ACL: Automated Command Line

Cheow Lip Goh (Paul)
December 18, 2006

Contents

Chapter 1 – Introduction to ACL

ACL (Automated Command Line) is an interpreted language designed to automated repetitive tasks on the command line. The interpreter will be based on Java mainly because existing implementation of Expect suffers from the problems of C, where you need a separate interpreter for every OS. Using Java will ensure that the interpreter is useful regardless of the OS used and be able to run it whenever a JVM for the OS exist.

1.1 Design of ACL

In the next few subsections, a list of desirable language features of ACL will be described.

1.1.1 Very Simple and Familiar Syntax and Semantics

ACL is designed to be very similar to the syntax and semantics of C/Java, making it a very easy and intuitive language to pick up for most people. One major difference that ACL has from C/Java is that the variables of ACL don't have to be pre-defined like in C/Java, but rather the variable will come into existence the moment you assign a value to it, much like what common scripting languages like Tcl/tk or Python does.

1.1.2 Cross Platform Compatibility

ACL will be able to run whenever there is a java virtual machine. However, as of this writing, ACL only works with linux and does not work with Windows. Although much effort has been expanded to make ACL work under Windows, it seems that the Java classes that interact with the spawned process in Windows is very platform specific and currently does not work very well.

1.1.3 ACL is Interpreted

ACL is interpreted to enable a programmer to test the code without the compilation process.

Chapter 2 – Tutorial

2.1 Running the ACL Program

After the source code of the ACL program is written, it is very easy to run the program. Under linux, the following command would run ACL:
java -cp “<path_to_antlr_installation>/antlr.jar:<path_to_ACL_installation>” AclMain -f <acl_source_file>

A concrete example would be:
java -cp “/home/paulgoh/antlr-2.7.6/antlr.jar:/home/paulgoh/ACL” AclMain -f test.acl

2.2 A Simple Example

In our simple example, we define a function called telnet_login, which, when supplied with the server, user and password, will login to a server. Later, we invoke the telnet_login function and send some command after logging in.

Here is the example:

```
/* Our first ACL example. */
// Definition of the telnet_login function to login to a certain server.
// Arguments:
// server - name or ip address of the server.
// user - username of the user logging in.
// password - password of the user logging in
function telnet_login(server, user, password) {
        // Open a telnet session to the server.
        session(“telnet “ + server);

        // Wait for the login prompt.
        receive("login:");
        // Send the username.
        send(user);
        // Wait for the password prompt.
        receive("word:");
        // Send the password.
        send(password);
}
```

```
// Definitions of variables.
user = “paulgoh”;
password = “secret”;
server = “localhost”;
prompt = “$”;

// Call the telnet_login function, passing in the appropriate variables.
telnet_login(server, user, password);

// Wait for the shell prompt.
receive(prompt);
// Send an ls command to list the content of the home directory.
send(“ls”);
----------
```

Let’s take a detailed look at the code.

ACL takes the standard C/Java style comments, i.e., /* A comment */ or // A comment. The simple example is sprinkled with comments that explain very clearly what each section of the ACL code is doing.

To define a function in ACL, one has to use the “function” keyword and supply a function name (i.e., telnet_login) together with a comma delimited list of variables. The function body contains the code to be executed when the function is invoked.

ACL has 3 built-in functions, namely session(), receive() and send(). The session() function takes a command in the form of a string that will spawn a new login session. In our example, a telnet session was spawned using “telnet “ + server. The “+” concatenates two strings together. If the concatenation is between a number and a string (i.e., “abc “ + 1), the end result is the string “abc 1”. The receive() function takes a string to wait for, and when the string is found in the spawned session, receive will return. The send() function sends a command specified by the string passed into it to the spawned session. Together, these 3 built-in functions will interact with the spawned login session.

Assigning a value to a variable will bring the variable into existence in ACL. i..e:

```
----------
a = 1;
b = “a string”;
----------
```

These two lines of code will create a variable called “a”, which has a type of integer, with a value of 1 and also another variable called “b”, which has a type of string, with a value of “a string”. Strings in ACL are surrounded by double quotes. As you can see, the variables come into existence in ACL simply by assigning values to it, and the type of the variables are implicit, depending on whether the value assigned to it is surrounded by double quotes (which makes it a string), or digits without double quotes (which makes it an integer).

Lastly, a function can be invoked by using the function name together with a list of arguments specified by the function definition.

Here is a sample of what will be observed as output when the example is run:

```
----------
[root@localhost test]# java -cp "/root/antlr-2.7.6/antlr.jar:/root/test" AclMain -f /root/test/test.acl
Trying 127.0.0.1...
Connected to localhost.localdomain (127.0.0.1).
Escape character is '^]'.
Red Hat Enterprise Linux AS release 3 (Taroon)
Kernel 2.4.21-4.ELsmp on an i686
login: paulgoh
Password:
Last login: Sun Dec 17 00:04:49 from localhost.localdomain
[paulgoh@localhost paulgoh]$ ls
ftp  test.acl
[paulgoh@localhost paulgoh]$ [root@localhost test]#
----------
```

Chapter 3 – Language Reference Manual

3.1 Lexical Conventions

ACL has 5 kinds of tokens: identifiers, keywords, constants, expression operators, and other separators. Blanks, tabs, newlines and comments are ignored unless they serve to separate tokens.

3.1.1 Comments

The first style of comments starts with /* and terminates with */. The seconds style starts with // and terminates with end of line.

3.1.2 Identifiers

An identifier is a series of letters and digits. The first character of an identifier must be a letter, followed by any number of letters or digits or underscore. Upper and lower case letters are distinct.

3.1.3 Keywords

The following table lists all the keywords in ACL. Keywords are reserved identifiers and should not be used for any other purpose:

session	Receive	send	if
Else	While	for	function
Return	Break	continue	

3.1.4 Constants

There are only two kinds of constants in ACL, namely Integer and String.

3.1.4.1 Integer Constants

An integer constant is a sequence of digits.

3.1.4.2 String Constants

A string constant is a sequence of characters surrounded by double quotes “. The forward slash followed by a double quote \” is used to represent a double quote within the string.

3.1.5 Other Tokens

(	)	{	}	;	,
+	-	*	/	++	--
>	<	>=	<=	==	=

3.2 Expressions

3.2.1 Primary Expressions

Primary expressions include identifiers, constants, parenthesized expressions and function calls.

3.2.1.1 Identifiers

An identifier could be an lvalue expression or rvalue expression. An rvalue expression will be evaluated.

3.2.1.2 Constants

A constant’s value determines its type (i.e., Integer or String). A constant can only be an rvalue.

3.2.1.3 Parenthesized Expressions

Parenthesized expressions’ type and value are identical to the unparenthesized expressions.

