ASML

Array Sorting and Manipulation Language

By
Olga Davidov
Olga.Davidov@riag.com

1. ASML: INTRODUCTION

1.1 Introduction

ASML is designed to be a simple, intuitive, and efficient language for performing mathematical manipulations on arrays. A number of existing broader domain languages and tools, such as C, C++, Java, Matlab, provide array manipulation functionality, yet many useful operations, such as sorting, shuffling, shifting are not given by these languages, and users need to either use libraries or define these operations themselves. Furthermore, such a universal tool as Matlab is oriented at a very broad domain – it deals with virtually every possible aspect in mathematics. Consequently, the amount of information a user has to absorb to learn and use such a tool is huge. While being very effective in what it does and very useful in many circumstances, it is an overkill for a user who's interested only in array-related functionality. ASML will provide most of the array manipulation functionality given in Matlab, plus add more useful operations that deal with arrays.

1.2 Simple

ASML is a simple, easy to learn, and intuitive language. It provides most of the intuitive and easy-to-understand functionality in Matlab dealing with arrays and adds a number of extra useful array-manipulation operations. ASML also provides a number of most common control flow constructs, such as loops and conditionals. ASML will be easy to learn, have easy, self-explanatory syntax. It should be readable and understandable to anyone familiar with array manipulation functionality.

1.3 Robust

Programs written in ASML are robust and reliable by virtue of simplicity of the language design. Namely, all numbers in the program are automatically treated as floating point numbers. So there are only two types of variables in the language: real numbers and arrays of real numbers. This allows preventing many error-prone situations due to type conversions. Additionally, there is extensive size checking in most of the operations. For example, if a user attempts to add two arrays of a different size, an error message will be issued.

1.3 Architecture Neutral and Portable

ASML will be an interpreted language, with its interpreter written in Java, which makes ASML architecture neutral. ASML programs can be easily seamlessly ported to any platform that has JVM running on it.

1.4 Interpreted

ASML will have a number of very high-level operation-intensive manipulations implemented as built-in functions. This justifies the need for this language to be interpreted. Another advantage of being interpreted is that commands can be evaluated and executed on the fly in the command-line interactive tool such as Matlab command-line interpreter. Commands in ASML can be executed either by calling ASML interpreter and passing it a text file with an ASML program or by calling the interpreter without arguments and executing each instruction one by one on the interpreter's command line.

1.5 High Level

ASML will implement a number of high-level operation-intensive built-in functions, such as array sorting via different algorithms, such as bubble sort, insertion sort, heap sort, selection sort, also random shuffling, shift, and other non-linear mathematical functions, such as logarithm and exponent. These high-level operations will be very useful to someone wishing to verify the end result of a complex operation implemented by hand, or say, compare the execution time of various sorting algorithms.

1.6 Model of computation

Programs written in ASML will be analyzed and interpreted by ANTLR, translated to Java code and then executed by JVM. Lexer, parser, and tree walker will be generated using ANTLR.

1.7 Data types

There are 2 data types in ASML: integer numbers and floating point numbers. Users can define arrays of floating point and integer numbers. Currently all numerals in the ASML program will be automatically cast to floating point numbers. The main concern of a user of ASML should be not on programming and typing issues, but on getting back the result of whatever operation the user is trying to perform.

1.8 Scope

All variables have to be declared prior to the body of the main program or a function where the variable is to be used. Definition of a function cannot be nested within another function. Like in C language, main program and all functions are declared on top level. A function body opens a new scope. Main program variables or variables from the previously active scope are not visible in the current local scope.

1.9 Error Handling

All lexical and syntactic errors are caught by the lexer and parser constructed by ANTLR from the lexer and parser grammars. The translator will check the sizes of the arrays used in per-element operations, such as addition of 2 arrays and report an error if sizes do not match. Also, an error will be reported if a scalar is being used where an array variable is expected, and vice versa.

1.10 Control Flow Structures

ASML supports most common control flow structures: looping constructs – while loop and for loop and conditional – if – elsif – else statement.

1.11 Example of Syntax

```
program test
float a[5];
int i, mn;
{
        print("Enter 5 numbers:");
        i = 0;
        while (i<length(a)) {
                read(a[i]);
```

```
                i = i+1;
        }

        printArray(a);

        i = 0;
        while (i<length(a)) {
                mn = min(a, i, length(a));
                swap(a, i, mn);
                i = i + 1;
        }

        printArray(a);

}
```

The example above shows the program called “test” that declares array a of 5 floating point numbers and 2 integer variables i and mn. First we output a prompt to enter 5 numbers and we store these numbers in the array a. Then the array is sorted using operators min and swap, and finally, we print out the sorted array.

1.12 Possible extensions

Currently user-defined functions can accept only scalar arguments by value. A valuable extension will be the ability to pass argument by name, as well as be able to pass array-type arguments.

Another extension is ability to declare global variables. These global variables would be accessible from a local function scope.

Also, increment and decrement operators should be added, since their usage fits naturally in array manipulation domain.

2. ASML: TUTORIAL

ASML resembles simplified syntax of C language. First, the main program is defined, followed by its declarations, followed by the main program body. After the main program any number of user-defined functions can be defined. The following is a “hello world” program:

```
program hello
{
     print(“Hello ASML world!”);
}
```

The following slightly more complicated program that shows the usage of user-defined function to compute the factorial:

```
program abc
int value, f;
{
        print("This is a simple ASML program");
        print("Enter a value:");
        read(value);
        f = fact(value);
        print("Factorial of ", value, " is ", f);
}

int fact(int x)
{
        if (x <= 1) {
                return 1;
        }
        else {
                return x*fact(x-1);
        }
}
```

In ASML program semi-colon is used as the statement terminator. A block of statements is wrapped in curly braces. All declarations should be placed after the program or function definition and before the opening curly brace of the program/function body.

The motivation of ASML language is to make array manipulation easier and expressed more naturally. The following example shows an example of using language operators to sort an array:

```
program sort
```

```
float a[5];
int i, mn;
{
        print("Enter 5 numbers:");
        i = 0;
        while (i<length(a)) {
                read(a[i]);
                i = i+1;
        }

        printArray(a);

        i = 0;
        while (i<length(a)) {
                mn = min(a, i, length(a));
                swap(a, i, mn);
                i = i + 1;
        }

        printArray(a);

}
```

The operators used in the program above are

min – takes array and 2 indexes and returns the index of the minimum value between the specified indexes in the given array

swap – takes an array and 2 indexes and swaps the elements with these indexes in the specified array.

