ARG!
Language Reference Manual

Ryan Eagan, Mike Goldin, River Keefer, Shivangi Saxena

1. Introduction

ARG is a language to be used to make programming a less frustrating experience. It is similar to C in its syntax, but is dynamically typed and fully type inferenced. The language focuses on making the coding simpler for users, and is especially good for scientific computing. It includes standard library functions for scientific computing and array manipulations, including easy sorting of array members.

2. Lexical Elements

There are six types of lexemes in ARG - identifiers, keywords, operators, separators and whitespace.

a. Identifiers

Identifiers are tokens that refer to variable or function names. They always start with a letter and may have an underscore or digits in them. Uppercase and lowercase letters are distinguished. Identifiers must be unique for each variable within the same scope.

b. Keywords

These keywords cannot be used as identifiers. They include:

```
if, else, return, while, STOP, FUNCTION
```

c. Operators

Operators are used for performing arithmetic operations, establishing precedence, casting and, in the case of the ! operator, modifying the STOP keyword.

```
-, +, *, /, (), !, unary -, sizeof, typeof, ==, !=, <=, >=, (int),
                (char), (double), (bool), [<Int>]
```

d. Separators

Separators are used to delimit code blocks, identifiers and statements. ARG includes the following separators:

```
{ } , ;
```

Whitespace is also a separator, but not a token.

e. Whitespace

Whitespace is used to indicate separation between two tokens. It is ignored unless specified as part of a character array. It may or may not be used between operators and operands. Any extra whitespace is treated as one white space.

f. Comments

`/*` indicates the start of a comment and `*/` indicates its end. Multi-line comments are supported.

3. Data Types

There are two types of data structures -

1. **Primitive**
    a. *Int*

    These are 32-bit structures that will only hold numeric values in the range −2,147,483,648 to 2,147,483,647.

    b. *Char*

    These are 8-bit structures which can contain any member of the ASCII character set, including escaped characters.

    c. *Double*

    Double values are 64 bits and include 1 sign bit, 11 exponent bits and 52 significand bits.

    d. *Boolean*

    Booleans are 8 bits and represent a logical true or false. All types evaluate to true except for the explicit boolean false.
2. **Derived**
    a. Arrays

    Arrays are collections of objects of the same type, stored contiguously in memory. They are derived by appending the `[<Int>]` operator to an identifier, where `<Int>` is an integer specifying the array's size.

4. Literals

a. Int Literal

An Int literal can be any decimal number 0 - 9 and may be preceded by the unary -.

b. Char Literal

A character literal can be any ASCII character, delimited by single quotes. They correspond to Char arrays.

c. String Literal

A string literal can be any set of ASCII characters, delimited by double quotes.

d. Double Literal

A double literal represents numbers with fractional values. They may include either a decimal point with a fraction or an 'e' followed by a signed integer. It can have a maximum value of 1e+63. Floats may be preceded by the unary -

e. Boolean Literal

Structures of this type may have one of two values: *true* or *false*. They are not strings.

5. Operators

a. Unary

- sizeof

Returns the size of an object in memory, in bytes.

- typeof

  Returns the datatype of the objects.

- -

  Negation for Ints and Doubles

- Casting - (int), (char), (double), (bool)
- !

The ! operator may be appended to the STOP keyword to modify the level of nestedness from which STOP should break out of.

- [<Int>]

  Appended to the right of an identifier, specifies that the identifier should be an array of size <Int>.

b. Binary

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Less Than/equal to (<=)
- Greater Than/equal to (>=)
- Equals (==)
- Not equals (!=)

c. Assignment (=)

- **Operator Precedence**
  a. Unary operators
  b. Multiplication, Division
  c. Addition, subtraction
  d. Relational operators (<=, >=, ==, !=)
  e. Assignment operator

  Operators at the same level are grouped left-to-right.

6. Program Structure

a. Declarations

Since ARG does automatic type-inferencing, the data type of a variable need not be specified. Declarations have the form: *identifier* = *value*.

b. Statements

i. Expression statements

These are statements which evaluate to some result which can be assigned to a variable, tested for logical truth or thrown away. These generally include assignment statements or calls to functions. They are of the form:

```
expression ;
```

ii. Conditional statements

Conditional statements evaluate the logical verity of an expression and branch execution on the basis of the evaluation. These statements may be of the form:

```
if (<boolean-expression>) statement;
if (<boolean-expression>) statement else statement;
```

iii. Iteration statements

Iterators are loops which run until a specified expression evaluates as true. Syntax-

```
while (<boolean-expression>) statement;
```

iv. return statement

Return is used to return values from one function to its caller. It can be used to return zero or more values. Syntax-

```
return ;
return expression;
```

Compound statements in any block can be put inside curly braces, e.g. -

```
{
    expression1;
    expression2;
}
```

v. STOP! statement

This statement is used to iteratively break out of code blocks. Each exclamation mark after the STOP! statement breaks out of one block. The program counter is placed at the point immediately following the last code block broken out of.

vi. FUNCTION statement

The FUNCTION keyword simply denotes the declaration of a new function. Minimally, it must be followed by an identifier, a left-paren, a right-paren and a curly-brace delimited code block. Optionally, arguments can be specified by placing comma-delimited identifiers within the parentheses.

c. Scope

In ARG, curly-braces delimit code blocks and scope. The "global" scope is itself a code block implicitly wrapped in curly-braces by the compiler. Identifiers declared in outer code blocks are accessible by inner code blocks, but the opposite is not true.

7. Example

```
/* argv[] and argc[] are available as global variables. */
countto = argv[1];   /* A Char */
attempts = argv[2];  /* An Int */

/* result may be a boolean or a char. */
result = counttochar(countto, attempts);
```

```
/* Anything which is not a boolean false is true. */
if(result == true) {
  /* Print result, a char in this instance but possibly any type, as a
string. */
  PRINT("%x\n", result);
}

/* Exit by breaking out of the program block. */
STOP!;

/* This function that uses integers to count up to a character. It can
return any type */
FUNCTION counttochar(countto, attempts) {

  /* Idiomatic (ARG!) presumes failure. */
  result = false;

  i = 0;
  WHILE(i < attempts) {

    IF(i == countto) {
      /* Set result, previously a bool with the explicit value false,
to a char with the implicit value true. */
      result = countto;
      /* Jump out of the if block and the while block. The use of two
exclamation marks specified two levels of nest breakout-ness. */
      STOP!!;
    }

    i++;
  }

  return result;
}
```