Asl: A simple language

Jordi Cortadella

April 30, 2011

1 Asl

`Asl` is a very simple imperative programming language created to illustrate how an interpreter works. The `Asl` interpreter has been designed using ANTLR v3 and Java. Let us start with a simple example written in `Asl`.

```
/**
 * A function that computes the factorial of a number
 * using a recursive definition of factorial.
 */
func factorial(n)
  if n <= 1 then return 1 endif;
  return n*factorial(n-1)
endfunc

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

The program has two functions: `factorial` and `main`. Every program must have a `main` function without parameters that is identified as the primary function where the program starts executing.

1.1 Data types and variables

`Asl` only has two data types: *integer* and *Boolean*. It does not have any complex data structures such as arrays or structs (that is why the language is so simple).

`Asl` is a typeless programming language, i.e., any variable can have values of any type and the type can change dynamically. As in many interpreted languages, variables are not declared. They are created when a value is assigned to them. Type checking is performed dynamically at runtime.

`Asl` does not have global variables. All the variables are local within the function in which they are created. The only communication mechanism between functions is parameter passing.

Functions are also typeless and they can return any type of value (or nothing) depending on their execution.

1.2 Assignment

It is the basic statement of an imperative language. The syntax is the following:

`variable =` *expression*

If the variable does not exist, it is created by the assignment. No type checking is performed with regard to the previous value of the variable since variables can change their type dynamically.

1.3 Input and output

`Asl` can only read from the standard input and write to the standard output. The `read` statement can only read integer values into a variable, whereas the `write` statement can either write a string or the value of an expression. The strings are interpreted in the same way as in Java using the `%` conversion. For example, `%n` is converted into a newline.

1.4 Control flow statements

The following program covers all the existing statements in the language. The function `is_prime` returns a Boolean result indicating whether `n` is prime or not. In case it is not prime, it also writes the smallest prime divisor of the number.

```
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
```

There are three statements for control flow:

`if` *expression* `then` *list_instructions* `else` *list_instructions* `endif`

`while` *expression* `do` *list_instructions* `endwhile`

`return` *expression*

The `else` clause of the `if` statement is optional. The semantics of these statements is the same as the one of similar instructions in other programming languages.

The semicolon `';'` is used to separate individual statements. Note that the semicolon is used to separate statements and not to terminate them. However, a semicolon after a final statement is also accepted and interpreted as a separator from an empty instruction.

1.5 Parameters

Parameters are also typeless. There are two mechanisms to pass parameters: by *value* and by *reference*. The mechanism is distinguished by adding the prefix `&` in the declaration of reference parameters.

In the previous example, the funcion `is_prime` has one parameter passed by reference (`&div`). The caller does not have to specify the parameter passing is by value or reference. The mechanism is implicitly inferred from the declaration of parameters in the callee.

Expressions can be passed by value, whereas only variables can be passed by reference.

1.6 Expressions

Expressions are written as sequences of operands and operators. `Asl` has a rich set of logical, relational and arithmetic operators. The following table contains all the operators in precedence order. The topmost operators are the ones with highest priority.

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

Type checking is done when evaluating expressions. The arithmetic operators require the operands to be integer, whereas the logical operators require the operands to be Boolean. Relational operators require both operands to have the same type. When comparing Boolean values, *true* is assumed to be greater than *false*.

2 The interpreter

The interpreter of `Asl` has been designed using ANTLRv3 and Java. The interpreter is installed by executing a ``make'' command in the root directory of the `Asl` distribution. The user might need to redefine the variables `ANTLR_ROOT` and `JAVA_LIBS` to specify the location of ANTLR and Java libraries.

The interpreter is invoked by executing the command `Asl <file>`. The interpreter can also be invoked with some options that can be listed by running `Asl -help`:

```
$ 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
```

The option `-ast` generates a textual representation of the Abstract Syntax Tree (AST) in dot format. If the `-dot` option is used, the representation is in dot format, that can be later transformed into some graphical format using `dot`, e.g.:

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

The option `-noexec` must be used when the user does not want to execute the program. It is useful when only a visualization of the AST is required.

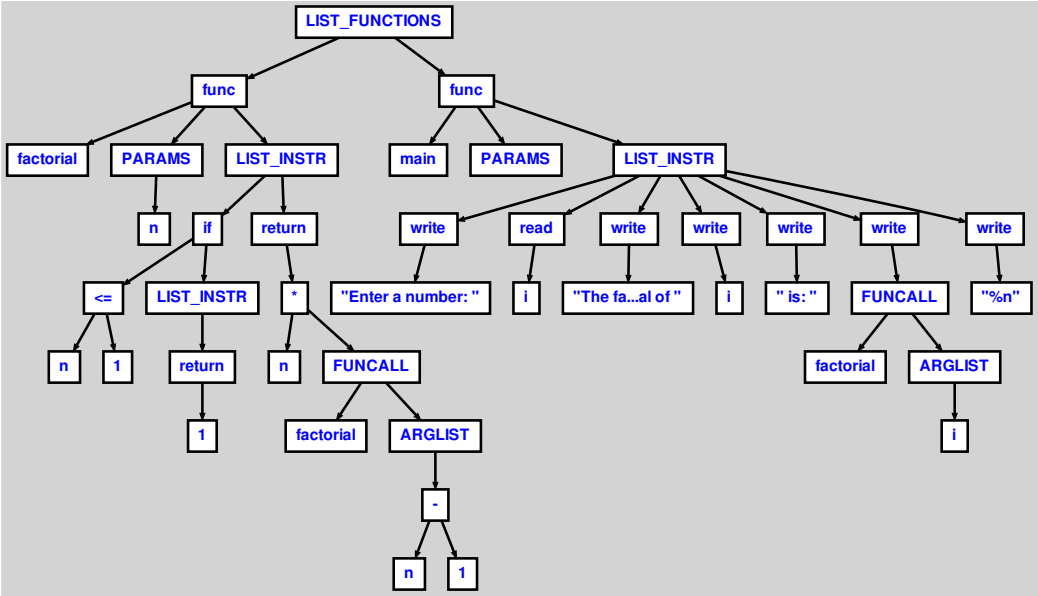

Figure 1: AST of the factorial example.

Figure 1 depicts the AST of the first example used in this document.

The option `-noexec` must be used when the user does not want to execute the program. It is useful when only a visualization of the AST is required.

The option `-trace` can be used for debugging. For example, let us consider the following program that generates the moves for the problem of Hanoi towers.

```
 1:    func main()
 2:      write "Number of disks: "; read n;
 3:      nmoves = 0;
 4:      hanoi(n, 1, 2, 3, nmoves);
 5:      write "Total number of moves: "; write nmoves; write "%n";
 6:    endfunc
 7:
 8:    func hanoi(n, from, aux, to, &mov)
 9:      if n > 0 then
10:        hanoi(n-1, from, aux, to, mov);
11:        write "Move from "; write from;
12:        write " to "; write to; write "%n";
13:        mov = mov + 1;
14:        hanoi(n-1, aux, to, from, mov);
15:      endif
16:    endfunc
```

In this example, the parameter `mov` is passed by reference and accumulates the number of moves. The rods are represented by the numbers 1, 2 and 3.

The execution the program with 3 disks would look as follows:

```
$ Asl hanoi.asl -trace hanoi.trace
Number of disks: 3
```

```
Move from 1 to 3
Move from 1 to 3
Move from 2 to 1
Move from 1 to 3
Move from 2 to 1
Move from 2 to 1
Move from 3 to 2
Total number of moves: 7
```

The file `hanoi.trace` reports the trace of function calls and returns with the values of the parameters and returned data. It also reports the value of the variables passed by reference at the return of each function. The contents of the file is the following:

```
main() <entry point>
|   hanoi(n=3, from=1, aux=2, to=3, &mov=0) <line 4>
|   |   hanoi(n=2, from=1, aux=2, to=3, &mov=0) <line 10>
|   |   |   hanoi(n=1, from=1, aux=2, to=3, &mov=0) <line 10>
|   |   |   |   hanoi(n=0, from=1, aux=2, to=3, &mov=0) <line 10>
|   |   |   |   return, &mov=0 <line 9>
|   |   |   |   hanoi(n=0, from=2, aux=3, to=1, &mov=1) <line 14>
|   |   |   |   return, &mov=1 <line 9>
|   |   |   return, &mov=1 <line 9>
|   |   |   hanoi(n=1, from=2, aux=3, to=1, &mov=2) <line 14>
|   |   |   |   hanoi(n=0, from=2, aux=3, to=1, &mov=2) <line 10>
|   |   |   |   return, &mov=2 <line 9>
|   |   |   |   hanoi(n=0, from=3, aux=1, to=2, &mov=3) <line 14>
|   |   |   |   return, &mov=3 <line 9>
|   |   |   return, &mov=3 <line 9>
|   |   return, &mov=3 <line 9>
|   |   hanoi(n=2, from=2, aux=3, to=1, &mov=4) <line 14>
|   |   |   hanoi(n=1, from=2, aux=3, to=1, &mov=4) <line 10>
|   |   |   |   hanoi(n=0, from=2, aux=3, to=1, &mov=4) <line 10>
|   |   |   |   return, &mov=4 <line 9>
|   |   |   |   hanoi(n=0, from=3, aux=1, to=2, &mov=5) <line 14>
|   |   |   |   return, &mov=5 <line 9>
|   |   |   return, &mov=5 <line 9>
|   |   |   hanoi(n=1, from=3, aux=1, to=2, &mov=6) <line 14>
|   |   |   |   hanoi(n=0, from=3, aux=1, to=2, &mov=6) <line 10>
|   |   |   |   return, &mov=6 <line 9>
|   |   |   |   hanoi(n=0, from=1, aux=2, to=3, &mov=7) <line 14>
|   |   |   |   return, &mov=7 <line 9>
|   |   |   return, &mov=7 <line 9>
|   |   return, &mov=7 <line 9>
|   return, &mov=7 <line 9>
return <line 5>
```

2.1 Organization of the interpreter

The source files of the interpreter are organized in three packages (see Fig. 2):

Asl: contains the class `Asl` with the entry point (`main`) of the program. It calls the *parser* and *interpreter*.

parser: contains the grammar and creator of the AST. In the original distribution, this directory only contains the file `Asl.g`. The other files (italics in Fig. 2) are created by ANTLR.

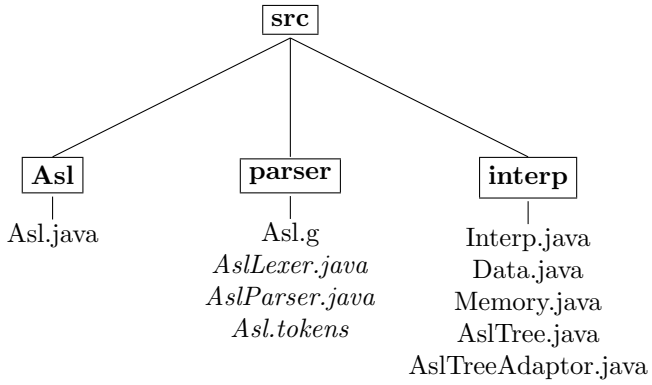

Figure 2: Structure of the `src` directory containing the source files of the interpreter.

interp: contains the interpreter of the AST organized in different files:

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

2.1.1 Interp.java

It has two public functions:

- `Interp(AslTree T)`. This is the constructor of the interpreter. It prepares the AST for the interpretation. It also creates a table that maps function names into AST nodes (the roots of the functions). The parameter `T` is the root of the AST.
- `void Run()`. It runs the program by interpreting the AST.

Inside this file, there are some essential functions that constitute the core of the interpreter. They execute function calls and instructions and evaluate expressions.

- `Data executeFunction(String funcname, AslTree args)`. This function executes a function with a given name in which `args` contains the AST of the list of arguments passed by the caller. The function also implements the parameter passing mechanism.
- `Data executeInstruction(AslTree t)`. This function executes an individual instruction. In case the instruction returns some data (i.e. a `return` instruction or some block of instructions that contains a `return`), this function returns the data. Otherwise, a `null` is returned.
- `Data evaluateExpression(AslTree t)`. This function returns the value produced by the evaluation of the expression represented by an AST.

During the execution of a program, the interpreter performs various semantic and execution checks (access to undefined variables, operations with incompatible types, division by zero, etc.) that may raise runtime exceptions. The interpreter also keeps track of the line number of the source program for an appropriate debugging when an exception is produced.

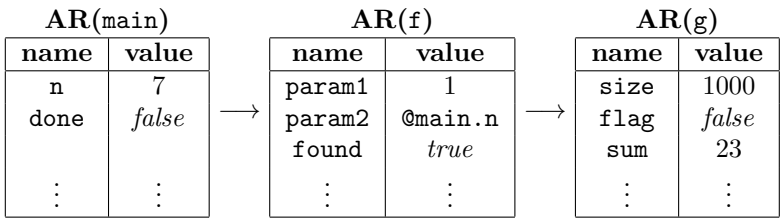

Figure 3: Dynamic organization of the memory as a stack of activation records(AR).

2.1.2 Data.java

This class represents a data item stored in memory. It provides methods to retrieve the type and value of the data and perform arithmetic, logical and relational operations. It also contains methods for semantic and runtime checks.

2.1.3 Memory.java

This class represents the memory of the virtual machine that interprets the program. `Asl` does not have global variables, thus all the visible variables are local. Data can only be shared through the pass of parameters or return of results. Parameters can be pass by value or reference.

Figure 3 depicts the structure of the memory. Each function has an associated *activation record* (AR) that stores all the variables in the scope. When a new function is called, a new AR is created. The AR is destroyed when the function terminates. Therefore, the memory is organized as a stack of ARs.

Each AR contains a set of entries with pairs $\langle$`name`, *value*$\rangle$. When an entry represents a parameter passed by reference, the *value* is a reference to another entry in another AR. The entries in the ARs are created at runtime each time a new variable is defined.

2.2 AslTree.java and AslTreeAdaptor.java

These two files contain two classes that are required to extend the information of the AST nodes. ANTLR allows this extension by creating a subclass of the default class to represent AST nodes (`CommonTree`).

The extension implemented in `Asl` is very simple and allows to store the integer and Boolean literals as Java `int`'s. This avoids the burden to parse a text representation each time the literal needs to be read.

The class `AslTree` can be further extended for other purposes.