3. Language Reference Manual

3.1 Introduction

ASML is a small language for array manipulation based on C. It uses ANTLR to specify its grammar and Java on the back end to implement the grammar.

3.2 Tokens

There are five kinds of tokens: identifiers, keywords, operators, literals, and other separators. Spaces, new lines and tabs constitute white space. The purpose of white space is to separate tokens, otherwise it is ignored. A token is a longest possible sequence of characters that may be matched to any token.

3.2 Comments

Both C and C++ style comments are supported. Multiline comments start with /* and end with */ sequence of characters. Single line comments start with // and continue to the end of the line.

3.3 Identifiers

Identifier is a sequence of letters and digits, which start with a letter. The first 100 symbols are significant.

3.4 Keywords

The following keywords are reserved in ASML:

break
const
continue
else
elsif
float
if
int
length
max
min

print
printArray
program
read
return
swap
while

3.5 Literals

There are integer literals, floating point literals and string literals. Integer literal is a sequence of one or more digits, optionally signed. Floating point literal is also optionally signed sequence of digits followed by a dot, followed by another sequence of digits. A string literal is a sequence of zero or more characters enclosed in double quotes. Double quote followed by another double quote is used to represent literal double quote character inside a string.

3.6 Types

ASML has 3 basic scalar types: int, float, string. Int and float types are equivalent to those in C, and string type is equivalent to char array in C (char[])

The only user-defined type is array type. Only one-dimensional arrays are allowed. Arrays can be indexed only by integers.

When arithmetic operations are performed on operands of integer and float types, the integer operand is converted to float.

The string type is only used for built-in I/O functions.

3.7 Operator Precedence

The following table lists operators in ASML in order of decreasing precedence.

[] ()	Array element dereference, function call
!	Logical negation
- +	Unary adding operators
* /	Binary multiplicative operators
- +	Binary additive operators
== < <= > >=	Relational operators
&& ||	Logical operators
=	Assignment operator

3.8 Declarations

Variables and constants must be declared before they are used. A variable declaration has the following form:
<type> <list of optionally initialized identifiers>;
For example:
int i, j;
float x, y;
int A[10];
float q = 3.0;

A constant declaration has the following form:

const <type> <list of initialized identifiers>;
const float pi = 3.1415;

User-defined types are arrays of built-in types (int and float). User-defined types are declared as follows:

type myType float[10];
type intArray int[50];

User-defined type must be declared before a variable of that type can be declared.

3.9 Statements

Statements can be either simple or compound. A simple statement is an expression; compound statement is a sequence of statements in curly braces:

```
{
        statement
        …
        statement
}
```

3.9.1 Conditional statement

The following id the syntax for the if statement:

```
if (expression)
        statement
elsif (expression)
```

```
        statement
….
else
        statement
```

3.9.2 Looping statement

The following is the syntax for the while loop:

```
while (expression)
{
        statement
}
```

3.9.3 Return statement

Return statement is used to return control from a function to the caller. In ASML every function must return a value. Return statement has the following form:
return expression;

3.9.4 Continue and Break statements

Continue statement stops execution of the current iteration of a loop and jumps to the beginning of the loop and evaluates the loop condition. Break statement is used to stop execution of the loop entirely and passes the control to the first statement outside the loop.

3.9.5 I/O statements

The following statements are used for I/O:

read(var_name) – reads a numeric value from standard input stream and stores it in the specified variable

print(“string literal”, expression) – takes a comma separated list of string literals and expressions and prints it out in the specified sequence on standard output.

printArray(array_name) – prints out space separated elements of the specified array on standard output

3.10 Scope

The scope of an identifier is limited to the module (main program or function) where that identifier is declared. When a function is called a new scope is created and the previous scope is not accessible until the function returns. Currently global identifiers are not supported. Recursion is allowed. In this case a new scope is created each time a function is called. Functions cannot be nested. Main program and all functions are defined on the top level.

3.11 Built-in Functions

Three functions are used for I/O: print, printArray and read.

print function takes a list of arguments; Each argument is converted to a string and they are concatenated together and printed out to standard output.

printArray function prints out elements of given array separated by spaces

read function accepts only int or float argument and attempts to read a value from standard input into specified variable.

The following array-manipulation functions are predefined:
min - returns index of min element
max - returns index of max element
length - returns array length
swap - swaps 2 elements with given indices

3.12 Functions

Functions are the only kind of subroutines supported in ASML. Functions can accept parameters and must return a value. Parentheses after the function name are required even if the parameter list is empty. Both return value and parameters must be of simple types: int or float. Currently parameters can only be passed by value.

3.13 Sample Program

```
program sample
float a[5], x;
int i, mn;
{
```

```
        print("Enter 5 numbers:");
        i = 0;
        while (i<length(a)) {
                read(a[i]);
                i = i+1;
        }

        printArray(a);

        i = 0;
        while (i<length(a)) {
                mn = min(a, i, length(a));
                swap(a, i, mn);
                i = i + 1;
        }

        printArray(a);

        print("Enter a value:");
        read(x);
        i = 0;
        if (x < 0) {
           x = -x;
        }

        while (i < x) {
           if (i > 20) {
              break;
           }
           elsif (i > 15) {
              i = i + 1;
              continue;
           }
           else {
              print("factorial of ", i, " is ",
factIter(i));
              i = i + 1;
           }
        }
        print ("i = ", i);

}

int factIter(int x)
int i, r;
{
        i = 1;
```

```
        r = 1;
        while (i <= x) {
           r = r*i;
           i = i+1;
        }
        return r;
}
```

4. PROJECT PLAN

4.1 Planning and specification

Timing of the project was largely dictated by the course schedule. Specification and design decisions were mostly done during the white paper composition and were finalized when Language Reference Manual was created. Minor on-going modifications to these decisions were introduced during the development phase.

4.2 Development and testing

Development was done in phases that resemble the logical phases of the compiler. Each phase was followed by a testing phase. First the lexer was written. Then parser was added to the grammar file. The grammar was run against a testing suit that made sure that only syntactically correct programs are accepted. Then Parser was modified to generate an AST. The tree was printed at that point to check that the desired tree structure is built. Then the tree walker was added to the grammar file, and instead of the actions, placeholders where used, which just printed a message of what the action would do. This made sure that the tree walker rules are actually correctly recognizing the tree structure. At the last stage, the actual actions were written and the corresponding java classes needed for those actions were written. Each stage was followed by adding testing cases to the testing suit and running the regression tests.

