Simple Text Processing Language (STePL)

Final Project Report

Nandan Naik (nan2118)
COMS W4115 – Summer 2010

Table of Contents

Introduction

STePL is a simple language for text processing. Its main features include retrieval of text from files, manipulation of text via regular expression matching and replacement, manipulation of text via string matching and replacement, and persistence of processed text to files.

STePL's features are inspired from AWK and it offers a subset of the capabilities offered by the language. STePL's syntax have been influenced by the Ruby Language's syntax and its simplicity.

Language Tutorial

This section will go into details about a useful STePL program. We will look learn about the structure of a STePL program by looking broadly at what parts there are. Then a line by line analysis of the program will be made.

Many Html Pages have a meta tag in the head section to communicate to search engines what keywords are most relevant to the contents of the page. These keywords can be useful to categorize an HTML page, or categorize a web-site in general (based on the most frequent keywords in all its HTML pages). This program written in STePL fetches three HTML pages from the disk, reads the keywords in the meta tag for the pages, and writes the keywords to a file on the disk.

```
1   stringArray HtmlFileArray 3
2   string OutputFile
3
4   string HtmlLine
5   string MetaKeywords
6
7   string MatchRegExString
8   string MatchString
9   string ReplaceString
10  string ResultString
11
12  int index
13  int counter
14
15  function getMetaKeywords
16          MatchRegExString  = "<meta\ name=\"keywords\"\ content=\".*"
17          ResultString = regexMatch HtmlLine MatchRegExString
18
19          MatchRegExString = "content=\".*"
20          ResultString = regexMatch ResultString MatchRegExString
21
22          MatchString = "content=\""
23          ReplaceString = ""
24          ResultString = strReplace ResultString MatchString ReplaceString
25
26          MatchString = "\">"
```

```
27          ReplaceString = ""
28          ResultString = strReplace ResultString  MatchString ReplaceString
29
30          MetaKeywords = ResultString
31  end
32
33  function main
34          HtmlFileArray[0] = "c:\\HtmlFile1.html"
35          HtmlFileArray[1] = "c:\\HtmlFile2.html"
36          HtmlFileArray[2] = "c:\\HtmlFile3.html"
37
38          OutputFile = "c:\\keywords.txt"
39
40          while index < 3 then
41                  counter = 1
42                  HtmlLine = getLine HtmlFileArray[index] counter
43
44                  while HtmlLine neq "" then
45                          getMetaKeywords
46
47                          if MetaKeywords neq "" then
48                            appendFile OutputFile MetaKeywords
49                            HtmlLine = ""
50                          end
51
52                          counter = counter + 1
53
54                          HtmlLine = getLine HtmlFileArray[index] counter
55                  end
56
57                  index = index + 1
58          end
59  end
```

Executing the Program:

Step 1: Compile the "stepl" executable by typing "stepl-make.bat".

Step 2: Run the tutorial.stpl file listed above by typing "stepl < tutorial.stpl"

Program Structure:

Broadly, the above program has 2 sections:

- A global variable declaration section
- A function declarations section

Variable Declarations

Variables in STePL are global and can be of two 4 types: int, intArray, string and stringArray. They must be declared outside functions. It is considered good style to declare them in the beginning of a STePL program. The above program defines its variables on lines 1 to 14. One interesting point is how arrays are declared – arrays in STePL are declared in the following manner "typeArray ArrayName ArraySize" eith spaces between each word. In accordance with this rule, line 1 declares that we want a stringArray of size 3. The user cannot set the initial value of a variable in the program. By default, all integer variables are initialized to 0s and string variables are initialized to empty strings(“”). This holds good for array variables as well – each element of an array is initialized as described.

Function Declarations
The function declaration section is on lines 15 to 59. All STePL programs must contain a function called “main” where the control for the program starts. In the tutorial above we have defined two functions: one called “main” and another custom function called “getMetaKeywords”.

Function “main”
The main function starts off by initializing the values of the HtmlFileArray to the path of the HTML files to be processed on lines 34 to 36. The path to the file that will store the result of the processing is set on line 38. The while loop spanning lines 40 to 58 iterates over the various files in the HtmlFileArray that are to be processed. Line 42 attempts to bring the first line of text from an HTML file. If there is content to be processed for that file, we called the getMetaKeywords function on line 45 to process it. If meta keywords are found line 47 detects it and line 48 writes the keywords to the output file. Line 49 then sets up the condition to break out of the inner while loop.

Function “getMetaKeywords”
The getMetaKeywords function is used to pull the meta keywords, out of an HtmlLine, if present. Lines 16 and 17 initialize a regular expression that matches the “content” tag in HTML and retrieve the text for the tag. Lines 22 to 28, replace certain strings such as “content=” and the trailing “/” from the matched content in the previous step. We have the keywords available to us by line 30.

The program stops at the end of the “main” function.

Language Reference Manual

1. Token Types and Interpretation

The following types of tokens can be present in a STePL program:

- Identifiers
- Constants (Integer and String)
- Expression Operators
- Keywords

Tokens are interpreted in a "greedy" manner. That is, a valid token is the longest string of characters in the input stream that could be grouped to form the token.

2. Whitespace

In general spaces, tabs and newlines are ignored. A space must be used to separate adjacent Identifiers and Constants.

3. Comments

The "#" character at the beginning of a line indicates that the line is a comment. Anything that follows the "#" character in that line is ignored.

4. Expressions

An expression is defined to be of one of the following forms:

- An Id (Ex. myvar)
- An Integer Literal (Ex. 123)
- A String Literal (Ex. "abc")
- An Array Element indicated via an Integer Index (Ex. MyArray[1])
- An Array Element indicated via an Id to an Integer Variable (Ex. MyArray[index])
- A Binary Expression of the form "Expression operation Expression" (Ex. 1 + 2)
- An assignment to a variable or an array element (Ex. Myvar = 1)
- A call to one of the built-in functions (Ex. print 1+2)
- A call to a custom function (Ex. myfunc)

5. Identifiers

Identifiers can consist of one alphabet followed by zero or more alphabets or integers. The alphabets must all belong to the ASCII character set. Identifiers are case sensitive. Therefore an identifier called “MyVariable” is different from one called “myVariable”.

6. Variable Types

There are four types of variables in STePL:

- Integers
- Integer Arrays
- Strings
- String Arrays

Variables in STePL can be of one of the four types listed above. An Identifier is used to denote the name of the variable. The user cannot set the initial value of the variables. By default, all integer variables are initialized to 0 and all string variables are initialized to the empty string (“”).

Elements of the String and Integer Arrays are accessed via an index starting at 0 until (ArraySize – 1).

The values of Integer and String variables can be accessed via the variable names. For example, if myvar is a variable: “print myvar” will print its value.

The values of Integer and String Arrays can be accessed via the identifier and an index in square brackets. For example, if myvar is an array, “print myvar[0]” prints the value of the first element in the array.

Note: Array elements can only be accessed either via an Identifier and an Integer literal, or via an Identifier and an Integer variable. They cannot be accessed by specifying an Identifier and an Integer expression (Ex. array[1+2]).

Variables in STePL are global and must be defined outside of functions. They can, however, only be assigned values inside functions.

7. String Constants

String constants must consist of characters in the ASCII character set. They must be delimited by double-quotes. Ex. “12345”.

A double-quote at the start of a String constant immediately followed by a double quote to end it signifies an empty string (“”).

Any double-quote characters that are part of a string constant must be escaped using a preceding back-slash character (“\”).

Ex. “The actor said, “\”Hello\”.” Here the word Hello is prefixed and suffixed by a backslash and a double quote (\”).

