1) >= '0' && LA(1) <= '9'))) {
                              matchRange('0','9');
                        }
                        else {
                              break _loop122;
                        }

                  } while (true);
                  }
            }
            else {
            }

            }
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mStringConstant(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = StringConstant;
            int _saveIndex;

            _saveIndex=text.length();
            match('"');
            text.setLength(_saveIndex);
            {
            _loop127:
            do {
                  if ((LA(1)=='"') && (LA(2)=='"')) {
                        {
                        _saveIndex=text.length();
                        match('"');
                        text.setLength(_saveIndex);
                        match('"');
                        }
                  }
                  else if ((_tokenSet_0.member(LA(1)))) {
                        {
                        match(_tokenSet_0);
                        }
                  }
                  else {
                        break _loop127;
```

```
                  }

            } while (true);
            }
            _saveIndex=text.length();
            match('"');
            text.setLength(_saveIndex);
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mWhitespace(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = Whitespace;
            int _saveIndex;

            {
            int _cnt130=0;
            _loop130:
            do {
                  switch ( LA(1)) {
                  case ' ':
                  {
                        match(' ');
                        break;
                  }
                  case '\t':
                  {
                        match('\t');
                        break;
                  }
                  case '\u000c':
                  {
                        match('\f');
                        break;
                  }
                  default:
                  {
                        if ( _cnt130>=1 ) { break _loop130; } else {throw new
NoViableAltForCharException((char)LA(1), getFilename(), getLine(),
getColumn());}
                  }
                  }
                  _cnt130++;
            } while (true);
            }
            if ( inputState.guessing==0 ) {
                  _ttype = Token.SKIP;
            }
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
```

```
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

      public final void mNewline(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = Newline;
            int _saveIndex;

            {
            if ((LA(1)=='\r') && (LA(2)=='\n')) {
                  match("\r\n");
            }
            else if ((LA(1)=='\n')) {
                  match('\n');
            }
            else if ((LA(1)=='\r') && (true)) {
                  match('\r');
            }
            else {
                  throw new NoViableAltForCharException((char)LA(1),
getFilename(), getLine(), getColumn());
            }

            }
            if ( inputState.guessing==0 ) {
                  _ttype = Token.SKIP;
            }
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }

/**
  All comments
*/
      public final void mComment(boolean _createToken) throws
RecognitionException, CharStreamException, TokenStreamException {
            int _ttype; Token _token=null; int _begin=text.length();
            _ttype = Comment;
            int _saveIndex;

            match("/*");
            {
            _loop139:
            do {
                  // nongreedy exit test
                  if ((LA(1)=='*') && (LA(2)=='/')) break _loop139;
                  if (((LA(1) >= '\u0003' && LA(1) <= '\u00ff')) && ((LA(2)
>= '\u0003' && LA(2) <= '\u00ff'))) {
                        {
```

```
                        boolean synPredMatched137 = false;
                        if (((LA(1)=='\r') && (LA(2)=='\n'))) {
                              int _m137 = mark();
                              synPredMatched137 = true;
                              inputState.guessing++;
                              try {
                                    {
                                    match('\r');
                                    match('\n');
                                    }
                              }
                              catch (RecognitionException pe) {
                                    synPredMatched137 = false;
                              }
                              rewind(_m137);
                              inputState.guessing--;
                        }
                        if ( synPredMatched137 ) {
                              match('\r');
                              match('\n');
                              if ( inputState.guessing==0 ) {
                                    newline();
                              }
                        }
                        else if ((LA(1)=='\r') && ((LA(2) >= '\u0003' &&
LA(2) <= '\u00ff'))) {
                              match('\r');
                              if ( inputState.guessing==0 ) {
                                    newline();
                              }
                        }
                        else if ((LA(1)=='\n')) {
                              match('\n');
                              if ( inputState.guessing==0 ) {
                                    newline();
                              }
                        }
                        else if ((_tokenSet_1.member(LA(1)))) {
                              {
                              match(_tokenSet_1);
                              }
                        }
                        else {
                              throw new
NoViableAltForCharException((char)LA(1), getFilename(), getLine(),
getColumn());
                        }

                        }
                  }
                  else {
                        break _loop139;
                  }

            } while (true);
            }
            match("*/");
```

```
            if ( inputState.guessing==0 ) {
                  _ttype = Token.SKIP;
            }
            if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
                  _token = makeToken(_ttype);
                  _token.setText(new String(text.getBuffer(), _begin,
text.length()-_begin));
            }
            _returnToken = _token;
      }


      private static final long[] mk_tokenSet_0() {
            long[] data = new long[8];
            data[0]=-17179870216L;
            for (int i = 1; i<=3; i++) { data[i]=-1L; }
            return data;
      }
      public static final BitSet _tokenSet_0 = new BitSet(mk_tokenSet_0());
      private static final long[] mk_tokenSet_1() {
            long[] data = new long[8];
            data[0]=-9224L;
            for (int i = 1; i<=3; i++) { data[i]=-1L; }
            return data;
      }
      public static final BitSet _tokenSet_1 = new BitSet(mk_tokenSet_1());

      }
```

```
// $ANTLR 2.7.2: "bslGrammar_2.g" -> "BSL_Parser.java"$
/*
 * Parser takes the tokens from the lexer and creates the
 * AST.
 * @author ANTLR
 * cunix id: antlr
 */

import antlr.TokenBuffer;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;
import antlr.ANTLRException;
import antlr.LLkParser;
import antlr.Token;
import antlr.TokenStream;
import antlr.RecognitionException;
import antlr.NoViableAltException;
import antlr.MismatchedTokenException;
import antlr.SemanticException;
import antlr.ParserSharedInputState;
import antlr.collections.impl.BitSet;
import antlr.collections.AST;
import java.util.Hashtable;
import antlr.ASTFactory;
import antlr.ASTPair;
import antlr.collections.impl.ASTArray;

public class BSL_Parser extends antlr.LLkParser       implements
BSLTokenTypes
```

```
 {

protected BSL_Parser(TokenBuffer tokenBuf, int k) {
  super(tokenBuf,k);
  tokenNames = _tokenNames;
  buildTokenTypeASTClassMap();
  astFactory = new ASTFactory(getTokenTypeToASTClassMap());
}

public BSL_Parser(TokenBuffer tokenBuf) {
  this(tokenBuf,2);
}

protected BSL_Parser(TokenStream lexer, int k) {
  super(lexer,k);
  tokenNames = _tokenNames;
  buildTokenTypeASTClassMap();
  astFactory = new ASTFactory(getTokenTypeToASTClassMap());
}

public BSL_Parser(TokenStream lexer) {
  this(lexer,2);
}

public BSL_Parser(ParserSharedInputState state) {
  super(state,2);
  tokenNames = _tokenNames;
  buildTokenTypeASTClassMap();
  astFactory = new ASTFactory(getTokenTypeToASTClassMap());
}

      public final void bslscript() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST bslscript_AST = null;

            {
            _loop3:
            do {
                  if ((LA(1)==LITERAL_start)) {
                        method();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop3;
                  }

            } while (true);
            }
            script();
            astFactory.addASTChild(currentAST, returnAST);
            match(Token.EOF_TYPE);
            bslscript_AST = (AST)currentAST.root;
```

```
            bslscript_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(BSLSCRIPT,"bslscript")).add(bslscript_AST)
);
            currentAST.root = bslscript_AST;
            currentAST.child = bslscript_AST!=null
&&bslscript_AST.getFirstChild()!=null ?
                  bslscript_AST.getFirstChild() : bslscript_AST;
            currentAST.advanceChildToEnd();
            bslscript_AST = (AST)currentAST.root;
            returnAST = bslscript_AST;
      }

      public final void method() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST method_AST = null;

            match(LITERAL_start);
            AST tmp3_AST = null;
            tmp3_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp3_AST);
            match(LITERAL_method);
            type();
            astFactory.addASTChild(currentAST, returnAST);
            methodName();
            astFactory.addASTChild(currentAST, returnAST);
            match(LPAREN);
            {
            switch ( LA(1)) {
            case LITERAL_void:
            case LITERAL_boolean:
            case LITERAL_byte:
            case LITERAL_char:
            case LITERAL_int:
            case LITERAL_short:
            case LITERAL_BuildingBlock:
            case LITERAL_Sequence:
            {
                  args();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case RPAREN:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(RPAREN);
            script();
            astFactory.addASTChild(currentAST, returnAST);
```

```
            {
            switch ( LA(1)) {
            case LITERAL_return:
            {
                  returnStatement();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case LITERAL_end:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(LITERAL_end);
            match(LITERAL_method);
            match(SEMICOLON);
            method_AST = (AST)currentAST.root;
            returnAST = method_AST;
      }

      public final void script() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST script_AST = null;

            {
            int _cnt15=0;
            _loop15:
            do {
                  if ((_tokenSet_0.member(LA(1)))) {
                        expression();
                        astFactory.addASTChild(currentAST, returnAST);
                        match(SEMICOLON);
                  }
                  else {
                        if ( _cnt15>=1 ) { break _loop15; } else {throw new
NoViableAltException(LT(1), getFilename());}
                  }

                  _cnt15++;
            } while (true);
            }
            script_AST = (AST)currentAST.root;
            returnAST = script_AST;
      }

      public final void type() throws RecognitionException,
TokenStreamException {

            returnAST = null;
```

```
            ASTPair currentAST = new ASTPair();
            AST type_AST = null;

            switch ( LA(1)) {
            case LITERAL_boolean:
            case LITERAL_byte:
            case LITERAL_char:
            case LITERAL_int:
            case LITERAL_short:
            {
                  primitive();
                  astFactory.addASTChild(currentAST, returnAST);
                  type_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_BuildingBlock:
            case LITERAL_Sequence:
            {
                  bslType();
                  astFactory.addASTChild(currentAST, returnAST);
                  type_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_void:
            {
                  AST tmp10_AST = null;
                  tmp10_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp10_AST);
                  match(LITERAL_void);
                  type_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = type_AST;
      }

      public final void methodName() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST methodName_AST = null;

            AST tmp11_AST = null;
            tmp11_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp11_AST);
            match(ID);
            methodName_AST = (AST)currentAST.root;
            returnAST = methodName_AST;
      }

      public final void args() throws RecognitionException,
TokenStreamException {
```

```
            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST args_AST = null;

            type();
            astFactory.addASTChild(currentAST, returnAST);
            AST tmp12_AST = null;
            tmp12_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp12_AST);
            match(ID);
            {
            _loop10:
            do {
                  if ((LA(1)==COMMA)) {
                        match(COMMA);
                        type();
                        astFactory.addASTChild(currentAST, returnAST);
                        AST tmp14_AST = null;
                        tmp14_AST = astFactory.create(LT(1));
                        astFactory.addASTChild(currentAST, tmp14_AST);
                        match(ID);
                  }
                  else {
                        break _loop10;
                  }

            } while (true);
            }
            args_AST = (AST)currentAST.root;
            args_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(ARGS,"args")).add(args_AST));
            currentAST.root = args_AST;
            currentAST.child = args_AST!=null
&&args_AST.getFirstChild()!=null ?
                  args_AST.getFirstChild() : args_AST;
            currentAST.advanceChildToEnd();
            args_AST = (AST)currentAST.root;
            returnAST = args_AST;
      }

      public final void returnStatement() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST returnStatement_AST = null;

            AST tmp15_AST = null;
            tmp15_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp15_AST);
            match(LITERAL_return);
            {
            switch ( LA(1)) {
            case ID:
            {
                  AST tmp16_AST = null;
```

```
                tmp16_AST = astFactory.create(LT(1));
                astFactory.addASTChild(currentAST, tmp16_AST);
                match(ID);
                break;
            }
            case Number:
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                constant();
                astFactory.addASTChild(currentAST, returnAST);
                break;
            }
            default:
            {
                throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(SEMICOLON);
            returnStatement_AST = (AST)currentAST.root;
            returnAST = returnStatement_AST;
      }

      public final void constant() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST constant_AST = null;

            switch ( LA(1)) {
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                constantLiteral();
                astFactory.addASTChild(currentAST, returnAST);
                constant_AST = (AST)currentAST.root;
                break;
            }
            case Number:
            {
                AST tmp18_AST = null;
                tmp18_AST = astFactory.create(LT(1));
                astFactory.addASTChild(currentAST, tmp18_AST);
                match(Number);
                constant_AST = (AST)currentAST.root;
                break;
            }
            default:
            {
                throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = constant_AST;
```

```
	}

	public final void expression() throws RecognitionException,
TokenStreamException {

		returnAST = null;
		ASTPair currentAST = new ASTPair();
		AST expression_AST = null;

		{
		switch ( LA(1)) {
		case LITERAL_new:
		{
			declaration();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		case LPAREN:
		case ID:
		case Number:
		case LITERAL_true:
		case LITERAL_false:
		case StringConstant:
		case LITERAL_exit:
		case LITERAL_break:
		{
			statement();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		case LITERAL_for:
		{
			forBlock();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		case LITERAL_if:
		{
			ifBlock();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		case LITERAL_while:
		{
			whileBlock();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		default:
		{
			throw new NoViableAltException(LT(1), getFilename());
		}
		}
		}
		expression_AST = (AST)currentAST.root;
		expression_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(EXPER,"exper")).add(expression_AST));
```