4.3 Programming Style Guide

The following is the programming style guide used in the project:

1. use descriptive names for identifiers
2. most critical sections of the code should have descriptive comments
3. every logically complete modification or addition to functionality should be version-controlled. Version control comments should explicitly specified what the change was
4. every module should include a brief description of its overall functionality
5. every file should state its name, purpose, and author name in the header
6. nested blocks of statements should be evenly indented and statements on the same level should be indented by the same amount of spaces.

4.4 Project timeline

The following is the project timeline table:

Task	Date due
White paper	09/23/03

Grammar specifications finalized	10/05/03
Syntax specifications finalized	10/26/03
Language Reference Manual	10/28/03
Lexer and Parser grammar	11/04/03
Semantic analysis (tree walker)	11/12/03
Interpreter backend	11/23/03
Final tests and fixes	12/07/03
Code complete	12/14/03
Final Report	12/18/03

4.5 Development environment

All development was done on Unix Solaris 7, using telnet sessions. SCCS was used as the version control system for the project. Lexer, parser and tree walker were generated by ANTLR tool using Java version 1.3.0. Backend of the interpreter was written in java, also using java version 1.3.0.

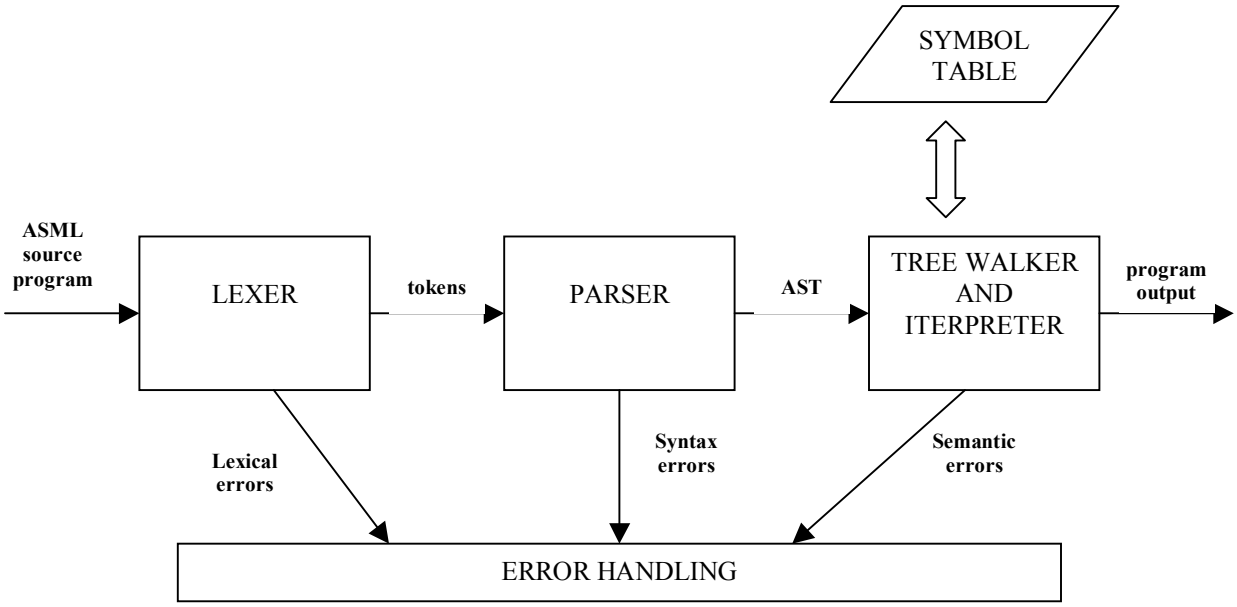
4.6 Project log

The actual tasks completed and their completion dates are shown in the following table:

Task	Date completed
White paper	09/23/03
Grammar specifications finalized	10/10/03
Syntax specifications finalized	10/28/03
Language Reference Manual	10/28/03
Lexer and Parser grammar completed	11/06/03
Parser modified to produce AST	11/11/03
Tree walker written (no actions)	11/11/03
Actions added + auxiliary java classes	11/23/03
Backend finalized	12/10/03
Final testing and fixes completed	12/16/03
Final report	12/18/03

5. ARCHITECTURAL DESIGN

The following is the block-diagram of the ASML interpreter:



There are three components in ASML interpreter: lexer, parser, and tree walker. The input source program is scanned by lexer and broken up into lexical tokens. Lexical errors are reported at this point. Next, the parser reads the stream of tokens and builds up an AST. At this time syntactical errors are reported. At the last stage the tree walker walks the syntax tree produced by the parser and interprets each node of the tree. At this point identifiers are associated with their corresponding entities in the symbol table, context switches are done during function calls, etc. Semantic errors are reported during the tree walking as well.

6. TEST PLAN

Testing was one of the major considerations during the project development. Small testing source programs were developed each time a new piece of functionality was added to the interpreter. For example, when the functionality for the "if" statement was added, the testing source program that uses "if" statements was added to the testing suit.

Here is the testing source program that was used to test the "continue" statement functionality:

```
program continueTest1
int i, j;
float k;
float a[10], b[5];
{
   i = 0;
   while (i<5) {
        if (i==1) {
           i=i+1;
           continue;
        }
        print("i = ", i);
        j=0;
        while (j<3) {
           if (i==3 && j==1) {
              j=j+1;
              continue;
           }
           print ("        j = ", j);
           j = j + 1;
        }
        i = i + 1;
   }
}
```

The following program was used to test the "swap" statement functionality:

```
program swapTest1
int i, j;
float k;
float a[10], b[5];
{
        a[4] = 10;
        a[7] = 17;
        print (a[4], " ", a[7]);
        swap(a, 4, 7);
        print (a[4], " ", a[7]);
```

```
        printArray(a);
        print (length(a), " " , length(c));
}
```

The following are 3 of the scripts that were used to test the declarations functionality:

```
program declsTest1
int i;
float a;
int x[3];
{
     i = 5;
     a = 3.89;
     x[0] = 4;
}


program declsTest2
int i;
float a;
int x[3];
{
     j = 5;
     a = 3.89;
     x[0] = 4;
}


program declsTest3
int i;
float a;
type abc int[3];
abc x;
{
     i = 5;
     a = 3.89;
     x = 4;
}
```

These test cases were chosen so that the interpreter produces the correct output on the valid syntax and semantics and produces error on incorrect program. Generally, there were several test cases used for each piece of functionality: one intended to reveal valid program and several to produce different kinds of errors. For example, in the lest testing case declsTest3, the problem is that x is an array and it is assigned value 4, which is a scalar. The expected output produced by the interpreter is as follows:

```
Error: cannot assign to x
Exception: java.lang.Exception: Error: x is an array
```

During the unit testing of each piece of functionality the interpreter was run against one testing script at a time. At this time the needed output file was generated for each testing case. At the integration testing the testing source program along with the sample output file was added to the testing program testASML.sh. This testing program executed each script at a time, comparing its output to the corresponding sample output file. If the differences were found, the program printed out those differences to the standard output.