The following escape sequences are allowed in STePL:

- “\\” for a backslash
- “\’” for a single quote
- “\”” for a double quote
- “\n” for a newline
- “\r” for a carriage return
- “\b” for a backspace
- “\ “ for a space
- “\ddd” for an ascii character where ddd signifies a 3 digit number

STePL does not have the need for a “NULL” value and is thus not supported in the language.

8. Integer Literals

Integer constants can only consist of the following characters: 0,1,2,3,4,5,6,7,8,9.
STePL does not have the need for a “NULL” value and is thus not supported in the language.

9. Keywords

Keywords are case sensitive. The following keywords are available in STePL:

Keyword	**Description**
appendFile	Appends the specified string to the end of the file pointed to by FilePath. Returns 0 on success and a 1 on failure. **Usage:** intReturnValue **= appendFile** FilePath AppendString **Conditions:** FilePath must be a string constant or a string variable. AppendString must be a string constant or a string variable. intReturnValue must be an integer variable.
else	Used in an if-then-else code block. If the condition for the if statement evaluates to a non-zero value, the statements associated with else are executed. **Usage:** **if** SomeInt == 1 **then** # Some STePL Statement(s) **else** # Some STePL Statement(s) **end**

end	Used to denote the end of a function-end, an if-then or if-then-else or a while-then-end block.
	
	Usage:
	
	function myfunction **then**
	 # Some STePL Statement(s)
	end
	
	if 0 **then**
	 # Some STePL Statement(s)
	end
	
	if 0 **then**
	 # Some STePL Statement(s)
	else
	 # Some STePL Statement(s)
	end
	
	while counter < 0 **then**
	 # Some STePL Statement(s)
	end

function	Used to denote the beginning of a named code block. Control can be transferred to this named block of code by using the function name. Functions in STePL do not take any parameters and do not have a return value. All variables have global scope and can thus be accessed inside functions. **Usage:** **function** FunctionName # Some STePL Statement(s) **end** **Conditions:** FunctionName must be an identifier.
getLine	Returns a line of text from the file pointed to by the FilePath at the line number specified by LineNumber. Returns an empty string if there is no content available at the specified LineNumber. **Usage:** StringVariable = getLine FilePath LineNumber **Conditions:** FilePath must be a string variable or a string literal. LineNumber must be an integer variable or integer literal. StringVariable must be a string variable.
int	Used to indicate that an identifier is an integer variable. **Usage: int** MyInteger **Conditions:** MyInteger must be an identifier. Users cannot assign an initial value to variables. Values can only be assigned to variables inside functions. By default all integer variables are set to 0.
intArray	Used to indicate that an indentifier is an integer array. **Usage: intArray** MyIntArray ArraySize **Conditions:** MyIntArray must be an identifier. ArraySize must be an integer literal. **Usage:** MyIntArray [index] = 12345 **Conditions:** index must be an integer literal or an integer variable. Users cannot assign an intial value to elements of Integer Arrays. Values can only be assigned to elements of integer arrays inside functions. By default all elements of integer arrays are set to 0.

if	Used to denote the start of an if-then-else code block. Must be followed by an expression that yields an integer value.
	
	Usage:
	if SomeInt == 1 **then**
	  # Some STePL Statement(s)
	else
	  # Some STePL Statement(s)
	end

main	Used to denote the block of code at which the flow of control is started in a STePL program. **Usage:** **function main** # Some STePL Statement(s) **end** **Conditions:** The main function must always be present in a STePL program.
regexMatch	Returns a string which matches the regular expression in the MatchRegExString. If no match was found, the an empty string is returned. **Usage:** MyString = **regexMatch** Content MatchRegExString **Conditions:** Content must be a string variable or a string literal. MatchRegExString must be a string variable or a string literal. MyString must be a string variable.
regexReplace	Returns a string after replacing in Content any occurrence of the regular expression in MatchRegExString with the corresponding string contained in ReplaceString. If no match was found, the ReplaceString is ignored and Content is returned. **Usage:** MyString = regexReplace Content MatchRegExString ReplaceString **Conditions:** Content must be a string variable or a string literal. MatchRegExString must be a string variable or a string literal. ReplaceString must be a string literal or a string variable. MyString must be a string variable.
string	Used to indicate a string variable. **Usage: string** MyString **Conditions:**

	MyString must be an identifier. Users cannot set the intial values of string variables. Values for string variables can only be assigned inside functions. By default all string variables are set to “”.
stringArray	Used to indicate a string array. **Usage: stringArray** MyStringArray ArraySize **Conditions:** MyStringArray must be an identifier. ArraySize must be an integer literal. **Usage:** MyStringArray [index] = “Value” **Conditions:** index must be an integer literal or an integer variable. Users cannot assign an intial value to elements of String Arrays. They can only be assigned inside functions. By default all elements of string array objects are set to “”.
strReplace	Returns a string after replacing any occurrence in Content of the string in MatchString with the corresponding string in the ReplaceString. If no match was found, the Content is returned. **Usage:** MyString = strReplace Content MatchString ReplaceString **Conditions:** Content must be an a string literal or a string variable. MatchString must be a string literal or a string variable. ReplaceString must be a string literal or a string variable. MyString must be a string variable.

while	Used to denote the start of a while-then-end block. Must be followed by an expression that yields an integer value.
	
	Usage:
	while counter < 0 **then**
	  # Some STePL Statement(s)
	end

then	Used to denote the end of the condition section of a an if-then, if-then-else or a while-then code block. Denotes the start of the section of the if or while block which runsif the condition evaluates to a zero value.
	
	Usage:
	while counter < 0 **then**
	  # Some STePL Statement(s)
	end

10. Operators

All operators in STePL associate from left to right.

Additive Operators

Operator	Usage	Description
+	Expression1 **+** Expression2	Adds two expressions that both yield integer values. Yields an integer value.
-	Expression1 **–** Expression2	Subtracts Expression 2 from Expression1 where both expressions yield integer values. Yields an integer value.

Other Operators

Operator	Usage	Description
[**and** **]**	[integerConstant]	Indicates which element in an Array we are operating on.

Relational Operators

Operator	Usage	Description
==	Expression1 **==** Expression2	Checks if two expressions that both yield integer values are equal. Yields 0 when they are equal and 1 when they are not.
!=	Expression1 **!=** Expression2	Checks if two expressions that both yield integer values are not equal. Yields 0 when they are not equal and 1 when they are.
<	Expression1 **<** Expression2	Yields 0 when Expression1 is less than Expression2 and 1 when it is not so. Both Expression1 and Expression2 must yield integer values.
<=	Expression1 **<=** Expression2	Yields 0 when Expression1 is less than or equal to Expression2 and 1 when it is not so. Both Expression1 and Expression2 must yield integer values.
>	Expression1 **>** Expression2	Yields 0 when Expression1 is greater than Expression2 and 1 when it is not so. Both Expression1 and Expression2 must yield integer values.
>=	Expression1 **>=** Expression2	Yields 0 when Expression1 is greater than or equal to Expression2 and 1 when it is not so. Both Expression1 and Expression2 must yield integer values.
and	Expression1 and Expression2	Yields 0 when both Expression1 and Expression2 are equal to 0. Both Expression1 and Expression2 must yield integer values.
or	Expression1 or Expression2	Yields 0 when either Expression1 or Expression2 is equal to 0. Both Expression1 and Expression2 must yield integer values.
&	Expression1 & Expression2	Yields a string that is the concatenation of Expression1 and Expression2.
eq	Expression1 eq Expression2	Yields 0 if Expression1 is the same string as Expression2 and 1 otherwise.
neq	Expression1 neq Expression2	Yields 0 if Expression1 is not the same string as Expression2 and 1 otherwise.

