SCL: Site Construction Language

Sudip Das
Clark Landis
Mohamed Nasser

COMS W4115 Programming Languages and Translators
May 13, 2003

Chapter 1

Introduction

1.1 What is SCL?

Site Construction Language (SCL) is a language for building web sites. SCL is a “glue” language that provides a framework for efficiently merging text, HTML, graphics and executable CGI scripts into a cohesive and easily navigable web site.

SCL is simple and easy to learn; it has small number of intuitive keywords and its syntax is similar to that of well known scripting languages. SCL is powerful: specialized commands and structures make it easy to generate entire sites in a few lines of code. SCL is efficient, because its core functions are highly optimized. SCL produces efficient sites, because SCL merges content without adding any additional code, comments, or styles, which have not been directly specified by the programmer. SCL uses libraries of modular CGI's to add features to the sites that it creates. SCL is well integrated with Unix, the OS of most web servers.

1.2 Why a new language: SCL?

There are many GUI-based web designing software packages available in the market today. GUI-based software is useful for designing web pages, but creating a dynamic website (or a well organized static website) involves more than just design work. These processes include writing CGIs, linking databases, integrating existing scripts, linking dynamic content, creating navigation systems, and creating administrative interfaces. The purpose of SCL is to simplify these tasks, allowing a user to efficiently and easily create a complex website.

The goals of SCL could largely be accomplished as a library to an existing language like Perl, but there are advantages to making SCL a stand-alone language:

1. SCL enables users to write cleaner and faster code; it is a language focused in website construction, and its built-in features are optimized for such tasks.

2. SCL is small and, easy-to-learn language; its syntax is similar to well known scripting languages

3. Many of SCL's primitives operate on files directly without making the programmer manage file handles.

4. SCL has special variables (like Perl's $_) that make web construction easier. For example, $LP holds a relative URL to the last page written to disk.

5. SCL should have a speed advantage over Perl, because it will be smaller, and the set of core functions, which it uses repeatedly, will be highly optimized. SCL is fast enough to generate pages on the fly.

1.3 Language Features

1.3.1 Template Processing

SCL processes templates. A template is a text file. SCL looks for "keywords" within a template and performs substitutions, usually insertions of larger blocks of text in their place. For example, suppose we had a template which included *header, *leftnav, and *body. The template was perhaps made by a graphic designer, and might look something like this.

```
<HTML><HEAD></HEAD><TITLE></TITLE>
<BODY bgColor=#ff9933 >
  *Header
 <TABLE cellspacing=0 cellpadding=0 border=0  >
 <TR align=center>
   <TD width=100>
    *LeftNav
   </TD>
   <TD width=400>
    *Body
    </TD>
 </TR>
</TABLE>
</BODY>
```

We can make a page, with just a few lines of SCL:

```
bind mybindings {
  *header   : header.jpg
  *body     : body.html
  *leftnav  : leftnav.html
};
$newpage = makepage("template.html", mybindings)
```

1.3.2 Smart Insertions

A powerful feature of the language is that the above insertions occur in a way that suits the file being inserted. Jpgs are inserted with <img> tags, html files are copied in, text files are converted using txt2html before being copied in, etc.

The lines below are syntactically the same in SCL, but have very different results.

```
bind {*body : body.html)
bind {*body : body.jpg)
```

CGI scripts and dynamic files are inserted into templates by way of SSI links.

```
bind {*body : body.cgi : SSI)
bind {*body : body.html : SSI)
```

1.3.3 Index generation

Suppose we want to make a set of pages from some existing content files: txt1.txt, txt2.txt, pic1.jpg and pic2.gif. In SCL, it is easy to generate a list of links from within a loop:

```
bind mybindings{
  *header   : header.jpg
  *leftnav  : leftnav.html : SSI
}
foreach $file in "txt1.txt txt2.txt pic1.jpg pic2.gif"
  {
     bind {*body : $file }
     $page = makepage("template.html",mybindings);
     write($page,$file.".html")
     link($LastPageLink,$file,"leftnav.html")
  }
```

$LastPageLink is a special variable containing a relative link to the last page created. At the end of the loop, leftnav.html contains a list of links to each page. Note that leftnav.html is dynamically linked to each page, so each page now contains a navigable index of the entire site.

1.3.4 Leveraging the power of the command line

Unlike popular GUI website builders, SCL is a scripting language, and as such it has access to a plethora of existing UNIX tools for text processing, image manipulation, html authoring and more. SCL treats commands it does not recognize as external programs; so integrating other Unix tools does not diminish SCL's readability.

"jslice" is a tool for slicing a graphic into pieces which fit perfectly into an html table. It can be to make templates among other things.
"ImageMagick" is a set of popular image manipulation tools
"text2html" is a utility for converting formatted text to html

Using these tools, someone could use SCL to make a photo album about their Summer Vacation with a really fancy border as follows:

```
jslice FancyBorder.gif 25 100 600 25 25 480 25;    / make a template with the appropriate
                                                     Geometry
text2html SummerVac.txt > SummerVac.html;          / convert some text to html.
$mydir="SummerVacDir";
bind mybindings{
   <img src="image6.jpg"> : leftnav.html : SSI;    /replace one of the slices
                                                      with a dynamic link to leftnav
}
foreach $file in (`ls $mydir`)                    / loop over the contents of a directory
{
     convert –size 640x480 $file.jpg $file.jpg;    /scale the picture to the right size
                                                   /with an ImageMagick tool.
        bind {
       <img src="image7.jpg"> : $file              /replace one of the slices with a pic
     }
     $page = makepage("FancyBorder.html",mybindings);
     write($page,$file.".html")
}
bind {
       <img src="image7.jpg"> : SummerVac.html    /replace one of the slices with a file
}
write(makepage("FancyBorder.html"),"index.html");
```

1.4 Conclusion

SCL will take all the tedium out of building websites. Programmers with a basic knowledge of Perl or shell script should be able to start making SCL scripts right away. Best of all, SCL will create sites that have been "pre-compiled" for the viewer. That is, content, including navigation system and borders, will have been pieced together before the user attempts to download the page. This "pre-compilation" deliveres content faster than any dynamic page generation method, and best of all, pages created in this way, at least the ones that do have dynamic content, can be cached, resulting in a large speedup in many situations.

Chapter 2

Language Tutorial

2.1 Installation

To install SCL on Linux, download and unzip the source code. You will see a directory called SCL. Go into the SCL directory and enter the following lines at the prompt. (Note that you must have root access to perform this installation.)

```
$ make
$ mkdir /usr/local/SCL/classes
$ make install
```

This will compile all the necessary files and move them to a directory called `/usr/local/SCL/classes`. It will also install a shell script called `SCL` in the `/usr/local/bin` directory. Be sure to add the `/usr/local/bin` directory to your PATH environment variable, to allow easy execution of SCL programs (see section 2.4).

2.2 Program Components

An SCL program consists of three main components:

2.2.1 Template file

An SCL program begins with a template. A template is a text file comprised of HTML skeleton code, with placeholders rather than actual content. The placeholders are replaced with content chosen by the user when SCL code is run.

2.2.2 Data files

Content that can be inserted into a template can include text files, HTML files, image files, CGI scripts, and dynamic files. These files should be created ahead of time by the user (or someone else) and placed in a directory prior to running an SCL file.

