ALBATROSS

Language Definition

CS 473: Compiler Design / Fall 2022

The Albatross language is an imperative language very similar to the C language. A program in Albatross has the form:

```
<variables>

<functions>

<statements>
```

Where:

- **Variables**: a sequence of global variables available to the rest of the program. All global variables are defined here.
- **Functions**: a sequence of functions available to the rest of the program. Functions may be defined in any order, and function `f` can call function `g` not yet defined, as long as function `g` is defined later in the same program.
- **Statements**: a list of statements that can call the functions defined above, and that can use the global variables defined above.

Types

Albatross has a few built-in types, and does not provide the ability for programmers to define their own types:

- Integer: Signed integer numbers which are 4-bytes long, denoted by `int`
- Character: 1-byte long characters, denoted by `char`
- Arrays of integers: Sequence of integers located sequentially in memory, denoted by `int[]`
- Strings: Sequences of characters located sequentially in memory, denoted by `string`
- Arrays of characters: Same as strings, and can be used interchangeably, denoted by `char[]`
- Void: Only used as a return type for functions, means that the function does not return any value. Denoted by `void`

Variable definition

All variables, global and local, are defined in the same way:

```
<name> <type> := <expression> ;
```

Where:

- **Name**: is the name of the variable. A global variable name is accessible for the rest of the program, a local variable name is accessible for the function in which it is defined.
- **Type**: the type of the variable.
- **Expression**: An expression that evaluates to a value of the correct type, which is the initial value for the variable. Initializing variables is mandatory in Albatross.

Function definition

Functions in Albatross are defined as follows:

```
<name> <type> ( <arguments> ) { <variables> <statements> }
```

Where:

- **Name**: is the name of the function, available anywhere in the same program (even before the function is defined).
- **Type**: is the return type of the function.
- **Arguments**: is a list of arguments. The list can be empty for functions that do not accept any arguments (i.e., `( )`). Each argument is defined as `<name> <type>` where **name** is the name of the argument, available as a variable inside the function body; and type is the **type** of the argument. Arguments are separated by commas.
    - Example: `(argument1 int , argument2 string)`
- **Variables**: a sequence of local variables available to the function
    - See section Variable definition above
- **Statement**: a list of statements which constitutes the body of the function. The statements should be executed in the order they are defined.

Statements

The Albatross language has the statements defined below:

- **Set**: Takes the form `<lhs> := <expression> ;` and means: execute `expression` first, then assign its value to lhs.
    - **lhs**: short for "left-hand side", can be a variable or an array access.
- **Return**: Takes the form `return <expression> ;` and means: execute `expression` first, then use the resulting value as the return value of the current function. When used outside of a function, this statement can return an integer as the exit code of the program. `expression` is optional on functions that return `void`.
- **If**: Takes the form `if (<expression> ) then { <statement> } else { <statement }` and means: execute the `expression` and execute the `then` statement if the result is not zero, otherwise evaluate the `else` statement (if the result is zero). The `else` portion is optional.
- **While**: Takes the form `while (<expression> ) { <statement> } otherwise { <statement }` and means: execute the `expression`. If it the value is not zero, execute the body (first `statement`), and repeat (i.e., execute the `expression` again). If the value is zero, finish executing the `while` statement.
    - When the `while` statement never executes the body, it executes the `otherwise` statement once. The `otherwise` statement is optional.
- **Repeat**: Takes the `repeat (<expression> ) { <statement> }` form and means: execute the `expression` once, which should evaluate to an integer. Then, repeat the statement that many times without re-evaluating the `expression` again.

Expressions

- **Integer constants**: Take the form of a number (e.g., 473) and results in the value of that constant.
- **Character constants**: ~~Take the form of a character surrounded by quotes (e.g., 'a') and results in the value of that character.~~ Not supported.
- **String constants**: Take the form of a sequence of characters surrounded by double-quotes (e.g., "CS473") and evaluates to that string.
- **Array constants**: Not supported
- **Binary expressions**: Take the form `<expression> <operator> <expression>` and means: execute each expression and perform the operator on the resulting values. The order of evaluation is not defined and left up to each implementation. Binary expressions are not applicable to strings and arrays.
    - **Operators**:
        - **+** Sum/addition
        - **-** Minus/subtraction

        - ***** Times/multiplication
        - **/** Over/division. The behavior on a division by zero is not defined and left up to each implementation
        - **%** Remainder. The behavior of a remainder by zero is no defined
        - **&** Bitwise AND
        - **|** Bitwise OR
        - **^** Bitwise XOR

- **Logical expressions**: Take the form `<expression> <operator> <expression>` and means: execute each expression and perform the logical operation on the resulting values, resulting in zero for false and not zero for true. The order of evaluation is not defined and left up to each implementation. Logical expressions are not applicable to strings and arrays.
    - **Operators**:
        - **&&** Logical AND
        - **||** Logical OR
        - **<** Less than
        - **<=** Less than or equal to
        - **>** Greater than
        - **>=** Greater than or equal to
        - **==** Equals
        - **<>** Not equals
- **Unary operators:** There is only one such operator, which takes the form `! <expression>` which evaluates to zero if the expression evaluates to non-zero, and evaluates to non-zero if the expression evaluates to zero
- **Variable access:** Takes the form `<name>` and evaluates to the value stored in the variable `name`
- **Array access:** Takes the form `<name> [ <expression> ]` and evaluates `expression` to an integer index, then accesses the array stored by the variable `name` at the given index.
- **Function call**: Takes the form `<name> ( <expressions> )` where `name` is a function name and `expressions` is a comma-separated list of expressions. Evaluates each expression and calls the function by passing the resulting value of each expression as the argument in the same position.
    - The order of evaluation of the expressions is not defined and left up to each implementation
    - When the function terminates (by executing a return statement), the program resumes execution immediately after the function call expression.

Comments

Comments in Albatross start with the character # and last until the end of the current line.

Operator precedence and associativity

The table below explains the operator precedence and associativity in the Albatross language. Lower numbers of precedence mean higher precedence (i.e., top of the table has higher precedence than bottom of the table).

Precedence	Operator		Associativity
1	Function call, array access	() []	Left-to-right
2	Logical NOT	!	Right-to-left
3	Multiplication, division, remainder	* / %	Left-to-right
4	Addition, subtraction	+ -	
5	Relational operators	< <= > >=	
6	Equality	== <>	
7	Bitwise AND	&	
8	Bitwise XOR	^	
9	Bitwise OR	|	
10	Logical AND	&&	
11	Logical OR	||	
12	Assignment	:=	

Scoping Rules

Variables, local or global; and arguments have **lexical scoping** in Albatross. If a local variable or argument has the same name as a global variable, then the body of such function cannot access the global variable. Different variables in the same scope with the same name are not allowed. Different functions with the same name are not allowed. Functions with the same name as global variables are not allowed.

Intrinsic Functions

The following functions should be available to all Albatross programs. Attempting to define a function with the same name results in a compilation error:

- `exit void (int)` – exits the program with the return code provided by the argument
- `printint void (int)` – prints the integer to the screen
- ~~`printchar void (int)` – prints the character to the screen~~
- `printstring void (string)` – prints the string to the screen