ScriptEdit Language Reference Manual

Bhavesh Patira
bp2214@columbia.edu

Bethany M. Soule
bms2126@columbia.edu

Deni Pejanovic
dp2232@columbia.edu

MarC Vinyes
mv2258@columbia.edu

May 8, 2007

Chapter 1

Introduction

1.1 Description

ScriptEdit is a language that allows you to automatically generate text from a limited set of instructions. You can write new files with this text, or insert it into an existing file. The instructions may use the text of the edited file, external files, or text generated by other standard input/output based applications as input. ScriptEdit is similar to a macro processor in the sense that it replaces text with other text, but it can also create new files from a single source.

Motivations and Goals: The main goal of this language is to allow the user to edit files and the script operations that are needed to create their content all from within one single source file. Often, editing content text files (HTML, LaTeX, XML, etc) is a process that involves several different steps and programs—like separate bash scripts, a text editor, and other console programs (e.g., using ImageMagick to edit images or using Matlab to create graphs that will be linked). ScriptEdit is a way to put all those different process calls together with the content text file.

Some advantages of scripting operations within the source content file:

Visualization Contents and related scripts are written in the same area so it's easy to check that they are coherent using standard editors.

Inline execution of associated scripts If a long set of programs have to be executed each time, normally a shell script grouping them is created and then executed, but then new users that edit the file should be aware that this file should be executed and will have to figure out if the script has to be run before or after editing some files. Sometimes, the process can be tedious and hard to reproduce like run program1, edit file1, run program2, edit file2. So ScriptEdit eases this task by running the scripts and outputting the text at the same time in the right order. Moreover modifications of the format of the input file are also more easily synchronized with the script.

Templates and macros are easy Some languages like C already have a good preprocessor that allows using templates—#include, #ifdef, etc—but others like HTML or XML don't, and users are asking for them. Macros are normally placed in the editor but each editor uses different implementations of them, so you have to learn multiple ways to do the same.

Other languages have already been built to solve this sort of problem. In the particular case of processing HTML text files: HTP (`http://htp.sourceforge.net/`) and HPP (`http://citeseer.ist.psu.edu/douglis97hpp.html`) are good examples, but they are more specialized than what we have in mind. Perhaps a more comparable existing language is M4 (`www.gnu.org/software/m4`).

Chapter 2

Language Tutorial

2.1 Getting Started

This short tutorial can be used to quickly familiarize yourself with ScriptEdit. The first step is to ensure that you have properly installed the environment needed to run ScriptEdit code. Second, we will look at the most basic examples of ScriptEdit code. Finally, we will walk through a more complex ScriptEdit example.

1. Check Environment: Ensure Java and ANTLR are properly installed

   To begin using ScriptEdit, you will need to do the following:

2. uncompress the source archive into an empty directory
3. ensure that you have a JDK version 1.4 or above
4. verify that ANTLR is available and can be run with the command antlr
5. make sure the current directory and antlr.jar are in the CLASSPATH
6. compile the source code to create the Java code for the translator's scanner and parser from Simplexer.g, and compile the ANTLR-generated files along with the rest of the ScriptEdit Java code into .class files and a ScriptEdit.jar file.

```
$javac d /scriptedit *.java
```

2.2 How to run ScriptEdit

Check that ScriptEdit.class and antlr.jar are in your class path and type the following commands at the command prompt to create output from .se input files:

```
$java Main <SOURCE>
```

<SOURCE>: Is the ScriptEdit source file. ScriptEdit will only allow one input file for a source. If needed, ScriptEdit allows for other files to be embedded within the SOURCE by using the #file statement (see LRM). Source files should be a plain text format that ends with a .se extention. Upon successful compilation, ScriptEdit will create (or replace) an output file with of the same name as the source file without the .se extension. myScriptEditCode.html.se $-$ > myScriptEditCode.html

2.3 Examples of simple ScriptEdit programs

ScriptEdit code can be written using any text editor software. The examples below will help familiarize you with some basic ScriptEdit syntax and allow you to quickly begin creating your own sciptEdit code. The first example is the classic hello world. A variable is created to hold a string "hello world" and then output using the write command.

2.3.1 Example 1: using def command to run HelloWorld.html.se

```
<html>
<!--simple hello world program:-->

#def(greeting){hello world!}
$(greeting)

</html>

$java Min HelloWorld.html.se

OUTPUT FILE: HelloWorld.html

<html>
<!--Simple hello world program: -->
Hello world!
</html>
```

As demonstrated above, ScriptEdit will take the initial ScriptEdit source code the ends in .se, and create a output file the in which the extension is omitted. While the example above shows an HTML file, ScriptEdit can be used for any language. Keep the following points in mind:

1. ScriptEdit instructions begin with a hash (#) or a dollar sign ($)

2. Variable names are enclosed with parentheses and then assigned to whatever is contained within curly brackets immediately following it .
3. All non-script edit commands are ignored by the compiler, and therefore the output is untouched.
4. Comments should be native to the underlying language, as ScriptEdit does not have its own commenting convention
5. Spaces, tabs and all special characters are significant, there should be no spacing between a statement and its corresponding opening parenthesis or curly bracket.
6. Any special characters that you do not wish to be interpreted by as a ScriptEdit instruction should be preceded (escaped) by a backslash character.

The second example below demonstrates a program allowing you to do a loop a set number of time and concatenate a string to form the desired output.

2.3.2 Example 2: My first ScriptEdit page

```
Welcome to my first ScriptEdit page

#def(yearborn){1985}                //define a variable and set value
                                    //to your year of birth
$def(yourage){2006-$(yearborn))     //compute age
You are #write{$(yourage) years old //print age

#def(newage){$(yourage)}
#def(i){1}
#while( $(i) <=3 ){
                        #set(newage){#calc($(yourage)+$(i))}
$(i) year#if($(i)>1){s } from now you will be $(newage) years old
                        #set(i){#calc($(i)+1)}
                  }
OUTPUT:
Welcome to my first ScriptEdit page
You are 21 years old
1 year from now you will be 22 years old
2 years from now you will be 23 years old
3 years from you will be 24 years old
```

Some additional general guidelines to creating ScriptEdit code:

1. In general, script edit uses a combination of parenthesis, (), and curly brackets, {}.

For example:

```
#def(name){myName}
```

The distinction here is that parenthesis usually indicate a string constant whereas curly brackets can be string contants or alternatively more ScriptEdit code nested within.

```
#def(greeting){#if($(language)=en){Hello}#else{Hola}}
```

Here the rvalue assignment itself is made up of ScriptEdit statements.

2.3.3 Example 3: Creating a function

In the next example, we will demonstrate how to create and invoke a function within script edit. Note, the function variables are static in scope.

```
my first function

#def(double,arg1){#calc(2*$(arg1))}
call double on 7:
$(double,7)

OUTPUT:
my first function
call double on 7:
14
```

2.3.4 Example 4:

The last example below will demonstrate how to read a comma delimited input file, containing a list of data, and output an html table the has the data in the first column and an html link to google with the data item imbedded into the URL querystring.

Given myDataFile.data contains data in the following format:

```
Deni Pejanovic, Bhavesh Patira, Marc Vinyes, Bethany Soule
```

The code below, stored in file “myGoogleSearch.html.se”, will create links to the data items so that a user can quickly look those items up in google.

```
<!--
Creating a list of google search links
from data in an external text file...
-->

#def(inputStream){#file{myDataFile.data}}
#def(baseURL){http://www.google.com/search?hl=en&q=}
```

```
<html>
<table border=1>
<tr>
<td> Name of Individual to search </td>
<td> Google Link </td>
</tr>
<tr>
#while($(inputStream)!=)
{
#set(TokenString){#gettoken(inputStream){,}}
<tr>
<td>$(TokenString)</td>
<td>
<a href="$(baseURL)$(TokenString)"> Search Google </a>
</td>
</tr>
}
</table>
```

Hence, when this code is interpreted, a new file named “myGoogleSearch.html” is created with the following content:

```
<!--
Creating a list of google search links
from data in an external text file...
-->

<html>
<table border=1>
<tr>
<td> Name of Individual to search </td>
<td> Google Link </td>
</tr>
<tr>


<tr>
<td>Deni Pejanovic</td>
<td>
<a href="http://www.google.com/search?hl=en&q=Deni Pejanovic"> Search Google </a>
</td>
</tr>


<tr>
<td>Bhavesh Patira</td>
```

```
<td>
<a href="http://www.google.com/search?hl=en&q=Bhavesh Patira"> Search Google </a>
</td>
</tr>


<tr>
<td>MarC Vinyes</td>
<td>
<a href="http://www.google.com/search?hl=en&q=MarC Vinyes"> Search Google </a>
</td>
</tr>


<tr>
<td>Bethany Soule</td>
<td>
<a href="http://www.google.com/search?hl=en&q=Bethany Soule"> Search Google </a>
</td>
</tr>

</table>
```

Chapter 3

Language Reference Manual

3.1 Main lexical conventions

We have the following kinds of tokens: string constants, keywords, characters that separate the arguments of a keyword, identifiers, conditions, mathematical expressions, the declaration operator(`#`) and the content operator (`$`). Their definitions are provided throughout this document.

Whitespace, including tabs and newlines are generally relevant everywhere except when indicated otherwise within this manual. For example: leading and trailing whitespaces are not ignored:

{`My String Variable`} is not equivalent to {`My      String      Variable`} is not equivalent to {`      My String Variable      `}.

3.1.1 String constants

String constants are all consecutive sets of ascii characters including but not limited to dashed, numbers and spaces that aren't keywords, characters that separate the arguments of a statement, identifiers, conditions, mathematical expressions, the declaration operator(`#`) or the content operator (`$`).

The `#, $, (, ), {, }` characters can be escaped with backslash (\) if needed.

3.1.2 Comments

Since ScriptEdit is intended to be embedded within the text of another file, we do not provide a ScriptEdit-specific comment. To comment your code use the native comment style in the string constants.

For example if you are editing an HTML file you would use:

```
<HTML>
        <!-- your comments -->
<HTML>
```

3.1.3 Identifiers (Names)

An identifier is a case sensitive sequence of letters, digits, and underscore (_). The first character must be a letter.

3.1.4 Keywords, declaration and content operators

The following identifiers are keywords that are reserved for specific use as follows: The `#` within a line signifies the start of a ScriptEdit command, and is usually immediately proceeded by a keyword without any space character separation.

```
#file    #def     #if      #write
#else    #while   #exec    #calc
#for     #next    #do      #gettoken
#set
```

The declaration (`#`) and content operators (`$`) may be considered a string constant or part of a variable/function declaration depending on rules that are specified by their semantics.

The lexical conventions of each statement are explained along with its semantics.

3.1.5 Conditions

Conditions are exclusively used as part of the `#if` and `#while` constructs. They evaluate to either true or false.

Allowable binary operators are:

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

Conditions are composed by pairs of statements separated by the binary operator where each statement evaluates to a string value for comparison. Since all variables are identified by their string value, only strings are compared.

Relational operators

Operators >, >=, <, <= are valid only within the conditional portion of an `#if` or a `#while`. Valid things to compare are strings, variables, and numbers.

Operator	Use	Description
`>`	`${v1}>${v2}`	returns true if the integer value of v1 is greater than the integer value of v2
`>=`	`${v1}>=${v2}`	returns true if the integer value of v1 is greater or equal to the integer value of v2
`<`	`${v1}<${v2}`	returns true if the integer value of v1 is less than the integer value of v2
`<=`	`${v1}<=${v2}`	returns true if the integer value of v1 is less or equal than the integer value of v2
`=`	`v1=v2`	returns true if the string v1 is equal to string v2
`!=`	`v1!=v2`	returns true if the string v1 is NOT equal to string v2

Conditional operators

Operator	Use	Description
`&`	`cond1 & cond2`	returns true if cond1 and cond2 are both true
`|`	`cond1 | cond2`	returns true if either cond1 or cond2 are true
`!`	`!cond1`	returns true if condition cond1 is NOT true

Operator precedence

Precedence in evaluation is as follows:

operator	associativity
`* /`	left (highest)
`+ -`	left
`< <= > >=`	non-associative
`= !=`	non-associative
`&`	left
`|`	left
`=`	right (lowest)

3.1.6 Other tokens

Characters that separate the arguments of a keyword are "{", "}", "(", ")", and ",". The { } are generally used to denote statement blocks and "(", ")" generally encloses statement or function arguments, but variable calls are specified using the characters { } for different purposes. More details are given for each particular statement.