2.2.3 SCL file

The SCL file contains the code written in SCL. When executed, an SCL file can generate not just a web page, but an entire indexed web site based on the template and data files. Let us look at an example.

2.3 An Example

Here is `template.html`, an example of a template file:

```
<HTML><HEAD></HEAD><TITLE></TITLE>
<BODY bgColor=#ff9933 >
   PutHeaderHere
 <TABLE cellspacing=0 cellpadding=0 border=0  >
  <TR align=center>
   <TD width=100 bgColor=#99ff33>
    PutNavHere
   </TD>
   <TD width=400 bgColor=#ff9999>
    PutBodyHere
   </TD>
  </TR>
 </TABLE>
</BODY>
```

Notice that the template file is simply an HTML file, with placeholders such as `PutBodyHere` and `PutNavHere` put where you would normally put a body and links. If you are familiar with HTML,

The following is `simple.scl`, a very simple SCL program. It replaces the placeholders of the template with the specified files, and writes the resulting web page to `index.html`.

```
bind mybindings {
   PutHeaderHere : "header.html"  ;
   PutBodyHere : "body.jpg" ;
   PutNavHere : "nav.html";
}
writepage("template.html",mybindings,"index.html");
```

2.4 Running an SCL Program

Running an SCL program is simple once you have performed the installation. Once you have written the .scl file and have placed the template and data files in the directories that you specified in your .scl file, you can run the .scl file by entering the following at the prompt:

```
$ SCL myslfile.scl
```

The output pages will be placed in the output directory that you specified in your .scl file.

2.5 SCL Data Types

There are two basic data types in SCL: the binding, and the variable.

2.5.1 Bindings

A binding is how you match data files to template placeholders. Each binding has a unique name. Each line in a binding can have a modifier, for instance "nav.html" has an SSI flag that indicates it is a dynamic file.

```
bind mybindings {
  PutHeaderHere : "header.html" ;
  PutBodyHere : "body.jpg";
  PutNavHere : "nav.html
}
```

2.5.2 Variables

All variables in SCL are strings. They require no declaration and are initialized with the value "null". To distinguish from bindings, they have a dollar sign prefix.

There are several operators that act on variables or strings. The concatenation operator (a period) is used to join two strings. The typical mathematical operators can be used to do math on strings by treating them like numbers.

There are certain variable names that have been reserved for special variables. These names include `$LastPageLink`, `$LinkPath`, `$ReadPath`, `$WritePath`, and `$Return`. The use of these special variables will be described in the next few sections.

2.6 Functions and Loops

There are several functions built into SCL. These include `makepage, link, read, write,` and `writepage`. SCL also allows users to create their own functions.

2.6.1 Makepage

The makepage function merges the template with the specified binding, and returns a string containing HTML code for the resulting web page.

```
$mypage = makepage("template.html",mybindings)
```

Makepage makes smart insertions according to the type of data file you are trying to insert. HTML files are copied in. Text files are translated to HTML using txt2html, then copied in. Graphics files are linked with the necessary `<IMG>` tags.

2.6.2 Link

The link function adds to an index a link to a given web page, with a given label. For instance, the following adds to `nav.html` a link to `intro.html`, with the label `Introduction`.

```
link("intro.html", "Introduction", "nav.html")
```

There is a special variable, `$LinkPath` that, if defined, will automatically prepend the specified directory path to the linked pages. In this way, linked pages can be in a separate directory, but you do not have to explicitly write out the directory path with each call to `link`.

2.6.3 File Handling: Read, Write, Writepage

The `read` function takes a filename as input and returns the content of that file as a long string.

```
$mytemplate = read("template.html");
```

The `write` function writes a string to a file, overwriting any existing file of the same name. For instance, the following exports the contents of the variable `$mypage` to a file called `index.html`.

```
write($mypage, "index.html")
```

The `writepage` function combines `makepage` and `write` into one command.

```
writepage($mytemplate,mybindings,"index.html");
```

There are two special variables used in conjunction with file handlers: `$ReadPath` and `$WritePath`. They behave similarly to `$LinkPath`, except they act on the functions `read` and `write`.

2.6.4 Foreach

The foreach loop iterates over a list of strings separated by spaces. It executes the loop once for each element of the list. For example, the following lines of code will print "cat" "rat" and "dog" on three separate lines:

```
foreach $word in "cat rat dog"
{
      echo($word)

}
```

2.6.5 User-defined Functions

Users can also define their own functions. User-defined functions take one variable as input. Functions are declared as follows:

```
function #appendHTML ($name){
      $Return=$name.".html";
}
```

The special variable `$Return` stores the string that is returned by the function call. A function is called as follows:

```
$filename = #appendHTML("dog.jpg");
```

`The result of this function call is that $filename` contains the string `"dog.jpg.html"`.

2.7 Another Example

Here is an example of an SCL file that generates a web site comprised of an index file with links to four web pages, one corresponding to each of the four files listed in the `foreach` statement.

```
$mytemplate=read("template.html");

bind mybindings {
   PutHeaderHere : "header.html"  ;
   PutNavHere : "nav.html" : SSI;
}

write ("","nav.html");
foreach $file in "messy.jpg bath.jpg intro.html todo.html"
{
    bind mybindings {  PutBodyHere   : $file ; }
    writepage($mytemplate,mybindings,$file.".shtml");
    link ($LastPageLink,$file,"nav.html");
}
writepage($mytemplate,mybindings,"index.shtml");
```

Chapter 3

Language Reference Manual

3.1 Introduction

This manual describes the SCL programming language.

3.2 Lexical Conventions

A program consists of a website specification (a sequence of SCL statements) stored in a file, along with one or more text files that are used as templates for site construction. The user supplies text files, image files, HTML files, and/or CGI scripts to be included in the website.

3.2.1 Statement and Line Terminators

Statements are terminated with a semicolon. Lines are terminated with either a carriage return character `'\r'` followed by a newline character `'\n'`, or by a lone newline character.

3.2.2 Whitespace and Comments

Whitespace is defined as the space character `' '`, the tab character `'\t'`, a line terminator, or a comment. Comments can be single-line or multi-line. Single-line comments begin with the characters `//` and end with a line terminator. Multi-line comments are enclosed between `/*` and `*/`.

3.2.3 Tokens

There are six classes of tokens: identifiers, identifier types, keywords, string constants, operators, and separators. Whitespace is ignored except as a token separator. Whitespace may be required in order to separate otherwise adjacent tokens.

In the token separation process, the next token is the longest string of characters that could constitute a token.

3.2.4 Identifiers

An identifier consists of a letter followed by a sequence of letters, digits, periods, or underscores. An identifier must start with a letter. $ and # which denote identifiers as variables or functions respectively are seprate tokens.

3.2.5 Keywords

The following identifiers are reserved as keywords, and may be not used otherwise:
`link, bind, else, foreach, function, if, makepage, read, while, write, writepage, echo, erase, system, split, format_index, unbind.`

3.2.6 String Constants

A string constant is composed of a sequence of characters enclosed between a pair of double quotation marks. A double quote can be included in a string constant by using two double quotes.

A comand string constant is composed of a sequence of characters enclosed between a pair of back quotation marks. A back quote can be included in a string constant by using two back quotes. When SCL evaluates a command string, it returns the output of the enclosed command when executed by a unix shell.