7. LESSONS LEARNED

I have learned a couple of important things during this project:

- When there are new tools to be used, allow more time to learn to use the tool then you would otherwise do. One of the major challenges for me was to understand the functionality of ANTLR, and I realize that foreseeing this early in the project would make it easier for me to finish the project.

- Working on a project on your own is definitely significantly easier than working with a group of people. It greatly reduces coordination and synchronization time to get people to agree on a certain decision and most importantly, make pieces work together. On the other hand, there is also a drawback: you allow yourself to be more sloppy and take more shortcuts, that you wouldn't do if you worked in a group and had other people depending on you. This sloppiness often leads to many problems in the code and in timing of the project that otherwise would not happen.

- Using version control and commenting my code was not a new thing for me in this project. These issues once again proved to be greatly important in successfully completing the work, as several times I had to go back to previous versions when a new functionality broke something that was working before.

- Overall, I consider this to be a fairly successful project, as I finally got most of the functionality to actually work, even though there are several features that definitely belong to this project that I did not get to implement. But the important thing is that I know how to make them work, it's just a matter of time. I learned a great deal about how a compiler works. It also made me understand several things about other compilers that I'm using, since now I understand why certain things work the way they do.

8. APPENDIX

```
// asml.g
header {

        // Grammar file for ASML language
        // Modules: Lexer, Parser, Tree walker
        // W4115
        // Olga Davidov
        // olga.davidov@riag.com
}

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



//----------------------------------------------------------------------------
// The ASML Parser
//----------------------------------------------------------------------------
class ASMLRecognizer extends Parser;
options {
                        exportVocab = ASML;
                        defaultErrorHandler = true;
                        k = 2;
                        buildAST=true;
}

tokens {
      DECLS;
      STMTS;
      VARDECL;
      CONSTDECL;
      FUNCDEFS;
      SCALAR;
      ARRAY;
      CASE;
      FNCALL;
      PARAMS;
}


{
 public static void main(String[] args)
 {
        try {
                if (args.length > 0 ) {
                        for(int i=0; i< args.length;i++) {
                                System.err.println("Parsing " + args[i]);
                                parseFile(new FileInputStream(new File(args[i])));
                        }
                 }
```

```
                else {
                        System.err.println("Usage: java ASMLRecogizer <list of files to be parsed>");
                }
        }
        catch(Exception e) {
                System.err.println("exception: "+e);
                //e.printStackTrace(System.err);
        }
 }


 public static void parseFile(InputStream s) throws Exception {
  try {
        ASMLLexer lexer = new ASMLLexer(s);
        ASMLRecognizer parser = new ASMLRecognizer(lexer);
        parser.program();
        AST t = parser.getAST();
        //System.out.println(t.toStringTree());
        ASMLTreeParser treeParser = new ASMLTreeParser();
        treeParser.program(t);
  }
  catch (Exception e) {
        System.err.println("parser exception: "+e);
        //e.printStackTrace();
  }
 }
}

// Top level rules
//---------------------------------------------------------------------------

program :
                                "program"^ IDENT!
                                subprogramBody
                                funcDefinitions;



subprogramBody :
                                declarations
                                LCURLY!
                                        statements
                                RCURLY!;


declarations :
                                (declaration)* { #declarations = #([DECLS, "decls"], #declarations); };

declaration :
                                varDeclaration |
                                constDeclaration |
                                typeDeclaration ;


statements :
                                (statement)* { #statements = #([STMTS, "stmts"], #statements); };
```

```
funcDefinitions :
                                        (funcDefinition)* { #funcDefinitions = #([FUNCDEFS, "funcDefs"],
#funcDefinitions); };


// Declarations
//---------------------------------------------------------------------------


varDeclaration :
        (("int"|"float") IDENT LPAREN) => funcDeclaration |
                                        typeName varDeclarationItem (COMMA! varDeclarationItem)* SEMI!
                                        { #varDeclaration = #([VARDECL, "varDecl"], #varDeclaration); };

constDeclaration :
                                        "const"! typeName constDeclarationItem (COMMA!
constDeclarationItem)* SEMI!
                { #constDeclaration = #([CONSTDECL, "constDecl"], #constDeclaration); };

typeDeclaration : "type"^ IDENT ("int"|"float") LBRACKET! INT_LITERAL RBRACKET! SEMI!;


funcDeclaration : ("int"|"float") IDENT^ LPAREN! (formalParameters)? RPAREN! SEMI!;


varDeclarationItem:
            IDENT^ (ASSIGN! constValue)? {#varDeclarationItem = #([SCALAR, "scalar"],
#varDeclarationItem); } |
            IDENT^ LBRACKET! INT_LITERAL RBRACKET! {#varDeclarationItem = #([ARRAY,
"array"], #varDeclarationItem); };

constDeclarationItem :
            IDENT^ ASSIGN! constValue;


constValue :
                                        INT_LITERAL |
                                        FLOAT_LITERAL;


typeName :      IDENT |
                                        "int" |
                                        "float";


// functions
//---------------------------------------------------------------------------

funcDefinition :
                                        typeName IDENT^ LPAREN! (formalParameters)? RPAREN!
                                        subprogramBody;

formalParameters :
```

```
                                        parameterSpec (COMMA! parameterSpec)*
                                        {#formalParameters = #([PARAMS, "params"], #formalParameters); };

parameterSpec : typeName IDENT^;


// Statements
//---------------------------------------------------------------------------

statement :
                                        assignmentStatement |
                                        breakStatement |
                                        continueStatement |
                                        returnStatement |
                                        ifStatement |
                                        whileStatement |
                                        ioStatement |
                                        arrayStatement;



assignmentStatement : variableReference ASSIGN^ expression SEMI!;

breakStatement : "break"^ SEMI!;

continueStatement : "continue"^ SEMI!;


returnStatement : "return"^ expression SEMI!;


ifStatement :
                                        "if"^ ifpart ( "elsif"! ifpart )* ( "else"! LCURLY! statements RCURLY!
)?;

protected ifpart :
                                        LPAREN! expression RPAREN! LCURLY! statements RCURLY!