Mathematical expressions are specific to statement `#calc` and are explained along with `#calc` statements.

3.1.7 Statements

Statements are executed from beginning of file to end and from left to right. Control-flow structures such as functions are evaluated in place. The basic scripedit statement begins with a keyword, followed sometimes by a variable with parentheses ( ) and an statement block contained within brackets { }.

3.2 Semantics

Scoping behavior varies dependent on context as described herein. However, a general rule is that variables in nested scopes inside the current scope can't be modified from outside their nested scope, but variables from outer scopes can be accessed using the proper statements.

3.2.1 Variable declaration

Variables may be declared and set to a value using one of these two alternatives:

- `#def(variablename){body}`

  A new scope is created and the code contained in "body" is executed and an output string OUT is produced. At the end this new scope is destroyed.

  A variable with"variablename" is created in the current scope and its value is set to the string OUT. If a variable with same name already exists in the current scope, an error is generated.

- `#set(variablename){body}`

  A new scope is created and the code contained in "body" is executed and an output string OUT is produced. At the end this new scope is destroyed.

  We then lookup "variablename" in the current scope, following up parent scopes until it is found, and reset the value of the variable with the new

OUT string. If “variablename” has not been previously declared by `#def` an error is generated.

Syntax notes: Whitespace including tabs and newlines are allowed between (variablename) and {body} in both cases.

3.2.2 Function declaration

Functions are a generalization of a variable and they are declared with a very smilar syntax. They differ in that functions take arguments whereas variables do not. Arguments must be valid identifiers.

Functions may be declared using either `#def` or `#set`, with the same scoping conventions as follows:

- `#def(fn,arg1,arg2,..,argN){body}`

  The function is stored in the current scope’s function table. Nothing is executed within the body of the function until it is called. If a function in the current scope with the same name already exists an error will be generated.

- `#set(fn,arg1,arg2,..,argN){body}`

  We search for a declaration of “fn” in the current scope, following up through enclosing scopes until the most recent previous definition of “fn” is found, and this definition is reset to the value of the current definition. If “fn” has not been previously declared, an error will be generated.

Syntax notes: Whitespace including tabs and newlines are allowed between `(functionname)` and `{body}` in both cases, as well as amongst the arguments to the function.

3.2.3 Variable call

The variable is fetched from the nearest current or outer scope. The constant string value stored in a variable is inserted to the text. Variable call syntax is: `$(varname)`.

3.2.4 Function call

Parameters to a function call can be either variables or string constants. Therefore whitespace matters within the function call, and no spaces are allowed around the function name or variable arguments. Stringconstant arguments are take from one comma to the next (or the close parens).

```
$(fn,$(arg1),...,$(argn))
```

The function is fetched from the nearest current or outer scope. Then a new scope is created, and the function tree is walked, executing the function code, evaluating to a string. Then the recently created scope is destroyed, and the string is returned to the function's caller.

3.2.5 File include

```
#file(filename)
```

The code of filename is inserted at this point at interpretation-time (without creating a new scope) and their instructions are executed as if they were just written in the file where `#file` is invoked.

Note: ScriptEdit only supports forward slashes () in file path names. Also, files that are embedded will not be modified. A copy is inserted into the original ScriptEdit file. For this reason, renaming an included file with a .se extension is not needed.

3.2.6 #if and #else

```
#if(condition)
{body}
#else
{body}
```

Statements within the {} are executed if the if condition evaluates to true. Else is an optional statement that is executed whenever the if statement is not. When else binding is ambiguous, else binds to the nearest elseless if statement.

Syntax notes: Whitespace including tabs and newlines are allowed between `(condition)` and `{body}` in both cases.

3.2.7 #while

```
#while(condition)
{body}
```

This statement will allow you to iterate through statements as long as the condition given evaluates to true. The condition cannot contain additional definitions. Note that since the while uses strong syntax(denoted by the parenthesis) the loop control variable must be definedbefore it is used. The condition is evaluated before each execution of the loop.

The following loop is completed 3 times.

Example:

```
#def(x){0}
#while ($(x)<3)
{...statements...
#set(x){#calc($(x)+1)}}
```

Syntax notes: Whitespace including tabs and newlines is optionally allowed between `(condition)` and `{body}` but ignored by ScriptEdit.

3.2.8 #for, #next, #do

```
#for{body1}
#next{body2}
#next{body3}
...
#next{bodyN}
#do{body}
```

A new scope is created, then `body1` is executed, then `body` is executed, then `body2` is executed, then again `body`, then `body3` then `body`... For each statement block specified with `#for`{$body_i$} and then with `#next`{$body_j$}, those are executed followed by the statement block specified by `#do`{`body`} .

3.2.9 #exec

```
#exec(command){stdinput}
```

The exec command will accept *command* as a string constant and return the output from the OS as a string constant.

This command allows interoperability between the program and the operating system. Only a one-line command argument may be inserted here, but an arbitrary number of exec commands may be computed sequentially as needed.

Syntax notes: Whitespace including tabs and newlines are allowed between `(command)` and `{body}` in both cases.

3.2.10 #calc

`#calc(mathematical expression)`

Attempts to evaluate the arguments of the expression as integers, compute the result and put it back into a string. Note that since calc is a binary operator, parenthesis that traditionally establish precedence for operators is not needed nor accepted. Valid arguments of the mathematical expression are integers, variables, and `#calc` statements. Supports basic arithmetic $(+, -, *, /)$. Division is done by java integer division rules, and only integers are allowed. Variables within the mathematical expression are converted into integer values, computed, and then returned as a string representing the integer result. If this conversion is not possible, `#calc` may return an undesired result.

Example: `#def(salary){1000}#calc($(salary)*2)` will output a string "2000".

Syntax notes: Mathematical expressions consist of exactly two string variables or constant strings separated by one of the following binary operators $(+, -, *, /)$.

3.2.11 #gettoken

`#gettoken(variablename)(DELIM)`

This statement is used to tokenize a variable. When invoked it will return the string up to but not including the delimiter, removing the delimiter from the string. It does this destructively. `(DELIM)` may be left off and `#gettoken` defaults to delimiting by whitespace.

3.2.12 #write

`#write{filename}{body}`

Utility for outputting result of ScriptEdit executions to a file. If `filename` already existed, its content is overwritten.

Chapter 4

Project Plan

4.1 Processes used for planning, specifications, development, testing

Collaboration between team members was done through a series of face-to-face meeting used to hash out the initial roles, responsibilities and timelines. During these meeting we agreed as to what needed to be completed by each member and the date it was expected. To augment our meetings, minutes were used and distributed as a follow-up to assure there was no miscommunication. In addition, a spreadsheet of tasks was used as a project plan that identified resources, milestones and dates to assure proper completion of the project. Intermittingly, emails among members where used to clarify any nuances or questions relating to the applications, however all team members where cc'ed on all communication to assure there was full disclosure of all issues that could potentially have an effect on others.

4.2 Programming style guide

Based on the initial proposal of the language, the style of the language seemed quite clear to all team members, but we nonetheless created and distributed the following style guide.

Collaborative coding All code modifications must be properly commented, it should include your initials, the date and a brief description of the code implemented. Parameters (both in and out), return values, and exceptions must be properly documented within the code. Avoid unusual hacks that make it difficult to follow. If any complex code is used that is difficult to follow, you must include extra comments explaining the issue at hand. If you comment out a whole section of code, indicate the reason.

Follow best practices Use logical indentation (with spaces, not hard tabs) and always use variable names that are unambiguous. Code should flow logically. In general, if something can be done more efficiently but is more difficult for others to understand select the version which is easier for others to follow.

Experimental code If you are creating any experimental code, you need to append an underscore followed by your name. For example "treewalker_bethany" will allow other team members to know that you are testing some functionality that is not ready to be inserted into the main project.

Uploading code Code that is ready to be inserted into the main project can be done so immediately as long as you send an email to the group ensuring that they synchronize their local project with that of the SVN repository. It is a good practice to sync regularly and especially before uploading or modifying code. The email also serves as a reminder for other members to test the code.

Testing code Your code should be unit tested prior to inserting into the repository. Once in the repository, send an email to Deni to begin testing the additional code.

Rollback If any code needs to be rolled back to a previous version, we need to bring this up in a regular group meeting as the consequences could have an effect on other code.

ScriptEdit Style Here are some key style issues to adhere to so that our code is consistant.

1. ScriptEdit is case sensitive; follow convention of using #lowercase.
2. ScriptEdit is sensitive to added whitespace. Watch for leading/trailing whitespace that could break code. ScriptEdit is also sensitive to tabs.
3. In general, ScriptEdit uses a combination of parentheses, (), and curly brackets, {}. The distinction here is that parentheses usually indicate a string constant whereas curly brackets can be a string constant but also can allow additional ScriptEdit code to be nested within.

For Example:

```
#def(name){myName}
```

The example shows that the def statement uses (name) can only take a string constant, whereas the {myName} portion can alternatively be imbedded with additional ScriptEdit statements.

4.3 Project timeline

The following timeline represents major milestones along primary responsibilities:

Task	Description	Assignment	Start	End	Status
Language proposals	Each member creates 2 page proposal for consideration by group	All Team	1/22	1/29	100%
Determine language	Meet with professor and settle on language	All Team	1/29	1/31	100%
Environment	Install ANTL, review tutorial, set up SVN; Eclipse, plugins functioning	All Team	1/22	1/31	100%
Proposal	Document language purpose and example of usage	Marc, Deni	1/31	2/7	100%
Scanner	Create the initial version of the scanner	Marc, Bhavesh	2/7	2/15	100%
LRM	Create and document language specification	Marc, Deni, Bethany, Bhavesh	2/7	3/5	100%
Tree generation	Scanner tokens properly create tree structure		2/15	2/21	100%
Tree walker	Tree walker that correctly iterates through tree	Bethany	2/21	2/28	100%
Regression testing	Create sample test programs, compare to desired output	Deni	2/28	3/5	100%

Table 4.1: Project timeline

4.4 Roles and responsibilities

All team members wanted to play a role in coding the language. The team divided work primary and secondary responsibilities as follows:

Bethany Primary role: developing the language treewalker functionality. Secondary role: project lead. Assuring we met regularly, adhered to the proposed timeline. Assisting with documentation.

Marc Architect and primary decision maker on language behavior. Secondary role developing key functions and parser. Code review of others work

Bhavesh Developed much of the core functional behavior of the language. Secondary role: Code review.

Deni Documentation for LRM, and final report. Secondary role: testing application including regression testing.

While the roles show primary responsibilities, it was agreed that each would participate in all responsibilities to a lesser degree (such as testing and coding) so that there was overlap in workload.

4.5 Environment

Java SDK 1.5.0 using Eclipse as our foundation tool, with the ANTLR (v 2.4) plug-in. Collaboration repository is using an SVN plug-in for Eclipse. The centralized SVN repository is used for version control and is password protected. Two team members used windows, one used Linux, and one used a Mac—no OS related issues arose during the project.

Version control was handled by SVN, but each member also communicated via email to inform others if any major changes were made so that all team members could anticipate any issues.

4.6 Project log

4.6.1 Settling on an Idea

1/22 During our first few meetings, each participant was to prepare and present an idea for a language they would like to build along with a preliminary one page whitepaper and one page example of their language.