Assignment Operators

Operator	Usage	Description
=	Variable = Expression	Assigns the value of Expression to a variable. A String Variable can be assigned String Literals, Other String Variables, Integer Literals or I **Conditions:** An assignment to a variable can only occur inside of functions.

11. Storage Classes and Scope

All variables in STePL belong to the global storage class. This entails that they are initiliazed when a STePL program is started and accessible across every function in the program.

There is no support for STePL programs to be split into multiple files. Thus, any variables in a STePL program are by default available in any function in that program.

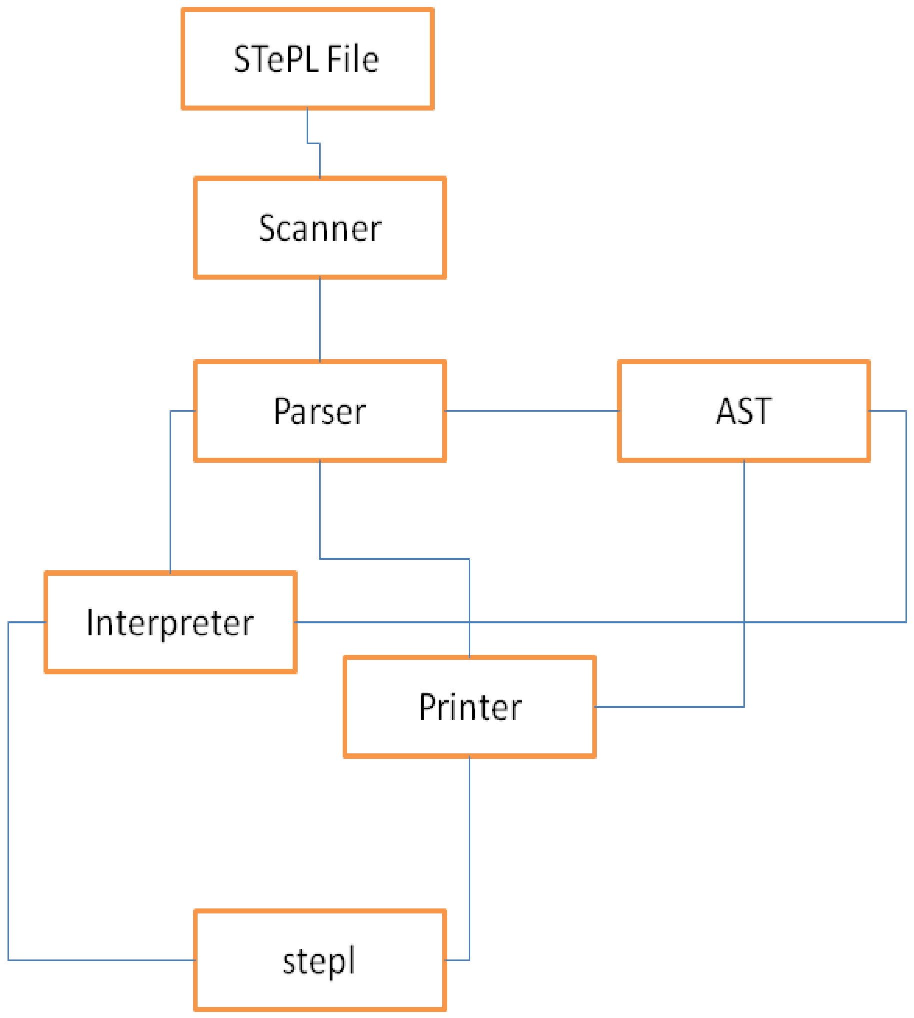
Project Plan

Process used for Planning, Specification, Development and Testing:
An iterative development methodology was used for this project. Simple versions of the Scanner, Parser, AST, Interpreter and Printer were created based on the "microc" sample discussed in class. A Windows Batch file based Test Harness (stepl-test.bat) was created to run the test stepl files located in the "tests" folder. Additional iterations added more features to the language, and made corresponding changes across the files. More tests were added in conjunction with the features developed. Testing was performed at every milestone in the project via the Test Harness.

Project Log:

Event	**Date**
Language Proposal submitted.	June 7th 2010
Preliminary version of Scanner created.	June 20th 2010
Language Reference Manual submitted.	June 25th 2010
Preliminary versions of Parser, AST, Interpreter, Top-level file, Test Harnes and Test files created.	July 10th 2010
All operators added to Scanner and AST. Parser and Interpreter were correspondingly changed. Added more STePL test files and updated Test Harness.	July 15th 2010
All statements added to Scanner and AST. Parser and Interpreter were correspondingly changed. Added more STePL test files and updated Test Harness.	July 26th 2010
All built-in functions added to Scanner and AST. Parser and Interpreter were correspondingly changed. Added more STePL test files and updated Test Harness.	August 3rd 2010
Final Project Report	August 13th 2010

Architectural Design



Step 0: The top level program "stepl" is invoked, which in turn invokes the scanner.

Step 1: The STePL source file is tokenized by the Scanner.

Step 2: The Scanner passes the tokens along to the Parser.

Step 3: The Parser (which is aware of the AST) passes the abstracted information onto either the Interpreter or the Printer (depending on which flag is chosen in the top-level program "stepl").

Step 4: The Interpreter or Printer (both of which are aware of the AST) makes sense of what to do with the information given to it by the parser. The interpreter performs the actions for the language(coded in Ocaml) and the printer simply prints the information it has gotten from the parser.

Step 5: The top-level program "stepl" (which is the one which actually initiates the entire process) returns control to the program which invoked it.