3.2.1.4 Function Calls

Function calls consist of a function identifier followed by a parenthesized list of arguments to the function. Each argument is an expression. The function call itself is an rvalue expression.

3.2.2 Arithmetic Expressions

Arithmetic expressions take primary expressions as operands and evaluate those using operators.

3.2.2.1 Binary Operators

Binary arithmetic operators include +, -, *, /. They indicate addition, subtraction, multiplication and division. Binary operators can only operate on Integer constants operands. Multiplication and division operators have a higher precedence than addition and subtraction operators. When the precedence of the operators are the same, i.e., addition and subtraction, the associatively is from left to right.

3.2.3 Relational Expressions

Relational operators >, <, >=, <=, !=, == are binary relational operator representing whether the first operand is greater than, lesser than, greater than or equal, lesser than or equal, not equal and equal to the second operand. These operators evaluate to 0 if true and 1 if false.

3.2.4 Assignment Expressions

The assignment operator is =. The assignment expression consist of an lvalue to the left of the assignment operator, and an rvalue to the right of the assignment operator. The lvalue must be a modifiable identifier. The rvalue can be any identifier or constant.

3.3 Statements

Except as indicated, statements execute in sequence.

3.3.1 Expression Statement

Most statements are expression statements, which have the form:

expression ;

Expressions statements are usually assignments or function calls.

3.3.2 Compound Statement

Compound statement is a group of statements surrounded by open and close braces. It is used when a group of statements is needed in place of a single statement.

3.3.3 Conditional Statement

The two forms of condition statements are:

If (expression) then statement
If (expression) then statement else statement

In the first form, the first substatement is executed if the expression evaluates to a non-zero. In the second form, the second substatement is executed if the expression evaluates to zero. The else ambiguity is resolved by connecting the else to the last encountered elseless if.

3.3.4 While Statement

The while statement has the form:

while (expression) statement

As long as the value of the expression remains non-zero, the substatement is executed repeatedly

3.3.5 For Statement

The for statement has the form:

for (expression-1 ; expression-2 ; expression-3) statement

expression-1 specifies the initialization for the loop.
expression-2 specifies the test to perform such that as long as the expression-2 evaluations to non-zero, the for loop will continue to run.
expression-3 specifies the incrementation made after the end of each loop iteration.

3.3.6 Break Statement

The break statement has the following form:

break ;

If executed, the break statement will cause the termination of the smallest enclosing while or for statement. Execution control will pass on to the statement following the terminated statement.

3.3.7 Continue Statement

The continue statement has the following form:

continue ;

If executed, the continue statement will pass control to the loop continuation portion of the smallest enclosing while or for statement.

3.3.8 Return Statement

The return statement two forms:

return ;
return expression ;

The return statement is always used within a function. The first form of return statement returns no value. The second form of the return statement will return the value of the expression to the function caller.

3.4 Function Definition

A function definition has the following form:

```
function function-name ( argument-list ) {
        statement
}
```

function-name is the name of the function. argument-list is a comma separated list of arguments to be passed into the function. argument-list could be empty. The return statement is optional and could be used when the function needs to return, with or without a value.

3.5 Built-in Functions

ACL has 3 built in functions, namely session, receive and send. These 3 functions are the key to automatically interacting with the command line locally or remotely.

3.5.1 Session Function

The session function, when invoked, will spawn a local or remote command line session, depending on the shell program used to spawn the session (i.e., telnet, ssh, etc). The session function is invoked just like a regular function and takes an identifier or a string constant. If an identifier is used, the identifier must evaluate to a string.

3.5.2 Receive Function

The receive function's invocation is only meaningful after the session function is called. The receive function will wait for a certain sequence of characters to be sent back by the spawned session to determine if there is a match. The receive function is invoked just like a regular function and takes an identifier or a string constant. If an identifier is used, the identifier must evaluate to a string.

3.5.3 Send Function

The send function's invocation is only meaningful after the session function is called. The send function will send a certain sequence of character to the spawned session. The send function is invoked just like a regular function and takes an identifier or a string constant. If an identifier is used, the identifier must evaluate to a string. The newline or carriage return is automatically used in the send function, so there is no need to explicitly specify a newline character "\n" in the string.

Chapter 4 – Project Plan

4.1 Project Planning Process

There were no formal roles and responsibilities assignment as this project was solely done by one person. The process used for the development of the interpreter is some form of rapid prototyping of the important components of the interpreter, to provide confidence and ensure that there is a path to the project completion. Here are some of the prototypes that were constructed:

1. The command line interaction program written in java. This is the first prototype that was developed, suggested by the professor to ensure that a method to interact with the command line is available to complete the project.
2. A small prototype of the lexer and parser are created to gain a deeper understanding of the lexical analysis and parsing process. Later, the lexer and parser were extended to include the whole ACL language. The lexer and parser were completed when the LRM was written.
3. After the lexer and parser are completed, the interpreter code was added. The addition of the interpreter code will include static semantics analysis where the tokens are checked for correct ordering and consistency. Also, the executions of various loops like “while” and “for” are added. At this stage, the function call mechanisms are added too, together with the symbol table and scoping (i.e., new scope within a function, while inheriting the parent scope.).
4. Lastly, the built-in functions that will interact with the command line are added and these built-in functions are ingrated with the command line interaction prototype that was developed in the beginning.

4.2 Programming Style Guide

There isn’t any formal programming style guide as I am the only programmer. However, here are some coding styles that I personally follow:

4.2.1 Antlr Coding Style

The colon ":" is always placed at the 9th column unless the string in front of the colon is too long. The semi colon ";" is always placed one space after a one line rule, however, for a multiple line rule, the semi colon ";" is always placed at the same column of the colon ":", but one line after the last line of the rule.

The names used in lexer are all in upper-case while the names used in the parser are all in lower case.

