Asl

A Simple Language

Jordi Cortadella

Dept. LSI, UPC

February 13, 2012

The Asl language

- Asl is a simple language with only scalar variables (*integer* and *Boolean*).
- Variables are not declared.
- Typeless language: the type of a variable is defined when it is assigned.
- The type of a variable can change dynamically.
- Type checking is performed at runtime.
- Functions are also typeless and they can return any type of value (or nothing) depending on their execution.

Asl example

```
func factorial(n)
  if n <= 1 then return 1 endif;
  return n*factorial(n-1)
endfunc

func main()
  write "Enter a number: "; read i;
  write "The factorial of "; write i;
  write " is: "; write factorial(i);
  write "%n"
endfunc
```

Another example

```
func main()
  write "Enter a number: "; read x;
  d = 1;
  p = is_prime(x,d);
  if p then write "It is prime.%n"
  else write "It is not prime.%n" endif;
  if not p then
    write d; write " is a divisor of ";
    write x; write ".%n"
  endif
endfunc

func is_prime(n, &div)
  if n = 1 then return false endif;
  div = 2;
  while div*div <= n do
    if n%div = 0 then return false endif;
    div = div + 1;
  endwhile;
  return true
endfunc
```

Operator precedence in expressions

logical negation and unary sign	`not, +, -`
multiplicative arithmetic operations	`*, /, %`
additive arithmetic operations	`+, -`
relational operators	`=, !=, <, <=, >, >=`
logical and	`and`
logical or	`or`

All binary operators are left-associative

Executing a program

Interpreter options

```
$ Asl -help
usage: Asl [options] file
  -ast <file>      write the AST
  -dot             dump the AST in dot format
  -help            print this message
  -noexec          do not execute the program
  -trace <file>    write a trace of function calls during
                   the execution of the program
```

Writing the AST (text)

```
$ Asl -ast file.ast -noexec file.asl
```

Visualizing the AST (pdf)

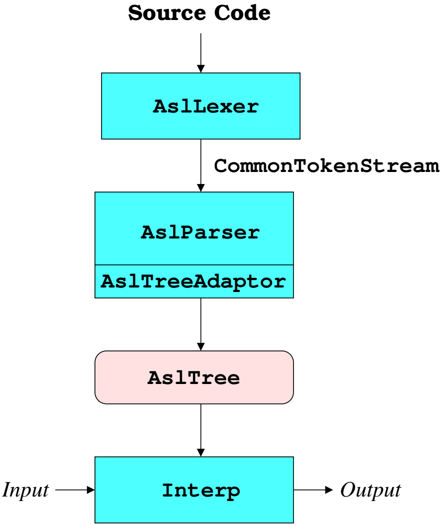
```
$ Asl -dot -ast file.dot -noexec file.asl
$ dot -Tpdf file.dot -o file.pdf
```

Executing a program

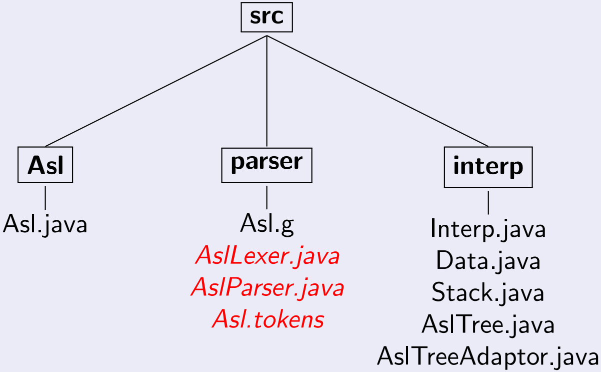
Writing an execution trace

```
$ Asl -trace fibonacci.trace fibonacci.asl
Enter ther order of the Fibonacci number: 4
Fibonacci(4)=5
$ cat fibonacci.trace
main() <entry point>
|   fib_rec(n=4) <line 8>
|   |   fib_rec(n=3) <line 28>
|   |   |   fib_rec(n=2) <line 28>
|   |   |   |   fib_rec(n=1) <line 28>
|   |   |   |   return 1 <line 27>
|   |   |   |   fib_rec(n=0) <line 28>
|   |   |   |   return 1 <line 27>
|   |   |   return 2 <line 28>
|   |   |   fib_rec(n=1) <line 28>
|   |   |   return 1 <line 27>
|   |   return 3 <line 28>
|   |   fib_rec(n=2) <line 28>
|   |   |   fib_rec(n=1) <line 28>
|   |   |   return 1 <line 27>
|   |   |   fib_rec(n=0) <line 28>
|   |   |   return 1 <line 27>
|   |   return 2 <line 28>
|   return 5 <line 28>
return <line 11>
```