1/23-1/31 Each of us presented our ideas. Among them was a gaming language, a graph algorithm prototyping language, and ScriptEdit. We debated for quite some time, and eventually met with professor Edwards after class to determine which idea was the most viable.

1/31/07 After further discussion, we were concerned that the gaming language might be a poor choice as it had the potential to end up as a large dictionary file, with minimal value as a language. The graph algorithm language had the pitfall of being overly complex. We then had consensus to move forward with the ScriptEdit, a text editing language that could be used for a variety of applications including HTML. With that, it was clear that Marc would play a lead architectural role.

4.6.2 Creating Roles and responsibilities

2/1-2/5 From the outset, the role of chief architect would go to the member who proposed that particular language. All members wanted to play a role in the design, but clearly having one person make a final determination settled a lot of discussions that could have been done in one of many ways. The other roles fell into place after a few discussions. Section 4.4 details the final primary roles and responsibilities of the group, although there was considerable overlap in several areas as noted.

4.6.3 Determine tools and setup

2/1-2/5 There was a consensus early on that we to use Eclipse as our foundation tool. The ANTLR and SVN plug-ins were installed by all members without any major issues. Bethany set up the SVN repository early on and we tested it with our initial documentation. We found working with SVN to be a mostly hassle-free experience. Among the different languages, Java seemed to be the common foundation for the team members.

4.6.4 Conceptualization, General Language behavior

2/1-2/20 A series of discussions followed to determine some basic features and characteristics of the language. Chiefly, we were concerned with scoping rules and their impact. Also, we needed to define the most basic data types and how and why they were used. Much of the decisions leaned towards simplicity so that we could get them to perform properly with the notion that if time permitted we would enhance the language. We unanimously elected to use static scoping and keep almost all data types as sting constants.

4.6.5 Language Reference Manual

2/10-3/5 Based on our whitepaper/proposal, the LRM was to extend and define the language. Unfortunately, without having the language built, we realized that some of the language specifics would be changed as it was being coded.

The initial version of the language reference manual had input from all team members. In order to keep the manual accurate and presentable we elected to use LaTeX. We found the application to be very functional and helpful during the editing process. While not entirely complete by the initial due date, the creation helped us define much of the language as well as identify some of the potential problems we would run into. Several versions of the LRM were circulated among team members for input and editing as it was being finalized through several rounds of revisions.

4.6.6 Parser coding

2/15-2/21 Coding began early on with the development of the parser. We knew this piece had to be up and running right away or we would risk falling behind schedule on other critical path items that depended on this piece being completed. The parser tokenized the input file and was tested using a java program which put the tokens into a tree for visual inspection. This process was tested and seemed to work very well from the outset. Marc spearheaded the effort of getting the initial parser up and running.

4.6.7 Testing

2/21- 4/5 Testing was done throughout the entire project timeline. Aside from each team member unit testing their individual code, the complete code was repeatedly tested as pieces were being rolled up into the final project. Several scripts were created to do a more robust test to assure all components that previously worked continued to perform as desired. Deni was primarily responsible for coordinating test efforts.

4.6.8 Treewalker coding

2/15- 2/21 The treewalker which creates the hierarchical structure of the tokens and operators was a more involved task. Bethany spearheaded the effort of getting the initial treealker working properly.

4.6.9 Language coding

2/21 -2/28 Creating all of the ANTLR code that actually defines the language was probably the most important and involved tasks of the project. Bhavesh spearheaded much of this effort, assigning tasks around to all of us, and keeping us in contact, making sure things were happening.

Chapter 5

Architectural Design

The main components and processes of Scirptedit are described herein.

1. Upon envoking the ScriptEdit Java executable, the main Java reads the input file and controls the compilation process.
2. The ScriptEdit source file is first opened by the pre-processor which sole function it to pass through the source file and perform any in-place includes (all additional files must be assessable in an available directory in order to be absorbed into the source).
3. The compiler then reopens the aggregated file for processed by the lexer (contained in the simpLexer.g file)
4. The lexer tokenizes the input stream by scanning the input sequentially and determining the separation points between tokens.
5. The output of the lexer is then pushed through the parser (which is also contained in simpParser class within the simpLexer.g file). ScriptEdit syntax is defined using ANTLR to define how to build the AST tree.
6. Once the tree is complete, the treewalker (SimpTreeWalker.g) is engaged to begin traversing the tree and interpret and replace the output as needed.
7. If the source file is successfully compiled by ScriptEdit, a secondary output file is created. The outfile naming is dependent on the the name of the input file given. We recommend convention is to use a *.se extension to designate a ScriptEdit sourcefile. If the *.se extension is omitted, it will be added

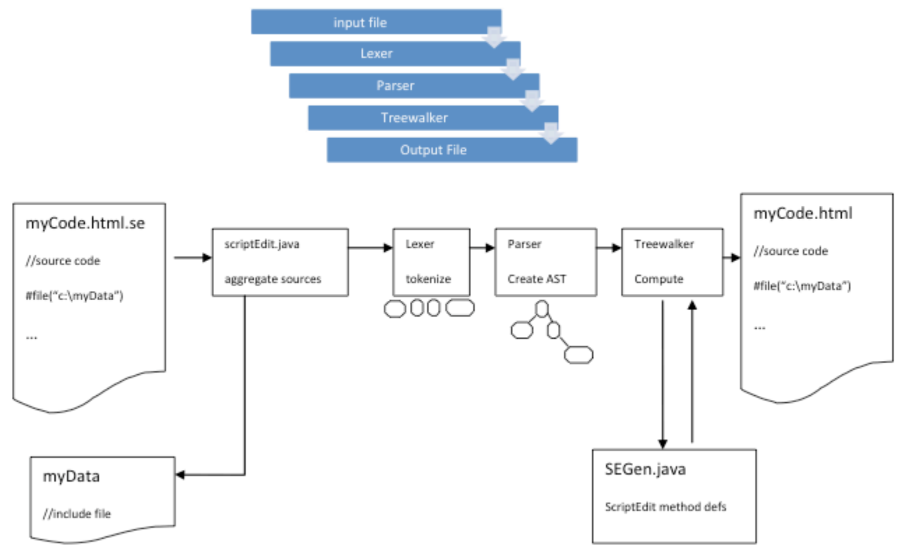

Figure 5.1: A high level visual representation of the process stages

Source File Name	Compiler action	Output File Name	Notes
Mycode.html.se (recommended use)	Create a file named MyCode.html (removing .se extension)	MyCode.html	New output file is created named MyCode.html
MyCode.html (missing .se extension)	Copy of source file into a file called MyCode.html.se which will be your new source code	MyCode.html	MyCode.html will no longer contain your source code, your source will be found in myCode.html.se
MyCode (no extension)	Copy of source created named MyCode.se. MyCode will be overwritten with output of compiler	MyCode	Mycode will no longer contain your source code
MyCode.txt.html.se (more than 2 extensions)	Output file will be file name sans .se extension.	MyCode.txt.html	Output file will be name of complete file without .se extension

Table 5.1: stages of the interpreter

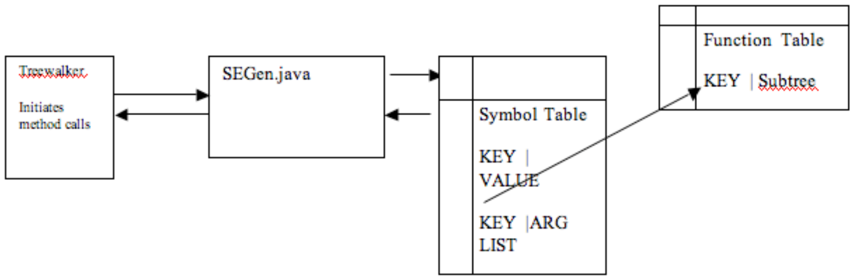

Figure 5.2: SEGen contains the heart of ScriptEdit language methods called by the treewalker. In response, SEGen may go back to the tree to read or write information. SEGen has the following methods defined uses a symbol table which stores Key-Value pairs, and when in some functions are defined a the value becomes a list or incoming arguments to the function. In the later case, a new entry is created in the function table which stores Key-Subtree information. ScriptEdit uses a k value of 10.

Chapter 6

Test Plan

In order to create a functioning language within the timeframe of this project, we adopted a philosophy to start with the simplest working framework, and then continue to add functionality to the core system and test it as it was being added. Each team member unit tested their code prior to incorporating it into the main project. At such time, and email was sent to the remainder of the group to review the code and assure any detectable errors where effectively handled. As the functionality was enhanced, more and more coded needed to be tested in an automated fashion. A test directory was created which help the test scripts or each command. These scripts were then called by a centralized file that tested all the commands in one comprehensive pass. In addition, some complex scripts that ensured prior working code continued to work after the enhancements were incorporated. By incorporating regression testing we were assured that no major failures were encountered that would stall further development.

6.0.10 Test Code

The following test cases represent a portion of the code routinely run against the ScriptEdit compiler that assures the code continues to behave appropriately. More test scripts can be found in AppendixA.2.

Test Case 1: This code tests several functionality at once.

```
;ksjndvljkf
#def(check){5}
$(check)
**********
define double
#def(double,arg1){#calc(2*$(arg1))}
```

```
call double on 7:
$(double,7)
value of check is:
$(check)
call double on check:
$(double,$(check))
***********
***********
define triple
#def( triple  ,
arg1  ){boo}
tryout triple:
$(triple, 3)
***********
#set(check){6}
$(check)
#def(check1){String}
#calc(5+$(check))
#calc(6+#calc(6+5))
#if(#calc(2+1)=#calc(1+2)){
nothing
#calc(10)
$(check1)
#set(check1){NewString}
$(check1)}
#else{
#def(check){5}}
$(check1)
#while(gibber=gibber){
whilenothing:D}
#for{
#def(check1){gibberish}
#calc(4)}
#next{
gibberish}
#next{
gibberish 2}
#do{
enjoy}
#write{code/examples/example.txt}{
newfile
#calc(5)}
```

Test Case 2 The following code shows an example of the testing done for scoping.

```
testing scoping...
#def(m){0}
#def(var1){global}
#while($(m)<2)
{
#set(m){#calc($(m)+1)}
#def(var1){local}
$(var1)
}
$(var1)

next text...
Should output 1, 2, 3, 2, 1
#def(n){0}
#def(var2){1}
#while($(n)<1)
{
#set(n){#calc($(n)+1)}
#def(var2){2}
$(var2)
if($(var2)=2){
#def(var2){3}
$(var2)
}
$(var2)
}
$(var2)
```

Test Case 3 The following code tests one comparator of a while loop (in this case the less than ¡ operator). It gives an sense of the testing needed for as many possible scenarios of comparing two values, strings, integers, string with integers etc testing while...

```
#def(i){1}
#while($(i)<3)
{
#set(i){#calc($(i)+1)}
loop $(i)
}

#def(k){#calc(3)}
#while($(k)< #calc(6))
{
#set(k){#calc($(k)+1)}
loop $(k)
}

#def(l){9}
#while($(l)< #calc(011))
{
#set(l){#calc($(l)+1)}
loop $(l)
}
```

6.1 Test Suites

Testing was achieved by running a series of sample test files in series. Additionally, a java tree is displayed during execution which displays the AST within a window to allow for an easier visual inspection of the tree structure.

6.2 Test Automation

Testing was achieved by running a series of sample input files. Additionally, to help automate regression testing during development , four sample scripts were run that invoked all of the completed functionality in a less rigorous fashion.

Once the sample scripts ran correctly, a more rigorous test was run that tested the complete suite of functionality.

