TRAVEL ASSIST SCRIPTING LANGUAGE

Final Project Report

Yogi Saxena ys2332@columbia.edu

TABLE OF CONTENT

1. Introduction

Overview

There are several websites that offer quotes for air fare and other services such as lodging and rental cars. These websites do not provide tools and/or options that can be customized for such searches. The user is required to revisit the site and start a new search.

The Travel Assist Scripting Language aims to provide a simple yet powerful tool to search for latest fares/rates for desired services. The user could have the executable program run periodically and keep track of the various options.

Goals

1.2.1 Ease of Use

The main goal of this language is to keep the syntax simple so that any user without prior programming experience can use it.

1.2.2 Portable

As the end program will result in a java class file the program can be run across various operating systems such as windows, linux and Solaris.

1.2.3 Customizable

The users will have the option to either import java libraries or include the final .java file or class file in their own applications to add or enhance the features or applications.

2.0 Features/Syntax

2.1 Statement

A basic statement can contain reserved or conditional keywords, operators and form a complete instruction. A statement will always end with a semi-colon.

2.2 Data Types

2.2.1 Numeric Unsigned Integer will be used to represent quantity and date as 09012007. Also time will be represented as hours and minutes as 1824. No second's resolution.

2.2.2 Numeric Float will be used to represent the price as 256.55.

2.2.3 Boolean true or false for conditional evaluation of statements.

2.3 Reserved Keywords

Certain keywords will be reserved.

2.3.1 Search – To indicate the start of a new search.
2.3.2 Repeat – To repeat the search at a specified interval.

2.4 Conditional Keywords

2.4.1 If – else basic conditional statement.
2.4.2 For – Basic for loop.
2.4.5 While – basic while loop.

2.5 Operators

2.5.1 The following logical Operators will be supported.

Logical AND	and
Logical OR	or

2.5.2 The following Relational Operators will be supported.

Less then	<
Greater then	>
Equal to	==
Not equal to	!=
Assignment	=

2.7 Functions

2.7.1 The following functions will be standard to the language:

2.7.1.1 Print – Can be used to output results to STDOUT.

2.7.1.2 Read - Can be used to read in user data in a defined format.

2.7.1.3 Email - Can be used to email the results to an email id.

2.7.2 Functions defined by users in will be supported.
functionName (arg1, arg2);

2.8 Grammar

The following grammar will be used in the language:

;	Semi-colon to mark the end of a statement
,	Comma to separate a list of arguments in a function
( )	Left and right parenthesis to group the values of a function
[ ]	Square brackets for array representation
{ }	Curly braces for grouping a set of statements in a if else block

3.0 Operator Precedence

The operator precedence and associativity will be similar or same as the C language since its common and it's widely used.

3.2 Scanner and Parser

ANTLR 2.0 will be used to generate the scanning and parsing modules of the language.

4.0 Examples

Keyword Search followed by the airport codes to Origin and Destination followed the desired quantity and the dates.

```
Search EWR SFO 2 09012007 09052007
If ( Fare lt 250.00 )
{
        Email(ys2332@columbia.edu);
}
else
{
        Repeat;
}
```

5.0 Tutorial

5.1 Print “Hello World”

```
C = “Hello World”;
print( C );
```

The above program will create a variable C of type String and assign it the string “Hello World” and print will print “Hello World” to the screen or STDOUT. The program should be run as following:

```
java Main HelloWorld.txt.
```

5.2 Search

```
search  EWR SFO 121807 122407 2
```

The above program will result in searching a data file and printing the result to STDOUT.

java Main search.txt

5.3 User Defined Functions

```
a=2;
b=3
func foo( a, b ) {
        if ( a < b ) {
                print(a);
        }
                }
foo(a,b);
```

The above program creates a user defined function foo that take 2 integers and compares the two and returns the smaller one.

6 Language Reference Manual

There are several websites that offer a web based interface to get quotes for air fare, accommodation and other services such as rental cars. The user is required to enter several pieces information related to his/her travel plans such as dates, destination and quantity. This information is not stored and the user is required to re enter when he starts a new search. Most of the users are looking for the cheapest fare for their travel. This may require multiple searches with a combination of different dates and/or destinations.

The TASL language attempts to provide a simple scripting language that would make it easier for the travelers to do their search. The user input will be stored and used for periodic searches carried out at a pre-defined interval. If the results of the search match the user defined criteria then the results could be stored in a file and/or the user could be alerted via email.

7 Lexemes

7.1 Identifiers

Identifiers refer to or identify function names, data objects, class names etc. They consist of a string of characters such as one or more uppercase or lower case alphabets and numbers. The first character of an identifier must start with an alphabet. Upper case and lower case letters are treated differently. Keywords are reserved identifiers and cannot be re defined.

7.2 Keywords

The following reserved identifiers are used as keywords:

bool	int	void	if
else	function	while	break
return	search	repeat	email
for	float	modify	delete
reserve			

7.3 Numbers

A number is a string of digits and a decimal point. The types of Numbers supported in the language are Integer and float. An Integer consists of a sequence of digits. A floating point number consists of an integer part, a decimal point and a fraction part.

Example: 10, 100.00, 250.00

7.4 String

String is a sequence of zero or more characters. String literals are character strings surrounded by double quotes ("EWR"). String literals can include any valid character, including white-space characters and character escape sequences.

7.5 Operators

An operator is a token that specifies an operation on at least one operand, and yields some result (a value, designator, side effect or some combination). Operands are expressions or constants.

The following are the operators used in TASL.

+	-	/	*	=
= =	>	<	&	|

8 Data Types

The language defines the following data types:

Numeric: Integer represents date and quantity, float represents price.

True or False: These represent Boolean values.

9 Declarations

9.2 General declaration

All objects must be declared before use. The general syntax of a declaration is as follows:

type-specifier init-declarator-list(opt);

init-declarator-list:

declarator
init_declarator-list, declarator

The type specifiers can be int, float or boolean.

9.3 Arrays

Arrays can be declared with square brackets [ ]. The following is the syntax for declaring an array.

type-specifier declarator [constant-expression-list(opt)];

the type-specifier can be int or float.

10 Functions

10.2 Function Types

A function has the derived type “function returning type”. The type can be a data type except array or a function type. A function that does not return any value is referred to as a void function.

Functions can be defined in the following ways:

1) A function definition can create a function designator, define its parameters and their types, defined the type of its return value and construct the body of the function.
2) A function declaration specifies the properties of a function defined elsewhere.

10.3 Function Definitions

A function definition includes the body of the function. Function definitions can appear in any order, and can be referenced in one or more source files. Function definitions cannot be nested.

A function definition has the following syntax:

function-definition:
declaration-specifiers (opt) declarator declaration-list(opt) compound-statement

declaration specifiers
The declaration-specifiers (type qualifier and type-specifier) can be listed in any order. Type specifiers are: int, float and boolean.

