3. Grammar of AleC++

Previous chapter dealt with the way characters form symbols according to lexical rules. Grammar of a language determines a set of legal ways those symbols can be combined. The grammar of AleC++ encompasses basic grammar (discussed in this chapter), the grammar of object programming, and constructs for simulation description (following chapters).

The overview of the grammar of AleC++, which will be given will be rather brief since the authors assume the reader already knows C/C++. The instructions given will be concentrated on the constructs of AleC++; if the reader needs further explanation the authors recommend consulting any of the manuals of C/C++.

3.1. Declarations

Declarations introduce a new symbol, give its characteristics, type, attributes, and in certain cases the memory allocation of the symbol. Declared symbols point out the objects with certain characteristics. The most important are:

Type - if a set of allowable operations, memory space, and correct bit-pattern is to be determined for an object, the object need to be of a certain type.

Allocation - an object can be allocated in a stack if the local variable is in the static area, that is if it is static or global variable.

Duration - time for which an object legally occupies a particular memory location

Visibility - a part of the text allowing access to the object by giving the name of the object

Category - (**AleC++**) - beside the standard C/C++ categories, object can be a signal, an element. The status of an object will depend upon its category

3.1.1. Basic Data Types

AleC++ has intricate (basic) and composite (derived) types of data. Basic types are:

`void` -	absence of type; used for counters and special applications
`char` -	character; 1 byte
`int` -	integer (UNIX implementation - 4 bytes)
`double` -	real type - double precision (UNIX - 8 bytes)

Note**:** Alecsis does not support unlabelled (`unsigned`**)** operations; type `float` is a synonym for `double`; all integer types are of the same length being 4 bytes. It follows from the previously said that the following types are equivalent:

```
int
short
long
unsigned
short unsigned
unsigned long int
```

Note**:** `unsigned char` is not a legal combination, since all data of `char` type are labelled.

The imposed restrictions are a consequence of the characteristics of the simulation engine that performs the instructions of AleC++ code. Virtual processor is the software emulator of the real microprocessor and it performs those instructions. The need for greater efficiency and speed of the virtual processor eliminated certain types of real (machine) types. These reductions do not limit the capabilities of the language due to the specialized application of AleC++. The reader should keep in mind that according to ANSI-C standard lengths of integer types are interrelated by:

```
short ≤ int ≤ long,
```

which is valid for AleC++, as well.

New symbol can be defined in a way generic to all types:

```
type t;      // variable of type "type"
type *t;     // pointer to a variable of type "type"
type t[2];   // t is a vector of type "type", length 2
type t[2:0]; /* t is an inverted vector of type "type", length 3
             (AleC++ specific !) */
type &t;     // t is a reference (address of data) of "type"
void f(type);    /* f is a function that takes one argument of
                 type "type" and does not return a value */
void (*f)(type); /* f is a pointer to a function that takes one
        argument of type "type" and does not return a value */
```

In general, a simple declaration consists of a type, and declaring phrase, in the following way:

declaring phrase:
identifier
ptr_operator declaring phrase
(declaring phrase)
declaring phrase [constant_expression]
declaring phrase [constant_expression : constant_expression]
declaring phrase (declaration_parameters)

ptr_operator:
_* <cv_qualifier>_
&<cv_qualifier>

cv_qualifier:
`const`
`volatile`

Qualifiers `const` and `volatile` appear within the declaration of pointers and references, but can appear within a type, as well. In C-jargon, l-value is the expression, which can appear on the left side of the "=" (assignment), that is its value can be changed. All other expressions are called r-value, and they cannot be changed. Qualifier `const` means the object can (and must) be initialized, but that nothing can be stored in it (not an l-value). This is done to protect certain object from change, such as the truth tables of logic operators, various physical constants, etc. You can read more about these qualifiers in the next chapters.

3.1.2. Allocation method

Allocation method is the mechanism by which objects are created during the program execution, or simulation. AleC++ recognizes three ways to do this operations; all of them are marked by key words **`auto`**, **`static`**, and **`extern`**. Due to the characteristics of the virtual processor of the simulator the fourth key word, **`register`** does not produce any effect, and has the same meaning as **`auto`**.

