13205 C Programming Lecture 2 - 16/Sept/2019

Fundamentals of Programming Languages

 Scribe: A. Gomes

This lecture aims to respond to the following questions:

- Does a programming language build upon an alphabet as, for example, the Portuguese language?
- Does a programming language follow the rules of a specific grammar?
- Does a programming language own a dictionary?

More specifically, the main goals of this lecture are:

- To introduce the concepts of alphabet, grammar, and dictionary in programming languages.
- To show how to construct a statement, which is the principal building block of a program.

Dec.	Symbol	Description	Dec.	Symbol	Dec.	Symbol	Dec.	Symbol
0	NUL	Null char	32		64	@	96	‘
1	SOH	Start of Heading	33	!	65	A	97	a
2	STX	Start of Text	34	”	66	B	98	b
3	ETX	End of Text	35	#	67	C	99	c
4	EOT	End of Transmission	36	$	68	D	100	d
5	ENQ	Enquiry	37	%	69	E	101	e
6	ACK	Acknowledgment	38	&	70	F	102	f
7	BEL	Bell	39	’	71	G	103	g
8	BS	Back Space	40	(	72	H	104	h
9	HT	Horizontal Tab	41	)	73	I	105	i
10	LF	Line Feed	42	*	74	J	106	j
11	VT	Vertical Tab	43	+	75	K	107	k
12	FF	Form Feed	44	,	76	L	108	l
13	CR	Carriage Return	45	-	77	M	109	m
14	SO	Shift Out / X-On	46	.	78	N	110	n
15	SI	Shift In / X-Off	47	/	79	O	111	o
16	DLE	Data Line Escape	48	0	80	P	112	p
17	DC1	Device Control 1 (oft. XON)	49	1	81	Q	113	q
18	DC2	Device Control 2	50	2	82	R	114	r
19	DC3	Device Control 3 (oft. XOFF)	51	3	83	S	115	s
20	DC4	Device Control 4	52	4	84	T	116	t
21	NAK	Negative Acknowledgement	53	5	85	U	117	u
22	SYN	Synchronous Idle	54	6	86	V	118	v
23	ETB	End of Transmit Block	55	7	87	W	119	w
24	CAN	Cancel	56	8	88	X	120	x
25	EM	End of Medium	57	9	89	Y	121	y
26	SUB	Substitute	58	:	90	Z	122	z
27	ESC	Escape	59	;	91	[	123	{
28	FS	File Separator	60	<	92	\	124	—
29	GS	Group Separator	61	=	93	]	125	}
30	RS	Record Separator	62	>	94	^	126	∼
31	US	Unit Separator	63	?	95	_	127	DEL

Table 1: ASCII code or alphabet. “Dec.” represents the decimal index of each symbol.

1 ASCII Alphabet

When, we write up a document (e.g., a report, a scientific article, or a book) in English, we use the Latin alphabet (A to Z) and English-specific grammar. Similarly, when we write up a program, we use the ASCII code as alphabet and a programming language-specific grammar. ASCII code is the alphabet for all high-level programming languages like, for example, C, Java, Python, and so on. ASCII code is shown in Table 1. A brief glance at Table 1 shows us that ASCII is a superset of the Latin alphabet.

Indeed, ASCII includes not only the Latin alphabet, but also other printable characters like digits 0 to 9, punctuation marks, and also a few miscellaneous symbols, that is, almost every character on a keyboard. The ASCII also includes unprintable symbols like the symbol 127 that concerns the command DEL (delete), as well as the first 32 characters in the ASCII code, which are control codes used to control peripherals, including printers. The ASCII code was extended too include more symbols, particularly those punctuated Latin symbols like "ó" and "ç" found in some European languages as the Portuguese. Such an extended alphabet is called *extended* ASCII code, which includes 256 symbols rather than 128 of the ASCII code.

Recall that the reason why we need to compile a C program program is that the machine does not "understand" the ASCII alphabet of C and other high-level programming languages. The only alphabet that a machine is able to interpret is the binary alphabet $\{0, 1\}$. Consequently, every single program in C must be compiled to generate its binary code. This binary code is also known as machine code, the one we need to run a program.