4.2.2 Java Coding Style

Generally, the standard java coding convention was followed. Here are some of the more important coding style that I use:

1. Indentions are always 4 spaces literally (no tabs are used.)
2. The open braces "{" always comes after the first line of code that required the open braces (i.e., not in a new line.) and the close braces "}" always appears in a new line after the last line of the construct that requires the close braces.
3. Class names always begin in upper-case letters (i.e., AclMain).
4. Method and variable names always begins in lower-case letters. Subsequent syllabus in the name will always begin in upper-case (i.e., getName)
5. Open and close braces are always used, even when one is not explicitly required. i.e., open braces is used even if there is only one line of code following an "if" statement.
6. Variable names should be self-explanatory. i.e., no additional comments should be required to explain the variable.
7. Operators between two operands should be separated by a space. i.e., a = b + c;

4.3 Project Timeline

Date	Task Due
September 26	White paper completed.
October 13	Prototype of the command line interaction program completed.
October 17	LRM, lexer and parser completed.
November 15	Interpreter backend completed (Including static semantics analysis and symbol table).
November 30	Function and integration of command line interaction program completed (Including built-in functions.)
December 5	Final tests
December 12	Code complete

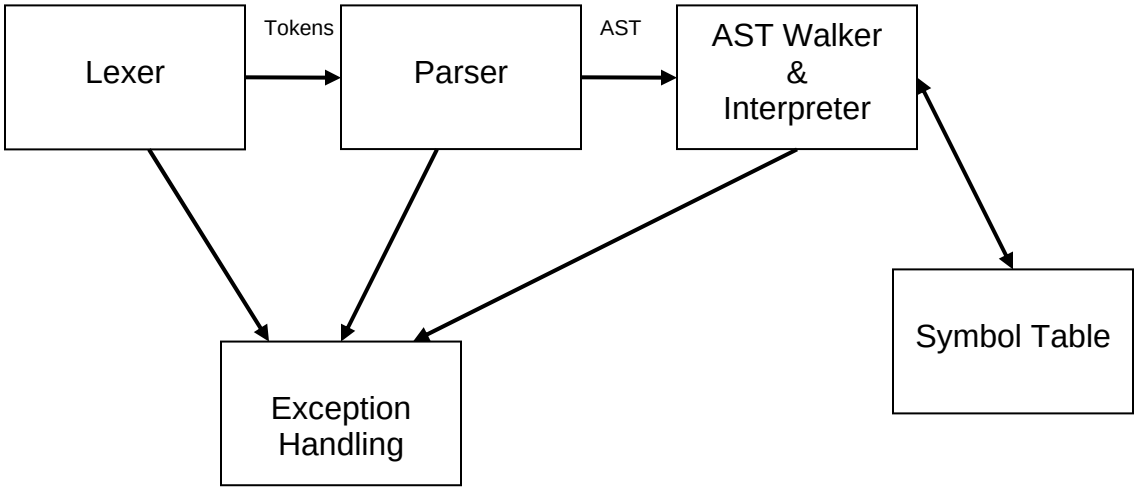
4.4 Software Development Environment

All source codes are compiled either on Windows XP or Redhat Linux AS 3.0. No source control program was used as this is a one person development effort. The lexer, parser and tree walker are all written using Antlr 2.7.6. The interpreter backend is written using java with jdk 1.5.

4.5 Project Log

Date	Task Completed
September 25	White paper completed.
October 16	Prototype of the command line interaction program completed.
October 19	LRM and compiler frontend completed.
November 14	Interpreter backend completed (Including static semantics analysis and symbol table).
November 24	Automated regression tests completed and available to test the interpreter after every code change.
December 2	Function and integration of command line interaction program completed (Including built-in functions.)
December 10	Final tests.
December 12	Code complete.
December 18	Final report is completed.

Chapter 5 – Architectural Design



The design of ACL is very simple, much like a standard interpreter. The source file will be read into the Lexer, which sends the tokens to the parser, and the parser produces the AST for the tree walker. The tree walker, depending on the structure of the AST, will invoke the interpreter, implemented by AclInterpreter, to execute the action required by the ACL source code. Therefore, the tree walker will hold onto an instance of the AclInterpreter for invocation purposes.

The AclInterpreter contains an instance of the root symbol table that will point to other child symbol tables (much like a linked list of symbol tables) as nested functions are being invoked.

The exception handling component is just a simple exception class that prints a descriptive error whenever some problems in the ACL source code is encountered.

Since the interface from the AST walker to the interpreter is the most interesting, I would describe some of the important interfaces in AclInterpreter in more details here:

getSymbolTable() – This method will retrieve the symbol table.
assign() – This method is used when a variable assignment is required.
getData() – This method is used to get the value of a variable.
invokeFunction() – This method is used during function invocation. This is the most complicated method in the interpreter.
registerFunction() – This method is used to register a newly defined function with the interpreter.
runBuiltInFunction() – This method is used to run one of the built-in function.

Chapter 6 – Test Plan

As a QA engineer myself, I understand fully that testing the compiler sufficiently is of paramount importance. There are 3 main types of tests that are performed on ACL. They are:

1. Automated regression tests.
2. Manual detailed tests on specific areas of the compiler.
3. Manual tests on the built-in functions, which is hard to test in any automated fashion due to its interactive nature with the command line.

6.1 Automated Regression Tests

These automated regression tests are automated and are performed after every single session of adding new code. Since these tests only take a few seconds to run, they are automatically run after compilation. These tests are not as exhaustive as the manual tests, but they provide a reasonable assurance that the new code does not break any existing and working features.

Here is the regression tests ACL source code:

```
/* Test Cases. */

// Assignment tests.
a = 1;
b = "testing";
a = b;
c = 1;
d = a;
d = c;

// String concatenation tests.
e = 1 + " testing";
f = "testing " + 1;

// Arithmetic tests.
g = 1 + 2;
h = 3 / 2;
i = 3 / 1;
j = 3 * 4;
k = 5 - 2;
l = k + i;
m = k - i;
n = k * i;
```

```
o = k / i;

// if else tests.
if (g == 3) {
    p = g;
} else {
    p = 0;
}

if (g != 3) {
    q = g;
} else if (i < 4) {
    q = i + 1;
}

if (g > 3) {
    r = 0;
} else {
    r = 1;
}

// while tests
s = 0;
while (s <= 15) {
    s = s + 1;
    if (s <= 15) {
        continue;
    }
}

t = 15;
while (t >= 1) {
    t = t - 1;
    if (t == 7) {
        break;
    }
}

// for tests
v = 0;
for (u = 0; u < 20; u = u + 1) {
    v = v + 1;
}
```

```
// function tests
function abc(a, b, c) {
    w = a;
    x = b;
    y = a + b + c;
    return y;
}

w = abc(10, 20, 30);

function def(a, b, c) {
    x = abc(a, b, c);
    return x;
}

x = def(1, 2, 3);
y = def(abc(1, 1, 1), def(1, 1, 1), abc(1, 1, 1));
```