Interpretation flow



Files of the interpreter



src

Asl

Asl.java

parser

Asl.g

AslLexer.java

AslParser.java

Asl.tokens

interp

Interp.java

Data.java

Stack.java

AslTree.java

AslTreeAdaptor.java

Files in *red* are automatically generated by ANTLR.

Main program (Asl.java)

```java
AslLexer lex = new AslLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lex);

AslParser parser = new AslParser(tokens);
AslTreeAdaptor adaptor = new AslTreeAdaptor();
parser.setTreeAdaptor(adaptor);
AslParser.prog_return result = null;

try {
    result = parser.prog();
} catch (Exception e) {}
int nerrors = parser.getNumberOfSyntaxErrors();
if (nerrors > 0) { ... }

AslTree t = (AslTree)result.getTree();
Interp I = null;
int linenumber = -1;
try {
    I = new Interp(t, tracefile);
    I.Run();
} catch (RuntimeException e) { ...
} catch (StackOverflowError e) { ...
}
```

Lexer (Asl.g)

```
EQUAL       : ’=’ ;
NOT_EQUAL   : ’!=’ ;
LE          : ’<=’;
...
PLUS        : ’+’ ;
MINUS       : ’-’ ;
MUL         : ’*’;
...
NOT         : ’not’;
AND         : ’and’ ;
...
WHILE       : ’while’ ;
DO          : ’do’ ;
...
ID          : (’a’..’z’|’A’..’Z’|’_’) (’a’..’z’|’A’..’Z’|’0’..’9’|’_’)* ;
INT         : ’0’..’9’+ ;

COMMENT     : ’//’ ~(’\n’|’\r’)* ’\r’? ’\n’ {$channel=HIDDEN;}
            | ’/*’ ( options {greedy=false;} : . )* ’*/’ {$channel=HIDDEN;}
            ;

STRING      :  ’"’ ( ESC_SEQ | ~(’\\’|’"’) )* ’"’ ;

fragment ESC_SEQ:   ’\\’ (’b’|’t’|’n’|’f’|’r’|’\"’|’\’’|’\\’) ;

WS          : ( ’ ’ | ’\t’ | ’\r’ | ’\n’) {$channel=HIDDEN;};
```

The grammar (Asl.g)

```
// A program is a list of functions
prog    : func+ EOF ;

// A function has a name, a list of parameters and a block of instructions
func    : FUNC ID params block_instructions ENDFUNC ;

// The list of parameters (it can be empty)
params  : '(' paramlist? ')' ;
paramlist: param (',' param)* ;

// Parameters with & as prefix are passed by reference
param   :   '&' ID | ID ;

// A list of instructions
block_instructions:        instruction (';' instruction)* ;

// Different types of instructions
instruction
        :       assign          // Assignment
        |       ite_stmt        // if-then-else
        |       while_stmt      // while statement
        |       funcall         // Call to a procedure (no result produced)
        |       return_stmt     // Return statement
        |       read            // Read a variable
        |       write           // Write a string or an expression
        |                       // Nothing
        ;
```

The grammar (Asl.g)