Example:

```
main ( ) {
        search “EWR” “SFO” 10182007 10242007 2
        if ( Results( ) ) {
                email(xyz@columbia.edu);
        }
}
```

```
boolean function Results( ) {

        if ( FARE[0] < desiredValue) {
                return true;
        }
        return false;
}
```

10.4 Function Declarations

For all functions if the function definition is located after the calling function, the function must be declared before calling it.

10.5 Function Parameters and Arguments

The functions exchange information by means of parameters and arguments. The term parameter refers to any declaration within the parentheses following the function name in a function declaration or definition. The term argument refers to any expression within the parenthesis of a function call.

The following rules apply to the parameters and arguments:

1) The number of arguments in a function call must be the same as the number of parameters declared by the function definition.
2) The maximum number of arguments (and corresponding parameters) is 50 for a single function.
3) Arguments are separated by commas. The comma is not an operator in this context and the arguments can be evaluated by the compiler in any order.
4) Arguments are passed by value.
5) Modifying a parameter does not modify the corresponding argument passed by the function call.

10.6 Function invocation and return

Function Call:
A function call can be a single statement followed by a “;”.

Return Statement:
The return statement is used to return from the function at the point the return statement is specified. Return statements can return a value. The return statement is terminated by a “;”.

10.7 Built-In Functions

10.7.1 print()

Sends the output to the screen

10.7.2 load()

File I/O functions.

10.7.3 email (xxx@yyy.edu)

Emails the results to the supplied email id.

11 Expressions and Operators

11.2 Primary Expressions

Simple expressions are called primary expressions; they denote values. Primary expressions include previously declared identifiers, constants, string literals and parenthesized expressions.

Primary expressions have the following syntax:

> *primary expression:*
>   *identifier*
>   *constant*
>   *strings*
>   *(expression)*

11.2.1 Identifier

An identifier is a primary expression provided it is declared as designating an object or a function

11.2.2 Constant

A constant is a primary expression. Its type depends on its form (ie either boolean or int or float).

11.2.3 Expression

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

11.3 Postfix Expressions

Postfix expressions include array references, function calls and postfix increment and decrement expressions. The operators in postfix expressions have left-to-right associativity. Postfix expressions have the following syntax:

Postfix-expression:
> *Primary-expression*
> *postfix-expression ++*
> *postfix-expression [expression]*
> *postfix-expression (argument-expression-list)*

11.3.1 Array References

A postfix expression followed by an expression in square brackets in a postfix expression denoting a subscripted array reference.

> *array-reference:*
> *postfix-expression [expression]*

11.3.2 Function Calls

A function call is a postfix expression consisting of a function designator followed by parentheses.
The order of evaluation of any expressions in the function parameter list is undefined, but there is a sequence point before the actual call. The parentheses can contain a list of arguments (separated by commas) or can be empty.

Function calls have the following syntax:

> *Function-call:*
> *Postfix-expression(argument-expression-list(opt))*
>
> *Argument-expression-list(opt):*
> *assignment-expression*
> *argument-expression-list(opt), assignment-expression*

11.4 Unary Operators

Unary expressions are formed by combining a unary operator with a single operand. All unary operators are of equal precedence and have right-to-left associativity.
Unary-operator: one of & * + -

11.5 Binary Operators

The binary operators are categorized as follows:

- Multiplicative operators: multiplication (*) and division (/)
- Additive operators: addition (+) and subtraction (-)
- Relational Operators: less than (<), greater then (>)
- Equality operators: equality (==) and inequality (!=)
- Logical Operators: AND (&) and OR (|)

11.5.1 Multiplicative Operators:

The multiplicative operators are *, /. Operands must have the arithmetic type. Operands are converted, if necessary, according to the usual arithmetic conversion rules.

The * operator performs multiplication and the / operator performs division.

11.5.2 Additive operators:

The additive operators + and – perform addition and subtraction. Operands are converted, if necessary, according to the usual arithmetic conversion rules.

11.5.3 Relational operators:

The relational operators compare two operands and produce a result of type int. The result is 0 if the relation is false and 1 if it is true. The operators are: less than (<), greater than (>). Both operands must have an arithmetic type.

The relational operators associate from left to right.

11.5.4 Equality Operators:

The equality operator, equal (==) and not equal (! =), produce a result of type int, so that the result is 1 if both operands have the same value and 0 if they do not.

11.5.5 Logical Operators:

The logical operators are AND (&) and OR (|). These operators guarantee left-to-right evaluation. The result of the expression (of type double) is either 0 (false) or 1 (true). The operands need not have the same type, but both types must be scalar. If the compiler can make an evaluation by examining only the left operand, the right operand is not evaluated.

11.6   Assignment Operators

Assignment results in the value of the target variable after the assignment. They can be used as sub-expressions in larger expressions.

Assignment expression has two operands: a modifiable value on the left and an expression on the right. A simple assignment consists of the equal sign (=) between the two operands:

Exp1=Exp2;

The value of expression Exp2 is assigned to Exp1. The type is the type of Exp1, and the result is the value of Exp1 after completion of the operation.

12 Statements

The following types of statements are supported.

- Expression statements
- Compound statements
- Selection statements
- Iteration statements
- Break statements
- search statement
- modify statement
- delete statement
- reserve statement

12.2 Expression statement

Most statements are expression statements, which have the form

```
    expression-statement:
        expression(opt);
```

12.3 Compound statements

Several statements can be combined to form a compound statement. The body of a function is a compound statement.

```
compound-statement:
    { declaration-list(opt) statement-list (opt)}

declaration-list:
    declaration
    declaration-list declaration
statement-list:
    statement
    statement-list statement
```

12.4 Selection statements

The if statement

If expression

then
    statement

else (opt)
    statement

The statement following the control expression is executed if the value of the control expression is true. An if statement can be written with an optional else clause that is executed if the control expression is false.

12.5 The iteration statement

Iteration statements are used for looping.

iteration-statement:
    while (expression) statement
    for (expression (opt) ; expression (opt); expression (opt)) statement

In the while statement the substatement is executed repeatedly so long as the value of the expression remains unequal to 0.

In the for statement, the first expression is evaluated once, and thus specifies initialization of the loop. The second expression must be a arithmetic type and is evaluated before each iteration, and if it becomes equal to 0 the for is terminated.

12.6 Break statement

The break statement causes the termination of the enclosing while and for statements. The control is passed to the statement following the terminated statement.

12.7 Return statement

The return statement causes the program control to return to the caller. An optional expression following the return keyword will cause the function to return the lvalue of the expression of the caller. If required, the expression is converted, as if by assignment, to the type of function in which it appears.

12.8 Search statement

The search statement is used to query the database for fares. The following is the syntax:

search origin_Identifier destination_Identifier departure_date return_date quantity

12.9 Modify Statement
The modify statement is used to modify the database for reservations. The following is the syntax:

modify origin_Identifier destination_Identifier departure_date return_date quantity

12.10 Delete Statement
The delete statement is used to delete the record of a reservation in the database. The following is the syntax:

delete origin_Identifier destination_Identifier departure_date return_date quantity

12.11 Reserve Statement
The reserve statement is used to insert a reservation in the database. The following is the syntax:

reserve origin_Identifier destination_Identifier departure_date return_date quantity

13 Project Plan

13.1 Planning

There was no specific process followed since I was the only team member. The goal was try to keep the project simple and deliver the deliverables on time. Since there were no parallel programming involved I decided to not use any version control system. A script was created that would take a backup of all the required files and put it in a new directory. A note was added to indicate the milestone for the backup.