Here is the actual java code found in the interpreter to execute the automatic analysis of regression tests results:

```
    public void regressionTest() {
        String expectedStringValue = null;
        int expectedIntValue = 0;
        String variableName = null;

        checkString("a", "testing");
        checkString("b", "testing");
        checkInteger("c", 1);
        checkInteger("d", 1);
        checkString("e", "1 testing");
        checkString("f", "testing 1");
        checkInteger("g", 3);
        checkInteger("h", 1);
        checkInteger("i", 3);
        checkInteger("j", 12);
        checkInteger("k", 3);
        checkInteger("l", 6);
        checkInteger("m", 0);
        checkInteger("n", 9);
        checkInteger("o", 1);
        checkInteger("p", 3);
        checkInteger("q", 4);
        checkInteger("r", 1);
        checkInteger("s", 16);
```

```
        checkInteger("t", 7);
        checkInteger("u", 20);
        checkInteger("v", 20);
        checkInteger("w", 3);
        checkInteger("x", 9);
        checkInteger("y", 9);
    }

    public void checkString(String variableName, String expectedStringValue) {
        if
(((AclString)getData(variableName)).getValue().equalsIgnoreCase(expectedStrin
gValue)) {
            System.out.println("Variable: \"" + variableName + "\" Expected value: \""
+ expectedStringValue + "\" found. Passed.");
        } else {
            System.out.println("Variable: \"" + variableName + "\" Expected value: \""
+ expectedStringValue + "\" not found. Failed.");
        }
    }

    public void checkInteger(String variableName, int expectedIntValue) {
        if (((AclInteger)getData(variableName)).getValue() == expectedIntValue) {
            System.out.println("Variable: \"" + variableName + "\" Expected value: " +
expectedIntValue + " found. Passed.");
        } else {
            System.out.println("Variable: \"" + variableName + "\" Expected value: " +
expectedIntValue + " not found. Failed.");
        }
    }
----------
```

Here is what one might expect to see if the regression tests all passed:

```
D:\Personal\Columbia University\COMS W4115\Project\acl>java -cp
"D:\Personal\Columbia University\COMS W4115\Project\antlr-
2.7.6\antlr.jar;D:\Personal\Columbia University\COMS W4115\Project\acl"
AclMain -f test.acl
Variable: "a" Expected value: "testing" found. Passed.
Variable: "b" Expected value: "testing" found. Passed.
Variable: "c" Expected value: 1 found. Passed.
Variable: "d" Expected value: 1 found. Passed.
Variable: "e" Expected value: "1 testing" found. Passed.
Variable: "f" Expected value: "testing 1" found. Passed.
Variable: "g" Expected value: 3 found. Passed.
Variable: "h" Expected value: 1 found. Passed.
Variable: "i" Expected value: 3 found. Passed.
Variable: "j" Expected value: 12 found. Passed.
Variable: "k" Expected value: 3 found. Passed.
Variable: "l" Expected value: 6 found. Passed.
Variable: "m" Expected value: 0 found. Passed.
Variable: "n" Expected value: 9 found. Passed.
Variable: "o" Expected value: 1 found. Passed.
Variable: "p" Expected value: 3 found. Passed.
Variable: "q" Expected value: 4 found. Passed.
Variable: "r" Expected value: 1 found. Passed.
Variable: "s" Expected value: 16 found. Passed.
Variable: "t" Expected value: 7 found. Passed.
Variable: "u" Expected value: 20 found. Passed.
Variable: "v" Expected value: 20 found. Passed.
Variable: "w" Expected value: 3 found. Passed.
Variable: "x" Expected value: 9 found. Passed.
Variable: "y" Expected value: 9 found. Passed.
```

6.2 Manual Detailed Tests on Specific Areas of the Compiler.

Examples of such manual detailed tests are:

// Assign a floating point number to a variable.
a = 1.1;

// Assigning an invalid string to a variable.
A = ““a b c””

// Assigning a non-existent variable yyy to xxx
xxx = yyy;

Since these tests mostly results in immediate failure and printing of error messages, they require more effort to automate and due to the lack of time, are done manually.

6.3 Manual tests on the built-in functions

These tests are even harder to perform than tests done on the language syntax itself, as the built-in functions interact directly with a spawned remote session like a telnet session. The screen output has to be visually inspected to determine if the interpreter is doing the right thing. An example of such test is already illustrated in the tutorial.

Chapter 7 – Lessons Learned

1. Writing an interpreter gave me an overall greater appreciation for the hard work and considerations present in the programming languages I have used. It allowed me greater insights into why a compiler/interpreter behaves this way or why a language was designed in such a manner.
2. I am glad that I did not wait until the last minute to start working on the interpreter as learning a new tool like ANTLR takes quite a significant amount of time. Furthermore, I managed to complete the development of the interpreter and still have time to do some testing.
3. Careful testing of any software is indispensable. I was able to find a lot of bugs during testing. The automated regression tests has been invaluable in finding bugs. I also found it fruitful to write suitable testcases for a new feature immediately after the feature has been written. This process of adding feature and immediately testing it will provide more assurance that the interpreter is working within expectations.
4. It is a blessing to work on this project alone. This is because as a working professional with a wife and daughter, it is not easy to find time to work on the interpreter. I would have to treasure any spare time that was available and work on the interpreter.