At the end, we implemented in script edit its own testing suite. The code is the following and will give another example of its possible applications:

```
*** SCRIPT-EDIT AUTO-TEST SUITE ***
#set(examplesdir){./code/examples/simpletests}
#def(tests){#exec{sh -c "ls -x $(examplesdir)/*.se"}}

Files tested:
$(tests)

#while($(tests)!=)
{#set(test){#gettoken(tests){
}}
--------------------------------------------------------------
TEST $(test)
--------------------------------------------------------------
OUTPUT:
#set(result){#file{$(test)}}
#set(trueout){#file{$(test).true}}
#if($(result)=$(trueout)){SUCCESS}#else{FAILURE}
}
```

Chapter 7

Lessons Learned

The team members agree that there were many tasks that were executed well, and certain other tasks which where we could have done better.

Scoping From the outset, we were concerned that scope creep might engulf us into a huge project which we could not finish during the project timeline. We consciously weighed the amount of effort needed and the benefits to determine that much of the functionality we considered to be cool simply wasnt possible within the timeframe. We therefore, wanted to ensure that all the base code worked flawlessly, and that further enhancements could be added piecemeal if time permitted. All team members were satisfied with this approach, and some of the added bells-and-whistles such as the pre-processor, and file handling, still made it into the final project. We were satisfied with this approach.

Collaboration Some team members had difficulty with collaborative check-in and check-out procedures that caused some distress. Luckily, the nuances of working with SVN were uncovered earlier on in our development that allowed us to all work effectively on the same code. Of note, we found that the slightest modification to a file (e.g. adding a space), would prevent the "update" command from performing properly. Periodically, "sync" needed to be used to do a file by file compare to assure your project had the latest code which the team shared. We held many sessions, where we worked in the same room, which immensely helped by answering questions immediately as they arose, instead of struggling with a problem, and awaiting replies to emails.

7.1 Individual Team Member Comments

7.1.1 Deni:

While we seemed to manage the timeline fairly effectively, I would certainly recommend that future PLT students start early. If we did not have a small working code from the beginning, we would have been fighting an uphill battle to get more code in and tested by the deadline. I appreciate the fact that the LRM was due early on (before the code was really fully developed) because it allowed us uncover many of the major issues we would be encountering in the coming months this really helped us get a handle on how much work was involved. Regarding testing, I found that we definitely could have benefited from a automated testing tool. While our scripts were very effective at first, as the code grew, it seemed we had reached the limits of their effectiveness, and could have gained some valuable time had we employed attesting tool from the beginning.

7.1.2 Marc:

I think we should have spent a little bit more discussing the structure at the beginning, identifying the most general parts of our project and agreeing on some basic programming style so that it is easier to understand each other's code. I also learned the importance of maintaining a set of working examples where we add from the beginning and we make sure that after each new implementation, the previous examples still work.

7.1.3 Bhavesh:

- I have learned a lot about how to write Grammar, especially how to split up statements and then walk them recursively
- ANTLR does not do anything more than creating smart Case statements
- Its actually quite easy to implement recursive logic in ANTLR
- The ANTLR Parser does a good job of creating a tree out of tokens.
- There are more than one ways of doing the same thing, and each has its own benefits
- It is a challenge to have team-mates commit time (everyone puts in the maximum effort, but not time)
- This project seems to focus less on good program design and more on good program implementation

7.1.4 Bethany:

- Better encapsulation of different sections of the project would have made it easier to work as a team.
- Thank goodness for deadlines. Seems like this is the sort of project that will always take exactly as long as you have to work on it.
- I think we spent far too much time early on discussing the structure and the big picture of what we were gonna do, and not enough time doing. We could have reduced our carbon emissions to kyoto standards by picking a bare-bones subset of the language and diving straight in to implement it.
- The tools you have to work with have such an immense effect on the work you do. (I loathe eclipse. Learning Antlr was like infering all of voodoo magic through inference from watching a half-deaf, half-blind, wizened, creole-speaking woman perform one ritual).

Appendix A

Code

A.1 ScriptEdit

A.1.1 Main.java

```
package code.test;

/*
 * Simple front-end for an ANTLR lexer/parser that dumps the AST
 * textually and displays it graphically.  Good for debugging AST
 * construction rules.
 *
 * Behrooz Badii, Miguel Maldonado, and Stephen A. Edwards
 */

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


public class Main {
public static String fileout;

  public static void main(String args[]) {

if (args[0]==null){
System.out.println("ERROR: Unspecified input file.");
```

```
return;
}

String filein=args[0];
fileout=args[0];
String lastext="";
int i;
for(i=filein.length()-1;(i>=0) && (filein.charAt(i)!='/') && (filein.charAt(i)!='.');i

BufferedReader source=null;
if (!lastext.equals("se")){
try{
//Create a backup of the input file at inputfile.se
source = new BufferedReader(new FileReader( filein ));
}
catch ( IOException iox ) {
      System.out.println("ERROR: I can't find this input file." );
      return;
    }
filein=args[0]+".se";
try{
File fW=new File(filein);
if (fW.exists()){
System.out.println("ERROR: There's already an script-edit file ("+args[0]
+".se"+") associated to the input file ("+args[0]+")\nSet "+args[0]+".se"
+" as input file or rename it.");
return;
}

PrintWriter dest = new PrintWriter( new BufferedWriter(new FileWriter(filein )) );

String line = source.readLine();
String linetmp="";

while ( line != null )
{
  linetmp=line;
  line = source.readLine();
  if (line!=null)
  dest.println(linetmp);
  else dest.print(linetmp);
}
dest.close();
source.close();
    }
    catch ( IOException iox )
    {
      System.out.println("ERROR writing file. I can’t make a backup of the input file." );
      return;
    }

}
else{
fileout=fileout.substring(0, i);
if (fileout.equals("")){
System.out.println("ERROR: .se is not a valid input file." );
return;
}
}

DataInputStream input;
try {
  //DataInputStream input = new  DataInputStream(new FileInputStream("code/examples/example0.txt'
      input = new  DataInputStream(new FileInputStream(filein));
}
      catch ( FileNotFoundException iox ){
       System.out.println("ERROR: I can’t find this input file.");
       return;
      }

      if (input == null){
       System.out.println("ERROR: I can’t read input file.");
       return;
      }

      try {
      // Create the lexer and parser and feed them the input
      SimpLexer lexer = new SimpLexer(input);

      /*
      System.out.println("Elucidate me with a list of the tokens:");
      Token str;
      str=lexer.nextToken();
      while(str.getText()!=null) {System.out.println(str); str=lexer.nextToken();}
      */

      if(lexer == null){
       System.out.println("Lexer null");
      }

      SimpParser parser = new SimpParser(lexer);
      parser.file(); // "body" is the main rule in the parser
      if(parser == null){
       System.out.println("Parser null");
      }

      // Get the AST from the parser
      SimpTreeWalker walker = new SimpTreeWalker();

      CommonAST tree = (CommonAST)parser.getAST();

      // Print the AST in a human-readable format
      //System.out.println("Display Parse Tree as a String List");
      //System.out.println(tree.toStringList());

      /*System.out.println("Display Tree from the Tree Walker");
      String test[] = walker.getTokenNames();
      for(int i=0; i < test.length; i++){
        System.out.print("  token "+i+": ");
        System.out.println(test[i]);
      }*/

      // Open a window in which the AST is displayed graphically
      //System.out.println("Display Parse Tree graphically");
      //ASTFrame frame = new ASTFrame("AST from the Simp parser", tree);
      //frame.setVisible(true);

      System.out.println("walk the tree?");
      walker.fileroot(tree);
      //  System.out.println(tree.statement1(parseTree).toString());

    } catch(Exception e) { e.printStackTrace(); }
  }

}

\end{vebatim}


\subsection{Simplexer.g}

\begin{verbatim}

header {
package code.test;
}
```

```
class SimpLexer extends Lexer;

options {
k = 10;
testLiterals = false;
exportVocab = simp;
charVocabulary = '\3'..'\377';
}


DEF: {LA(2)=='d' }? {LA(3)=='e' }? {LA(4)=='f' }? {LA(5)=='(' }? "#def(";
SET: {LA(2)=='s' }? {LA(3)=='e' }? {LA(4)=='t' }? {LA(5)=='(' }? "#set(";
SHARP: "#";
DOLLAR: '$';
VAR: {LA(2)=='(' }? "$(";
CALC: {LA(2)=='c' }? {LA(3)=='a' }? {LA(4)=='l' }? {LA(5)=='c' }? {LA(6)=='(' }? "#calc(";
FILE: {LA(2)=='f' }? {LA(3)=='i' }? {LA(4)=='l' }? {LA(5)=='e' }? "#file";
FOR: {LA(2)=='f' }? {LA(3)=='o' }? {LA(4)=='r' }? "#for";
NEXT: {LA(2)=='n' }? {LA(3)=='e' }? {LA(4)=='x' }? {LA(5)=='t' }? {LA(6)=='{' }? "#next{";
DO: {LA(2)=='d' }? {LA(3)=='o' }? {LA(4)=='{' }? "#do{";
EXEC: {LA(2)=='e' }? {LA(3)=='x' }? {LA(4)=='e' }? {LA(5)=='c' }? "#exec";
GETTOKEN: {LA(2)=='g' }? {LA(3)=='e' }? {LA(4)=='t' }? {LA(5)=='t' }? {LA(6)=='o' }?
          {LA(7)=='k' }? {LA(8)=='e' }? {LA(9)=='n' }? {LA(10)=='(' }? "#gettoken(";
WRITE: {LA(2)=='w' }? {LA(3)=='r' }? {LA(4)=='i' }? {LA(5)=='t' }? {LA(6)=='e' }? "#write";
IF:{LA(2)=='i' }? {LA(3)=='f' }? {LA(4)=='(' }? "#if(";
WHILE: {LA(2)=='w' }? {LA(3)=='h' }? {LA(4)=='i' }? {LA(5)=='l' }? {LA(6)=='e' }? {LA(7)=='(' }?
       "#while(";
ELSE: {LA(2)=='e' }? {LA(3)=='l' }? {LA(4)=='s' }? {LA(5)=='e' }? "#else";
//CONTINUE: {LA(2)=='c' }? {LA(3)=='o' }? {LA(4)=='n' }? {LA(5)=='t' }? {LA(6)=='i' }?
            {LA(7)=='n' }? {LA(8)=='u' }? {LA(9)=='e' }? "#continue";
//BREAK: {LA(2)=='b' }? {LA(3)=='r' }? {LA(4)=='e' }? {LA(5)=='a' }? {LA(6)=='k' }? "#break";

PLUS  : '+' ;
MINUS : '-' ;
MUL   : '*' ;
DIV   : '/' ;
EQUAL : '=';
GT : '>';
LT : '<';
GE : ">=";
LE : "<=";
NE : "!=";
COMMA: ',';
LPAREN : '(';
RPAREN : ')';
```

```
LCURLY : '{';
RCURLY : '}';
NEWLINE : ( '\n' | '\r'
          | ('\r' '\n') => '\r' '\n')
   {newline();};
//        {newline(); $setType(Token.SKIP);};

WS : ( ' ' | '\t' )+
//{System.out.println("token space");}
;

protected DIGIT : '0'..'9' ;
protected LETTER : ('a'..'z' | 'A'..'Z');

ID options {testLiterals = true;}
: first:LETTER (LETTER | DIGIT | '_')*
//{ System.out.println("token id!"+first.getText());}
;

// number
NUMBER : (PLUS|MINUS)?(DIGIT)+;

ESCAPECHAR: '\\';

MEANINGLESS : ~('+'|'-'|'*'|'/'|'='|'<'|'>'|','|'('|')'|'{'|'}'|'\n'
|'\t'|'\r'|' '|'0'..'9'|'a'..'z' | 'A'..'Z'|'#'|'$'|'\\')
//{ System.out.println("meaningless!");}
;

class SimpParser extends Parser;
options {
k=2;
buildAST=true;
exportVocab=simp;
}

tokens{
FILEROOT;
BTW_IF_AND_ELSE;
STRINGCONST;
STRINGCONSTFILE;
STRINGCONSTCOND;
ARG;
ARGS;
STRINGCONSTARG;
FORBODY;
NEXTBODY;
NEXTBODS;
WRITEBODY;
GETTOKENBODY;
TOWRITE;
FILEBODY;
WHILEBODY;
STDIN;
BODYFILE;
BODY;
DOLLV;
EXECBODY;
DEFBODY;
DARGS;
DARG;
}

invisible: (NEWLINE!|WS!)*;

invisiblecandidate: (NEWLINE|WS)*
{ #invisiblecandidate = #([BTW_IF_AND_ELSE, "BTW_IF_AND_ELSE"],invisiblecandidate);};
//little trick to manage a particular case:
//if after the "if" statement, there's an "else" statement, blank spaces and
//line breaks are ignored, otherwise they should be handled appropiately

//dollv : (dollarvar|argvar)
//{ #dollv = #([DOLLV, "DOLLV"],dollv);}

private escape: ESCAPECHAR! ( ESCAPECHAR | SHARP | RPAREN
  | DOLLAR  | PLUS  | MINUS
  | MUL  | DIV  | EQUAL
  | GT  | LT  | GE
  | LE  | NE  | LCURLY | RCURLY
  | LPAREN  | COMMA  | DEF | SET | VAR | CALC | FILE | FOR
  | NEXT | DO | EXEC | GETTOKEN | WRITE | IF | WHILE | ELSE | CONTINUE | BREAK)
//   {System.out.println("Found a escaped character!");}
  ;

private backslash: ESCAPECHAR;

dollarvar: VAR^ varpart;
varpart: ID argscall RPAREN! { #varpart = #([DOLLV, "DOLLV"],varpart);};

argscall: ( COMMA! argvar )*
{ #argscall = #([ARGS, "ARGS"],argscall);};
```

```
argvar: (stringconstarg | (VAR^ varpart))
{ #argvar = #([ARG, "ARG"],argvar);};

argsdef: (COMMA! argsdefpart)*
{ #argsdef = #([DARGS, "DARGS"],argsdef);};
argsdefpart: invisible ID invisible
{ #argsdefpart = #([DARG, "DARG"],argsdefpart);};


//mathexpr
mathexpr: mathexpr2 ((PLUS^|MINUS^) mathexpr2)*;
mathexpr2: mathatom ((MUL^ | DIV^) mathatom)*;
mathatom : invisible (dollarvar | NUMBER^ | calcx) invisible;

cond
: condexpr ( (EQUAL^|NE^|GT^|LT^|GE^|LE^) condexpr )?;

condexpr
: dollarvar
| mathexpr
| stringconstcond
;

file
: bodyfile EOF!
{ #file = #([FILEROOT, "FILEROOT"],file); }
;

//string constants (text without script-edit statements)
stringconst
//all except a RCURLY
: (ID  | NUMBER  | WS
  | NEWLINE | SHARP  | RPAREN
  | DOLLAR  | PLUS  | MINUS
  | MUL  | DIV  | EQUAL
  | GT  | LT  | GE
  | LE  | NE  | LCURLY
  | LPAREN  | MEANINGLESS | COMMA | escape |backslash)+
{ #stringconst = #([STRINGCONST, "STRINGCONST"],stringconst);
//System.out.println("one");
};

stringconstcurly
//in the root, the RCURLY doesn't matter at all
: (id:ID //{System.out.println(id.getText());}
  | NUMBER
  | ws: WS //{System.out.println("Space"+ws.getText());}
  | NEWLINE
  | sh:SHARP ID//{System.out.println(sh.getText());}
  | RPAREN
  | DOLLAR  | PLUS  | MINUS
  | MUL  | DIV  | EQUAL
  | GT  | LT  | GE
  | LE  | NE  | LCURLY | RCURLY
  | LPAREN  | MEANINGLESS | COMMA
  | escape |backslash
  )+
{ #stringconstcurly = #([STRINGCONSTFILE, "STRINGCONSTFILE"],stringconstcurly);
};


stringconstcond
//all exept a RPAREN or EQUAL^|NE^|GT^|LT^|GE^|LE^
: (id:ID //{System.out.println(id.getText());}
  | NUMBER
  | ws: WS //{System.out.println("Space"+ws.getText());}
  | NEWLINE
  | sh:SHARP ID//{System.out.println(sh.getText());}
  | DOLLAR  | PLUS  | MINUS
  | MUL  | DIV
  | LCURLY  | RCURLY
  | LPAREN  | MEANINGLESS | COMMA
  | escape |backslash
  )*
//: (ID  | NUMBER  | WS
 // | NEWLINE | SHARP  | COMMA
 // | DOLLAR  | PLUS  | MINUS
 // | MUL  | DIV  | LCURLY
 // | LPAREN  | MEANINGLESS | RCURLY | escape )*
{ #stringconstcond = #([STRINGCONSTCOND, "STRINGCONSTCOND"],stringconstcond); }
;

stringconstarg
//all exept a COMMA or a RPAREN
: (ID  | NUMBER  | WS
  | NEWLINE | SHARP
  | DOLLAR  | PLUS  | MINUS
  | MUL  | DIV  | EQUAL
  | GT  | LT  | GE
  | LE  | NE  | LCURLY
  | LPAREN  | MEANINGLESS | RCURLY | escape |backslash)*
{ #stringconstarg = #([STRINGCONSTARG, "STRINGCONSTARG"],stringconstarg); }
```

```
;

// body
bodyfile: (stringconstcurly | writex | filex | execx | gettoken
           | calcx | defx | setx | forx | whilex | ifx | dollarvar)*;
body: (stringconst | writex | filex | execx | gettoken
       | calcx | defx | setx | forx | whilex | ifx | dollarvar )* ;
funcbody : (stringconst | writex | filex |  execx | gettoken
            | calcx | defx | setx | forx | whilex | ifx | dollarvar )* ;

calcx: CALC^ mathexpr RPAREN!;
defx : DEF^ invisible ID invisible argsdef invisible RPAREN! invisible defbody;
setx : SET^ invisible ID invisible argsdef invisible RPAREN! invisible defbody;
defbody : LCURLY! body RCURLY! { #defbody = #([DEFBODY, "DEFBODY"],defbody);};

forx : FOR^ forbody invisible! nextbods dox;
nextbods: (nextx)+ { #nextbods = #([NEXTBODS, "NEXTBODS"],nextbods);};
forbody : LCURLY! body RCURLY! { #forbody = #([FORBODY, "FORBODY"],forbody); };
nextx: NEXT! body RCURLY! invisible{ #nextx = #([NEXTBODY, "NEXTBODY"],nextx); };
dox : DO^ body RCURLY!;
whilex : WHILE^ cond RPAREN! invisible whilebody;
whilebody : LCURLY^ body RCURLY!;
ifx : IF^ cond RPAREN! invisible ifbody invisiblecandidate (elsex)?;
ifbody : LCURLY^ body RCURLY! ;
elsex : ELSE! LCURLY^ body RCURLY! ;
gettoken: GETTOKEN^ invisible ID invisible RPAREN! (gettokenbody)?;
gettokenbody: LCURLY! body RCURLY! { #gettokenbody = #([GETTOKENBODY, "GETTOKENBODY"],
//continuex: CONTINUE^;
//breakx: BREAK^;
writex: WRITE^ towrite  writebody;
towrite : LCURLY! body RCURLY! invisible{ #towrite = #([TOWRITE, "TOWRITE"],towrite);
writebody: LCURLY! body RCURLY! { #writebody = #([WRITEBODY, "WRITEBODY"],writebody);
filex: FILE^ filebody;
filebody: LCURLY! body RCURLY! { #filebody = #([FILEBODY, "FILEBODY"],filebody); };
//varfilex: VARFILE^ (LPAREN! ID argscall RPAREN!)? LCURLY! body RCURLY!;

stdin: LCURLY! body RCURLY!
{ #stdin = #([STDIN, "STDIN"],stdin); };
execx: EXEC^ execbody;
execbody: LCURLY! body RCURLY! { #execbody = #([EXECBODY, "EXECBODY"],execbody); };
```

