INFO-F-403 – LANGUAGE THEORY AND COMPILING

S-COBOL

Jean-Sébastien LERAT and Gilles GEERAERTS

Academic year 2013–2014

1 Introduction

> *The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.*[1] E. W. Dijkstra

COBOL (**CO**mmon **B**usiness **O**riented **L**anguage) is a a project initiated by the United Stats Department of Defense in order to simplify the implementation of management tools. This outcome of this project was the definition of a rather verbose language which was pretty popular between 1960 and 1980. This is why COBOL can be found, still today, as the software core of many big firms (e.g. banks, insurances). This project offers you the rare opportunity to develop your skills in this vintage language (definitely a key point to mention in your C.V.)

This is the first half of the compiler project. In this part of the project, you will be asked to deal with the syntactical analysis part of the compiler, i.e., write the *scanner* that splits a piece of S-COBOL into a sequence of *tokens*.

2 The S-COBOL language

The S-COBOL (**S**implified-COBOL) is a subset of the original COBOL. A lot of lexical and syntactical units have been removed for the sake of this project. Comments hold on a single line, start with an asterisk ('*') or a slash ('/') and end with a dot followed by a new line ('.\n').

We have already identified the lexical units for you. The exhaustive list is (keywords are witten in `typewriter font`): `identification`, `division`, `program-id`, `author`, `.` (a dot), `\n` (a new line), `date-written`, `environment`, `configuration`, `section`, `source-computer`, `object-computer`, `data`, `working-storage`, `pic`, `value`, `procedure`, `end`, `program`, `stop`, `run`, `move`, `to`, `compute`, `add`, `subtract`, `multiply`, `divide`, `giving`, `,` (comma), `(` (left parenthesis), `)` (right parenthesis), - (minus sign), + (plus sign), = (equals sign), * (asterisk), / (slash), `not`, `true`, `false`, < (lower sign), > (greater sign), <= (lower or equals), >= (greater or equals), `and`, `or`, `if`, `else`, `end-if`, `perform`, `until`, `accept`, `display`, ***identifier***, ***image***, ***integer***, ***real*** and ***string***.

This long list of lexical unit contains special units written in bold, the others are static and they are called the keywords. Here is a short description of these special units:

[1] Sorry... ;-)

identifier must start by an alphabetic character which can be followed by at most fifteen alphanumeric characters or _ or −. For instance `c0mp1ler` is a valid identifier, while `123abc` is not.

image In S-COBOL, images are used to denote numerical types, and are used when declaring variables. The basic numerical type is a single-digit unsigned number (in base 10), and is denoted by `9`, i.e., a variable of type `9` can hold any single-digit number. To get multiple-digit numbers, one uses the notation `9(`n`)`, where n is an integer number giving the maximal length of the number. For instance `9(5)` denotes any number from 0 to 99 999, i.e. a number with maximum 5 digits. As can be seen `9` is nothing but a short hand for `9(1)`. Numbers can be signed: just prefix the type by `s`. For instance, `s9(3)` is a signed number between -999 and 999 (maximum three digits).

Finally, we will also allow decimal numbers. To use this type, one should concatenate an expression of the form `9(`n`)` (to indicate the maximal number of allowed digits *before* the decimal point) with the `v` character (to denote the decimal point), followed by a second expression of the form `9(`n`)` (to indicate the maximal number of allowed digits *after* the decimal point). Again, `9` can freely be used as a shorthand to `9(1)` and `s` can be prefixed to denote a signed number. For instance, `s9v9(3)` denotes the set of signed numbers in the interval $[-9.999, 9.999]$.

integer an integer token is a non-empty sequence of digit(s) and cannot contain meaningless zero(s). Moreover the integer can optionally prefixed with a sign + or −. For instance, `42` and `+42` are valid integers, while `0042` is not.

real a real token is composed of two parts but can also contains only one of them. The first part is an integer and the second part is a dot ('.') followed by a sequence of digit(s). Moreover like integers, reals can optionally prefixed with a sign + or −. For instance, `42.0` and `-42.5` are valid integers, while `01.5` is not. Additionally, reals must contains at least one digit.

string is a sequence of alphanumeric characters or one of the characters following characters: `+`, `-`, `*`, `/`, `:`, `!`, `?`, or the blank character, contained inside simple quotes (`'`). For example: `'hello World'` is a string.

Figure 1 gives an example of a program written in S-COBOL. A program is structured in four **divisions** : *identification* defines the metadata, *environment* defines the compiler parameters, *data* defines the data used by the program (typically the variables) and *procedure* contains the source code of the program. As you can see in the example program, the read instruction is the keyword *accept* and the write instruction is the keyword *display*.