3.2.7 Operators

These are in precedence order:

Mathematical operators: `unary -, * / , + -`
These operators do integer arithmetic. Floating point operations are not supported.

String Operators: .
. concatenates strings.

Comparison operators, == and !=
== returns "true" if its operands are equal and "false" if they are not.
!= returns "false" if its operands are equal and "true" if they are not.

All of these operators can be used in the same expression.

3.2.8 Separators

The following characters are separators: `{` `}` `(` `)` `:` `;`

3.3 Program Structure

SCL programs consist of a sequence of statements. Execution begins at the first statement and ends when control goes beyond the last statement in the file.

Scope is dynamic and operates differently for variables, bindings and functions.

3.3.1 Variable Scope

Each function call spawns a new scope, previous scopes are accessible and shall be referred to as outer scopes.

Function parameters are always created in the scope that is created with that function.

When an assignment to a variable first occurs after a new scope is entered, that variable is created in the current scope, unless it already exists in an outer scope.

Variables in external scopes may be used and changed.

3.3.2 Function Scope

Functions are also defined in a scope and may be nested. This is like a dynamic version of block structure.

Unlike variables, new function definitions, can only occur in the current scope.

Functions in outer scopes may be used, if they are not defined in the current scope, but outer scope functions may not be redefined.

Functions may be redefined within the current scope.

Because scoping is dynamic, function definitions must precede function calls in the program flow.

Because function parameters are created in the new scope, recursion operates as expected. A recursive program is part of our test suite.

3.3.3 Binding Scope

Bindset scope is global.   Bindsets contain bindings that may be changed individually. Dynamic block structured scoping for bindings within bindsets, is basically only understandable to computers, we elected to go with global scoping for bindsets.

3.4 Types and Assignments

Identifiers are names that can refer to objects of different types: variables, bindings, bind associations (placeholders and replacements), and user-defined functions. A special character is placed in front of the name to define the type to which an identifier refers. A variable identifier is prefixed by a `$`. A binding identifier is prefixed by nothing. A user-defined function identifier is prefixed by a `#`.

Objects of different types can share the same name, i.e. a program can use a variable `$foo`, a binding `foo`, and a function `#foo`. Identifiers that refer to bind associations have no prefix.

Declarations are not necessary, since the prefixes prevent confusion between types. Variables can be created as needed.

3.4.1 Variables

A variable refers to a string. It acquires a value by use of the assignment operator.
A variable can take the value of a string constant, the value of another variable, or the return value of a user-defined function.

Variable Assignment → '`$`' *Identifier* '`=`'   *Rstring*
Rstring → (*StringConstant* | *Variable* | *Function* | *Operator Expression*)

3.4.2 Bindings

A binding refers to a list of pairwise associations between template placeholders and placeholder replacements. Placeholders refer to strings within a template. Replacements refer to names of user-supplied text/image/HTML/CGI files. A binding acquires a value by use of the `bind` keyword. A binding is defined as follows:

Binding → '`bind`' *BindName* '`{`' (*BindLine*)* '`}`'
BindName → *Identifier*

BindLine → *Placeholder* `':'` *Replacement {: modifier}* `';'`
Placeholder → *Identifier*
Replacement → *String Expression*
Modifier → *Identifier*

3.4.3 Special Variables

SCL has a number of special variables designed to make web page generation easy.

`$LastPageLink` contains the URL of the last page written.

The following variables can be changed by the user.
`$ReadPath` contains the path to read files from.
`$WritePath` contains the path to write files to.
`$LinkPath` contains a path prepended to all links written by the system.

The above are defined in the outermost scope.

By setting `$LinkPath` at the beginning of a program, not changing it, and by prepending to any local manually generated links, exporting it to any incorporated dynamic scripts etc.: ***A programmer can make a website that can be moved from server to server or up and down directory tree, with a single command re-execution of the SCL code.***

`$Return` is returned as the function value when a function finishes. If `$Return` is not set, the function will return "null."

3.5 Statements

Statements are either atomic (one line), or a set of atomic statements enclosed in braces:

Statement → *AtomicStatement* | `'{'` (*AtomicStatement*)* `'}'`

There are five types of atomic statements:

3.5.1 Assignments

The syntax for assignments was discussed in section 3.4.

3.5.2 Control statements

There are three types of control statements: `if-then-else, while,` and `foreach.`

IfStmt → `'if ('`*Rstring*`') then'` *Statement* (`'else'`*Statement*)? `';'`
WhileStmt → `'while ('` *Rstring* `')'` *Statement* `';'`
ForEachStmt → `'foreach'` *Rstring* `'"'` *Rlist* `'"'` *Statement* `';'`
Rlist → (*Rstring*) // elements are delimited with spaces.
Rstring → (*StringConstant* | *Variable* | *Function*)

If an *Rstring* returns "null" it is false, otherwise true for control purposes.

3.5.3 Function definitions

FunctionDef → `'function #'`*Identifier* `'('`*VarList* `')'` *Statement* `';'`
VarList → (*Variable*)

The statement is typically a set of braced atomic statements. If there is no assignment to $Return, then the function implicitly returns "null".

3.5.4 Function calls

FunctionCall → `'#'`*Identifier* `'('`*VarList* `');'`

All variables are passed by value.

Built-in SCL functions, such as `read, write, makepage,` and `writepage`, are not prefixed with a #.

3.5.5 Binding assignments

The syntax for binding assignments was discussed in section 3.4.

3.6 Primitive Language Functions and Procedures.

Some SCL primitives are designated procedures, and will not return a value. The SCL parser will not allow these to be used in a function context. Functions may however be used in a procedure context.

3.6.1 Procedures

write (string content, string filename)
Writes content to filename.

format_index (string filename, string delimiter)
Puts delimiter between lines in a file

unbind (ident bindset, string bindkey)
Removes bindkey from bindset.

split (string content, string key, string filename)
Creates two files, filname.top and filename.bot which contain content before and after key.

echo (string)
Writes string to the console

erase (string filename)
Erases filename

link (string URL, string label, string filename)
Appends a link to URL with label to filename.

writepage (string template, ident bindset, string filename)
Processes template and bindset and writes the result to filename.

3.6.2 Functions

makepage (string template, ident bindset)
Processes template and bindset and returns the result.

read (string filename)
Reads filname and returns content in a string.

trim (string filename)
Trims path and extension from filename.

system (string command)
Executes commnad in a shell and returns the result.

Chapter 4

Project Plan

4.1 Team Responsibilities

Tasks were ultimately divided among team members as follows:

Clark Landis: Back end and tree walker
Mohamed Nasser: Lexer, parser, and testing
Sudip Das: Documentation

4.2 Project Timeline

The following deadlines were set for key project development goals:

2-18-2003: White paper complete, core language features defined
3-14-2003: Development environment and code conventions defined
3-21-2003: Grammar complete
3-27-2003: Language reference manual complete
3-31-2003: Lexer and Parser complete
4-14-2003: Tree Walker complete
5-05-2003: Back End complete
5-13-2003: Project complete

4.3 Software Development Environment

The project was initially developed on newcunix using Java 1.4 and Antlr 2.7.2. RCS was used for source code version control. The source code was later moved to a CVS repository to allow each team member to use less restrictive off-campus computing environments (Linux, Mac OS X).

