# Compilation Techniques

## Laboratory 1

**Introduction**

The formal languages and translation techniques of data from an input format to an output one are a fundamental domain in the information technology. The computers function using a binary language in which everything is described only in terms of 1 and 0, true or false. In the same time the microprocessor instructions (the machine code) are very simple. Generally speaking these are comprised only from memory transfers, basic arithmetic and logic operations, conditioned and unconditioned execution transfer, etc. To program a computer using only these elements is an extremely difficult and time consuming task and it is used only in some specific situations. It is much easier to describe the required algorithm in a form which is closer to the problem to be solved (the description in a high level language) and the computer will convert this description in its specific instructions (the machine code representation).

The knowledge of the compilation techniques brings a series of very important benefits, among which we enumerate:

- The programmer understand better how the computer executes the instructions from the program written by him, which allows him to optimize better the source code. Even using the actual compilers there are great differences of speed and consumed resources between codes written in different ways. In domains in which this aspect is very important (microcontrollers, high performance computing, simulation, games, etc) every optimization which can be used counts and some of these optimizations can be truly understood only if it is known the way in which the source code is translated in machine code and the way in which this code is executed.
- Most of the applications require input data which is described in a certain format. This can be taken from a file, from keyboard, from network, etc. The format of this data can be standard (such as XML, JSON) or can be application specific. If the format is standard the programmer has different libraries which help him to read and write data in that format. If the format is application specific, the programmer needs to implement the mechanisms which allow him to read data, to validate its structure and to save it. These methods of input data reading and validation are in fact the subject of the translation techniques which, if the source format is more complex, become very important.
- Usually the programmers need to know many programming languages. The knowledge of the formal, fundamental aspects of the programming languages makes much easier the learning of new languages. In the same way in which in the human languages exist some basic aspects such as verb, noun or adjective which are found in any spoken language, also in the programming languages there are some fundamental constructions which once known make the learning process of a new language much easier.
- The formal description of a language has vast implications and indeed we can say that the fundamental difference between a language used by people in the daily life and a programming language is mostly the fact that the latter has defined for each construction a semantic (meaning) very clear and which is not ambiguous. In the spoken languages there are words with multiple senses and in the same time a grammatical construction may mean different things. In the programming languages each instruction is completely defined in all possible contexts, which makes the meaning of that instruction unique. The habit to use formal descriptions brings a better clarity in thinking and in phrasing and usually creates the habit to understand better much of the aspects implied in the acts of describing and communication.

In the CT laboratory we will implement a simple compiler for a subset of the C programming language, named AtomC. For this we will cover the essential steps in writing a compiler. These steps are:

- **Lexical analysis** – it takes individual characters from the source code and assembles them in lexical atoms (tokens) such as numbers, identifiers, operators, etc. A lexical atom is the primary component, indivisible which will be used subsequently. In the same time some components which will not be needed (spaces, comments) are discarded.
- **Syntactic analysis** – it assembles the lexical atoms in the grammatical constructs specific to the language. For example an expression is made from operands, operators and parenthesis arranged according to certain rules.
- **Domain analysis** – it associate each encountered identifier to its definition. For example in the C language a variable or a function must be first declared only after that it can be used.
- **Type analysis** – it determine for each construction which are the used data types (inputs) and the output ones. For example in C the addition between an integer and a real number will have as result a real number.
- **Virtual machine** – it represents the output language generated by the compiler. The compilers can translate the source code in a language specific to a certain microprocessor (machine code) or to a virtual language. Java and C# also use virtual machines.
- **Code generation** – It is the final step in which the source code is translated in destination instructions. In the same time some optimizations can be performed such as constant expressions evaluation (for example the code necessary for the expression "1+2" is substituted with the code which uses the number "3").

The final result of the laboratory will be a compiler for the language AtomC which will be capable to compile programs written in this language and to execute them.

**Regular expressions**

The regular expressions are a way to describe some simple lexical constructions. They are used often in the situations in which we need to process different texts to extract from them the required information. Practically speaking all common programming languages have either direct support for regular expressions or they offer libraries to use them. There are many variations regarding the syntax of the regular expressions and their facilities, but we will be used a more restricted set, which is found in most of these formats.

A regular expression is compared with the input text and it results true or false if that text is conform or not with the regular expression. The comparison can be made once for all the text or it can search all the positions in text which contains character sequences conform to the given expression. In the same time the conform expressions can be returned for further processing (ex: all the e-mail addresses can be extracted from a text) or the text can be split in subtexts using as separators the conform sequences (ex: a text composed of many lines of numbers separated through commas can be split first by lines and after that by commas to result the numbers from each line).

A regular expression is mostly composed from:

- **individual characters** – these characters are directly tested with the ones from the input text. If some characters have predefined significations, they will be prefixed with the operator backslash (****), in the same way as in the strings in C.
  Ex: ***a*** – it will match only the character „a" from the source text
- **character classes** – they begin and end with brackets (**[]**) and they contains all the possible characters for the input character. If it is needed to test a range of characters, the bounding characters are written with minus (**-)** between them. There can be many ranges and individual characters. If a character class begins with hat (**^**), this has a negation role and all the characters will be considered except the given ones.