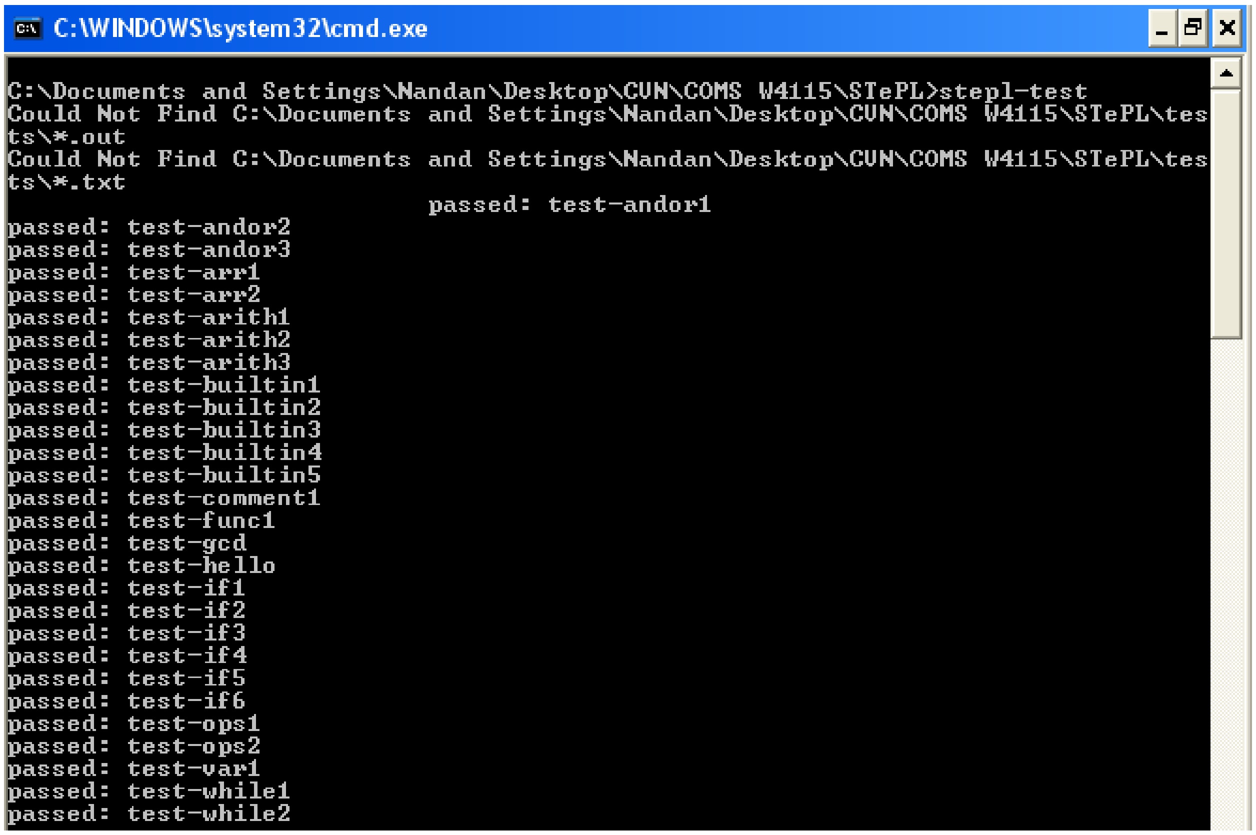
Test Plan

Test Cases (in the “tests” folder):

Case	**File Name**
Tests and / or operators.	test-andor1.stpl
Tests and / or operators where values come from variables.	test-andor2.stpl
Test and /or operators where values come from expressions.	test-andor-3.stpl
Simple arithmetic test (addition)	test-arith1.stpl
More complicated arithmetic test (addition and subtraction)	test-arith2.stpl
Test arithmetic using variables	test-arith3.stpl
Test declaring an int Array and accessing its element.	test-arr1.stpl
Test declaring a string Array and accessing its element.	test-arr2.stpl
Test appendFile built-in function.	test-builtin1.stpl
Test getLine built-in function.	test-builtin2.stpl
Test regexMatch built-in function.	test-builtin3.stpl
Test regexReplace built-in function.	test-builtin4.stpl
Test strReplace built-in function.	test-builtin5.stpl
Test comments.	test-comment.stpl
Test calling a user defined function.	test-func1.stpl
Tests gcd algorithm.	test-gcd.stpl
Tests “hello world” functionality.	test-hello.stpl
Tests a simple if statement with no else.	test-if1.stpl
Tests if statement with else.	test-if2.stpl
Tests expr=1 for if statement with no else.	test-if3.stpl
Tests expr=1 for if statement with else.	test-if4.stpl
Tests if statement with multiple statements inside each block.	test-if5.stpl
Tests nested ifs.	test-if6.stpl
Tests simple while block.	test-while1.stpl
Tests nested while blocks.	Test-while2.stpl

Test Automation:
A Windows Batch file called “stepl-test.bat” was used as a Test Harness to perform automated testing. At each milestone where the scanner, parser, ast and interpreter were changed, test STePL programs were written and the Test Harness was appropriately extended to cover the new features.

The Test Harness can be seen in action by first doing a “stepl-make” to build the stepl executable and then running it via “stepl-test”.



C:\WINDOWS\system32\cmd.exe

```
C:\Documents and Settings\Nandan\Desktop\CVN\COMS W4115\STePL>stepl-test
Could Not Find C:\Documents and Settings\Nandan\Desktop\CVN\COMS W4115\STePL\tes
ts\*.out
Could Not Find C:\Documents and Settings\Nandan\Desktop\CVN\COMS W4115\STePL\tes
ts\*.txt
                            passed: test-andor1
passed: test-andor2
passed: test-andor3
passed: test-arr1
passed: test-arr2
passed: test-arith1
passed: test-arith2
passed: test-arith3
passed: test-builtin1
passed: test-builtin2
passed: test-builtin3
passed: test-builtin4
passed: test-builtin5
passed: test-comment1
passed: test-func1
passed: test-gcd
passed: test-hello
passed: test-if1
passed: test-if2
passed: test-if3
passed: test-if4
passed: test-if5
passed: test-if6
passed: test-ops1
passed: test-ops2
passed: test-var1
passed: test-while1
passed: test-while2
```

Lessons Learned

Domain Specific Languages (DSLs):

- Coming into this class, the notion of being able to develop my own language and an interpreter for it was extremely exciting.The idea of "Domain Specific Languages (DSLs)" is increasingly gaining traction in the industry and my experience in this class has given me a solid foundation for further exploration in this area.
- This class also changed the way I approach problems that are addressable by software. In the past, I have built Object Oriented systems (where objects interacted with each other, based on certain rules) and Procedural systems (where functions called each other, based on certain rules). But, being able to build a language which allows problems in a domain to be addressed, instead of just a few interacting objects or functions, is a paradigm shift for me.

The OCaml Language:

- The OCaml language was quite different from any that I have previously come across. But it has turned out to be pleasantly different.
- I was extremely impressed with how much support the OCaml compiler provides users to cover all possible use cases while pattern matching. This support was invaluable when the AST was being changed during iterative development cycles. Any additional type being added to the AST immediately caused warnings to fix all related areas which were being impacted. This made it extremely easy for me to track down which parts I had to provide code for. In other languages, I would have ended up tracking the impact across different parts of the system manually.

OCamlLex:

- I found the support in OCamlLex to match regular expressions to be extremely powerful. I had initially defined "identifiers" in my LRM as consisting only of alphabets. This was based on the expectation that it would be difficult to match an identifier starting with an alphabet, and containing zero or more alphabets and strings. But, OcamlLex showed me how such concerns do not amount to much, and that it is extremely easy to match tokens using regular expressions.