4.4 Programming Style Guide

Programmers were encouraged to use common coding standards. Here is a list of some of the standards that were practiced:

4.4.1 Formatting

- All indents are made using tabs.
- Matching braces are always in the same column as their construct.
- All if, while, and for statements use braces even if they control just one statement.
- Use spacing in such a way to make assignments and comparisons clear.
- Use parentheses to make arithmetic expressions and comparisons clear.
- Separate each method by a blank line.

4.4.2 Identifiers

- All identifiers use letters and numbers only.
- All class identifiers are mixed case, with the first letter capitalized, then the first letter of each word in the name capitalized.
- All methods and variables are mixed case, with the first letter lowercase, then the first letter of each subsequent word in the name capitalized
- Methods that set object state begin with the word "set"
- Methods that retrieve object state begin with the word "get"

4.4.3 Comments

- Use /** */ for Javadoc comments at head of each method
- Use // to comment out blocks of code, or for single-line comments
- Don't use /* */ for comments

4.5 Project Log

The following lists actual dates of significant project milestones:

1-21-2003: Project initiated
2-07-2003: Selection of SCL as language to implement
2-15-2003: Key features of SCL defined
2-18-2003: White paper submitted
3-12-2003: Development environment (RCS on newcunix) configured and online
3-21-2003: Initial back end implementation
3-24-2003: Initial lexer and parser implementation
3-27-2003: Language reference manual submitted
4-12-2003: Initial tree walker implementation, initial test program
4-25-2003: Development environment reconfigured (CVS)
4-26-2003: Loops, Smart Binding implemented
4-28-2003: Installer implemented
5-08-2003: Unix integration implemented
5-10-2003: User-defined functions implemented
5-11-2003: Math, conditionals implemented
5-12-2003: CGI wrapper helper files implemented

Chapter 5

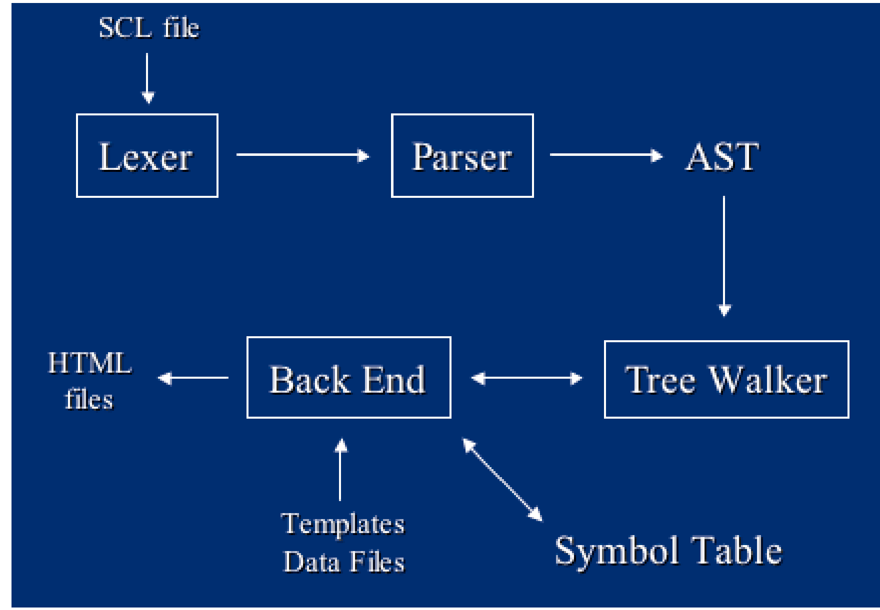
Architectural Design

5.1 Architecture

The SCL interpreter consists of a lexer, a parser / AST generator, an AST walker, and a back end. The lexer, parser, and tree walker were generated by ANTLR, a parser generator for Java. The back end was implemented in Java, as well as the interpreter controller. The resulting Java classes are:

```
SCL.class
SCLLexer.class
SCLParser.class
SCLParserTokenTypes.class
SCLTreeWalker.class
SCLBE.class
```

The relationship between these components are depicted in the following figure:



The input to the interpreter is an SCL file. The file is tokenized by the SCLLexer, and then is parsed by the SCLParser. The SCLParser then generates an abstract syntax tree. An SCLTreeWalker is created to walk the AST. The tree walker contains a reference to an SCLBE (back end) object. Each SCLBE object contains its own set of four symbol tables: one for bindings, one for variables, one for functions, and one for function parameters. When the tree walker finds a particular node in the AST, the corresponding method in the back end is called.

Multiple SCLTreeWalker and SCLBE objects are created when processing loops and functions. When entering a loop, a new Tree Walker is created with a reference to the same Back End as the parent. Thus the inside of a loop uses the same symbol tables as its parent. When entering a function call, a TreeWalker is created with a reference to a child Back End. Inside the function, if a symbol is not found in the child's symbol table, it looks to the parent's symbol table.

No code is generated by the SCL compiler. Rather, the output is a collection of HTML files that are created by the use of the `write` function in the SCL file.

The SCLLexer and SCLParser classes were originally implemented by Mohamed, then were updated by Clark. All other classes were implemented by Clark.

5.2 Error Checking

Syntax error checking is performed by the code generated by ANTLR. If an error occurs during parsing of an SCL file, the compiler will return a statement reporting the line number and column number of where the parse error occurred, as well as what token was expected and what was actually read. Here is an example, which occurs when the user leaves out an input to a multi-input function like `link`:

```
line 13:28: expecting COMMA, found ')'
```

Semantic error checking is not required for SCL. One reason for this is due to the design of its syntax, particularly its convention of prefixing variable and function names. Since variables and user-defined functions have $ and # prefixes, respectively, the parser always distinguishes between them and bindings, which have no prefix. Anytime one is expected semantically, it will be discovered syntactically whether it is the correct type. Another reason that semantic error checking is not needed is because there is no such thing as an undefined variable. Unassigned variables and functions return "null" rather than some unknown quantity.

Chapter 6

Test Plan

6.1 Test Suite

The test suite is composed of six test files. Every time an update was made in the source code, these six tests were run and compared against a benchmark. These six tests were quite successful in determining whether a source code update caused any new bugs. In the future, more tests will be added, and the testing process will be automated.