2 Tokens

In English, we use alphabet symbols to form words, and words to form sentences. In C programming, we use ASCII symbols to form tokens, and tokens to form statements. Basically, a token is the smallest independent unit or entity in a program. The C preprocessor and compiler sees a program as a sequence of tokens. The rules that dictate the formation of tokens and statements constitute what we call the grammar. As for natural languages as English, Portuguese, and French, the grammar differs from one programming language to another.

Tokens represent unit entities. Such entities may possess a name (or identifier) and a value. Entities divide into the following categories: names (or identifiers), operators, literals, keywords, and punctuators.

2.1 Names

We identify constants, variables, and functions through their names or identifiers. Let us see how such names are syntactically formed.

Constants

The concept of constant resembles the concept of constant in mathematics. A *constant* is a named entity whose value does not change over time. That is, a constant has a name and a value. The name of a constant is an identifier (or word) that obeys the following syntax rules:

- It is case-sensitive.
  For example, the names `Maya` and `maya` are distinct because `M` is not the same as `m`.
- Its length is 31 maximum.
  For example, `temperature` is a valid name, but not `temperaturetemperaturetemperature` because the latter name has more than 31 characters.
- Its first character is either a letter or underscore symbol.
  For example, valid names are `x` and `_x`, but not `1_x` because the latter name starts with the digit `1`.

- Any subsequent character of the name is either a letter or underscore symbol, but it can be also a digit.

Figure 1 shows the syntax diagram concerning the third and fourth syntax rules in the formation of names (or identifiers) for constants. Taking into consideration that we are discussing names or identifiers, the same rules applies to names of variables and functions.

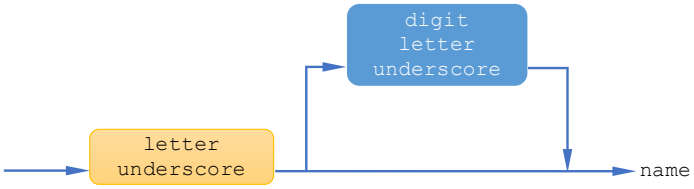

Figure 1: Syntax diagram for names (or identifiers) of constants, variables, and functions.

Variables

The concept of variable also resembles the concept of variable in mathematics. A *variable* is a named entity whose value may change over time. Similarly to a constant, a variable has a name and a value. The name of a variable is an identifier that shares the same syntax rules as the name of a constant, as illustrated in Figure 1.

Functions

As in mathematics, the concept of function may possess several input variables, and may produce several output values. For example, assuming that the two input variables x and y take on the values 1.0 and 2.0, the function $f(x, y) = x^2 + y^1 - 9$ produces the single value -4. Another example is the function $g(u, v) = (u^2 - v, v^2 + u)$ which has two input variables u and v, and two output values produced by $u^2 - v$ and $v^2 + v$.

Thus, a *function* is a named entity with multiple input values and multiple output values, where the output multi-value depends on the input multi-value. Therefore, a function has a name, an input multi-value, and an output multi-value. The name of a function is an identifier that shares the same syntax rules as the constant and variable names, as shown in Figure 1.

2.2 Operators

In C, operators work as pre-defined functions. They belong to the dictionary (or lexicon) of the C language. Furthermore, it is not possible to create new operators in C. Consequently, there is no syntax rule to form new operators. The most common operators in C are the following: assignment, arithmetic, logic, relational, incrementation, and decrementation.

Assigment Operator

This operator is recognized by the token `=`. This operator allows us to assign a value to a variable (in memory). For example, the expression `x=5.0` allows us to assign the real or floating-point value `5.0` to the variable `x` in memory. The good thing about the assignment operator is that it allows us to change the value of a variable in memory. In other words, it is a write operation in memory.

Arithmetic Operators

The most common arithmetic operators in C are the following:

- `+` : addition of two numbers
- `-` : subtraction of two numbers
- `*` : product of two numbers
- `/` : division of two numbers
- `%` : remainder of division of two integer numbers

Arithmetic operators are binary because they require two operands like in mathematics. For example, given the operands 5.87 and 3.21, the expressions `5.87 + 3.21`, `5.87 - 3.21`, `5.87 * 3.21`, and `5.87 / 3.21` are valid. But, the expression `5.87 % 3.21` is not valid because the operands are not integer numbers.