A.1.2 SimpTreeWalker.g

```
header {
package code.test;
```

```
import java.util.*;
import antlr.CommonAST;
import java.io.*;
import antlr.*;
import antlr.TokenStreamException;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;
}

class SimpTreeWalker extends TreeParser;

options {
    importVocab = simp;
}

{
SEgen se;
OutputMethods output=null;
int depth=0;
}


fileroot returns [String r]
{
r="";
String res="";
String finalRes = "";
String f = "";
ArrayList paramlist;
String out = "";
}
: #(FILEROOT
{
if (depth==0) {
output = new OutputMethods(); se = new SEgen("Global");
} else {
//if you want to create a new scope to include a file
//SEgen newScope = new SEgen("FILEINCLUDE");
//newScope.parent = se;
}
//includes will call this node against, but depth will have increased
depth++;

}
(res=bodyfile {
```

```
r+=res;
out+=res;
})+
{
depth--;
if (depth==0) { output.write(out); output.close(); }
}

)
|#(LCURLY
{
SEgen newScope = new SEgen("LCURLY");
newScope.parent = se;
se = newScope;String ifResult = "";
}
(finalRes=bodyfile {ifResult+=finalRes;
})+
{
output.debug("Value of ifresult");
r+=ifResult;
output.debug("CURLY" + " : " + f);
se = se.parent;
})
|#(DOLLV idvar:ID paramlist=paramdef
   {
   ArrayList fnargs;
   CommonAST tree;SEgen funSe;
   String varName = idvar.getText();
   }
{
Object obj;
if (paramlist.isEmpty()){ //it's a regular variable
obj=se.var(varName);
if (obj==null)
output.error("ERROR: calling undeclared variable: \""+varName+"\"");
r+=(String)obj;

} else { //it's a function(ish)
obj=se.funName(varName);
if (obj==null)
output.error("ERROR: calling undeclared function: \""+varName+"\"");
funSe=(SEgen)obj;
tree = (CommonAST)funSe.funcTable.get(varName);

//obj=se.var(varName);
output.debug("Value of scope " + funSe.name);
```

```
obj=funSe.symTable.get(varName);
if (obj==null)
output.debug("ERROR: argument list not stored properly for function: "+varName);
fnargs=(ArrayList)obj;

SEgen newScope = new SEgen(varName);
newScope.parent = se;
se = newScope;

if (fnargs.size()!=paramlist.size()) {
//complain, throw an exception, or something?
output.debug("Expected "+fnargs.size()+" args, got "+paramlist.size());
break;
}

//add params to the symboltable in this new scope.
for (int i=0; i<fnargs.size(); i++) {
se.symTable.put(fnargs.get(i),paramlist.get(i));
}

output.debug("OK. We're going to try to walk the *function* subtree now" + varName);
f=fileroot(tree);//.getFirstChild());
//output.debug("result of fileroot(tree): "+f);
output.debug(f);
output.debug("#FN("+varName+","+paramlist+"):"+f);

r+=f;

se=se.parent;
}
}
)
|#(WHILEBODY
{
SEgen newScope = new SEgen("WHILE");
newScope.parent = se;
se = newScope;
String whileResult = "";
}
(finalRes=bodyfile {whileResult+=finalRes;
})+
{
r+=whileResult;
output.debug("CURLY" + " : " + f);
se = se.parent;
})
```

```
    |#(DEFBODY
    {
        SEgen newScope = new SEgen("DEF");
        newScope.parent = se;
        se = newScope;
    } (finalRes=bodyfile {f+=finalRes;} )*
        {
            if (r!=null)
            {r+=f; output.debug("DEF : " + f);}
            se=se.parent;
        })

|#(FORBODY {String forResult="";} (finalRes=bodyfile {forResult+=finalRes;} )+
{
r+=forResult;
output.debug("FOR" + " : " +  r + "and f " + forResult);
})
|#(NEXTBODY {String nextResult="";} (finalRes=bodyfile {nextResult+=finalRes;} )+
{
r+=nextResult;
output.debug("NEXT" + " : " +  r + "and f " + nextResult);
})
|#(DO {String doResult="";}(finalRes=bodyfile {doResult+=finalRes;} )+
{
r+=doResult;
output.debug("DO" + " : " +  r + "and f " + doResult);
})
|#(WRITEBODY
{
SEgen newScope = new SEgen("WRITEBODY");
newScope.parent = se;
se = newScope; String toWrite = "";
}
(finalRes=bodyfile {toWrite+=finalRes;} )+
{

output.debug("Writing to file - " + " : " + toWrite);
r+=toWrite;
se = se.parent;
})
|#(EXECBODY
{
SEgen newScope = new SEgen("EXECBODY");
newScope.parent = se;
se = newScope; String toWrite = "";
}
(finalRes=bodyfile {toWrite+=finalRes;} )+
{

output.debug("String to Exec - " + " : " + toWrite);
r+=toWrite;
se = se.parent;
})

|#(TOWRITE
{
SEgen newScope = new SEgen("TOWRITE");
newScope.parent = se;
se = newScope; String toWrite = "";
}
(finalRes=bodyfile {toWrite+=finalRes;} )+
{

output.debug("File to write- " + " : " + toWrite);
r+=toWrite;
se = se.parent;
})
|#(FILEBODY (finalRes=bodyfile {f+=finalRes;} )*
{
output.debug("file" + " : " + f);r+=f;
})
|#(GETTOKENBODY (finalRes=bodyfile {f+=finalRes;} )*
{
output.debug("reading gettoken delimiters" + " : " + f);
r+=f;
})
;

bodyfile returns [String r]
{
r="";
String result="";
ArrayList argslist;
ArrayList paramlist;
ArrayList nextbodies;
String i6;
Integer calcResult = new Integer(0);
Boolean condRet = false, whileCondRet = false;
String ifbody = "";
//CommonAST fntree;
```

```
}
: #(DEF i1:ID argslist=argsdef defpart:.  //(result=bodyfile)+
{
Object obj; String id=i1.getText();

// STOP AND READ !!!!!!!!!!!!!!!!!!!!!!!!!
//IF YOU FIX ANYTHING HERE, YOU SHOULD GO FIX IT BELOW AT #(SET TOO!!!

if (argslist.isEmpty()) { //regular var
obj=se.var(id);
result=fileroot(defpart);
if (obj!=null) {
output.error("ERROR: redeclaration of variable: \""+id+"\"");
//se.reset(id, result);
} else {
output.debug("Declaring variable \""+id+"\"");
se.varDef(id,result);
}

output.debug("variable #def("+id+"): "+result);
} else {  //it's a function(ish)
obj=se.funName(id);
CommonAST tree; SEgen funSe;
output.debug("Declaring function "+id);
funSe=(SEgen)obj;

if (obj!=null) {
output.error("ERROR: redeclaration of variable: \""+id+"\"");
//se.resetfun(id,argslist,defspart);
} else {
//halfwalk(rest);
//ASTFrame frame = new ASTFrame("AST from the function "+i1.getText(), defpart);
    //frame.setVisible(true);
se.fndef(id,argslist,defpart);
}
output.debug("function #def("+id+"): "+argslist);
}

}
   )
 |
 #(SET is1:ID argslist=argsdef defspart:.  //(result=bodyfile)+
{
Object obj; String id=is1.getText();

// STOP AND READ !!!!!!!!!!!!!!!!!!!!!!!!!
```

```
//IF YOU FIX ANYTHING HERE, YOU SHOULD GO FIX IT ABOVE AT #(DEF TOO!!!

if (argslist.isEmpty()) { //regular var
obj=se.var(id);
result=fileroot(defspart);
if (obj!=null) {
//output.error("ERROR: redeclaration of variable: \""+id+"\"");
se.reset(id, result);
} else {
output.debug("Declaring variable \""+id+"\"");
se.varDef(id,result);
}

output.debug("variable #def("+id+"): "+result);
} else {  //it's a function(ish)
obj=se.funName(id);
CommonAST tree; SEgen funSe;
output.debug("Declaring function "+id);
funSe=(SEgen)obj;

if (obj!=null) {
//output.error("ERROR: redeclaration of variable: \""+id+"\"");
se.resetfun(id,argslist,defspart);
} else {
//halfwalk(rest);
//ASTFrame frame = new ASTFrame("AST from the function "+i1.getText(), defpart);
    //frame.setVisible(true);

se.fndef(id,argslist,defspart);
}
output.debug("function #def("+id+"): "+argslist);
}
}
   )
| #(FOR forPart:. nextbodies=nextbods doPart:.  //(result=bodyfile)+
{
SEgen newScope = new SEgen("FOR");
newScope.parent = se;
se = newScope;

r+=fileroot(forPart);
r+=fileroot(doPart);
Iterator iter = nextbodies.iterator();
while(iter.hasNext())
{
fileroot((AST)iter.next());
r+=fileroot(doPart);
}
se=se.parent;
}
   )
| #(CALC calcResult=mathexpr
{ //output.debug("Entering calc");
r+=calcResult;
output.debug("CALC result: "+r);
//output.debug("Leaving calc");
})
| #(VAR
{
int argCounter = 0;
//output.debug("Matched VAR ... going to fileroot");
}
result=fileroot
{
output.debug("VAR: "+result);
//output.debug(result);
r+=result;
}
)
| #(STRINGCONSTFILE
(i4:. {
String stringConst = i4.getText();
r+=stringConst ;
})*

  )
| #(STRINGCONST (i5:.
  { //output.debug("For String COnstant");
  r+=i5.getText();
  output.debug(i5.getText());
  })*
)
| #(EXEC execResult:EXECBODY
  {;
  String argValue = fileroot(execResult);
  output.debug("StringConst is " + argValue);
  r+=se.sysExec(argValue);
  })
 | #(IF cond1Ret:. ifpart:. btw:BTW_IF_AND_ELSE (elsepart:.)?
  {
  //output.debug("Valye of ret "  + condRet + "what is r " + r) ;
  Boolean x = cond(cond1Ret);
  //output.debug("Vale of x "+ condRet);
if(x){
  output.debug("inside condret = true " );
  r+=fileroot(ifpart);
  } else if(elsepart!=null){

    r+=fileroot(elsepart);
}
  }
)
 | #(WHILE whilecond:. whilepart:.
  {
  //output.debug("Valye of ret "  + condRet + "what is r " + r) ;
  Boolean x = cond(whilecond);
  //output.debug("Vale of x "+ condRet);
  int i=0;
while(x){
//i++;
  output.debug("inside condret = true " + x + i);
  r+=fileroot(whilepart);
  /* if(i>5)
  break;*/
  x = cond(whilecond);
  output.debug("Value of conditional return " + x);
  }
}
)
 | #(WRITE fileToWrite:TOWRITE writebody:WRITEBODY
  {
String fileWriter = "";
  try {
  fileWriter = fileroot(fileToWrite);
  output.debug("Writing to file - " + fileWriter);
        BufferedWriter out = new BufferedWriter(new FileWriter(fileWriter));
        out.write(fileroot(writebody));
        out.close();
     } catch (IOException e) {
     output.error("ERROR: can't write to file: " + fileWriter);
     }

}
)

| #(FILE fileToInclude:FILEBODY
{
String file=fileroot(fileToInclude);
```

```
DataInputStream input=null;
try {
input = new  DataInputStream(new FileInputStream(file));
}
catch ( FileNotFoundException iox ) {
output.error("ERROR: including file that doesn't exist: "+file);
}

try {
  System.out.println("Including file "+file+" (error message lines refer now to this fi
  //If there's any errors in the included file you should now that the lines are
  //relative to it

  //we need to show this with the console error
      // Create the lexer and parser and feed them the input
      SimpLexer lexerincluded = new SimpLexer(input);

  SimpParser parserincluded = new SimpParser(lexerincluded);

      parserincluded.file();
      CommonAST treeincluded = (CommonAST)parserincluded.getAST();
  //ASTFrame frame = new ASTFrame("Tree of include", treeincluded);
      //frame.setVisible(true);
      r+=fileroot(treeincluded);

      System.out.println("End including "+file);
      //end it

    } catch(Exception e) {
     e.printStackTrace();
     output.error("ERROR: parsing the included file: "+file);
    }

}
)
| #(GETTOKEN
varid:ID (gettokenbody:GETTOKENBODY)?
{
String varName=varid.getText();
Object obj=se.var(varName);
if (obj==null)
output.error("ERROR: gettoken of undefined variable "+varName);

String target=(String)obj;
String delim;
if (gettokenbody!=null){
```

```
delim=fileroot(gettokenbody);
output.debug("Gettoken delimiters: "+delim);
}
else{
output.debug("Gettoken: no delimiters specified, assuming blank space only");
delim=" ";
}


output.debug("Gettoken reads: "+target);

int k=0;int d;
boolean keepgoing=true;
while (k<target.length() && (keepgoing))
{
for (d=0;d<delim.length();d++)
if (target.charAt(k)==delim.charAt(d))
{keepgoing=false; break;}
if (keepgoing)
{
r+=target.charAt(k);