Ex: ***[abc]*** – any of the characters a,b or c (but only one)
***[0-9]*** – any numerical character
***[^()]*** – any character except parenthesis
***[a-zA-Z]*** – any letter, uppercase or lowercase

- **sequence, $e_1e_2$** – the consecutive regular expressions $e_1$ and $e_2$ (without any space between them) will test in the source text if there exists consecutive and without any other intermediary characters the corresponding subtexts. It has the signification of „**and"** (logical conjunction).
  Ex: ***air*** – tests the existence of the word "air" (three consecutive characters)
  ***a[a-z]*** – any word composed of two lowercase letters which begins with the letter „a": „ac", „aa", „am", …
- **alternative, $e_1|e_2$** – the operator **or** (**|**) tests if any of the given expressions matches (disjunction).
  Ex: ***air|earth*** – the word „air" or the word „earth"
- **parenthesis, ()** – changes the evaluation order of the operators
  Ex: ***a(i|cto)r*** – the word „air" or the word „actor". If it would not have been the parenthesis (expression **ai|ctor**), because the alternative is considered after sequence (the alternative has lower priority), the expression would have tested the words "ai" or "ctor".
- **dot,** . – any character
  Ex: ***a.r*** – „air", „a0r", „a-r", „aTr"
- **optional, $e_1$?** – the expression $e_1$ is optional (it can be absent). The operator **?** is postfixed (it is placed after expression). Ex: ***ai?r*** – the word „ar" or the word „air". The operator **?** has a higher priority than the sequence and it is considered before it. If we would have written ***(ai)?r***, it would have tested the words „air" or „r".
- **optional with possibility of repetition, $e_{1*}$** – the expression $e_1$ can be repeated any number of times including 0 times (it can be absent). The operator ***** is postfixed. In literature this operator is also named the Kleene star or closure.
  Ex: a[a-z]* – any word which begins with the letter „a" and can optionally have any number of lowercase characters: „a", „am", „air", „accord". The operator * has a greater priority than the sequence and it is considered before it. If we would have written ***(a[a-z])****, it would have tested optional sequences (including of length 0) made by groups of two lowercase letters, first letter of each group being „a": „" (empty sequence), „am", „abac".
- **required with possibility of repetition, $e_1$+** – the expression $e_1$ can be repeated any number of times, but it must occur at least once. The operator **+** is postfixed. The expression **$e_1$+** is equivalent with **$e_1e_1$***.
  Ex: -[0-9]+ – any negative integer number composed of at least one digit. The operator **+** has a greater priority than the sequence and it is considered before it. If we would have written ***(-[0-9])+***, it would have tested optional sequences (at least one sequence), which begin each one with „-„ followed by a digit: „-1", „-0-5", „-2-7-3".

Besides the above components which we will use from now on, a regular expression may also contain other components, such as: "**^**" – the beginning of the input string (or the beginning of a line), "**$**" - the end of the input string (or the end of the line), „**{}**" – repetition for a specified number of times, etc.

**Observations:**

- Any character in a regular expression is significant. For this reason the spaces are used only when these needs to be explicitly tested.

- The **slash** (**/**) character has no special meaning inside a regular expression, but it is used by some programs to mark the beginning/ending of the regular expression. For this reason, if there are errors on its use, it must be escaped with backslash (****) or it can be put inside a character class (**[]**).
- Inside character classes the operators lose their significance and become simple characters: **[a.]** will test the character „a" or the character dot (**.**), not any character.
- The regular expressions make distinction between lowercase and uppercase characters: **air** will test only the word „air", not also „Air" or „AIR".
- If we need to test some characters which have a special significance (operators, parentheses, dot, backslash (****), etc), these must be prefixed by backslash (****).

**Examples:**

- **[a-zA-Z_][a-zA-Z0-9_]*** – an identifier which begins with any letter (lowercase or uppercase) or underscore and after that can optionally continue with any letter, digit or underscore.
- **[0-9]+(\.[0-9]+)?** – a number made from at least one digit optionally followed by a decimal part. If the decimal part exists, it must contain at least one digit: „12", „0.5", „3.14159"
- **[a-zA-Z]+( *[.,?!\-_]? *[a-zA-Z]+)*** – a sequence of words made only from letters. It starts obligatory with a word. The next words are optional and separated one of each other by any number of spaces, optionally followed by one of the „.,?!-_" and again optionally any spaces.

It can be seen that by using regular expressions a varied class of lexical structures can be represented. The structures which cannot be represented with regular expressions are the recursive ones, for example the correct use of parenthesis in an expression.

Because the more complex regular expressions can be quite difficult to understand, it is recommended to test them on different sequences of test. There are many online applications which do this test, for example http://regexpal.com/, or https://regex101.com/. In these applications the regular expression and a test text are entered. In text will be automatically highlighted the sequences which correspond to the regular expression.

***Application 1.1:*** *In a text there are calendar dates in the format „day/month/year", in which the day and the month are made from 1-2 digits and the year is made from 2 or 4 digits. Write a regular expression which finds all these dates.*
*Ex: John was born on 21/5/1990 and Mary on 7/12/92.*

***Application 1.2:*** *Propose a text which contains a sequence of characters which correspond to the regular expression: [A-Z][a-zA-Z]*,[a-z]+ [0-9]+(/.)*(\.|\?)*

***Application 1.3:*** *Write a regular expression which tests a simplified syntax for the above described character classes: it must begin with „[„ and end with „]"; optionally after „[„ can be „^"; inside can be individual characters or ranges of characters; a character can be simple of prefixed by „\".*