Logic Operators

In C, logic operators have the following names: `&&`, `||`, `!`. The operator `&&` represents the logic conjuntion (AND), the operator `||` represents the logic disjunction (OR), while the operator `!` represents the logic negation (NOT). In Table 2, we see the truth tables for logic operators. For example, given the values `T` (true) and `F` (false) for the variables `x` and `y`, respectively, the resulting values for the expressions `x && y`, `x || y`, and `!x` are `F`, `T`, and `F`, respectively.

x	y	$x \&\& y$
T	T	T
T	F	F
F	T	F
F	F	F

x	y	$x \|\| y$
T	T	T
T	F	T
F	T	T
F	F	F

x	$!x$
T	F
F	T

Table 2: Truth tables for logic operators.

Relational Operators

In C, there are various relational operators, namely:

- `<` : less than
- `<=` : less or equal to
- `==` : equal to
- `!=` : not equal to
- `>` : greater than
- `>=` : greater or equal to

These operators are binary. For example, assuming that the operands `x` and `y` take on the values `1.2` and `3.4`, the expressions `x<y`, `x<=y`, `x==y`, `x!=y`, `x>y`, and `x>=y` produce the values `T`, `T`, `F`, `T`, `F`, and `F`, respectively. Thus, relational expressions produce logical values (either true or false) as the logical expressions above.

Incremental and Decremental Operators

There is only a single incremental operator (`++`) and a single decremental operator (`--`). These operators are unary, that is, they work with a single operand. For example, assuming that the integer variable `x` takes on the value `23`, the expression `x++` increments the value of `x` to `24`.

2.3 Literals

A literal is an unnamed entity that represents a constant. Let us see some examples:

```
5677    : an integer literal
3.78    : a real or floating-point literal
g       : a character literal
story   : a string literal
```

Let us now see how an integer literal is formed. Looking at Figure 2, we see that C supports integer literals using three distinct number systems: octal, decimal, and hexadecimal.

The base of the octal number system consists of 8 digits: $\{0, 1, 2, 3, 4, 5, 6, 7\}$. A literal is treated as octal when it starts with '`0`', followed by any combination of digits of the octal base (see top branch in Figure 2). For example the literal `05677` is octal, but `05679` is not because the digit `9` does belong to the octal base.

The base of the decimal number system consists of 10 digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. A literal is treated as decimal when it starts with any digit 1 through 9, followed by any combination of digits of the decimal base (see middle branch in Figure 2). For example the literal `5677` is decimal, but not `05677` because it starts with `0`.

The base of the hexadecimal number system consists of 16 digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$. A literal is treated as hexadecimal when it starts with '`0`', followed by '`x`' or '`X`', followed by any combination of digits of the hexadecimal base (see bottom branch in Figure 2). For example the literal `0x5677` is hexadecimal, but not `05677` because it misses `x` after '`0`'.

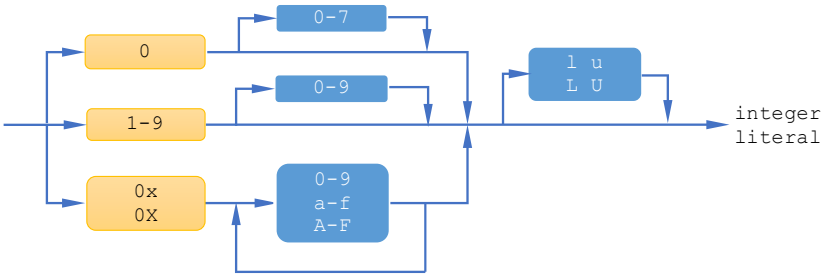

Figure 2: Syntax diagram for integer literals.

The suffixes '`l`' and '`L`' may be used to specify *long* integer literals, while '`u`' and '`U`' are for *unsigned* integer literals. For example the literal `345L` is a long integer literal. We will come to this discussion further ahead in the context of data types in C.

2.4 Keywords

In C, there are several keywords. They belong to the lexicon or dictionary of the C language. A keyword is a reserved word that can be only used in specific contexts. For example, we use the keyword `int` to define an integer constant or integer variable. Specifically, the expression `int x` states that `x` is an integer variable, while `const int y` states that `y` is an integer constant. Indeed, `const` is another keyword. Note that keywords cannot be used as constants or variables or any other names.