Complete Code Listing

scanner.mll

```
{
        open Parser;;
  let buffer = Buffer.create 16;;
        let char_for_decimal_code str = str.[0] <- '0'; Char.chr (int_of_string str land 0xFF);;
}

let alpha = ['a'-'z' 'A'-'Z']
let newline = '\n' | "\r\n"
let numeric = ['0'-'9']
let other = ['`' '~' '-' '_' '=' '+'
     '[' '{' ']' '}' '|'
                                        ';' ':'
                                        ',' '<' '.' '>' '/' '?']
let shift_numeric = ['!' '@' '#' '$' '%' '^' '&' '*' '(' ')']

let alphanumeric = alpha | numeric
let string = alpha | numeric | other | shift_numeric

rule token = parse

| '#'            { comment lexbuf }

(* --------- *)
(* operators *)
| '+'                  { PLUS }
| '-'                                      { MINUS }
| '[' (numeric+ as index) ']' { INDEX(int_of_string index) }
| '[' ((alpha+ alphanumeric*) as lxm) ']' { ID_INDEX(lxm) }
| "=="                                     { EQUALS }
| "!="           { NEQ }
| '<'            { LT }
| "<="             { LEQ }
| '>'            { GT }
| ">="                               { GEQ }
| "and"                { AND }
| "or"                      { OR }
| '='                      { ASSIGN }
| '&'                      { AMP }
| "eq"                { EQ }
| "neq"                    { NEQ }

| "main"           { MAIN("main") }

(* ------------------ *)
(* built-in functions *)
| "appendFile"                              { APPEND_FILE }
| "getLine"                                 { GET_LINE }
| "print"          { PRINT }
| "regexMatch"        { REGEX_MATCH }
| "regexReplace"              { REGEX_REP }
| "strReplace"                { STR_REP }

(* ---------- *)
(* statements *)
| "else"                                          { ELSE }
| "end"                                           { END }
| "function"                                  { FUNCTION }
| "if"                        { IF }
| "then"                       { THEN }
| "while"                      { WHILE }

(* --------- *)
(* variables *)
| "int"                        { INT }
| "intArray"                   { INTARRAY }
| "string"                     { STRING }
| "stringArray"                  { STRINGARRAY }

(* -------- *)
(* literals *)
| numeric+ as lxm                  { INT_LITERAL(int_of_string lxm) }
| '"'                           { Buffer.clear buffer; STRING_LITERAL(string_literal lexbuf) }
| (alpha+ alphanumeric*) as lxm       { ID(lxm) }

| eof                                             { EOF }

| _                          { token lexbuf }

(* ------- *)
(* string literal *)
and string_literal = parse
| '"'                   { Buffer.contents buffer }
| "\\\\"                 { Buffer.add_char buffer '\\'; string_literal lexbuf }
| "\\\""                 { Buffer.add_char buffer '\"'; string_literal lexbuf }
| "\\'"                 { Buffer.add_char buffer '\''; string_literal lexbuf }
| "\\n"                 { Buffer.add_char buffer '\n'; string_literal lexbuf }
| "\\r"                 { Buffer.add_char buffer '\r'; string_literal lexbuf }
| "\\t"                 { Buffer.add_char buffer '\t'; string_literal lexbuf }
| "\\b"                 { Buffer.add_char buffer '\b'; string_literal lexbuf }
| "\\ "                 { Buffer.add_char buffer '\ '; string_literal lexbuf }
| "\\" ['0'-'9'] ['0'-'9'] ['0'-'9']
                    { Buffer.add_char buffer (char_for_decimal_code (Lexing.lexeme lexbuf));
                                                                         string_literal lexbuf }
| string as lxm           { Buffer.add_char buffer lxm; string_literal lexbuf }

(* ------- *)
(* comment *)
and comment = parse
| newline                     { token lexbuf }
| _                          { comment lexbuf }
```

parser.mly

```
%{ open Ast %}

%token PLUS MINUS
%token EQUALS NOTEQUALS LT LEQ GT GEQ AND OR ASSIGN AMP EQ NEQ
%token APPEND_FILE GET_LINE REGEX_MATCH PRINT REGEX_REP STR_REP
%token ELSE END FUNCTION IF THEN WHILE
%token INT INTARRAY STRING STRINGARRAY
%token EOF

%token <string> MAIN
%token <string> ID
%token <int> INDEX
%token <string> ID_INDEX
%token <int> INT_LITERAL
%token <string> STRING_LITERAL

%nonassoc ELSE
%nonassoc INDEX
%nonassoc ID_INDEX
%nonassoc NOELSE
%nonassoc APPEND_FILE GET_LINE PRINT REGEX_MATCH REGEX_REP STR_REP

%left ASSIGN
%left EQUALS NOTEQUALS EQ NEQ
%left LT LEQ GT GEQ AND OR
%left PLUS MINUS AMP

%start program
%type <Ast.program> program

%%

program:
        | /* nothing */ { ([], []) }
        | program vdecl { ($2 :: fst $1), snd $1 }
        | program fdecl { fst $1, ($2 :: snd $1) }

vdecl:
        | INT ID                 { { vname = $2; vtype = Ast.Int; vsize = 1; } }
        | INTARRAY ID INT_LITERAL    { { vname = $2; vtype = Ast.IntArray; vsize = $3; } }
        | STRING ID                { { vname = $2; vtype = Ast.String; vsize = 1; } }
        | STRINGARRAY ID INT_LITERAL { { vname = $2; vtype = Ast.StringArray; vsize = $3; } }

fdecl:
        | FUNCTION ID stmt_list END   { {fname = $2; fbody = List.rev $3} }
        | FUNCTION MAIN stmt_list END { {fname = $2; fbody = List.rev $3} }

stmt_list:
        | /* nothing */  { [] }
        | stmt_list stmt { $2::$1 }

stmt:
        | expr                              { Expr($1) }
```

```
            | IF expr THEN stmt_list END %prec NOELSE   { If($2, $4, []) }
            | IF expr THEN stmt_list ELSE stmt_list END { If($2, $4, $6) }
            | WHILE expr THEN stmt_list END             { While($2, $4) }

expr:
            | INT_LITERAL         { IntLiteral($1) }
            | STRING_LITERAL      { StringLiteral($1) }
            | ID                  { Id($1) }
            | ID INDEX            { ArrayElement($1, $2) }
            | ID ID_INDEX         { ArrayElementViaId($1, $2) }
  | ID ASSIGN expr      { Assign ($1, $3) }
            | ID INDEX ASSIGN expr { AssignArrayElement ($1, $2, $4) }

            | expr PLUS expr      { Binop($1, Add, $3) }
            | expr MINUS expr     { Binop($1, Sub, $3) }
            | expr EQUALS expr    { Binop($1, Equals, $3) }
            | expr NOTEQUALS expr { Binop($1, NotEquals, $3) }
            | expr LT expr        { Binop($1, Lt, $3) }
            | expr LEQ expr       { Binop($1, Lteq, $3) }
            | expr GT expr        { Binop($1, Gt, $3) }
            | expr GEQ expr       { Binop($1, Gteq, $3) }
            | expr AND expr       { Binop($1, And, $3) }
            | expr OR expr        { Binop($1, Or, $3) }
            | expr AMP expr       { Binop($1, Amp, $3) }
            | expr EQ expr        { Binop($1, Eq, $3) }
            | expr NEQ expr       { Binop($1, Neq, $3) }

            | APPEND_FILE expr expr    { AppendFile($2, $3) }
            | GET_LINE expr expr       { GetLine($2, $3) }
            | PRINT expr               { Print($2) }
            | REGEX_MATCH expr expr    { RegexMatch($2, $3) }
            | REGEX_REP expr expr expr { RegexReplace($2, $3, $4) }
            | STR_REP expr expr expr   { StrReplace($2, $3, $4) }
```

ast.mli

```
type op = Add | Sub | Equals | NotEquals | Lt | Lteq | Gt | Gteq | And | Or | Amp | Eq | Neq


type expr =
        | IntLiteral of int
        | StringLiteral of string
        | Id of string
        | ArrayElement of string * int
        | ArrayElementViaId of string * string
        | Binop of expr * op * expr
        | Assign of string * expr
        | AssignArrayElement of string * int * expr
        | AppendFile of expr * expr
        | GetLine of expr * expr
        | Print of expr
        | RegexMatch of expr * expr
        | RegexReplace of expr * expr * expr
        | StrReplace of expr * expr * expr
        | Call of string


type stmt =
        | Block of stmt list
        | Expr of expr
        | If of expr * stmt list * stmt list
        | While of expr * stmt list


type var_type = Int | IntArray | String | StringArray


type var_decl = {
        vname: string; (* variable name *)
        vtype: var_type; (* variable type *)
        vsize: int; (* variable size *)
        }


type func_decl = {
        fname: string;  (* function name *)
        fbody: stmt list (* list of stmts in function body *)
        }


type program = var_decl list * func_decl list (* global vars, funcs  *)
```