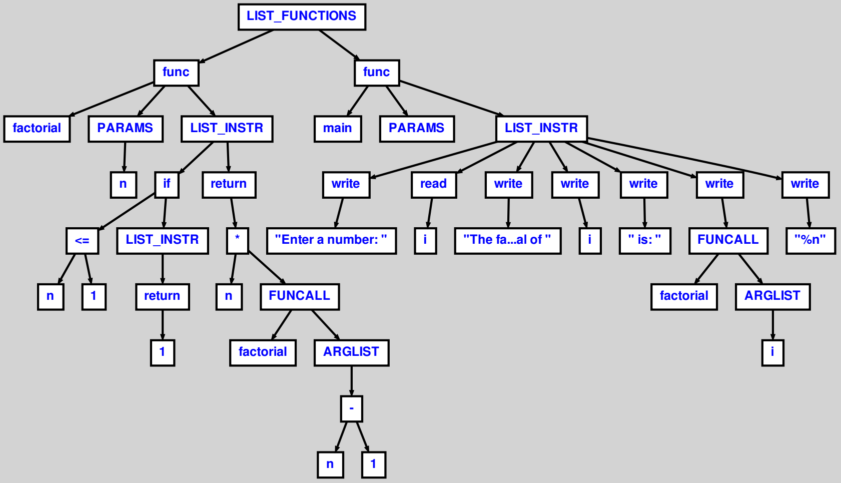
```
assign      : ID EQUAL expr ;

ite_stmt    : IF expr THEN block_instructions (ELSE block_instructions)? ENDIF ;

while_stmt  : WHILE expr DO block_instructions ENDWHILE ;

return_stmt : RETURN expr? ;

read        : READ ID ;

write       : WRITE (expr | STRING ) ;

// Grammar for expressions with boolean, relational and aritmetic operators
expr        : boolterm (OR boolterm)* ;
boolterm    : boolfact (AND boolfact)* ;
boolfact    : num_expr ((EQUAL | NOT_EQUAL | LT | LE | GT | GE) num_expr)? ;
num_expr    : term ( (PLUS | MINUS) term)* ;
term        : factor ( (MUL | DIV | MOD) factor)* ;
factor      : (NOT | PLUS | MINUS)? atom ;
atom        : ID | INT |   (TRUE | FALSE) | funcall | ’(’ expr ’)’ ;

funcall     : ID ’(’ expr_list? ’)’ ;
expr_list   : expr (’,’ expr)* ;
```

Abstract Syntax Tree

```
func factorial(n)
  if n <= 1 then
    return 1
  endif;
  return n*factorial(n-1)
endfunc

func main()
  write "Enter a number: ";
  read i;
  write "The factorial of ";
  write i;
  write " is: ";
  write factorial(i);
  write "%n"
endfunc
```

```
(LIST_FUNCTIONS
    (func factorial
        (PARAMS n)
        (LIST_INSTR
            (if
                (<= n 1)
                (LIST_INSTR
                    (return 1)
        )   )   )
        (return
            (* n
                (FUNCALL factorial
                    (ARGLIST
                        (- n 1)
    )   )   )   )   )
    (func main PARAMS
        (LIST_INSTR
            (write "Enter a number: ")
            (read i)
            (write "The factorial of ")
            (write i)
            (write " is: ")
            (write
                (FUNCALL factorial
                    (ARGLIST i)
            )   )
            (write "%n")
)   )   )
```

ANTLR Abstract Syntax Tree



ANTLR Abstract Syntax Tree

- ANTLR-generated parsers can build trees.
- ANTLR provides a default tree structure called `CommonTree`.
- Every node contains:
  - A list of children
  - A payload (the token from which the node was created)
- Imaginary tokens can be defined to create tokens that do not correspond to any input lexema.

AST operations

```
Tree getChild(int i);      // Gets the i-th child of the node
int getChildCount();       // Returns the number of children of the node
void addChild(Tree t);     // Add t as a child of the node
boolean isNil();           /* Indicates the node is a nil node but may
                           still have children, meaning that the tree is a
                           flat list */
```

ANTLR AST: Token attributes

Attr	Type	Description
text	**String**	The text matched for the token; translates to a call to `getText()`.
type	**int**	The token type (nonzero positive integer) of the token such as INT; translates to a call to `getType()`.
line	**int**	The line number on which the token occurs, counting from 1; translates to a call to `getLine()`.
pos	**int**	The character position within the line at which the token's first character occurs counting from zero; translates to a call to `getCharPositionLine()`.
index	**int**	The overall index of this token in the token stream, counting from zero; translates to a call to `getTokenIndex()`.
channel	**int**	The token's channel number. The parser tunes the only one channel, effectively ignoring off-channel tokens. The default channel is 0 (`Token.DEFAULT_CHANNEL`), and the default hidden channel is `Token.HIDDEN_CHANNEL`.
tree	**Object**	When building trees, this attribute points to the tree node created for the token; translates to a local variable reference that points to the node, and therefore, this attribute does not live inside the `Token` object itself.