                                        {#ifpart = #([CASE, "case"], #ifpart); };

whileStatement : "while"^ LPAREN! expression RPAREN!
                                                        LCURLY! statements RCURLY!;

variableReference: IDENT^ (LBRACKET! expression RBRACKET!)?;

ioStatement :
                                        "print"^ LPAREN! printItem (COMMA! printItem)* RPAREN! SEMI!
|
                                        "printArray"^ LPAREN! IDENT RPAREN! SEMI! |
                                        "read"^ LPAREN! variableReference RPAREN! SEMI!;

printItem :
                                        expression |
                                        STRING_LITERAL;

arrayStatement : "swap"^ LPAREN! IDENT COMMA! expression COMMA! expression RPAREN!
SEMI!;
```

```
// Expressions
//---------------------------------------------------------------------------


primitiveElement :
                                        constValue |
                                        (IDENT LPAREN) => functionCallExpression |
                                        variableReference |
                                        arrayExpression |
                                        LPAREN! expression RPAREN!;

arrayExpression
        : ("max" LPAREN IDENT RPAREN) => "max"^ LPAREN! IDENT RPAREN!
        | ("max" LPAREN IDENT COMMA expression COMMA expression RPAREN) =>
                "max"^ LPAREN! IDENT COMMA! expression COMMA! expression RPAREN!
        | "max"^ LPAREN! expression COMMA! expression RPAREN!
        | ("min" LPAREN IDENT RPAREN) => "min"^ LPAREN! IDENT RPAREN!
        | ("min" LPAREN IDENT COMMA expression COMMA expression RPAREN) =>
                "min"^ LPAREN! IDENT COMMA! expression COMMA! expression RPAREN!
        | "min"^ LPAREN! expression COMMA! expression RPAREN!
        | "length"^ LPAREN! IDENT RPAREN!
        ;



functionCallExpression : IDENT LPAREN! (actualParameters)? RPAREN!
        {#functionCallExpression = #([FNCALL, "fncall"], #functionCallExpression);};

actualParameters :
        expression (COMMA! expression)*;

booleanNegationExression : (NOT^)* primitiveElement;

signExpression : (PLUS^|MINUS^)* booleanNegationExression;

multiplyingExpression : signExpression ( (MUL^|DIV^) signExpression )*;

addingExpression : multiplyingExpression ( (PLUS^|MINUS^) multiplyingExpression )*;

relationalExpression : addingExpression ( (EQUALS^|NOT_EQUALS^|GT^|GTE^|LT^|LTE^)
addingExpression )*;

expression : relationalExpression ( (AND^|OR^) relationalExpression )*;



//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------


//-----------------
// The ASML scanner
//-----------------
class ASMLLexer extends Lexer;
```

```
options {
   charVocabulary = '\0'..'\377';
            exportVocab=ASML;
            testLiterals=false;
            k=2;
}




// Whitespace -- ignored
WS
  : ( ' '
    | '\t'
    | '\f'

    | ( "\r\n"
      | '\r'
      | '\n'
      )
      { newline(); }
    )

    { $setType(Token.SKIP); }
  ;


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

// Literals
INT_LITERAL : ('0'..'9')+ (('.' ('0'..'9')*) => '.' ('0'..'9')* {$setType(FLOAT_LITERAL);})?;


STRING_LITERAL
  : '"'!
    ( '"' '"'!
    | ~('"'|'\n'|'\r')
    )*
```

```
   ( ""!
   | // nothing -- write error message
   )
  ;



// Operators
NOT        : '!';
OR         : "||";
AND        : "&&";
DOT        : '.'  ;
ASSIGN     : "=" ;
SEMI       : ';'  ;
COMMA      : ','  ;
EQUALS     : "=="  ;
LBRACKET   : '['  ;
RBRACKET   : ']'  ;
LPAREN     : '('  ;
RPAREN     : ')'  ;
LCURLY     : '{'  ;
RCURLY     : '}'  ;
NOT_EQUALS : "!=" ;
LT         : '<'  ;
LTE        : "<=" ;
GT         : '>'  ;
GTE        : ">=" ;
PLUS       : '+'  ;
MINUS      : '-'  ;
MUL        : '*'  ;
DIV        : '/'  ;


IDENT
  options {testLiterals=true;}
  : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
  ;







//----------------------------------------------------------------------------
// The ASML Tree walker
//----------------------------------------------------------------------------
{
        import java.io.*;
        import java.util.*;
}

class ASMLTreeParser extends TreeParser;

options {
   importVocab=ASML;
}
```

```
{
        UserTypeArray userTypes = new UserTypeArray();
  SymbolTable st = new SymbolTable();
        int inWhile = 0;
  int breakMode = 0;
  int continueMode = 0;
  AST functions;
  int inFunction = 0;
  double returnValue = 0;
  int returnMode = 0;
}

// top level
//------------------------------------------------------------------------

program:
        #("program" declarations s:STMTS f:FUNCDEFS)
        {
                functions = f;
                statements(s);
        };

// declarations
//------------------------------------------------------------------------

declarations:
        #(DECLS (varDeclaration | constDeclaration | typeDeclaration | funcDeclaration)*);

localDeclarations:
        #(DECLS (varDeclaration | constDeclaration)*);

varDeclaration
        {
                double init = 0;
        }:
        #(VARDECL type:typeName ({init=0;}#(SCALAR #(var:IDENT (init=constValue)?))
        {
                try {
                        if (!type.getText().equals("int") && !type.getText().equals("float")) {
                                if (init!=0) {
                                        System.out.println("Warning: variable "+var+" cannot be
initialized");
                                }
                                else {
                                        int baseSize = userTypes.getSize(type.getText());
                                        String baseType = userTypes.getBaseType(type.getText());
                                        st.addArray(var.getText(), baseType, baseSize);
                                }
                        }
                        else {
                                st.addVar(var.getText(), type.getText(), init);
                        }
                }
                catch (Exception e) {
                        System.err.println("Error: cannot declare symbol " + var);
```

```
                        System.err.println("Exception: " + e);
                }
        }
 | #(ARRAY #(array:IDENT size:INT_LITERAL))
        {
                if (!type.getText().equals("int") && !type.getText().equals("float")) {
                        System.out.println("Warning: cannot declare array of type "+type);
                }
                else {
                        try {
                                st.addArray(array.getText(), type.getText(),
Integer.parseInt(size.getText()));
                        }
                        catch (Exception e) {
                                System.err.println("Error: cannot declare symbol " + array);
                                System.err.println("Exception: " + e);
                        }
                }
        }
        )*);

constDeclaration
        {
                double init = 0;
        }:
        #(CONSTDECL type:typeName (#(var:IDENT init=constValue)
        {
                try {
                        if (!type.getText().equals("int") && !type.getText().equals("float")) {
                                System.out.println("Warning: cannot declare constant of type "+type);
                        }
                        else {
                                st.addConst(var.getText(), type.getText(), init);
                        }
                }
                catch (Exception e) {
                        System.err.println("Error: cannot declare symbol " + var);
                        System.err.println("Exception: " + e);
                }