3.1.2.1. Auto

Local variables created in the stack, which expands and contracts during the execution stage are marked this way. The duration of these objects is measured from the point of declaration, and ends with the visibility of the object. It is a mistake to process or use the address of such an object beyond the existence range, because they do not exist outside that range. The formal parameters of functions can be defined in this way, only.

3.1.2.2. Static

Static objects are declared at the beginning of the execution phase, and last trough the simulation. In case of a function, the value form the previous call will be used. If a static object is declared on the global level (beyond the function) its existence range is the length of the whole file. In this case the object cannot be externally connected or referred to.

3.1.2.3. Extern

These object are physically situated in some library, and the declaration give the information about the type (but not about the size, since for multidimensional object the first dimension can be left out.) The `extern` declaration makes the object declared in some other file visible in the current file, too.

These objects are classified as static according to the duration; and as global, according to the existence range. During the time of interpreting their final address is not known, which warrants an additional phase before the simulation - linking references of external symbols with the addresses of the objects raised from the library.

Declaring the allocation method is not necessary (**auto** is understood for local objects). In absence of the type **int** is understood.

```
extern char cmatrix[][255];// the first dimension is not necessary
static char *s="static string";
auto i = 2;                    // understood as type int
int i = 2;                     // understood as type "auto"
```

3.1.3. Declaration of new types

AleC++ does not fully recognize new types declaration, since it uses structural and not full type equivalence. According to the structural equivalence, the types are equivalent if they can be separated into the same basic types, while according to the full equivalence those two types would have to have the same type name, without regards to the common basis. To circumvent this obstacle synonyms are created using the key word typedef. A name created using typedef can be used where standard types can.

```
typedef int Meters, Kilometers;
```

We have declared two new names for `int` type. Structurally, those two types are equivalent due to the same basis although the semantics can be incorrect. The syntax of `typedef` is identical as the allocation method description:

```
typedef int  *iptr, ivec[20], imat[20][50];    // new types
static  int  *ptr, vec[20], mat[20][50];       // new static variables
iptr ip;      // ip is pointer of int
ivec iv;      // iv is vector of type int of length 20
imat im;      // im is matrix of type int of length 20x50
```

3.1.4. Address alignment

The address alignment is of importance for installation of Alecsis on different hardware. Great number of microprocessor needs for the addresses of all operands to be divisible by the length of the machine word. On DOS-run computers all addresses need to be even (words are 2 bytes long), while the most of the UNIX-run computers need the addresses to be divisible by 4 (words are 4 bytes long). Certain number of processors (ex. MIPS) needs the addresses to be of double type and divisible by 8. Since the virtual processor of Alecsis works as an emulator of the real microprocessor, it has the same characteristics as the microprocessor of the computer where it is installed.

Alignment method of the computer is easily checked by printing out the addresses of various types and their division by 2, 4 or 8.

Address alignment is of importance only in the installation phase. In Alecsis `Makefile`, flags for alignment are defined. After the installation, address alignment is not of importance for the designer.

3.1.5. Structures

Structures is the common name for three composite types - **`structures`, `unions`, and `classes`**. These represent a collection of heterogeneous data declared without order or number. The total length of structures and classes is equal to the sum of lengths of individual members (or somewhat larger due to the address alignment). Unions have the length equal to the length of the **largest member**. Structures and classes can carry all of their elements simultaneously, while unions can carry only one (since all member of a union begin from the same address). Structures can generally have a name (tag), and may not in case a new type is formed:

```
struct S { int a; double b; char c[30]; };
class C { struct S s; double a[20]; };
typedef union { int i; double d; } INT_OR_DOUBLE;
```

The names of the structures can be used as new types (C++). In C, the name of the structure cannot be used if it is not preceded by the key word `struct/union`. The definition of class K from the above example can be:

```
class K { S s; double a[20]; };
```

If the name of the structure is masked by the name from another visibility area, the key word `struct/union/class` can resolve the ambiguity.

In version 2.0 C++ structures and classes have the same characteristics, except all members of structures have `public` right of access, while classes are `private` (unless defined otherwise). The authors implemented the old C rules, reading that a **function cannot be a member of a structure**. If it makes you a problem, instead of a structure, you can use a class with all members explicitly set to `public`.