//append everything until the first delimiter
//that's the token that is output
}
k++;
}
while (k<target.length()){
for (d=0;d<delim.length();d++)
if (target.charAt(k)==delim.charAt(d))
{break;}
if (d==delim.length()) break; else k++;
}
//output.debug((String)se.var(varName));
//set the value with what is remaining
if (!se.reset(varName,target.substring(k)))
output.error("ERROR: Gettoken has undeclared variable: \""
+varName+"\" as argument (reminder: functions are not allowed)");
output.debug("Gettoken sets "+varName+" to "+target.substring(k));
}
)
;

/*
halfwalk returns [CommonAST t]
{
}
: #( '{'
*/

cond returns [Boolean r]
{
r = false;
Object r1 = 0;
Object r2 = 0;
}
: #(EQUAL r1tree:. r2tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
try{
r1 = stringConstCond(r1tree);
// output.debug("Value of tree" + r1);
if (r1.equals("")){throw new Exception("");};
r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
  }catch (Exception e){
try{

r1 = dollarVAR(r1tree);
if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);

}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r1tree);
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
r2 = stringConstCond(r2tree);
if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) == 0){
```

```
r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r2tree);
if(r2.equals("")){throw new Exception("");};
r2type = 2;
if(r1type == 1 || r1type==2)
{resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) == 0){

r=true;
}
}
//output.debug("Value of exception tree" + r2);
}catch (Exception e1)
{
r2 = mathexpr(r2tree);
r2type = 3;
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();

}
}

//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}

resultFound = 1;
r=r1Val==r2Val? true:false;

}

}
)
```

```
 | #(GT r3tree:. r4tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
try{
r1 = stringConstCond(r3tree);
// output.debug("Value of tree" + r1);
if (r1.equals("")){throw new Exception("");};

r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
  }catch (Exception e){
try{

r1 = dollarVAR(r3tree);
if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);
//output.debug("Value of exception tree" + r1);
}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r3tree);
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
r2 = stringConstCond(r4tree);
if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) > 0){

r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r4tree);
if(r2.equals("")){throw new Exception("");};
r2type = 2;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) > 0){

r=true;
}
}
//output.debug("Value of exception tree" + r2);
}catch (Exception e1)
{
r2 = mathexpr(r4tree);
r2type = 3;
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();

// return( ((Integer)r1).intValue()> ((Integer)r2).intValue());
}
}

//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}

resultFound = 1;
r=r1Val>r2Val? true:false;

}

}
)
| #(LT r5tree:. r6tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
try{
r1 = stringConstCond(r5tree);
// output.debug("Value of tree" + r1);
```

```
if (r1.equals("")){throw new Exception("");};
r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
  }catch (Exception e){
try{

r1 = dollarVAR(r5tree);
if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);
//output.debug("Value of exception tree" + r1);
}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r5tree);
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
r2 = stringConstCond(r6tree);
// output.debug("Value of tree" + r1);
if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) < 0){

r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r6tree);
if(r2.equals("")){throw new Exception("");};
r2type = 2;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) < 0){
```

```
r=true;
}
}
//output.debug("Value of exception tree" + r2);
}catch (Exception e1)
{
r2 = mathexpr(r6tree);
r2type = 3;
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();

// return( ((Integer)r1).intValue()> ((Integer)r2).intValue());
}
}

//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
output.debug("Downgrading");
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}
output.debug("Value of r1 " + r1Val + " " + r2Val);
resultFound = 1;
r=r1Val<r2Val? true:false;

}
}
)
| #(LE r7tree:. r8tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
try{
r1 = stringConstCond(r7tree);
// output.debug("Value of tree" + r1);
if (r1.equals("")){throw new Exception("");};
r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
  }catch (Exception e){
try{

r1 = dollarVAR(r7tree);
```

```
if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);
//output.debug("Value of exception tree" + r1);
}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r7tree);
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
r2 = stringConstCond(r8tree);
if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) <= 0){

r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r8tree);
if(r2.equals("")){throw new Exception("");};
r2type = 2;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) <= 0){

r=true;
}
}
}catch (Exception e1)
{
r2 = mathexpr(r8tree);
r2type = 3;
```

```
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();

}
}

//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
output.debug("Downgrading");
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}
output.debug("Value of r1 " + r1Val + " " + r2Val);
resultFound = 1;
r=r1Val<=r2Val? true:false;

}
}
)
| #(GE r9tree:. r10tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
try{
r1 = stringConstCond(r9tree);
// output.debug("Value of tree" + r1);
if (r1.equals("")){throw new Exception("");};
r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
  }catch (Exception e){
try{

r1 = dollarVAR(r9tree);
if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);
//output.debug("Value of exception tree" + r1);
}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r9tree);
```

```
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
r2 = stringConstCond(r10tree);
// output.debug("Value of tree" + r1);
if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) >= 0){

r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r10tree);
if(r2.equals("")){throw new Exception("");};
r2type = 2;
if(r1type == 1 || r1type==2)
{
resultFound = 1;
output.debug("value of r2 "  + r2 + " " + r2type);
if(((String)r1).compareTo((String)r2) >= 0){

r=true;
}
}
//output.debug("Value of exception tree" + r2);
}catch (Exception e1)
{
r2 = mathexpr(r10tree);
r2type = 3;
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();

// return( ((Integer)r1).intValue()> ((Integer)r2).intValue());
}
}
```

```
//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
output.debug("Downgrading");
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}
output.debug("Value of r1 " + r1Val + " " + r2Val);
resultFound = 1;
r=r1Val>=r2Val? true:false;

}
}
)
| #(NE r11tree:. r12tree:.
{
int r1type =0 ; int r2type = 0; int r1Val=0; int r2Val=0;
int resultFound = 0;
if (r11tree.getText().equals("STRINGCONSTCOND")) {
output.debug("content:"+r11tree.getText());
r1 = stringConstCond(r11tree);
r1type=1;
output.debug("value of r1 "  + r1 + " " + r1type);
}
  else {
try{

r1 = dollarVAR(r11tree);
//if(r1.equals("")){throw new Exception("");};
r1type= 2;
output.debug("value of r1 "  + r1 + " " + r1type);
//output.debug("Value of exception tree" + r1);
}catch (Exception e1)
{
output.debug("r1 " + r1.getClass());

r1 = mathexpr(r11tree);
r1type = 3;
output.debug("value of r1 "  + r1 + " " + r1type + " " + r1.getClass());
r1Val=((Integer)r1).intValue();
}
}
try{
```

```
r2 = stringConstCond(r12tree);
// output.debug("Value of tree" + r1);
//if (r2.equals("")){throw new Exception("");};
r2type = 1;
if(r1type == 1 || r1type==2)
{
output.debug("value of r2 "  + r2 + " " + r2type);
resultFound = 1;
if(((String)r1).compareTo((String)r2) != 0){
output.debug("found "  + r2 + " " + r1);

r=true;
}

}
  }catch (Exception e){
try{
r2 = dollarVAR(r12tree);
//if(r2.equals("")){throw new Exception("");};
r2type = 2;
output.debug("r1type in r2 "  + r1type + " " + r2type);
if(r1type == 1 || r1type==2)
{
output.debug("value of r2 "  + r2 + " " + r2type);
resultFound = 1;
if(((String)r1).compareTo((String)r2) != 0){

r=true;
}
}
//output.debug("Value of exception tree" + r2);
}catch (Exception e1)
{
r2 = mathexpr(r12tree);
r2type = 3;
output.debug("value of r2 "  + r2 + " " + r2type + r2.getClass());
r2Val=((Integer)r2).intValue();
}
}

//Either r1 or r2 or both have integer values
//Try to downgrade to Integer
if(resultFound == 0){
output.debug("Downgrading");
if(r1type != 3){
r1Val = Integer.parseInt((String)r1);
}
if(r2type != 3){
r2Val = Integer.parseInt((String)r2);
}
output.debug("Value of r1 " + r1Val + " " + r2Val);
resultFound = 1;
r=r1Val!=r2Val? true:false;

}
}
)
;

stringConstCond returns [String r]
{
r = "";
String inter= "";
} : #(STRINGCONSTCOND ((a:.) {
if (a!=null){
output.debug("Value of string const IN CONDITION" + a.getText()) ;
inter+=a.getText();
}
})*
{
output.debug("Value of inter" + inter);
r+=inter;
}
);


paramdef returns [ArrayList p]
{
p = new ArrayList();
AST t;
String s;
}
: #(ARGS (a:.
{
s=parameter(a.getFirstChild());
output.debug("PARAM is: "+s);
p.add(s);
})*
)
;

parameter returns [String s]
```

```
{
s="";
}
: #(VAR s=fileroot)
| #(STRINGCONSTARG (a:. {s+=a.getText();})*)
;

/*
t=i5.getFirstChild();
if (t.getNumberOfChildren()>0) {
s=bodyfile(t);
} else {
//output.debug("Executing paramdef 'ELSE' --- calling getText()");
//output.debug("t has "+t.getNumberOfChildren()+" Children");
s=t.getText();
}
output.debug("param is: "+s);
p.add(s);
})*
)
;
*/

argsdef returns [ArrayList a]
{a = new ArrayList();}
 : #(DARGS (i5:.
  {
  a.add(i5.getFirstChild().getText());
  output.debug("arg:"+i5.getFirstChild().getText());
  }
  )*
)
 ;

nextbods returns [ArrayList a]
{a = new ArrayList();}
 : #(NEXTBODS (i5:.
  {
  output.debug("Sending bACK BODIES");
  a.add(i5);
  //out.write("arg:"+i5.getFirstChild().getText());
  }
  )*
)
 ;
```

```
mathexpr returns [Integer r]
  {
   r=0;
   int a,b;
  }
  : #(PLUS  a=mathexpr2 b=mathexpr2
   {
   r+=(a+b);
   })
  | #(MINUS a=mathexpr2 b=mathexpr2
   {
   r+=(a-b);
   })
  | a=mathexpr2
  {r+=a;}
  ;

mathexpr2 returns [Integer r]
  {
   int n1,n2;
r=0;
  }
  : #(MUL  n1=mathatom n2=mathatom
   {
   r=n1*n2;
   })
  | #(DIV   n1=mathatom n2=mathatom
   {
r=n1/n2;
   })
| n1=mathatom
{r=n1;}
  ;

mathatom returns [Integer r]
  {
r=new Integer(0);
int v=0;
Integer check;
String varReturn = "";
  }
  : i:NUMBER
   {
r=se.num(i.getText());
   }
  | varReturn=dollarVAR
  {
   r=se.num(varReturn);
  }
  | #(CALC check=mathexpr
{ output.debug("Entering nested calc");
r=check;
output.debug("" + r);
})
  ;

dollarVAR returns [String r]
  { r="";}
  : #(VAR r=fileroot
   )
  ;
```