        }
)*);

typeDeclaration
        {
                String type = "";
                int size = 0;
        }:

        #("type" name:IDENT ("int" { type = "int"; }
         |"float" { type = "float"; }
              ) s:INT_LITERAL)
              {
                        size = Integer.parseInt(s.getText());
                        userTypes.add(name.getText(), type, size);
              };
```

```
funcDeclaration:
	#(IDENT ("int"|"float") (#(IDENT typeName))*) {};



typeName: "int" | "float" | IDENT;

constValue returns [double r]
	{
		r = 0;
	}
	: i:INT_LITERAL {r=Integer.parseInt(i.getText());}
	| f:FLOAT_LITERAL {r=Double.parseDouble(f.getText());};



// Statements
//------------------------------------------------------------------------


statements: #(STMTS (statement)*);


statement
		{
			double index = 0;
			double expr, a, b = 0;
			int isArray = 0;
			int done = 0;
			String out = "";

			if (breakMode == 1 || continueMode == 1 || returnMode == 1) {
				_t = _t.getNextSibling();
				_retTree = _t;
				return;
			}
		}
:
	#(ASSIGN
		(#(var:IDENT (index=expression {isArray=1;})?) expr=expression)
		{
			try {
				if(isArray==1) {
					st.set(var.getText(), (int)index, expr);
				}
				else {
					st.set(var.getText(), expr);
				}
			}
			catch (Exception e) {
				System.err.println("Error: cannot assign to " + var);
				System.err.println("Exception: " + e);
			}
		})
	|
```

```
        #("if" (#(CASE expr=expression case_part:STMTS)
              {
                if (done == 0 && expr == 1) {
                    statements(#case_part);
                    done = 1;
                }
              }
              )* (else_part:STMTS
              {
                if (done == 0) {
                    statements(#else_part);
                }
              })?)
        |
        #("while" cond:. body:.)
              {
                inWhile++;
                while (expression(#cond) == 1 && breakMode == 0) {
                  statements(#body);
                  continueMode = 0;
                }
                inWhile--;
                breakMode = 0;
              }
        |
        "break"
              {
                if (inWhile == 0) {
                  System.out.println("Error at break: not in while statement");
                }
                else {
                    breakMode = 1;
                }
              }
        |
        "continue"
              {
                if (inWhile == 0) {
                  System.out.println("Error at continue: not in while statement");
                }
                else {
                    continueMode = 1;
                }
              }
        |
        #("return" expr=expression)
                    {
                            if (inFunction == 0) {
                                    System.out.println("Error: return statement is not allowed in the main
program");
                            }
                            else {
                                    returnValue = expr;
                                    returnMode = 1;
                            }
                    }
```

```
        |
        #("print"
                (expr=expression {out += String.valueOf(expr);} | s:STRING_LITERAL {out += s;})*)
                {System.out.println(out);}
        |
        #("printArray" array:IDENT)
                {
                        try {
                                if(!st.isArray(array.getText())) {
                                        System.out.println("printArray: array expected");
                                }
                                else {
                                        st.printArray(array.getText());
                                }
                        }
                        catch (Exception e) {
                                System.err.println("Error at printArray");
                                System.err.println("Exception: " + e);
                        }
                }
        |
        #("read"
          #(var1:IDENT (index=expression {isArray=1;})?))
          {
            try {
              BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
              double value = Double.valueOf(in.readLine()).doubleValue();
                                if(isArray==1) {
                                        st.set(var1.getText(), (int)index, value);
                                }
                                else {
                                        st.set(var1.getText(), value);
                                }
                        }
                        catch (Exception e) {
                                System.err.println("Error: cannot read " + var1);
                                System.err.println("Exception: " + e);
                        }
                }
  |
  #("swap" var2:IDENT a=expression b=expression)
    {
      try {
        if(!st.isArray(var2.getText())) {
          System.out.println("swap: array expected");
        }
        else {
          st.swap(var2.getText(), (int)a, (int)b);
        }
      }
      catch (Exception e) {
          System.err.println("Error in swap");
          System.err.println("Exception: " + e);
      }
    }
;
```

```
// Expressions
//------------------------------------------------------------------------


expression returns [double r]
  {
                        double a,b;
                        int isArray=0;
                        double index=0;
                        r=0;
  }
  : #(PLUS a=expression b=expression)  { r=a+b; }
  | (#(MINUS . .)) => #(MINUS a=expression b=expression) { r=a-b; }
  | #(MINUS a=expression) { r=-a; }
  | #(MUL  a=expression b=expression)  { r=a*b; }
  | #(DIV  a=expression b=expression)  { r=a/b; }
  | int_lit:INT_LITERAL { r=(double)Integer.parseInt(int_lit.getText()); }
  | float_lit:FLOAT_LITERAL { r=Double.parseDouble(float_lit.getText()); }
  | #(AND a=expression b=expression) {if (a==1 && b==1) {r=1;} else {r=0;} }
  | #(OR a=expression b=expression) {if (a==1 || b==1) {r=1;} else {r=0;} }
  | #(EQUALS a=expression b=expression) {if (a == b) {r=1;} else {r=0;} }
  | #(NOT_EQUALS a=expression b=expression) {if (a != b) {r=1;} else {r=0;} }
  | #(LT a=expression b=expression) {if (a < b) {r=1;} else {r=0;} }
  | #(LTE a=expression b=expression) {if (a <= b) {r=1;} else {r=0;} }
  | #(GT a=expression b=expression) {if (a > b) {r=1;} else {r=0;} }
  | #(GTE a=expression b=expression) {if (a >= b) {r=1;} else {r=0;} }
  | #(var:IDENT (index=expression {isArray=1;})?)
                        {
                                try {
                                        if (isArray==1) {
                                                //array element
                                                r=st.get(var.getText(), (int)index);
                                        }
                                        else {
                                                // scalar variable
                                                r = st.get(var.getText());
                                        }
                                }
                                catch (Exception e) {
                                        System.err.println("Error: error getting variable " + var);
                                        System.err.println("Exception: " + e);
                                        System.exit(1);
                                }
                        }
  | ( #("max" IDENT . .) ) => #("max" var0:IDENT a=expression b=expression)
    {
      // max of a slice of an array
      try {