A definition of a structure can simultaneously be a declaration if variables are added:

```
struct S { int a, b; } s1, s2;    /* defines structure S and
                                     variables s1 and s2 of type S */
struct { int a, b; } s3;  // a structure without a name
struct { int a, b; };     /* a futile definition,
                          interpreter reports an error */
```

Bit-ranges are a separate structural type. Their meaning is fundamentally different from ordinary members, although they are declared similarly:

```
struct flags {
    int f1:2;
    int f2:3;
    int f3:1;
};
```

Member `f1` occupies 2 bits, `f2` - 3 bits, and `f3` - 1 bit. The total length of the structure is 6 bits, but due to the address alignment at least one word (4 bytes) will be allocated. Access to the members is the same as with the ordinary structure, but the user need to take care that the assigned values do not overflow the range defined by the number of bits following the character.

3.1.6. Enumeration type

Enumeration constants were mentioned in the previous chapter. To remind you, in AleC++ enumeration constants can be used only in the context of the variables of the same type. This implies that by the definition of the enumeration set a new type is declared:

```
enum bit { '0', '1'};                  // enumueration with name
typedef enum { '0', '1', 'x', 'z' } four_t; // enum. as a new type
enum { send, recv };    // futile declaration - error
bit b[4]="0011";        // vector of type bit
four_t s = 'x';         // scalar of type four_t
```

3.1.7. Multidimensional variables

Standard variables can be considered scalars, because a unique memory location exists for every one, which can be accessed using the variable name. Vector variables reserve memory space for as many members as indicated in the vector dimensions.

```
int i[10];   // vector of type int of length 10 from i[0] to i[9]
double m[5][10]; // matrix of type double of length 5x10
```

As in C, all multidimensional variables have 0 offset (vector *v*, of length *n* reserves locations *v*[0]-*v*[*n*-1]. Beside that, new allocation methods are introduced in AleC++, since all vectors are used for modelling of abstract registers (signal vectors and node vectors). If the length of the vector is defined using upper and lower limit the vector can have a nonzero offset. In case the upper limit is smaller than the lower we are talking about inverse direction:

```
int i1[10];        // vector of length 10
int i2[0:9];       // equivalent declaration, but with limits
int i3[1:10];      // vector of length 10, with the offset of one
int i4[9:0];       // vector of length 10 with inverse direction
int i5[10:1];      // same, w/ the offset of one
```

Note: it is incorrect to access indices outside their declared range. Location `i3[0]`, for example does not belong to the vector `i3` the same way the location `i3[11]` does not. Vectors with the inverse direction behave the same as the direct vectors except the order of assigning the values is reversed during the initialization.

```
digital3 v1[0:3] = "010x";      // position v1[0] has value '0'
digital3 v2[3:0] = "010x";      // position v2[0] has value 'x'
```

If the name of a multidimensional variable is found in the expressions it is implicitly converted from "vector v of type t" to "pointer of location [0] of vector v of type t." If the lower limit is not 0, the pointer will be point to the first element marked by the limit (`[1]` in vector `i3`). Inverse direction vectors will have the pointer show the element with the lower index (`[0]` for vector `i4`).

3.1.8. Initialization

The rules of initialization in AleC++ are slightly different compared to C++, in order for the inverse direction vectors to be correctly initialized. Object with the automatic allocation can be initialized using user-defined expression, but only if the expression evaluates to a scalar. Static or global objects can be initialized using constant expressions only; initialization of composite objects (vectors, matrices, strictures, etc.) is legal, as well.

```
int i = 2;       // automatic initialization
int j = i+1;     // this too
static char *s = "constant string";   // O.K. - static object
static char *s[] = {"first", "second", "third", "end"};//composite
double m1[][3] = { { 3,  3, 2},
                   { 2,  2, 1},
                   { 0, -1, 2} }; /* every char '{' reduces the
                                     dimension by 1 */
double m2[][3] = { 3,  3,  2,
                   2,  2,  1,
                   0, -1,  2 }; /* O.K. - interpreter can find
                                     its way around this */
struct S { int x; double d; char *s; };
S s1 = { 1, 2.2, "string" };
S s2[2] = { { 1, -2., "s1"}, { 4, 5.6, "s2" } };
```

As you can see the interpreter can calculate the first dimension of a vector based on the initializing phrase. Composite object is initialized from left to right according to the order of specific values in the initializing phrase, with the exception of the inverse direction vectors, which are initialized from right to left. Alignment of object types and initializing phrase is controlled during the initialization according to the standard assignment standards (=) with the implicit conversion, if necessary.