A.1.3 SEgen.java

```
package code.test;

import java.io.BufferedReader;

public class SEgen {
// String path;
// need an output buffer/file handle, whatever the hell you call it in java.
// probably want the symbol table here?
// FileOutputStream fout;
// For writing to a file, you call getWriter()
// In ANTLR finally, we need to do se.getWriter().close()

public Map symTable;

Map funcTable;

Map argsList;
public String name;

public SEgen parent = null;

// CommonAST currTree;

/*
 * When we encounter a def statement, we - Create a new scope object, this
 * scope object has - a symbol table Map => contains all resolved variables
 * and functions. In case of functions, it will contain a List of function
 * parameters. This can also be looked upon as a Map of resolved trees. The
 * fact that function parameters are stored will indicate that this is a
 * function and needs to be handled as one. This way, functions which have
 * arguments that are never used will be treated as variables ;) - a
 * argument list - In case this scope is actually a function call. We can
 * have an abstract view of every scope to be a function. If this is a
 * normal scope, this list will be empty. When a variable is looked up,
 * during first walk of the tree, it will be first looked up in the symbol
 * table and then here. If it is found here, we know this is a function
 * call, thus an entry can be made of that scope into the Map of functions. -
 * a Map of functions , containing the function => AST trees. These AST
 * trees may be semi-walked. Is there any way of doing that?? Need to assure
 * myself of the syntax. - A reference to the parent symbol table, to keep
 * track of variable references of the parent. Would not make any sense in a
 * function, when it is called, because all such references are resolved. -
 * When we encounter a def statement, create a new scope (pass the parent
 * scope to it), we just try to walk the tree, resolving as many references
 * as possible. The variable should be either in the symbol table OR in the
 * args list. We keep this half resolved tree AS IS in the function table. -
 * When a function call is made, we just place the parameter into the symbol
 * table and walk the tree. The key is, we do not pass the parent table
 * reference to it. If the tree finds the variable, very good. Otherwise too
 * bad. This is good, as we ensure when walking the tree first time that a
 * variable reference should either be in the argument list or a symbol
 * table
 */

public SEgen(String name) {
AST t;
// t.equalsTree();
symTable = new HashMap(); // create a new one for every Scope
funcTable = new HashMap();
argsList = new HashMap();
this.name= name;
// for test
// symTable.put("check" , "5");
}

/*
 * public BufferedWriter getWriter() { if (this.parent == null) return
 * this.out; else return parent.getWriter(); }
 */

/*
 * This is how i see functions executing.. Each function would accept a list
 * that it cares about and writes to files using the filehandle
 */
public String sysExec(String instr) {
char dest;
String token;
ArrayList args = new ArrayList();

int k=0;
while (k<instr.length()){
if (k<instr.length() && (instr.charAt(k)=='\'' || instr.charAt(k)=='\"'))
{ dest=instr.charAt(k); k++; }
else dest=' ';
token="";
while (k<instr.length() && instr.charAt(k)!=dest) {token+=instr.charAt(k); k++;}
k++;
while (k<instr.length() && instr.charAt(k)==' ') k++;
args.add(token);
}

Runtime runner = Runtime.getRuntime();
Process pro;
InputStream is;
InputStreamReader isr;
String line;
BufferedReader br;
StringBuffer res = new StringBuffer();
try {
String[] array = (String[])args.toArray(new String[args.size()]);
pro = runner.exec(array);
int check = pro.waitFor();
if (check == 0) {
is = pro.getInputStream();
isr = new InputStreamReader(is);
br = new BufferedReader(isr);
while ((line = br.readLine()) != null) {
res.append(line);
res.append("\n");
}
br.close();
isr.close();

is.close();
} else {
is = pro.getErrorStream();
isr = new InputStreamReader(is);
br = new BufferedReader(isr);
// System.out.println("Error message for exex:");
// getWriter().write("Error message for exex:");
while ((line = br.readLine()) != null) {
// getWriter().write(line);
res.append(line);
res.append("\n");

}
br.close();
isr.close();

is.close();
}
} catch (Exception e) {
// TODO Auto-generated catch block
e.printStackTrace();
} finally {
// System.out.println("Value is res " + res.toString());
return res.toString();
}
}

public void calc() {
}

public int num(String s) {
return Integer.parseInt(s);
}

public boolean reset(String var, String value) {
Object o = symTable.get(var);
String res = "";
SEgen thisParent = null;

if (o!=null) {
symTable.put(var, value);
return true;
}

thisParent = this.parent;
while (thisParent != null) {
o = thisParent.symTable.get(var);
if (o != null){
thisParent.symTable.put(var, value);
return true;
// at this point we have the parent where it is supposed to be
// found.. so
// we can change it
// Not sure if we should allow this within functions
// because it should surely be allowed in other bodies
// Maybe we will have to keep a flag for functions.. in which case
// we disallow parent lookup
}
else
thisParent = thisParent.parent;
}

return false;

}

public Object var(String id) {
// putting in logic for lookup of normal vars

Object o = symTable.get(id);

String res = "";

if (o == null) {
if (this.parent != null) {
o = this.parent.var(id);
// System.out.println("checking in scope "+this.parent.name);
}
}
return o;
}


public boolean resetfun(String id, ArrayList args, AST tree) {
Object o = funcTable.get(id);
String res = "";
SEgen thisParent = null;

if (o!=null) {
symTable.put(id, args);
funcTable.put(id, tree);
return true;
}

thisParent = this.parent;
while (thisParent != null) {
o = thisParent.funcTable.get(id);
if (o != null){
```

```
thisParent.symTable.put(id, args);
thisParent.funcTable.put(id, tree);
return true;
// at this point we have the parent where it is supposed to be
// found.. so
// we can change it
// Not sure if we should allow this within functions
// because it should surely be allowed in other bodies
// Maybe we will have to keep a flag for functions.. in which case
// we disallow parent lookup
}
else
thisParent = thisParent.parent;
}

return false;

}

public Object funName(String id){
Object o = funcTable.get(id);

if (o != null){
return this;
} else if (this.parent != null) {
return this.parent.funName(id);
} else {
return null;
}
}

/*
public Object funName(String id) {
//Object o = funcTable.get(id);
SEgen foundAt = null;
String res = "";

if (this.funcTable.get(id) == null) {

if (this.parent != null) {
 foundAt = (SEgen)this.parent.funName(id);
 //foundAt = (SEgen) o;
}
}

return foundAt;
```

```
}
*/
public Object fun(String id) {
Object o = funcTable.get(id);
String res = "";

if (o == null) {

if (this.parent != null) {
o = this.parent.fun(id);
}
}
return o;
}

// TODO: this really should be void and not returning debugging stmts
public void varDef(String id, String val) { // , ArrayList args, Object
// tree) {
symTable.put(id, val);
}

public void fndef(String id, ArrayList args, AST tree) {
symTable.put(id, args);
funcTable.put(id, tree);
/*
 * ASTFactory fac = new ASTFactory(); CommonAST fntree = new
 * CommonAST(new CommonToken(
 * code.test.SimpTreeWalkerTokenTypes.FILEROOT, "FILEROOT"));
 * fntree.addChild((CommonAST) fac.dupTree(tree));
 *
 * System.out.println("Display function tree graphically"); ASTFrame
 * fnframe = new ASTFrame(("AST for fn: "+id), fntree);
 * fnframe.setVisible(true);
 */
}

/*
 * System.out.println("no arguments"); } else { // build 'tree' into full
 * scriptedit tree, soas can walk // it w/our simptreewalker when the
 * function is called. ASTFactory fac = new ASTFactory(); CommonAST fntree =
 * new CommonAST(new CommonToken(
 * code.test.SimpTreeWalkerTokenTypes.FILEROOT, "FILEROOT")); SimpTreeWalker
 * stw = new SimpTreeWalker();
 *  // add the function tree as the first child fntree.addChild((CommonAST)
 * fac.dupTree((CommonAST) tree)); System.out.println("********" +
 * fntree.toStringList());
 *
 * funcTable.put(id, fntree); symTable.put(id, args);
 *  }
 */
}
```