           if (!st.isArray(var0.getText())) {
                                System.out.println("Error in max: array expected");
                        }
                        else {
                                r = st.max(var0.getText(), (int)a, (int)b);
```

```
                }
            }
            catch (Exception e) {
                System.err.println("Error in max");
                System.err.println("Exception: " + e);
            }

  }
| ( #("max" . .) ) => #("max" a=expression b=expression)  {r=(a>b)?a:b;}
| #("max" var1:IDENT)
  {
    // max of the whole array
    try {
        if (!st.isArray(var1.getText())) {
                        System.out.println("Error in max: array expected");
                }
                else {
                        r = st.max(var1.getText());
                }
            }
            catch (Exception e) {
                System.err.println("Error in max");
                System.err.println("Exception: " + e);
            }
     }
| ( #("min" IDENT . .) ) => #("min" var4:IDENT a=expression b=expression)
  {
    // min of the slice of an array
    try {
        if (!st.isArray(var4.getText())) {
                        System.out.println("Error in min: array expected");
                }
                else {
                        r = st.min(var4.getText(), (int)a, (int)b);
                }
            }
            catch (Exception e) {
                System.err.println("Error in min");
                System.err.println("Exception: " + e);
            }

  }
| ( #("min" . .) ) => #("min" a=expression b=expression)  {r=(a<b)?a:b;}
| #("min" var2:IDENT)
  {
    // min of the whole array
    try {
        if (!st.isArray(var2.getText())) {
                        System.out.println("Error in min: array expected");
                }
                else {
                        r = st.min(var2.getText());
                }
            }
            catch (Exception e) {
                System.err.println("Error in min");
```

```
                System.err.println("Exception: " + e);
            }
          }
| #("length" var3:IDENT)
  {
    // length of an array
    try {
        if (!st.isArray(var3.getText())) {
                        System.out.println("Error in length: array expected");
                }
                else {
                        r = st.getLength(var3.getText());
                }
            }
            catch (Exception e) {
                System.err.println("Error in length");
                System.err.println("Exception: " + e);
            }
          }
| #(func_node:FNCALL func_name:IDENT (.)* )
  {
    // resolve functiona call
    try {
      // array to store actual arguments
        double actual[] = new double[func_node.getNumberOfChildren()-1];

        // actual arguments are siblings of func_name
        AST param;
        int i = 0;
        param = func_name.getNextSibling();
        while (param != null) {
                actual[i++] = expression(param);
                param = param.getNextSibling();
        }

        // if no functions are defined
        if (functions == null) {
                System.out.println("Error: function " + func_name.getText() + " is not defined");
                System.exit(1);
        }

        // find the needed function definition
        AST funcNode = functions.getFirstChild();
        while(funcNode != null && !funcNode.getText().equals(func_name.getText())) {
                funcNode = funcNode.getNextSibling();
        }

        // did not find definition for this function
        if (funcNode == null) {
                System.out.println("Error: function " + func_name.getText() + " is not defined");
                System.exit(1);
        }

        // save off the current symbol table and create a new one for this call
        SymbolTable global_st = st;
        st = new SymbolTable();
```

```
// find the formal parameters in the func definition
AST paramsNode = null;
if (funcNode.getFirstChild().getNextSibling().getText().equals("params")) {
	paramsNode = funcNode.getFirstChild().getNextSibling();
}

int formalNumber = 0;
if (paramsNode != null) {
	formalNumber = paramsNode.getNumberOfChildren();
}
int actualNumber = actual.length;
if (actualNumber < formalNumber) {
	System.out.println("Error: too few arguments in call to " + func_name.getText());
	System.exit(1);
}
else if(actualNumber > formalNumber) {
	System.out.println("Error: too many arguments in call to " + func_name.getText());
	System.exit(1);
}


// add formal parameters to the new symbol table
// and initialize them with the actual parameters' values
if (paramsNode != null) {
	param = paramsNode.getFirstChild();
				i = 0;
	while(param != null) {
		st.addVar(param.getText(), param.getFirstChild().getText(), actual[i++]);
		param  = param.getNextSibling();
	}
}

inFunction++;

			// find declarations node
AST declsNode = funcNode.getFirstChild();
while (declsNode != null && !declsNode.getText().equals("decls")) {
	declsNode = declsNode.getNextSibling();
}

// process declarations
if (declsNode != null) {
	declarations(declsNode);
}

// find statements node
AST stmtsNode = funcNode.getFirstChild();
while (stmtsNode != null && !stmtsNode.getText().equals("stmts")) {
	stmtsNode = stmtsNode.getNextSibling();
}

// process statements
if (stmtsNode != null) {
	statements(stmtsNode);
}
```

```
            inFunction--;
            returnMode = 0;


            // reactivate the previous symbol table
            st = global_st;
            // return value of the expression is the return value of the function
            r = returnValue;
        }
        catch (Exception e) {
            System.err.println("Error resolving function call to " + func_name.getText());
            System.err.println("Exception: " + e);
            System.exit(1);
        }
    }

;
```

```
// SymbolTable.java
// Symbol Table for ASML language

// W4115

// Olga Davidov
// olga.davidov@riag.com

import java.util.*;

public class SymbolTable
{
	HashMap vars;
	HashMap consts;
	HashMap arrays;

	public SymbolTable()
	{
		vars = new HashMap();
		consts = new HashMap();
		arrays = new HashMap();
	}

	// adds scalar variable to the symbol table
	public void addVar(String name, String type, double value) throws Exception
	{
		if (!exists(name)) {
			vars.put(name, new Double(value));
		}
		else {
			throw new Exception("Error: symbol " + name + " is already defined");
		}
	}

	// adds constant to the symbol table
	public void addConst(String name, String type, double value) throws Exception
	{
		if (!exists(name)) {
			consts.put(name, new Double(value));
		}
		else {
			throw new Exception("Error: symbol " + name + " is already defined");
		}
	}

	// adds array to the symbol table
	public void addArray(String name, String type, int size) throws Exception
	{
		if (!exists(name)) {
			Object value;
			value = new double[size];
			arrays.put(name, value);
		}
		else {
			throw new Exception("Error: symbol " + name + " is already defined");
		}
```

```
}

// sets a scalar to given value
public void set(String name, double value) throws Exception
{
	if (!exists(name)) {
		throw new Exception("Error: undefined symbol: " + name);
	}
	if (isConst(name)) {
		throw new Exception("Error: constant " + name + " cannot be modified");
	}
	if (isVariable(name)) {
		vars.put(name, new Double(value));
	}
	if (isArray(name)) {
		throw new Exception("Error: " + name + " is an array");
	}
}

// sets array element to given value
public void set(String name, int index, double value) throws Exception
{
	if (!exists(name)) {