Extending the AST (AslTree.java)

Class AslTree

```
public class AslTree extends CommonTree {

    private int intValue; /* Field to store integer literals */

    // Constructor of the class
    public AslTree(Token t) { super(t); }

    // Function to get the child of the node
    public AslTree getChild(int i) {
        return (AslTree) super.getChild(i);
    }

    // Get the integer value of the node
    public int getIntValue() { return intValue;}

    // Define the integer value of the node
    public void setIntValue() {
        intValue = Integer.parseInt(getText());
    }
    ...
}
```

Extending the AST (AslTreeAdaptor.java)

Class AslTreeAdaptor

```java
/**
 * This is the tree adaptor for the extended class of AST nodes.
 * It re-defines some required methods to cast the AST tree nodes
 * to the new AST nodes.
 */

public class AslTreeAdaptor extends CommonTreeAdaptor {
    public Object create(Token t) {
        return new AslTree(t);
    }

    public Object dupNode(Object t) {
        if ( t==null ) return null;
        return create(((AslTree)t).token);
    }

    public Object errorNode(TokenStream input, Token start,
                            Token stop, RecognitionException e) {
        return null;
    }
}
```

Building the AST: defining imaginary tokens (Asl.g)

```
options {
    output = AST;
    ASTLabelType = AslTree;
}

// Imaginary tokens to create some AST nodes

tokens {
    LIST_FUNCTIONS; // List of functions (root of the tree)
    ASSIGN;       // Assignment instruction
    PARAMS;       // List of parameters in the declaration of a function
    FUNCALL;      // Function call
    ARGLIST;      // List of arguments passed in a function call
    LIST_INSTR;   // Block of instructions
    BOOLEAN;      // Boolean atom (for Boolean constants "true" or "false")
    PVALUE;       // Parameter by value in the list of parameters
    PREF;         // Parameter by reference in the list of parameters
}
```

Building the AST using operators (Asl.g)

The operator ^ makes the node root of the subtree generated by the rule.
The operator ! removes the node from the subtree.

Grammar for expressions

```
expr     :   boolterm (OR^ boolterm)* ;

boolterm:   boolfact (AND^ boolfact)* ;

boolfact:   num_expr ((EQUAL^ | NOT_EQUAL^ | LT^ | LE^ | GT^ | GE^) num_expr)? ;

num_expr:   term ( (PLUS^ | MINUS^) term)* ;

term     :   factor ( (MUL^ | DIV^ | MOD^) factor)* ;

factor   :   (NOT^ | PLUS^ | MINUS^)? atom ;

atom     :   ID
         |   INT
         |   (b=TRUE | b=FALSE)  -> ^(BOOLEAN[$b,$b.text])
         |   funcall
         |   ’(’! expr ’)’!
         ;
```

Building the AST with rewrite rules (Asl.g)

A fragment of grammar with rewrite rules

```
prog    : func+ EOF -> ^(LIST_FUNCTIONS func+) ;

func    : FUNC^ ID params block_instructions ENDFUNC! ;

params  : ’(’ paramlist? ’)’ -> ^(PARAMS paramlist?) ;

paramlist: param (’,’! param)* ;

param   :   ’&’ id=ID -> ^(PREF[$id,$id.text])
        |   id=ID     -> ^(PVALUE[$id,$id.text])
        ;
block_instructions
        :   instruction (’;’ instruction)*
            -> ^(LIST_INSTR instruction+)
        ;

assign  :   ID eq=EQUAL expr -> ^(ASSIGN[$eq,":="] ID expr) ;

funcall :   ID ’(’ expr_list? ’)’ -> ^(FUNCALL ID ^(ARGLIST expr_list?)) ;
```

Interpreter files

- Interp.java contains the core of the interpreter, traversing the AST and executing the instructions.
- Data.java contains the class to represent data values (integer and Boolean) and execute operations on them.
- Stack.java implements the memory of the interpreter with a stack of activation records that contain pairs of strings (variable names) and data (values).
- AslTree.java contains a subclass of the AST that extends the information included in every AST node.
- AslTreeAdaptor.java contains a subclass required by ANTLR to have access to the extended AST.