```
            currentAST.root = expression_AST;
            currentAST.child = expression_AST!=null
&&expression_AST.getFirstChild()!=null ?
                  expression_AST.getFirstChild() : expression_AST;
            currentAST.advanceChildToEnd();
            expression_AST = (AST)currentAST.root;
            returnAST = expression_AST;
      }

      public final void declaration() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST declaration_AST = null;

            match(LITERAL_new);
            type();
            astFactory.addASTChild(currentAST, returnAST);
            AST tmp20_AST = null;
            tmp20_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp20_AST);
            match(ID);
            {
            switch ( LA(1)) {
            case LBRACKET:
            {
                  dataStructureAccess();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case SEMICOLON:
            case LITERAL_type:
            case EQUALS:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            {
            switch ( LA(1)) {
            case LITERAL_type:
            {
                  typedecl();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case SEMICOLON:
            case EQUALS:
            {
                  break;
            }
            default:
```

```
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            {
            switch ( LA(1)) {
            case EQUALS:
            {
                  assgnValue();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case SEMICOLON:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            declaration_AST = (AST)currentAST.root;
            declaration_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(DECLS,"decls")).add(declaration_AST));
            currentAST.root = declaration_AST;
            currentAST.child = declaration_AST!=null
&&declaration_AST.getFirstChild()!=null ?
                  declaration_AST.getFirstChild() : declaration_AST;
            currentAST.advanceChildToEnd();
            declaration_AST = (AST)currentAST.root;
            returnAST = declaration_AST;
      }

      public final void statement() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST statement_AST = null;

            {
            switch ( LA(1)) {
            case ID:
            {
                  {
                  AST tmp21_AST = null;
                  tmp21_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp21_AST);
                  match(ID);
                  {
                  switch ( LA(1)) {
                  case LBRACKET:
                  {
                        dataStructureAccess();
                        astFactory.addASTChild(currentAST, returnAST);
```

```
                break;
            }
            case LPAREN:
            case RPAREN:
            case SEMICOLON:
            case EQUALS:
            case COLON:
            case PERIOD:
            {
                break;
            }
            default:
            {
                throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            {
            switch ( LA(1)) {
            case EQUALS:
            {
                assgnValue();
                astFactory.addASTChild(currentAST, returnAST);
                break;
            }
            case COLON:
            {
                methodCall();
                astFactory.addASTChild(currentAST, returnAST);
                break;
            }
            case LPAREN:
            {
                sequenceCalls();
                astFactory.addASTChild(currentAST, returnAST);
                break;
            }
            case PERIOD:
            {
                property();
                astFactory.addASTChild(currentAST, returnAST);
                break;
            }
            case RPAREN:
            case SEMICOLON:
            {
                break;
            }
            default:
            {
                throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            }
            break;
        }
```

```
            case LPAREN:
            case Number:
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                  operation();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case LITERAL_exit:
            case LITERAL_break:
            {
                  atomicStatement();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            statement_AST = (AST)currentAST.root;
            returnAST = statement_AST;
      }

      public final void forBlock() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST forBlock_AST = null;

            AST tmp22_AST = null;
            tmp22_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp22_AST);
            match(LITERAL_for);
            forConditional();
            astFactory.addASTChild(currentAST, returnAST);
            ifinside();
            astFactory.addASTChild(currentAST, returnAST);
            match(LITERAL_next);
            forBlock_AST = (AST)currentAST.root;
            returnAST = forBlock_AST;
      }

      public final void ifBlock() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST ifBlock_AST = null;

            AST tmp24_AST = null;
            tmp24_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp24_AST);
```

```
            match(LITERAL_if);
            operation();
            astFactory.addASTChild(currentAST, returnAST);
            match(LITERAL_then);
            ifinside();
            astFactory.addASTChild(currentAST, returnAST);
            {
            _loop45:
            do {
                  if ((LA(1)==LITERAL_elseif)) {
                        elseif();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop45;
                  }

            } while (true);
            }
            {
            switch ( LA(1)) {
            case LITERAL_else:
            {
                  elsePart();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case LITERAL_end:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(LITERAL_end);
            match(LITERAL_if);
            ifBlock_AST = (AST)currentAST.root;
            returnAST = ifBlock_AST;
      }

      public final void whileBlock() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST whileBlock_AST = null;

            AST tmp28_AST = null;
            tmp28_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp28_AST);
            match(LITERAL_while);
            operation();
            astFactory.addASTChild(currentAST, returnAST);
            ifinside();
```

```
            astFactory.addASTChild(currentAST, returnAST);
            match(LITERAL_next);
            whileBlock_AST = (AST)currentAST.root;
            returnAST = whileBlock_AST;
      }

      public final void dataStructureAccess() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST dataStructureAccess_AST = null;

            AST tmp30_AST = null;
            tmp30_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp30_AST);
            match(LBRACKET);
            {
            switch ( LA(1)) {
            case Number:
            {
                  AST tmp31_AST = null;
                  tmp31_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp31_AST);
                  match(Number);
                  {
                  switch ( LA(1)) {
                  case COMMA:
                  {
                        match(COMMA);
                        AST tmp33_AST = null;
                        tmp33_AST = astFactory.create(LT(1));
                        astFactory.addASTChild(currentAST, tmp33_AST);
                        match(Number);
                        break;
                  }
                  case RBRACKET:
                  {
                        break;
                  }
                  default:
                  {
                        throw new NoViableAltException(LT(1), getFilename());
                  }
                  }
                  }
                  break;
            }
            case RBRACKET:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
```

```
            }
            match(RBRACKET);
            dataStructureAccess_AST = (AST)currentAST.root;
            returnAST = dataStructureAccess_AST;
      }

      public final void typedecl() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST typedecl_AST = null;

            AST tmp35_AST = null;
            tmp35_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp35_AST);
            match(LITERAL_type);
            bioType();
            astFactory.addASTChild(currentAST, returnAST);
            typedecl_AST = (AST)currentAST.root;
            returnAST = typedecl_AST;
      }

      public final void assgnValue() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST assgnValue_AST = null;

            AST tmp36_AST = null;
            tmp36_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp36_AST);
            match(EQUALS);
            {
            switch ( LA(1)) {
            case ID:
            {
                  {
                  AST tmp37_AST = null;
                  tmp37_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp37_AST);
                  match(ID);
                  {
                  switch ( LA(1)) {
                  case LBRACKET:
                  {
                        dataStructureAccess();
                        astFactory.addASTChild(currentAST, returnAST);
                        break;
                  }
                  case LPAREN:
                  case RPAREN:
                  case SEMICOLON:
                  case COLON:
                  case PERIOD:
                  {
```

```
                break;
        }
        default:
        {
                throw new NoViableAltException(LT(1), getFilename());
        }
        }
        }
        {
        switch ( LA(1)) {
        case COLON:
        {
                methodCall();
                astFactory.addASTChild(currentAST, returnAST);
                break;
        }
        case LPAREN:
        {
                sequenceCalls();
                astFactory.addASTChild(currentAST, returnAST);
                break;
        }
        case PERIOD:
        {
                property();
                astFactory.addASTChild(currentAST, returnAST);
                break;
        }
        case RPAREN:
        case SEMICOLON:
        {
                break;
        }
        default:
        {
                throw new NoViableAltException(LT(1), getFilename());
        }
        }
        }
        }
        break;
}
case LPAREN:
case Number:
case LITERAL_true:
case LITERAL_false:
case StringConstant:
{
        operation();
        astFactory.addASTChild(currentAST, returnAST);
        break;
}
default:
{
        throw new NoViableAltException(LT(1), getFilename());
}
}
```

```
            }
            assgnValue_AST = (AST)currentAST.root;
            returnAST = assgnValue_AST;
      }

      public final void bioType() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST bioType_AST = null;

            switch ( LA(1)) {
            case LITERAL_DNA:
            {
                  AST tmp38_AST = null;
                  tmp38_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp38_AST);
                  match(LITERAL_DNA);
                  bioType_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_RNA:
            {
                  AST tmp39_AST = null;
                  tmp39_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp39_AST);
                  match(LITERAL_RNA);
                  bioType_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_AMINOACID:
            {
                  AST tmp40_AST = null;
                  tmp40_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp40_AST);
                  match(LITERAL_AMINOACID);
                  bioType_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_PROTEIN:
            {
                  AST tmp41_AST = null;
                  tmp41_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp41_AST);
                  match(LITERAL_PROTEIN);
                  bioType_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_STRING:
            {
                  AST tmp42_AST = null;
                  tmp42_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp42_AST);
                  match(LITERAL_STRING);
                  bioType_AST = (AST)currentAST.root;
                  break;
```

```
            }
            case LITERAL_UNKNOWN:
            {
                  AST tmp43_AST = null;
                  tmp43_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp43_AST);
                  match(LITERAL_UNKNOWN);
                  bioType_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = bioType_AST;
      }

      public final void methodCall() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST methodCall_AST = null;

            match(COLON);
            {
            switch ( LA(1)) {
            case ID:
            case Number:
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                  parameters();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case RPAREN:
            case SEMICOLON:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            methodCall_AST = (AST)currentAST.root;
            methodCall_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(MCALL,"mcall")).add(methodCall_AST));
            currentAST.root = methodCall_AST;
            currentAST.child = methodCall_AST!=null
&&methodCall_AST.getFirstChild()!=null ?
                  methodCall_AST.getFirstChild() : methodCall_AST;
            currentAST.advanceChildToEnd();
```

```
            methodCall_AST = (AST)currentAST.root;
            returnAST = methodCall_AST;
      }

      public final void sequenceCalls() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST sequenceCalls_AST = null;

            AST tmp45_AST = null;
            tmp45_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp45_AST);
            match(LPAREN);
            AST tmp46_AST = null;
            tmp46_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp46_AST);
            match(Number);
            {
            switch ( LA(1)) {
            case COMMA:
            {
                  match(COMMA);
                  AST tmp48_AST = null;
                  tmp48_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp48_AST);
                  match(Number);
                  break;
            }
            case RPAREN:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(RPAREN);
            sequenceCalls_AST = (AST)currentAST.root;
            returnAST = sequenceCalls_AST;
      }

      public final void property() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST property_AST = null;

            match(PERIOD);
            AST tmp51_AST = null;
            tmp51_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp51_AST);
            match(ID);
```

```
            property_AST = (AST)currentAST.root;
            property_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(PROP,"prop")).add(property_AST));
            currentAST.root = property_AST;
            currentAST.child = property_AST!=null
&&property_AST.getFirstChild()!=null ?
                  property_AST.getFirstChild() : property_AST;
            currentAST.advanceChildToEnd();
            property_AST = (AST)currentAST.root;
            returnAST = property_AST;
      }

      public final void operation() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST operation_AST = null;

            orexpr();
            astFactory.addASTChild(currentAST, returnAST);
            operation_AST = (AST)currentAST.root;
            returnAST = operation_AST;
      }

      public final void atomicStatement() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST atomicStatement_AST = null;

            switch ( LA(1)) {
            case LITERAL_exit:
            {
                  AST tmp52_AST = null;
                  tmp52_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp52_AST);
                  match(LITERAL_exit);
                  atomicStatement_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_break:
            {
                  AST tmp53_AST = null;
                  tmp53_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp53_AST);
                  match(LITERAL_break);
                  atomicStatement_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = atomicStatement_AST;
```

```
	}

	public final void parameters() throws RecognitionException,
TokenStreamException {

		returnAST = null;
		ASTPair currentAST = new ASTPair();
		AST parameters_AST = null;

		{
		switch ( LA(1)) {
		case ID:
		{
			AST tmp54_AST = null;
			tmp54_AST = astFactory.create(LT(1));
			astFactory.addASTChild(currentAST, tmp54_AST);
			match(ID);
			{
			switch ( LA(1)) {
			case LBRACKET:
			{
				dataStructureAccess();
				astFactory.addASTChild(currentAST, returnAST);
				break;
			}
			case LPAREN:
			{
				sequenceCalls();
				astFactory.addASTChild(currentAST, returnAST);
				break;
			}
			case RPAREN:
			case SEMICOLON:
			case COMMA:
			{
				break;
			}
			default:
			{
				throw new NoViableAltException(LT(1), getFilename());
			}
			}
			}
			break;
		}
		case Number:
		case LITERAL_true:
		case LITERAL_false:
		case StringConstant:
		{
			constant();
			astFactory.addASTChild(currentAST, returnAST);
			break;
		}
		default:
		{
			throw new NoViableAltException(LT(1), getFilename());
```

```
            }
            }
            }
            {
            _loop41:
            do {
                  if ((LA(1)==COMMA)) {
                        match(COMMA);
                        {
                        switch ( LA(1)) {
                        case ID:
                        {
                              AST tmp56_AST = null;
                              tmp56_AST = astFactory.create(LT(1));
                              astFactory.addASTChild(currentAST, tmp56_AST);
                              match(ID);
                              {
                              switch ( LA(1)) {
                              case LBRACKET:
                              {
                                    dataStructureAccess();
                                    astFactory.addASTChild(currentAST,
returnAST);
                                    break;
                              }
                              case LPAREN:
                              {
                                    sequenceCalls();
                                    astFactory.addASTChild(currentAST,
returnAST);
                                    break;
                              }
                              case RPAREN:
                              case SEMICOLON:
                              case COMMA:
                              {
                                    break;
                              }
                              default:
                              {
                                    throw new NoViableAltException(LT(1),
getFilename());
                              }
                              }
                              }
                              break;
                        }
                        case Number:
                        case LITERAL_true:
                        case LITERAL_false:
                        case StringConstant:
                        {
                              constant();
                              astFactory.addASTChild(currentAST, returnAST);
                              break;
                        }
                        default:
```