Static objects are initialized once, at the beginning of the program work cycle. Dynamic (automatic) objects are initialized every time the flow of the program comes across an initialization, while the present initialization ceases to exist with the object itself.

3.1.9. Declaration of functions

Functions occupy the central place in AleC++, which supports all rules of declaring and using functions in C/C++.

Note: Before the function call you need to define the function declaration (consist of the type the function returns assigned to its name, and a string of parameter, if any). Function that returns no value is labelled `void`**.** Parameter of the function that does not accept any parameters is also labelled as void.

Note: There is a fundamental difference in the way of declaring functions in old and new ANSI-C:

```
int f1 ();
int f2 (void);
double **f3(char *, double)
char *f4 (int (*f)(int, double));
```

According to the old C (Kernighan-Ritchie) function `f1` returns `int` and accepts unlimited number of parameters. In ANSI-C, C++, and AleC++ the function returns `int`**,** but does not accept any parameter (identical with the declaration of `f2`**).** Function `f3` returns the pointer to pointer to `double` type, and accepts two parameters: one pointer to `char` type, and another of `double` type. The last function, `f4`, returns the pointer to `char`, and accepts one parameter that is the pointer to the function that returns `int`, and accepts two parameters - of `int` and `double` types.

In the declaration of the prototypes of functions the names of the functions are not important, since the prototypes (parametric profiles) of functions are used for check of the legality of the function call (regarding the number and types of the actual arguments compared to the expected ones). We use prototypes of functions in case of overload (many functions with the same name, but different parametric profile.) To summarize, we use function declaration:

- to check the agreement between the number of expected (formal) parameters, and the number and types of the actual arguments at the time of the function call;
- for application of standard conversion types according to the mechanism used in initialization or assignment (=);
- to reach a decision about the nature of the overload function (overload will be discussed in detail in the next chapter.)

3.2. Definitions

An object is declared for the purpose of giving information about its type, dimensionality and other aspects needed for its processing. In case declaration gives information about the memory reserved for the object, as well as the information about the internal structure, it is a **definition**. If on the global level, declarations:

```
extern int vec[2][3];
extern char s[];
```

are declarations only, for they are used in order for the interpreter to create a correct code of expressions in which matrix `vec`, and string `s` are used. Declarations

```
int vec[2][3];
char s[20];
```

are at the same time definitions, since they give the information to the interpreter to reserve 2x3x4=24 byte of static memory for the matrix `vec` and is 20x1=20 bytes of memory for string `s`. **There can be many declarations of the same object** within the same file (and all the other files added by `#include` preprocessor directive), **but only one definition.** Global (or static) data or functions can be defined on the global level. AleC++ introduces two more fundamental objects that can be defined - **module** and **module card**, but you can read about that in the chapter about the simulation.

3.2.1. Definition of functions

All functions are defined on the global level. Definition of functions is similar to their declaration, with two extra conditions:

- The names of formal parameters need to be specified (if the parameter are needed), beside the types.

- It is necessary to define the body of the function in the extension of the prototype consisting of the array of declarations and commands enclosed by parentheses '{' and '}'.

```
main () {
    printf ("Hello, object-oriented world of simulation!\n");
}
```

In C, functions can be called by other functions according to the program algorithm. The program begins from the **`main`** function. In AleC++, functions are usually called from concurrent processes during the simulation; however, it is legal to write a program that has only C or C++ constructs, and the function `main`. If no module labelled **`root`** exists (a simulation counterpart of `main` function) in the originating file the simulator will deliver the address of the function to the virtual processor to commence execution.