Interpreter class (Interp.java)

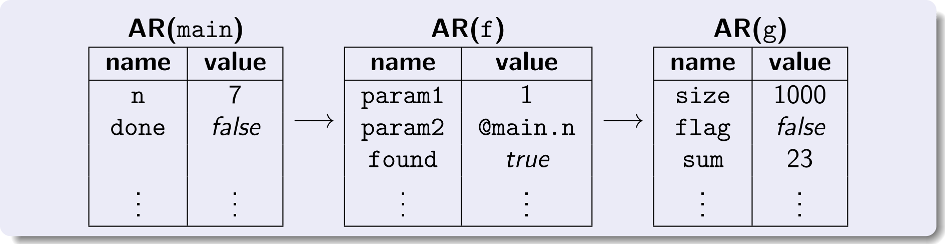
```
public class Interp {

    private Stack Stack;  // Stack of the Virtual Machine

    /* Map between function names (keys) and ASTs (values).
     * Each entry of the map stores the root of the AST
     * correponding to the function.*/
    private HashMap<String,AslTree> FuncName2Tree;

    ...

    /* Constructor of the interpreter. It prepares the main
     * data structures for the execution of the main program. */
    public Interp(AslTree T, String tracefile) { ... }

    /* Runs the program by calling the main function without par
    public void Run() { executeFunction ("main", null); }
    ...
}
```

Interpreter: data (Data.java)

```java
public class Data {
    public enum Type {VOID, BOOLEAN, INTEGER;} // Types of data
    private Type type; // Type of data
    private int value; // Value of the data

    /* Constructor for integers */
    Data(int v) { type = Type.INTEGER; value = v; }

    /* Constructor for Booleans */
    Data(boolean b) { type = Type.BOOLEAN; value = b ? 1 : 0; }

    /* Indicates whether the data is integer */
    public boolean isInteger() { return type == Type.INTEGER; }

    /* Gets the value of an integer data. */
    public int getIntegerValue() {
        assert type == Type.INTEGER; return value;
    }

    /* Defines an integer value for the data */
    public void setValue(int v) { type = Type.INTEGER; value = v; }
    ...
}
```

Stack

- No global variables.
- Every activation record (AR) contains the variables in the scope of a function.
- Parameters can be passed by reference.



AR(`main`)

name	**value**
`n`	7
`done`	*false*
⋮	⋮

→

AR(`f`)

name	**value**
`param1`	1
`param2`	`@main.n`
`found`	*true*
⋮	⋮

→

AR(`g`)

name	**value**
`size`	1000
`flag`	*false*
`sum`	23
⋮	⋮

Stack class (Stack.java)

```java
public class Stack {

    /* Stack of activation records */
    private LinkedList<HashMap<String,Data>> Stack;

    /* Reference to the current activation record */
    private HashMap<String,Data> CurrentAR = null;

    /* Constructor of the stack */
    public Stack() { ... }

    /* Creates a new activation record on the top of the stack */
    public void pushActivationRecord(String name, int line) { ... }

    /* Destroys the current activation record */
    public void popActivationRecord() { ... }

    /* Defines the value of a variable. */
    public void defineVariable(String name, Data value) { ... }

    /* Gets the value of the variable. */
    public Data getVariable(String name) { ... }
    ...
}
```

Interp class: important functions (Interp.java)

```
public class Interp {
    ....
    /* Executes a function. */
    private Data executeFunction (String funcname, AslTree args) { ... }

    /* Executes a block of instructions. */
    private Data executeListInstructions (AslTree t) { ... }

    /* Executes an instruction. */
    private Data executeInstruction (AslTree t) { ... }

    /* Evaluates the expression represented in the AST t. */
    private Data evaluateExpression(AslTree t) { ... }

    /* Gathers the list of arguments of a function call. It also checks
     * that the arguments are compatible with the parameters.
     * Returns the list of evaluated arguments. */
    private ArrayList<Data> listArguments (AslTree AstF, AslTree args) { ... }

    ...
}
```

Executing a function (Interp.java)

```
private Data executeFunction (String funcname, AslTree args) {

    AslTree f = FuncName2Tree.get(funcname);
    if (f == null) throw
       new RuntimeException("␣function␣" + funcname + "␣not␣declared");

    ArrayList<Data> Arg_values = listArguments(f, args);

    // List of parameters of the callee
    AslTree p = f.getChild(1);
    int nparam = p.getChildCount(); // Number of parameters

    Stack.pushActivationRecord(funcname, lineNumber());

    // Copy the parameters to the current activation record
    for (int i = 0; i < nparam; ++i) {
        String param_name = p.getChild(i).getText();
        Stack.defineVariable(param_name, Arg_values.get(i));
    }

    Data result = executeListInstructions (f.getChild(2));

    // If the result is null, then the function returns void
    if (result == null) result = new Data();

    Stack.popActivationRecord();
    return result;
}
```