```
                        {
                              throw new NoViableAltException(LT(1),
getFilename());
                        }
                        }
                        }
                  }
                  else {
                        break _loop41;
                  }

            } while (true);
            }
            parameters_AST = (AST)currentAST.root;
            returnAST = parameters_AST;
      }

      public final void ifinside() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST ifinside_AST = null;

            {
            _loop49:
            do {
                  if ((_tokenSet_0.member(LA(1)))) {
                        expression();
                        astFactory.addASTChild(currentAST, returnAST);
                        match(SEMICOLON);
                  }
                  else {
                        break _loop49;
                  }

            } while (true);
            }
            ifinside_AST = (AST)currentAST.root;
            ifinside_AST = (AST)astFactory.make( (new
ASTArray(2)).add(astFactory.create(IFER,"ifer")).add(ifinside_AST));
            currentAST.root = ifinside_AST;
            currentAST.child = ifinside_AST!=null
&&ifinside_AST.getFirstChild()!=null ?
                  ifinside_AST.getFirstChild() : ifinside_AST;
            currentAST.advanceChildToEnd();
            ifinside_AST = (AST)currentAST.root;
            returnAST = ifinside_AST;
      }

      public final void elseif() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST elseif_AST = null;
```

```
            AST tmp58_AST = null;
            tmp58_AST = astFactory.create(LT(1));
            astFactory.makeASTRoot(currentAST, tmp58_AST);
            match(LITERAL_elseif);
            operation();
            astFactory.addASTChild(currentAST, returnAST);
            match(LITERAL_then);
            ifinside();
            astFactory.addASTChild(currentAST, returnAST);
            elseif_AST = (AST)currentAST.root;
            returnAST = elseif_AST;
      }

      public final void elsePart() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST elsePart_AST = null;

            AST tmp60_AST = null;
            tmp60_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp60_AST);
            match(LITERAL_else);
            ifinside();
            astFactory.addASTChild(currentAST, returnAST);
            elsePart_AST = (AST)currentAST.root;
            returnAST = elsePart_AST;
      }

      public final void forConditional() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST forConditional_AST = null;

            match(LPAREN);
            {
            switch ( LA(1)) {
            case LITERAL_new:
            {
                  declaration();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case SEMICOLON:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(SEMICOLON);
```

```
            {
            switch ( LA(1)) {
            case LPAREN:
            case Number:
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                  operation();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case SEMICOLON:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(SEMICOLON);
            {
            switch ( LA(1)) {
            case ID:
            {
                  AST tmp64_AST = null;
                  tmp64_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp64_AST);
                  match(ID);
                  assgnValue();
                  astFactory.addASTChild(currentAST, returnAST);
                  break;
            }
            case RPAREN:
            {
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            }
            match(RPAREN);
            forConditional_AST = (AST)currentAST.root;
            returnAST = forConditional_AST;
      }

      public final void orexpr() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST orexpr_AST = null;
```

```
            andexpr();
            astFactory.addASTChild(currentAST, returnAST);
            {
            _loop62:
            do {
                  if ((LA(1)==PARALLEL)) {
                        AST tmp66_AST = null;
                        tmp66_AST = astFactory.create(LT(1));
                        astFactory.makeASTRoot(currentAST, tmp66_AST);
                        match(PARALLEL);
                        andexpr();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop62;
                  }

            } while (true);
            }
            orexpr_AST = (AST)currentAST.root;
            returnAST = orexpr_AST;
      }

      public final void andexpr() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST andexpr_AST = null;

            compOp();
            astFactory.addASTChild(currentAST, returnAST);
            {
            _loop65:
            do {
                  if ((LA(1)==DOBAND)) {
                        AST tmp67_AST = null;
                        tmp67_AST = astFactory.create(LT(1));
                        astFactory.makeASTRoot(currentAST, tmp67_AST);
                        match(DOBAND);
                        compOp();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop65;
                  }

            } while (true);
            }
            andexpr_AST = (AST)currentAST.root;
            returnAST = andexpr_AST;
      }

      public final void compOp() throws RecognitionException,
TokenStreamException {

            returnAST = null;
```

```
ASTPair currentAST = new ASTPair();
AST compOp_AST = null;

addexpr();
astFactory.addASTChild(currentAST, returnAST);
{
_loop69:
do {
      if (((LA(1) >= DOBEQUAL && LA(1) <= NEQUAL))) {
            {
            switch ( LA(1)) {
            case DOBEQUAL:
            {
                  AST tmp68_AST = null;
                  tmp68_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp68_AST);
                  match(DOBEQUAL);
                  break;
            }
            case LESSTHAN:
            {
                  AST tmp69_AST = null;
                  tmp69_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp69_AST);
                  match(LESSTHAN);
                  break;
            }
            case GREATERTHAN:
            {
                  AST tmp70_AST = null;
                  tmp70_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp70_AST);
                  match(GREATERTHAN);
                  break;
            }
            case LEQUAL:
            {
                  AST tmp71_AST = null;
                  tmp71_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp71_AST);
                  match(LEQUAL);
                  break;
            }
            case GEQUAL:
            {
                  AST tmp72_AST = null;
                  tmp72_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp72_AST);
                  match(GEQUAL);
                  break;
            }
            case NEQUAL:
            {
                  AST tmp73_AST = null;
                  tmp73_AST = astFactory.create(LT(1));
                  astFactory.makeASTRoot(currentAST, tmp73_AST);
                  match(NEQUAL);
```

```
                              break;
                        }
                        default:
                        {
                              throw new NoViableAltException(LT(1),
getFilename());
                        }
                        }
                        }
                        addexpr();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop69;
                  }

            } while (true);
            }
            compOp_AST = (AST)currentAST.root;
            returnAST = compOp_AST;
      }

      public final void addexpr() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST addexpr_AST = null;

            mulexpr();
            astFactory.addASTChild(currentAST, returnAST);
            {
            _loop73:
            do {
                  if ((LA(1)==PLUS||LA(1)==DASH)) {
                        {
                        switch ( LA(1)) {
                        case PLUS:
                        {
                              AST tmp74_AST = null;
                              tmp74_AST = astFactory.create(LT(1));
                              astFactory.makeASTRoot(currentAST, tmp74_AST);
                              match(PLUS);
                              break;
                        }
                        case DASH:
                        {
                              AST tmp75_AST = null;
                              tmp75_AST = astFactory.create(LT(1));
                              astFactory.makeASTRoot(currentAST, tmp75_AST);
                              match(DASH);
                              break;
                        }
                        default:
                        {
                              throw new NoViableAltException(LT(1),
getFilename());
```

```
                        }
                        }
                        }
                        mulexpr();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop73;
                  }

            } while (true);
            }
            addexpr_AST = (AST)currentAST.root;
            returnAST = addexpr_AST;
      }

      public final void mulexpr() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST mulexpr_AST = null;

            unaryexpr();
            astFactory.addASTChild(currentAST, returnAST);
            {
            _loop77:
            do {
                  if (((LA(1) >= STAR && LA(1) <= LITERAL_mod))) {
                        {
                        switch ( LA(1)) {
                        case STAR:
                        {
                              AST tmp76_AST = null;
                              tmp76_AST = astFactory.create(LT(1));
                              astFactory.makeASTRoot(currentAST, tmp76_AST);
                              match(STAR);
                              break;
                        }
                        case SLASH:
                        {
                              AST tmp77_AST = null;
                              tmp77_AST = astFactory.create(LT(1));
                              astFactory.makeASTRoot(currentAST, tmp77_AST);
                              match(SLASH);
                              break;
                        }
                        case LITERAL_mod:
                        {
                              AST tmp78_AST = null;
                              tmp78_AST = astFactory.create(LT(1));
                              astFactory.makeASTRoot(currentAST, tmp78_AST);
                              match(LITERAL_mod);
                              break;
                        }
                        default:
                        {
```

```
                              throw new NoViableAltException(LT(1),
getFilename());
                        }
                        }
                        }
                        unaryexpr();
                        astFactory.addASTChild(currentAST, returnAST);
                  }
                  else {
                        break _loop77;
                  }

            } while (true);
            }
            mulexpr_AST = (AST)currentAST.root;
            returnAST = mulexpr_AST;
      }

      public final void unaryexpr() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST unaryexpr_AST = null;

            switch ( LA(1)) {
            case LPAREN:
            {
                  match(LPAREN);
                  {
                  switch ( LA(1)) {
                  case DASH:
                  {
                        AST tmp80_AST = null;
                        tmp80_AST = astFactory.create(LT(1));
                        astFactory.makeASTRoot(currentAST, tmp80_AST);
                        match(DASH);
                        unaryexpr();
                        astFactory.addASTChild(currentAST, returnAST);
                        break;
                  }
                  case LPAREN:
                  case ID:
                  case Number:
                  case LITERAL_true:
                  case LITERAL_false:
                  case StringConstant:
                  case LITERAL_exit:
                  case LITERAL_break:
                  {
                        statement();
                        astFactory.addASTChild(currentAST, returnAST);
                        break;
                  }
                  default:
                  {
                        throw new NoViableAltException(LT(1), getFilename());
```

```
                  }
                  }
                  }
                  match(RPAREN);
                  unaryexpr_AST = (AST)currentAST.root;
                  break;
            }
            case Number:
            case LITERAL_true:
            case LITERAL_false:
            case StringConstant:
            {
                  constant();
                  astFactory.addASTChild(currentAST, returnAST);
                  unaryexpr_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = unaryexpr_AST;
      }

      public final void primitive() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST primitive_AST = null;

            switch ( LA(1)) {
            case LITERAL_boolean:
            {
                  AST tmp82_AST = null;
                  tmp82_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp82_AST);
                  match(LITERAL_boolean);
                  primitive_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_byte:
            {
                  AST tmp83_AST = null;
                  tmp83_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp83_AST);
                  match(LITERAL_byte);
                  primitive_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_char:
            {
                  AST tmp84_AST = null;
                  tmp84_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp84_AST);
                  match(LITERAL_char);
```

```
                  primitive_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_int:
            {
                  AST tmp85_AST = null;
                  tmp85_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp85_AST);
                  match(LITERAL_int);
                  primitive_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_short:
            {
                  AST tmp86_AST = null;
                  tmp86_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp86_AST);
                  match(LITERAL_short);
                  primitive_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = primitive_AST;
      }

      public final void bslType() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST bslType_AST = null;

            switch ( LA(1)) {
            case LITERAL_BuildingBlock:
            {
                  AST tmp87_AST = null;
                  tmp87_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp87_AST);
                  match(LITERAL_BuildingBlock);
                  bslType_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_Sequence:
            {
                  AST tmp88_AST = null;
                  tmp88_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp88_AST);
                  match(LITERAL_Sequence);
                  bslType_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
```

```
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = bslType_AST;
      }

      public final void constantLiteral() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST constantLiteral_AST = null;

            switch ( LA(1)) {
            case LITERAL_true:
            {
                  AST tmp89_AST = null;
                  tmp89_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp89_AST);
                  match(LITERAL_true);
                  constantLiteral_AST = (AST)currentAST.root;
                  break;
            }
            case LITERAL_false:
            {
                  AST tmp90_AST = null;
                  tmp90_AST = astFactory.create(LT(1));
                  astFactory.addASTChild(currentAST, tmp90_AST);
                  match(LITERAL_false);
                  constantLiteral_AST = (AST)currentAST.root;
                  break;
            }
            case StringConstant:
            {
                  stringConstant();
                  astFactory.addASTChild(currentAST, returnAST);
                  constantLiteral_AST = (AST)currentAST.root;
                  break;
            }
            default:
            {
                  throw new NoViableAltException(LT(1), getFilename());
            }
            }
            returnAST = constantLiteral_AST;
      }

      public final void stringConstant() throws RecognitionException,
TokenStreamException {

            returnAST = null;
            ASTPair currentAST = new ASTPair();
            AST stringConstant_AST = null;

            AST tmp91_AST = null;
            tmp91_AST = astFactory.create(LT(1));
            astFactory.addASTChild(currentAST, tmp91_AST);
```

```
        match(StringConstant);
        stringConstant_AST = (AST)currentAST.root;
        returnAST = stringConstant_AST;
}


public static final String[] _tokenNames = {
        "<0>",
        "EOF",
        "<2>",
        "NULL_TREE_LOOKAHEAD",
        "DECLS",
        "MCALL",
        "PROP",
        "BSLSCRIPT",
        "IFER",
        "EXPER",
        "ARGS",
        "\"start\"",
        "\"method\"",
        "LPAREN",
        "RPAREN",
        "\"end\"",
        "SEMICOLON",
        "ID",
        "COMMA",
        "\"return\"",
        "\"new\"",
        "\"type\"",
        "EQUALS",
        "COLON",
        "PERIOD",
        "\"if\"",
        "\"then\"",
        "\"elseif\"",
        "\"else\"",
        "\"while\"",
        "\"next\"",
        "\"for\"",
        "\"DNA\"",
        "\"RNA\"",
        "\"AMINOACID\"",
        "\"PROTEIN\"",
        "\"STRING\"",
        "\"UNKNOWN\"",
        "PARALLEL",
        "DOBAND",
        "DOBEQUAL",
        "LESSTHAN",
        "GREATERTHAN",
        "LEQUAL",
        "GEQUAL",
        "NEQUAL",
        "PLUS",
        "DASH",
        "STAR",
        "SLASH",
```