interpreter.ml

```
open Ast

(* ==================================================== *)
module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

(* ==================================================== *)
exception ReturnException of string * (int array NameMap.t * string array NameMap.t * func_decl
NameMap.t)


let string_of_char = String.make 1;;

(* ============================================================ *)
(* Evaluate an expression and return (value, updated environment) *)
let rec eval env = function
            | IntLiteral(i) -> string_of_int(i), env
            | StringLiteral(s) -> s, env
            (* ------------------------------------------------ *)
  | Id(var) ->
                              let (int_var_decls, string_var_decls, func_decls) = env
                              in
                              if NameMap.mem var int_var_decls then
                                      string_of_int((NameMap.find var int_var_decls).(0)), env

                              else if NameMap.mem var string_var_decls then
                                ((NameMap.find var string_var_decls).(0)), env
                              else if NameMap.mem var func_decls then
                                      let fdecl = (NameMap.find var func_decls)
                                      in
                                      "0", call fdecl env
                              else
                                      raise (Failure ("undeclared identifier " ^ var))
            (* ------------------------------------------------ *)
            | ArrayElement(s, i) ->
                              let (int_var_decls, string_var_decls, func_decls) = env
                              in
                              if NameMap.mem s int_var_decls then
                                      string_of_int((NameMap.find s int_var_decls).(i)), env

                              else if NameMap.mem s string_var_decls then
                                ((NameMap.find s string_var_decls).(i)), env
                              else
                        raise (Failure ("undeclared identifier " ^ s))
            (* ------------------------------------------------ *)
            | ArrayElementViaId(s1, s2) ->
                              let (int_var_decls, string_var_decls, func_decls) = env
                              in
                              let i =
                                      if NameMap.mem s2 int_var_decls then
                                              (NameMap.find s2 int_var_decls).(0)
```

```
                                                else
                                                        raise (Failure ("undeclared identifier " ^ s2))
                                        in
                                        if NameMap.mem s1 int_var_decls then
                                                string_of_int((NameMap.find s1 int_var_decls).(i)), env

                                        else if NameMap.mem s1 string_var_decls then
                                          ((NameMap.find s1 string_var_decls).(i)), env
                                        else
                                  raise (Failure ("undeclared identifier " ^ s1))
                (* ------------------------------------------------- *)
  | Binop(e1, op, e2) ->
                        let v1, env = eval env e1
                                        in
                let v2, env = eval env e2
                                        in
                        let boolean i = if i then 0 else 1
                                        in
                        (match op with
                 |      Add -> string_of_int( int_of_string(v1) + int_of_string(v2) )
                                         | Sub -> string_of_int( int_of_string(v1) - int_of_string(v2) )

                                         | Equals -> string_of_int( boolean (int_of_string(v1) = int_of_string(v2)) )
                                         | NotEquals -> string_of_int( boolean (int_of_string(v1) != int_of_string(v2)) )
                                         | Lt -> string_of_int( boolean (int_of_string(v1) < int_of_string(v2)) )
                                         | Lteq -> string_of_int( boolean (int_of_string(v1) <= int_of_string(v2)) )
                                         | Gt -> string_of_int( boolean (int_of_string(v1) > int_of_string(v2)) )
                                         | Gteq -> string_of_int( boolean (int_of_string(v1) >= int_of_string(v2)) )
                                                | And -> string_of_int( boolean( (int_of_string(v1) = 0) && (int_of_string(v2)
= 0) ) )

                                                | Or -> string_of_int( boolean( (int_of_string(v1) = 0) || (int_of_string(v2) =
0) ) )

                                                | Amp -> v1 ^ v2
                                                | Eq -> string_of_int (boolean (v1 = v2))
                                                | Neq -> string_of_int (boolean (v1 <> v2))
                                                ), env
                (* ------------------------------------------------- *)
  | Assign(var, e) ->
                                                let v, (int_var_decls, string_var_decls, func_decls) = eval env e

                                                in
                                                let assign_int (int_var_decls, string_var_decls, func_decls) var v =
                                                        (NameMap.find var int_var_decls).(0) <- int_of_string(v);
                                                        (int_var_decls, string_var_decls, func_decls)
                                                in
                                                let assign_string (int_var_decls, string_var_decls, func_decls) var v =
                                                        (NameMap.find var string_var_decls).(0) <- v;
                                                        (int_var_decls, string_var_decls, func_decls)
                                                in
                                                if NameMap.mem var int_var_decls then
                                                        v, (assign_int (int_var_decls, string_var_decls, func_decls) var v)

                                                else if NameMap.mem var string_var_decls then
                                                        v, (assign_string (int_var_decls, string_var_decls, func_decls) var v)
                                                else
```

```
                                                    raise (Failure ("undeclared identifier " ^ var))
          (* -------------------------------------------------- *)
 | AssignArrayElement(var, i, e) ->
                                          let v, (int_var_decls, string_var_decls, func_decls) = eval env e

                                          in
                                          let assign_int_arr_element (int_var_decls, string_var_decls, func_decls) var i
v =
                                                    (NameMap.find var int_var_decls).(i) <- int_of_string(v);
                                                    (int_var_decls, string_var_decls, func_decls)
                                          in
                                          let assign_string_arr_element (int_var_decls, string_var_decls, func_decls)
var i v =
                                                    (NameMap.find var string_var_decls).(i) <- v;
                                                    (int_var_decls, string_var_decls, func_decls)
                                          in
                                          if NameMap.mem var int_var_decls then
                                                    v, (assign_int_arr_element (int_var_decls, string_var_decls,
func_decls) var i v)
                                          else if NameMap.mem var string_var_decls then
                                                    v, (assign_string_arr_element (int_var_decls, string_var_decls,
func_decls) var i v)
                                          else
                                                    raise (Failure ("undeclared identifier " ^ var))
          (* -------------------------------------------------- *)
          | AppendFile (e1, e2) ->
                                          let filepath, env = eval env e1
                                          in
                    let appendString, env = eval env e2
                                          in
                              let out_channel = open_out_gen [Open_append;Open_creat] 0o777 filepath
                                          in
                                          output_string out_channel appendString;
                                          close_out out_channel;
                                          "0", env
 (* -------------------------------------------------- *)
          | GetLine (e1, e2) ->
                                          let filepath, env = eval env e1
                                          in
                    let lineNumber, env = eval env e2
                                          in
                                          (* ---------------------- *)
                                          (* Define helper function *)
                                          let rec input_line_n in_channel curr_line_num targ_line_num =

                                           let line = input_line in_channel
                                                    in
                                           if curr_line_num >= targ_line_num then
                                            line, env
                                           else
                                            input_line_n in_channel (curr_line_num+1) targ_line_num;

                                                    in
                                          (* ---------------------- *)
```

```
                                                (try
                                                               let in_channel = open_in_gen [Open_rdonly] 0o777 filepath
                                                               in
                                                               try
                                                                let v, env = input_line_n in_channel 1 (int_of_string
lineNumber);
                                                                       in
                                                                       close_in in_channel;
                                                                       v, env
                                                               with End_of_file -> close_in in_channel; ("", env)
                                                       with End_of_file -> ("", env))

        (* -------------------------------------------------- *)
        | Print (e) ->
                                               let v, env = eval env e
                                               in
                                       print_string (v);
                                               "0", env

        (* -------------------------------------------------- *)
        | RegexMatch (e1, e2) ->
                                               let content, env = eval env e1
                                               in
                       let regex_string, env = eval env e2
                                               in
                                               let regex = Str.regexp regex_string
                                               in
                                               (let rec srch_fwd_and_match regex content pos =
                                                               try
                                                               let _ = Str.search_forward regex content pos
                                                               in
                                                                       let result = Str.matched_string content
                                                                       in

                                                                       if result = "" then

                                                                         srch_fwd_and_match regex content (pos+1)
                                                                               else
                                                                                       result

                                                               with
                                                                       | Not_found -> srch_fwd_and_match regex content
(pos+1)
                                                                       | Invalid_argument(a) -> ""

                                                in
                                                srch_fwd_and_match regex content 0), env
        (* -------------------------------------------------- *)
        | RegexReplace (e1, e2, e3) ->
                                               let content, env = eval env e1
                                               in
                       let regex_string, env = eval env e2
                                               in
                                               let regex = Str.regexp regex_string
                                               in
```

```
                                                let rep_string, env = eval env e3
                                                in
                                                (Str.global_replace regex rep_string content), env
            (* ------------------------------------------------ *)
            | StrReplace (e1, e2, e3) ->
                                                let content, env = eval env e1
                                                in
                        let match_string, env = eval env e2
                                                in
                                                let regex = Str.regexp_string match_string
                                                in
                                                let rep_string, env = eval env e3
                                                in
                                                (Str.global_replace regex rep_string content), env

            (* ------------------------------------------------ *)
  | Call(f) ->
                         let (int_var_decls, string_var_decls, func_decls) = env
                                                in
                                    let fdecl =
                                    try
                                                            NameMap.find f func_decls
                                    with Not_found -> raise (Failure ("undefined function " ^ f))
                                    in
                                    try
                                    let env = call fdecl env
                                                in
                                                "0", env
                                    with ReturnException(v, env) -> v, env

(* ==================================================== *)
(* Execute a statement and return an updated environment *)
and exec env = function
            | Block(stmts) -> List.fold_left exec env stmts
            (* ------------------------------------------------ *)
  | Expr(e) -> let _, env = eval env e
                      in
                                                                        env
            (* ------------------------------------------------ *)
  | If(e, stmts1_rev, stmts2_rev) ->
                        let stmts1 = List.rev stmts1_rev
                                    in
                                    let stmts2 = List.rev stmts2_rev
                                    in
                                    (* First evaluate e and store the result in v *)
                                    let v, env = eval env e
                                    in
                                    (* If v == 0, exec s1 otherwise exec s2 *)
                                    List.fold_left exec env (if int_of_string(v) == 0 then

                          stmts1

                        else

                          stmts2)
```

```
          (* ------------------------------------------------- *)
 | While(e, stmts_rev) ->
                              let stmts = List.rev stmts_rev
                              in
                              let rec loop env =
                                        (* First evaluate e and store the result in v *)
                    let v, env = eval env e
                                        in
                                        (* Keep looping if v == 0 then exec s *)
                    if int_of_string(v) == 0 then
                                                  loop (List.fold_left exec env stmts)
                                        else
                                                  (* otherwise return updated env *)
                                                  env
                              in
                              loop env

(* ============================================================ *)
(* Invoke a function and return an updated global symbol table *)
and call fdecl env =
          (* ======================================================================= *)
          (* Execute each statement in sequence, return updated global symbol table *)
          List.fold_left exec env fdecl.fbody


(* =============================== *)
(* Main entry point: run a program *)
let run (vars, funcs) =
          (* -------------------------------------------------- *)
          (* Put int and intArray declarations in a symbol table *)
          (* Initialize ints and intArray elements to 0 *)
          let int_var_decls = List.fold_left
                               (fun vars vdecl ->
                                                                                              match
vdecl.vtype with

          | Int | IntArray -> NameMap.add vdecl.vname (Array.make vdecl.vsize 0) vars


          | _ -> vars
                                                                                    )
                               NameMap.empty vars
          in

          (* ------------------------------------------------------ *)
          (* Put string and stringArray declarations in a symbol table *)
          (* Initialize strings and stringArray elements to "" *)
          let string_var_decls = List.fold_left
                                 (fun vars vdecl ->

          match vdecl.vtype with

| String | StringArray -> NameMap.add vdecl.vname (Array.make vdecl.vsize "") vars
```

```
        | _ -> vars
                                                                                                    )
                          NameMap.empty vars
        in

        (* ------------------------------------------- *)
(* Put function declarations in a symbol table *)
        let func_decls = List.fold_left
                        (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
                        NameMap.empty funcs
in

        (* -------------------------------------------------------------------- *)
        (* Run a program: find and run "main" *)

        try
          call (NameMap.find "main" func_decls) (int_var_decls, string_var_decls, func_decls)
        with Not_found ->
                raise (Failure ("did not find function 'main'"))
```