Executing a block of instructions (Interp.java)

```java
/**
 * Executes a block of instructions. The block is terminated
 * as soon as an instruction returns a non-null result.
 * Non-null results are only returned by "return" statements.
 * * t is the AST of the block of instructions.
 * * returns the data returned by the instructions
 *   (null if no return statement has been executed).
 */

 private Data executeListInstructions (AslTree t) {
     assert t != null;
     Data result = null;
     int ninstr = t.getChildCount();
     for (int i = 0; i < ninstr; ++i) {
         result = executeInstruction (t.getChild(i));
         if (result != null) return result;
     }
     return null;
 }
```

Executing one instruction (Interp.java)

```
private Data executeInstruction (AslTree t) {

    Data value; // The returned value

    // A big switch for all type of instructions
    switch (t.getType()) {

        case AslLexer.ASSIGN: // Assignment
            value = evaluateExpression(t.getChild(1));
            Stack.defineVariable (t.getChild(0).getText(), value);
            return null;

        case AslLexer.IF: // If—then—else
            value = evaluateExpression(t.getChild(0));
            checkBoolean(value);
            if (value.getBooleanValue()) return executeListInstructions(t.getChild(1));
            // Is there else statement ?
            if (t.getChildCount() == 3) return executeListInstructions(t.getChild(2));
            return null;

        case AslLexer.WHILE: // While
            while (true) {
                value = evaluateExpression(t.getChild(0));
                checkBoolean(value);
                if (!value.getBooleanValue()) return null;
                Data r = executeListInstructions(t.getChild(1));
                if (r != null) return r;
            }

        case AslLexer.RETURN:  // Return
            if (t.getChildCount() != 0) {
                return evaluateExpression(t.getChild(0));
            }
            return new Data(); // No expression: returns void data
        ...
    }
}
```

Executing one instruction (Interp.java)

```java
private Data executeInstruction (AslTree t) {
    ...

    switch (t.getType()) {
        ...
        case AslLexer.READ: // Read statement
            String token = null;
            Data val = new Data(0);;
            try {
                token = stdin.next();
                val.setValue(Integer.parseInt(token));
            } catch (NumberFormatException ex) {
                throw new RuntimeException ("Format␣error␣when␣reading␣a␣number:␣" + token);
            }
            Stack.defineVariable (t.getChild(0).getText(), val);
            return null;

        case AslLexer.WRITE: // Write statement
            AslTree v = t.getChild(0);
            // Special case for strings
            if (v.getType() == AslLexer.STRING) {
                System.out.format(v.getStringValue());
                return null;
            }

            // Write an expression
            System.out.print(evaluateExpression(v).toString());
            return null;

            // Function call
            case AslLexer.FUNCALL:
                executeFunction(t.getChild(0).getText(), t.getChild(1));
                return null;

            default: assert false; // Should never happen
        }
    }
}
```

Evaluating an expression: atoms (Interp.java)

```
private Data evaluateExpression(AslTree t) {

    Data value = null;
    int type = t.getType();
    // Atoms
    switch (type) {
        case AslLexer.ID: // A variable
            value = new Data(Stack.getVariable(t.getText())); break;

        case AslLexer.INT: // An integer literal
            value = new Data(t.getIntValue()); break;

        case AslLexer.BOOLEAN: // A Boolean literal
            value = new Data(t.getBooleanValue()); break;

        // A function call. Checks that the function returns a result.
        case AslLexer.FUNCALL:
            value = executeFunction(t.getChild(0).getText(), t.getChild(1));
            assert value != null;
            if (value.isVoid()) {
                throw new RuntimeException ("function␣expected␣to␣return␣a␣value
            }
            break;
        default: break;
    }

    ...
```

Evaluating an expression: one operand (Interp.java)

```java
private Data evaluateExpression(AslTree t) {
    ...

    // Unary operators
    value = evaluateExpression(t.getChild(0));
    if (t.getChildCount() == 1) {
        switch (type) {
            case AslLexer.PLUS:
                checkInteger(value); break;

            case AslLexer.MINUS:
                checkInteger(value);
                value.setValue(-value.getIntegerValue());
                break;

            case AslLexer.NOT:
                checkBoolean(value);
                value.setValue(!value.getBooleanValue());
                break;
            default: assert false; // Should never happen
        }
        return value;
    }

    ...
```