```
            "\"mod\"",
            "\"void\"",
            "\"boolean\"",
            "\"byte\"",
            "\"char\"",
            "\"int\"",
            "\"short\"",
            "\"BuildingBlock\"",
            "\"Sequence\"",
            "Number",
            "\"true\"",
            "\"false\"",
            "StringConstant",
            "\"exit\"",
            "\"break\"",
            "LBRACKET",
            "RBRACKET",
            "Whitespace",
            "Newline",
            "Comment"
      };

      protected void buildTokenTypeASTClassMap() {
            tokenTypeToASTClassMap=null;
      };

      private static final long[] mk_tokenSet_0() {
            long[] data = { -576460749584326656L, 1L, 0L, 0L};
            return data;
      }
      public static final BitSet _tokenSet_0 = new BitSet(mk_tokenSet_0());

      }
```

```
/*
 * BSL_TreeParser creates the java file using the AST created by the parser.
 * @author Amna, Igor, Jay, Jared
 * cunix id: ---
 */


import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
import javax.swing.*;
import java.util.*;

public class BSL_TreeParser{
  private static String bslScriptFile;
  private static Hashtable methodHash;
  private static Hashtable idHash;
  private static int lineCount = 1;

  private static void writeDeclaration(AST node) {
    AST typeNode = node.getFirstChild();
    String type = "", id = "";
    if ((typeNode.getText()).equals("Sequence")){
```

```
      if ( ( (typeNode.getNextSibling()).getNextSibling()) != null && ( (
(typeNode.getNextSibling()).getNextSibling()).getText()).equals(
          "[") ) {
        AST arrayPart = (typeNode.getNextSibling()).getNextSibling();
        String name = (typeNode.getNextSibling()).getText();
        bslScriptFile += "\t\t" + typeNode.getText() + " " +
            name;
        id = name;
        type = "Sequence";
        String tempArr = "[] = new Sequence[";
        try {
          int firstNum =
Integer.parseInt((arrayPart.getFirstChild()).getText());
          tempArr += firstNum + "]";
          tempArr += ";\n";
          typeNode = (typeNode.getNextSibling()).getNextSibling();
          String seqType = "UNKNOWN";
          if(typeNode.getNextSibling() != null){
            if(((typeNode.getNextSibling()).getText()).equals("type")){
              seqType =
((typeNode.getNextSibling()).getFirstChild()).getText();
            }
            else{
              System.out.println("BSL Error <67>: Unexpected token here: " +
                                 (typeNode.getNextSibling()).getText());
              System.out.println("\nError encountered on line: " +
lineCount);
              System.exit(1);
            }
            if(((typeNode.getNextSibling()).getNextSibling()) != null){
              System.out.println("BSL Error <67>: Unexpected token here: " +
                                 (typeNode.getNextSibling()).getText());
              System.out.println("\nError encountered on line: " +
lineCount);
              System.exit(1);
            }
          }
          for(int i = 0; i < firstNum; i++){
            tempArr+="\t\t"+ name + "[" + i +"]" +
                ".setType(\"" + seqType + "\");\n";
          }
          if((arrayPart.getFirstChild()).getNextSibling() != null){
            System.err.println("BSL Error <66>: Array dimension greater than
1");
            System.out.println("\nError encountered on line: " + lineCount);
            System.exit(1);
          }
        }
        catch (NumberFormatException e) {
          System.out.println("BSL Error <68>: Number is expected");
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }
        bslScriptFile += tempArr;
      }
      else {
        bslScriptFile += "\t\t" + typeNode.getText() + " " +
```

```
            (typeNode.getNextSibling()).getText() + " = new Sequence();\n";
        type = typeNode.getText();
        typeNode = typeNode.getNextSibling();
        id = typeNode.getText();

        if (typeNode.getNextSibling() != null) {
          //handle the type part
          typeNode = typeNode.getNextSibling();
          if ( (typeNode.getText()).equals("type")) {
            AST temp = typeNode.getFirstChild();
            bslScriptFile += "\t\t" + id + ".setType(\"" + temp.getText() +
                "\");\n";
            if (typeNode.getNextSibling() != null) {
              typeNode = typeNode.getNextSibling();
            }
          }
          else {
            bslScriptFile += "\t\t" + id + ".setType(\"UNKNOWN\");\n";
          }

          if ( (typeNode.getText()).equals("=")) {
            bslScriptFile += "\t\t" + id + ".setSequence(\"" +
                (typeNode.getFirstChild()).getText() + "\");\n";
          }
        }
        else {
          bslScriptFile += "\t\t" + id + ".setType(\"UNKNOWN\");\n";
        }
      }
    }
    else if ((typeNode.getText()).equals("BuildingBlock")) {
      if ( ( (typeNode.getNextSibling()).getNextSibling()) != null && ( (
(typeNode.getNextSibling()).getNextSibling()).getText()).equals(
          "[") ) {
        AST arrayPart = (typeNode.getNextSibling()).getNextSibling();
        String name = (typeNode.getNextSibling()).getText();
        bslScriptFile += "\t\t" + typeNode.getText() + " " +
            name;
        id = name;
        type = "BuildingBlock";
        String tempArr = "[] = new BuildingBlock[";
        try {
          int firstNum =
Integer.parseInt((arrayPart.getFirstChild()).getText());
          tempArr += firstNum + "]";
          tempArr += ";\n";
          typeNode = (typeNode.getNextSibling()).getNextSibling();
          String seqType = "UNKNOWN";
          if(typeNode.getNextSibling() != null){
            if(((typeNode.getNextSibling()).getText()).equals("type")){
              seqType =
((typeNode.getNextSibling()).getFirstChild()).getText();
            }
            else{
              System.out.println("BSL Error <67>: Unexpected token here: " +
                                 (typeNode.getNextSibling()).getText());
```

```
              System.out.println("\nError encountered on line: " +
lineCount);
              System.exit(1);
            }
            if(((typeNode.getNextSibling()).getNextSibling()) != null){
              System.out.println("BSL Error <67>: Unexpected token here: " +
                                 (typeNode.getNextSibling()).getText());
              System.out.println("\nError encountered on line: " +
lineCount);
              System.exit(1);
            }
          }
          for(int i = 0; i < firstNum; i++){
            tempArr+="\t\t"+ name + "[" + i +"]" +
                ".setType(\"" + seqType + "\");\n";
          }
          if((arrayPart.getFirstChild()).getNextSibling() != null){
            System.err.println("BSL Error <66>: Array dimension greater than
1");
            System.out.println("\nError encountered on line: " + lineCount);
            System.exit(1);
          }
        }
        catch (NumberFormatException e) {
          System.out.println("BSL Error <68>: Number is expected");
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }
        bslScriptFile += tempArr;

      }
      else {
        bslScriptFile += "\t\t" + typeNode.getText() + " " +
            (typeNode.getNextSibling()).getText() + " = new
BuildingBlock();\n";
        type = typeNode.getText();
        typeNode = typeNode.getNextSibling();
        id = typeNode.getText();

        if (typeNode.getNextSibling() != null) {
          //handle the type part
          if ( (typeNode.getText()).equals("type")) {
            AST temp = typeNode.getFirstChild();
            bslScriptFile += "\t\t" + id + ".setBBType(\"" + temp.getText() +
                "\");\n";
            if (typeNode.getNextSibling() != null) {
              typeNode = typeNode.getNextSibling();
            }
          }
          else {
            bslScriptFile += "\t\t" + id + ".setBBType(\"UNKNOWN\");\n";
          }

          if ( (typeNode.getText()).equals("=")) {
            typeNode = typeNode.getNextSibling();
            bslScriptFile += "\t\t" + id + ".setBB(\"" +
                (typeNode.getFirstChild()).getText() + "\");\n";
```

```
          }
        }
      }
    }
    else {
      bslScriptFile += "\t\t" + typeNode.getText() + " " +
          (typeNode.getNextSibling()).getText();
      type = typeNode.getText();
      typeNode = typeNode.getNextSibling();
      id = typeNode.getText();

      if (typeNode.getNextSibling() != null) {
        typeNode = typeNode.getNextSibling();

        //check for a array part
        if ((typeNode.getText()).equals("[")) {
          String tempArScript = "";
          tempArScript += "[] = new " + type + "[";
          if (typeNode.getFirstChild() != null) {
            AST arrayAccess = typeNode.getFirstChild();

            tempArScript += arrayAccess.getText() + "]";

            if (arrayAccess.getNextSibling() != null) {
              arrayAccess = arrayAccess.getNextSibling();
              tempArScript = "[]" + tempArScript;

              tempArScript += "[" + arrayAccess.getText() + "]";
            }

            bslScriptFile += tempArScript + ";\n";
          }

          if (typeNode.getNextSibling() != null) {
            typeNode = typeNode.getNextSibling();
          }
        }

        //there is an equals part!
        if ((typeNode.getText()).equals("=")) {
          bslScriptFile += " = " + (typeNode.getFirstChild()).getText() +
";\n";
        }
        else{
          bslScriptFile += ";\n";
        }
      }
      else{
        bslScriptFile += ";\n";
      }

    }
    //set up the symbol table with the id and type
    idHash.put(id, type);
  }

  private static void writeFor(AST node){
```

```
    bslScriptFile += "\t\tfor(";

    if(((node.getFirstChild()).getText()).equals("decls")){
      writeDeclaration(node.getFirstChild());
    }
    AST child = node.getFirstChild();
    if(child.getNextSibling() != null){
      String temp = "";
      temp = writeCond(child.getNextSibling(), temp);
      bslScriptFile += "\t\t" + temp;
    }
    child = child.getNextSibling();
    if(child.getNextSibling() != null){
      handleAssign(child.getNextSibling());
    }
    bslScriptFile += ";";

    if(child.getNextSibling() != null){
      bslScriptFile += (child.getNextSibling()).getText() + "=";
      child = child.getNextSibling();
      temp1 = "";
      bslScriptFile +=
handleAssign((child.getNextSibling()).getFirstChild());
    }

    bslScriptFile += "){\n";
    child = child.getNextSibling();
    if (child.getNextSibling() != null) {
       child = (child.getNextSibling()).getFirstChild();
       while (child != null) {
         handleNode(child);
         lineCount ++;
         child = child.getNextSibling();
       }
     }
     bslScriptFile += "\n\t\t}\n";

  }
  private static void writeIf(AST node) {
    AST ifNode = node.getFirstChild();
    bslScriptFile += "\t\tif (";
    //conditionals
    bslScriptFile += writeCond(ifNode, "") + ") {\n";
    AST elsePart = null;
    //write body
    if (ifNode.getNextSibling() != null) {
      //check to see if else or else if exists
      if ((ifNode.getNextSibling()).getNextSibling() != null) {
        elsePart = (ifNode.getNextSibling()).getNextSibling();
      }
      ifNode = (ifNode.getNextSibling()).getFirstChild();
      while (ifNode != null) {
        handleNode(ifNode);
        lineCount ++;
        ifNode = ifNode.getNextSibling();
      }
    }
```

```
    //handle else and elseif parts
    while (elsePart != null) {
      bslScriptFile += "\t\t}\n";
      if ((elsePart.getText()).equals("elseif")) {
        bslScriptFile += "\t\telse if(";
        ifNode = elsePart.getFirstChild();
        bslScriptFile += writeCond(ifNode, "") + ") {\n";
        ifNode = (ifNode.getNextSibling()).getFirstChild();
      }
      else if ((elsePart.getText()).equals("else")) {
        bslScriptFile += "\t\telse {\n";
        elsePart = elsePart.getNextSibling();
        ifNode = elsePart.getFirstChild();
      }
      while (ifNode != null) {
        handleNode(ifNode);
        lineCount ++;
        ifNode = ifNode.getNextSibling();
      }
      elsePart = elsePart.getNextSibling();
    }

    bslScriptFile += "\t\t}\n";
  }

  private static void writeWhile(AST node) {
    AST ifNode = node.getFirstChild();
    bslScriptFile += "\t\twhile (";
    //conditionals
    bslScriptFile += writeCond(ifNode, "") + ") {\n";

    //write body
    if (ifNode.getNextSibling() != null) {
      ifNode = (ifNode.getNextSibling()).getFirstChild();
      while (ifNode != null) {
        handleNode(ifNode);
        lineCount ++;
        ifNode = ifNode.getNextSibling();
      }
    }

    bslScriptFile += "\t\t}\n";
  }

  private static String writeCond(AST node, String temp) {
    if ((node.getText()).equals("&&") || (node.getText()).equals("||")) {
      AST tempAST = node.getFirstChild();
      temp += " (" + writeCond(tempAST, temp) + ") " + node.getText() + " ("
+ writeCond(tempAST.getNextSibling(), temp) + ") ";
    }
    else if ((node.getText()).equals("==") || (node.getText()).equals("!=")
             || (node.getText()).equals(">")
             || (node.getText()).equals("<")
             || (node.getText()).equals(">=")
             || (node.getText()).equals("<=")) {
      AST firstID = node.getFirstChild();
```