Appendix A – Code Listing

```
// grammar.g: Antlr code for the lexer, parser and tree
walker
// Author: Cheow Lip Goh (Paul)

class AclLexer extends Lexer;

options {
    k = 2 ;
    testLiterals = false ;
    charVocabulary='\u0000'..'\u007F' ; // allow ascii
}

PLUS    : '+' ;
MINUS   : '-' ;
TIMES   : '*' ;
DIV     : '/' ;
ASSIGN  : '=' ;
LPAREN  : '(' ;
RPAREN  : ')' ;
LBRACE  : '{' ;
RBRACE  : '}' ;
NOT     : '!' ;
LT      : '<' ;
GT      : '>' ;
SEMI    : ';' ;
COMMA   : ',' ;
EQ      : "==" ;
NE      : "!=" ;
LE      : "<=" ;
GE      : ">=" ;

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

ID options { testLiterals = true ; }
        : LETTER ( LETTER | DIGIT | '_')* ;
NUMBER  : (DIGIT)+ ;

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

COMMENT : ("/*" ( options { greedy = false ; } : . )* "*/"
        | "//" (~( '\n' | '\r' ))* ( '\r' '\n' | '\n'))
```

```
            {$setType(Token.SKIP);}
        ;

STRING  : '"'!
          ( ~('"' | '\n') )*
          '"'!
        ;

class AclParser extends Parser;
options {
    k = 2 ;
    buildAST = true ;
}

program : (statement)* EOF!
            {#program = #([STATEMENT,"PROG"], program); }
        ;

statement : if_statement
          | for_statement
          | while_statement
          | function_declaration_statement
          | function_call_statement
          | assignment_statement
          | return_statement
          | break_statement
          | continue_statement
          | LBRACE! (statement)* RBRACE!
            {#statement = #([STATEMENT,"STATEMENT"],
statement);}
          | empty_statement:SEMI
{#empty_statement.setType(EMPTY_STATEMENT);}
          ;

if_statement : "if"^ LPAREN! expression RPAREN! statement
             ( options { greedy = true ; } : "else"!
statement )?
             ;

for_statement : "for"^ LPAREN! for_init SEMI! for_condition
SEMI! for_iterator RPAREN! statement
              ;

for_init : (for_assignment_statement)? {#for_init =
#(#[FOR_INIT, "FOR_INIT"], #for_init);}
          ;
```

```
for_condition : (expression)? {#for_condition =
#(#[FOR_CONDITION, "FOR_CONDITION"], #for_condition);}
               ;

for_iterator : (for_assignment_statement)? {#for_iterator =
#(#[FOR_ITERATOR, "FOR_ITERATOR"], #for_iterator);}
              ;

for_assignment_statement : lvalue ASSIGN^ expression
                         ;

while_statement : "while"^ LPAREN! expression RPAREN!
statement
                ;

function_declaration_statement : "function"^ ID LPAREN!
parameter_list RPAREN! function_body
                               ;

parameter_list : ID ( COMMA! ID )*
                    {#parameter_list =
#([PARAMETER_LIST,"PARAMETER_LIST"], parameter_list); }
               | /* nothing. */
                    {#parameter_list =
#([PARAMETER_LIST,"PARAMETER_LIST"], parameter_list); }
               ;

function_body : LBRACE! (statement)* RBRACE!
                    {#function_body =
#([STATEMENT,"FUNCTION_BODY"], function_body); }
              ;

function_call_statement : function_call SEMI!
                        ;

function_call : ID LPAREN! argument_list RPAREN!
                    {#function_call =
#([FUNCTION_CALL,"FUNCTION_CALL"], function_call); }
              ;

argument_list : expression ( COMMA! expression )*
                    {#argument_list =
#([ARGUMENT_LIST,"ARGUMENT_LIST"], argument_list); }
               | /* nothing. */
                    {#argument_list =
#([ARGUMENT_LIST,"ARGUMENT_LIST"], argument_list); }
               ;
```

```
assignment_statement : lvalue ASSIGN^ expression SEMI!
                     ;

return_statement : "return"^ (expression)? SEMI!
                 ;

break_statement : "break"^ SEMI!
                ;

continue_statement : "continue"^ SEMI!
                   ;

expression : relational_expression ( ( EQ^ | NE^ )
relational_expression )*
                   ;

relational_expression : additive_expression ( ( GT^ | LT^ |
GE^ | LE^ ) additive_expression )*
                      ;

additive_expression : multiplicative_expression ( ( PLUS^ |
MINUS^ ) multiplicative_expression )*
                    ;

multiplicative_expression : rvalue ( ( TIMES^ | DIV^ )
rvalue )*
                          ;

rvalue : lvalue | NUMBER | STRING
       | LPAREN! expression RPAREN!
       | function_call
       ;

lvalue : ID^
       ;

{
    import java.io.*;
    import java.util.*;
}

class AclWalker extends TreeParser;
{
    AclInterpreter interpreter = new AclInterpreter();
}
```

```
expr returns [AclType r]
{
    AclType a, b;
    r = null;
    AclType [] al = null;
    String [] pl = null;
}
    : #(EQ a=expr b=expr {r = a.eq(b);})
    | #(NE a=expr b=expr {r = a.ne(b);})
    | #(GT a=expr b=expr {r = a.gt(b);})
    | #(LT a=expr b=expr {r = a.lt(b);})
    | #(GE a=expr b=expr {r = a.ge(b);})
    | #(LE a=expr b=expr {r = a.le(b);})
    | #(PLUS a=expr b=expr {r = a.plus(b);})
    | #(MINUS a=expr b=expr {r = a.minus(b);})
    | #(TIMES a=expr b=expr {r = a.times(b);})
    | #(DIV a=expr b=expr {r = a.div(b);})
    | ID {r = interpreter.getData(#ID.getText());}
    | NUMBER {r = new
AclInteger(Integer.parseInt(#NUMBER.getText(), 10));}
    | STRING {r = new AclString(#STRING.getText());}
    | #(ASSIGN ID a=expr
        {interpreter.assign(#ID.getText(), a.copy());})
    | #("for_assign" ID a=expr
        {interpreter.assign(#ID.getText(), a.copy());})
    | #("if" a=expr thenPart:. (elsePart:.)?)
    {
        if (((AclInteger)a).getValue() != 0) {
            expr(#thenPart);
        } else if (elsePart != null) {
            expr(#elsePart);
        }
    }
     | #("for"
         #(FOR_INIT (initPart:.)?)
         #(FOR_CONDITION (conditionPart:.)?)
         #(FOR_ITERATOR (iteratorPart:.)?)
             (forBody:.))
     {
         if (initPart != null) {
             expr(#initPart);
         }
         if (#conditionPart != null) {
         while
(((AclInteger)expr(#conditionPart)).getValue() != 0) {
                expr(#forBody);
                if (interpreter.breakFlagSet()) {
```

```
                    interpreter.resetFlags();
                    break;
                }
            expr(#iteratorPart);
                interpreter.resetFlags();
            }
        } else {
            while (true) {
                expr(#forBody);
                if (interpreter.breakFlagSet()) {
                    interpreter.resetFlags();
                    break;
                }
            expr(#iteratorPart);
                interpreter.resetFlags();
            }
        }
     }
    | #("while" whileCondition:. whileBody:.)
    {
        while
(((AclInteger)expr(#whileCondition)).getValue() != 0) {
            expr(#whileBody);
            if (interpreter.breakFlagSet()) {
                interpreter.resetFlags();
                break;
            }
            interpreter.resetFlags();
        }
    }
    | #("function" ID pl=parameterList functionBody:.)
    {
        interpreter.registerFunction(#ID.toString(), pl,
#functionBody);
    }
    | #(FUNCTION_CALL ID al=argumentList)
        {r = interpreter.invokeFunction(this,
#ID.getText(), al);}
    | #(STATEMENT (statement:. {if (interpreter.proceed())
r = expr(#statement);})*)
    | #("return" (returnExpression:.)?)
    {
        if (#returnExpression != null) {

interpreter.setReturnValue(expr(#returnExpression));
        }
    }
```

```
    | #("break" {interpreter.setBreakFlag();})
    | #("continue" {interpreter.setContinueFlag();})
     | EMPTY_STATEMENT
    ;

parameterList returns [String [] pl]
{
    Vector<String> v;
    pl = null;
}
    : #(PARAMETER_LIST {v = new Vector<String>();}
        (s:ID {v.add(s.getText());})*)
            {pl =
interpreter.convertParametersToStringArray(v);}
    ;

argumentList returns [AclType [] al]
{
    Vector<AclType> v;
    AclType a;
    al = null;
}