Therefore, AleC++ can be used as an interpreter for C/C++. It is useful for development of new functions. A part of some complex component model can be developed as a library function. Such functions can be tested in the same way as in C before the simulation constructs are added. More reliable, and more error-free models are created this way.

AleC++ is designed as a superset of C/C++. However, some constructs of C++ are not allowed in the current version as we were oriented mostly to construct that we need for simulation. **Not implemented** are:
- virtual functions,
- virtual base classes,
- conversion functions,
- copy constructor,
- implicit conversion using constructors,
- `new` and `delete` for vectors.

All variables in a function are allocated in the stack (including the formal parameters), except the ones explicitly defined as static. Formal parameters create special visibility area, which is narrower then the global level. By creating a new visibility area inside a bigger one, in general the possibility exists for the variable under the same name to be declared without the danger of redefining it. In this case the variable in the wider area masks the one in the narrower, until the narrower area is left.

```
int i;              // global i is visible
f1 (int i, int j) { // formal i masks global i
    int j;          // local j masks formal j in the whole fn
    if (i>>2) {
        int i=3;    // formal i is masked, as well
        ...
    }
    ...             // formal i is visible again
}       // the body of the fn is closed
// the area of the formal parameters is closed
// global i is valid again
```

Actual arguments (expressions) are stored in the stack during the function call, according to order of specification, and are passed by **value.** Passing the argument value to functions (and not the argument address) is a characteristic of C, and AleC++ supports this mechanism. It is possible to pass an argument according to the address if the formal parameter is the argument reference (e.g. `&a`). You can read more about that in the chapter on object programming.

Passing by value means the functions gets a copy, and not the original of the parameter, so the change in the copy does not affect the original. **All objects** are passed by value including structures or large classes, except multidimensional variables, which are converted into pointers to the first element. This means that the original and not a copy of the pointer is passed to the function, which implies that matrices, and vectors can be changed inside a functions. In order to avoid any unwarranted change in these objects, the formal parameter can be labelled **`const`**, which prevents accidental change of a vector or a matrix.

⚠ AleC++ does not support the old way function parameters declaration during their definition. This means that definition

```
int f1 (a, b) double a, b; { ... }
```

is not correct. Function f1 has to be defined in the following manner:

```
int f1 (double a, double b) { ... }
```

3.3. Expressions

All legal combinations of identifiers and operators, which follow syntax rules, are expressions. Among these are rules on right-a-way of the operator, alignment of types, etc. Usually, these expressions are combinations of symbols and arithmetic-logic operators.

Note: The result of the application of the operator on the expressions is also an expression.

AleC++ supports all rules for expressions of C/C++, and introduces some new ones, which will be discussed in detail in the chapter on simulation.

An expression is constant if it is entirely made out of combinations of constants. A vast number of situations in AleC++ demand a constant expression. Alecsis evaluates the constant expression during the compilation. Therefore, the expression

```
50 * 2 / 5      // can be evaluated
```

does not cause creation of instructions for multiplication and division for the interpreter, since the compiler evaluates it as an integer constant 20. In a similar way, implicit conversion, in case of combination of a variable, and a constant can be accomplished without the creation of instruction by a direct conversion of the constant. Example is:

```
double d;
d+1;            // 1 is converted into 1.0
```

An expression is unary if it consists of operators and expressions, and binary if the operator needs two operands, which are also expressions. All expressions in AleC++ are unary or binary, with the exception of conditional expressions (as in C/C++), which demands three expressions:

conditional_expression:
    expression1 ?expression2 : expression3

The number of possible combinations in expressions is very limited, which is why the interpreter in dealing with binary expressions with different types applies the standard rules of conversion or **promotion,** in order to have both operands as of the same type. Promotion in arithmetic expressions means that the expression of a 'lower' type is converted to the 'higher' type, which becomes the type of the result. In the case of assignment, the rules demand for the right-side expression to be converted to the type of the left-side expression before the operation is performed. The rules of promotion and conversion of types are applied in the following cases:

- in all binary expressions;
- during the initialization;
- before the actual arguments are passed to the function;
- when the result of the function is returned using the command `return.`