```
    AST sib = firstID.getNextSibling();
    if ( (sib.getText()).equals("prop") || (sib.getText()).equals("[") ||
        (sib.getText()).equals("(")) {
     temp1 = "";
      temp += handleAssign(firstID) + node.getText() +
          writeCond(sib.getNextSibling(), temp);
    }
    else {
      temp += writeCond(firstID, temp) + " " + node.getText() + " " +
          writeCond(firstID.getNextSibling(), temp);
    }
  }
  else{
    if(node.getNextSibling() != null){
      AST sib = node.getNextSibling();
      if ( (sib.getText()).equals("prop") || (sib.getText()).equals("[") ||
           (sib.getText()).equals("(")) {
        temp1 = "";
        temp += handleAssign(node);
      }
      else
        temp += node.getText();
    }
    else
      temp += node.getText();
  }
  return temp;
}

private static String writeMcall(String methodName, AST node) {
  if (methodHash.containsKey(methodName)) {
    MData currentMData = (MData)methodHash.get(methodName);
    String methodCall = currentMData.library;

    String[] argsTypes = currentMData.argTypes;

    if (methodName.equals("Print")) {
      node = node.getFirstChild();

      if (idHash.containsKey(node.getText())){
        temp1 = "";
        methodCall += handleAssign(node);
      }
      else {
        temp1 = "";
        methodCall += "\"" + handleAssign(node) + "\"";
      }

      if (node.getNextSibling() != null) {
        node = node.getNextSibling();
        if ((node.getText()).equals("[") || (node.getText()).equals("("))
          node = node.getNextSibling();

        if (node != null)
          methodCall += " , \"" + node.getText() + "\"";
      }
```

```
        methodCall += ")";
      }
      else {
        int numberOfArgs = currentMData.numArgs;
        int currentArPos = 0;
        node = node.getFirstChild();
        while (numberOfArgs > 0) {
          if (node.getNextSibling() != null) {
            if (((node.getNextSibling()).getText()).equals("[") ||
                ((node.getNextSibling()).getText()).equals("(")) {
              methodCall += handleAssign(node);
              node = node.getNextSibling();

              numberOfArgs --;
              if (numberOfArgs > 0) {
                node = node.getNextSibling();
                methodCall += " , ";
              }
            }
          }

          if (numberOfArgs > 0) {
            String argName = node.getText();
            if (idHash.containsKey(argName)) {
              String type = (String)idHash.get(argName);
              if ((argsTypes[currentArPos]).equals(type)) {
                methodCall += argName;
              }
              else {
                System.out.println("BSL Error <97>: Incorrect type. " +
argsTypes[currentArPos] + " expected." +
                                   " Found: " + type);
                System.out.println("\nError encountered on line: " +
lineCount);
                System.exit(1);
              }
            }
            else {
              //check to see if this name is a character, user cannot
              try {
                Integer.parseInt(argName);
                if ((argsTypes[currentArPos]).equals("int")) {
                  methodCall += argName;
                }
                else {
                  System.out.println("BSL Error <97>: Incorrect type. " +
argsTypes[currentArPos] + " expected." +
                                     " Found unkown type named: " + argName);
                  System.out.println("\nError encountered on line: " +
lineCount);
                  System.exit(1);
                }
              }
              catch (NumberFormatException e) {
                System.out.println("BSL <limit reached>: Please don't use non
Number constants");
```

```
                System.out.println("\nError encountered on line: " +
lineCount);
                System.exit(1);
              }
            }
          }
          numberOfArgs --;
          // if there were two arguments, print out the second argument
          if (numberOfArgs > 0) {
            methodCall += " , ";
          }
          currentArPos ++;
          node = node.getNextSibling();
        }
        //check if more params than args
        if (node != null){
          System.out.println("BSL Error <158>: More arguments provided than
necessary.");
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }

        methodCall += ")";
      }
      return methodCall;
    }
    else {
      System.out.println("BSL Error <105>: Unknown Method : " + methodName);
      System.out.println("\nError encountered on line: " + lineCount);
      System.exit(1);
    }
    return "";
  }

  private static String writeProp (String propOwner, AST node) {
    node = node.getFirstChild();
    String prop = node.getText();
    if (idHash.containsKey(propOwner)) {
      String type = (String)idHash.get(propOwner);
      if (type.equals("Sequence")) {
        if (prop.equals("length"))
          return propOwner + ".getLength()";
        else if (prop.equals("seqType"))
          return propOwner + ".getType()";
        else if (prop.equals("value"))
          return propOwner + ".value()";
      }
      else {
        if (prop.equals("length"))
          return propOwner + "." + prop;
        else {
          System.out.println("BSL Error <95>: Unknown property for this
object: " + prop);
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }
      }
```

```
    }
    else {
      System.out.println("BSL Error <98>: Decleration necessary: " +
propOwner);
      System.out.println("\nError encountered on line: " + lineCount);
      System.exit(1);
    }

    return "";
  }

  private static String temp1 = "";
  private static String previousId = ""; //need for writing method calls

  private static String handleAssign(AST node){

    if(node.getFirstChild() != null &&
(node.getFirstChild()).getNextSibling() == null){
        temp1 += handleAssign(node.getFirstChild());
    }

    else if(node.getFirstChild() != null &&
(node.getFirstChild()).getNextSibling() != null){
      AST child = node.getFirstChild();
      previousId = child.getText();

      if(((child.getNextSibling()).getText()).equals("mcall")) {
        temp1 += writeMcall(previousId, child.getNextSibling());
        child = child.getNextSibling();
        if (child.getNextSibling() != null) {
          temp1 += " " + node.getText() + " ";
          handleAssign(child.getNextSibling());
        }
      }
      else if (((child.getNextSibling()).getText()).equals("prop")){
        temp1 += writeProp(previousId, child.getNextSibling());
        child = child.getNextSibling();
        if (child.getNextSibling() != null) {
          temp1 += " " + node.getText() + " ";
          handleAssign(child.getNextSibling());
        }
      }
      else if (((child.getNextSibling()).getText()).equals("[")){
        AST temp = (child.getNextSibling()).getFirstChild();
        String firstNum = temp.getText();
        String secondNum = "";
        boolean twoNum = false;
        if (temp.getNextSibling() != null) {
          twoNum = true;
          secondNum = (temp.getNextSibling()).getText();
        }

        temp1 += child.getText() + "[" + firstNum + "]";
        if (twoNum)
          temp1 += "[" + secondNum + "]";

        child = child.getNextSibling();
```

```
        if (child.getNextSibling() != null) {
          temp1 += " " + node.getText() + " ";
          handleAssign(child.getNextSibling());
        }
      }
      else if (((child.getNextSibling()).getText()).equals("(")){
        if (idHash.containsKey(previousId)) {
          String type = (String)idHash.get(previousId);

          if (type.equals("Sequence")) {
            AST temp = (child.getNextSibling()).getFirstChild();
            String firstNum = temp.getText();
            String secondNum = "";
            boolean twoNum = false;
            if (temp.getNextSibling() != null) {
              twoNum = true;
              secondNum = (temp.getNextSibling()).getText();
            }

            try {
              if (twoNum) {
                temp1 += previousId + "." + "getSequence(";
                temp1 += Integer.parseInt(firstNum) + " , ";
                temp1 += Integer.parseInt(secondNum) + ")";
              }
              else {
                temp1 += previousId + "." + "getBlock(";
                temp1 += Integer.parseInt(firstNum) + ")";
              }
            }
            catch (NumberFormatException e) {
              System.out.println("BSL Error <68>: Number is expected");
              System.out.println("\nError encountered on line: " +
lineCount);
              System.exit(1);
            }
          }
          else {
            System.out.println("BSL Error <158>: Not Correct Type: " + type);
            System.out.println("\nError encountered on line: " + lineCount);
            System.exit(1);
          }
        }
        else {
          System.out.println("BSL Error <98>: Decleration necessary: " +
previousId);
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }

        child = child.getNextSibling();
        if (child.getNextSibling() != null) {
          temp1 += " " + node.getText() + " ";

          handleAssign(child.getNextSibling());
        }
      }
```

```
      else{
        handleAssign(child);
        temp1 += " " + node.getText() + " ";
        handleAssign(child.getNextSibling());
      }
    }

    else if (node.getNextSibling() != null &&
((node.getNextSibling()).getText()).equals("prop")) {
      temp1 += writeProp(node.getText(), node.getNextSibling());
    }

    else if (node.getNextSibling() != null &&
((node.getNextSibling()).getText()).equals("mcall")) {
      temp1 += writeMcall(node.getText(), node.getNextSibling());
    }

    else if (node.getNextSibling() != null &&
((node.getNextSibling()).getText()).equals("[")) {
      AST temp = (node.getNextSibling()).getFirstChild();
      String firstNum = temp.getText();
      String secondNum = "";
      boolean twoNum = false;
      if (temp.getNextSibling() != null) {
        twoNum = true;
        secondNum = (temp.getNextSibling()).getText();
      }

      temp1 += node.getText() + "[" + firstNum + "]";
      if (twoNum)
        temp1 += "[" + secondNum + "]";
    }

    else if (node.getNextSibling() != null &&
((node.getNextSibling()).getText()).equals("(")) {
      if (idHash.containsKey(previousId)) {
        String type = (String)idHash.get(node.getText());

        if (type.equals("Sequence")) {
          AST temp = (node.getNextSibling()).getFirstChild();
          String firstNum = temp.getText();
          String secondNum = "";
          boolean twoNum = false;
          if (temp.getNextSibling() != null) {
            twoNum = true;
            secondNum = (temp.getNextSibling()).getText();
          }

          try {
            if (twoNum) {
              temp1 += node.getText() + "." + "getSequence(";
              temp1 += Integer.parseInt(firstNum) + " , ";
              temp1 += Integer.parseInt(secondNum) + ")";
            }
            else {
              temp1 += node.getText() + "." + "getBlock(";
              temp1 += Integer.parseInt(firstNum) + ")";
```

```
            }
          }
          catch (NumberFormatException e) {
            System.out.println("BSL Error <68>: Number is expected");
            System.out.println("\nError encountered on line: " + lineCount);
            System.exit(1);
          }
        }
        else {
          System.out.println("BSL Error <158>: Not Correct Type: " + type);
          System.out.println("\nError encountered on line: " + lineCount);
          System.exit(1);
        }
      }
      else {
        System.out.println("BSL Error <98>: Decleration necessary: " +
node.getText());
        System.out.println("\nError encountered on line: " + lineCount);
        System.exit(1);
      }
    }

    else {
      if (idHash.containsKey(node.getText())) {
        temp1 += node.getText();
      }
      else {
        try {
          int num = Integer.parseInt(node.getText());
          temp1 += node.getText();
        }
        catch (NumberFormatException e) {
          temp1 += "\"" + node.getText() + "\"";
        }
      }
    }

    return temp1;
  }

  private static void handleMethod(AST node) {
    String returnType = node.getText();
    node = node.getNextSibling();
    String methodName = node.getText();
    node = node.getNextSibling();
    bslScriptFile += "\t" + "private static " + returnType + " " +
methodName;
    bslScriptFile += "(";
    int numArgs =  0;
    String[] argTypesMethod = new String[1];
    String[] argsMethod = new String[1];
    if ((node.getText()).equals("args")) {
      AST child = node.getFirstChild();

      while (child != null) {
        numArgs++;
```

```
        bslScriptFile += child.getText() + " " +
(child.getNextSibling()).getText();

        idHash.put((child.getNextSibling()).getText(), child.getText());
        child = child.getNextSibling();
        if (child.getNextSibling() != null) {
          bslScriptFile += " , ";
        }
        child = child.getNextSibling();
      }

      //set up the array with the necessary types
      argTypesMethod = new String[numArgs];
      argsMethod = new String[numArgs];
      child = node.getFirstChild();
      for(int i=0; i < numArgs; i++){
        //arg type
        argTypesMethod[i] = child.getText();
        child = child.getNextSibling();
        //arg name
        argsMethod[i] = child.getText();
        child = child.getNextSibling();
      }

      node = node.getNextSibling();
    }

    // write the method body
    bslScriptFile += ") {\n";
    while (node != null) {
      handleNode(node);
      lineCount ++;
      node = node.getNextSibling();
    }
    bslScriptFile += "\t}\n";

    //add this method to the methodHash
    MData methodData = new MData(methodName, numArgs, argsMethod,
argTypesMethod, methodName + "(", returnType);
    if (methodHash.containsKey(methodName)) {
      //this method exists so replace
      methodHash.put(methodName, methodData);
    }
    else {
      //just put it in
      methodHash.put(methodName, methodData);
    }

    //clear the symbol hash table
    idHash = new Hashtable();

    lineCount ++;
  }

private static void handleNode(AST node) {
    if((node.getText()).equals("method")){
      handleMethod(node.getFirstChild());
```