printer.ml

```
open Ast

let rec string_of_expr = function
             | StringLiteral(s) -> s
  | IntLiteral(l) -> string_of_int l
  | Id(s) -> s
             | ArrayElement(s, e) -> s ^ "[" ^ string_of_int(e) ^ "]"
             | ArrayElementViaId(s1, s2) -> s1 ^ "[" ^ s2 ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
                                        | Add -> "+"
                                        | Sub -> "-"
      | Equals -> "=="
                                        | NotEquals -> "!="
      | Lt -> "<"
                                        | Lteq -> "<="
                                        | Gt -> ">"
                                        | Gteq -> ">="
                                        | And -> "and"
                                        | Or -> "or"
                                        | Amp -> "&"
                                        | Eq -> "eq"
                                        | Neq -> "neq"
                                        ) ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
             | AssignArrayElement(v, i, e) -> v ^ "[" ^ string_of_int i ^ "] = " ^ string_of_expr e
             | AppendFile(e1, e2) -> "appendFile " ^ string_of_expr e1 ^ " " ^ string_of_expr e2
             | GetLine(e1, e2) -> "getLine " ^ string_of_expr e1 ^ " " ^ string_of_expr e2
             | Print(e) -> "print " ^ string_of_expr e
             | RegexMatch(e1, e2) -> "regexMatch " ^ string_of_expr e1 ^ " " ^ string_of_expr e2
             | RegexReplace(e1, e2, e3) -> "regexReplace " ^ string_of_expr e1 ^ " " ^ string_of_expr e2 ^ " " ^
string_of_expr e3
             | StrReplace(e1, e2, e3) -> "strReplace " ^ string_of_expr e1 ^ " " ^ string_of_expr e2 ^ " " ^
string_of_expr e3
             | Call(f) -> f

let rec string_of_stmt = function
    Block(stmts) -> "\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "\n"
  | Expr(expr) -> string_of_expr expr ^ "\n";
  | If(e, stmts, []) -> "if " ^ string_of_expr e ^ " then \n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "\n
end \n"
  | If(e, stmts1, stmts2) ->  "if " ^ string_of_expr e ^ " then \n" ^
                          "\n" ^ String.concat "" (List.map string_of_stmt stmts1) ^ "\n" ^

                            "else\n" ^

                            "\n" ^ String.concat "" (List.map string_of_stmt stmts2) ^ "end \n"
  | While(e, stmts) -> "while " ^ string_of_expr e ^ " then \n" ^ String.concat "" (List.map string_of_stmt stmts)
^ "\n end \n"

let string_of_var_type = function
```

```
        | Int -> "int "
        | String -> "string "
        | IntArray -> "intArray "
        | StringArray -> "stringArray "

let string_of_vdecl vdecl =
        (string_of_var_type vdecl.vtype) ^
        vdecl.vname ^ " \n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "\n" ^
  String.concat "" (List.map string_of_stmt fdecl.fbody) ^ " \n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

stepl.ml

```
let print = false

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  if print then
    let listing = Printer.string_of_program program in
    print_string listing
  else
    ignore (Interpreter.run program)
```

stepl-make.bat

```
ocamllex scanner.mll
ocamlyacc parser.mly

ocamlc -c ast.mli
ocamlc -c parser.mli

ocamlc -c scanner.ml
ocamlc -c parser.ml
ocamlc -c interpreter.ml
ocamlc -c printer.ml
ocamlc -c stepl.ml

ocamlc str.cma -o stepl.exe scanner.cmo parser.cmo interpreter.cmo printer.cmo stepl.cmo
```

stepl-clean.bat