13.2 Project Style Guide

1) Use C++ naming convention.
2) Use separate files for new classes and avoid making 1 file too big.
3) Try to keep indentation to 2.
4) Put a comment where necessary to explain the logic.

13.3 Project Time Line

Date	Project Progress
09/11	Come up with a language
09/11	Refer to past languages/projects
09/18	Get familiar with ANTLR

09/25	Deliver White Paper on language TASL
10/18	Deliver Reference Manual
11/20	Work on Grammar and Walker
11/27	Grammar and Walker finished
12/07	Start working on translator/Inerpreter
12/17	Start working on project report
12/18	Deliver Project Report

13.4 Software Development Environment

The entire project was developed on SUN SOLARIS sparc.

Tools Used:

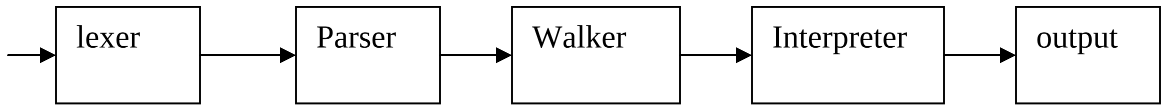
1) ANTLR: To generate the parser and scanner

Languages Used:

1) Java is used throughout to develop the TASL language.
2) Shell script: To generate test automation script and make file.

14 Architectural Design

14.1 Block Diagram



The above is a simple block diagram of the architecture. The lexer, parser and tree walker are implemented using ANTLR. The walker identifies the pattern from the input (ex: function calls) and executes the appropriate method in the class TASLInterpreter.

The TASLInterpreter takes the arguments passed by the walker as input and executes the appropriate methods invoked and returns the output to STDOUT.

TASL defines certain data Types which are wrappers to existing java data types. When a parser identifies a new data type the walker invokes the appropriate data type class and stores it in a symbol table which is a hash map.

15 Test Plan

A shell script doit was created to automate the test cases. There were simply 3 test cases.

1) This test case is used to test if the variable are initialized and stored correctly with in the symbol table.
2) This test case is used to test user defined functions.
3) This test case is used to test a simple search operation.
4) This test case is used to test operator precedence.
5) All of the test cases give the parse tree by default and report the output.

16 Lessons Learned

16.1

The following was learned during the course of developing the project:

1) Java is not friendly for executing OS level system commands.
2) ANTLR is an excellent tool for scanning and parsing once you get familiar with the grammar.
3) Developing a new language seems a possible task after doing this project/course.
4) I wish I had started working on the project earlier.

If I could work with other CVN students (pair-wise programming over the net) the project would have been significantly better.

16.2 Acknowlegement:

I would like to acknowledge the Mx Language team. I have used their code to implement the concept of TASL language.

17 File List

```
-rw-r--r--   1 yogis    yogis      28462 Dec 18 21:46
TASLAntlrLexer.java

-rw-r--r--   1 yogis    yogis       1275 Dec 18 20:17
TASLAntlrLexerTokenTypes.java

-rw-r--r--   1 yogis    yogis      44278 Dec 18 21:46
TASLAntlrParser.java

-rw-r--r--   1 yogis    yogis       1294 Dec 18 21:46
TASLAntlrTokenTypes.java
-rw-r--r--   1 yogis    yogis      20494 Dec 18 21:24
TASLAntlrWalker.java

-rw-r--r--   1 yogis    yogis       1292 Dec 18 21:24
TASLAntlrWalkerTokenTypes.java

-rw-r--r--   1 yogis    yogis       1494 Dec 18 21:27 TASLBool.java
-rw-r--r--   1 yogis    yogis       3837 Dec 18 21:28 TASLDataType.java
-rw-r--r--   1 yogis    yogis       2387 Dec 18 21:29 TASLDouble.java
-rw-r--r--   1 yogis    yogis        209 Dec 18 21:31
TASLException.java
-rw-r--r--   1 yogis    yogis       1864 Dec 18 21:31 TASLFunction.java
-rw-r--r--   1 yogis    yogis       3624 Dec 18 21:32 TASLInt.java
-rw-r--r--   1 yogis    yogis       7624 Dec 18 21:32
TASLInterpreter.java
-rw-r--r--   1 yogis    yogis        909 Dec 18 21:33 TASLString.java
-rw-r--r--   1 yogis    yogis       2759 Dec 18 21:33
TASLSymbolTable.java
-rw-r--r--   1 yogis    yogis        487 Dec 18 21:34 TASLVariable.java
-rw-r--r--   1 yogis    yogis       5621 Dec 18 21:46 grammar.g
-rw-r--r--   1 yogis    yogis       4962 Dec 18 21:24 walker.g
```