```
    }

    node = node.getFirstChild();
    String statementType = node.getText();

    if (statementType.equals("decls")) {
      writeDeclaration(node);
    }
    else if (statementType.equals("if")) {
      writeIf(node);
    }
    else if (statementType.equals("for")) {
      writeFor(node);
    }
    else if (statementType.equals("while")) {
      writeWhile(node);
    }
    else if(statementType.equals("exit")){
      bslScriptFile += "\tSystem.exit(0);\n";
    }
    else {
      if (node.getNextSibling() != null) {
        String yellowBelly = node.getText();
        node = node.getNextSibling();
        if ((node.getText()).equals("=")) {
          if (idHash.containsKey(yellowBelly) &&
              ((String) idHash.get(yellowBelly)).equals("Sequence")) {
            bslScriptFile += yellowBelly + ".setSequence(";
            System.out.println(bslScriptFile);
          }
          else
            bslScriptFile += "\t\t" + yellowBelly + " " + node.getText() + "
";
          temp1 = "";
          bslScriptFile += handleAssign(node);
          if (idHash.containsKey(yellowBelly) &&
              ((String) idHash.get(yellowBelly)).equals("Sequence")) {
            bslScriptFile += ");\n";
          }
          else
            bslScriptFile += ";\n";
        }
        else if ((node.getText()).equals("prop")) {
          bslScriptFile += "\t\t" + writeProp(yellowBelly, node) + ";\n";
        }
        else if ((node.getText()).equals("mcall")) {
          bslScriptFile += "\t\t" + writeMcall(yellowBelly, node) + ";\n";
        }
        else if ((node.getText()).equals("[")) {
          AST temp = node.getFirstChild();
          String firstNum = temp.getText();
          String secondNum = "";
          boolean twoNum = false;
          if (temp.getNextSibling() != null) {
            twoNum = true;
            secondNum = (temp.getNextSibling()).getText();
          }
```

```
          bslScriptFile += "\t\t" + yellowBelly + "[" + firstNum + "]";
           if (twoNum)
             bslScriptFile += "[" + secondNum + "]";

           if (node.getNextSibling() != null) {
             node = node.getNextSibling();
             if ((node .getText()).equals("=")) {
               bslScriptFile += " " + node.getText() + " ";
               temp1 = "";
               bslScriptFile += handleAssign(node)+";\n";
             }
           }
           else
             bslScriptFile += ";\n";
         }
         else if ((node.getText()).equals("(")) {
           if (idHash.containsKey(yellowBelly)) {
             String type = (String)idHash.get(yellowBelly);

             if (type.equals("Sequence")) {
               AST temp = node.getFirstChild();
               String firstNum = temp.getText();
               String secondNum = "";
               boolean twoNum = false;
               if (temp.getNextSibling() != null) {
                 twoNum = true;
                 secondNum = (temp.getNextSibling()).getText();
               }

               try {
                 if (twoNum) {
                   bslScriptFile += "\t\t" + yellowBelly + "." +
"getSequence(";
                   bslScriptFile += Integer.parseInt(firstNum) + " , ";
                   bslScriptFile += Integer.parseInt(secondNum) + ")";
                 }
                 else {
                   bslScriptFile += "\t\t" + yellowBelly + "." + "getBlock(";
                   bslScriptFile += Integer.parseInt(firstNum) + ")";
                 }
               }
               catch (NumberFormatException e) {
                 System.out.println("BSL Error <68>: Number is expected");
                 System.out.println("\nError encountered on line: " +
lineCount);
                 System.exit(1);
               }
               bslScriptFile += ";\n";
             }
             else {
               System.out.println("BSL Error <158>: Not Correct Type: " +
type);
               System.out.println("\nError encountered on line: " +
lineCount);
               System.exit(1);
             }
```

```
          }
          else {
            System.out.println("BSL Error <98>: Decleration necessary: " +
yellowBelly);
            System.out.println("\nError encountered on line: " + lineCount);
            System.exit(1);
          }
        }
      }//if next sibling is not null in else
      else {
          temp1 = "";
          bslScriptFile += "\t\t" + handleAssign(node) + ";\n";
      }//end of the if elses for the next sibling
    }//matches else for the main ifelses
  }

  private static void initIDST() {
    idHash = new Hashtable();
  }

  private static void initMethodST() {
    methodHash = new Hashtable();

    String[] argsAlign = {"seq1", "seq2"};
    String[] argTypesAlign = {"Sequence", "Sequence"};
    MData alignData = new MData("Align", 2, argsAlign, argTypesAlign,
"Aligner.Align(", "Sequence");
    methodHash.put("Align", alignData);

    String[] argsCompare = {"seq1", "seq2"};
    String[] argTypesCompare = {"Sequence", "Sequence"};
    MData compareData = new MData("Compare", 2, argsCompare, argTypesCompare,
"Comparer.Compare(", "boolean");
    methodHash.put("Compare", compareData);

    String[] argsComplement = {"seq1"};
    String[] argTypesComplement = {"Sequence"};
    MData complementData = new MData("Complement", 1, argsComplement,
argTypesComplement, "Complementor.Complement(", "Sequence");
    methodHash.put("Complement", complementData);

    String[] argsConsensus = {"seq1", "seq2"};
    String[] argTypesConsensus = {"Sequence", "Sequence"};
    MData consensusData = new MData("Consensus", 2, argsConsensus,
argTypesConsensus, "Combiner.Consensus(", "Sequence");
    methodHash.put("Consensus", consensusData);

    String[] argsFind = {"seq1", "seq2"};
    String[] argTypesFind = {"Sequence", "Sequence"};
    MData findData = new MData("Find", 2, argsFind, argTypesFind,
"Finder.find(", "int");
    methodHash.put("Find", findData);

    String[] argsFindAll = {"seq1", "seq2"};
    String[] argTypesFindAll = {"Sequence", "Sequence"};
    MData findAllData = new MData("FindAll", 2, argsFindAll, argTypesFindAll,
"Finder.findAll(", "int[]");
```

```
    methodHash.put("FindAll", findAllData);

    String[] argsFindPrimer = {"seq1", "seq2"};
    String[] argTypesFindPrimer = {"Sequence", "Sequence"};
    MData findPrimerData = new MData("FindPrimer", 2, argsFindPrimer,
argTypesFindPrimer, "Finder.findPrimer(", "int");
    methodHash.put("FindPrimer", findPrimerData);

    String[] argsPrint = {"value", "location"};
    String[] argTypesPrint = {"Generic", "Optional"};
    MData printData = new MData("Print", 2, argsPrint, argTypesPrint,
"Printer.print(", "void");
    methodHash.put("Print", printData);

    String[] argsReverse = {"seq1"};
    String[] argTypesReverse = {"Sequence"};
    MData reverseData = new MData("Reverse", 1, argsReverse, argTypesReverse,
"Finder.reverse(", "Sequence");
    methodHash.put("Reverse", reverseData);

    String[] argsTranslate = {"seq1", "form"};
    String[] argTypesTranslate = {"Sequence", "String"};
    MData translateData = new MData("Translate", 2, argsTranslate,
argTypesTranslate, "Finder.translate(", "Sequence");
    methodHash.put("Translate", translateData);

    String[] argsAccessFile = {"filename"};
    String[] argTypesAccessFile = {"String"};
    MData accessFileData = new MData("AccessFile", 1, argsAccessFile,
argTypesAccessFile, "SequenceBuilder.accessFile(", "Ssequence[]");
    methodHash.put("AccessFile", accessFileData);

    /*
         String[] argsByteFile = {"filename"};
         String[] argTypesByteFile = {"String"};
         MData byteFileData = new MData("ByteFile", 1, argsByteFile,
argTypesByteFile, "SequenceBuilder.byteFile(", "byte[]");
         methodHash.put("ByteFile", byteFileData);
     */
  }


  public static void main(String[] args){
    try{
      if (args.length > 1 || args.length == 0) {
        System.out.println("BSL Error: Correct usage -- java BSL_TreeParser
<filename>.bsl");
        System.exit(1);
      }

      String fileName = args[0];

      if (!fileName.endsWith(".bsl") && !fileName.endsWith(".BSL")){
        System.out.println("BSL Error: Only files with the bsl extension are
allowed.");
        System.exit(1);
      }
```

```
      File tempFile = new File(fileName);

      if (!tempFile.exists()) {
        System.out.println("BSL Error: oooh MY Gawd. The file does not
exist.");
        System.exit(1);
      }

      BufferedReader input = new BufferedReader(new FileReader(fileName));
      BSL_Lexer lexer = new BSL_Lexer(input);
      BSL_Parser parser = new BSL_Parser(lexer);

      //Parse the script
      parser.bslscript();
      initMethodST();
      initIDST();
      CommonAST t = (CommonAST) parser.getAST();
      //ASTFrame frame = new ASTFrame("AST JTree", t);
      //frame.setVisible(true);
      //write the header for the java file
      fileName = fileName.substring(0,fileName.length()-4);
      bslScriptFile = "public class " +fileName + " {\n";
      boolean wroteMainMethod = false;

      AST currentNode = t.getFirstChild();

      while (currentNode != null) {
        if (!(currentNode.getText()).equals("method") && !wroteMainMethod) {
          // no method calls
          bslScriptFile +=  "\tpublic static void main(String[] args) {\n";
          wroteMainMethod = true;
        }
        else {
          //handle this method
        }

         //handle the node
         handleNode(currentNode);
         lineCount ++;
         currentNode = currentNode.getNextSibling();
      }

      //end Main
      bslScriptFile += "\t}\n";
      //end class
      bslScriptFile += "}\n";
      try {
        File javaFile = new File(fileName + ".java");
        FileWriter writer = new FileWriter(javaFile);
        writer.write(bslScriptFile);
        writer.flush();
        writer.close();
        // automatically compile the generated java file
        Runtime.getRuntime().exec("cmd /c javac " + fileName + ".java");
        System.out.println("BSL Compilation done. Please run your file using:
"+
```

```
                            "java " + fileName);
      } catch(IOException e){
        e.printStackTrace();
      }
      //System.out.println(bslScriptFile);
    }
    catch(Exception e){
      e.printStackTrace();
      System.exit(0);
    }
  }
}
```

```
/**
 * Alinger takes care of align method call.
 * @author Jay Kota
 *  jrk103
 */

import java.lang.*;

public class Aligner {

  private static char DASH='-';

  private static int gapPenalty = -3;
  private static int matchScore = 6;
  private static int mismatchScore = -1;

  private static String sequence1;
  private static String sequence2;
  private static String alignedSeq1;
  private static String alignedSeq2;

  public Aligner() {
    gapPenalty = -3;
    matchScore = 6;
    mismatchScore = -1;
  }

  public Aligner(int gapPenalty_, int matchScore_, int mismatchScore_) {
    gapPenalty = gapPenalty_;
    matchScore = matchScore_;
    mismatchScore = mismatchScore_;
  }

  private static int[][] buildMatrix(int[][] scores) {
    try {
      scores = new int[sequence1.length()+1][sequence2.length()+1];

      // initialize the extremes
      for (int i=0; i < sequence1.length(); i++) {
        scores[i][0]=0;
      }
      for (int i=0; i < sequence2.length(); i++) {
        scores[0][i]=0;
      }
```

```
      char seq1Char, seq2Char;
      // fill in the matrix
      for (int i=1; i <= sequence1.length(); i++) {
        seq1Char = sequence1.charAt(i-1);
        for (int j=1; j <= sequence2.length(); j++) {
          seq2Char = sequence2.charAt(j-1);
          scores[i][j] = max(scores[i-1][j]+gapPenalty,
                             scores[i-1][j-1]+getScore(seq1Char, seq2Char),
                             scores[i][j-1]+gapPenalty);
        }
      }

      return scores;
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("BSL Error <10> : Internal array access error in
Align.");
      System.exit(0);
    }
    catch (StringIndexOutOfBoundsException e) {
      System.out.println("BSL Error <11> : Invalid Sequence access error in
Align.");
      System.exit(0);
    }
    return scores;
}
  private static int max(int i, int j, int k) {
    if (i > j) {
      if (i > k)
        return i;
      else
        return k;
    }
    else {
      if (j > k)
        return j;
      else
        return k;
    }
  }

  private static int getScore(char seq1Char, char seq2Char) {
    if (Character.isLetter(seq1Char)) {
      seq1Char = Character.toUpperCase(seq1Char);
    }
    if (Character.isLetter(seq2Char)) {
      seq2Char = Character.toUpperCase(seq2Char);
    }

    int char1Val = Character.getNumericValue(seq1Char);
    int char2Val = Character.getNumericValue(seq2Char);

    if (char1Val == char2Val) {
      return matchScore;
    }
    else {
      return mismatchScore;
```

```
    }
  }

  private static void backTrack(int[][] scores) {
    //System.out.println("backTracking");
    try {
      int j = sequence2.length();
      int i = 0; int temp = 0;

      for (int a = 0; a <= sequence1.length(); a++) {
        if (scores[a][j] > temp) {
          temp = scores[a][j];
          i = a;
        }
      }

      alignedSeq1 = "";
      alignedSeq2 = "";

      /*for (int z=0; z <= sequence1.length(); z++) {
        for (int y=0; y <= sequence2.length(); y++) {
          System.out.print(scores[z][y]);
        }
        System.out.println();
      }
      System.out.println("start at " + i);*/

      while (scores[i][j]>0) {
/*        System.out.println(scores[i][j]);
        System.out.println(scores[i-1][j]+gapPenalty);
        System.out.println(getScore(sequence1.charAt(i-1),
                                    sequence2.charAt(j-1)));
        System.out.println(sequence2.charAt(j-1));
        System.out.println();*/

        // there is a gap in sequence 2
        if (scores[i][j] == scores[i-1][j]+gapPenalty) {
          alignedSeq1 = sequence1.charAt(i-1) + alignedSeq1;
          alignedSeq2 = DASH + alignedSeq2;
          i = i-1;
        }
        // matching symbols
        else if (scores[i][j] == scores[i-1][j-
1]+getScore(sequence1.charAt(i-1),
          sequence2.charAt(j-1))) {
          alignedSeq1 = sequence1.charAt(i-1) + alignedSeq1;
          alignedSeq2 = sequence2.charAt(j-1) + alignedSeq2;
          i = i-1;
          j = j-1;
        }
        // gap in sequence 1
        else {
          alignedSeq1 = DASH + alignedSeq1;
          alignedSeq2 = sequence2.charAt(j-1) + alignedSeq2;
          j = j-1;
        }
      }
```