		throw new Exception("Error: undefined symbol: " + name);
	}
	if (isConst(name)) {
		throw new Exception("Error: constant " + name + " cannot be modified");
	}
	if (isVariable(name)) {
		throw new Exception("Error: " + name + " is a scalar");
	}
	if (isArray(name)) {
		double[] array = (double[]) arrays.get(name);
		if (index < 0 || index >= array.length) {
			throw new Exception("Error: array index out of bounds");
		}
		else {
			array[index] = value;
			arrays.put(name, array);
		}
	}
}

// returns a value of an array element with given index
public double get(String name, int index) throws Exception
{
	if (!exists(name)) {
		throw new Exception("Error: undefined symbol: " + name);
	}
	if (isArray(name)) {
		double[] array = (double[]) arrays.get(name);
		if (index < 0 || index >= array.length) {
			throw new Exception("Error: array index out of bounds");
		}
		else {
			return array[index];
```

```
                }
        }
        else {
                throw new Exception("Error: symbol " + name + " is not an array");
        }
}

// returns value of a scalar
public double get(String name) throws Exception
{
        if (!exists(name)) {
                throw new Exception("Error: undefined symbol: " + name);
        }
        if (isConst(name)) {
                return ((Double)consts.get(name)).doubleValue();
        }
        if (isVariable(name)) {
                return ((Double)vars.get(name)).doubleValue();
        }
        else {
                throw new Exception("Error: symbol " + name + " is not scalar");
        }
}


// is identifier a variable
public boolean isVariable(String name) {
        return vars.containsKey(name);
}

// is identifier a constant
public boolean isConst(String name) {
        return consts.containsKey(name);
}

// is identifier an array
public boolean isArray(String name) {
        return arrays.containsKey(name);
}

// is identifier defined in the symbol table
public boolean exists(String name) {
        return vars.containsKey(name) || consts.containsKey(name) || arrays.containsKey(name);
}

// returns index of max element of an array
public int max(String name) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("max: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                int max = 0;
                for (int i = 0; i<array.length; i++) {
                        if (array[i] > array[max]) {
```

```
                                        max = i;
                                }
                        }
                        return max;
                }
}

// returns index of max element of an array slice
public int max(String name, int a, int b) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("max: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                int max = a;
                for (int i = a; i<b; i++) {
                        if (array[i] > array[max]) {
                                max = i;
                        }
                }
                return max;
        }
}

// returns index of min element of an array
public int min(String name) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("min: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                int min = 0;
                for (int i = 0; i<array.length; i++) {
                        if (array[i] < array[min]) {
                                min = i;
                        }
                }
                return min;
        }
}

// returns index of min element of an array slice
public int min(String name, int a, int b) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("min: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                int min = a;
                for (int i = a; i<b; i++) {
                        if (array[i] < array[min]) {
                                min = i;
                        }
```

```
                }
                return min;
        }
}

// exchanges elements with specified indexes
public void swap(String name, int i, int j) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("swap: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                double temp = array[i];
                array[i] = array[j];
                array[j] = temp;
                arrays.put(name, array);
        }
}

// prints an array
public void printArray(String name) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("printArray: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                for (int i = 0; i<array.length; i++) {
                        System.out.print(array[i] + " ");
                }
                System.out.println("");
        }
}

// returns length of an array
public double getLength(String name) throws Exception
{
        if (!isArray(name)) {
                throw new Exception("getLength: array expected: " + name);
        }
        else {
                double[] array = (double[]) arrays.get(name);
                return (double)array.length;
        }
}

// prints out the contents of the symbol table
public void showContents() {
        Iterator iter;
        System.out.println("----------------------");
        System.out.println("Vars:");
        iter = vars.keySet().iterator();
        while(iter.hasNext()) {
                System.out.println((String)iter.next());
        }
```

```
                System.out.println("Consts:");
                iter = consts.keySet().iterator();
                while(iter.hasNext()) {
                        System.out.println((String)iter.next());
                }
                System.out.println("Arrays:");
                iter = arrays.keySet().iterator();
                while(iter.hasNext()) {
                        System.out.println((String)iter.next());
                }
                System.out.println("----------------------");
        }



        static public void main(String argv[]) throws Exception
        {
                // used for unit testing
        }

}
```

```
// UserType.java
// user defined type for ASML language

// W4115

// Olga Davidov
// olga.davidov@riag.com

public class UserType
{
	String name;
	String baseType;
	int size;

	public UserType() {
		name = "";
		baseType = "";
		size = 0;
	}

	public UserType(String n, String b, int s) {
		name = n;
		baseType = b;
		size = s;
	}

	// returns name
	public String getName() {
		return name;
	}

	// returns array size
	public int getSize() {
		return size;
	}

	// returns array base type
	public String getBaseType() {
		return baseType;
	}
}
```

```
// UserTypeArray.java
// Collection of user defined types for ASML language

// W4115

// Olga Davidov
// olga.davidov@riag.com

import java.util.*;

public class UserTypeArray extends Vector
{
	// define a new type
	public void add(String inName, String inBaseType, int inSize) {
		add(new UserType(inName, inBaseType, inSize));
	}

	// get array size of a given type
	public int getSize(String inName) throws Exception
	{
		Iterator iter = iterator();
		while (iter.hasNext()) {
			UserType type = (UserType) iter.next();
			if (type.getName().equals(inName)) {
				return type.getSize();
			}
		}
		throw new Exception("Not found: type " + inName);
	}

	// get base type of a given user type
	public String getBaseType(String inName) throws Exception
	{
		Iterator iter = iterator();
		while (iter.hasNext()) {
			UserType type = (UserType) iter.next();
			if (type.getName().equals(inName)) {
				return type.getBaseType();
			}
		}
		throw new Exception("Not found: type " + inName);
	}

	static public void main(String argv[]) throws Exception
	{
		// used for unit testing
	}

}
```