18 Appendix

File 1: grammar.g

```
/*
 * grammar.g : the lexer and the parser, in ANTLR grammar.
 */

class TASLAntlrLexer extends Lexer;

options{
    k = 2;
    charVocabulary = '\3'..'\377';
    testLiterals = false;
    exportVocab = TASLAntlr;
}

{
    int nr_error = 0;
    public void reportError( String s ) {
        super.reportError( s );
        nr_error++;
    }
    public void reportError( RecognitionException e ) {
        super.reportError( e );
        nr_error++;
    }
}

protected
ALPHA   : 'a'..'z' | 'A'..'Z' | '_' ;

protected
DIGIT   :   '0'..'9';

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

NL      : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
          { $setType(Token.SKIP); newline(); }
        ;

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

LPAREN  : '(';
RPAREN  : ')';
MULT    : '*';
PLUS    : '+';
MINUS   : '-';
```

```
RDV     : '/';
MOD     : '%';
SEMI    : ';';
LBRACE  : '{';
RBRACE  : '}';
LBRK    : '[';
RBRK    : ']';
ASGN    : '=';
COMMA   : ',';
PLUSEQ  : "+=";
MINUSEQ : "-=";
MULTEQ  : "*=";
RDVEQ   : "/=";
MODEQ   : "%=";
GE      : ">=";
LE      : "<=";
GT      : '>';
LT      : '<';
EQ      : "==";
NEQ     : "!=";
TRSP    : '\'';

ID  options { testLiterals = true; }
        : ALPHA (ALPHA|DIGIT)*
        ;

NUMBER  : (DIGIT)+ ('.' (DIGIT)*)? (('E'|'e') ('+'|'-')? (DIGIT)+)? ;

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


class TASLAntlrParser extends Parser;

options{
    k = 2;
    buildAST = true;
    exportVocab = TASLAntlr;
}


tokens {
   STATEMENT;
   VAR_LIST;
   EXPR_LIST;
   FUNC_CALL;
   FOR_CON;
   UPLUS;
   UMINUS;
}

{
    int nr_error = 0;
    public void reportError( String s ) {
        super.reportError( s );
        nr_error++;
    }
    public void reportError( RecognitionException e ) {
        super.reportError( e );
        nr_error++;
    }
}

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

statement
        : for_stmt
        | if_stmt
        | break_stmt
      | continue_stmt
        | return_stmt
        | load_stmt
        | assignment
      | print_stmt
      | search_stmt
        | func_call_stmt
        | LBRACE! (statement)* RBRACE!
            {#statement = #([STATEMENT,"STATEMENT"], statement); }
        ;

for_stmt
        : "for"^ LPAREN! for_con RPAREN! statement
        ;

for_con
        : ID ASGN! range (COMMA! ID ASGN! range)*
            {#for_con = #([FOR_CON,"FOR_CON"], for_con); }
      ;

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

break_stmt
        : "break"^ (ID)? SEMI!
        ;

continue_stmt
        : "continue"^ (ID)? SEMI!
        ;

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

load_stmt
```

```
        : "include"^ STRING SEMI!
        ;

assignment
        : l_value ( ASGN^ | PLUSEQ^ | MINUSEQ^ | MULTEQ^ | LDVEQ^
                    | MODEQ^ | RDVEQ^
                  ) expression SEMI!
        ;

print_stmt
      : "print"^ LPAREN! ID RPAREN! SEMI!
      ;

search_stmt
      : "search"^ STRING STRING NUMBER NUMBER NUMBER SEMI!
      ;

func_call_stmt
        : func_call SEMI!
        ;

func_call
        : ID LPAREN! expr_list RPAREN!
            {#func_call = #([FUNC_CALL,"FUNC_CALL"], func_call); }
        ;

expr_list
        : expression ( COMMA! expression )*
            {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
        |   /*nothing*/
            {#expr_list = #([EXPR_LIST,"EXPR_LIST"], expr_list); }
        ;

func_def
        : "func"^ ID LPAREN! var_list RPAREN! func_body
        ;

var_list
        : ID ( COMMA! ID )*
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }
        |   /* nothing*/
            {#var_list = #([VAR_LIST,"VAR_LIST"], var_list); }
        ;

func_body
        : ASGN! a:expression SEMI!
            {#func_body = #a; }
        | LBRACE! (statement)* RBRACE!
            {#func_body = #([STATEMENT,"FUNC_BODY"], func_body); }
        ;

expression
        : logic_term ( "or"^ logic_term )*
        ;

logic_term
        : logic_factor ( "and"^ logic_factor )*
        ;

logic_factor
        : ("not"^)? relat_expr
        ;

relat_expr
        : arith_expr ( (GE^ | LE^ | GT^ | LT^ | EQ^ | NEQ^)
arith_expr )?
        ;

arith_expr
        : arith_term ( (PLUS^ | MINUS^) arith_term )*
        ;

arith_term
        : arith_factor ( (MULT^ | LDV^ | MOD^ | RDV^) arith_factor )*
        ;

arith_factor
        : PLUS! r_value
          {#arith_factor = #([UPLUS,"UPLUS"], arith_factor); }
        | MINUS! r_value
          {#arith_factor = #([UMINUS,"UMINUS"], arith_factor); }
        | r_value (TRSP^)*;

r_value
        : l_value
        | func_call
        | NUMBER
        | STRING
        | "true"
        | "false"
        | LPAREN! expression RPAREN!
        ;

l_value
        : ID^ ( LBRK! index RBRK! )*
        ;

index
        : range (COMMA! range)?
          {#index = #([EXPR_LIST,"INDEX"], index); }
        ;

range
        : expression (COLON^ (expression)? | DCOLON^ expression )? |
COLON^
        ;

cl_statement
        : ( statement | func_def )
        | "exit"
          { System.exit(0); }
        | EOF!
          { System.exit(0); }
        ;
```

File 2 Walker.g

```
/*
 * walker.g : the AST walker.
 */

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

class TASLAntlrWalker extends TreeParser;
options{
    importVocab = TASLAntlr;
}

{
    static TASLDataType null_data = new TASLDataType( "<NULL>" );
    TASLInterpreter ipt = new TASLInterpreter();
}

expr returns [ TASLDataType r ]
{
    TASLDataType a, b, c, d, e;
    Vector v;
    TASLDataType[] x;
    String s = null;
    String[] sx;
    r = null_data;
}
        : #("or" a=expr right_or:.)
            {
                if ( a instanceof TASLBool )
                    r = ( ((TASLBool)a).var ? a : expr(#right_or) );
                else
                    r = a.or( expr(#right_or) );
            }
        | #("and" a=expr right_and:.)
            {
                if ( a instanceof TASLBool )
                    r = ( ((TASLBool)a).var ? expr(#right_and) : a );
                else
                    r = a.and( expr(#right_and) );
            }
        | #("not" a=expr)            { r = a.not(); }
        | #(GE a=expr b=expr)        { r = a.ge( b ); }
        | #(LE a=expr b=expr)        { r = a.le( b ); }
        | #(GT a=expr b=expr)        { r = a.gt( b ); }
        | #(LT a=expr b=expr)        { r = a.lt( b ); }
        | #(EQ a=expr b=expr)        { r = a.eq( b ); }
        | #(NEQ a=expr b=expr)       { r = a.ne( b ); }
        | #(PLUS a=expr b=expr)      { r = a.plus( b ); }
        | #(MINUS a=expr b=expr)     { r = a.minus( b ); }
        | #(MULT a=expr b=expr)      { r = a.times( b ); }
        | #(LDV a=expr b=expr)       { r = a.lfracts( b ); }
        | #(RDV a=expr b=expr)       { r = a.rfracts( b ); }
        | #(MOD a=expr b=expr)       { r = a.modulus( b ); }
        | #(UPLUS a=expr)            { r = a; }
        | #(UMINUS a=expr)           { r = a.uminus(); }
        | #(PLUSEQ a=expr b=expr)    { r = a.add( b ); }
        | #(MINUSEQ a=expr b=expr)   { r = a.sub( b ); }
        | #(MULTEQ a=expr b=expr)    { r = a.mul( b ); }
        | #(LDVEQ a=expr b=expr)     { r = a.ldiv( b ); }
        | #(RDVEQ a=expr b=expr)     { r = a.rdiv( b ); }
        | #(MODEQ a=expr b=expr)     { r = a.rem( b ); }
        | #(ASGN a=expr b=expr)      { r = ipt.assign( a, b ); }
        | #(FUNC_CALL a=expr x=mexpr)
            { r = ipt.funcInvoke( this, a, x ); }
        | num:NUMBER                 { r =
ipt.getNumber( num.getText() ); }
        | str:STRING                 { r = new
TASLString( str.getText() ); }
        | "true"                     { r = new TASLBool( true ); }
        | "false"                    { r = new TASLBool( false ); }
        | #(id:ID                    { r =
ipt.getVariable( id.getText() ); })
        | #("if" a=expr thenp:. (elsep:.)?)
            {
                if ( !( a instanceof TASLBool ) )
                    return a.error( "if: expression should be bool" );
                if ( ((TASLBool)a).var )
                    r = expr( #thenp );
                else if ( null != elsep )
                    r = expr( #elsep );
            }
        | #(STATEMENT (stmt:. { if ( ipt.canProceed() ) r =
expr(#stmt); } )*)
        | #(LOOP loopbody:.
                (loopid:ID           { s = loopid.getText(); }
                )?
           )
            {
                while ( ipt.canProceed() )
                {
                    r = expr( #loopbody );
                    ipt.loopNext( s );
                }
                ipt.loopEnd( s );
            }
        | #("break" (breakid:ID      { s = breakid.getText(); }
                  )?
           )                         { ipt.setBreak( s ); }
      | #("print" (a=expr))        { ipt.print(a); }
      | #("search" a=expr b=expr c=expr d=expr e=expr)
            { ipt.search(a,b,c,d,e); }
        | #("continue" (contid:ID    { s = contid.getText(); }
                      )?
           )                         { ipt.setContinue( s ); }
        | #("return" ( a=expr        { r = ipt.rvalue( a ); }
                    )?
           )                         { ipt.setReturn( null ); }
        | #("func" fname:ID sx=vlist fbody:.)
            { ipt.funcRegister( fname.getText(), sx, #fbody ); }
        ;
```

```
mexpr returns [ TASLDataType[] rv ]
{
    TASLDataType a;
    rv = null;
    Vector v;
}
        : #(EXPR_LIST         { v = new Vector(); }
                ( a=expr      { v.add( a ); }
                )*
           )                  { rv = ipt.convertExprList( v ); }
        | a=expr              { rv = new TASLDataType[1]; rv[0] = a; }
        |  #(FOR_CON          { v = new Vector(); }
                ( s:ID a=expr { a.setName( s.getText() ); v.add(a); }
                )+
            )                 { rv = ipt.convertExprList( v ); }
        ;

vlist returns [ String[] sv ]
{
    Vector v;
    sv = null;
}
        : #(VAR_LIST     { v = new Vector(); }
              (s:ID      { v.add( s.getText() ); }
              )*
          )              { sv = ipt.convertVarList( v ); }
        ;
```