```
      // handle space at the extremes
      if (alignedSeq1.length() < sequence2.length()) {
        int space = sequence2.length() - alignedSeq1.length();
        for (int a = 1; a <= space; a++) {
          alignedSeq1 = DASH + alignedSeq1;
          alignedSeq2 = sequence2.charAt(j-a) + alignedSeq2;
        }
      }
      else if (alignedSeq2.length() < sequence1.length()) {
        int space = sequence1.length() - alignedSeq2.length();
        int alseq1len = alignedSeq1.length();
        for (int a = 1; a <= space; a++) {
          alignedSeq2 = alignedSeq2 + DASH;
          alignedSeq1 = alignedSeq1 + sequence1.charAt(alseq1len + (a-1));
        }
      }
      //System.out.println(" AH " + sequence1);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("BSL Error <10> : Internal array access error in
Align " +
                         "in backTrack().");
      System.exit(0);
    }
    catch (StringIndexOutOfBoundsException e) {
      System.out.println("BSL Error <11> : Invalid Sequence access error in
Align " +
                         "in backTrack().");
      System.exit(0);
    }
}

  public static Sequence Align(Sequence uno, Sequence dos) {
    // get the sequences from the Sequence
    sequence1 = uno.value();
    sequence2 = dos.value();

    int[][] scores = new int[sequence1.length()+1][sequence2.length()+1];

    // check to make sure they are of the same type
    if (!(uno.getType()).equals(dos.getType())) {
      System.out.println("BSL Error <101> : Sequences of different types
cannot be aligned.");
      System.exit(0);
    }

    // fill the matrix
    scores = buildMatrix(scores);

    // build the aligned sequences
    backTrack(scores);

    // build the aligned sequence
    Sequence temp = new Sequence();
    temp.setType(uno.getType());
    temp.setSequence(alignedSeq1);
```

```
    return temp;
  }

  protected static String getSecondSequence() {
    return alignedSeq2;
  }
}
```

```
// package bsl;

/*
 * BuildingBlock.java
 * @author Jared Eng
 * cunix id: jwe37
 */


import java.io.*;
import java.awt.*;
import java.lang.*;



public class BuildingBlock{

  private String bb;
  private String bbType;

  public BuildingBlock() {
    bbType = Sequence.UNKNOWN;
  }

  public BuildingBlock(String str) {
    bbType = str;
  }

  public boolean isNucleotide(String str) {
    // If str is a nucleotide, return 1,
    // If not, return 0.
    boolean flag;
// BETWEEN 97 AND 122, capiltize it

    String strUpper = "";

    strUpper = str.toUpperCase();

    if (strUpper.equals("A"))
      flag=true;
    else if (strUpper.equals("C"))
      flag=true;
    else if (strUpper.equals("G"))
      flag=true;
    else if (strUpper.equals("T"))
      flag=true;
    else if (strUpper.equals("U"))
      flag=true;
    else
      flag=false;
```

```
  return flag;
}


public String getBBtype() {
  return (bbType);
}

public void setBB(String str) {

  try
  {
    if (isNucleotide(str)) {
      bb = str;
    }
    else
      throw new Exception("BSL Error <20>: Not a valid nucleotide."); ;
  } catch (Exception e)
  {
    System.err.println("BSL Error <20>: Not a valid nucleotide.");
    System.exit(1);
  }

}

public String toString() {
  return bb;
}

public String value() {
  return bb;
}

public boolean isBB() {
  boolean flag;
  if (this.isNucleotide(bb))
  {
    if (bbType.equals(Sequence.DNA))
      flag = true;
    else if (bbType.equals(Sequence.AMINOACID))
      flag = true;
    else if (bbType.equals(Sequence.UNKNOWN))
      flag = true;
    else
      flag = false;
  }
  else
    flag = false;
  return flag;
}
/*
   public static void main(String[] args) {
  System.out.println("\nInitializing...\n");

  BuildingBlock myBuildingBlock1 = new BuildingBlock("DNA");
  myBuildingBlock1.setBB("A");
```

```
    System.out.println("BuildingBlock = " + myBuildingBlock1.value());
    System.out.println("BuildingBlock Type = " +
myBuildingBlock1.getBBtype());
    System.out.println("BuildingBlock Confirmation = " +
myBuildingBlock1.isBB());

    BuildingBlock myBuildingBlock2 = new BuildingBlock("DNA");
    myBuildingBlock2.setBB("Z");
    System.out.println("\nBuildingBlock = " + myBuildingBlock2.value());
    System.out.println("BuildingBlock Type = " +
myBuildingBlock2.getBBtype());
    System.out.println("BuildingBlock Confirmation = " +
myBuildingBlock2.isBB());

     }
   */
}
```

```
// package bsl;

/*
 * Sequence.java
 * @author Jared Eng
 * cunix id: jwe37
 */

import java.io.*;
import java.awt.*;
import java.util.*;
import java.lang.*;

public class Sequence{
  /*
    DNA or RNA
   ------------------------------------------
   Symbol       Meaning      Nucleic Acid
   ------------------------------------------
    A            A           Adenine
    C            C           Cytosine
    G            G           Guanine
    T            T           Thymine
    U            U           Uracil
*/

  /*
   AMINOACID
    ------------------------------------------
    Symbol   Meaning
    ------------------------------------------
    A        Alanine
    B        Aspartic Acid, Asparagine
    C        Cystine
    D        Aspartic Acid
    E        Glutamic Acid
    F        Phenylalanine
    G        Glycine
    H        Histidine
    I        Isoleucine
```

```
  K        Lysine
  L        Leucine
  M        Methionine
  N        Asparagine
  P        Proline
  Q        Glutamine
  R        Arginine
  S        Serine
  T        Threonine
  V        Valine
  W        Tryptophan
  X        Unknown
  Y        Tyrosine
  Z        Glutamic Acid, Glutamine

 */
public static final String DNA = "DNA";
public static final String RNA = "RNA";
public static final String AMINOACID = "AMINOACID";
public static final String PROTEIN = "PROTEIN";
public static final String UNKNOWN = "UNKNOWN";
public static final String STRING = "STRING";

private String seqType;
private String seq;

public Sequence() {
  seqType = "UNKNOWN";
}

public Sequence(String str) {
  if (str.equals(DNA) || str.equals(RNA) ||
      str.equals(PROTEIN) || str.equals(UNKNOWN) || str.equals(STRING)) {
    seqType = str;
  }
  else{
    System.err.println("Error: Invalid sequence type");
    System.exit(1);
  }
}

public boolean isSequence() {
  return true;
}

private static boolean isAMINO(String str){
  boolean flag = false;
  String strUpper = "";
  String strTest = "";
  strUpper = str.toUpperCase();

  for (int i = 0; i < strUpper.length(); i++) {
    strTest = strUpper.substring(i, i + 1);
    if (strTest.equals("J"))
      return false;
    else if (strTest.equals("O"))
      return false;
```

```
    else if (strTest.equals("U"))
      return false;
    else {
      flag = true;
    }
  }
  return flag;
}

private static boolean isDNA(String str) {
  boolean flag = false;
  String strUpper = "";
  String strTest = "";
  strUpper = str.toUpperCase();

  for (int i=0; i < strUpper.length(); i++)
  {
    strTest = strUpper.substring(i,i+1);
    if (strTest.equals("A"))
      flag=true;
    else if (strTest.equals("C"))
      flag=true;
    else if (strTest.equals("G"))
      flag=true;
    else if (strTest.equals("T"))
      flag=true;
    else if(strTest.equals("-"))
      flag = true;
    else {
      return false;
    }
  }

  return flag;
}


public static boolean isRNA(String str) {
  boolean flag=false;
  String strUpper = "";
  String strTest = "";
  strUpper = str.toUpperCase();

  for (int i=0; i<strUpper.length(); i++)
  {
    strTest = strUpper.substring(i,i+1);
    if (strTest.equals("A"))
      flag=true;
    else if (strTest.equals("C"))
      flag=true;
    else if (strTest.equals("G"))
      flag=true;
    else if (strTest.equals("U"))
      flag=true;
    else if(strTest.equals("-"))
      flag = true;
```

```
    else {
      return false;
    }
  }

  return flag;
}
public String toString(){
  return seq;
}
public static boolean isProtein(String str) {
  boolean flag = false;
  if (isAMINO(str)){
    flag = true;
  }
  return flag;
}

public void setType(String str) {
  if (str.equals(DNA) || str.equals(RNA) || str.equals(AMINOACID) ||
      str.equals(PROTEIN) || str.equals(UNKNOWN) || str.equals(STRING)) {
    seqType = str;
  }
  else{
    System.err.println("Error: Invalid sequence type");
    System.exit(1);
  }
}

public String getType() {
  return seqType;
}

public String value() {
  return seq;
}

public String getSequence(int x, int y) {
  return (seq.substring(x,y));
}


public void setSequence(String str) {
  try {
    if ( seqType.equals(DNA) && isDNA(str))
      seq = str;
    else if ( seqType.equals(RNA) && isRNA(str))
      seq = str;
    else if ( seqType.equals(PROTEIN) && isAMINO(str))
      seq = str;
    else if ( seqType.equals(STRING) || seqType.equals(UNKNOWN))
      seq = str;
    else
      throw new Exception ("BSL Error <21>: Invalid Sequence.");
  } catch (Exception e) {
    System.out.println("BSL Error <21>: Invalid Sequence.");
    System.exit(1);
```

```
    }
  }


  /*
    public setSequence(String str) {
        seq = str;
    }
    public setSequence(String str) {
        seq = str;
    }

     public static void main(String[] args) {
      System.out.println("\nInitializing...\n");

        Sequence mySequence = new Sequence("RNA");
        mySequence.setType("DNA");
        mySequence.setSequence("ACGTACGT");
      System.out.println("Sequence = " + mySequence.value());
      System.out.println("Sequence Type = " + mySequence.getType());
      System.out.println("Sequence Substring = " + mySequence.Sequence(0,4));
      System.out.println("Sequence confirmation = " +
mySequence.isSequence());
     }
*/

}
```

```
/*
 * MData used by the treeparser in keeping track of methods
 * @author amna
 * cunix id: aq41
 */

public class MData{

  public String mName;
  public int numArgs;
  public String[] args;
  public String[] argTypes;
  public String library;
  public String returnType;

  public MData(String mName, int numArgs, String[] args, String[] argTypes,
String library, String returnType){
    this.mName = mName;
    this.numArgs = numArgs;
    this.args = args;
    this.argTypes = argTypes;
    this.library = library;
    this.returnType = returnType;
  }
}
```

```
/**
 * Combiner handles the consensus method call
 * @author Jay Kota
 * jrk103
 */
```

```
public class Combiner {

  private static char QUESTION = '?';

  private static Sequence sequence1;
  private static Sequence sequence2;
  private static String consensus;

  private static int confidence;

  public Combiner() {
    confidence = 100;
  }

  public static Sequence Consensus(Sequence seq1, Sequence seq2) {
    sequence1 = seq1;
    sequence2 = seq2;

    // check the types

    buildConsensus();

    // build consensus
    Sequence temp = new Sequence();
    temp.setType(seq1.getType());
    temp.setSequence(consensus);
    return temp;
  }

  private static void confidence(char currentBlock) {
    // DNA

    // RNA

    // AMINO ACID

    // PROTEIN
  }

  private static void buildConsensus() {
    Aligner myAligner = new Aligner();

    String alignedMother, alignedFather;

    alignedMother = (myAligner.Align(sequence1, sequence2)).value();
    alignedFather = myAligner.getSecondSequence();

    char seq1Char, seq2Char;

    consensus = "";

    // for DNA, RNA, AMINO ACID, PROTEIN
    for (int i=0; i < alignedMother.length(); i++) {
      seq1Char = alignedMother.charAt(i);
      seq2Char = alignedFather.charAt(i);
```

```
      if (seq1Char == seq2Char) {
        consensus = consensus + seq1Char;
        confidence(seq1Char);
      }
      else if (seq1Char == '-' && seq2Char != '-') {
        consensus = consensus + seq2Char;
        confidence(seq2Char);
      }
      else if (seq2Char == '-' && seq1Char != '-') {
        consensus = consensus + seq1Char;
        confidence(seq1Char);
      }
      else { //mismatch!
        consensus = consensus + QUESTION;
        confidence(QUESTION);
      }
    }

    /* for STRING AND UNKNOWN
    for (int i=1; i <= alignedMother.length(); i++) {
      seq1Char = alignedMother.charAt(i);
      seq2Char = alignedFather.charAt(i);

      if (seq1Char == seq2Char) {
        consensus = consensus + seq1Char;
      }
      else { //mismatch!
        consensus = consensus + '?';
      }
    }
    */
  }
}
```

```
// package bsl;

/*
 * Comparer.java handles the compare method call
 * @author Jared Eng
 * cunix id: jwe37
 */


import java.io.*;
import java.awt.*;
import java.lang.*;

public class Comparer {

      public static boolean Compare(Sequence tempSeq1, Sequence tempSeq2) {
          if (tempSeq1.value() == tempSeq2.value())
              return true;
          else
              return false;
      }

     public static void main(String[] args) {
          Sequence mySequence1 = new Sequence("DNA");
```