6.1.1 test1.scl

This program tests if the special path variables work properly with link, read, and write.

```
/*
   This is a test
   of a multiline
   comment
*/

system("rm -rf pages");                  // Remove old pages directory!
system("mkdir pages");                   // Make a new one.

$LinkPath="/";                           // Set $LinkPath for
installation at server root.
$ReadPath="source/";                   // Read from source/
$WritePath="pages/";                     // Write to pages/

$mytemplate=read("template.html");

bind mybindings {
   PutHeaderHere : "header.html";
   PutNavHere :    "pages/nav.html" : SSI ;
   PutVNavHere :    "pages/vnav.html" : SSI ;
}

erase("nav.html");
foreach $file in `ls data`
{
    echo("file: ".$file);
    bind mybindings {  PutBodyHere   : "data/".$file : SSI ; }
```

```
    writepage($mytemplate,mybindings,$file.".shtml");
    link ($LastPageLink,$file,"nav.html");
}
writepage($mytemplate,mybindings,"../index.shtml");
system("cp pages/nav.html pages/vnav.html");
format_index("vnav.html","<br><br>");
```

Here is the expected output of the program:

```
file: bath.jpg
file: halloween.jpg
file: intro.html
file: messy.jpg
file: todo.html
```

6.1.2 test2.scl

This program tests functions and scope of a variable with respect to a function parameter.

```
/*
   This is a test
   of a multiline
   comment

   I this is the first code with a function call.

*/

function #sayHello ( $who ) {           //Function Definition
   echo ("Hello ".$who."!");           //Prints Parameter
   echo ("Hello ".$dog."!");          // Prints value from outer scope!
   $Return="Goodbye for now...";         //No early return.
}                             // To return a value assign it to $Return

$dog="OuterScopeDog";
$who="OuterScopeWho";

$what = #sayHello("dog");                //Function Call
echo($what);
echo ("Hello ".$who."!");              //Prints outer Scope values
echo ("Hello ".$dog."!");

echo (-5*(2+-3));                        // Prints 11
$dog = 10;
$dog = $dog + 1;
echo ($dog);                          // Prints 11
```

Here is the expected output of the program:

```
Hello dog!
Hello OuterScopeDog!
```

```
Goodbye for now...
Hello dog!
Hello OuterScopeDog!
5
11
```

6.1.3 test3.scl

This program tests nested functions and scoping of variables.

```
/*
   This is a nested scope test program
*/

function #dog ( $name ) {            //Function Definition

   function #foot ( $name ) {

       function #toe ( $name ) {
            $Return = $name." is a claw.  ".$name." is smelly.";
            $x="Hello from toe!";
       }

       $Return = $name." is a paw.  ".#toe($name."'s toe");
   }

   $Return=$name." is a Dog. ".#foot($name."'s foot");
}

echo(#dog("Neutron"));

echo("Here x has no value, because it's only defined in the inner
Scope: ".$x);
$x="Hello from OuterScope";
echo("Now X has a value in the Outer Scope, because I just set it:
".$x);
echo(#dog("Electra"));
echo("After calling the function again, x changes, because now it's
defined in the outer scope: ".$x);


function #foot ($name){
    $Return="Foot of ".$name;
}

echo("This is #foot from the outer scope:  ".#foot("Clark"));
```

Here is the expected output of the program:

```
Neutron is a Dog. Neutron's foot is a paw.  Neutron's foot's toe is a
claw.  Neutron's foot's toe is smelly.
Here x has no value, because it's only defined in the inner Scope: null
```

```
Now X has a value in the Outer Scope, because I just set it: Hello from
OuterScope
Electra is a Dog. Electra's foot is a paw.  Electra's foot's toe is a
claw.  Electra's foot's toe is smelly.
After calling the function again, x changes, because now it's defined
in the outer scope: Hello from toe!
This is #foot from the outer scope:  Foot of Clark
```

6.1.4 test4.scl

This program tests if-then-else.

```
/*
Conditional Test Program
*/

echo ( "rat"=="cat");
echo ( "rat"!="cat");

foreach $animal in "rat cat bat dog"
{
   echo ("This animal is a ".$animal);
   if $animal=="cat"
       echo ("    I like cats!");
   else
       echo ("    Oh well... ");
}
```

Here is the expected output of the program:

```
false
true
This animal is a rat
    Oh well...
This animal is a cat
    I like cats!
This animal is a bat
    Oh well...
This animal is a dog
    Oh well...
```

6.1.5 test5.scl

This program tests recursion.

```
/*
   Recursive Factorial Program!
*/
```

```
function #fact ( $num )             //Function Definition
{   if $num==1
        $Return=1;
   else
        $Return=$num*#fact($num-1);
 }

echo("5! = ".#fact(5));


/*
   While Loop!  Counts 1-4.
*/

$x=1;

while $x!=5
{
    echo($x);
    $x=$x+1;
}
```

Here is the expected output of the program:

```
5! = 120
```

6.1.6 test6.scl

This program tests the while statement.

```
/*
   While Loop!  Counts 1-4.
*/

$x=1;

while $x!=5
{
    echo($x);
    $x=$x+1;
}
```

Here is the expected output of the program:

```
1
2
3
4
```

Chapter 7

Lessons Learned

Mohamed reports that one important lesson he learned from this project was the importance of keeping things simple. A relatively simple grammar made parser and tree generation feasible. Also, keeping the scope of the language focused was one reason why our group was able to have a working compiler at the end.

Clark reports that he learned the importance of consistent communication and follow-up with group members. The lack of good follow-up may have been in a factor in Clark often ending up doing work that was assigned to other members, so that he would not have to wait to continue making progress. By the end, he was doing all of the compiler implementation by himself because he was the one most familiar with the code. This was a situation that should have been avoided.

Sudip reports that the most important lesson he learned is that semester-long group projects are not low-priority tasks. Sudip admits that he should have learned ANTLR sooner so that he could have implemented the tree walker, function calls, loops, and conditionals, as was his original job. He regrets that the SCL team did little if any of the programming together in one place.