File 3 Main.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

/**
 * The main class
 */
class Main {
    static boolean verbose = true;

    public static void execFile( String filename ) {
        try
        {
            InputStream input = ( null != filename ) ?
                (InputStream) new FileInputStream( filename ) :
                (InputStream) System.in;

            TASLAntlrLexer lexer = new TASLAntlrLexer( input );


            TASLAntlrParser parser = new TASLAntlrParser( lexer );

            // Parse the input program
            parser.program();

            if ( lexer.nr_error > 0 || parser.nr_error > 0 )
            {
                System.err.println( "Parsing errors found. Stop." );
                return;
            }


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

          verbose = true;

            if ( verbose )
            {

                // Print the resulting tree out in LISP notation
                System.out.println(
                    "================ tree structure
======================" );
                System.out.println( tree.toStringList() );
            }

            TASLAntlrWalker walker = new TASLAntlrWalker();
            // Traverse the tree created by the parser

            if ( verbose )
```

```
                System.out.println(
                    "================ program output
====================" );

            TASLDataType r = walker.expr( tree );

            if ( verbose )
            {
                System.out.println(
                    "================ variables ====================" );
                walker.ipt.symt.what();
            }

        } catch( IOException e ) {
            System.err.println( "Error: I/O: " + e );
        } catch( RecognitionException e ) {
            System.err.println( "Error: Recognition: " + e );
        } catch( TokenStreamException e ) {
            System.err.println( "Error: Token stream: " + e );
        } catch( Exception e ) {
            System.err.println( "Error: " + e );
        }

    }

    public static void commandLine() {
        InputStream input = (InputStream) new
DataInputStream( System.in );
        TASLAntlrWalker walker = new TASLAntlrWalker();

        for ( ;; )
        {
            try
            {
                while( input.available() > 0 )
                    input.read();
          }
            catch ( IOException e ) {}

            System.out.print( "TASL> " );
            System.out.flush();

            TASLAntlrLexer lexer = new TASLAntlrLexer( input );
            TASLAntlrParser parser = new TASLAntlrParser( lexer );

            try
            {
                parser.cl_statement();
                CommonAST tree = (CommonAST)parser.getAST();
                TASLDataType r = walker.expr( tree );
                if ( verbose && r != null)
                    r.print();
            } catch( RecognitionException e ) {
                System.err.println( "Recognition exception: " + e );
            } catch( TokenStreamException e ) {
                if ( e instanceof TokenStreamIOException ) {
                    System.err.println( "Token I/O exception" );
                    break;
                }
                System.err.println( "Error: Token stream: " + e );
            } catch( TASLException e ) {
                System.err.println( "Error: Interpretive: " + e );
                e.printStackTrace();
            } catch( RuntimeException e ) {
                System.err.println( "Error: Runtime: " + e );
                e.printStackTrace();
            } catch( Exception e ) {
                System.err.println( "Error: " + e );
                e.printStackTrace();
            }

        }
    }

    public static void main( String[] args ) {

        verbose = args.length >= 1 && args[0].equals( "-v" );

        boolean batch = args.length >=1 && args[0].equals( "-b" );

        if ( args.length >= 1 && args[args.length-1].charAt(0) != '-' )
            execFile( args[args.length-1] );
        else if ( batch )
            execFile( null );
        else
            commandLine();

        System.exit( 0 );
    }
}
```

File 4: TASLInterpreter.java

```java
import java.util.*;
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

/** Interpreter routines that is called directly from the tree walker.
 */
class TASLInterpreter {
    TASLSymbolTable symt;

    final static int fc_none = 0;
    final static int fc_break = 1;
    final static int fc_continue = 2;
    final static int fc_return = 3;

    private int control = fc_none;
    private String label;

    private Random random = new Random();

    public TASLInterpreter() {
        symt = new TASLSymbolTable( null, null );
    }

    public TASLDataType[] convertExprList( Vector v ) {
        /* Note: expr list can be empty */
        TASLDataType[] x = new TASLDataType[v.size()];
        for ( int i=0; i<x.length; i++ )
            x[i] = (TASLDataType) v.elementAt( i );
        return x;
    }

    public static String[] convertVarList( Vector v ) {
        /* Note: var list can be empty */
        String[] sv = new String[ v.size() ];
        for ( int i=0; i<sv.length; i++ )
            sv[i] = (String) v.elementAt( i );
        return sv;
    }

    public static TASLDataType getNumber( String s ) {
        if ( s.indexOf( '.' ) >= 0
             || s.indexOf( 'e' ) >= 0 || s.indexOf( 'E' ) >= 0 )
            return new TASLDouble( Double.parseDouble( s ) );
        return new TASLInt( Integer.parseInt( s ) );
    }

    public TASLDataType getVariable( String s ) {
        // default static scoping
        TASLDataType x = symt.getValue( s, true, 0 );
        if ( null == x )
```