```
          mySequence1.setSequence("ACGT");

          Sequence mySequence2 = new Sequence("DNA");
          mySequence2.setSequence("ACGT");

         System.out.println("Sequence1 confirmation = " +
mySequence1.isSequence());
         System.out.println("Sequence2 confirmation = " +
mySequence2.isSequence());

          boolean same = Compare(mySequence1, mySequence2);
          System.out.println(mySequence1.value() + " and " +
mySequence2.value() + " are " + same);

     }

}
```

```
/**
 *
 * @author: Igor Marfin
 *
 * CS4118 PLT - BSL Compiler Class
 *
 *
 * class Complementor handles the complement method call
 */

//package bsl;

public class Complementor {

    private Sequence seq_bar;
    private final char DASH = '-';

    public Sequence Complement(Sequence s) {

      String sequence = s.value();
      String temp = "";
      String comSeq = "";

      boolean isDNA = (s.getType()).equalsIgnoreCase("DNA");
      boolean isRNA = (s.getType()).equalsIgnoreCase("RNA");;

      if (isDNA || isRNA) {

          for (int i=0; i < sequence.length(); i++) {
            if (sequence.charAt(i) == 'G')
                temp = "C";
            else if (sequence.charAt(i) == 'C')
                temp = "G";
            else if (sequence.charAt(i) == 'A') {
                if (isDNA)
                  temp = "T";
                else // must be an RNA sequence
                  temp = "U";
            }
            else if (sequence.charAt(i) == 'T') {
```

```
                if (isRNA) {
                  System.out.println("BSL Error<105>: Illegal character - T -
is found in an RNA sequence! Aborting complement operation.");
                  System.exit(1);
                }
                temp = "A";
            }
            else if (sequence.charAt(i) == 'U') {
                if (isDNA) {
                  System.out.println("BSL Error<105>: Illegal character - U -
is found in a DNA sequence! Aborting complement operation.");
                  System.exit(1);
                }
                temp = "A";
            }
            else if (sequence.charAt(i) == DASH)
                temp = "" + DASH;
            else {
                System.out.println("BSL Error<102>: Illegal character found
in the sequece! Aborting complement operation.");
                System.exit(1);
            }
            comSeq += temp;
          } // end of for loop
      }
      else {
          // if type isn't DNA or RNA, throw error
          System.out.println("BSL Error<101>: Illegal sequence type! Only RNA
and DNA sequences can be complemented.");
          System.exit(1);
      }
        seq_bar.setType(s.getType());
        seq_bar.setSequence(comSeq);
      return seq_bar;
    } // end of method

} // end of class
```

```
/**
 *
 * @author: Igor Marfin
 *
 * CS4118 PLT - BSL Compiler Class
 *
 *
 * class sequenceBuilder
 */

//package bsl;

import java.io.*;
/*
 * BuildingBlock.java
 * @author Jared Eng
 * cunix id: jwe37
 */

public class SequenceBuilder {
```

```
  public Sequence[] accessFile(String file) {
    Sequence[] seqArray = null;
    try {
      FileReader reader = new FileReader(file);
      BufferedReader in1 = new BufferedReader(reader);
      BufferedReader in2 = new BufferedReader(reader);
      String input1 = in1.readLine();
      String input2 = in2.readLine();
      // count number of lines in file
      int i = 0;
      int lines = 0;
      while (input1 != null) {
        lines++;
        input1 = in1.readLine();
      }
      seqArray = new Sequence[lines];
      while (input2 != null) {
        if (!(input2.equals(""))) {
          Sequence temp = new Sequence();
          temp.setSequence(input2);
          seqArray[i] = temp;
          i++;
          input2 = in2.readLine();
        }
      }
    }
    catch (IOException e) {
      System.out.println("BSL Error<10>: I/O error " + e);
      System.exit(0);
    }
    return seqArray;
  } // end of method

} // end of class
```

```
/**
 *
 * @author: Igor Marfin
 *
 * CS4118 PLT - BSL Compiler Class
 *
 *
 * class Printer
 */

//package bsl;

import java.io.*;

public class Printer {

  // printing to console as default value
  public static void print(Object obj) {
    if (obj == null)
      System.out.println("Warning! Trying to print a null value.");
    else
      System.out.println(obj.toString());
```

```
}

public static void print(int num) {
  System.out.println(num);
}

public static void print(char c) {
  System.out.println(c);
}

public static void print(boolean b) {
  System.out.println(b);
}

public static void print(byte bt) {
  System.out.println(bt);
}

public static void print(short st) {
  System.out.println(st);
}

public static void print(int num, String f) {
  try {
    FileWriter writer = new FileWriter(f);
    writer.write(num + "");
    writer.close();
  }
  catch (IOException e) {
    System.out.println("BSL Error<1>: I/O error " + e);
    System.exit(1);
  }
}

public static void print(char c, String f) {
  try {
    FileWriter writer = new FileWriter(f);
    writer.write(c + "");
    writer.close();
  }
  catch (IOException e) {
    System.out.println("BSL Error<1>: I/O error " + e);
    System.exit(1);
  }
}

public static void print(boolean b, String f) {
  try {
    FileWriter writer = new FileWriter(f);
    writer.write(b + "");
    writer.close();
  }
  catch (IOException e) {
    System.out.println("BSL Error<1>: I/O error " + e);
    System.exit(1);
  }
}
```

```
  public static void print(byte bt, String f) {
    try {
      FileWriter writer = new FileWriter(f);
      writer.write(bt + "");
      writer.close();
    }
    catch (IOException e) {
      System.out.println("BSL Error<1>: I/O error " + e);
      System.exit(1);
    }
  }

  public static void print(short st, String f) {
    try {
      FileWriter writer = new FileWriter(f);
      writer.write(st + "");
      writer.close();
    }
    catch (IOException e) {
      System.out.println("BSL Error<1>: I/O error " + e);
      System.exit(1);
    }
  }

  public static void print(Object obj, String f) {
    try {
      FileWriter writer = new FileWriter(f);
      writer.write(obj.toString());
      writer.close();
    }
    catch (IOException e) {
      System.out.println("BSL Error<1>: I/O error " + e);
      System.exit(1);
    }

    /*
      catch (EOFException e) {
     System.out.println("BSL Error<2>: EOF error " + e);
     System.exit(1);
      }
      catch (FileNotFoundException e) {
     System.out.println("BSL Error<3>: " + e);
     System.out.println("File you specified was not found and will be
created!");
     //         System.exit(1);
      }
     */

    } // end of print


  } // end of class
```

```
/*
 * Finder takes care of the find, findall, findprimer functions
 * @author Amna Q.
 * cunix id: aq41
```

```
 */

import java.util.*;

public class Finder{

  // class variables declared here
  Hashtable convert = new Hashtable();

    public static int find(Sequence seq1, Sequence seq2){
      String string1 = seq1.value();
      String string2 = seq2.value();
      return string2.indexOf(string1);
    }

    public static int[] findAll(Sequence seq1, Sequence seq2){
      String string1 = seq1.value();
      String string2 = seq2.value();
      int x = 0;
      int[] temp = new int[string2.length()];
      int i = 0;

      x = string2.indexOf(string1, x);

      temp[i] = x;


      x = string2.indexOf(string1, x+string1.length());

      while(x != -1){
        i++;
        temp[i] = x;

        x = string2.indexOf(string1, x+string1.length());
      }
      int[] array = new int[i+1];

      for(int j = 0; j <= i; j++){
        array[j] = temp[j];
      }
      return array;
    }


    public static Sequence reverse(Sequence seq){
      String string = seq.value();
      String string2 = "";

      for(int i = string.length(); i > 0 ; i--){
        string2 += string.substring(i-1, i);
      }
      Sequence temp = new Sequence();
      temp.setType(seq.getType());
      temp.setSequence(string2);
      return temp;

    }
```

```
    public static int findPrimer(Sequence primer1, Sequence sequence1){
      String primer = primer1.value();
      String sequence = sequence1.value();

      int x = sequence.indexOf(primer);
      boolean done = false;
      int length = 0;

      while(!done){

        length = primer.length();

        if(x >= 0 || length < 4){
          done = true;
        }

        else{

          primer = primer.substring(0,length - 1);
          x = sequence.indexOf(primer);
        }
      }

      return x;
    }

    public Sequence translate(Sequence seq, String type){

      Sequence seq2 = null;
      buildTHash();

      if(type.equals(Sequence.DNA) && (seq.getType()).equals(Sequence.RNA)){
        //convert rna to dna ... replace U with T
        seq2 = dnaToRNA(seq);
      }
      else if(type.equals(Sequence.RNA) &&
(seq.getType()).equals(Sequence.DNA)){
        //convert dna to rna ... replace T with U
        seq2 = rnaToDNA(seq);
      }
      else if(type.equals(Sequence.PROTEIN) &&
(seq.getType()).equals(Sequence.DNA)){
        //convert dna to protein
        // convert dna to rna first, then convert rna to amino acid
        Sequence seq1 = dnaToRNA(seq);
        seq2 = rnaToProtein(seq1);
      }
      else if(type.equals(Sequence.PROTEIN) &&
(seq.getType()).equals(Sequence.RNA)){
        //rna to protein
        seq2 = rnaToProtein(seq);
      }
      return seq2;
    }

    private Sequence rnaToProtein(Sequence seq){
```

```
  Sequence seq1 = new Sequence();
  String seqString = seq.value();
  String temp;
  String newString = "";

  // "best match" algorithm - leftover blocks are trimmed
  // the original sequence but have at least 3 blocks
  int i = 0;
  while (i+2 < seqString.length() ) {
    temp = seqString.substring(i , i + 2);
    temp = temp.toUpperCase();
    newString += (String)convert.get(temp);
    i += 3;
  }
  seq1.setType(Sequence.AMINOACID);
  seq1.setSequence(newString);
  return seq1;
}

private Sequence dnaToRNA(Sequence seq){
  String temp = seq.value();
  temp = temp.replace('T', 'U');
  temp = temp.replace('t','u');

  Sequence seq1 = new Sequence();
  seq1.setType(Sequence.RNA);
  seq1.setSequence(temp);
  return seq1;
}

private Sequence rnaToDNA(Sequence seq){
  String temp = seq.value();
  temp = temp.replace('U', 'T');
  temp = temp.replace('u','t');

  Sequence seq1 = new Sequence();
  seq1.setType(Sequence.DNA);
  seq1.setSequence(temp);
  return seq1;
}

private void buildTHash() {
  // first position is U
  convert.put("UUU", "F");
  convert.put("UUC", "F");
  convert.put("UUA", "L");
  convert.put("UUG", "L");
  convert.put("UCU", "S");
  convert.put("UCC", "S");
  convert.put("UCA", "S");
  convert.put("UCG", "S");
  convert.put("UAU", "Y");
  convert.put("UAC", "Y");
  convert.put("UAA", "STOP");
  convert.put("UAG", "STOP");
  convert.put("UGU", "C");
  convert.put("UGC", "C");
```

```
    convert.put("UGA", "STOP");
    convert.put("UGG", "W");
    // first position is C
    convert.put("CUU", "L");
    convert.put("CUC", "L");
    convert.put("CUA", "L");
    convert.put("CUG", "L");
    convert.put("CCU", "P");
    convert.put("CCC", "P");
    convert.put("CCA", "P");
    convert.put("CCG", "P");
    convert.put("CAU", "H");
    convert.put("CAC", "H");
    convert.put("CAA", "Q");
    convert.put("CAG", "Q");
    convert.put("CGU", "R");
    convert.put("CGC", "R");
    convert.put("CGA", "R");
    convert.put("CGG", "R");
    // first position is A
    convert.put("AUU", "I");
    convert.put("AUC", "I");
    convert.put("AUA", "I");
    convert.put("AUG", "M");
    convert.put("ACU", "T");
    convert.put("ACC", "T");
    convert.put("ACA", "T");
    convert.put("ACG", "T");
    convert.put("AAU", "N");
    convert.put("AAC", "N");
    convert.put("AAA", "K");
    convert.put("AAG", "K");
    convert.put("AGU", "S");
    convert.put("AGC", "S");
    convert.put("AGA", "R");
    convert.put("AGG", "R");
    // first position is G
    convert.put("GUU", "V");
    convert.put("GUC", "V");
    convert.put("GUA", "V");
    convert.put("GUG", "V");
    convert.put("GCU", "A");
    convert.put("GCC", "A");
    convert.put("GCA", "A");
    convert.put("GCG", "A");
    convert.put("GAU", "D");
    convert.put("GAC", "D");
    convert.put("GAA", "E");
    convert.put("GAG", "E");
    convert.put("GGU", "G");
    convert.put("GGC", "G");
    convert.put("GGA", "G");
    convert.put("GGG", "G");

  }

} // end of class
```

```
    /*
        public static void main(String[] args){
      System.out.println("find: ab, bcadabdb "+  find("ab", "bcadabdb"));
      System.out.println("findAll: ab, bacadabdbab ");
      int[] indexes = findAll("ab", "bacadabdbab ");
      for(int i = 0; i < indexes.length; i++){
      System.out.println(indexes[i]);
      }
      System.out.println("reverse: " +
reverse("abcdefghijklmnopqrstuvwxyz"));
      System.out.println("findPrimer: abcdefg, bcabcdabdb "+
findPrimer("abcdefg", "bcabcdabdb"));
      }
      }
 */
```