    : #(ARGUMENT_LIST {v = new Vector<AclType>();}
        (a=expr {v.add(a);})*)
            {al =
interpreter.convertArgumentsToStringArray(v);}
    ;
```

```
// AclMain.java: Main function for the ACL interpreter.
// Author: Cheow Lip Goh (Paul)

import antlr.*;
import antlr.collections.*;
import java.io.*;
import java.util.*;

public class AclMain {
    private static String sourceFileName = null;

    // Method to parse the command line.
    private static void parseCommandLine(String [] args) {
        if (args[0].equalsIgnoreCase("-f")) {
            // -f flag specifies the source file of the ACL
code.
            sourceFileName = new String(args[1]);
        } else {
            System.out.println("Usage: java AclMain -f
<acl_source_file>");
        }
    }

    // Method to execute the ACL source file.
    private static void executeSourceFile() {
        try {
            InputStream inputFile = (InputStream) new
FileInputStream(sourceFileName);

            AclLexer lexer = new AclLexer(inputFile);
            AclParser parser = new AclParser(lexer);

            parser.program();

            AclWalker walker = new AclWalker();
            AST tree = parser.getAST();

            // This line will print the AST.
            // System.out.println(tree.toStringList());

            walker.expr(tree);

            // This line will print the symbol table at the
end of the source code interpretation.
            //
walker.interpreter.getSymbolTable().print("Symbol table at
the end of the program.");
```

```
            // This line will execute the regression test
with the appropriate test.acl source code.
            // walker.interpreter.regressionTest();


walker.interpreter.getBuiltInFunctions().destroy();
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    public static void main(String [] args) throws
Exception {
        parseCommandLine(args);

        executeSourceFile();
    }
}
```

```
// AclInterpreter.java: Interpreter for ACL.
// Author: Cheow Lip Goh (Paul)

import antlr.*;
import antlr.collections.*;
import java.util.*;

class AclInterpreter {
    private AclSymbolTable symbolTable = null;
    private AclBuiltInFunctions builtInFunctions = null;
    private AclType returnValue = null;
    private boolean breakFlag = false;
    private boolean continueFlag = false;

    public AclInterpreter() {
        symbolTable = new AclSymbolTable(null);
        builtInFunctions = new AclBuiltInFunctions();
        builtInFunctions.initialize(symbolTable);
    }

    public AclSymbolTable getSymbolTable() {
        return symbolTable;
    }

    public void printError(String msg) {
        throw new AclException(msg);
    }

    public void assign(String keyName, AclType data) {
        AclSymbolTable parentSymbolTable = null;
        if (symbolTable.containsValue(keyName)) {
            symbolTable.setValue(keyName, data);
        } else {
            parentSymbolTable = symbolTable.getParent();
            while (parentSymbolTable != null) {
                if
(parentSymbolTable.containsValue(keyName)) {
                    parentSymbolTable.setValue(keyName,
data);
                    break;
                } else {
                    parentSymbolTable =
parentSymbolTable.getParent();
                }
            }
            if (parentSymbolTable == null) {
                symbolTable.setValue(keyName, data);
```

```
            }
        }
    }

    public AclType getData(String keyName) {
        AclSymbolTable parentSymbolTable = null;
        AclSymbolTable tempSymbolTable = null;
        AclType returnData = null;
        if (symbolTable.containsValue(keyName)) {
            returnData = symbolTable.getValue(keyName);
        } else {
            parentSymbolTable = symbolTable.getParent();
            while (parentSymbolTable != null) {
                if
(parentSymbolTable.containsValue(keyName)) {
                    returnData =
parentSymbolTable.getValue(keyName);
                    break;
                } else {
                    parentSymbolTable =
parentSymbolTable.getParent();
                }
            }
        }

        if (returnData != null) {
            return returnData;
        } else {
            printError("Invalid Variable: " + keyName);
            return null;
        }
    }

    public String [] convertParametersToStringArray(Vector
v) {
        String [] tempStringArray = new String[v.size()];
        for (int i = 0; i < tempStringArray.length; i++ ) {
            tempStringArray[i] = (String) v.elementAt( i );
        }
        return tempStringArray;
    }

    public AclType [] convertArgumentsToStringArray(Vector
v) {
        AclType [] tempAclTypeArray = new
AclType[v.size()];
        for (int i = 0; i < tempAclTypeArray.length; i++) {
```

```
            tempAclTypeArray[i] = (AclType)v.elementAt(i);
        }
        return tempAclTypeArray;
    }

    public AclType invokeFunction(AclWalker walker, String
functionName, AclType [] argumentList) {
        AclType testFunction = null;
        AclSymbolTable parentSymbolTable = null;
        if (symbolTable.containsValue(functionName)) {
            testFunction =
(AclType)symbolTable.getValue(functionName);
        } else {
            parentSymbolTable = symbolTable.getParent();
            while (parentSymbolTable != null) {
                if
(parentSymbolTable.containsValue(functionName)) {
                    testFunction =
parentSymbolTable.getValue(functionName);
                    break;
                } else {
                    parentSymbolTable =
parentSymbolTable.getParent();
                }
            }
        }

        // Check that the function identifier exist.
        if (testFunction == null) {
            printError(functionName + " function does not
exist!");
            return null;
        }

        // Check that the function identifier is indeed a
function.
        if (!(testFunction instanceof AclFunction)) {
            printError(functionName + " is not a
function!");
            return null;
        }

        AclFunction tempFunction =
(AclFunction)testFunction;

        if (tempFunction.isBuiltInFunction()) {
```

```
            return runBuiltInFunction(functionName,
argumentList);
        }

        // Check that the number of arguments are correct.
        int parameterListSize =
tempFunction.getFunctionParameterList().length;
        int argumentListSize = argumentList.length;
        if (parameterListSize != argumentListSize) {
            printError("Number of arguments for function "
+ functionName + " is wrong. Expected " + parameterListSize
+ " arguments. Found " + argumentListSize + " arguments.");
            return null;
        }

        // Create a new symbol table for the function call.
        symbolTable = new AclSymbolTable(symbolTable);

        // Populate the new symbol table with values.
        for (int i = 0; i < argumentList.length; i++) {

symbolTable.setValue((tempFunction.getFunctionParameterList
())[i], argumentList[i].copy());
        }

        // Call the function.
        AclType r = null;
        try {
            r =
walker.expr(((AclFunction)tempFunction).getFunctionBody());
        } catch (Exception e) {
            e.printStackTrace();
        }

        // This line will print the symbol table of the
current function call.
        // symbolTable.print("Symbol table in function
call: " + functionName + ".");

        // Function call ends, restore parent symbol table.
        symbolTable = symbolTable.getParent();

        AclType tempReturnValue = returnValue;
        returnValue = null;

        if (tempReturnValue != null) {
            return tempReturnValue;
```

```
        } else {
            return new AclInteger(0);
        }

    }

    public void registerFunction(String functionName,