```
            return new TASLVariable( s );
        return x;
    }

    public void print( TASLDataType s ) {
        // default static scoping
      if ( s instanceof TASLInt ) {
            System.out.println(+((TASLInt)s).var);
      }
      if ( s instanceof TASLDouble) {
            System.out.println(+((TASLDouble)s).var);
      }
      if ( s instanceof TASLBool) {
            System.out.println(((TASLBool)s).var);
      }
      if ( s instanceof TASLString) {
            System.out.println(((TASLString)s).var);
      }
    }

    public void search( TASLDataType Orig, TASLDataType Dst,
TASLDataType startDate,
                  TASLDataType endDate, TASLDataType numTickets) {

      String cmd="/usr/bin/grep "+((TASLString)Orig).var+" ./data.txt";
          String commandOutput = null;
          int x=255;
          try {
               Process p = Runtime.getRuntime().exec(cmd);
            BufferedReader pOutput = new BufferedReader( new
InputStreamReader(p.getInputStream()));
            String line;
            StringBuffer tmpCommandOutput = new StringBuffer();

            try
            {
                  while((line = pOutput.readLine()) != null ){
                    tmpCommandOutput.append(line).append("\n");
                  }
                  commandOutput = tmpCommandOutput.toString();
            }

            catch (IOException e) {}

            p.waitFor(); pOutput.close();
            x = p.exitValue();
          }
          catch (IOException e) {}
            catch (InterruptedException e) {}
          System.out.println(commandOutput);
    }



    public TASLDataType rvalue( TASLDataType a ) {
        if ( null == a.name )
            return a;
        return a.copy();
    }

    public TASLDataType deepRvalue( TASLDataType a ) {
        if ( null == a.name ){
            return a;
      } else {
            return a.copy();
      }
    }

    public TASLDataType assign( TASLDataType a, TASLDataType b ) {
        if ( null != a.name )
        {
            TASLDataType x = deepRvalue( b );
            x.setName( a.name );
            symt.setValue( x.name, x, true, 0 );  // scope?
            return x;
        }

        return a.error( b, "=" );
    }

    public TASLDataType funcInvoke(
        TASLAntlrWalker walker,  TASLDataType func,
        TASLDataType[] params ) throws antlr.RecognitionException {

        // func must be an existing function
        if ( !( func instanceof TASLFunction ) ){
            return func.error( "not a function" );
      }

        // otherwise check numbers of actual and formal arguments
        String[] args = ((TASLFunction)func).getArgs();
        if ( args.length != params.length )
            return func.error( "unmatched length of parameters" );

        // create a new symbol table
        symt = new
TASLSymbolTable( ((TASLFunction)func).getParentSymbolTable(),
                                   symt );

        // assign actual parameters to formal arguments
        for ( int i=0; i<args.length; i++ )
        {
            TASLDataType d = rvalue( params[i] );
            d.setName( args[i] );
            symt.setValue( args[i], d, false, 0 );
        }

        // call the function body
        TASLDataType r = walker.expr( ((TASLFunction)func).getBody() );

        // no break or continue can go through the function
        if ( control == fc_break || control == fc_continue )
            throw new TASLException( "nowhere to break or continue" );
        // if a return was called
        if ( control == fc_return )
            tryResetFlowControl( ((TASLFunction)func).name );

        // remove this symbol table and return
        symt = symt.dynamicParent();

        return r;
    }

    public void funcRegister( String name, String[] args, AST body ) {
        symt.put( name, new TASLFunction( name, args, body, symt ) );
    }

    public void setBreak( String label ) {
        this.label = label;
        control = fc_break;
    }

    public void setContinue( String label ) {
        this.label = label;
        control = fc_continue;
    }

    public void setReturn( String label ) {
        this.label = label;
        control = fc_return;
    }

    public void tryResetFlowControl( String loop_label ) {
        if ( null == label || label.equals( loop_label ) )
            control = fc_none;
    }

    public void loopNext( String loop_label ) {
        if ( control == fc_continue )
            tryResetFlowControl( loop_label );
    }

    public void loopEnd( String loop_label ) {
        if ( control == fc_break )
            tryResetFlowControl( loop_label );
    }

    public boolean canProceed() {
        return control == fc_none;
    }

    public void forNext( TASLDataType[] mexpr, TASLInt[] values ) {

        values[ values.length-1 ].var++;
        if ( control == fc_continue )
            tryResetFlowControl( mexpr[0].name );
    }

    public void forEnd( TASLDataType[] mexpr ) {
        if ( control == fc_break )
```

```
            tryResetFlowControl( mexpr[0].name );

        // remove this symbol table
        symt = symt.dynamicParent();
    }

    public TASLDataType include( TASLDataType file ) {
        if ( file instanceof TASLString )
        {
            try
            {
                InputStream input =
                    (InputStream) new
FileInputStream( ((TASLString)file).var );

                TASLAntlrLexer lexer = new TASLAntlrLexer( input );
                TASLAntlrParser parser = new TASLAntlrParser( lexer );

                parser.program();
                if ( lexer.nr_error > 0 || parser.nr_error > 0 )
                    throw new TASLException( "parsing erros" );
                CommonAST tree = (CommonAST)parser.getAST();
                TASLAntlrWalker walker = new TASLAntlrWalker();
                // share the symbol table
                walker.ipt.symt = this.symt;
                return walker.expr( tree );
            }
            catch ( Exception e )
            {
                throw new TASLException( "include failed" );
            }
        }

        throw new TASLException( "Data Type??" );
    }
}
```

File 5: TASLBool.java

```java
import java.io.PrintWriter;
/**
 * the wrapper class for boolean
 */
class TASLBool extends TASLDataType {
    boolean var;

    TASLBool( boolean var ) {
        this.var = var;
    }

    public String typename() {
        return "bool";
    }

    public TASLDataType copy() {
        return new TASLBool( var );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( var ? "true" : "false" );
    }
    public TASLDataType and( TASLDataType b ) {
        if ( b instanceof TASLBool )
            return new TASLBool( var && ((TASLBool)b).var );
        return error( b, "and" );
    }
    public TASLDataType or( TASLDataType b ) {
        if ( b instanceof TASLBool )
            return new TASLBool( var || ((TASLBool)b).var );
        return error( b, "or" );
    }
    public TASLDataType not() {
        return new TASLBool( !var );
    }

    public TASLDataType eq( TASLDataType b ) {
        // not exclusive or
        if ( b instanceof TASLBool )
            return new TASLBool( ( var && ((TASLBool)b).var )
                               || ( !var && !((TASLBool)b).var ) );
        return error( b, "==" );
    }

    public TASLDataType ne( TASLDataType b ) {
        // exclusive or
        if ( b instanceof TASLBool )
            return new TASLBool( ( var && !((TASLBool)b).var )
                               || ( !var && ((TASLBool)b).var ) );
        return error( b, "!=" );
    }
}
```

File 6: TASLDataType.java

```java
import java.io.PrintWriter;