A.1.4 OutputMethods.java

```
package code.test;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import antlr.collections.AST;

import antlr.*;

public class OutputMethods{

BufferedWriter out;

public OutputMethods() {
// Only at the root... create the file writer here
try {
debug("OUT FILENAME: "+Main.fileout);
out = new BufferedWriter(new FileWriter(Main.fileout));

// the sehtml file that is executed will be the root.. and will
// tell us the name of the file to be generated
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}

public void write(String s) {
try {
```

```
this.out.write(s);
}
catch ( IOException iox )
{
System.out.println("ERROR: writing the output file");
}
}

public static void error(String s){
System.err.println(s);
System.exit(0);

}

public static void debug(String s) {
//System.out.println(s);
}

public void close(){

try {
debug("CLOSE OUT FILENAME: "+Main.fileout);
out.close();
}
catch ( IOException iox )
{
System.out.println("ERROR: writing the output file");
}
}

}
```

A.2 Testing

A.2.1 calc.txt.se

```
#if(#calc(1+5)=6){plus...success}
#else{plus...FAILURE}

#if(#calc(10-5)=5){minus..success}
#else{minus...FAILURE}

#if(#calc(10000*2)=20000){multiplication...success}
```

```
#else{multiplication...FAILURE}

#if(#calc(10/2)=5){division...success}
#else{division...FAILURE}

#if(#calc(20/7)=2){calc truncs to integers...success}
#else{calc truncs to integers...FAILURE}

#if(#calc(#calc(5+8) *  #calc(5-3))=26){complex calc...success}
#else{complex calc...FAILURE}
```

A.2.2 def.txt.se

```
VARIABLES:
+++++++++++
test simple variable definition & call: #def(n){0}
$(n) should be 0
reassignment: #set(n){1}
$(n) should be 1
self-assignment: #set(n){#calc($(n)+1)}
$(n) should be 2
strings: #set(s){several words}
$(s)
check ws rules in set & call: #set(s){


s
t
r
i
n
g


}
$(s)

more ws: #set( s
)


{s!}
$(s) =?= s!
++++++++++++
FUNCTIONS:
++++++++++++
test simple function definition & call: #set(double,a){#calc(2*$(a))}
$(double,3) should be 6
pass a variable: #set(n){4} $(double,$(n)) should be 8
multiple arguments: #set(printly,a1,a2){  ^^$(a1)vv$(a2)^^  }
$(printly,boe,eob)
$(printly, a b c,c b a )
with escaped characters in params:
$(printly, a\,b\,c,c\,b\,a )

checking ws rules in fn set&call:
#set( ws_,
a,
b,
c){$(a),$(b),$(c)}
should define a function, no errors
$(ws_,| ,  ,   |) should look like this: | ,  ,   |
$(ws_, $(s),$(s) , $(s) ) should break

+++++++++++
FUNCTION SCOPING:
+++++++++++
#set(n){0}
fn double that uses a local copy of n=2
#set(double,a1)
{#set(n){2}
#calc($(n)*$(a1))}
$(double,6) should be 12
$(n) should be 0

fn mult that uses global n
#set(mult,a1)
{#calc($(n)*$(a1))}
$(mult,6) should be 0
$(n) should be 0

fn double that sets global n
#set(double,a1)
{#set(n){2}
#calc($(n)*$(a1))}
$(n) should be 0
$(double,6) should be 12
$(n) should be 2
```

```
#set(deftest){
#de
#set(empty){}
$(empty)
#set(greeting){hello world}
#set(myName){JonSmith}
$(greeting), My Name is $(myName)
defining a variable and changing its value...
#set(num){5} Now testing set and set
#set(num){4}
$(num) should be equal to 4
#set(emptyString, a2){}
empty here >>$(emptyString, )<<

#set(my_Name){JonSmith}
#set(myName5){JonSmith}

}

$(deftest)
```

A.2.3 exec_python.txt.se

```
#def(python)
{a=['Bethany','Bhavesh','MarC','Denni']
for item in a:
print 'Hello',item,'!'
}

#exec{python -c "$(python)"}
```

A.2.4 exec.txt.se

```
should ouput hello world!...
#exec{echo "hello world!"}
```

A.2.5 file.txt.se

```
including multiple test cases
```

```
>>>>>>>>>>>>>>>>>>>>the def test:
here it is: #file{code/examples/simpletests/def.txt.se}

>>>>>>>>>>>>>>>>>>>>the if test:
here it is: #file{code/examples/simpletests/if.txt.se}

>>>>>>>>>>>>>>>>>>>>plus a normal text file (the java main):
here it is: #file{code/test/Main.java}
```

A.2.6 func.txt.se

```
#def(double,arg1){#calc(2*$(arg1))}
$(double,7)
#def(triple,arg1){#calc($(double,$(arg1)) + $(arg1))}
tryout triple:
$(triple,$(triple,3))
#def(triple1,arg1){#calc($(triple,$(arg1)) + $(arg1))}
$(triple1,0)
```

A.2.7 gettoken.txt.se

```
#def(names){Denni, Bhavesh ; Bethany , MarC}
#gettoken(names){ ;,}
#gettoken(names){ ;,}
#gettoken(names){ ;,}
#gettoken(names){ ;,}

#set(names){Denni Bhavesh Bethany MarC}
#gettoken(names)
#gettoken(names)
#gettoken(names)
#gettoken(names)
```

A.2.8 if1.txt.se

```
#def(testflag2){else binding... SUCCESS}
#if(1=1){
#if(1=2){}
}
#else{#set(testflag2){else bound to wrong if....FAILURE}}
```

```
$(testflag2)

#def(testIflessElse){success}
Else appearing without an if...$(testIflessElse)
```

A.2.9 if.txt.se

```
#def(iftest){
Testing if/else...
#def(testflag){FAILURE}
#def(my_Name){JonSmith}
#if(4=4){
#set(testflag){success}
}
testing if statement with num=num... $(testflag)
#set(testflag){FAILURE}
#if($(my_Name)=JonSmith){
#set(testflag){success}
}
if string=string... $(testflag)
}
$(iftest)

#if(1=2){number comparison...FAILURE}
#else{basic else test...success}

#def(testflag2){else binding... SUCCESS}
#if(1=1){
#if(1=2){}
}
#else{#set(testflag2){else bound to wrong if....FAILURE}}
$(testflag2)

#def(testIflessElse){success}
Else appearing without an if...$(testIflessElse)
```

A.2.10 scoping.txt.se

```
testing scoping...
#def(m){0}
#def(var1){global}
#while($(m)<2)
```

```
{
#set(m){#calc($(m)+1)}
#set(var1){local}
$(var1)
}
$(var1)

next text...
Should output 1, 2, 3, 2, 1
#def(n){0}
#def(var2){1}
#while($(n)<1)
{
#set(n){#calc($(n)+1)}
#set(var2){2}
$(var2)
if($(var2)=2){
#set(var2){3}
$(var2)
}
$(var2)
}
$(var2)
```

A.2.11 stringconst1.txt.se

```
Patata caliente \#de \#defali #def. #defali \#def(b){}
test )\nSe
```

A.2.12 while.txt.se

```
testing while...

#def(i){1}
#while($(i)<3)
{
#set(i){#calc($(i)+1)}
loop $(i)
}

#def(j){3}
```

```
#while($(j)>1)
{
#set(j){#calc($(j)-1)}
loop $(j)
}

#def(k){#calc(3)}
#while($(k)< #calc(6))
{
#set(k){#calc($(k)+1)}
loop $(k)
}

#def(l){9}
#while($(l)< #calc(011))
{
#set(l){#calc($(l)+1)}
loop $(l)
}
```