2.5 Punctuators

A punctuator is a symbol or character that separates tokens, such as, the comma `,` the semi-colon `;` brackets `{ }`, parentheses `( )`, etc. A punctuator does specify an operation that generates a value. However, in some contexts, punctuators may also work as an operator. The multi-functional nature of tokens in C is

`auto`	`else`	`long`	`switch`
`break`	`enum`	`register`	`typedef`
`case`	`extern`	`return`	`union`
`char`	`float`	`short`	`unsigned`
`const`	`for`	`signed`	`void`
`continue`	`goto`	`sizeof`	`volatile`
`default`	`if`	`static`	`while`
`do`	`int`	`struct`	`double`

Table 3: Keywords in C language.

commonplace. For example, the token `&` allows us to get the address of a variable in memory, but it also works as the bitwise conjunction operator.

3 Terms and Expressions

A *term* is one of the following entities: name (or identifier), literal, and parenthesized expression (see Figure 3). The first two entities are tokens, while the third is a sequence of tokens. A term is also known as a primary expression.

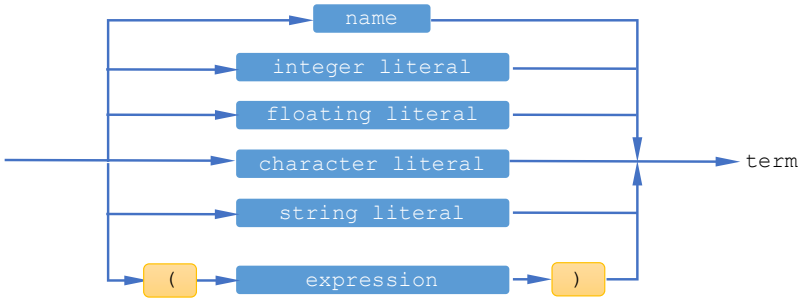

Figure 3: Syntax diagram for terms.

In turn, an *expression* is a sequence of one or more terms separated by infix operators (see Figure 4). Note that an infix operator is an operator between operands (i.e., terms in this case). For example, $t_0 \oplus t_1$ is an expression with 2 terms (t_0 and t_1) and 1 infix operator ($\oplus$) in the middle.

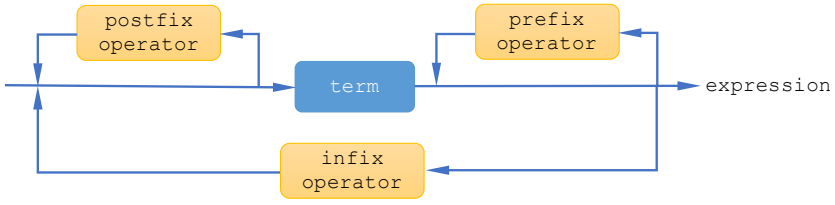

Figure 4: Syntax diagram for expressions.

Note that a term may be preceeded by a prefix operator, and followed by a postfix operator. A prefix operator comes before a term; for example, in the expression `++x + 5`, the prefix operator `++` increments the value of `x` before using the term `x`, which is followed by the infix operator `+` and the term `5`. Analogously, in

the expression `5 + x++`, the postfix operator `++` increments the value of `x` after using the term `x` in the sum with the term `5`.

Let us see some examples of expressions:

```
5                 : a single integer literal
x                 : a single variable name
5 + x             : two terms separated by the infix operator +
x * y - z         : three terms separated by two infix operator (* and -)
(x * y) * (x-z)   : two parenthesized expressions separated by the infix operator *
!x                : a single variable name preceeded by the prefix operator !
y++               : a single variable name followed by the postfix operator ++
```

It worthy noting that *an expression always evaluates to a value.*

4 Statements

A statement is an expression followed by a semi-colon, as illustrated in Figure 5. Statements are the building blocks of a program. Indeed, a program is a sequence of statements.

Figure 5: Syntax diagram for statements.

Let us see a few examples of statements:

```
++x;            : an expression with a prefix operator followed by a variable name and a semi-colon
y = 2 * x +1;   : an expression with the single variable name y followed by the assigment operator,
                : another expression, and a semi-colon
int x;          : an expression with the keyword int followed by a variable name and a semi-colon
```

4.1 Declarations

A declaration is a particular statement. Indeed, a declaration serves the purpose of allocating memory for variables and constants. Usually, declarations come in the beginning of a program because we need to reserve memory for variables and constants before using them. Indeed, *a constant or a variable cannot be used in a program without declaring each before.* This principle applies to any program in C.