String [] parameterList, AST functionBody) {
        symbolTable.setValue(functionName, new
AclFunction(functionName, parameterList, functionBody));
    }

    public AclType runBuiltInFunction (String functionName,
AclType [] argumentList) {
        return builtInFunctions.run(functionName,
argumentList);
    }

    public void setReturnValue(AclType returnValue) {
        this.returnValue = returnValue.copy();
    }

    public boolean proceed() {
        return (!breakFlag && !continueFlag);
    }

    public void setBreakFlag() {
        breakFlag = true;
    }

    public void setContinueFlag() {
        continueFlag = true;
    }

    public void resetFlags() {
        breakFlag = false;
        continueFlag = false;
    }

    public boolean breakFlagSet() {
        return breakFlag;
    }

    public boolean continueFlagSet() {
        return continueFlag;
    }
```

```
    public AclBuiltInFunctions getBuiltInFunctions() {
        return builtInFunctions;
    }

    public void regressionTest() {
        String expectedStringValue = null;
        int expectedIntValue = 0;
        String variableName = null;

        checkString("a", "testing");
        checkString("b", "testing");
        checkInteger("c", 1);
        checkInteger("d", 1);
        checkString("e", "1 testing");
        checkString("f", "testing 1");
        checkInteger("g", 3);
        checkInteger("h", 1);
        checkInteger("i", 3);
        checkInteger("j", 12);
        checkInteger("k", 3);
        checkInteger("l", 6);
        checkInteger("m", 0);
        checkInteger("n", 9);
        checkInteger("o", 1);
        checkInteger("p", 3);
        checkInteger("q", 4);
        checkInteger("r", 1);
        checkInteger("s", 16);
        checkInteger("t", 7);
        checkInteger("u", 20);
        checkInteger("v", 20);
        checkInteger("w", 3);
        checkInteger("x", 9);
        checkInteger("y", 9);
    }

    public void checkString(String variableName, String
expectedStringValue) {
        if
(((AclString)getData(variableName)).getValue().equalsIgnore
Case(expectedStringValue)) {
            System.out.println("Variable: \"" +
variableName + "\" Expected value: \"" +
expectedStringValue + "\" found. Passed.");
        } else {
```

```
            System.out.println("Variable: \"" +
variableName + "\" Expected value: \"" +
expectedStringValue + "\" not found. Failed.");
        }
    }

    public void checkInteger(String variableName, int
expectedIntValue) {
        if (((AclInteger)getData(variableName)).getValue()
== expectedIntValue) {
            System.out.println("Variable: \"" +
variableName + "\" Expected value: " + expectedIntValue + "
found. Passed.");
        } else {
            System.out.println("Variable: \"" +
variableName + "\" Expected value: " + expectedIntValue + "
not found. Failed.");
        }
    }
}
```

```
// AclSymbolTable.java: The symbol table for ACL.
// Author: Cheow Lip Goh (Paul)

import java.util.*;

class AclSymbolTable {
    AclSymbolTable parent = null;
    HashMap<String, AclType> current = null;

    public AclSymbolTable(AclSymbolTable parent) {
        current = new HashMap<String, AclType>();
        this.parent = parent;
    }

    public AclSymbolTable getParent() {
        return parent;
    }

    public void setParent(AclSymbolTable parent) {
        this.parent = parent;
    }

    public void setValue(String keyName, AclType data) {
        current.put(keyName, data);
    }

    public AclType getValue(String keyName) {
        return (AclType)current.get(keyName);
    }

    public boolean containsValue(String keyName) {
        return current.containsKey(keyName);
    }

    public void print(String msg) {
        System.out.println("### " + msg + " ###");
        Iterator i = current.keySet().iterator();
        while (i.hasNext()) {
            String s = (String)i.next();
            AclType a = (AclType)this.getValue(s);
            if (a instanceof AclInteger) {
                System.out.println(s + " = " +
((AclInteger)a).getValue());
            } else if (a instanceof AclString) {
                System.out.println(s + " = " +
((AclString)a).getValue());
            }
```

```
            }
        }
    }
```

```
// AclType.java: The variable type base class for ACL.
// Author: Cheow Lip Goh (Paul)

public class AclType {
    protected String typeName = null;
    protected String name = null;

    public AclType copy() {
        return new AclType();
    }

    public String typeName() {
        return "unknown";
    }

    public void printError(String msg) {
        throw new AclException(msg);
    }

    public AclType plus(AclType b) {
        printError("AclType base class can't do \"+\".");
        return null;
    }

    public AclType minus(AclType b) {
        printError("AclType base class can't do \"-\".");
        return null;
    }

    public AclType times(AclType b) {
        printError("AclType base class can't do \"*\".");
        return null;
    }

    public AclType div(AclType b) {
        printError("AclType base class can't do \"/\".");
        return null;
    }

    public AclType eq(AclType b) {
        printError("AclType base class can't do \"==\".");
        return null;
    }

    public AclType ne(AclType b) {
        printError("AclType base class can't do \"!=\".");
        return null;
```

```
    }

    public AclType gt(AclType b) {
        printError("AclType base class can't do \">\".");
        return null;
    }

    public AclType lt(AclType b) {
        printError("AclType base class can't do \"<\".");
        return null;
    }

    public AclType ge(AclType b) {
        printError("AclType base class can't do \">=\".");
        return null;
    }

    public AclType le(AclType b) {
        printError("AclType base class can't do \"<=\".");
        return null;
    }
}
```

```
// AclInteger.java: The integer specialized class of
AclType for storing an integer.
// Author: Cheow Lip Goh (Paul)

public class AclInteger extends AclType {
    private int value = 0;

    public AclInteger(int value) {
        this.value = value;
    }

    public AclType copy() {
        return new AclInteger(value);
    }

    public String typeName() {
        return "integer";
    }

    public int getValue() {
        return value;
    }

    public AclType plus(AclType b) {
        if (b instanceof AclInteger) {
            return new AclInteger(value +
((AclInteger)b).getValue());
        } else if (b instanceof AclString) {
            return new AclString(value +
((AclString)b).getValue());
        } else {
            printError("b is not of the type integer or
string.");
            return null;
        }
    }

    public AclType minus(AclType b) {
        if (b instanceof AclInteger) {
            return new AclInteger(value -
((AclInteger)b).getValue());
        } else {
            printError("b is not of the type integer.");
            return null;
        }
    }
```

```
    public AclType times(AclType b) {
        if (b instanceof AclInteger) {
            return new AclInteger(value *
((AclInteger)b).getValue());
        } else {
            printError("b is not of the type integer.");
            return null;
        }
    }

    public AclType div(AclType b) {
        if (b instanceof AclInteger) {
            return new AclInteger(value /
((AclInteger)b).getValue());
        } else {
            printError("b is not of the type integer.");
            return null;
        }
    }

    public AclType eq(AclType b) {
        if (b instanceof AclInteger) {
            if (value == ((AclInteger)b).getValue()) {
                return new AclInteger(1);
            } else {
                return new AclInteger(0);
            }
        } else {
            printError("b is not of the type integer.");
            return null;