/**
 *
 * Error messages are generated here.
 */
public class TASLDataType
{
    String name;   // used in hash table

    public TASLDataType() {
        name = null;
    }

    public TASLDataType( String name ) {
        this.name = name;
    }

    public String typename() {
      if(this.name != null)
        return this.name;
      return "unknown";
    }

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

    public void setName( String name ) {
        this.name = name;
    }

    public TASLDataType error( String msg ) {
        throw new TASLException( "illegal operation: " + msg
                               + "( <" + typename() + "> "
                               + ( name != null ? name : "<?>" )
                               + " )" );
    }

    public TASLDataType error( TASLDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new TASLException(
            "illegal operation: " + msg
            + "( <" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " and "
            + "<" + typename() + "> "
            + ( name != null ? name : "<?>" )
            + " )" );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( "<undefined>" );
```

```
}

public void print() {
    print( new PrintWriter( System.out, true ) );
}

public void what( PrintWriter w ) {
    w.print( "<" + typename() + ">  " );
    print( w );
}

public void what() {
    what( new PrintWriter( System.out, true ) );
}

public TASLDataType assign( TASLDataType b ) {
    return error( b, "=" );
}

public TASLDataType transpose() {
    return error( "\'" );
}

public TASLDataType uminus() {
    return error( "-" );
}

public TASLDataType plus( TASLDataType b ) {
    return error( b, "+" );
}

public TASLDataType add( TASLDataType b ) {
    return error( b, "+=" );
}

public TASLDataType minus( TASLDataType b ) {
    return error( b, "-" );
}

public TASLDataType sub( TASLDataType b ) {
    return error( b, "-=" );
}

public TASLDataType times( TASLDataType b ) {
    return error( b, "*" );
}

public TASLDataType mul( TASLDataType b ) {
    return error( b, "*=" );
}

public TASLDataType lfracts( TASLDataType b ) {
    return error( b, "/" );
}

public TASLDataType rfracts( TASLDataType b ) {
    return error( b, "/\'" );
    }

    public TASLDataType ldiv( TASLDataType b ) {
        return error( b, "/=" );
    }

    public TASLDataType rdiv( TASLDataType b ) {
        return error( b, "/\'=" );
    }

    public TASLDataType modulus( TASLDataType b ) {
        return error( b, "%" );
    }

    public TASLDataType rem( TASLDataType b ) {
        return error( b, "%=" );
    }

    public TASLDataType gt( TASLDataType b ) {
        return error( b, ">" );
    }

    public TASLDataType ge( TASLDataType b ) {
        return error( b, ">=" );
    }

    public TASLDataType lt( TASLDataType b ) {
        return error( b, "<" );
    }

    public TASLDataType le( TASLDataType b ) {
        return error( b, "<=" );
    }

    public TASLDataType eq( TASLDataType b ) {
        return error( b, "==" );
    }

    public TASLDataType ne( TASLDataType b ) {
        return error( b, "!=" );
    }

    public TASLDataType and( TASLDataType b ) {
        return error( b, "and" );
    }

    public TASLDataType or( TASLDataType b ) {
        return error( b, "or" );
    }

    public TASLDataType not() {
        return error( "not" );
    }
}
```

File 7: TASLDouble.java

```java
import java.io.PrintWriter;

/**
 * The wrapper class for double
 */
class TASLDouble extends TASLDataType {
    double var;

    public TASLDouble( double x ) {
        var = x;
    }

    public String typename() {
        return "double";
    }

    public TASLDataType copy() {
        return new TASLDouble( var );
    }

    public static double doubleValue( TASLDataType b ) {
        if ( b instanceof TASLDouble )
            return ((TASLDouble)b).var;
        if ( b instanceof TASLInt )
            return (double) ((TASLInt)b).var;
        b.error( "cast to double" );
        return 0;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Double.toString( var ) );
    }

    public TASLDataType uminus() {
        return new TASLDouble( -var );
    }

    public TASLDataType plus( TASLDataType b ) {
        return new TASLDouble( var + doubleValue(b) );
    }

    public TASLDataType add( TASLDataType b ) {
        var += doubleValue( b );
        return this;
    }

    public TASLDataType minus( TASLDataType b ) {
        return new TASLDouble( var - doubleValue(b) );
    }

    public TASLDataType sub( TASLDataType b ) {
        var -= doubleValue( b );
        return this;
    }

    public TASLDataType times( TASLDataType b ) {
        return new TASLDouble( var * doubleValue(b) );
    }

    public TASLDataType mul( TASLDataType b ) {
        var *= doubleValue( b );
        return this;
    }

    public TASLDataType modulus( TASLDataType b ) {
        return new TASLDouble( var % doubleValue(b) );
    }


    public TASLDataType rem( TASLDataType b ) {
        var %= doubleValue( b );
        return this;
    }

    public TASLDataType gt( TASLDataType b ) {
        return new TASLBool( var > doubleValue(b) );
    }

    public TASLDataType ge( TASLDataType b ) {
        return new TASLBool( var >= doubleValue(b) );
    }

    public TASLDataType lt( TASLDataType b ) {
        return new TASLBool( var < doubleValue(b) );
    }

    public TASLDataType le( TASLDataType b ) {
        return new TASLBool( var <= doubleValue(b) );
    }

    public TASLDataType eq( TASLDataType b ) {
        return new TASLBool( var == doubleValue(b) );
    }

    public TASLDataType ne( TASLDataType b ) {
        return new TASLBool( var != doubleValue(b) );
    }
}
```

File 8 TASLException.java

```java
/**
 * Exception class: messages are generated in various classes
 */
class TASLException extends RuntimeException {
    TASLException( String msg ) {
        System.err.println( "Error: " + msg );
    }
}
import java.io.PrintWriter;
import antlr.collections.AST;

/**
 * The function data type (including internal functions)
 *
 */
class TASLFunction extends TASLDataType {
    // we need a reference to the AST for the function entry
    String[] args;
    AST body;            // body = null means an internal function.
    TASLSymbolTable pst;   // the symbol table of static parent
    int id;              // for internal functions only

    public TASLFunction( String name, String[] args,
                       AST body, TASLSymbolTable pst) {
        super( name );
        this.args = args;
        this.body = body;
        this.pst = pst;
    }

    public TASLFunction( String name, int id ) {
        super( name );
        this.args = null;
        this.id = id;
        pst = null;
        body = null;
    }

    public final boolean isInternal() {
        return body == null;
    }

    public final int getInternalId() {
        return id;
    }

    public String typename() {
        return "function";
    }

    public TASLDataType copy() {
        return new TASLFunction( name, args, body, pst );
    }

    public void print( PrintWriter w ) {
        if ( body == null )
        {
            w.println( name + " = <internal-function> #" + id );
        }
        else
        {
            if ( name != null )
                w.print( name + " = " );
            w.print( "<function>(" );
            for ( int i=0; ; i++ )
            {
                w.print( args[i] );
                if ( i >= args.length - 1 )
                    break;
                w.print( "," );
            }
            w.println( ")" );
        }
    }

    public String[] getArgs() {
        return args;
    }

    public TASLSymbolTable getParentSymbolTable() {
        return pst;
    }

    public AST getBody() {
        return body;
    }
}
```

File 9: TASLInt.java

```java
import java.io.PrintWriter;