Figure 6: Syntax diagram concerning the declaration of a variable.

Declaration of a Variable

The declaration of a variable obeys to the syntax rule depicted in Figure 6. It shows that we must specify the data type in front of the variable name. For example, the declaration `int x;` tells us that the variable

`x` only holds an integer value at a time, where `int` specifies the integer data type of `x`. Furthermore, the variable `x` is not allowed to store a value that is not integer.

Let us now see a C code snippet, where we find the declaration of a single variable `x`:

```
#include <stdio.h>

int main()
{
  int x;

  x = 12;
  (...)
}
```

The declaration `int x;` allocates memory for the variable `x`. After its memory allocation, `x` resides in memory, but it is still empty because no value has been assigned to it. Then, the assignment statement `x = 12;` copies the integer value 12 into `x`. Thus, the assignment statement is a way of changing the value of a variable.

Declaration of a Constant

The declaration of a constant is similar to the declaration of a variable, as shown in Figure 7. For example, in the C program below, `x` is an integer constant whose value is `5`. In syntactic terms, the declaration of a constant requires the qualifier `const` before its type. Besides, its value must be defined in the declaration.

Figure 7: Syntax diagram concerning the declaration of a constant.

```
#include <stdio.h>

int main()
{
  const int x = 5;
  float y;

 (...)

}
```

Thus, declaring a constant triggers two actions: (i) allocation of memory for the constant; (ii) copying its value into its location in memory. In contrast, the declaration `float y;` of the variable `y` only causes the allocation of memory for it.

4.2 Assignment Statement

This statement assigns a value to a variable in memory. Such a value is the result of evaluating an expression, as shown in Figure 8.

As said above, assigning a value to a variable is a write operation in memory. That is, one writes or copies a value to the memory location of a variable. For example, in the assignment statement `y=x+2.1;`,

Figure 8: Syntax diagram of the assignment statement.

the expression is `x+2.1` and evaluates to 7.1 because the constant `x` has the value 5 (cf. program below). The value 7.1 is then assigned to the variable `y` in memory.

```
#include <stdio.h>

int main()
{
  const int x = 5;
  float y;

  y=x+2.1;
  printf("O valor de y=%f\n",y);
}
```

4.3 Block Statement

A block is a statement that contains a sequence of statements (see Figure 9). For example, the `main` program shown just above is a block that starts with { and ends with }. Such a block contains five statements, the first two of which are declarations. Note that the semi-colon ; indicates the end of each statement.

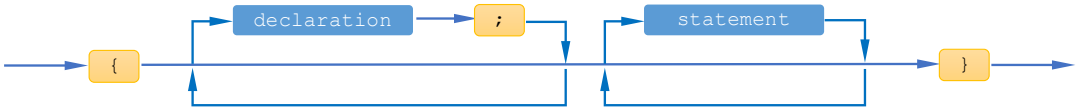

Figure 9: Syntax diagram of the block statement.

The declarations of a program allows us to build the data store or repository of a program in memory. In the program below, we have two declarations concerning two variables, `x` and `y`, so the data store consists of `x` and `y`. The four statements of the program concern operators and functions that handle such data.

```
#include <stdio.h>
int main()
{
   int x;
   float y;

   printf(Type in an integer number: );
   scanf(%d,&x);
   y=x+5.4;
   printf(The value of y=%f\n,y);
}
```

This centralized model of data handled by functions (including operators) is illustrated in Figure 10. Here, we see that we have two output functions `printf`, with the first `printf` writing the string "`Type in`

`an integer number:  ` ” on screen, while the second `printf` outputs the string “`The value of y=10.5`” if we assume that the value `5` was read in the variable x via the input function `scanf` by typing `5` in keyboard.

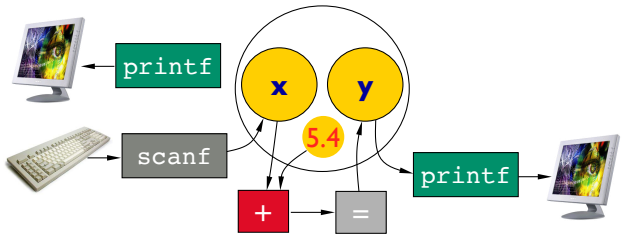

Figure 10: Syntax diagram of the block statement.