Each of these divisions can be composed of **sections** but we will only focus and use the main sections.

```
identification division.           /Define metadata.
   program-id. Algo-Euclide.
   author. Euclide.
   date-written. 300 BNC.
environment division.              /Define compiler parameters.
   configuration section.
   source-computer. x8086.
   object-computer. LLVM.
data division.                    /Define our variables.
   working-storage section.       /we define 3 variables (a, b, c).
```

```
      77 a s9(5).                 /s for signed.
      77 b s9(5).                 /9 for digit (int).
      77 c s9(5).                 /(5) for 5 digits.
procedure division.               /code of our program.
   main section.
      * Euclide's Algorithm.
      start.                      /The first label is the start point.
         accept a.                /read int from stdin and put it into a.
         accept b.                /read int from stdin and put it into b.
         perform find until b = 0./call find label until b equals 0.
         display 'valeur:'.       /write 'valeur:' on stdout.
         display a.               /write the content of a on stdout.
         stop run.                /stop the program.
      find.                       /create a label called 'find'.
         move c to b.
         perform diff until a < b./call diff label until a is less than b.
         move a to b.             /put the value of a into b.
         move c to a.             /put the value of c into a.
      diff.                       /create a label called 'diff'.
         * Compute a modulo b
         substract b from a.      /a becomes a-b.
```

Figure 1: An example of program written in S-COBOL

3 *Desiderata*

For this project, we ask you to develop a lexical analyzer for S-COBOL . You should:

1. Give regular expressions (RE) for each lexical unit.
2. Give the DFA that recognises *any* token of the S-COBOL language.
3. Implement the automaton as a lexical analyzer.

The lexical analyzer will be implemented in JAVA 1.6[2]. More precisely, you should hand in:

- A PDF report containing all REs, and presenting your work, with all the necessary justifications, choices and hypothesis.
- The deterministic automaton corresponding to the REs
- The source code of your lexical analyzer.
- The S-COBOL example files you have used to test your analyser.
- All required files to understand your work.

You should structure your files in five folders:

doc contains the JAVADOC and the PDF report;

[2]If you want to program in another language, you have to discuss with the teaching assistant

dfa contains the deterministic automaton in the format of the DFA Simulator (`http://home.arcor.de/kai.w1986/dfasimulator/`);

test contains all your example files;

dist contains an executable JAR;

more contains all other files.

Your implementation[3] has to contain a public class (called *LexicalAnalyzer*) which contains a method that returns (your personnal representation of lexical units like an numeric identifier or an object) the next lexical unit read on the input (method called *nextToken()*). It should also contain the executable public class (Main) allowing to use the command:

java -jar yourJarFile.jar Main

in order to run your program in a console and read each line (java.util.Scanner is recommended) on the standard input stream. This process will write on the standard output stream the detected lexical units as soon as possible. For instance, this input:

```
accept a.
perform find until b = 0.
```

will produce this ouput:

```
token: accept     lexical unit: ACCEPT_KEYWORD
token: a          lexical unit: IDENTIFIER
token .\'n'       lexical unit: END_OF_INSTRUCTION
token: perform    lexical unit: PERFORM_KEYWORD
token: find       lexical unit: IDENTIFIER
token: until      lexical unit: UNTIL_KEYWORD
token: b          lexical unit: IDENTIFIER
token: =          lexical unit: EQUALS_SIGN
token: 0          lexical unit: INTEGER
token .\n         lexical unit: END_OF_INSTRUCTION
```

Note that the token is the matched input characters and the lexical unit is your (personnal) short string description of the lexical unit. Moreover, when the standard output will be closed, you have to print your table of symbols (you are free to implement it as you wish).

The printing format of your table of symbols has to:

1. Print on the first line the keyword *variables*.
2. Print all variables in alphabetical order, one variable per line. For each variable, you will print the unique *identifier* followed by its *image* (*identifier* and *image* are separated by space(s)).
3. Print on the next line the keyword *labels*.
4. Print all labels in alphabetical order, one label per line. For each label, you will print the unique *identifier* followed by the *line number* where you encountered this label (*identifier* and *line number* are separated by space(s)).

[3] All non-standard libraries are forbidden.

This is similar to

```
variables
identifier1    image1
.
.
.
identifierN   imageN
labels
identifier1    line no 1
.
.
.
identifierM   line no M
```

You will compress your folder (only zip) and you will submit it on "Université Virtuelle" for **Wednesday, October 30th at 11:59 AM**. A printed copy will be handed in to the Secretariat (Maryka, NO8) for the same date. You are allowed to work in group of maximum two students.

Bon travail !