Appendix

Code Listing

A.1 SCL.g

```
/**
lexer, parser, and AST grammers for the SCL programming language.

Clark Landis
Sudip Das
Mohamed Nasser
05/13/03

*/


class SCLParser extends Parser;
options {
     buildAST=true;
     k=3;}


file : (statement)* EOF^ ;

statement : ( bind | procedure | assignment | defunc | block | loop |
conditional) ;

block :  LBRACE^ (statement)* RBRACE!;

loop : (FOREACH^ lstring "in"! rstring statement
       |WHILE^ rstring statement
       );

conditional : IF^ rstring statement (options {greedy=true;} :ELSE!
statement)?;

defunc: FUNCTION^ func_id LPAREN! var RPAREN! statement;

procedure : ( WRITE^ LPAREN! rstring COMMA! rstring RPAREN! SEMI!
            | FORMAT_INDEX^ LPAREN! rstring COMMA! rstring RPAREN!
SEMI!
            | UNBIND^ LPAREN! rstring COMMA! rstring RPAREN! SEMI!
            | SPLIT^ LPAREN! rstring COMMA! rstring COMMA! rstring
RPAREN! SEMI!
            | ECHO^ LPAREN! rstring RPAREN! SEMI!
            | ERASE^ LPAREN! rstring RPAREN! SEMI!
```

```
            | LINK^ LPAREN! rstring COMMA! rstring COMMA! rstring
RPAREN! SEMI!
            | WRITEPAGE^ LPAREN! rstring COMMA! bindname COMMA! rstring
RPAREN! SEMI!
            | function SEMI!);

rstring : comp_string ((COMPARE^ | NOTCOMPARE^) comp_string)? ;

comp_string : expr_string (PERIOD^ expr_string)* ;

expr_string : term_string  ((PLUS^ | MINUS^) term_string)* ;

term_string : sign_string ((MULTIPLY^ | DIVIDE^) sign_string)* ;

sign_string : (m:MINUS^ {#m.setType(SIGN_MINUS);})? atom_string;

atom_string : (STRING_CONSTANT | var | function | COMMAND_STRING |
(LPAREN^ rstring RPAREN!));

function :  ( MAKEPAGE^ LPAREN! rstring COMMA! bindname RPAREN!
            | READ^ LPAREN! rstring RPAREN!
            | TRIM^ LPAREN! rstring RPAREN!
            | SYSTEM^ LPAREN! rstring RPAREN!
            | POUND^ IDENT LPAREN! rstring RPAREN!
            );

lstring : ( var );

bindname : IDENT ;

bind : BIND^ bindname LBRACE! (bindline)* RBRACE! ;

bindline : (IDENT^ COLON! rstring (COLON! IDENT)* SEMI!
           );


var : (DOLLAR^ IDENT ) ;

func_id : (POUND^ IDENT ) ;

assignment : ((lstring ASSIGN^ rstring) (SEMI!)) ;



imaginaryTokenDefinitions :
     SIGN_MINUS;


class SCLLexer extends Lexer;
options {
   k=3;
   charVocabulary = '\0'..'\377';
}
tokens {
   BIND="bind";
   ECHO="echo";
   MAKEPAGE="makepage";
   READ="read";
```

```
   WRITE="write";
   ERASE="erase";
   LINK="link";
   FOREACH="foreach";
   SYSTEM="system";
   TRIM="trim";
   WRITEPAGE="writepage";
   FUNCTION="function";
   IF="if";
   ELSE="else";
   WHILE="while";
   SPLIT="split";
   FORMAT_INDEX="format_index";
   UNBIND="unbind";
}



IDENT : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;

WS     :
    (' '
    | '\t'
    | '\r' '\n' { newline(); }
    | '\n'      { newline(); }
    )
    { $setType(Token.SKIP); }
  ;

COMMENT
      :     "/*"
                (options {greedy=false;}:
            (      ('\r' '\n') => 'r' 'n'   {newline();}
            |      '\r'   {newline();}
            |      '\n'   {newline();}
            |      ~( '\r' | '\n' )
            )
                )*
            "*/" {$setType(Token.SKIP);} ;

SLCOMMENT : "//" ( ~('\n' | '\r' ))* {$setType(Token.SKIP);};


SEMI : ';' ;

PERIOD : '.' ;

COMMA : ',' ;

COLON : ':' ;

DOLLAR : '$' ;

POUND : '#' ;

ASSIGN : '=' ;

LPAREN : '(' ;
```

```
RPAREN : ')' ;

LBRACE : '{' ;

RBRACE : '}' ;

PLUS : '+' ;

MINUS : '-' ;

MULTIPLY : '*' ;

DIVIDE : '/' ;

COMPARE : '=''=' ;

NOTCOMPARE : '!''=';

STRING_CONSTANT : ('"'!(~('"')|('"'!'"'))*'"'!)        //Anything but
quotes.
                  |(('0'..'9')('0'..'9')*);       //Or an Integer.

COMMAND_STRING :  '`'!(~('`')|('`'!'`'))*'`'!;         //Anything but
back quotes!


class SCLTreeWalker extends TreeParser;
{      SCLTreeWalker(SCLBE BEref){
            BE=BEref;
       }

       String bindset;
       SCLBE BE;
}
tree : #(EOF (statement)*) | statement;

statement : (bind | assignment | procedure | defunc | block | loop |
conditional);

block : #(LBRACE (statement)*);

loop
{String vn=""; String rs="";} :
      ( #(a:FOREACH vn=var rs=rstring ) {BE.forEachLoop(vn,rs,a); }
      | #(b:WHILE c:rstring)  {BE.whileLoop(c,b);}
      );

conditional :
{String rs="";}
     #(a:IF  rs=rstring) {BE.ifStatement(rs,a); };

bind: #(BIND a:IDENT {bindset=a.getText();
                      //System.out.println("TW: bindset:"+bindset);
                     } (bindline)*);

bindline
   {String filename;}: #(d:IDENT filename=rstring (f:IDENT)* )
```

```
                {//System.out.println("TW: BIND
"+bindset+","+d+","+e+", "+f);

BE.setBinding(bindset,d.getText(),filename,(d.getNumberOfChildren()==1)
?"":f.getText());};

defunc :
  {String vname, fname;}
        #(a:FUNCTION fname=func_id vname=var)
{BE.setFunction(fname,vname,a);} ;

assignment
  {String b,vname;}
        : #(ASSIGN vname=var b=rstring) {//System.out.println("TW:
ASSIGN "+a+","+b);
                                BE.setSymbol(vname,b);} ;
var returns [String vname]
  {vname="";}
        : #(DOLLAR a:IDENT) {vname=a.getText();};


func_id returns [String fname]
  {fname="";}
        : #(POUND a:IDENT) {fname=a.getText();};

bindset returns [String bindsetname]
  {bindsetname="";}
        :  a:IDENT {bindsetname=a.getText();};


ident returns [String identname]
   {identname="";}
        :  a:IDENT {identname=a.getText();};


function returns [String ret]
{ String rs,paramValue,funcName; ret="";}
      : #(MAKEPAGE rs=rstring b:IDENT) {
ret=BE.makePage(rs,b.getText()); }
      | #(READ rs=rstring)  {ret=BE.readFile(rs);}
      | #(TRIM rs=rstring)  {ret=BE.trimPath(rs);}
      | #(SYSTEM rs=rstring) {ret=BE.getCommandResult(rs);}
      | #(POUND funcName=ident paramValue=rstring)
{ret=BE.funcCall(funcName,paramValue);};

/** rstring handles all of our string, math, comparison operators.
It also can be constant, variable, or function.
Small portions of Expression evaluation modelled on tutorial by
JS.MILLS */

rstring returns [String ret]
       {String f,v,s1,s2; ret="";}
      : f=function {ret=f;}
        | #(COMPARE s1=rstring s2=rstring) {if (s1.equals(s2))
ret="true"; else ret="false";}
        | #(NOTCOMPARE s1=rstring s2=rstring) {if (s1.equals(s2))
ret="false"; else ret="true";}
```

```
        | #(PERIOD s1=rstring s2=rstring) {ret=s1+s2;}               //
Period is the string Concat operator.
        | #(PLUS s1=rstring s2=rstring)
{ret=String.valueOf(BE.toInt(s1)+BE.toInt(s2));}
        | #(MULTIPLY s1=rstring s2=rstring)
{ret=String.valueOf(BE.toInt(s1)*BE.toInt(s2));}
        | #(DIVIDE s1=rstring s2=rstring)
{ret=String.valueOf(BE.toInt(s1) %  BE.toInt(s2));}
        | #(MINUS s1=rstring s2=rstring)
{ret=String.valueOf(BE.toInt(s1)-BE.toInt(s2));}
        | #(SIGN_MINUS s1=rstring ) {ret=String.valueOf((-
1)*BE.toInt(s1));}
        | #(LPAREN s1=rstring) {ret=s1;}
        | a:STRING_CONSTANT {ret=a.getText();}
        | b:COMMAND_STRING {ret=BE.getCommandResult(b.getText());}
        | #(DOLLAR c:IDENT) {ret=BE.getSymbol(c.getText());};

procedure
{String rs1,rs2,rs3;}
      : #(WRITE rs1=rstring rs2=rstring) {BE.printFile(rs1,rs2);}
      | #(FORMAT_INDEX rs1=rstring rs2=rstring)
{BE.formatIndex(rs1,rs2);}
      | #(UNBIND rs1=rstring rs2=rstring) {BE.unBind(rs1,rs2);}
      | #(SPLIT rs1=rstring rs2=rstring rs3=rstring)
{BE.split(rs1,rs2,rs3);}
      | #(LINK rs1=rstring rs2=rstring rs3=rstring)
{BE.addLink(rs1,rs2,rs3);}
      | #(WRITEPAGE rs1=rstring rs2=bindset rs3=rstring)
{BE.printFile(BE.makePage(rs1,rs2),rs3);}
      | #(ERASE rs1=rstring) {BE.printFile("",rs1);}
 //   | #(SYSTEM rs1=rstring)
{BE.setSymbol("SystemOutput",BE.getCommandResult(rs1));}
        | #(ECHO rs1=rstring) {System.out.println(rs1);}
        | function ;
```