        }
    }

    public AclType ne(AclType b) {
        if (b instanceof AclInteger) {
            if (value != ((AclInteger)b).getValue()) {
                return new AclInteger(1);
            } else {
                return new AclInteger(0);
            }
        } else {
            printError("b is not of the type integer.");
            return null;
        }
    }

    public AclType gt(AclType b) {
```

```
    if (b instanceof AclInteger) {
        if (value > ((AclInteger)b).getValue()) {
            return new AclInteger(1);
        } else {
            return new AclInteger(0);
        }
    } else {
        printError("b is not of the type integer.");
        return null;
    }
}

public AclType lt(AclType b) {
    if (b instanceof AclInteger) {
        if (value < ((AclInteger)b).getValue()) {
            return new AclInteger(1);
        } else {
            return new AclInteger(0);
        }
    } else {
        printError("b is not of the type integer.");
        return null;
    }
}

public AclType ge(AclType b) {
    if (b instanceof AclInteger) {
        if (value >= ((AclInteger)b).getValue()) {
            return new AclInteger(1);
        } else {
            return new AclInteger(0);
        }
    } else {
        printError("b is not of the type integer.");
        return null;
    }
}

public AclType le(AclType b) {
    if (b instanceof AclInteger) {
        if (value <= ((AclInteger)b).getValue()) {
            return new AclInteger(1);
        } else {
            return new AclInteger(0);
        }
    } else {
        printError("b is not of the type integer.");
```

```
                return null;
            }
        }
    }
```

```
// AclString.java: The string specialized class of AclType
for storing a string.
// Author: Cheow Lip Goh (Paul)

public class AclString extends AclType {
    private String value = null;

    public AclString(String value) {
        this.value = new String(value);
    }

    public AclType copy() {
        return new AclString(value);
    }

    public String typeName() {
        return "integer";
    }

    public String getValue() {
        return value;
    }

    public AclType plus(AclType b) {
        if (b instanceof AclInteger) {
            return new AclString(value +
((AclInteger)b).getValue());
        } else if (b instanceof AclString) {
            return new AclString(value +
((AclString)b).getValue());
        } else {
            printError("b is not of the type integer or
string.");
            return null;
        }
    }
}
```

```
// AclFunction.java: The function specialized class of
AclType for storing a function.
// Author: Cheow Lip Goh (Paul)

import antlr.*;
import antlr.collections.*;

public class AclFunction extends AclType {
    String functionName = null;
    private String [] parameterList = null;
    AST functionBody = null;

    public AclFunction(String functionName, String []
parameterList, AST functionBody) {
        this.functionName = functionName;
        this.parameterList = parameterList;
        this.functionBody = functionBody;
    }

    public AclFunction(String functionName) {
        this.functionName = functionName;
    }

    public String getFunctionName() {
        return functionName;
    }

    public String [] getFunctionParameterList() {
        return parameterList;
    }

    public AST getFunctionBody() {
        return functionBody;
    }

    public AclType copy() {
        return new AclFunction(this.functionName,
this.parameterList, this.functionBody);
    }

    public boolean isBuiltInFunction() {
        return functionBody == null;
    }
}
```

```
// AclException.java: Exception and error message class for
ACL.
// Author: Cheow Lip Goh (Paul)

class AclException extends RuntimeException {

    AclException(String message) {
        System.err.println("### Error: " + message + "
###");
    }

}
```

```
// AclBuiltInFunction.java: The class implementing the
built-in functions for ACL.
// Author: Cheow Lip Goh (Paul)

import java.io.*;

public class AclBuiltInFunctions {
    private InputStream in = null;
    private BufferedReader err = null;
    private BufferedWriter out = null;
    private Process p = null;
    private AclProcessThread pt = null;
    private AclReaderThread rt = null;
    private boolean sessionOpened = false;

    public AclType run(String functionName, AclType []
argumentList) {

        String tempString = null;

        if (argumentList[0] != null) {
            if (argumentList[0] instanceof AclString) {
                tempString =
((AclString)argumentList[0]).getValue();
            } else {
                throw new AclException("Argument to " +
functionName + "() built in function is not a string!");
            }
        } else {
            throw new AclException("Argument to " +
functionName + "() built in functionis empty!");
        }

        if (functionName.equals("session")) {
            pt = new AclProcessThread(tempString);
            pt.start();
            p = pt.getProcess();
            in = p.getInputStream();
            err = new BufferedReader(new
InputStreamReader(p.getErrorStream()));
            out = new BufferedWriter(new
OutputStreamWriter(p.getOutputStream()));

            rt = new AclReaderThread(in);
            rt.start();
            sessionOpened = true;
        } else if (functionName.equals("send")) {
```

```
            send(tempString);
        } else if (functionName.equals("receive")) {
            receive(tempString);
        }

        return new AclInteger(0);
    }

    public void initialize(AclSymbolTable symbolTable) {
        symbolTable.setValue("session", new
AclFunction("session"));
        symbolTable.setValue("send", new
AclFunction("send"));
        symbolTable.setValue("receive", new
AclFunction("receive"));
    }

    public void send(String cmd) {
        try {
            out.write(cmd);
            out.newLine();
            out.flush();
            Thread.sleep(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void receive(String cmd) {
        while (true) {
            if (rt.getBuffer().contains(cmd)) {
                rt.emptyBuffer();
                break;
            }
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public void destroy() {
        if (sessionOpened) {
            try {
                Thread.sleep(3000);
                pt.stop();
```

```
                rt.stop();

                in.close();
                out.close();
                err.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
// AclProcessThread.java: A process thread class that is
incharge of creating and destroying
// a spawned process for interaction with the command line.
// Author: Cheow Lip Goh (Paul)

class AclProcessThread implements Runnable {
    private String commandLine = null;
    private Process process = null;
    private Thread processThread = null;
    private boolean isClosed = false;
    private int exitValue;

    AclProcessThread(String commandLine) {
        this.commandLine = commandLine;
    }

    public void start() {
        processThread = new Thread(this);
        try {
            process =
Runtime.getRuntime().exec(commandLine);
        } catch (Exception e) {
            e.printStackTrace();
        }
        processThread.start();
    }

    public void run() {
        try {
            process.waitFor();
            exitValue= process.exitValue();
            isClosed = true;
        } catch (Exception e) {
            // Ignore the exception and as the thread is
dying.
        }
    }

    public void stop() {
        processThread.interrupt();
        process.destroy();
        try {
            processThread.join(1000);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```
    public Process getProcess() {
        return process;
    }
}
```