```
del stepl.exe
del parser.ml
del scanner.ml
del *.cmi
del *.cmo
del parser.mli
```

stepl-test.bat

```
@echo off

set testdir=tests

REM ****************************************************************
REM Ensure that stepl-make has been called before stepl-test is run.
if not exist stepl.exe (goto :NOMAKE)

REM **********************************************
REM First delete all the .out files and .txt files
del %testdir%\*.out
del %testdir%\*.txt

REM ***************************
REM Now run the stepl test files

stepl < %testdir%\test-andor1.stpl > %testdir%\test-andor1.out 2>&1
stepl < %testdir%\test-andor2.stpl > %testdir%\test-andor2.out 2>&1
stepl < %testdir%\test-andor3.stpl > %testdir%\test-andor3.out 2>&1
stepl < %testdir%\test-arr1.stpl > %testdir%\test-arr1.out 2>&1
stepl < %testdir%\test-arr2.stpl > %testdir%\test-arr2.out 2>&1
stepl < %testdir%\test-arith1.stpl > %testdir%\test-arith1.out 2>&1
stepl < %testdir%\test-arith2.stpl > %testdir%\test-arith2.out 2>&1
stepl < %testdir%\test-arith3.stpl > %testdir%\test-arith3.out 2>&1
stepl < %testdir%\test-builtin1.stpl > %testdir%\test-builtin1.out 2>&1
stepl < %testdir%\test-builtin2.stpl > %testdir%\test-builtin2.out 2>&1
stepl < %testdir%\test-builtin3.stpl > %testdir%\test-builtin3.out 2>&1
stepl < %testdir%\test-builtin4.stpl > %testdir%\test-builtin4.out 2>&1
stepl < %testdir%\test-builtin5.stpl > %testdir%\test-builtin5.out 2>&1
stepl < %testdir%\test-comment1.stpl > %testdir%\test-comment1.out 2>&1
stepl < %testdir%\test-func1.stpl > %testdir%\test-func1.out 2>&1
stepl < %testdir%\test-gcd.stpl > %testdir%\test-gcd.out 2>&1
stepl < %testdir%\test-hello.stpl > %testdir%\test-hello.out 2>&1
stepl < %testdir%\test-if1.stpl > %testdir%\test-if1.out 2>&1
stepl < %testdir%\test-if2.stpl > %testdir%\test-if2.out 2>&1
stepl < %testdir%\test-if3.stpl > %testdir%\test-if3.out 2>&1
stepl < %testdir%\test-if4.stpl > %testdir%\test-if4.out 2>&1
stepl < %testdir%\test-if5.stpl > %testdir%\test-if5.out 2>&1
stepl < %testdir%\test-if6.stpl > %testdir%\test-if6.out 2>&1
stepl < %testdir%\test-ops1.stpl > %testdir%\test-ops1.out 2>&1
stepl < %testdir%\test-ops2.stpl > %testdir%\test-ops2.out 2>&1
stepl < %testdir%\test-var1.stpl > %testdir%\test-var1.out 2>&1
stepl < %testdir%\test-while1.stpl > %testdir%\test-while1.out 2>&1
stepl < %testdir%\test-while2.stpl > %testdir%\test-while2.out 2>&1

REM ********************************************
REM Now verify the results are what was expected

set /p andor1= < %testdir%\test-andor1.out
set /p andor2= < %testdir%\test-andor2.out
set /p andor3= < %testdir%\test-andor3.out
set /p arr1= < %testdir%\test-arr1.out
set /p arr2= < %testdir%\test-arr2.out
set /p arith1=< %testdir%\test-arith1.out
set /p arith2=< %testdir%\test-arith2.out
set /p arith3=< %testdir%\test-arith3.out
set /p builtin1=< %testdir%\test-builtin1.txt
set /p builtin2=< %testdir%\test-builtin2.out
set /p builtin3=< %testdir%\test-builtin3.out
set /p builtin4=< %testdir%\test-builtin4.out
set /p builtin5=< %testdir%\test-builtin5.out
set /p comment1=< %testdir%\test-comment1.out
set /p func1=< %testdir%\test-func1.out
set /p gcd=< %testdir%\test-gcd.out
set /p hello=< %testdir%\test-hello.out
set /p if1=< %testdir%\test-if1.out
set /p if2=< %testdir%\test-if2.out
set /p if3=< %testdir%\test-if3.out
set /p if4=< %testdir%\test-if4.out
set /p if5=< %testdir%\test-if5.out
set /p if6=< %testdir%\test-if6.out
set /p ops1=< %testdir%\test-ops1.out
set /p ops2=< %testdir%\test-ops2.out
set /p var1=< %testdir%\test-var1.out
set /p while1=< %testdir%\test-while1.out
set /p while2=< %testdir%\test-while2.out

if %andor1% EQU 01110001 (echo passed: test-andor1) else (echo failed: test-andor1)
if %andor2% EQU 01110001 (echo passed: test-andor2) else (echo failed: test-andor2)
if %andor3% EQU 01110001 (echo passed: test-andor3) else (echo failed: test-andor3)
if %arr1% EQU 42 (echo passed: test-arr1) else (echo failed: test-arr1)
if %arr2% EQU hello (echo passed: test-arr2) else (echo failed: test-arr2)
if %arith1% EQU 42 (echo passed: test-arith1) else (echo failed: test-arith1)
if %arith2% EQU 4 (echo passed: test-arith2) else (echo failed: test-arith2)
if %arith3% EQU 4 (echo passed: test-arith3) else (echo failed: test-arith3)
if %builtin1% EQU yes (echo passed: test-builtin1) else (echo failed: test-builtin1)
if %builtin2% EQU yes (echo passed: test-builtin2) else (echo failed: test-builtin2)
if %builtin3% EQU yes (echo passed: test-builtin3) else (echo failed: test-builtin3)
if %builtin4% EQU no (echo passed: test-builtin4) else (echo failed: test-builtin4)
if %builtin5% EQU no (echo passed: test-builtin5) else (echo failed: test-builtin5)
if %comment1% EQU yes (echo passed: test-comment1) else (echo failed: test-comment1)
if %func1% EQU 42 (echo passed: test-func1) else (echo failed: test-func1)
if %gcd% EQU 2311 (echo passed: test-gcd) else (echo failed: test-gcd)
if %hello% EQU hello42 (echo passed: test-hello) else (echo failed: test-hello)
if %if1% EQU 4217 (echo passed: test-if1) else (echo failed: test-if1)
if %if2% EQU 4217 (echo passed: test-if2) else (echo failed: test-if2)
if %if3% EQU 17 (echo passed: test-if3) else (echo failed: test-if3)
if %if4% EQU 817 (echo passed: test-if4) else (echo failed: test-if4)
if %if5% EQU 456 (echo passed: test-if5) else (echo failed: test-if5)
if %if6% EQU 1 (echo passed: test-if6) else (echo failed: test-if6)
if %ops1% EQU 3-1991099019901990019910991 0099 (echo passed: test-ops1) else (echo failed: test-ops1)
if %ops2% EQU 120110 (echo passed: test-ops2) else (echo failed: test-ops2)
if %var1% EQU 42 (echo passed: test-var1) else (echo failed: test-var1)
if %while1% EQU 5432142 (echo passed: test-while1) else (echo failed: test-while1)
if %while2% EQU 1 (echo passed: test-while2) else (echo failed: test-while2)

REM *********************************************
REM Delete all the .out files and .txt files
```

```
del %testdir%\*.out
del %testdir%\*.txt

GOTO :END

:NOMAKE
echo.
echo Please run stepl-make before running stepl-test

:END

echo on
```

Note: Please see “tests” folder for 29 test STePL files.