A.2 SCLBE.java

```
/**
  SCLBE is a back end end object for the SCL programming language.
  -Clark Landis 05/13/03
*/

import java.util.*;
import java.io.*;
import antlr.CommonAST;             //We'll be making new tree walkers
from the backend, so we need the Antlr stuff.
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class SCLBE
{
```

```
  /**
  symbols is the variables table, it is a hash table for holding String
Variable Contents.
  The AST node for the start of each function body is stored in
functions.
  Parameters holds each functions parameters under the same key.
  Bindings holds sets of bindings.
  Set and get accessor functions are provided which handle casting etc.
 */
  HashMap symbols;
  HashMap functions;
  HashMap parameters;
  HashMap bindings;
  SCLBE mother;


  /** SCL Constructor just gets a new symbol table, and initializes
special variables.
      Note that the bindings table is global, but each new backend gets
a new symbol and function table.
      Special variables are only defined in the outermost scope, except
of course, for Return.
  */
  SCLBE (SCLBE motherRef)
  {
    symbols = new HashMap ();
    functions = new HashMap ();
    parameters = new HashMap ();
    setSymbol("Return","");
    if (motherRef==null) {
       bindings = new HashMap ();
       setSymbol("LinkPath","");
       setSymbol("ReadPath","");
       setSymbol("WritePath","");
       setSymbol("LastPageLink","");
    }
    else {bindings=motherRef.bindings;}
    mother=motherRef;
  }


  /** Converts a string to an int */
  public int toInt (String s){
  return (Integer.valueOf(s)).intValue();
 }




  /** Reads a file returns the result as a string*/
  public String readFile (String filename)
  {
        //System.out.print("Reading: "+filename+"\n\n\n");
        String s, nl;
          filename=getSymbol("ReadPath")+filename;
        try{
```

```
                BufferedReader infile = new BufferedReader (new
FileReader (filename));
                s = "";
                while ((nl = infile.readLine ()) != null)
                {
                      s = s + nl + "\n";
                }
                infile.close ();
          }
          catch (IOException ioe) {
                System.out.println ("Can't read " + filename);
                s = "Empty File";
          }
          return s;
    }


    /** not yet implimented*/
    public String trimPath(String fullPath){
          return fullPath;
    }




    /**Appends a link to filename.
    */
    public void addLink (String url, String label, String filename)
    {
            filename=getSymbol("WritePath")+filename;
          try
          {
                PrintWriter outfile =
                      new PrintWriter (new BufferedWriter (new FileWriter
(filename, true)));
                outfile.println ("<a href=\""+url+"\">"+label+"</a>");
                outfile.close ();
          }
          catch (IOException ioe)
          {
                System.out.println ("Can't write " + filename);
          }
    }




    /** Formats and Index file with a line delimiting string.*/
    public void formatIndex (String infileName, String delimiter)
    {
          String s, nl;
            String fileName=getSymbol("WritePath")+infileName;
          try{
                BufferedReader infile = new BufferedReader (new
FileReader (fileName));
                s = "";
                while ((nl = infile.readLine ()) != null)
                {
```

```
                        s = s + nl + delimiter + "\n";
                }
                infile.close ();
        }
        catch (IOException ioe) {
                System.out.println ("Can't read " + infileName);
                s = "Empty File";
        }
        printFile(s,infileName);
  }


  /**Writes a string to disk as filename*/
  public void printFile (String content, String filename)
  {
        String s, nl;
          filename=getSymbol("WritePath")+filename;
        try
        {
                PrintWriter outfile =
                        new PrintWriter (new BufferedWriter (new FileWriter
(filename)));
                outfile.print (content);
                outfile.close ();

setSymbol("LastPageLink",getSymbol("LinkPath")+filename);
        }
        catch (IOException ioe)
        {
                System.out.println ("Can't write " + filename);
        }
  }


  /**Executes a shell command and returns the result in a string*/
  public String getCommandResult(String Command){
        //System.out.println("Commnad: "+Command);
          String s,nl;
          s = "";
          try {
                BufferedReader infile = new BufferedReader (new
InputStreamReader (
(Runtime.getRuntime()).exec(Command).getInputStream()));
                while ((nl = infile.readLine ()) != null)
                {
                        s = s + nl +"\n";
                }
        }
        catch (IOException ioe) {
                System.out.println ("Error executing command");
        }
        return s;
  }



  /** Puts a string in the symbol table, used for updating variable
contents*/
```

```
  public void setSymbol (String key, String data) {
        //System.out.println("BACKEND: setting symbol <"+key+">  <--
"+data);
        if (outerSymbolOnly(key))
              mother.setSymbol(key,data);
          else
              symbols.put (key, data);
  }


  public boolean outerSymbol(String key){
       if (mother==null) return false;
       if (mother.symbols.containsKey(key)) return true;
       return mother.outerSymbol(key);
  }

  public boolean outerSymbolOnly(String key){
       if (symbols.containsKey(key)) return false;
       return outerSymbol(key);
 }


  /** Gets a string in the symbol table, used for reading variable
contents*/
  public String getSymbol (String key) {
      if (symbols.containsKey(key))
        return (String) symbols.get (key);
      if (mother!=null) return mother.getSymbol(key);
      return "null";
  }


  /** Puts a AST node in the function table, used for updating variable
contents*/
  public void setFunction (String key, String pname, AST defunc)
  {
        AST statement=defunc.getFirstChild();
          statement=statement.getNextSibling();
          statement=statement.getNextSibling();
        functions.put (key, statement);
        parameters.put (key, pname);
  }

  /** Gets a parameter name*/
  public String getParameter (String key)
  {
      if (parameters.containsKey(key))
        return (String) parameters.get (key);
      if (mother!=null) return mother.getParameter(key);
      return "null";
  }



  /** Gets a node from the function table, used for calling functions*/
  public AST getFunction (String key)
  {
      if (functions.containsKey(key))
```

```
        return (AST) functions.get (key);
      if (mother!=null) return mother.getFunction(key);
      return null;
  }



  /** Gets a filename from a bindset*/
  public String getBinding (String bindSet, String key)
  {
        return ((String[])((HashMap) bindings.get
(bindSet)).get(key))[0];
  }

  /** Gets a modifier from a bindset*/
  public String getModifier (String bindSet, String key)
  {
        return ((String[])((HashMap) bindings.get
(bindSet)).get(key))[1];
  }


  /** Returns the text to be inserted associated with a filename in a
binding.
    Right now it just returns the contents of the file.  This routine
will be
    updated to handle Smartlinking of non-text content */
  public String getBoundText (String bindSet, String key)
  {
        String modifier=getModifier(bindSet,key);
        String filename=getBinding(bindSet,key);
        String