In all these cases, except the first one, the rules of conversion of the assignment operation (=) are applied.

```
2 + 3              // constant expression
a ( 2, 3 )         // function call with two arguments
b[4]               // indexing of a vector
s->>m.a            // selecting of structure members
a >> b             // relation expression
++a, a++           // left and right increment
a += 2             // appended addition and assignment
a,b,c              // coma operations(developed from left to right)
a==b ? c: c+1      // conditional expression
a + (b + c)        // grouping using parentheses
```

3.4. Commands

The program is a set of declarations and commands. The meaning of a command is to change the state of the system by its execution. If that is not the case it is a null-effect command. The simplest command of AleC++ is an expression finished by a semicolon “;”.

```
i+2;
```

This command is legal, but has no effect, since nothing happened after the execution. Most of the commands include an assignment operation or a function call. Commands can be simple or composite. Composite commands are a set of commands and/or declarations enclosed by parentheses {}. This **block** creates a new name space. This allows for the masking of the variables with the same name in the external space.

All commands are executed according to the order of specification. The flow of the execution can be controlled using branching commands, **`if/else`.** A sequence of numbers can be repeated by using loops - **`while`**, **`do`**, and **`for`**. A loop can be interrupted using command **`break`**. A present loop cycle can be interrupted, using command **`continue`** (new cycle begins after that). Functions can return expressions to the environment they have been called from using **`return`**. If testing the value of a scalar integer expression using multiple options is needed, the command **`switch`** can be used.

Identifier **label** commands by putting the label and colon in front of the command. The purpose of the label is to mark the place for unconditional jump using **`goto`**. Note: The use of `goto` is recommended only in the extreme cases where the speed is more important the legibility of the code.

Two characteristics of C++ are adopted, that do not exist in C. First, commands and declarations can freely intermix within a block (C demands that declarations are placed at the beginning of the block, before commands). Second, the first of the three expressions in the `for` loop specification can be a declaration. For example:

```
{
  int i=1, j=2, k;          // one declaration
  k = i + j;                // command
  int l = k+1;              // declaration, again
  for (i=0; i<10; i++);     // empty for loop in C-style
  for (int q=0; q<10; q++);  /* q is in the same visibility area
                                 as the other variables */
  double d = k-j;           // another declaration
}
```

New variable, introduced during the initialization of the `for` loop in the example has the same existence range as the others, i.e. it is visible in the block where `for` loop is.

Command **`asm`** is unique to AleC++. It enables direct insertion of the assembler instruction in the C++ -like code. The assembler code for virtual processor is prepared by the compiler, but it can be done directly by the user, using assembler instructions. The set of instructions resembles the instructions for Motorola processor series 68020/30, and it is given in the Appendix. Programmers of low level libraries, and functions used frequently in modelling will find this to be effective shortcut to writing fast programs by eliminating unnecessary instructions.

This is the syntax for **`asm`**:

assembler:

  asm *asm_text ;*

  asm *{ sequence_asm_lines }*

sequence_asm_lines:

  asm_text

  sequence_asm_lines asm_separator asm_text

asm_text:

  opkod<.optip> <operand1> <, operand2>

asm_separator:

  '\n'

  ';'

This an example of the command:

```
void f1 (int i, int j) {
    int l=3, m;
    asm {
        add.l        i, j
        movq.l   %d4, %d0
        add.l        l, 2
        sub.l        %d4, %d0
        movq.l   m, %d0
    }
}
```

In the previous command variables `i` and `j` were added together, first. Suffix l marks the 'long' version of the instruction (suffix `b` means 'byte', and `d` 'double'). The result of addition is placed in the accumulator (`%d0`). It is temporarily moved to `%d4` registrar (instruction `movq`) to free the accumulator for other operations. Local variable `l` is added together with constant `2`, and the result is subtracted from the value in the register `%d4` (`sub`). Finally, the result is assigned to the local variable `m`. As you can see from this example, even simpler expressions, names of variables, etc. can be operands of instructions.

You can translate AleC++ code into corresponding assembly language code The compiler create it by using option **`-S`** from the command line. The file created in this way has extension `as`.