/**
 * the wrapper class of int
 */
class TASLInt extends TASLDataType {
    int var;

    public TASLInt( int x ) {
        var = x;
    }

    public String typename() {
        return "int";
    }

    public TASLDataType copy() {
        return new TASLInt( var );
    }

    public static int intValue( TASLDataType b ) {
        if ( b instanceof TASLDouble )
            return (int) ((TASLDouble)b).var;
        if ( b instanceof TASLInt )
            return ((TASLInt)b).var;
        b.error( "cast to int" );
        return 0;
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Integer.toString( var ) );
    }

    public TASLDataType uminus() {
        return new TASLInt( -var );
    }

    public TASLDataType plus( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLInt( var + intValue(b) );
        return new TASLDouble( var + TASLDouble.doubleValue(b) );
    }

    public TASLDataType add( TASLDataType b ) {
        var += intValue( b );
        return this;
    }

    public TASLDataType minus( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLInt( var - intValue(b) );
        return new TASLDouble( var - TASLDouble.doubleValue(b) );
    }
```

```
public TASLDataType sub( TASLDataType b ) {
    var -= intValue( b );
    return this;
}

public TASLDataType times( TASLDataType b ) {
    if ( b instanceof TASLInt )
        return new TASLInt( var * intValue(b) );
    return new TASLDouble( var * TASLDouble.doubleValue(b) );
}

public TASLDataType mul( TASLDataType b ) {
    var *= intValue( b );
    return this;
}

public TASLDataType lfracts( TASLDataType b ) {
    if ( b instanceof TASLInt )
        return new TASLInt( var / intValue(b) );
    return new TASLDouble( var / TASLDouble.doubleValue(b) );
}

public TASLDataType rfracts( TASLDataType b ) {
    return lfracts( b );
}

public TASLDataType ldiv( TASLDataType b ) {
    var /= intValue(b);
    return this;
}

public TASLDataType rdiv( TASLDataType b ) {
    return ldiv( b );
}

public TASLDataType modulus( TASLDataType b ) {
    if ( b instanceof TASLInt )
        return new TASLInt( var % intValue(b) );
    return new TASLDouble( var % TASLDouble.doubleValue(b) );
}


public TASLDataType rem( TASLDataType b ) {
    var %= intValue( b );
    return this;
}

public TASLDataType gt( TASLDataType b ) {
    if ( b instanceof TASLInt )
        return new TASLBool( var > intValue(b) );
    return b.lt( this );
}

public TASLDataType ge( TASLDataType b ) {
    if ( b instanceof TASLInt )
        return new TASLBool( var >= intValue(b) );
    return b.le( this );
    }

    public TASLDataType lt( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLBool( var < intValue(b) );
        return b.gt( this );
    }

    public TASLDataType le( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLBool( var <= intValue(b) );
        return b.ge( this );
    }

    public TASLDataType eq( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLBool( var == intValue(b) );
        return b.eq( this );
    }

    public TASLDataType ne( TASLDataType b ) {
        if ( b instanceof TASLInt )
            return new TASLBool( var != intValue(b) );
        return b.ne( this );
    }
}
```

File 10 TASLString.java
```
import java.io.PrintWriter;

/**
 * the wrapper class for string
 */
class TASLString extends TASLDataType {
    String var;

    public TASLString( String str ) {
        this.var = str;
    }

    public String typename() {
        return "string";
    }

    public TASLDataType copy() {
        return new TASLString( var );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.print( var );
        w.println();
    }

    public TASLDataType plus( TASLDataType b ) {
        if ( b instanceof TASLString )
            return new TASLString( var + ((TASLString)b).var );

        return error( b, "+" );
    }

    public TASLDataType add( TASLDataType b ) {
        if ( b instanceof TASLString )
        {
            var = var + ((TASLString)b).var;
            return this;
        }

        return error( b, "+=" );
    }
}
```

File 11: TASLSymbolTAble.java

```
import java.util.*;
import java.io.PrintWriter;

/**
 * Symbol table class: dual parent supported: static and dynamic
 */
class TASLSymbolTable extends HashMap {
    TASLSymbolTable static_parent, dynamic_parent;
    boolean read_only;

    public TASLSymbolTable( TASLSymbolTable sparent, TASLSymbolTable
dparent ) {
        static_parent = sparent;
        dynamic_parent = dparent;
        read_only = false;
    }

    public void setReadOnly() {
        read_only = true;
    }

    public final TASLSymbolTable staticParent() {
        return static_parent;
    }

    public final TASLSymbolTable dynamicParent() {
        return dynamic_parent;
    }

    public final TASLSymbolTable parent( boolean is_static ) {
        return is_static ? static_parent : dynamic_parent;
    }

    public final boolean containsVar( String name ) {
        return containsKey( name );
    }

    private final TASLSymbolTable gotoLevel( int level, boolean
is_static ) {
        TASLSymbolTable st = this;

        if ( level < 0 )
        {
            // global variable
            while ( null != st.static_parent )
                st = st.parent( is_static );
        }
        else
        {
            // local variable
            for ( int i=level; i>0; i-- )
            {
                while ( st.read_only )
                {
                    st = st.parent( is_static );
                }
                if ( null != st.parent( is_static ) )
                    st = st.parent( is_static );
                else
                    break;
            }
        }

        return st;
    }

    public final TASLDataType getValue( String name, boolean is_static,
                                        int level ) {
        TASLSymbolTable st = gotoLevel( level, is_static );
        Object x = st.get( name );

        while ( null == x && null != st.parent( is_static ) )
        {
            st = st.parent( is_static );
            x = st.get( name );
        }

        return (TASLDataType) x;
    }

    public final void setValue( String name, TASLDataType data,
                                boolean is_static, int level ) {

        TASLSymbolTable st = gotoLevel( level, is_static );
        while ( st.read_only )
        {
            st = st.parent( is_static );
        }

        st.put( name, data );
    }

    public void what( PrintWriter output ) {
        for ( Iterator it = values().iterator() ; it.hasNext(); )
        {
            TASLDataType d = ((TASLDataType)(it.next()));
            if ( !( d instanceof TASLFunction &&
((TASLFunction)d).isInternal() ) )
                d.what( output );
        }
    }

    public void what() {
        what( new PrintWriter( System.out, true ) );
    }
}
```

File 12: File TASLVariable.java

```
import java.io.PrintWriter;

/**
 * The wrapper class for unsigned variables
 */
class TASLVariable extends TASLDataType {
    public TASLVariable( String name ) {
        super( name );
    }

    public String typename() {
        return "undefined-variable";
    }

    public TASLDataType copy() {
        throw new TASLException( "Variable " + name + " has not been
defined" );
    }

    public void print( PrintWriter w ) {
        w.println( name + " = <undefined>" );
    }
}
```