filetype=filename.substring(filename.lastIndexOf('.')+1);
        String linkPath=getSymbol("LinkPath");
        //System.out.println("<<"+filename+", "+modifier+",
"+filetype+">>");
        if (filetype.equals("jpeg") || filetype.equals("jpg") ||
filetype.equals("gif") || filetype.equals("png"))
              return "<img src=\""+linkPath+filename+"\">";
        if (filetype.equals("c") || filetype.equals("cc") ||
filetype.equals("cxx") || filetype.equals("java")
            || filetype.equals("c++") || filetype.equals("lisp") ||
filetype.equals("pas") || filetype.equals("cpp"))
              return getCommandResult("code2html "+filename);
        if (filetype.equals("scl") || filetype.equals("h") )
              return getCommandResult("code2html -l c "+filename);
        if (filetype.equals("txt") || filetype.equals("g") )
              return getCommandResult("txt2html "+filename);
              if (modifier.equals("SSI")){
                    return "<!--#include
virtual=\""+linkPath+filename+"\"-->";
              }
              else
                    return readFile (filename);
  }


  /** Sets a a binding in a bindSet */
```

```
  public void setBinding (String bindSet, String key, String data,
String modifier)
  {
        //System.out.println("BACKEND: setting binding: <"+bindSet+",
"+key+">  <-- "+data+", "+modifier);
        String[] dataArray={data,modifier};
        if (bindings.containsKey (bindSet)) {
              ((HashMap) bindings.get (bindSet)).put (key, dataArray);
        }
        else {
              HashMap s = new HashMap ();
              s.put (key, dataArray);
              bindings.put (bindSet, s);
        }
  }



  /** unbinds a binding in a bindSet */
  public void unBind (String bindSet, String key)
  {
        if (bindings.containsKey (bindSet)) {
              ((HashMap) bindings.get (bindSet)).remove(key);
        }
  }




  public String topPart (String page, String key){
        return page.substring(0,page.indexOf(key));
  }

  public String bottomPart (String page, String key){
        return page.substring(page.indexOf(key)+key.length());
  }

  public void split (String page, String key, String filename){
        printFile(topPart(page,key),filename+".top");
        printFile(bottomPart(page,key),filename+".bot");
  }


  /** This is the MakePage function */
  public String makePage (String sTemplate, String bindSet)
  {
        //System.out.println("BACKEND: makePage "+sTemplate+",
"+bindSet);
        String s=sTemplate;
        String top,bot;
          String i,k;
        Iterator keys = ((HashMap) bindings.get (bindSet)).keySet
().iterator ();
        while (keys.hasNext ())
        {
              k=(String)keys.next();
              //System.out.println("s.replaceAll(k:"+k+" bindSet:
"+bindSet);
```

```
                i=getBoundText(bindSet,k);
                top=topPart(s,k);
                    bot=bottomPart(s,k);
                    s=top+i+bot;
                   // s=s.replaceAll(k,i);
          }
          return s;
    }


    /** For each loop*/
    public void forEachLoop(String vName, String valList, AST forEach){
          AST firstChild, statement;
          firstChild=forEach.getFirstChild();
          statement=firstChild.getNextSibling();
          statement=statement.getNextSibling();
          SCLTreeWalker walker;
          //System.out.println("vl: "+valList);
          StringTokenizer st = new StringTokenizer(valList);
          while (st.hasMoreTokens()) {
                setSymbol (vName,st.nextToken());
                try{ walker = new SCLTreeWalker(this);
                      walker.statement(statement);
                }
                catch(Exception e) { System.err.println(e.getMessage());}
          }
    }



    /** If Statement*/
    public void ifStatement(String boolString, AST ifTree){
          SCLTreeWalker walker;
          AST branch;
          int numChildren=ifTree.getNumberOfChildren();
          //System.out.println("BSt: "+boolString+"   #chil:
  "+numChildren);
          branch=ifTree.getFirstChild();
          //System.out.println("1"+branch.getText());
          branch=branch.getNextSibling();
          //System.out.println("2"+branch.getText());
          if (!boolString.equals("true")){
                branch=branch.getNextSibling();
                // System.out.println("3"+branch.getText());
          }
          if (branch!=null) {
                //System.out.println("Walking a statement");
                try{ walker = new SCLTreeWalker(this);
                      walker.statement(branch);
                }
                catch(Exception e) { System.err.println(e.getMessage());}
          }
    }
```

```
  /** While loop*/
  public void whileLoop(AST condition, AST whileTree){
        AST statement=whileTree.getFirstChild();
          statement=statement.getNextSibling();
          SCLTreeWalker walker;
        try{ walker = new SCLTreeWalker(this);
              //System.out.println(condition.getText());
              //System.out.println(walker.rstring(condition));

              while ((walker.rstring(condition)).equals("true")){
                    walker.statement(statement);
              }
        }
        catch(Exception e) { System.err.println(e.getMessage());}
  }



  /** Function Call */
  public String funcCall(String funcName, String paramValue){
        String retVal="";

        SCLBE newBE=new SCLBE(this);                // make a new
BackEnd, with this backend as it's mother.
        newBE.setSymbol(getParameter(funcName),paramValue);  // Set
parameter value in subscope;

        SCLTreeWalker walker;
        try{ walker = new SCLTreeWalker(newBE);
              walker.tree(getFunction(funcName));
        }
        catch(Exception e) { System.err.println(e.getMessage());}
        return newBE.getSymbol("Return");
  }


  public static void main (String[]args)
  {
        SCLBE L = new SCLBE (null);
        L.setSymbol ("mytemplate",L.readFile("template.html"));
        L.setBinding ("mybindings", "x_header", "header.html","null");
        L.setBinding ("mybindings", "x_leftnav",
"leftnav.html","null");
        L.setBinding ("mybindings", "x_body", "body.html","null");
        L.setSymbol("outstring",L.makePage ("mytemplate",
"mybindings"));
        L.printFile("outstring","index.html");
  }


}
```

A.3 SCL.java

```
/**

SCL.java is the launcher program for the SCL language.
It takes opens the file with  filename appearing as it's first
opperand, lexes, parses and walks this file.
It provides the walker with the top level backend object.


*/


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


public class SCL {
   public static void main(String args[]) {
      if(args.length==0) { error(); }

      FileInputStream fileInput = null;
      try {
         fileInput = new FileInputStream(args[0]);
      } catch(Exception e) { error(); }

      try {
         DataInputStream input = new DataInputStream(fileInput);

         SCLLexer lexer =   new SCLLexer(input);
         SCLParser parser = new SCLParser(lexer);
         parser.file();

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

         //System.out.println(parseTree.toStringList());
         //ASTFrame frame = new ASTFrame("The tree", parseTree);
         //frame.setVisible(true);

         SCLTreeWalker walker = new SCLTreeWalker(new SCLBE(null));
         walker.tree(parseTree);

      } catch(Exception e) { System.err.println(e.getMessage());}
   }

   private static void error() {
      System.out.println("USAGE:                   ");
      System.out.println("   java SCL scl_file");
      System.exit(0);
   }
}
```