Evaluating an expression: two operands (Interp.java)

```java
private Data evaluateExpression(AslTree t) {
    ...
    Data value2;
    switch (type) {
        // Relational operators
        case AslLexer.EQUAL:
        ... // the other relational operators
        case AslLexer.GE:
            value2 = evaluateExpression(t.getChild(1));
            if (value.getType() != value2.getType()) {
              throw new RuntimeException ("Incompatible␣types␣in␣relational␣expression");
            }
            value = value.evaluateRelational(type, value2);
            break;

        // Arithmetic operators
        case AslLexer.PLUS:
        ... // the other arithmetic operators
        case AslLexer.MOD:
            value2 = evaluateExpression(t.getChild(1));
            checkInteger(value); checkInteger(value2);
            value.evaluateArithmetic(type, value2);
            break;

        // Boolean operators
        case AslLexer.AND:
        case AslLexer.OR:
            // The first operand is evaluated, but the second
            // is deferred (lazy, short-circuit evaluation).
            checkBoolean(value);
            value = evaluateBoolean(type, value, t.getChild(1));
            break;

        default: assert false; // Should never happen
    }
    return value;
}
```

Boolean expressions: lazy evaluation (Interp.java)

```java
private Data evaluateBoolean (int type, Data v, AslTree t) {

    switch (type) {
        case AslLexer.AND:
            // Short circuit if v is false
            if (!v.getBooleanValue()) return v;
            break;

        case AslLexer.OR:
            // Short circuit if v is true
            if (v.getBooleanValue()) return v;
            break;

        default: assert false;
    }

    // Return the value of the second expression
    v = evaluateExpression(t);
    checkBoolean(v);
    return v;
}
```

Arithmetic and relational expressions: (Data.java)

```
public void evaluateArithmetic (int op, Data d) {
    assert type == Type.INTEGER && d.type == Type.INTEGER;
    switch (op) {
        case AslLexer.PLUS: value += d.value; break;
        case AslLexer.MINUS: value -= d.value; break;
        case AslLexer.MUL: value *= d.value; break;
        case AslLexer.DIV: checkDivZero(d); value /= d.value; break;
        case AslLexer.MOD: checkDivZero(d); value %= d.value; break;
        default: assert false;
    }
}

public Data evaluateRelational (int op, Data d) {
    assert type != Type.VOID && type == d.type;
    switch (op) {
        case AslLexer.EQUAL: return new Data(value == d.value);
        case AslLexer.NOT_EQUAL: return new Data(value != d.value);
        case AslLexer.LT: return new Data(value < d.value);
        case AslLexer.LE: return new Data(value <= d.value);
        case AslLexer.GT: return new Data(value > d.value);
        case AslLexer.GE: return new Data(value >= d.value);
        default: assert false;
    }
    return null;
}
```

Extension: arrays

Extend the Asl language with arrays.

- All the elements of the array have the same type (integer or Boolean).
- Indices of array `A` range from 0 to `A.size-1`.
- Arrays can only be passed as arguments by reference.
- The assignment to a non-existing location automatically extends the size of the array up to that location, filling up the non-assigned slots with *zero* or *false* depending on whether the array has integers or Booleans, respectively.
- Any assignment with a type different from the type of the array implies the allocation of a new array.

Extension: arrays

Examples

```
A[2] = 8;
// Generates an array with contents [0,0,8] (A.size=3)
A[5] = 1;
// Resizes the array with contents [0,0,8,0,0,1] (A.size=6)
A[3] = true;
// New array with Booleans [false,false,false,true] (A.size=4)
A = 13;
// The previous array is destroyed. A is now a scalar variable
A[4] = 6;
// A is again an array
z = A[8];
// Execution error: index out-of-bounds
```

Extension: arrays and function calls

Arrays are implicitly passed by reference. There is no semantic difference between `f(A)` and `f(&A)` when `A` is an array.

Examples

```
// A is an array of integers
...
m = min_vector(A);  // A is implicitly passed by reference
...
x = average(&A);    // A is explicitly passed by reference
...
return A;           // An array can be returned as result
...
```

The interpreter must handle the memory allocation/de-allocation for arrays. Beware of the returns from functions.